

# Data Structures and Algorithms in Python

[www.appmillers.com](http://www.appmillers.com)

Elshad Karimov



@karimov\_elshad



in/elshad-karimov

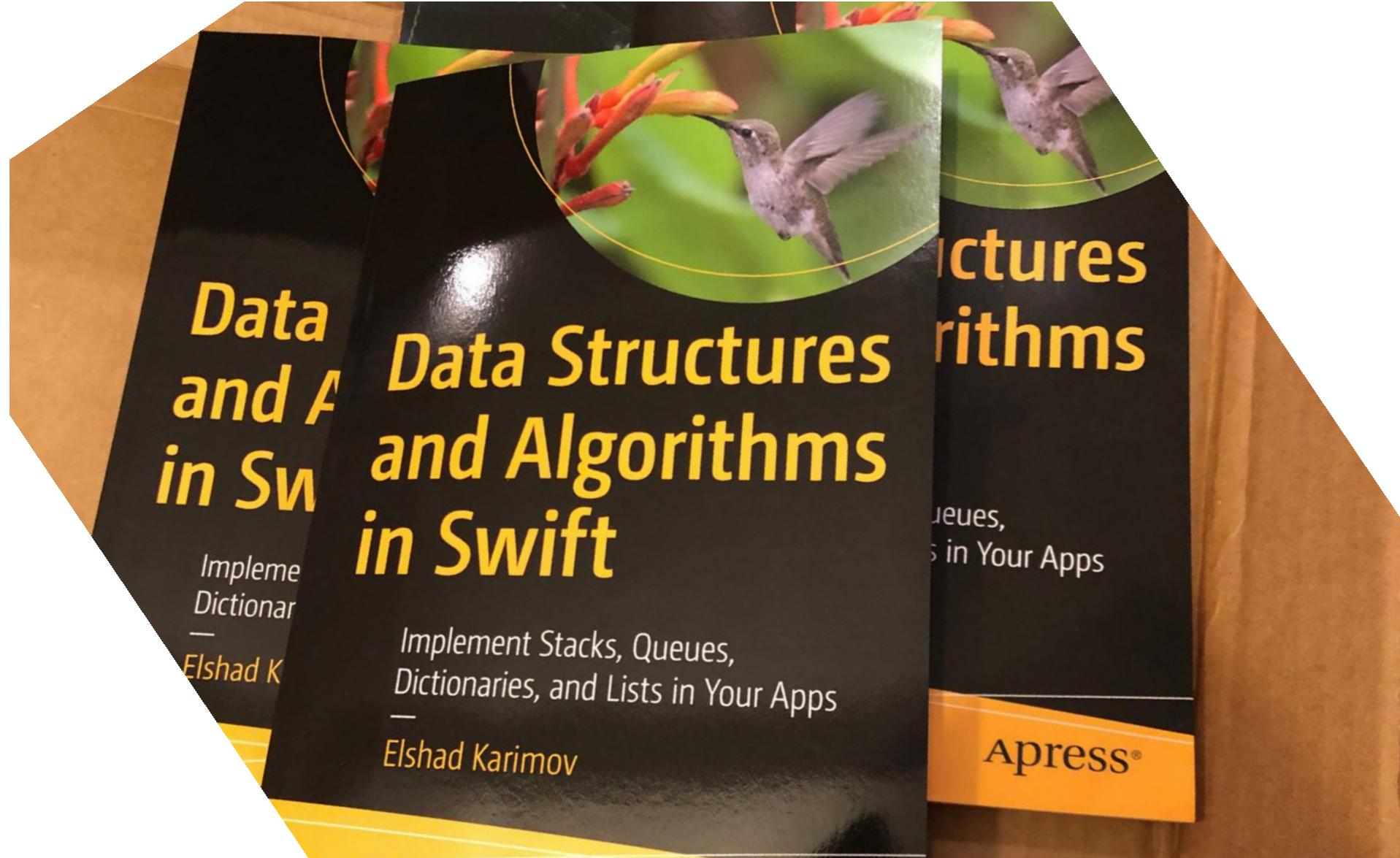


# PYTHON For EVERYONE

- 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

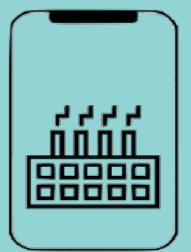


in/elshad-karimov



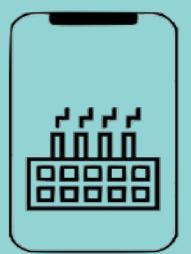
# What are Data Structures?

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



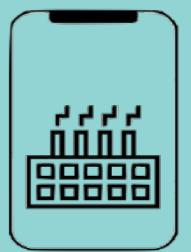
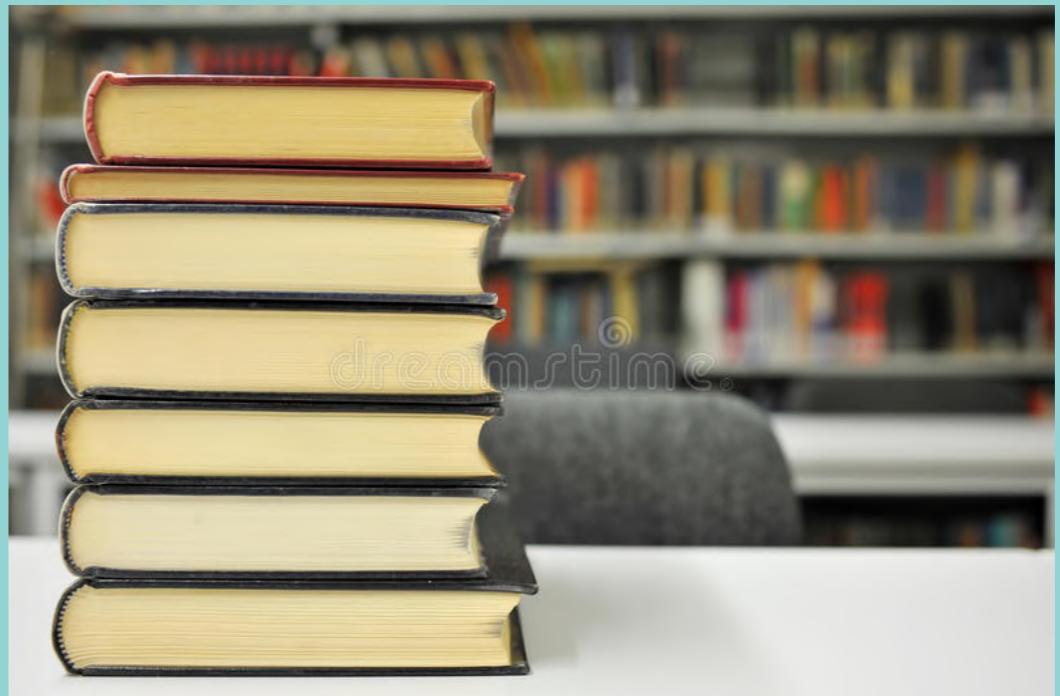
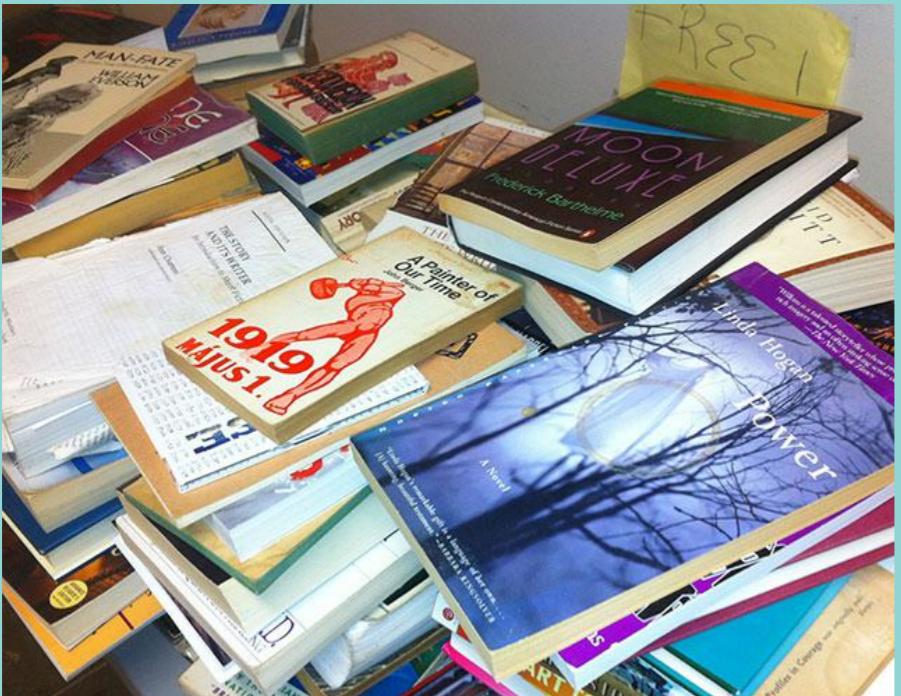
# What are Data Structures?

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



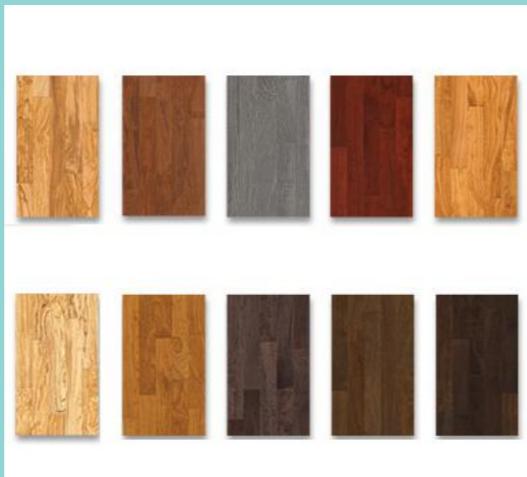
# What are Data Structures?

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



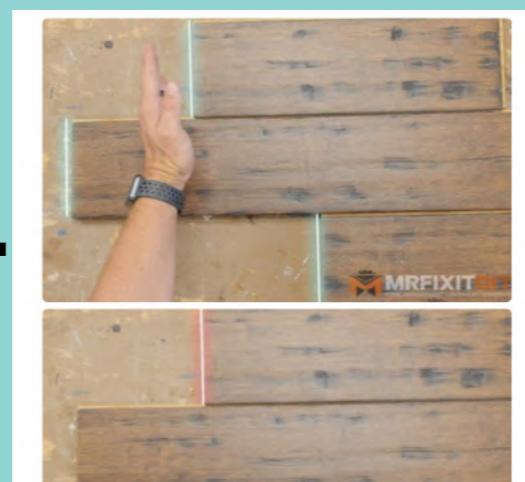
# What is an Algorithm?

- Set of steps to accomplish a task



Step 1: Choosing flooring

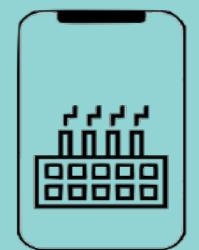
Step 2: Purchase and bring



Step 5: Trim door casing

Step 4: Determine the layout

Step 3: Prepare sub flooring



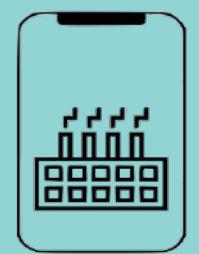
# Algorithms in our daily lives



**Step 1 : Go to bus stop**

**Step 2 : Take a bus**

**Step 3: Go to office**



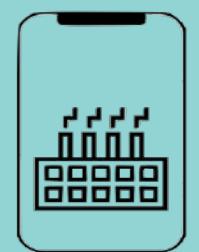
# Algorithms in our daily lives



**Step 1 : Go to Starbucks**

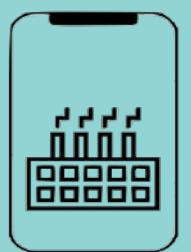
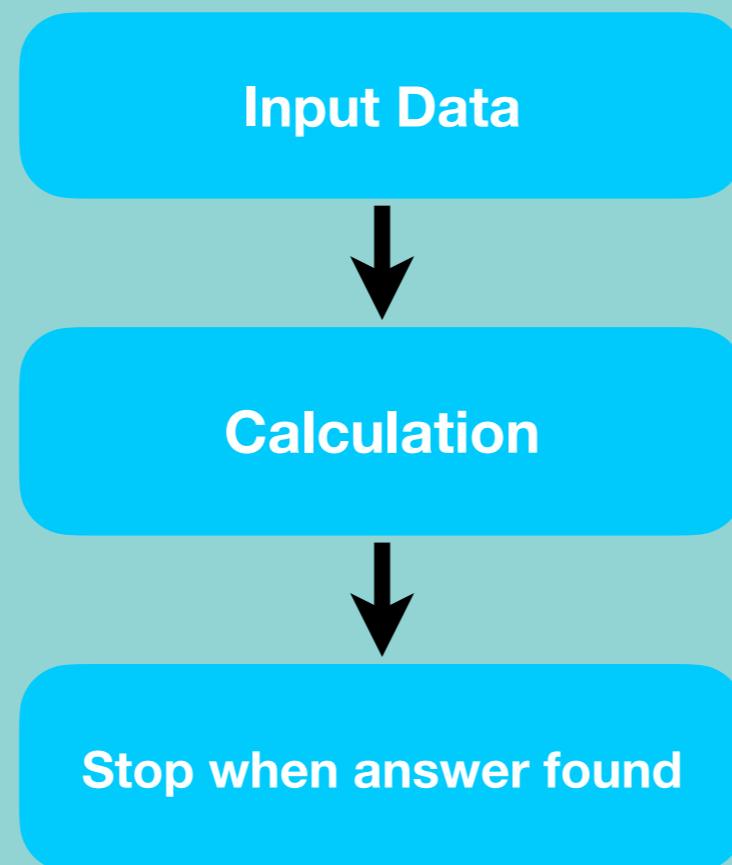
**Step 2 : Pay money**

**Step 3: Take a coffee**



# Algorithms in Computer Science

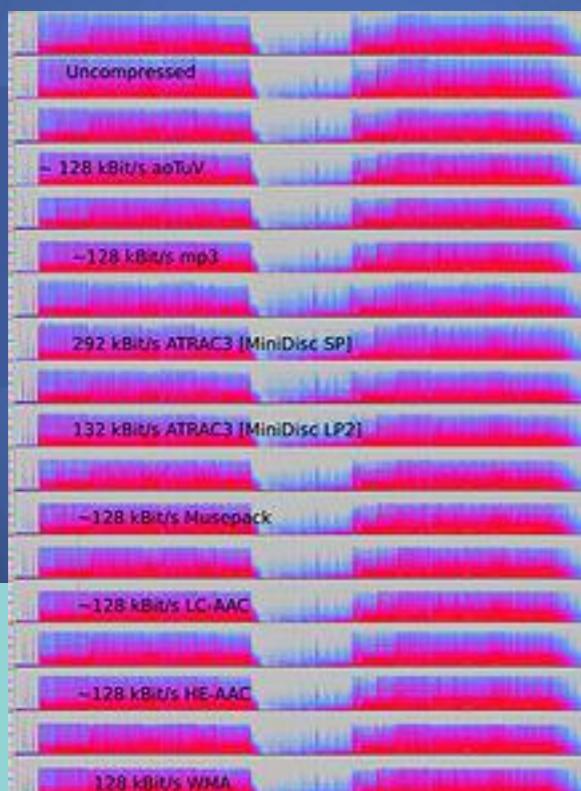
- Set of rules for a computer program to accomplish a task



# Sample algorithms that are used by big companies

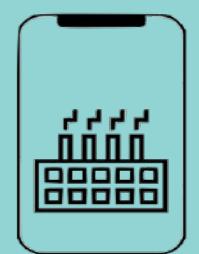
- How do Google and Facebook transmit live video across the internet?

## Compression algorithms



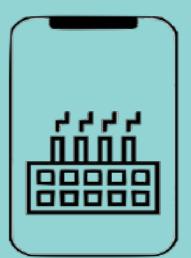
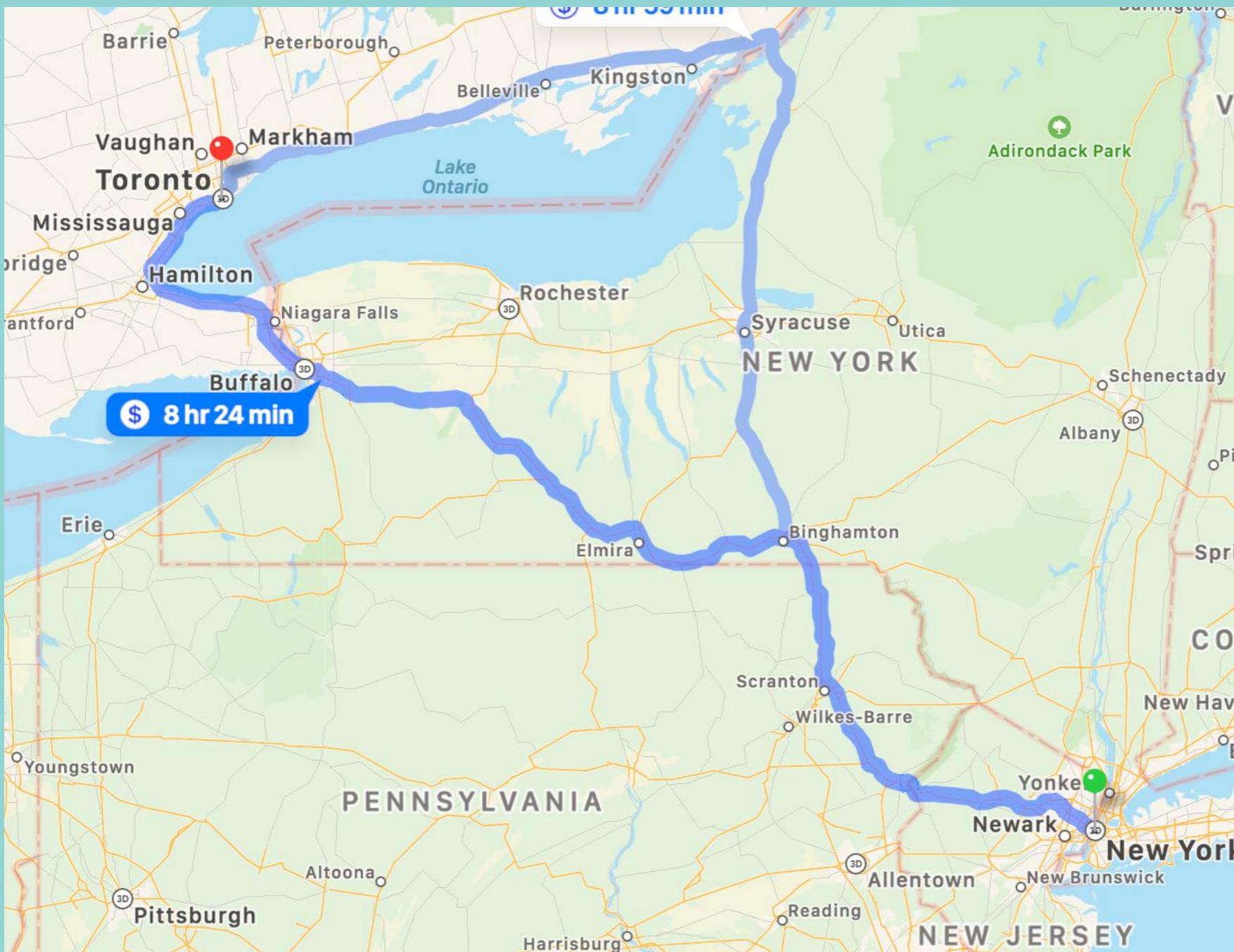
E

COMPRESSION ALGORITHMS			
Input datatype	Today's leading compression algorithm	Previous leading compression algorithm	Compression ratio improvement versus complexity increase
Text	Burrows-Wheeler	Lempel-Ziv-Welch	$1.1/5 = 0.22$
Speech	AMR at 8 kbits/s	GSM-FR at 13 kbits/s	$1.6/2 = 0.8$
Audio	AAC	MP3	$1.5/2 = 0.75$
Photo	JPEG2000	JPEG	$1.2/4 = 0.3$
Video	H.265	H.264	$2/3 = 0.667$



# Sample algorithms that are used by big companies

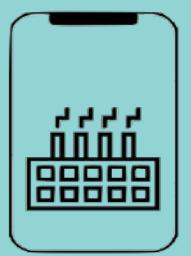
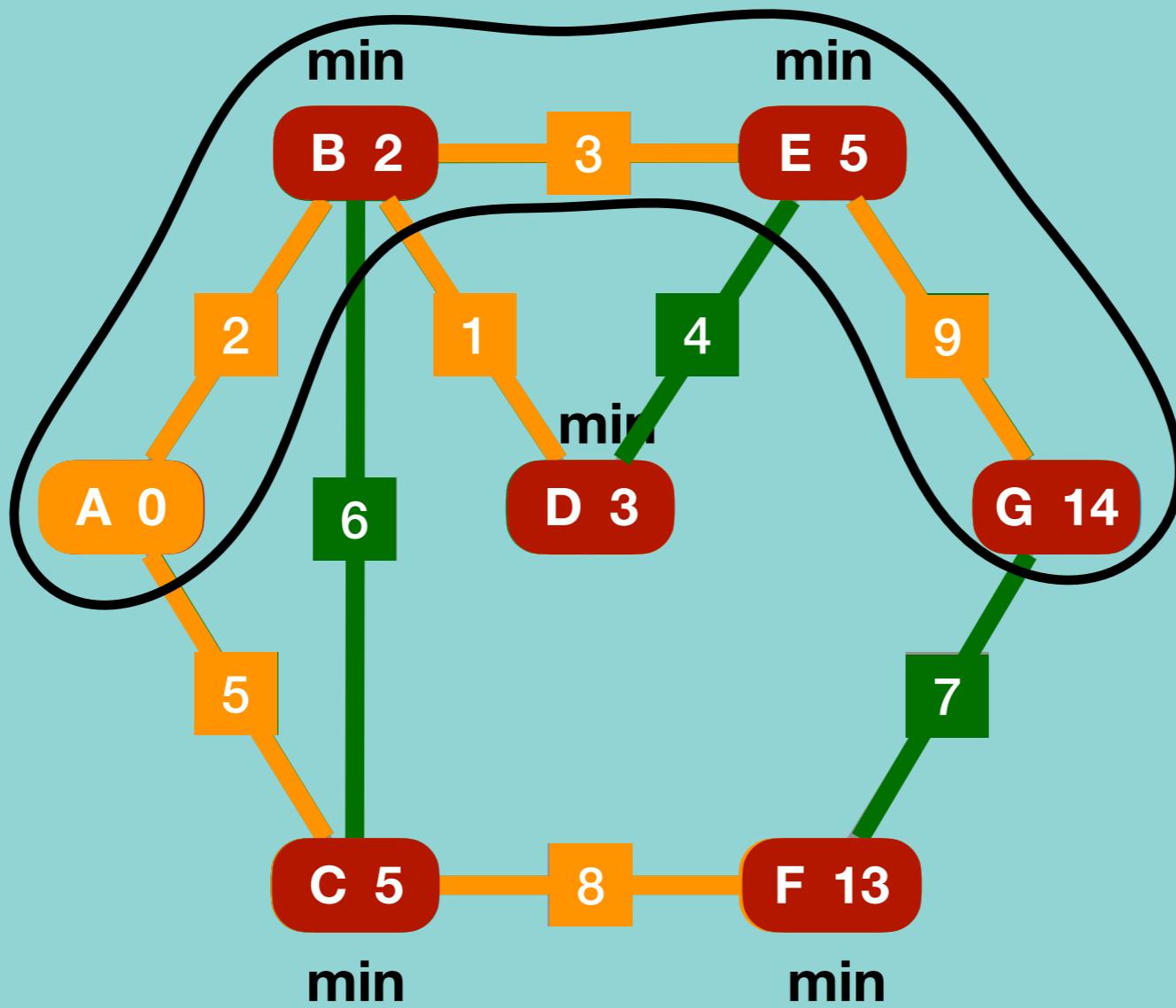
- How to find shortest path on the map?



# Sample algorithms that are used by big companies

- How to find shortest path on the map?

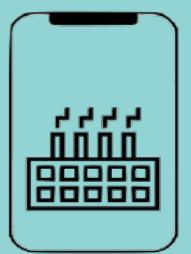
## Dijkstra's algorithm



# Sample algorithms that are used by big companies

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

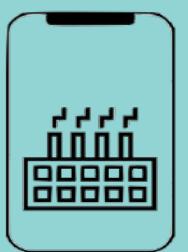
algorithms



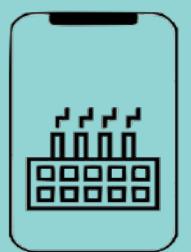
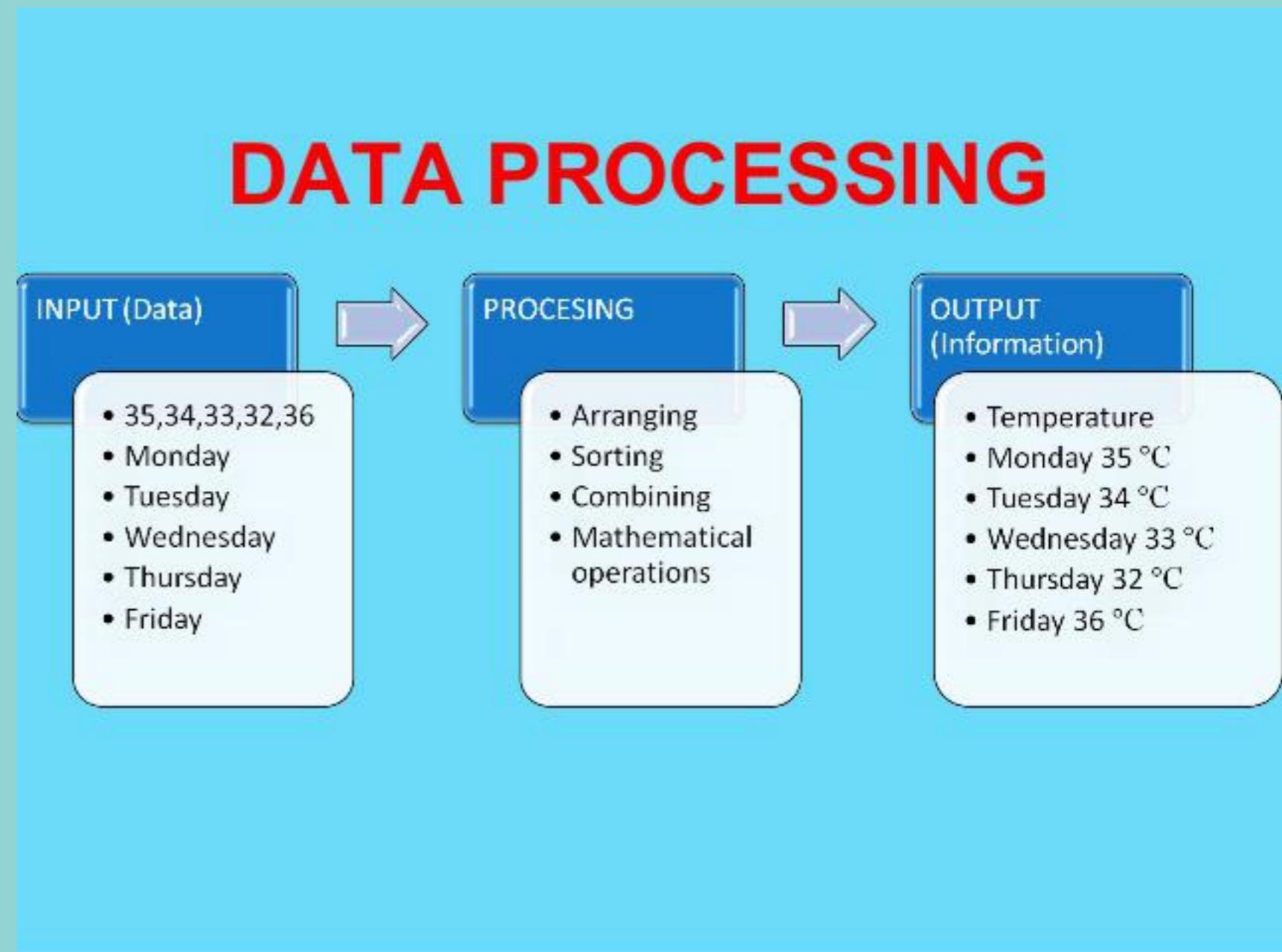
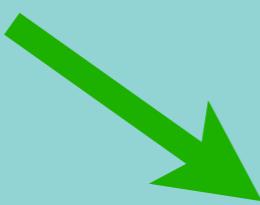
# What makes a good algorithm?

**1. Correctness**

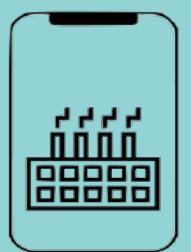
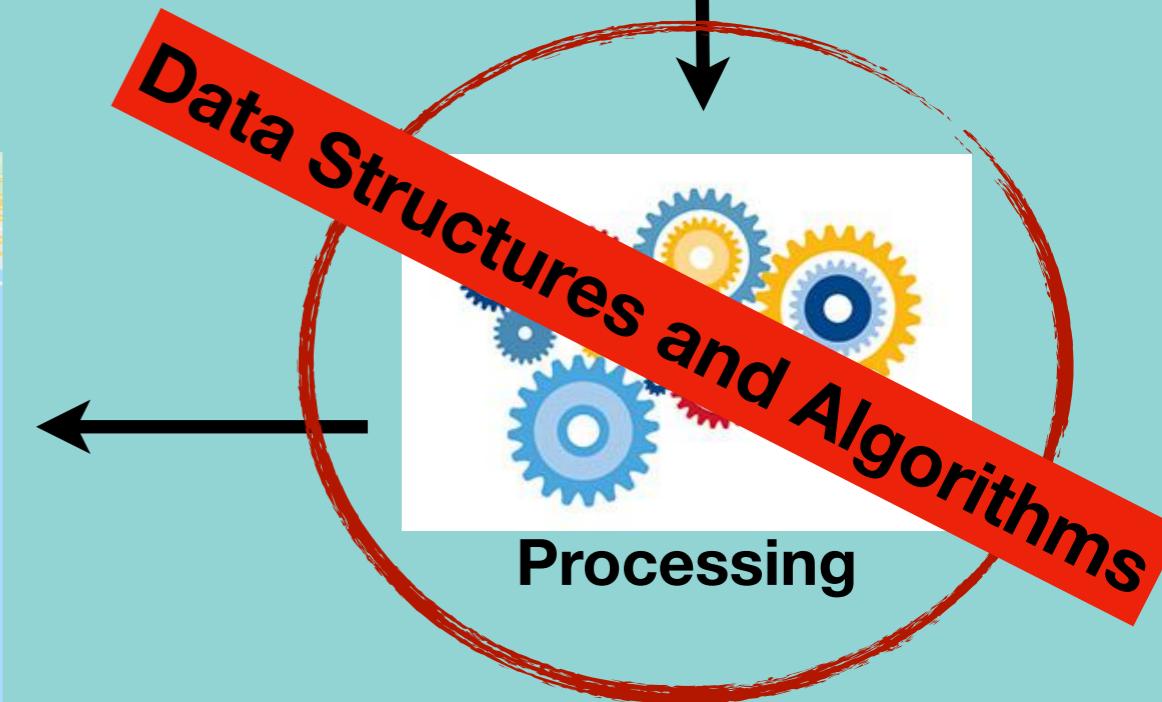
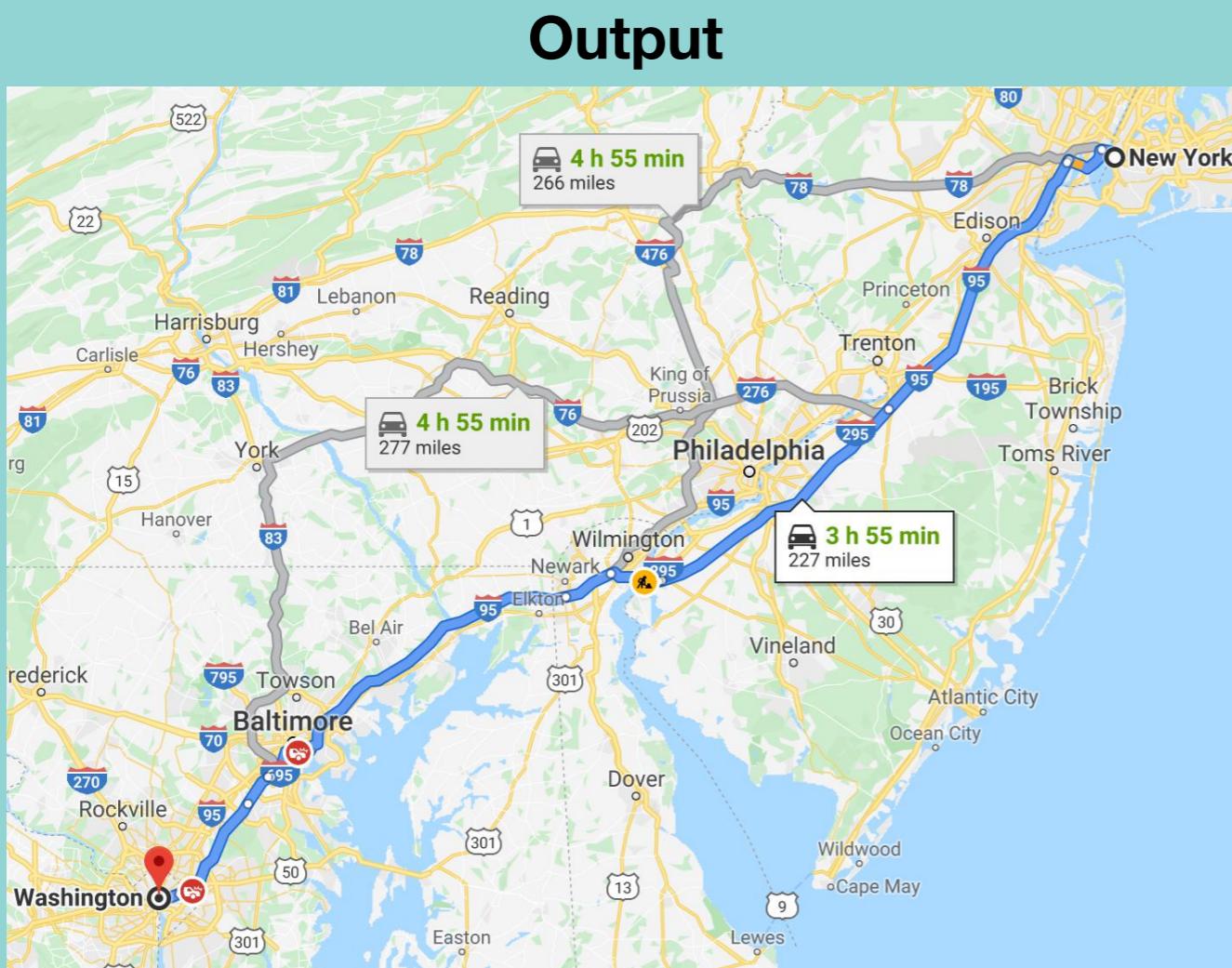
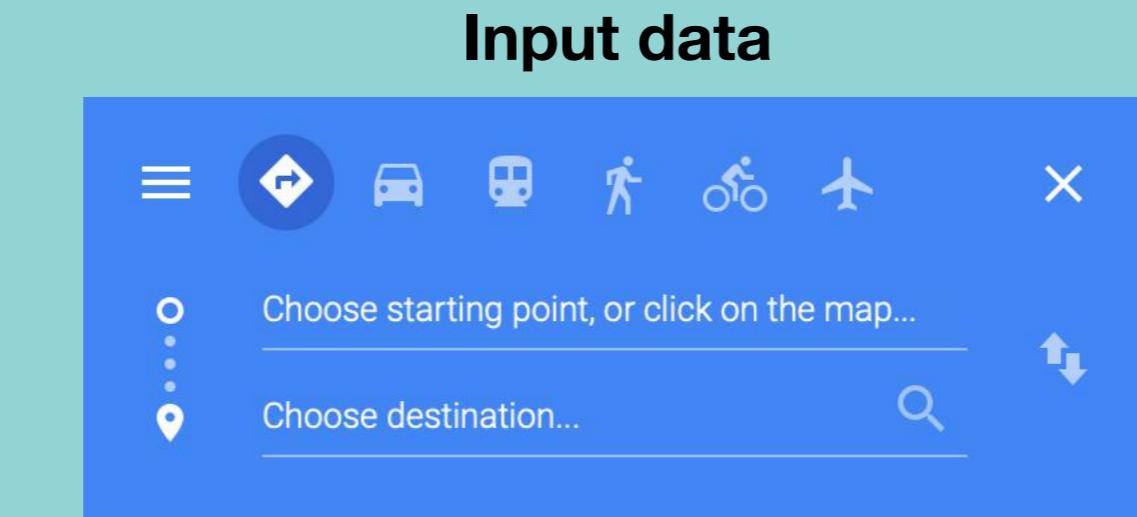
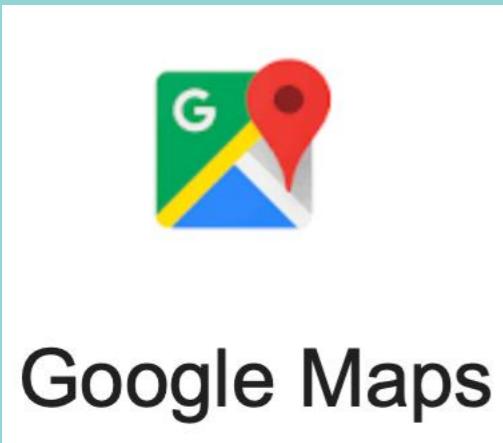
**2. Efficiency**



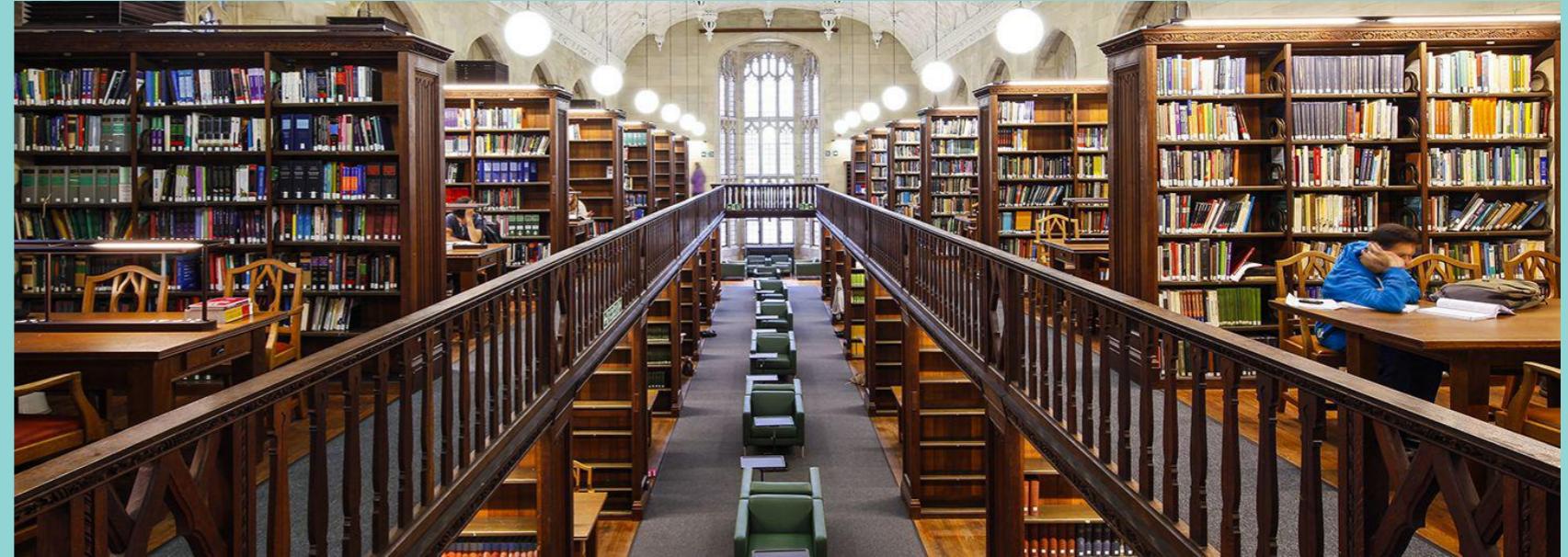
# Why are Data Structures and Algorithms important?



# Why are Data Structures and Algorithms important?

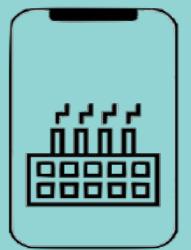
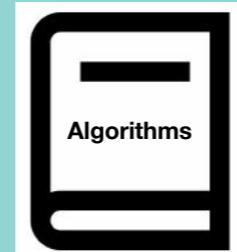


# Why are Data Structures and Algorithms important?

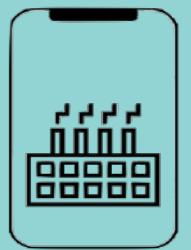


**COMPUTER SCIENCE**

**ALGORITHMS**



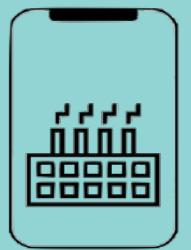
# Why are Data Structures and Algorithms important?



# Why are Data Structures and Algorithms in INTERVIEWS?

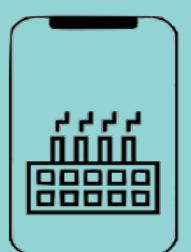
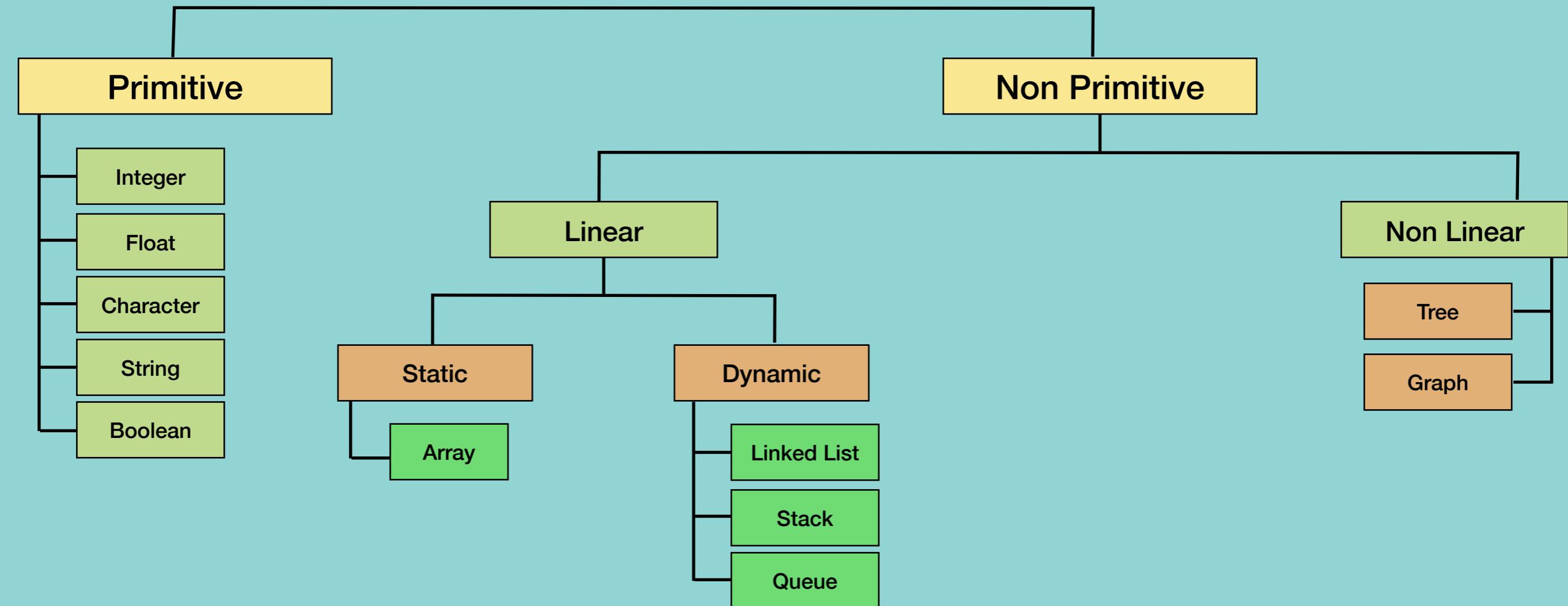


- **Problem solving skills**
- **Fundamental concepts of programming in limited time**



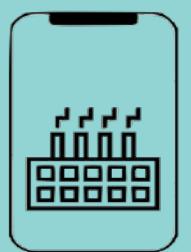
# Types of Data Structures

## Data Structures



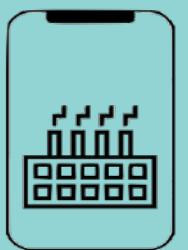
# Primitive Data Structures

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

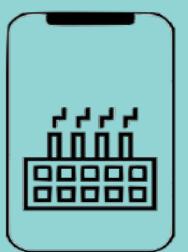
- **Simple recursive algorithms**
- **Divide and conquer algorithms**
- **Dynamic programming algorithms**
- **Greedy algorithms**
- **Brute force algorithms**
- **Randomized algorithms**



# 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
```

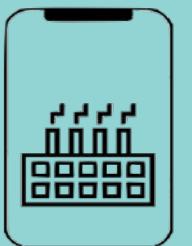


# Types of Algorithms

## 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**



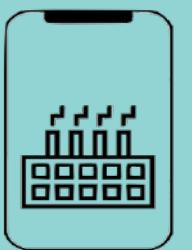
# Types of Algorithms

## 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



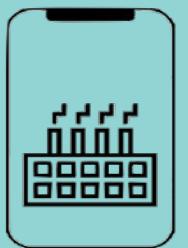
# Types of Algorithms

## Brute force algorithms

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

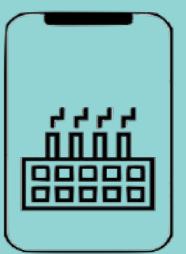
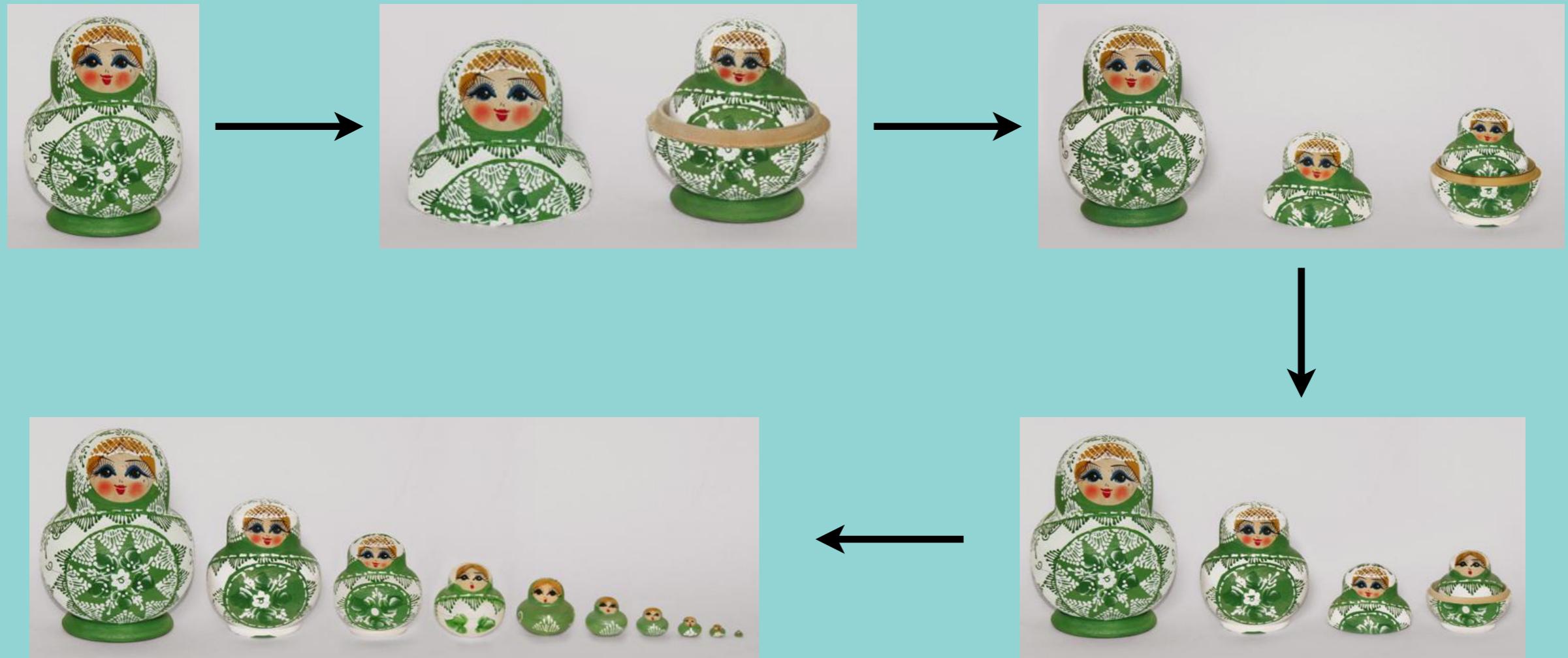
## Randomized algorithms

- 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?

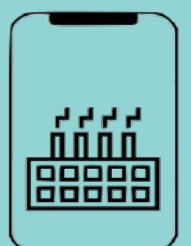
**Recursion = a way of solving a problem by having a function calling itself**



- 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. 🤣🤣🤣

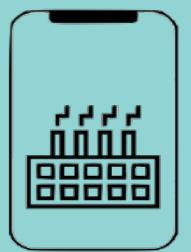


# What is Recursion?

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

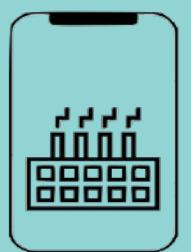


```
def openRussianDoll(doll):
    if doll == 1:
        print("All dolls are opened")
    else:
        openRussianDoll(doll-1)
```



# Why Recursion?

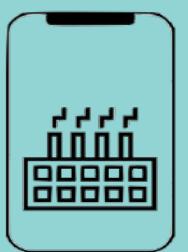
1. Recursive thinking is really important in programming and it helps you break down big problems into smaller ones and easier to use
  - when to choose recursion?
    - ▶ If you can divide the problem into similar sub problems
    - ▶ Design an algorithm to compute nth...
    - ▶ Write code to list 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)
```



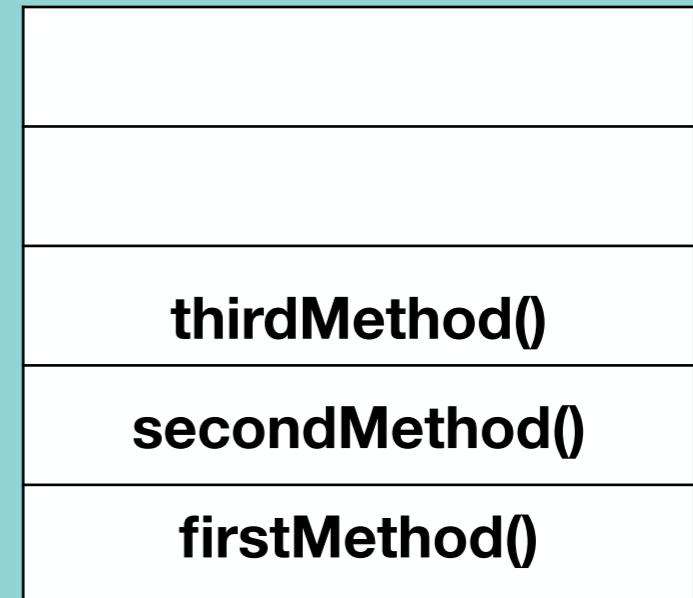
# How Recursion works?

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

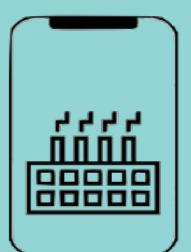
def secondMethod():
    thirdMethod()
    print("I am the second Method")

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

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



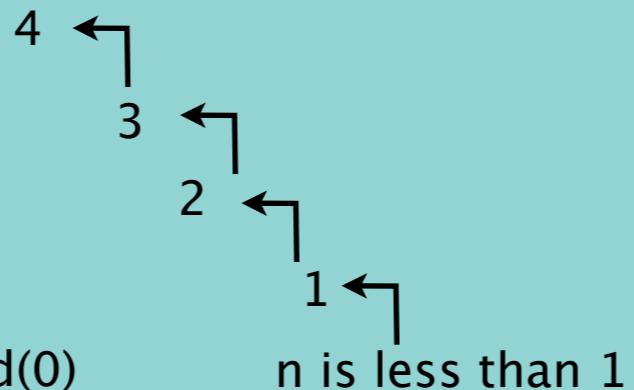
**STACK Memory**



# How Recursion works?

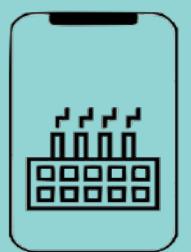
```
def recursiveMethod(n):
    if n<1:
        print("n is less than 1")
    else:
        recursiveMethod(n-1)
        print(n)
```

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



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

**STACK Memory**

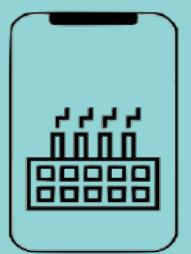


# Recursive vs Iterative Solutions

```
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
```

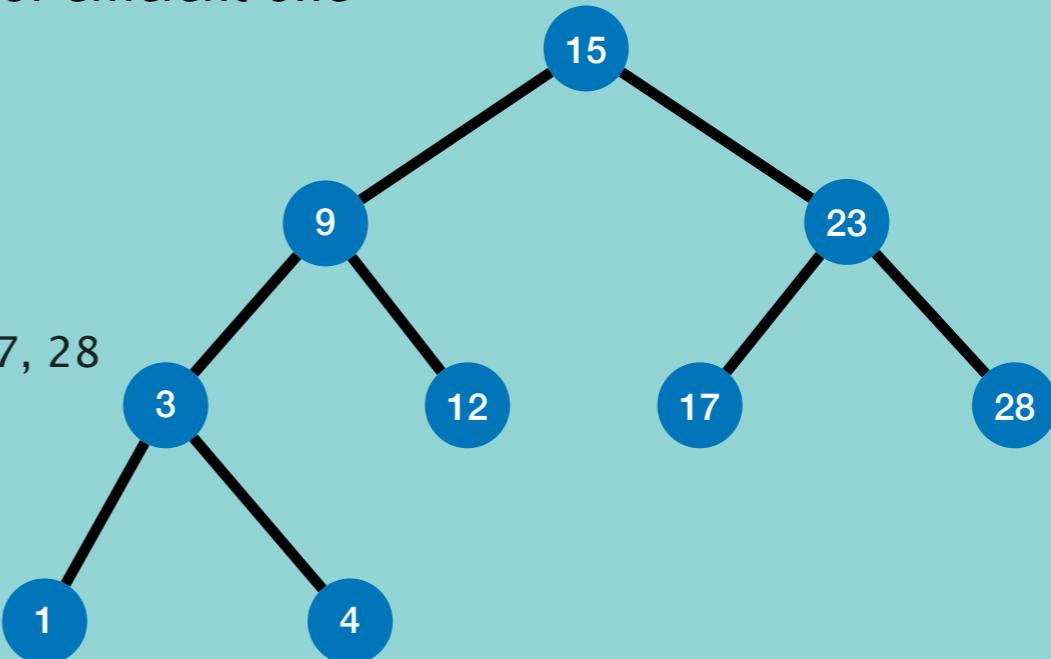
Points	Recursion	Iteration	
Space efficient?	No	Yes	No stack memory require in case of iteration
Time efficient?	No	Yes	In case of recursion system needs more time for pop and push elements to stack memory which makes recursion less time efficient
Easy to code?	Yes	No	We use recursion especially in the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



# When to Use/Avoid Recursion?

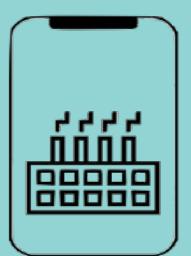
## When to use it?

- 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.
- $0!=1$ .

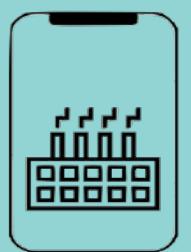
## Example 1

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

## 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

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

↓

$(n-1)!$

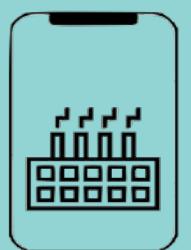
$$(n-1)! = (n-1) * (n-1-1) * (n-1-2) * \dots * 2 * 1 = (n-1) * (n-2) * (n-3) * \dots * 2 * 1$$

## Step 2 : Base case – the stopping criterion

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

## Step 3 : Unintentional case – the constraint

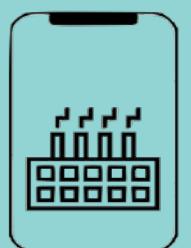
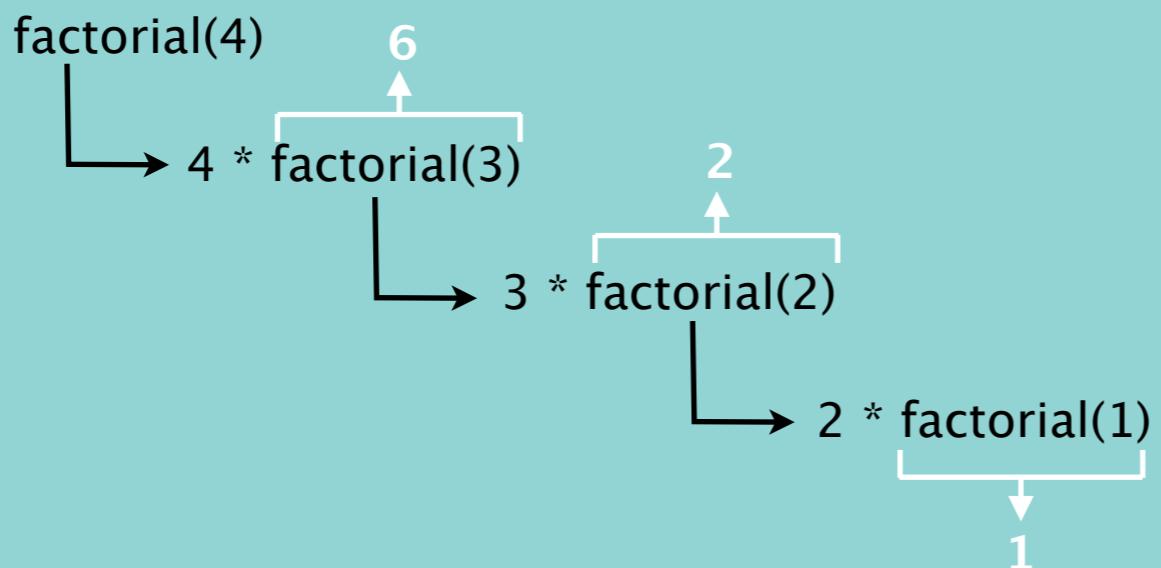
- $\text{factorial}(-1)$  ??
- $\text{factorial}(1.5)$  ??



# 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

$$5 = 3 + 2$$

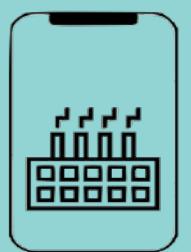
$$f(n) = f(n-1) + f(n-2)$$

## Step 2 : Base case – the stopping criterion

- 0 and 1

## Step 3 : Unintentional case – the constraint

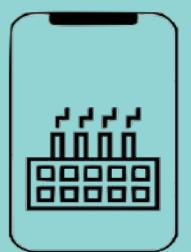
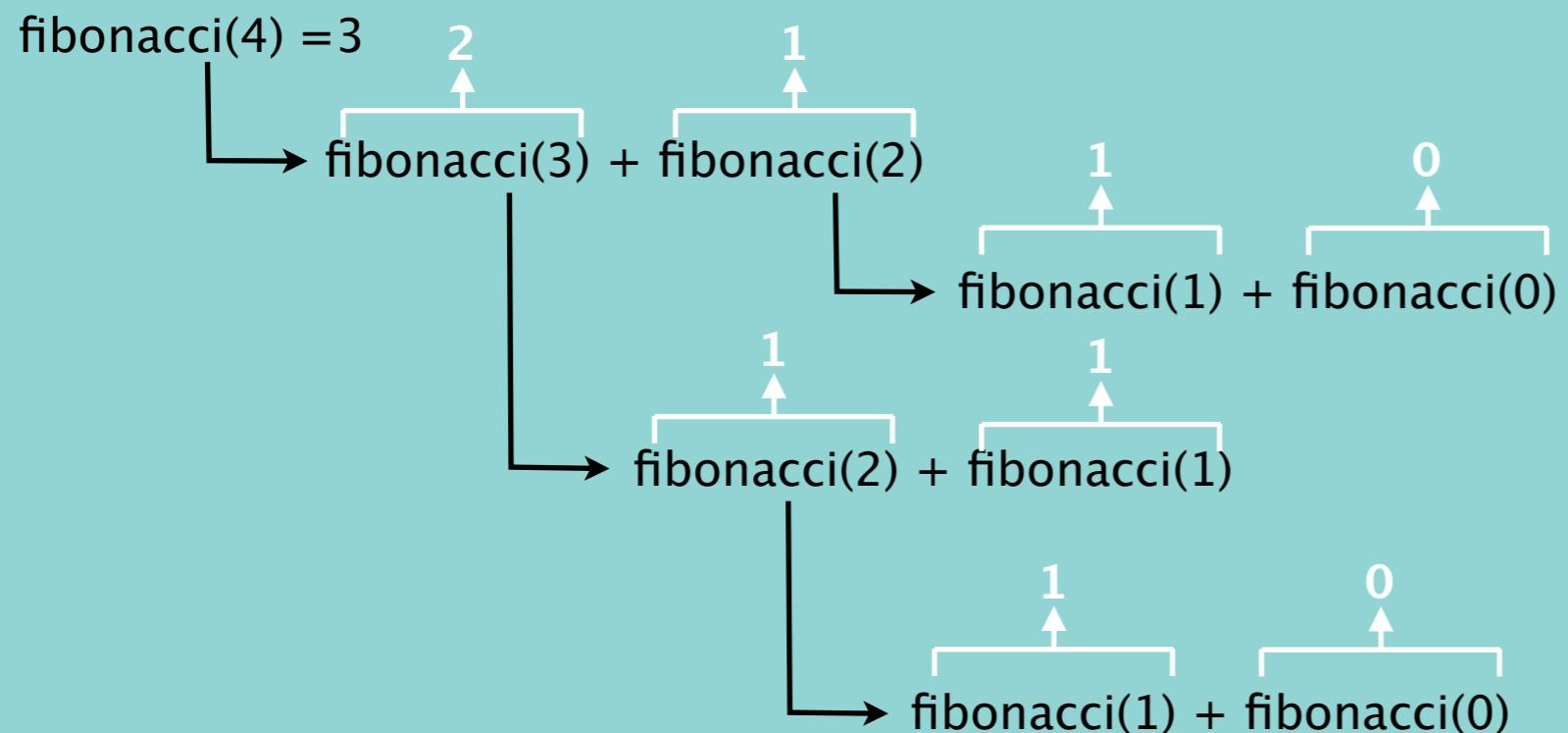
- fibonacci(-1) ??
- fibonacci(1.5) ??



# 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

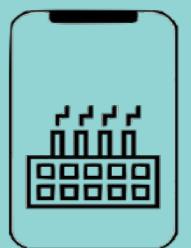
$11/10 = 1$  and Remainder = 1

## 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

$$n = 0$$

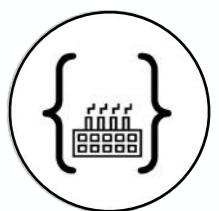
### Step 3 : Unintentional case - the constraint

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

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

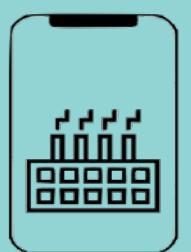
$$\gcd(a, b) = \gcd(b, a \bmod b)$$

## 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

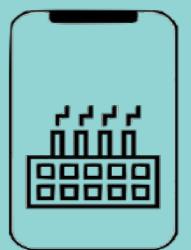
10 to binary

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

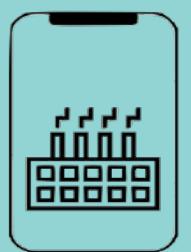


900



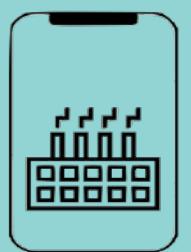
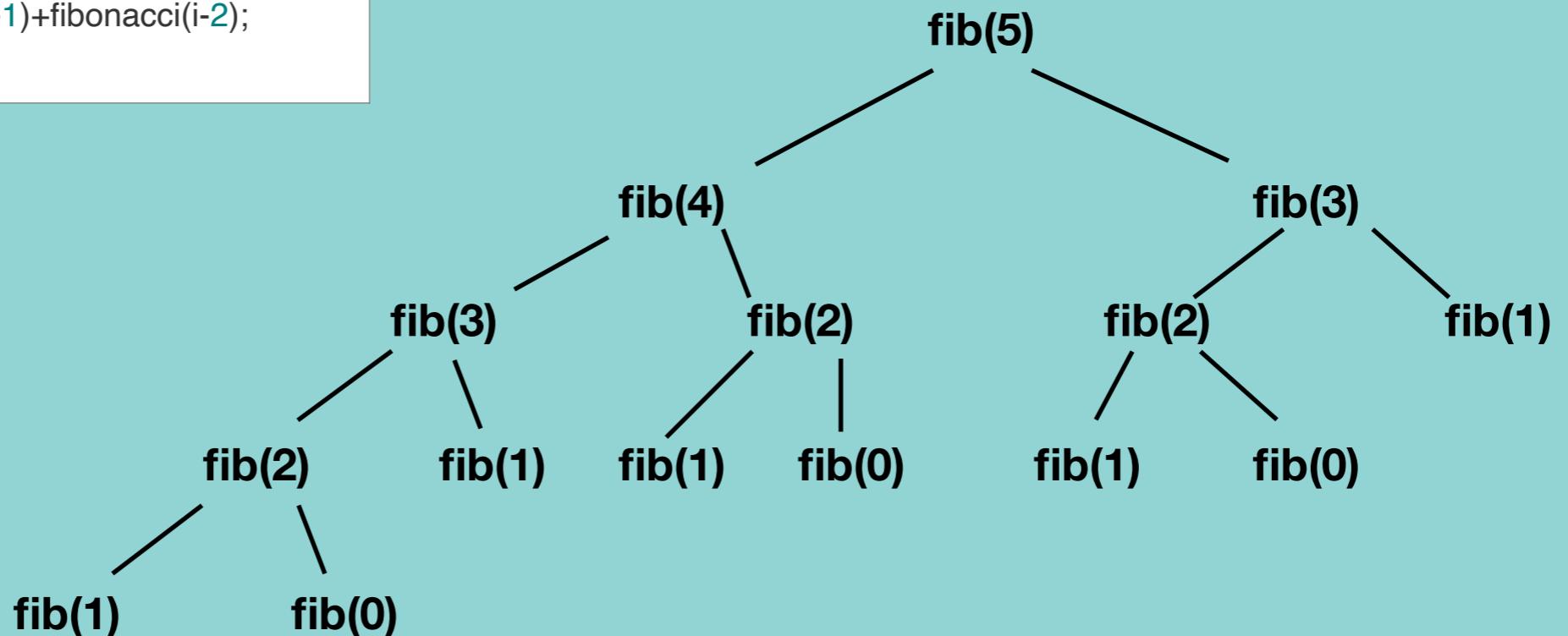
850

**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);  
}
```



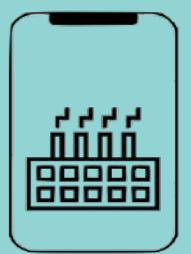
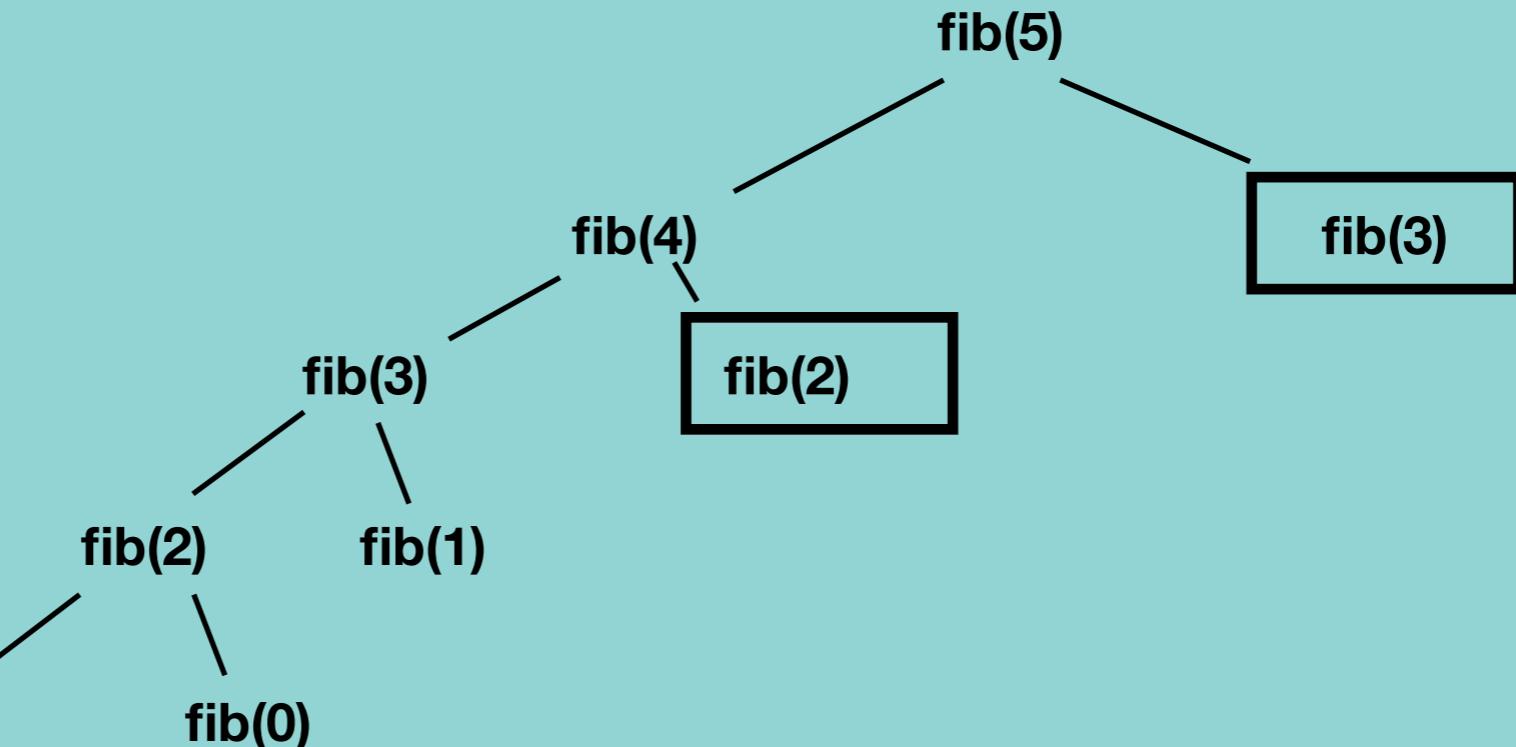
# Dynamic programming & Memoization

```
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;

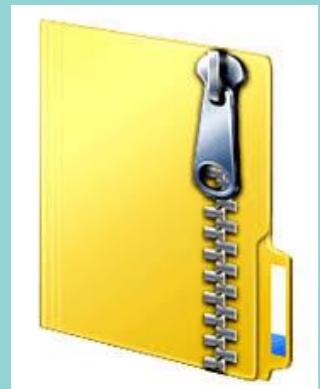
    for (i = 2; i <= n; i++)
    {
        /* Add the previous 2 numbers
        in the series
        and store it */
        f[i] = f[i-1] + f[i-2];
    }

    return f[n];
}
```

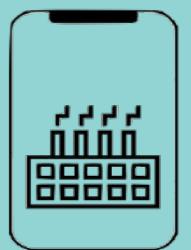


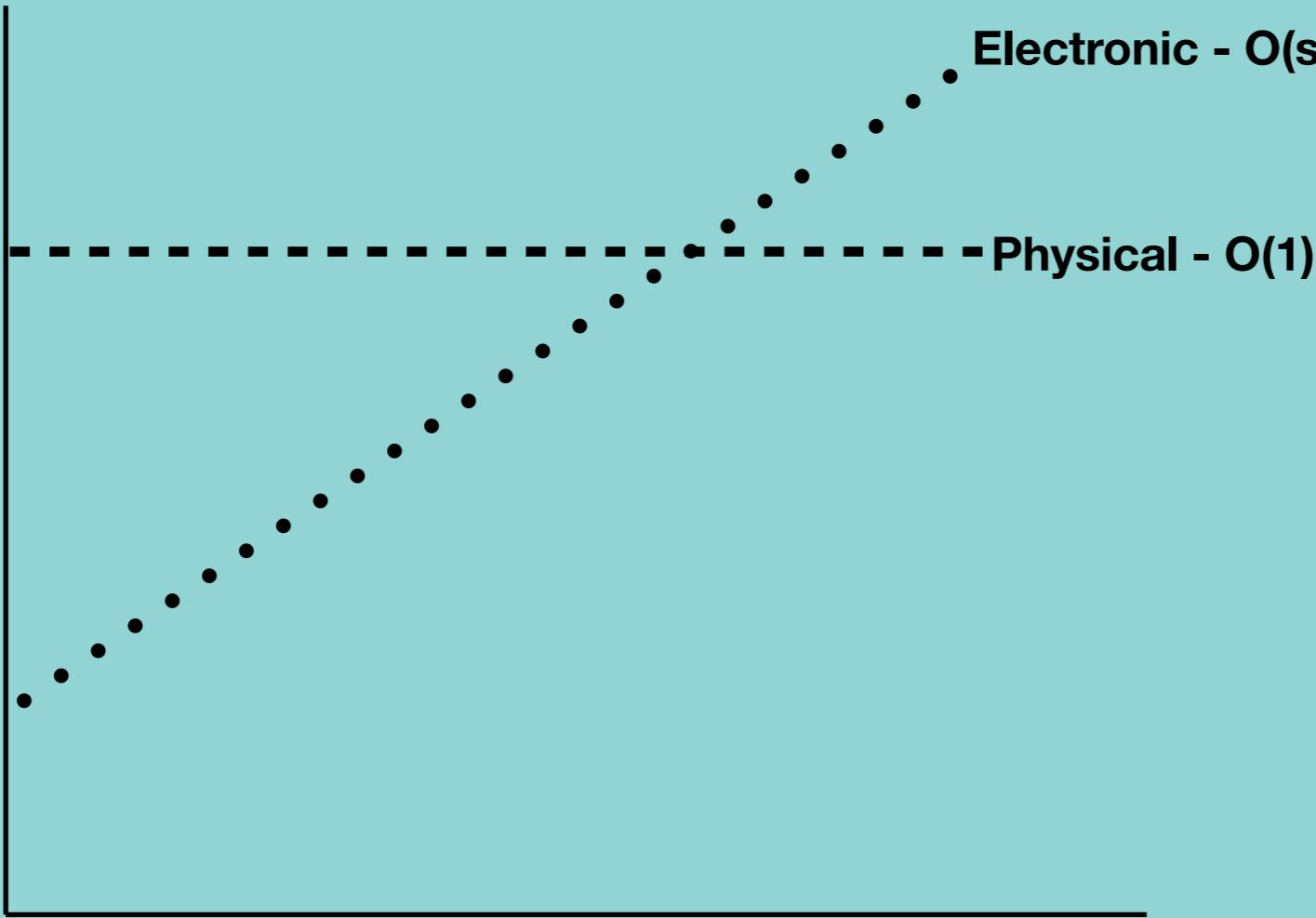
# Big O

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



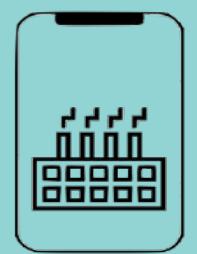
**Size : 1TB**





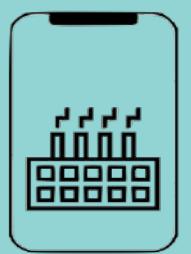
## Types

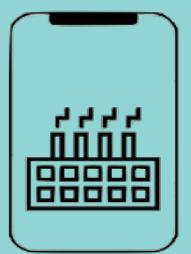
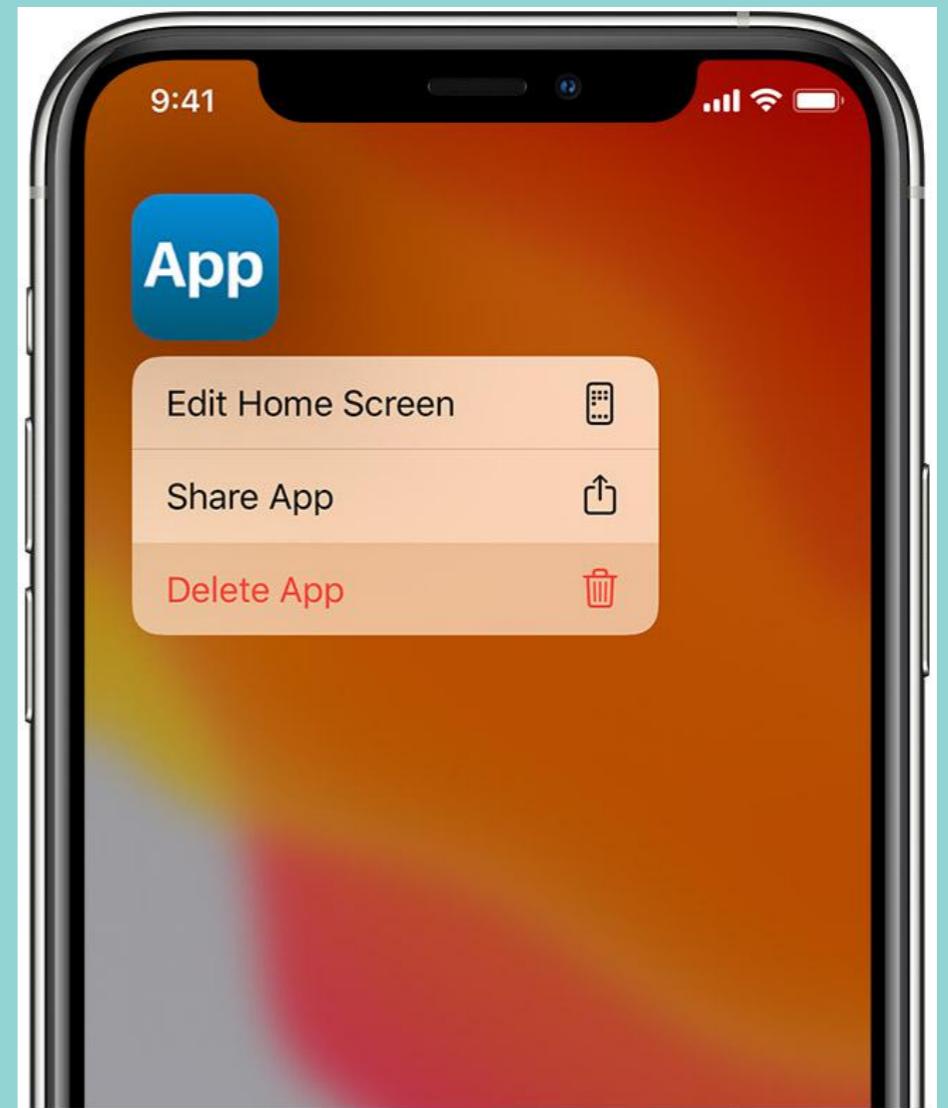
$O(N)$ ,  $O(N^2)$ ,  $O(2^N)$



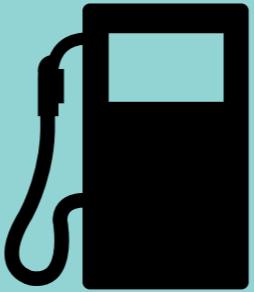
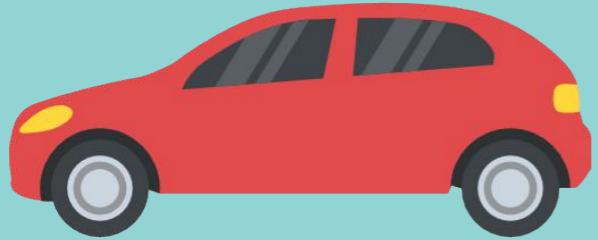


**Time complexity : O( $wh$ )**

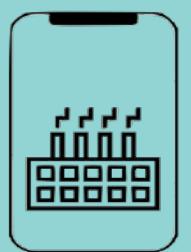




# Algorithm run time notations

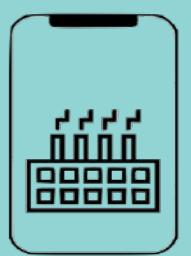
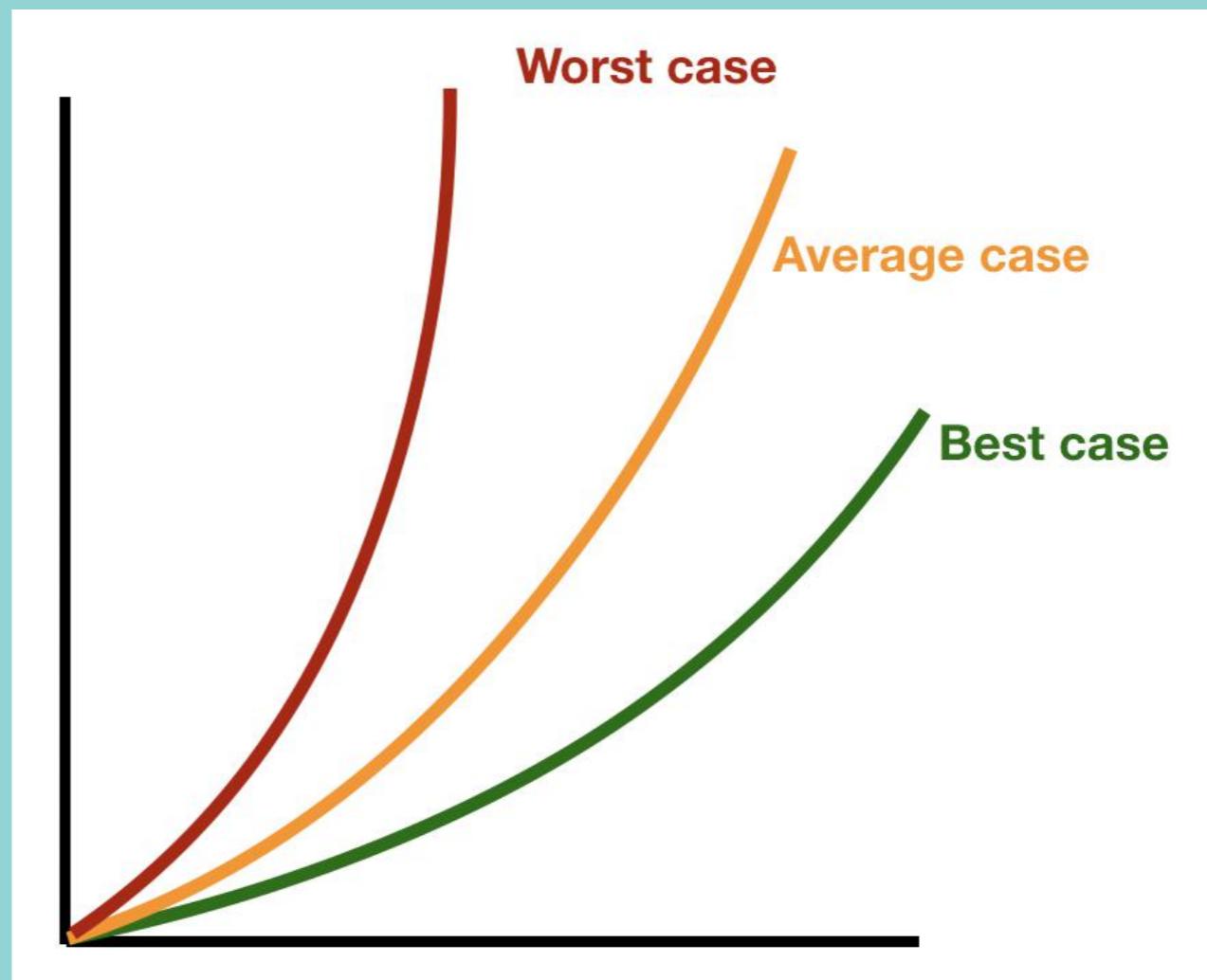


- **City traffic - 20 liters**
- **Highway - 10 liters**
- **Mixed condition - 15 liters**



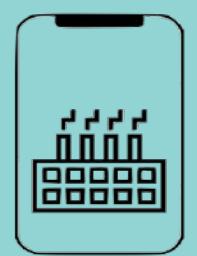
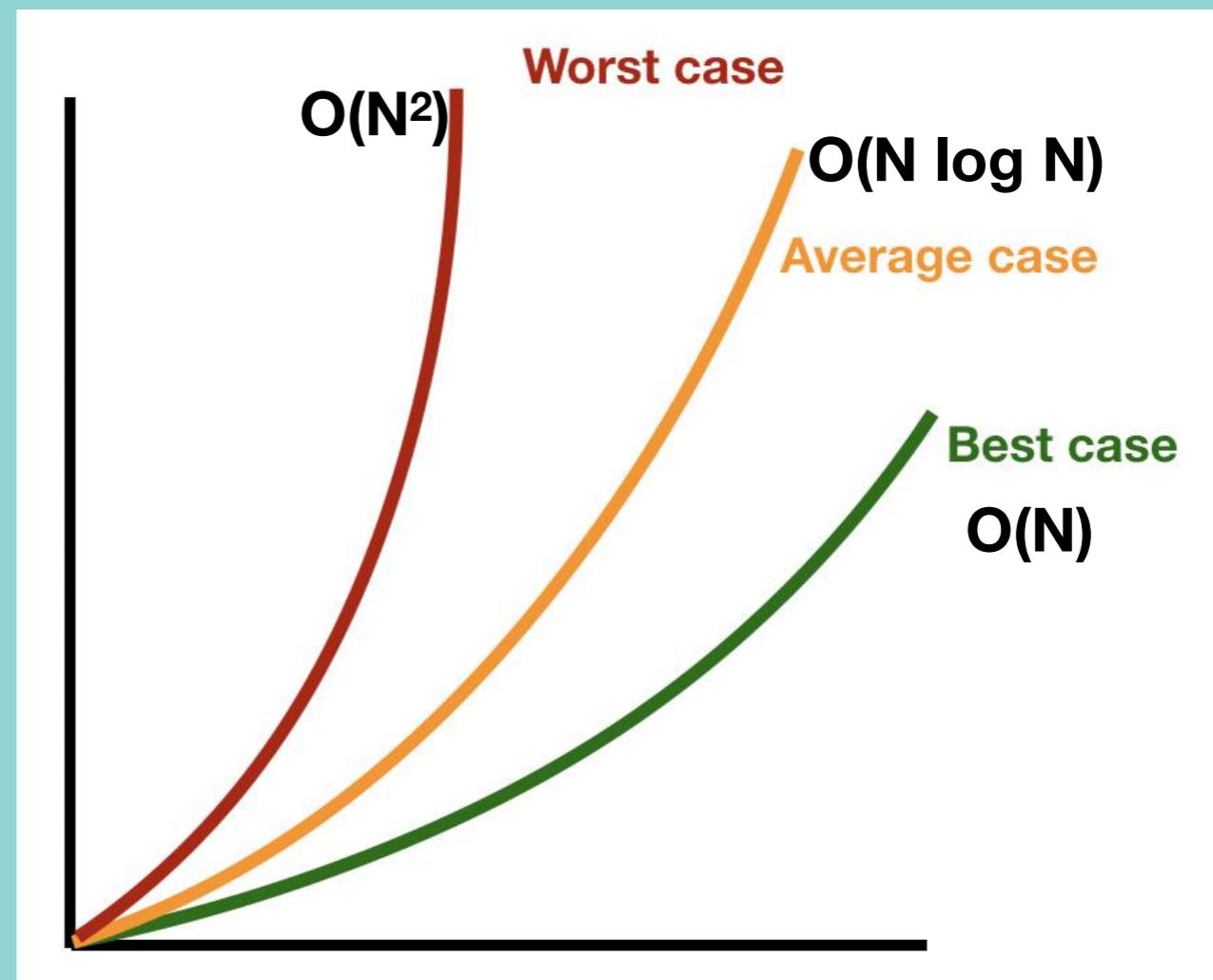
# Algorithm run time notations

- Best case
- Average case
- Worst case



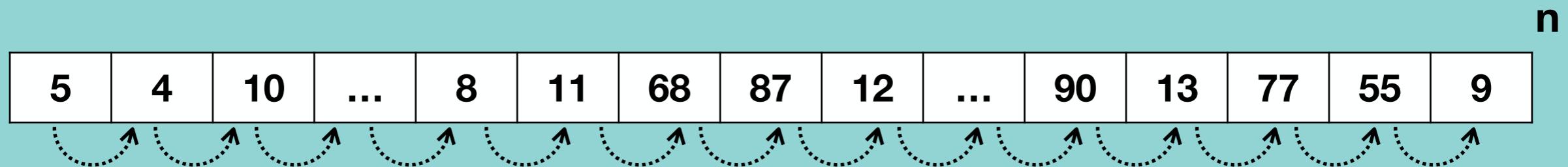
# Algorithm run time notations

## Quick sort algorithm



# Big O, Big Theta and Big Omega

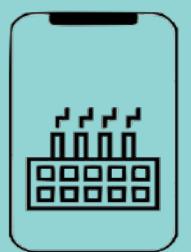
- **Big O** : It is a complexity that is going to be less or equal to the worst case.
- **Big -  $\Omega$  (Big-Omega)** : It is a complexity that is going to be at least more than the best case.
- **Big Theta (Big -  $\Theta$ )** : It is a complexity that is within bounds of the worst and the best cases.



**Big O -  $O(N)$**

**Big  $\Omega$  -  $\Omega(1)$**

**Big  $\Theta$  -  $\Theta(n/2)$**

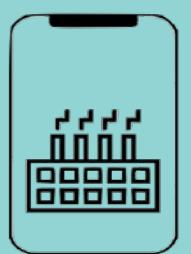


# Algorithm run time complexities

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

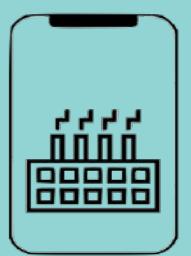
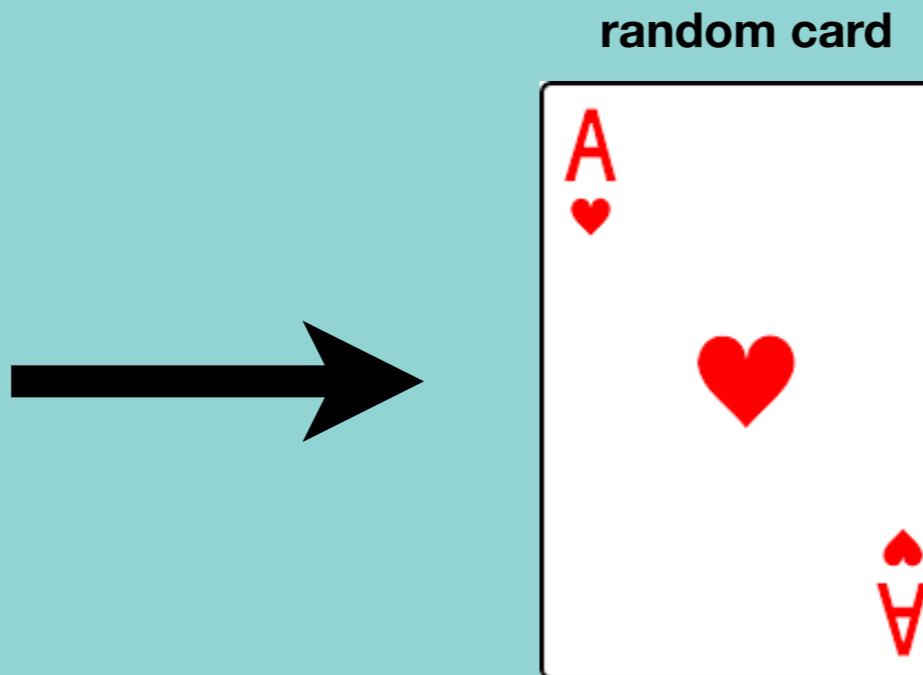
```
array = [1, 2, 3, 4, 5]
array[0] // It takes constant time to access first element
```



# 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

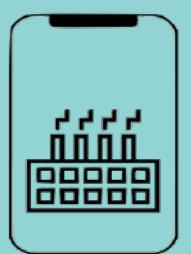


# Algorithm run time complexities

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

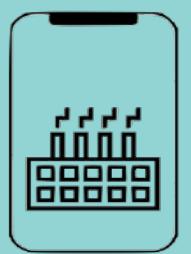
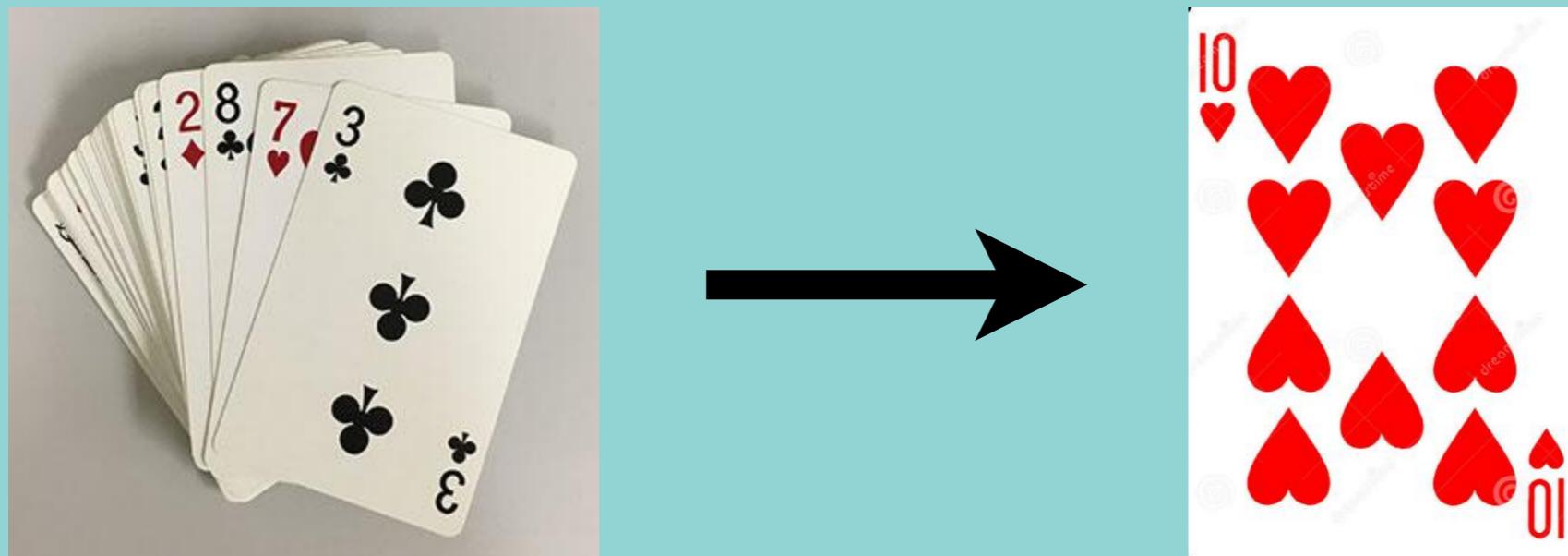
```
array = [1, 2, 3, 4, 5]
for element in array:
    print(element)
//linear time since it is visiting every element of array
```



# 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

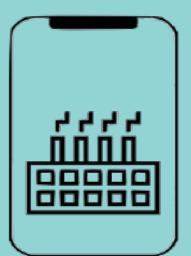


# Algorithm run time complexities

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

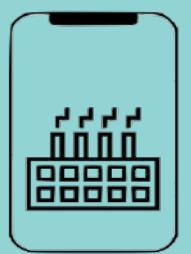
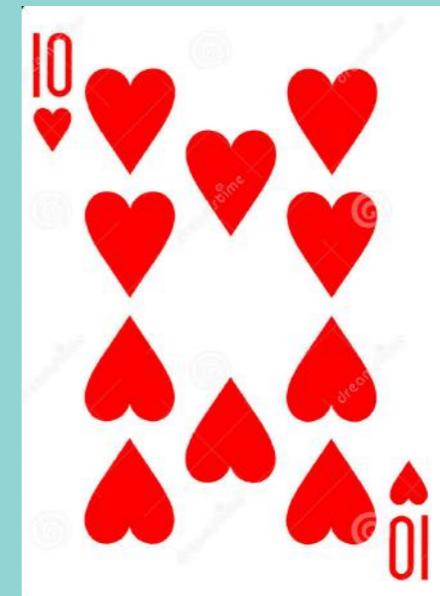
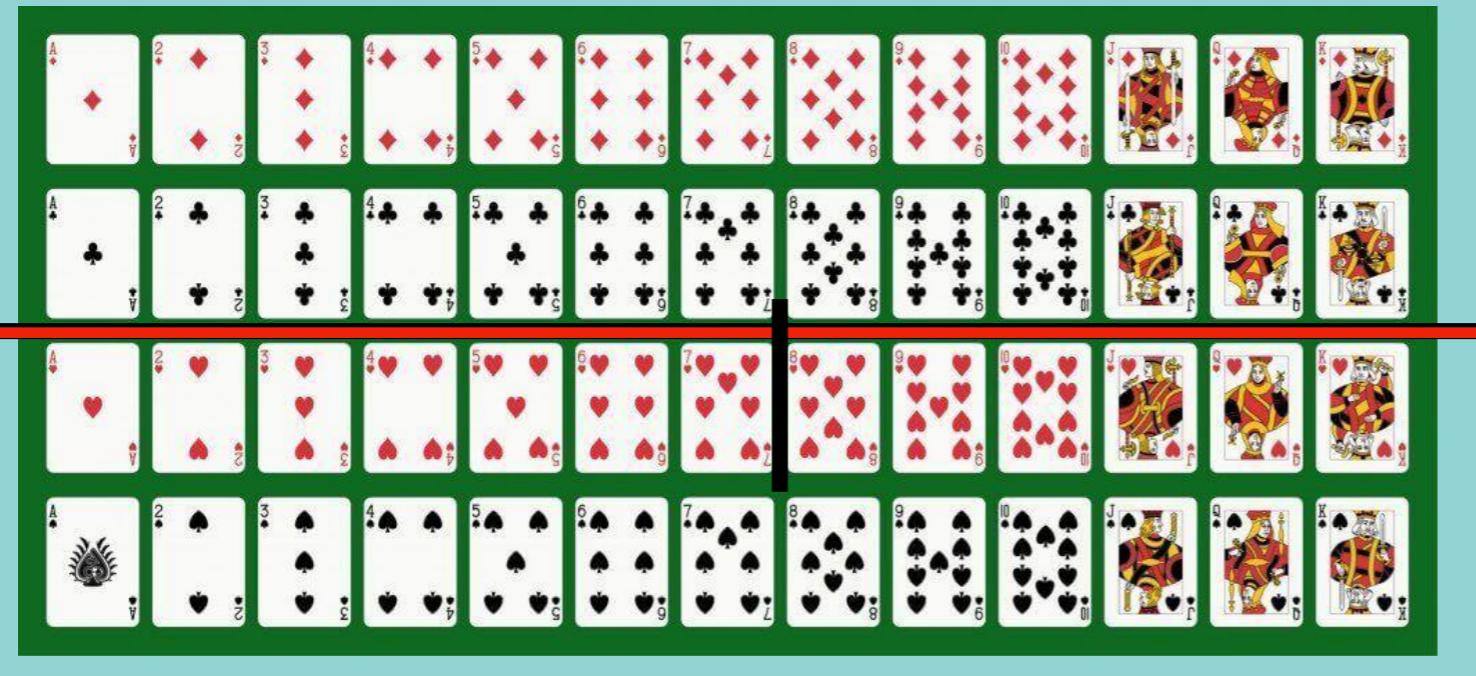
```
array = [1, 2, 3, 4, 5]
for index in range(0,len(array),3):
    print(array[index])
//logarithmic time since it is visiting only some elements
```



# 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 <sup>2</sup> )	Quadratic	Looking at every index in the array twice
O(2 <sup>N</sup> )	Exponential	Double recursion in Fibonacci

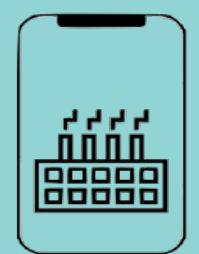
## O(LogN) - Logarithmic time

### 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 */
```

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



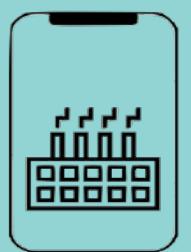
# Algorithm run time complexities

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)
```



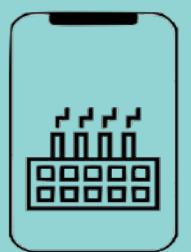
# Algorithm run time complexities

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]

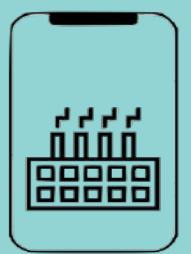
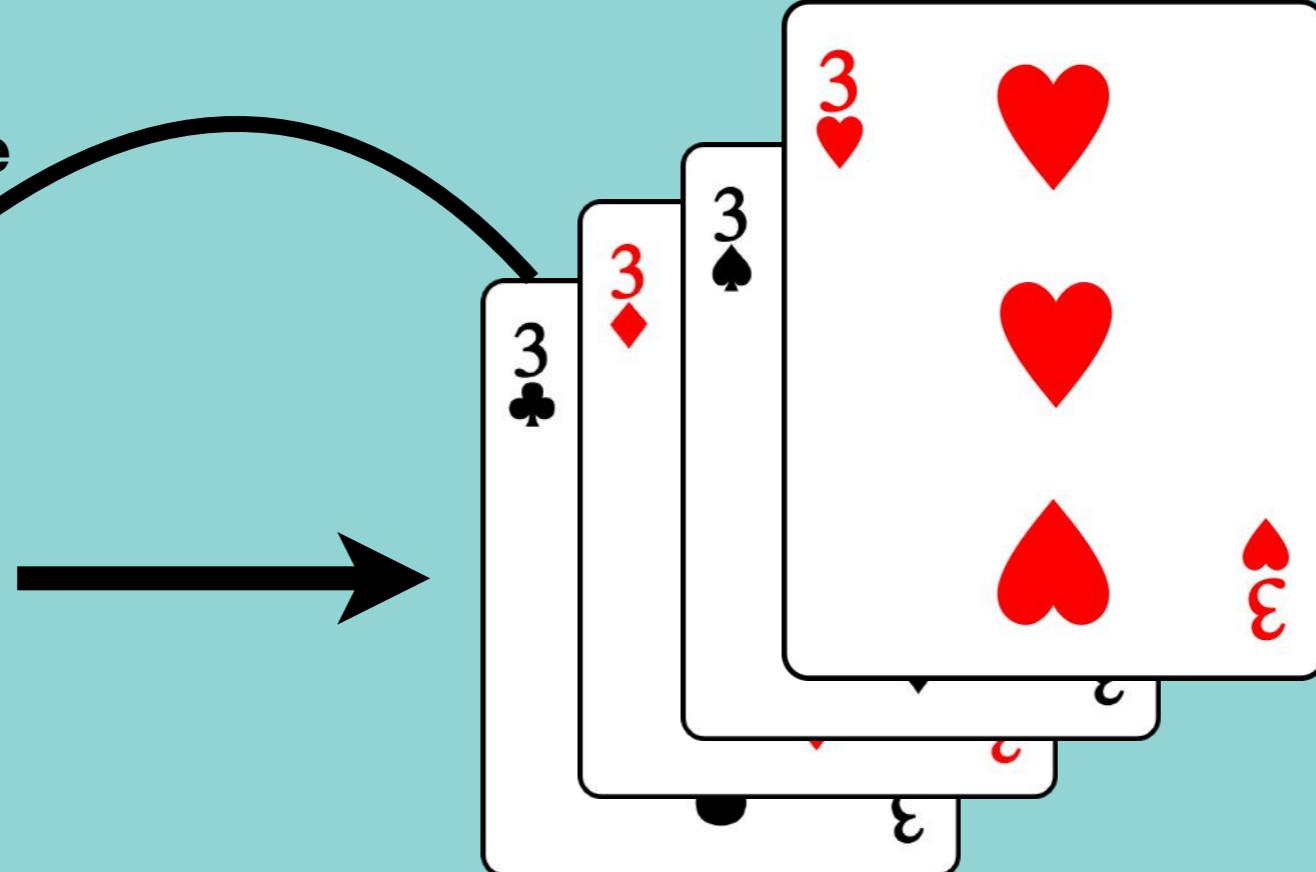
for x in array:
    for y in array:
        print(x,y)
```



# 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

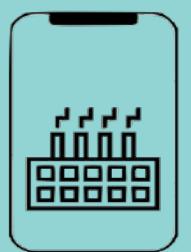


# Algorithm run time complexities

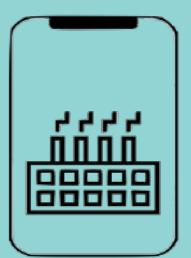
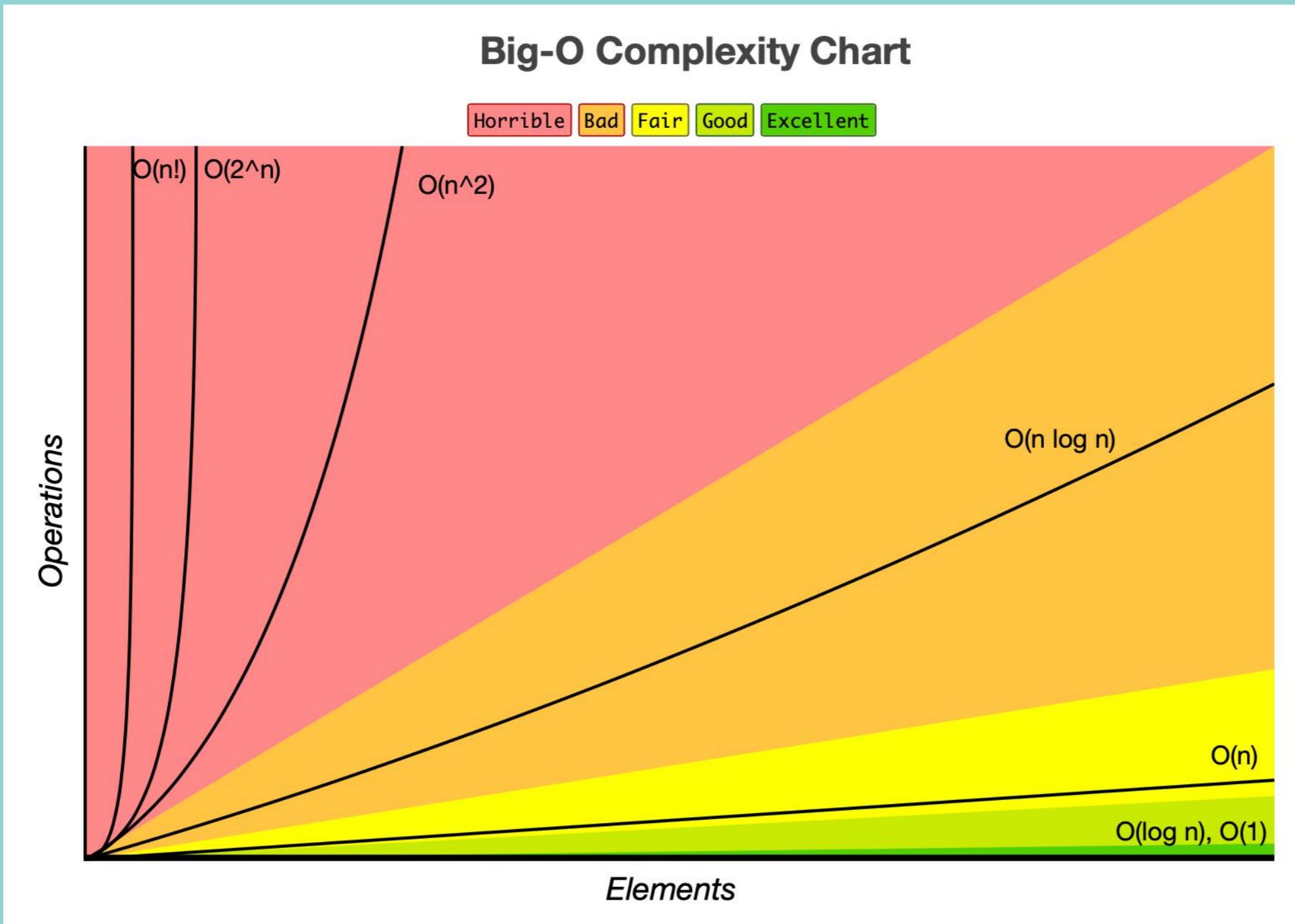
Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

## $O(2^N)$ - 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**

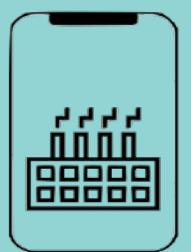
$$a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

**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(n<sup>2</sup>)**

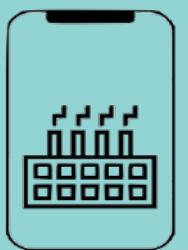


## Space complexity - example

```
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)

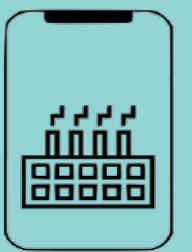


## Space complexity - example

```
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

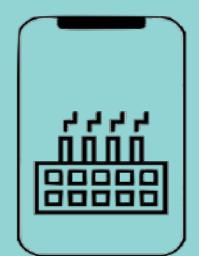
```
def sum(n):  
    if n <= 0:  
        return 0  
    else:  
        return n + sum(n-1)
```

## 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

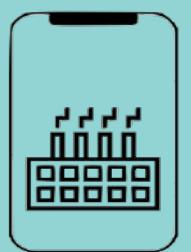
$$O(2N) \longrightarrow O(N)$$

## 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**

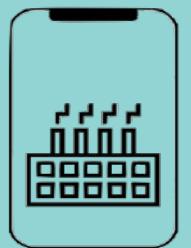


**Slow computer**  
**Slow memory access**  
**Higher constant**

- 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

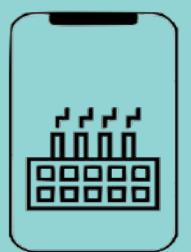
```
for a in arrayA:  
    print(a)  
  
for b in arrayB:  
    print(b)
```

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

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

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

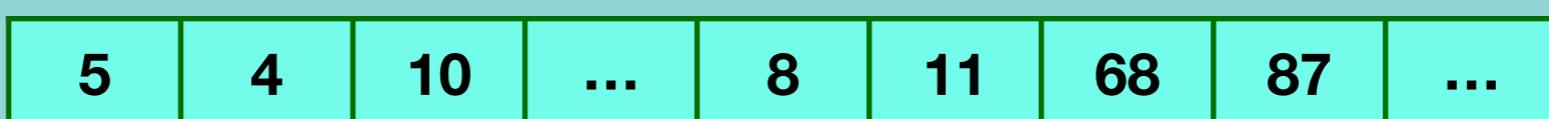
- If your algorithm is in the form “do this, then when you are all done, do that” then you add the runtimes.
- If your algorithm is in the form “do this for each time you do that” then you multiply the runtimes.



# How to measure the codes using Big O?

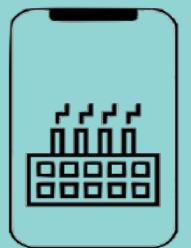
No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	$O(1)$
Rule 2	A simple “for” loop from 0 to n ( with no internal loops)	$O(n)$
Rule 3	A nested loop of the same type takes quadratic time complexity	$O(n^2)$
Rule 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



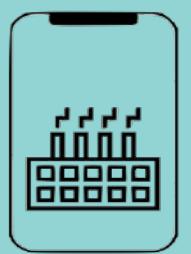
findMaxNumRec(A, 3) → max(A[3-1], 11) → max(12, 11)=12



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)
```

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

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

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

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

}

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

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

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

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

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

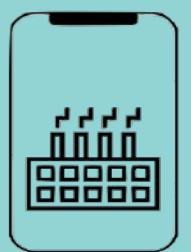
.

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

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

$$= n-1+1$$

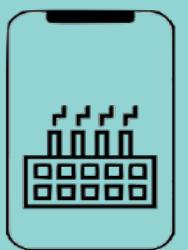
$$= n$$



# How to measure Recursive Algorithm that make multiple calls?

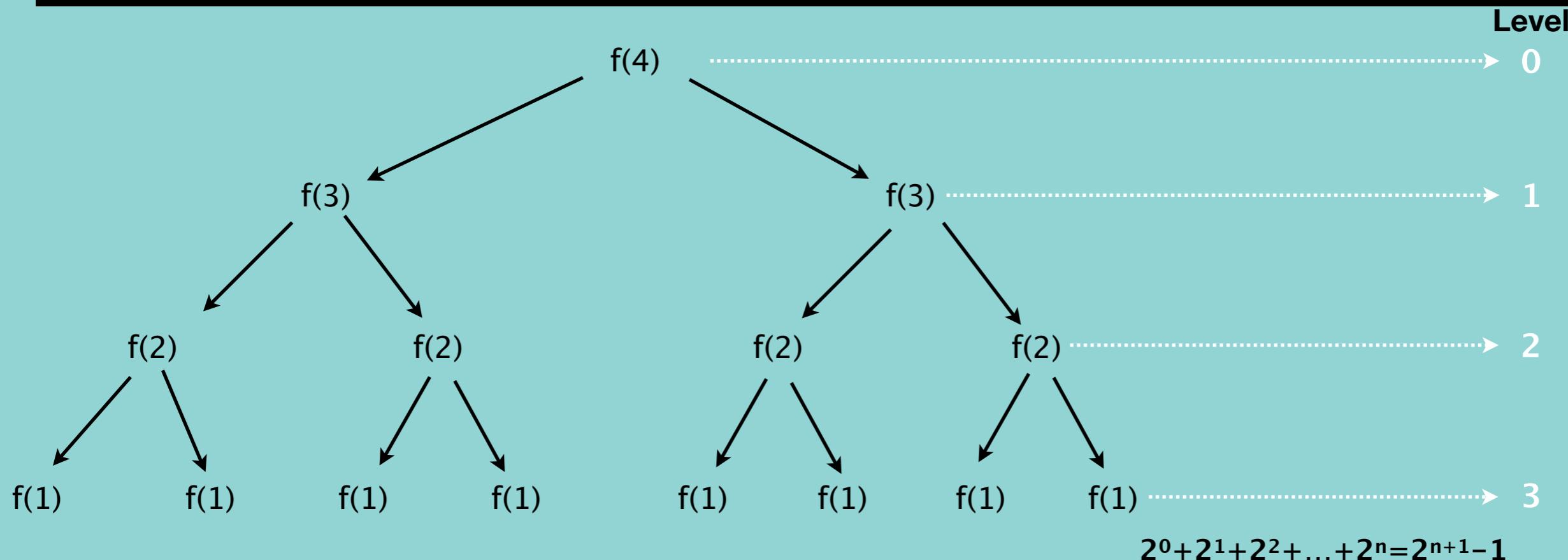
```
def findMaxNumRec(sampleArray, n):
    if n == 1:
        return sampleArray[0]
    return max(sampleArray[n-1], findMaxNumRec(sampleArray, n-1))
```

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



# How to measure Recursive Algorithm that make multiple calls?

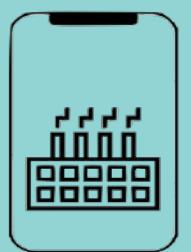
```
def f(n):
    if n <= 1:
        return 1
    return f(n-1) + f(n-1)
```



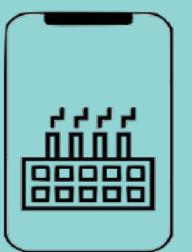
N	Level	Node#	Also can be expressed..	or..
4	0	1		$2^0$
3	1	2	$2 * \text{previous level} = 2$	$2^1$
2	2	4	$2 * \text{previous level} = 2 * 2^1 = 2^2$	$2^2$
1	3	8	$2 * \text{previous level} = 2 * 2^2 = 2^3$	$2^3$

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

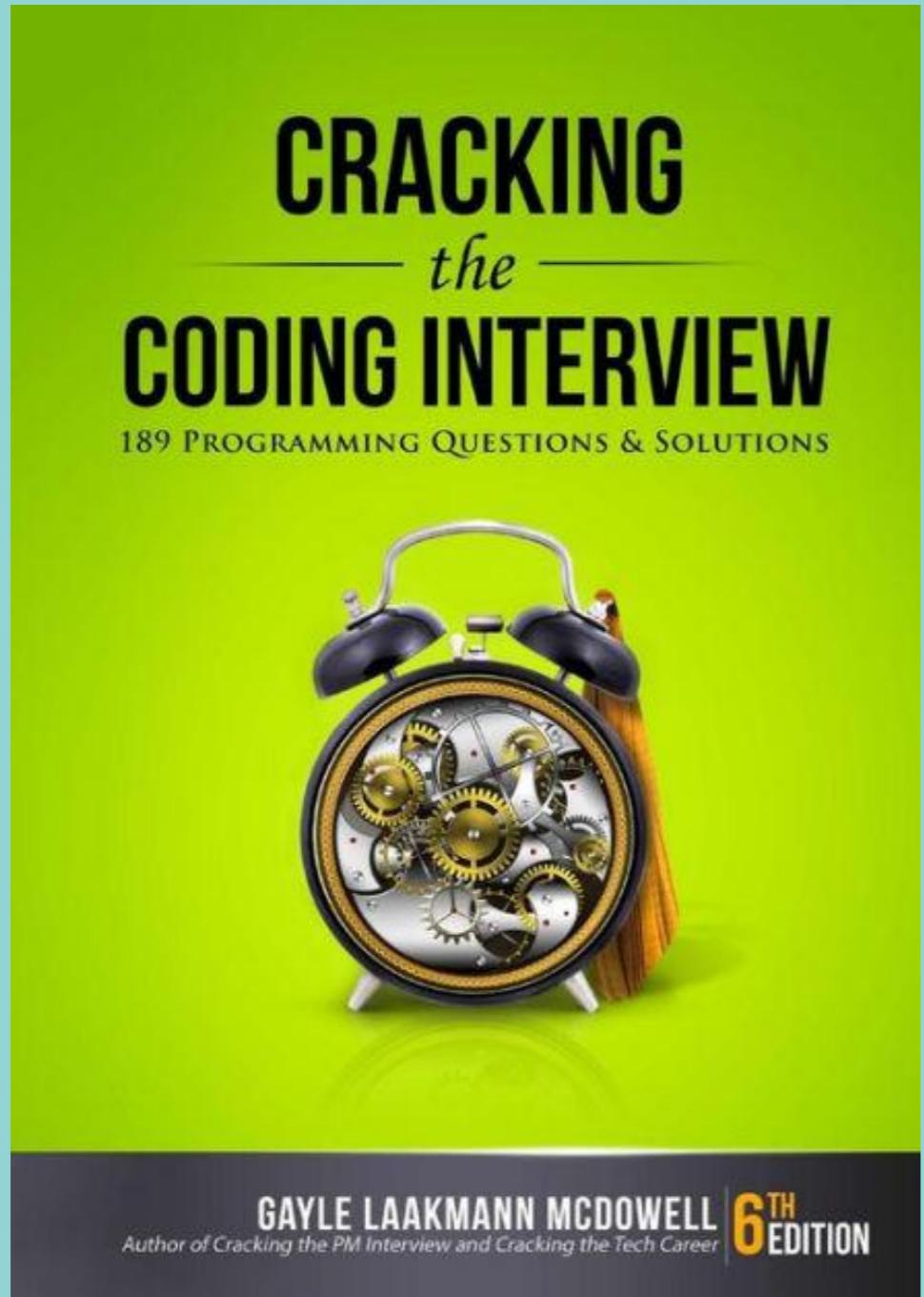
AppMillers  
www.appmillers.com



# Top 10 Big O interview Questions

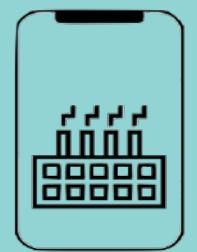


# Top 10 Big O interview Questions



in Java 😞

Java to Python 😊



# Interview Questions - 1

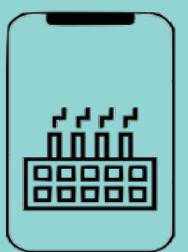
What is the runtime of the below code?

```
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)

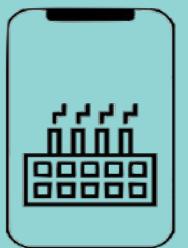


# Interview Questions - 2

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

$(n-1) + (n-2) + (n-3) + \dots + 2 + 1$

$= 1 + 2 + \dots + (n-3) + (n-2) + (n-1)$

$= n(n-1)/2$

$= n^2/2 + n$

$= n^2$

Time Complexity :  $O(N^2)$

## 2. Average Work

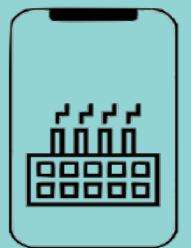
Outer loop – N times

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?

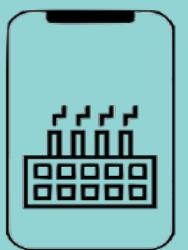
```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            if arrayA[i] < arrayB[j]:
                print(str(arrayA[i]) + "," + str(arrayB[j]))
```

```
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?

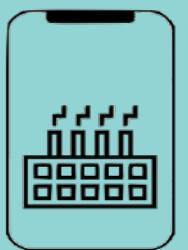
```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            for k in range(0,100000): -----> O(ab)
                print(str(arrayA[i]) + "," + str(arrayB[j])) -----> O(1)
```

a = len(arrayA)

b = len(arrayB)

100,000 units of work is still constant

Time Complexity : O(ab)



# Interview Question - 6

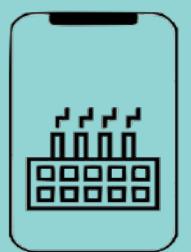
What is the runtime of the below code?

```
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)



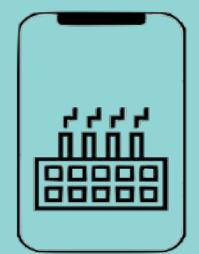
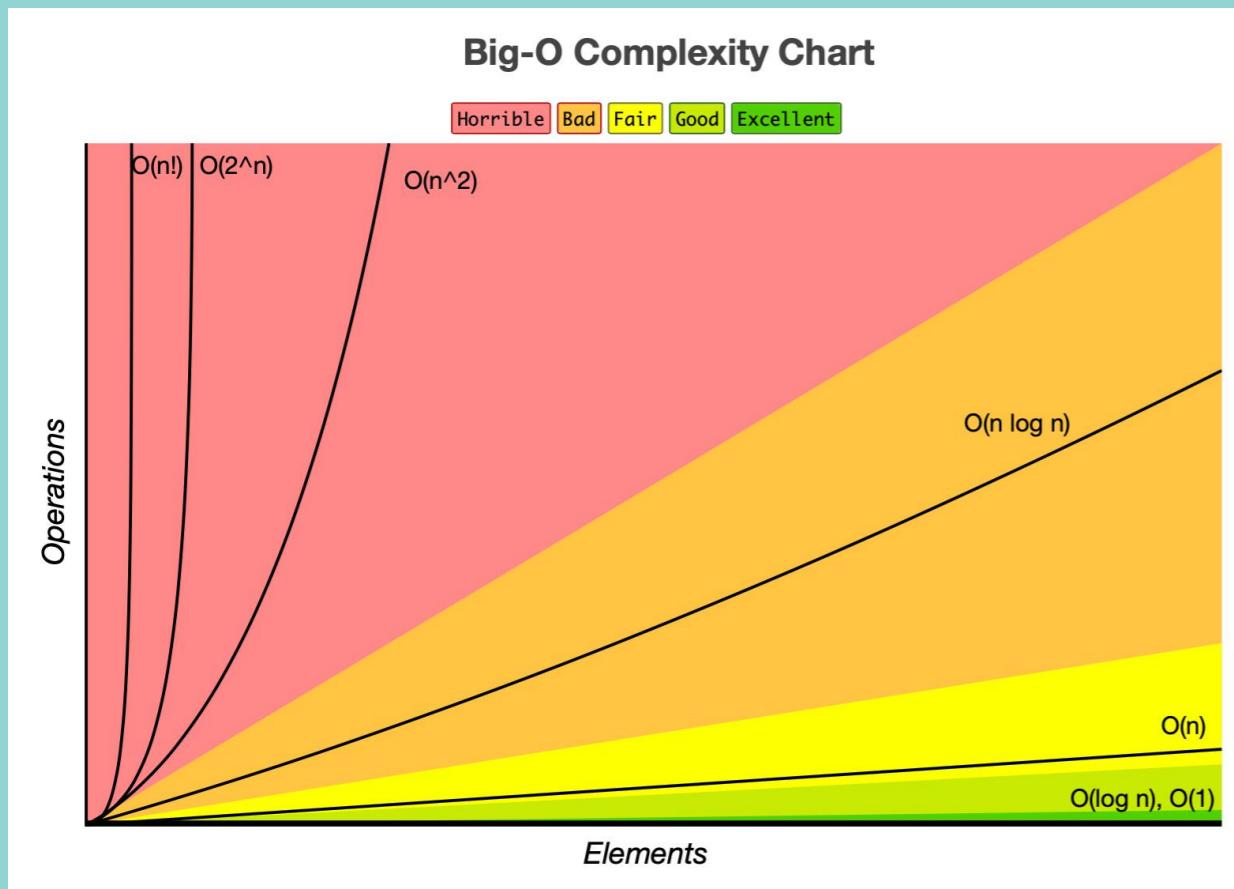
Time Complexity : O(N)



# Interview Questions - 7

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

1.  $O(N + \cancel{P})$ , where  $P < N/2 \longrightarrow O(N) \quad \checkmark$
2.  $\cancel{O(2N)} \longrightarrow O(N) \quad \checkmark$
3.  $\cancel{O(N+\log N)} \longrightarrow O(N) \quad \checkmark$
4.  $\cancel{O(N + N\log N)} \longrightarrow O(N\log N) \quad \times$
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)
```

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

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

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

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

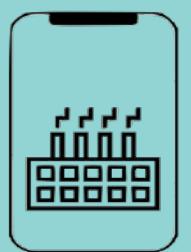
$$M(n-1) = O(1) + M((n-1)-1)$$

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

}

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

Time Complexity : O(N)



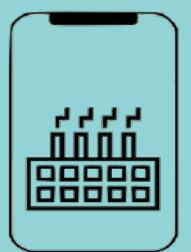
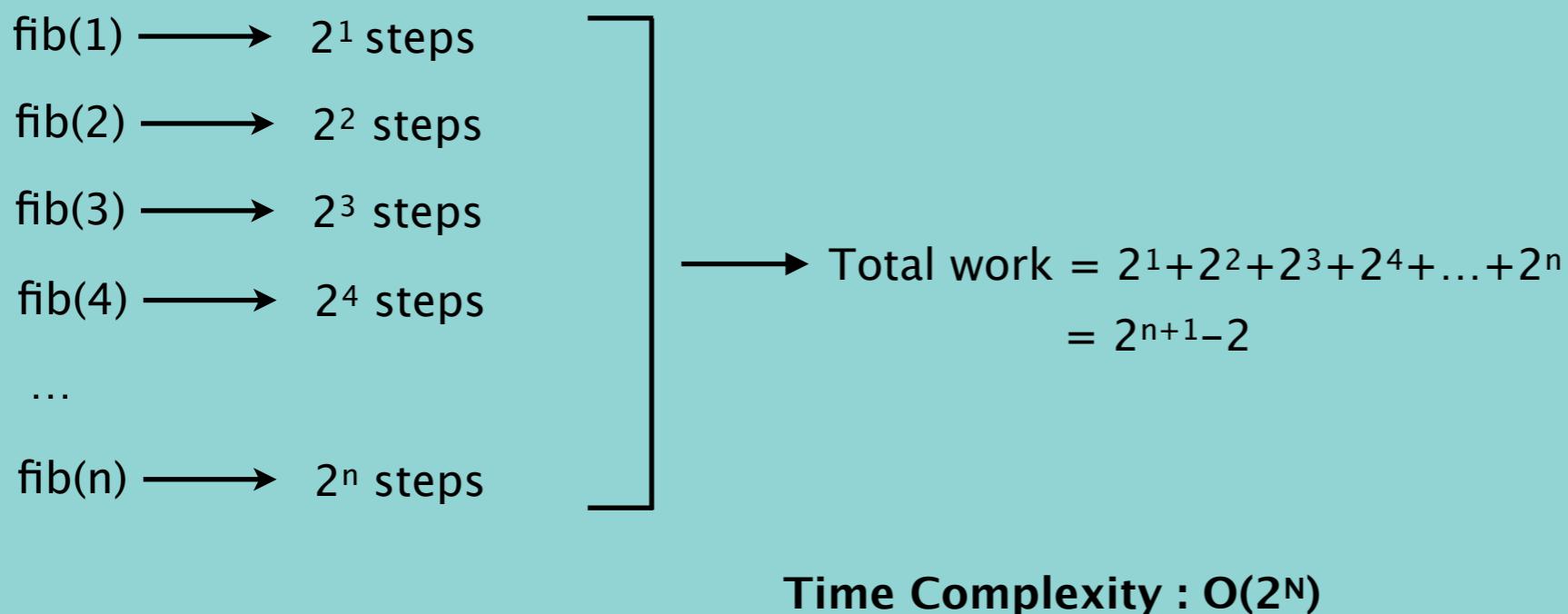
# Interview Question - 9

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<sup>depth</sup> -----> O(2<sup>N</sup>)**

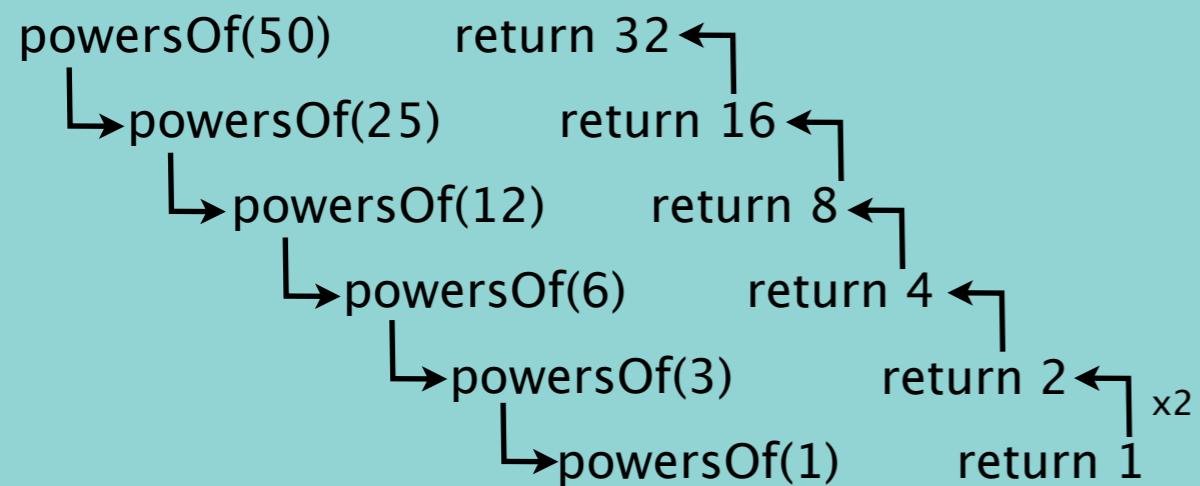


# Interview Question - 10

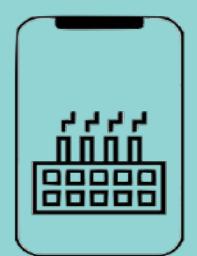
What is the runtime of the below code?

```
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
```

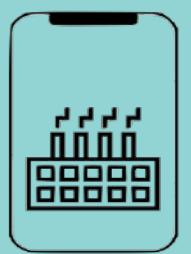
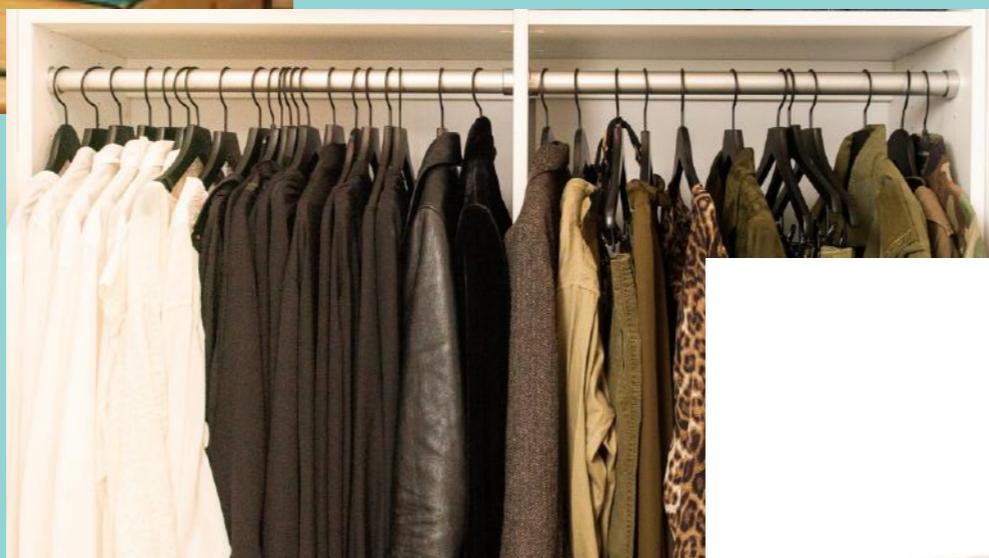
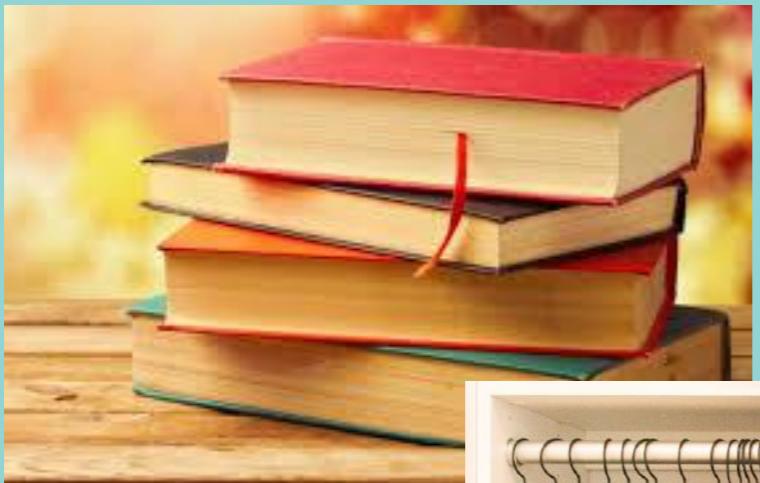
n=50



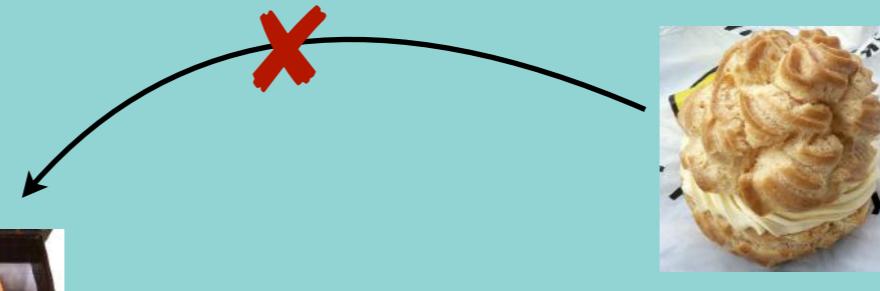
Time Complexity : O(logN)



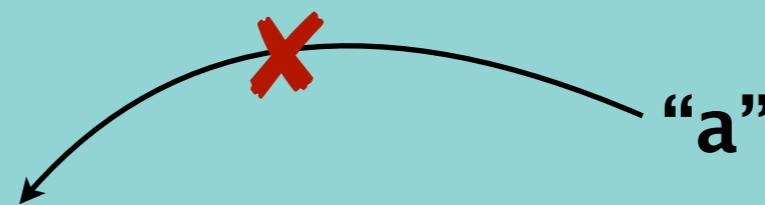
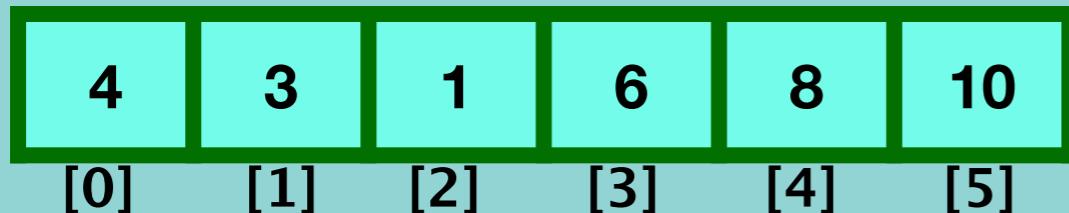
# Arrays



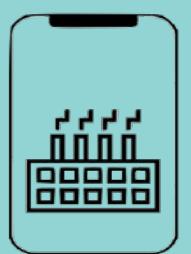
# Arrays



- 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

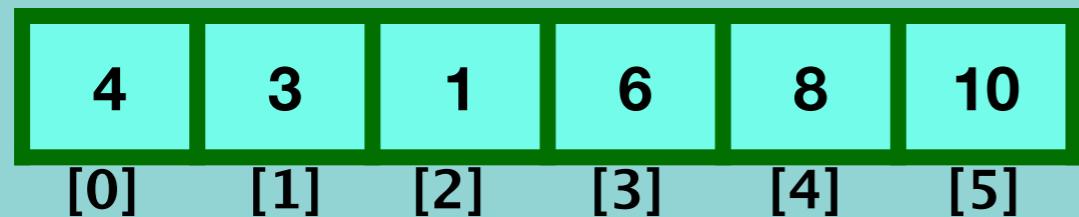


- Array can store data of specified type
- Elements of an array are located in a contiguous
- Each element of an array has a unique index
- The size of an array is predefined and cannot be modified



# 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.



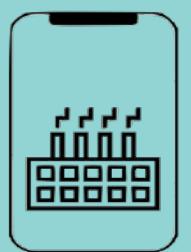
## Why do we need an Array?

3 variables

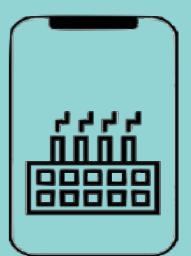
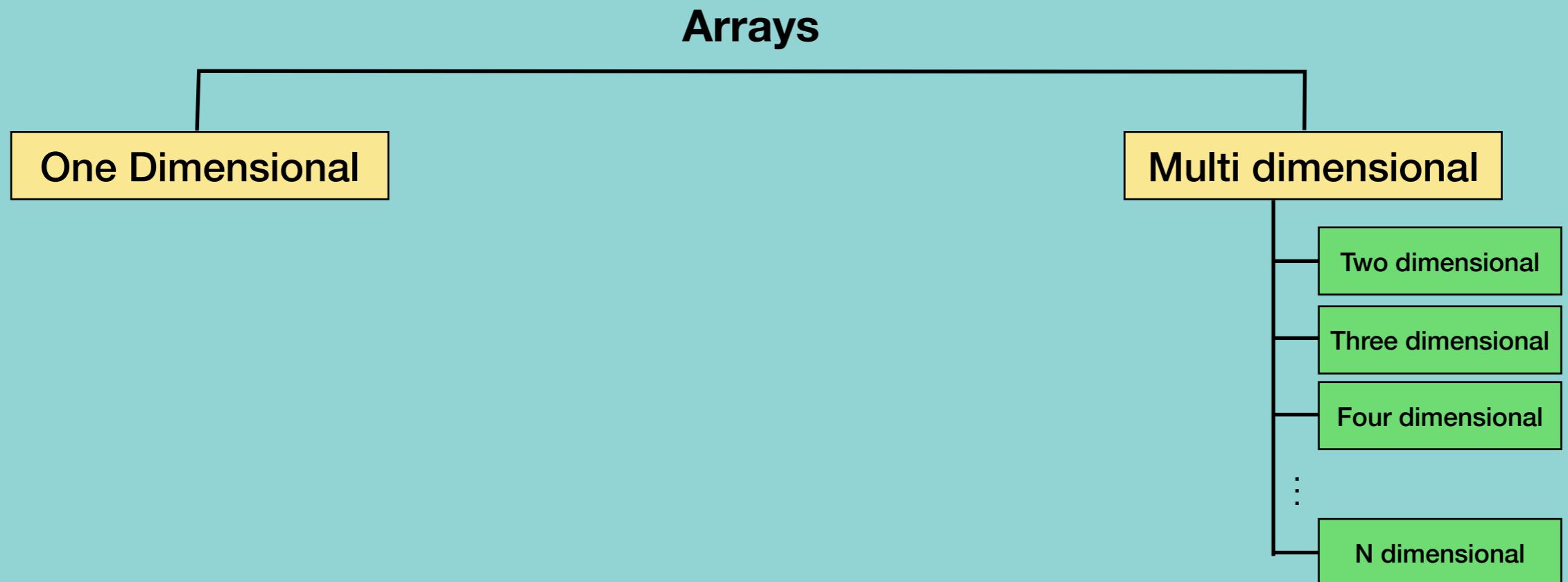
number1  
number2  
number3

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

The answer is an **ARRAY**



# Types of Array

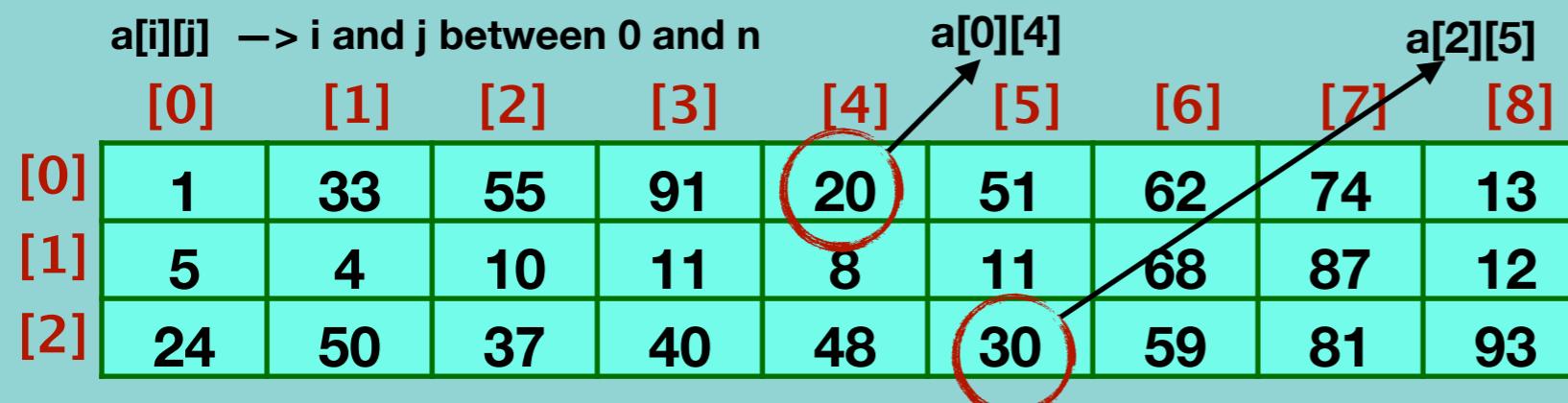


# Types of Array

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

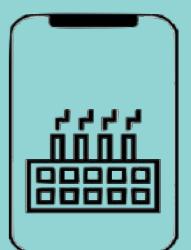
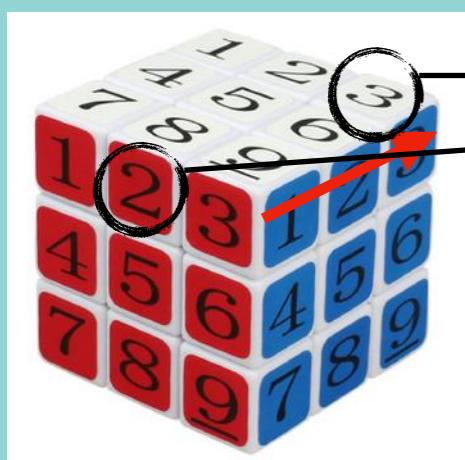


**Two dimensional array :** an array with a bunch of values having been declared with double index.



**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

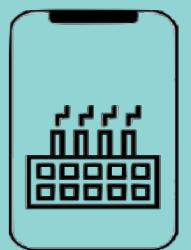
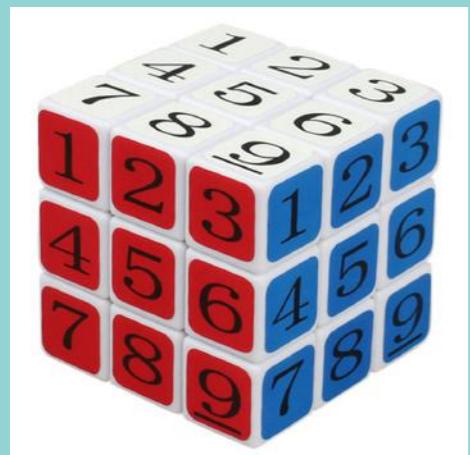
One dimensional array :

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

Two dimensional array

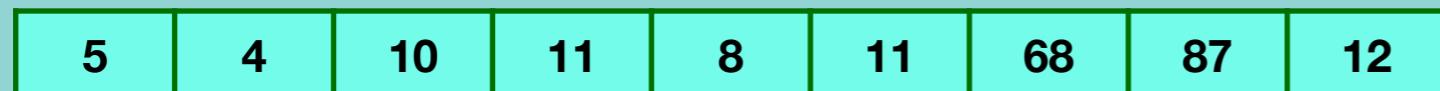
1	33	55	91	20	51	62	74	13
5	4	10	11	8	11	68	87	12
24	50	37	40	48	30	59	81	93

Three dimensional array

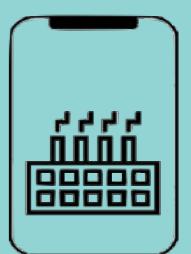
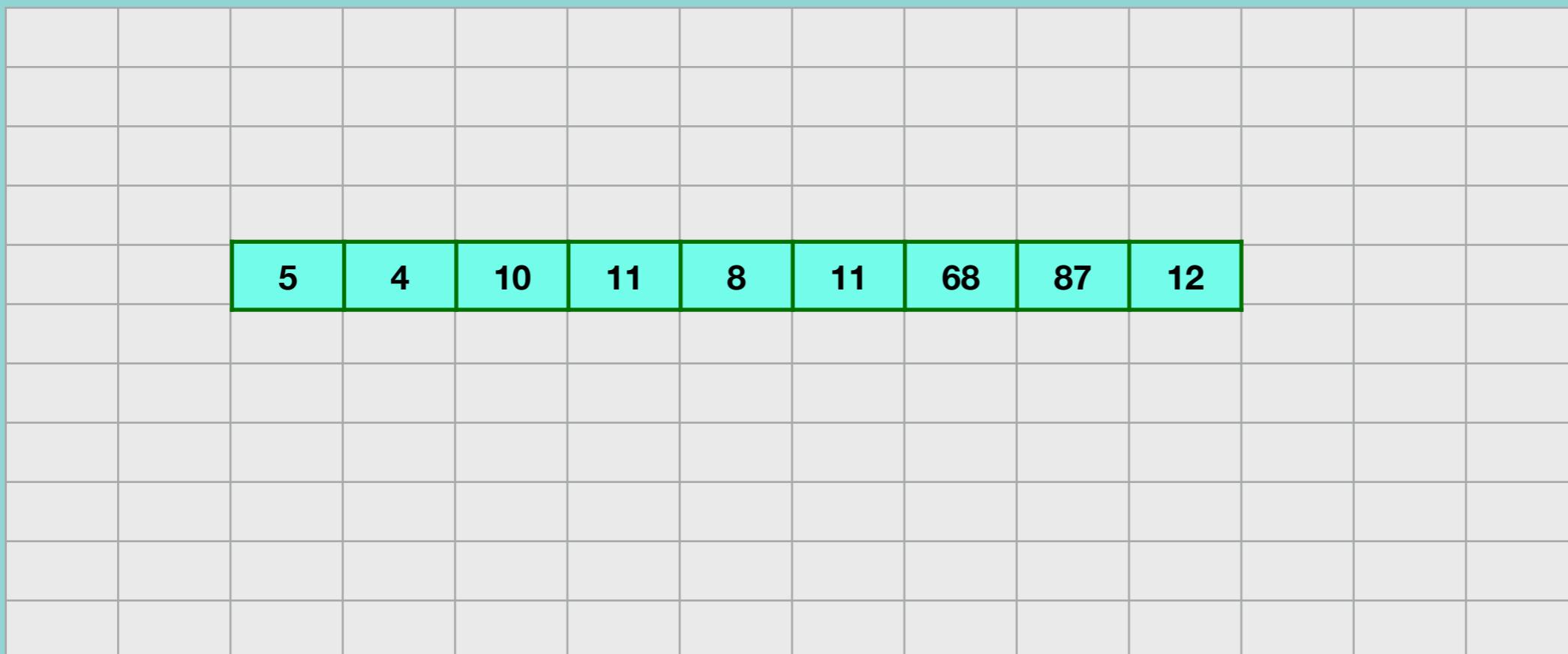


# Arrays in Memory

## One Dimensional



## Memory

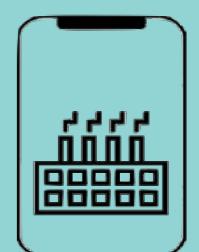
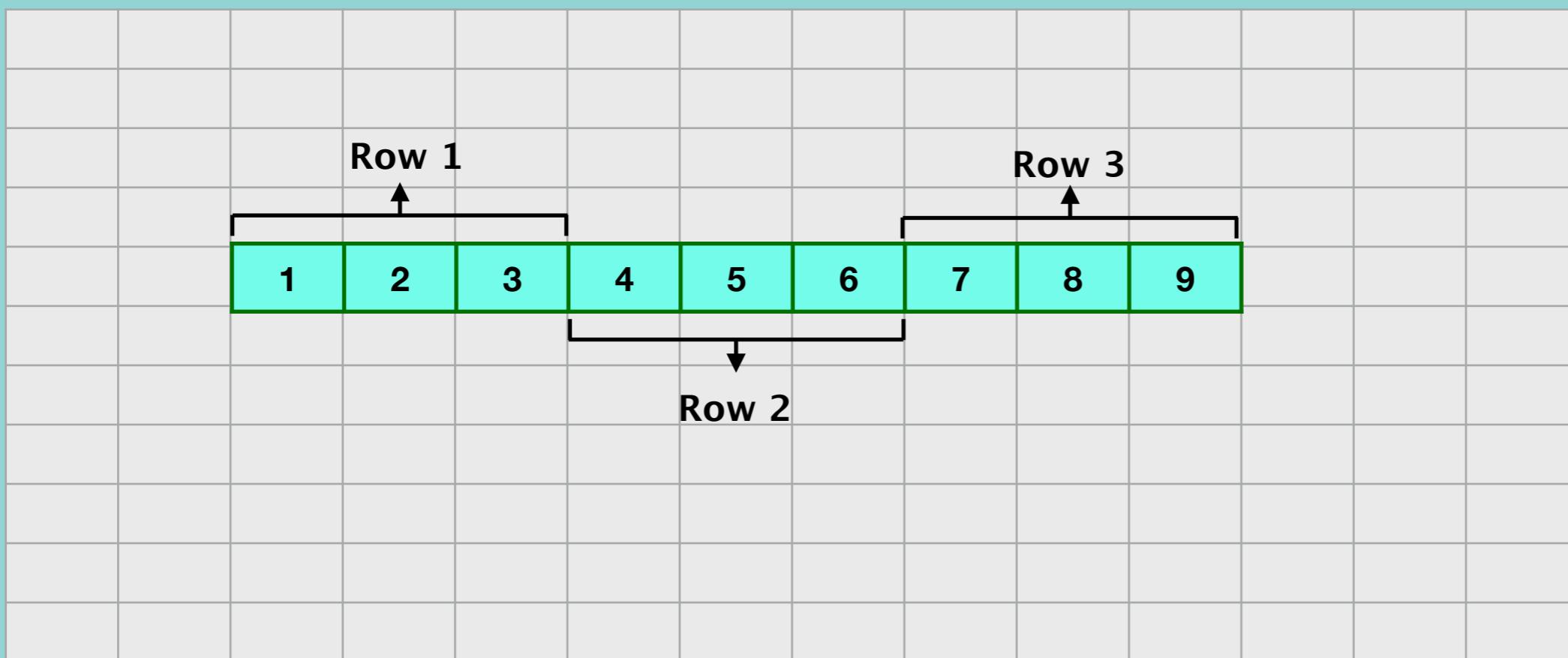


# Arrays in Memory

## Two Dimensional array

1	2	3
4	5	6
7	8	9

## Memory

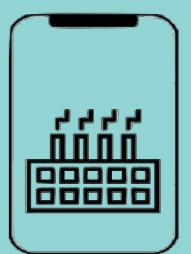
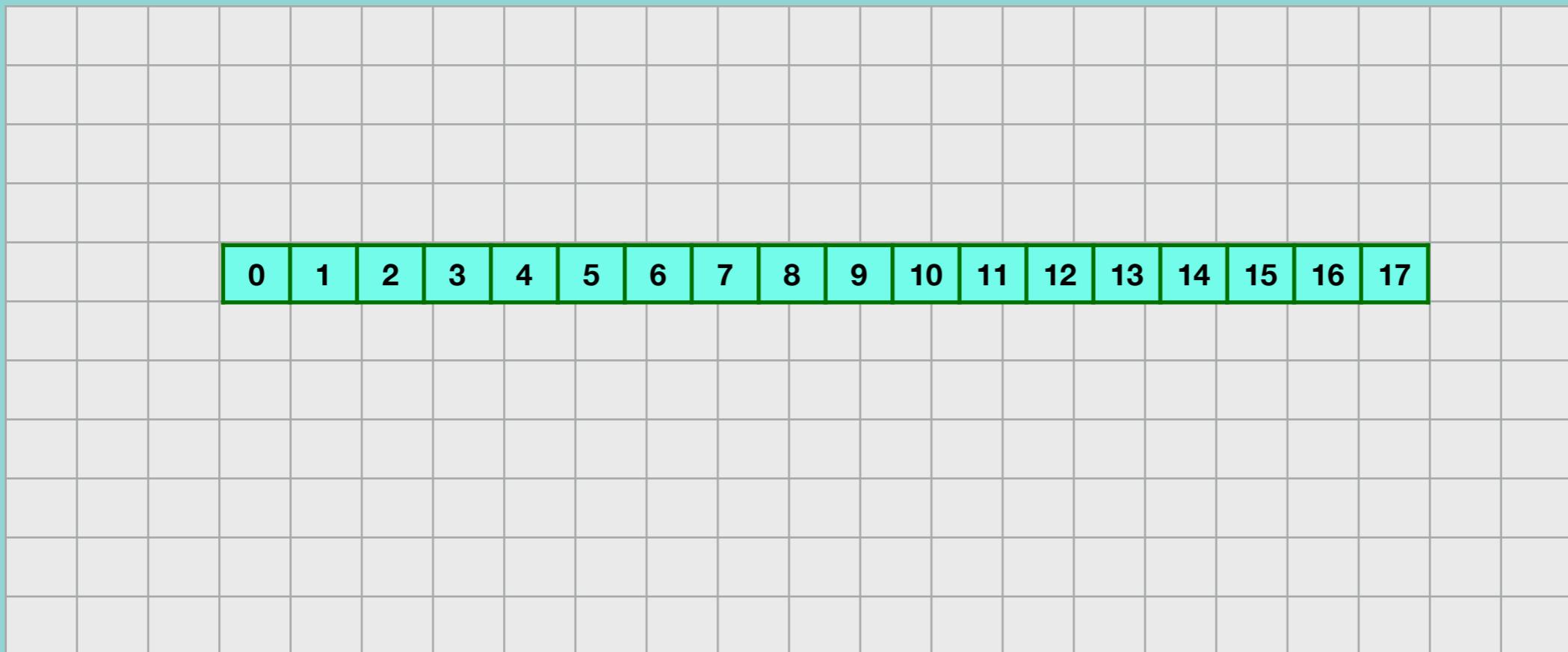


# Arrays in Memory

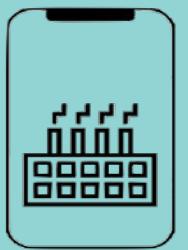
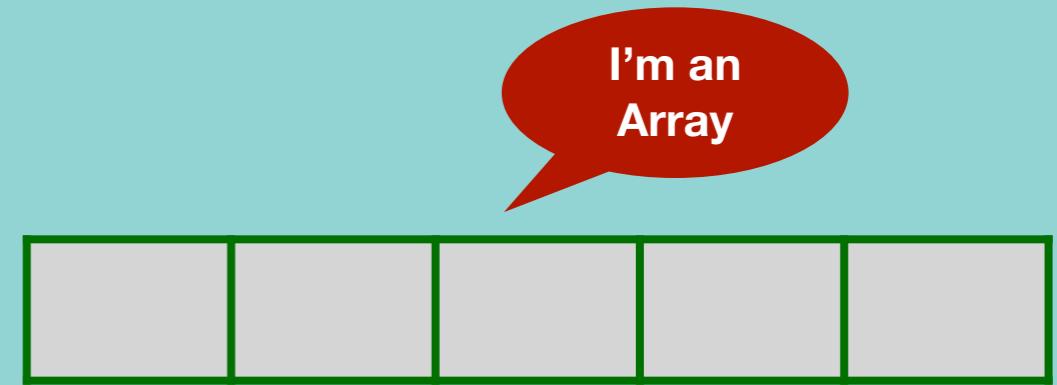
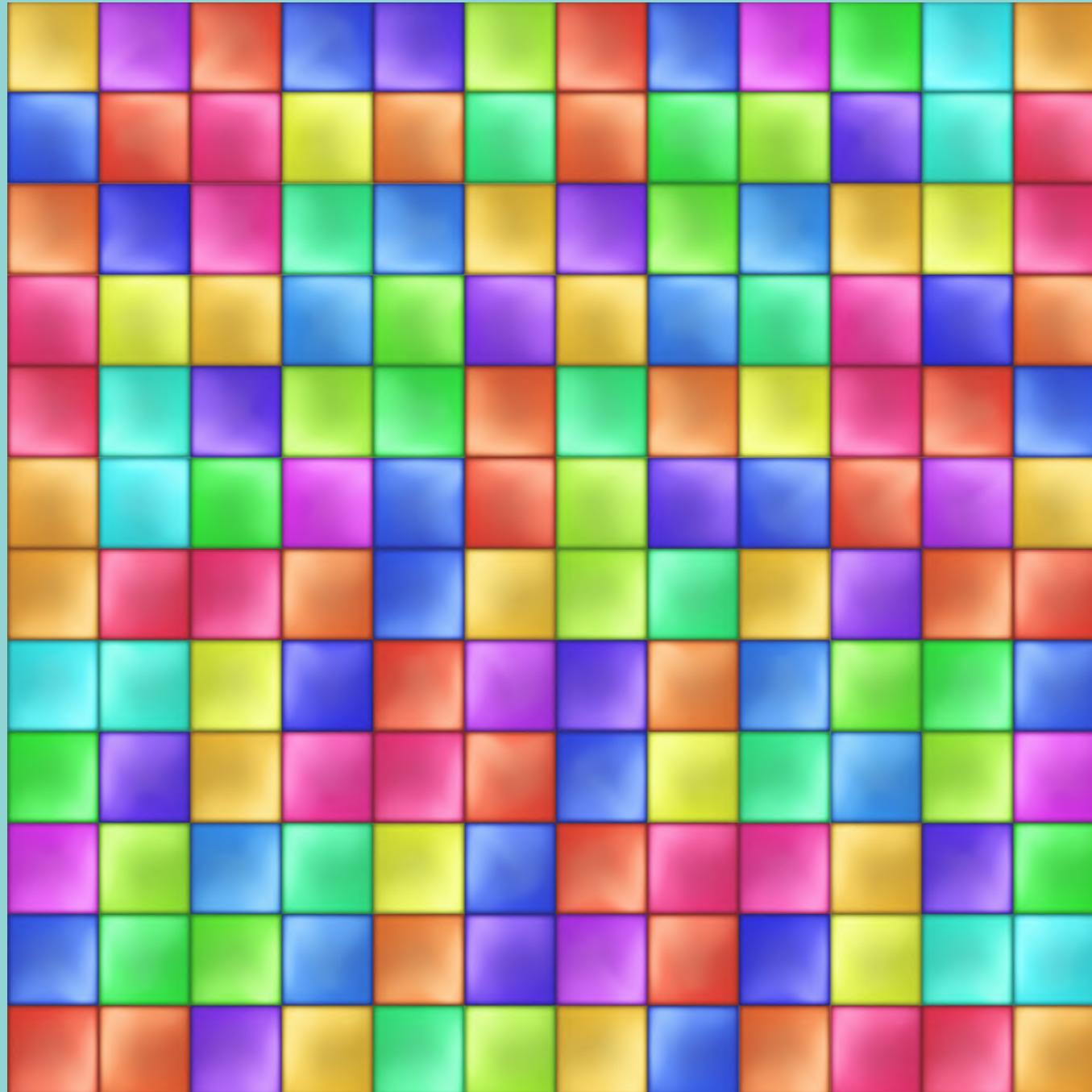
## Three Dimensional array

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

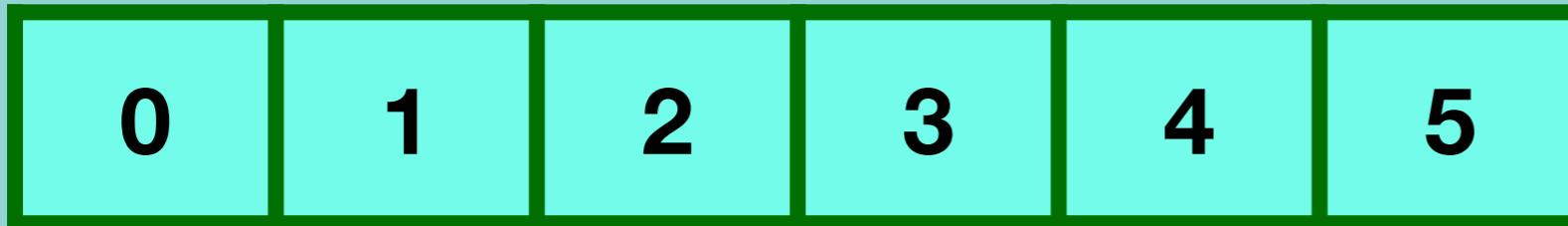
## Memory



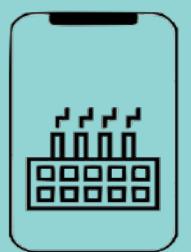
# What is an Array?



# What is an Array?



**Not allowed**

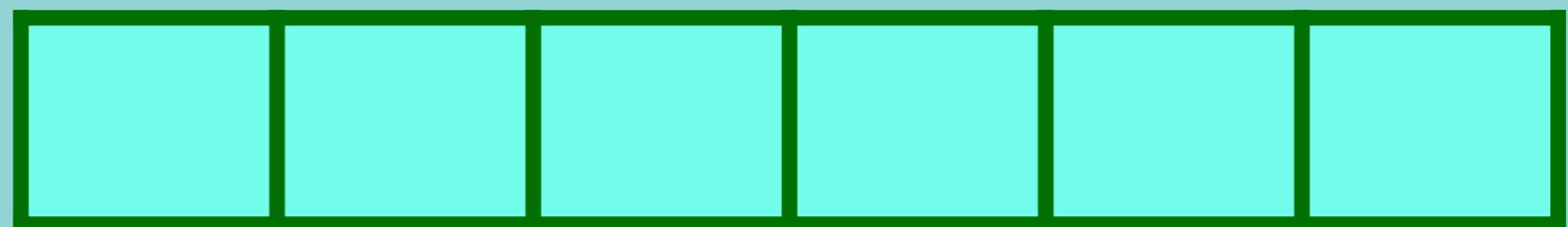


# Creating an array

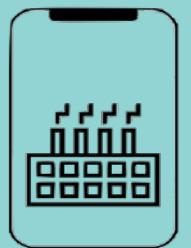
**When we create an array , we:**

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

**myArray =**



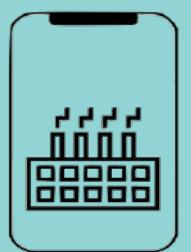
**myArray =**



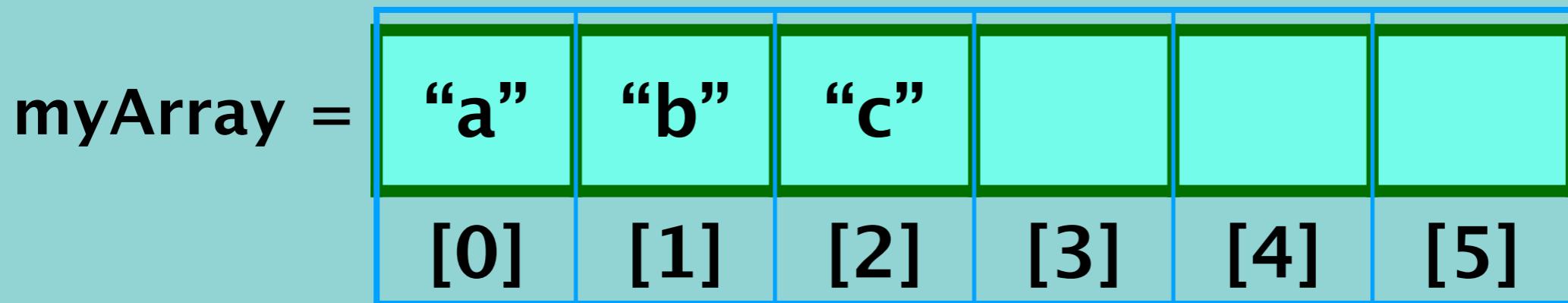
# Creating an array (Array Module)

```
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	

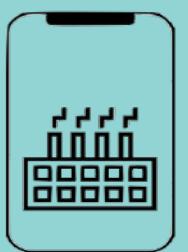


# Insertion

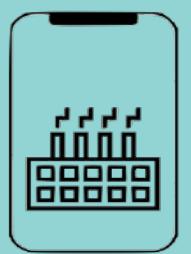
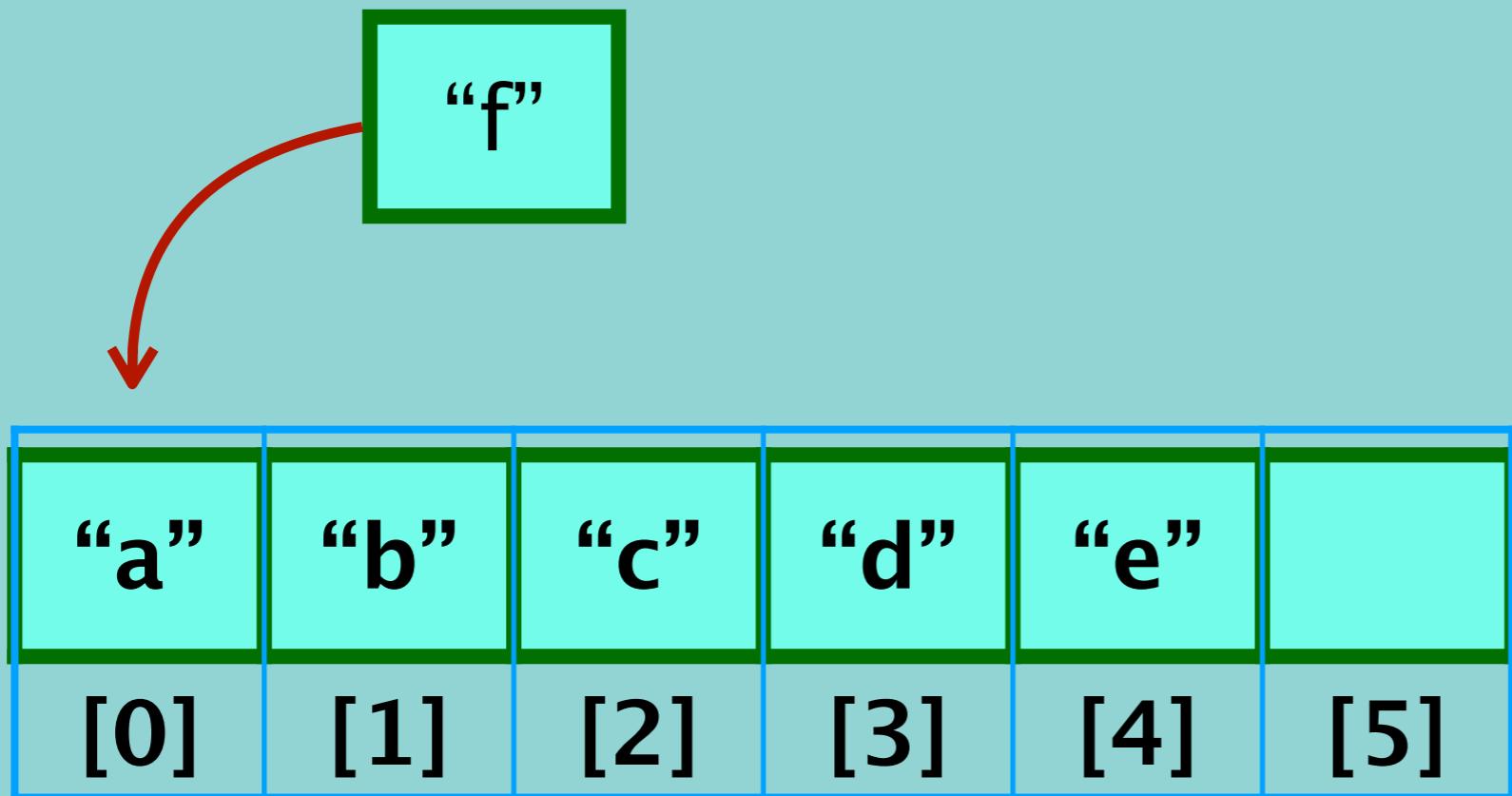


myArray[3] = “d”

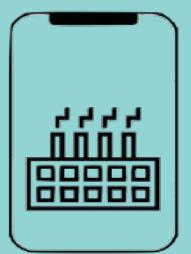
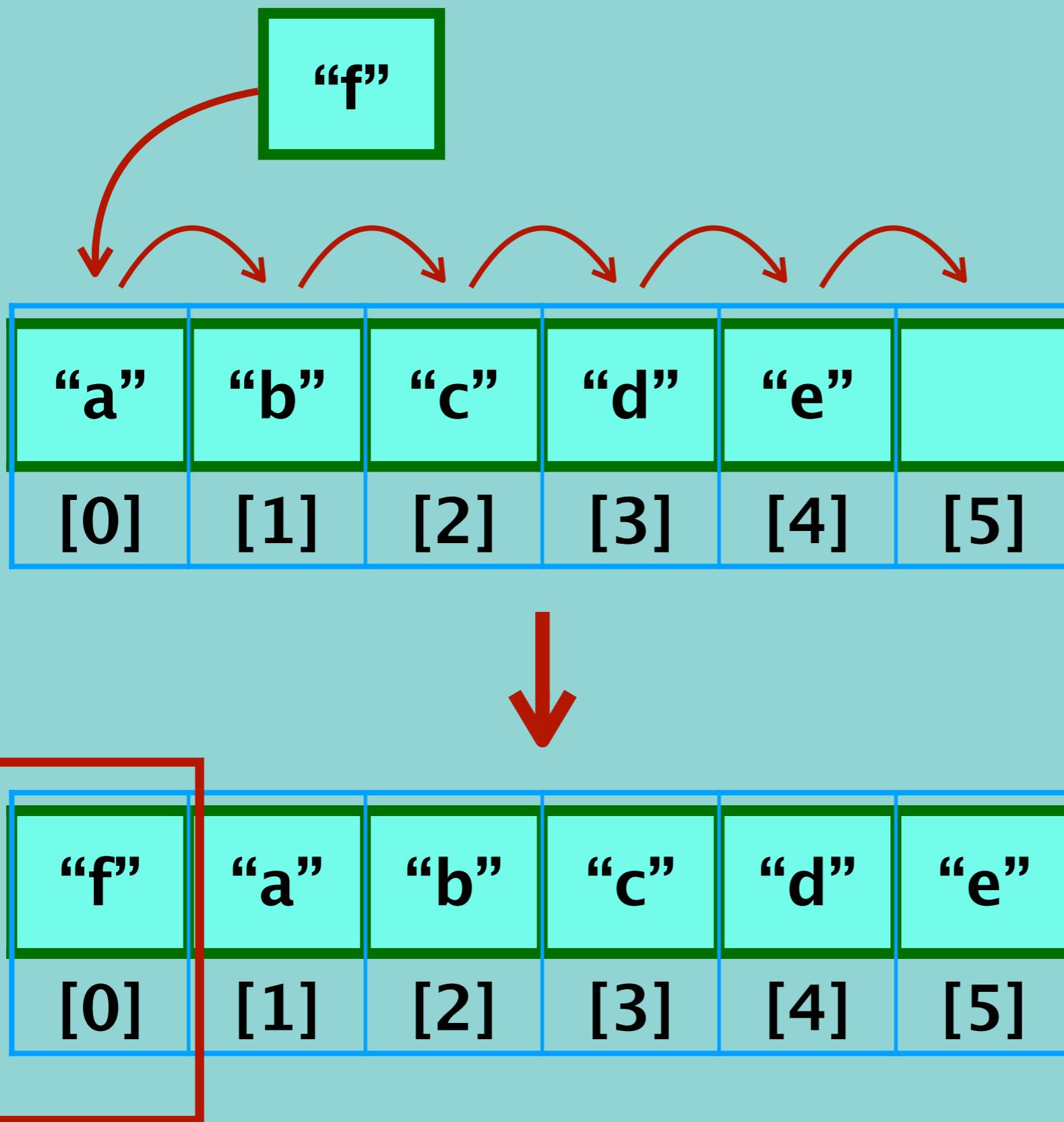
myArray[5] = “f”



# Insertion

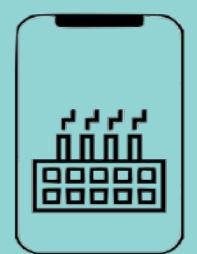
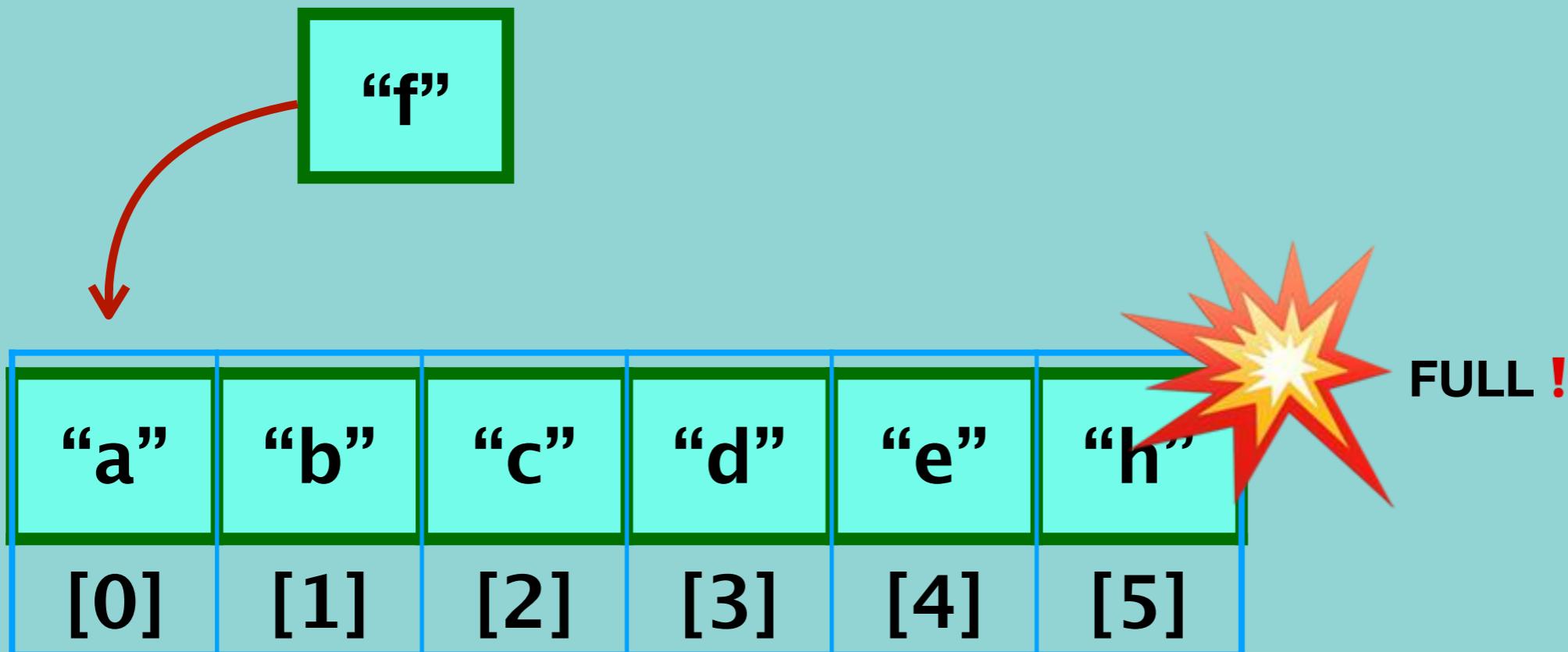


# Insertion





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

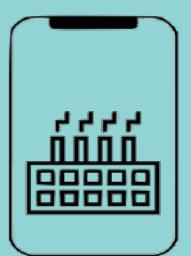


## Insertion , when an array is full.

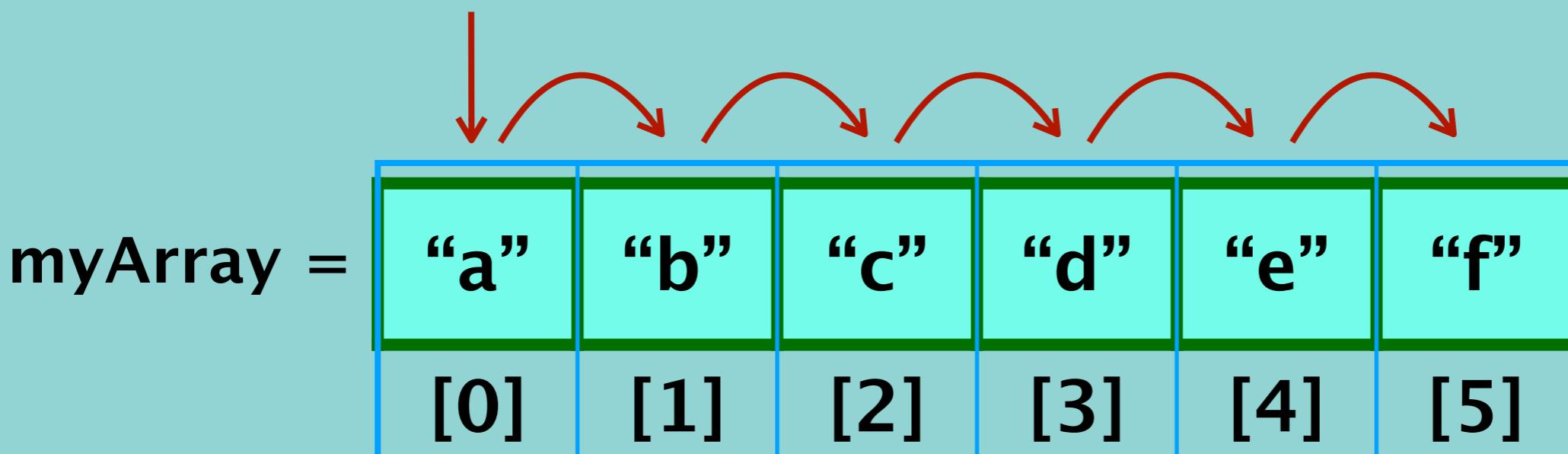
“a”	“b”	“c”	“d”	“e”	“h”
[0]	[1]	[2]	[3]	[4]	[5]



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



## Array traversal



**myArray[0] = "a"**

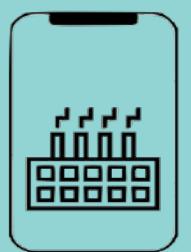
**myArray[1] = "b"**

**myArray[2] = "c"**

**myArray[3] = "d"**

**myArray[4] = "e"**

**myArray[5] = "f"**

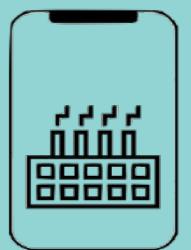


# Array traversal

```
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

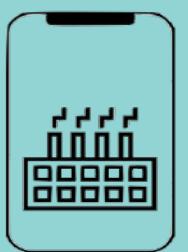
For example:

`myArray =`

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

`myArray[0] = “a”`

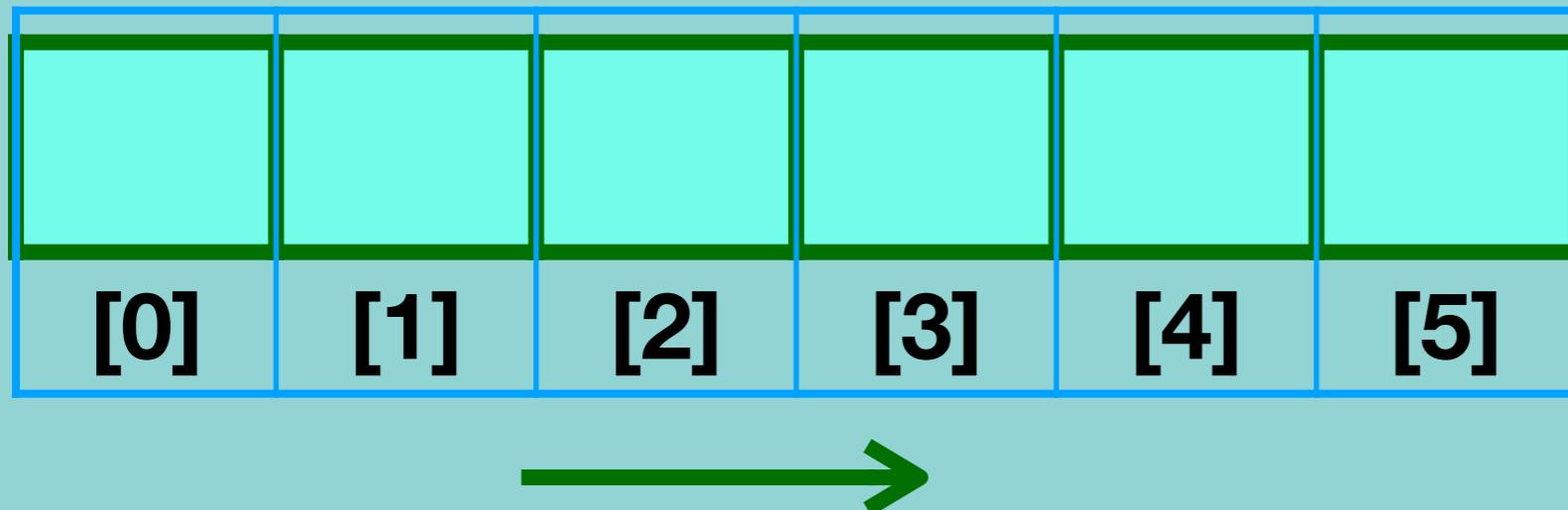
`myArray[3] = “d”`



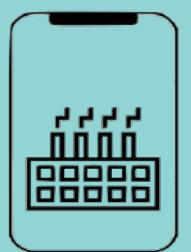
# Access array element

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

## INDEX



<arrayName>[index]

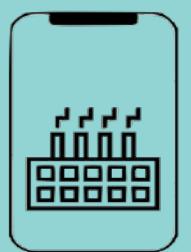


# Access an element of array

```
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)**

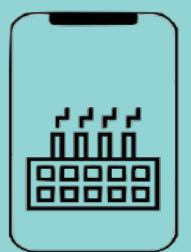
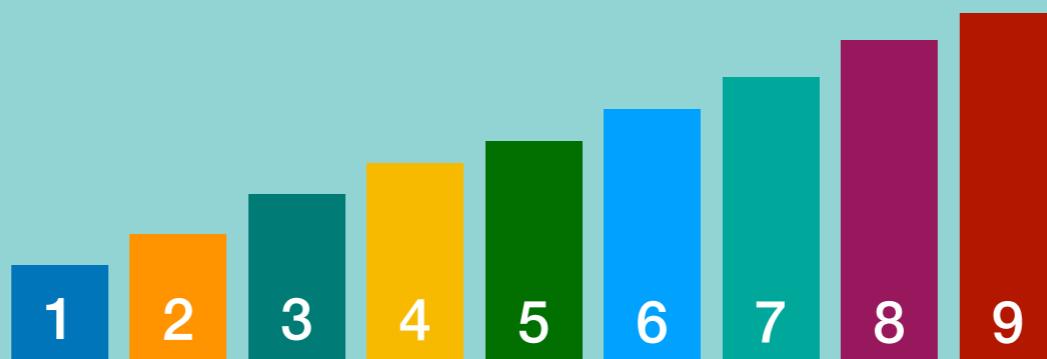


# Finding an element

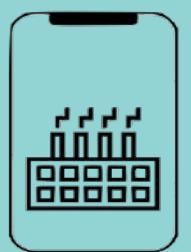
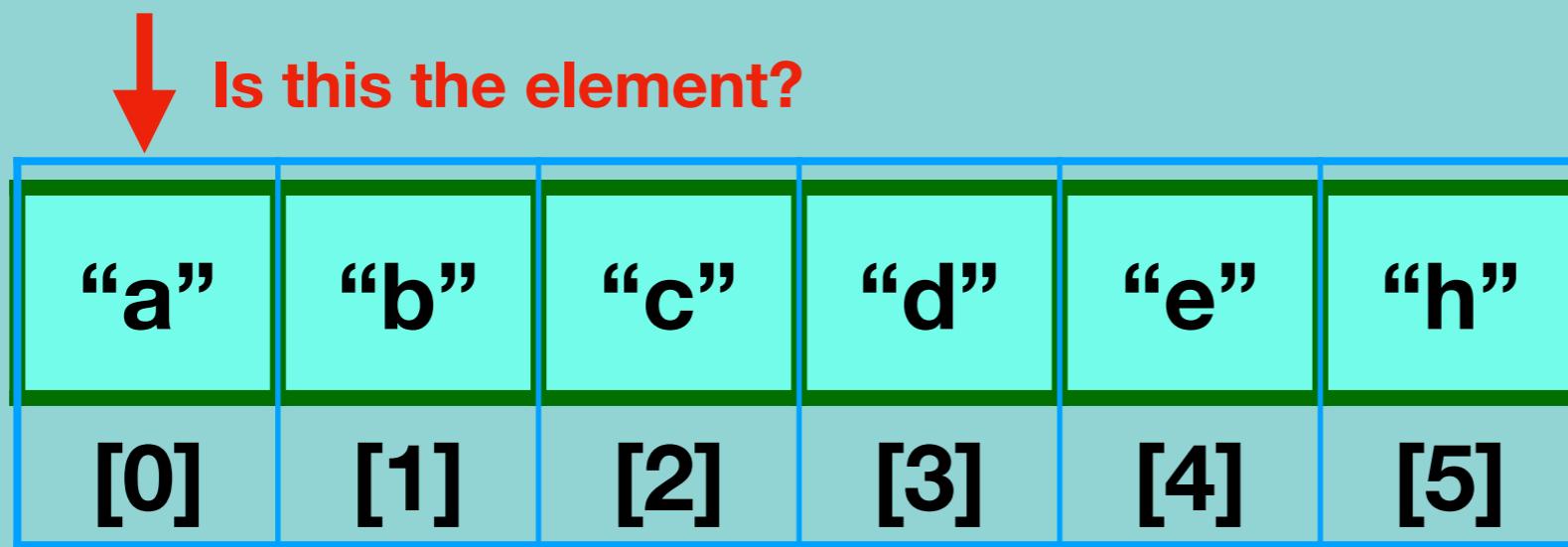
myArray =

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

myArray[2]



# Finding an element



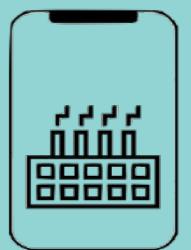
# Finding an element

```
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)
```

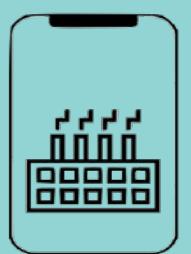


## Deletion

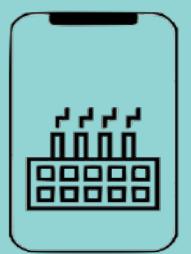
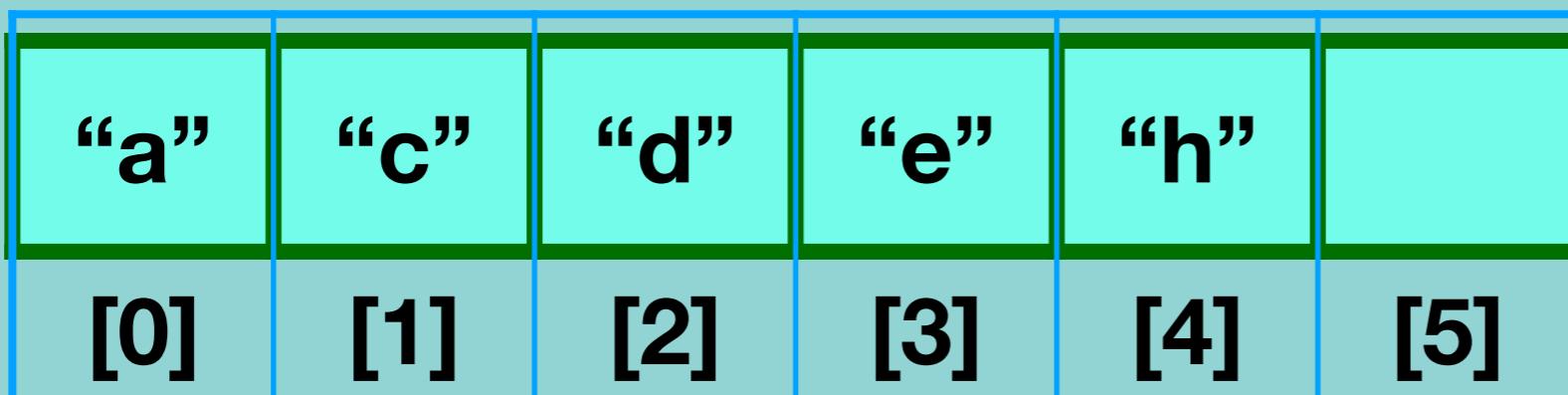
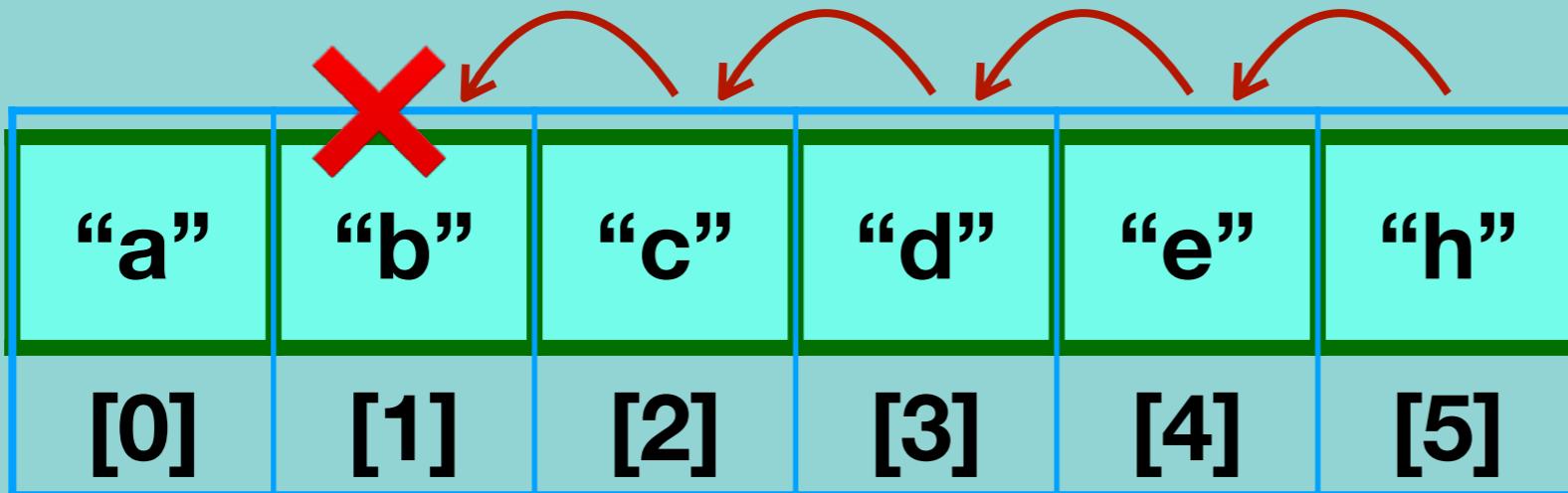
“a”	“b”	“c”	“d”	“e”	“h”
[0]	[1]	[2]	[3]	[4]	[5]

Not allowed!!

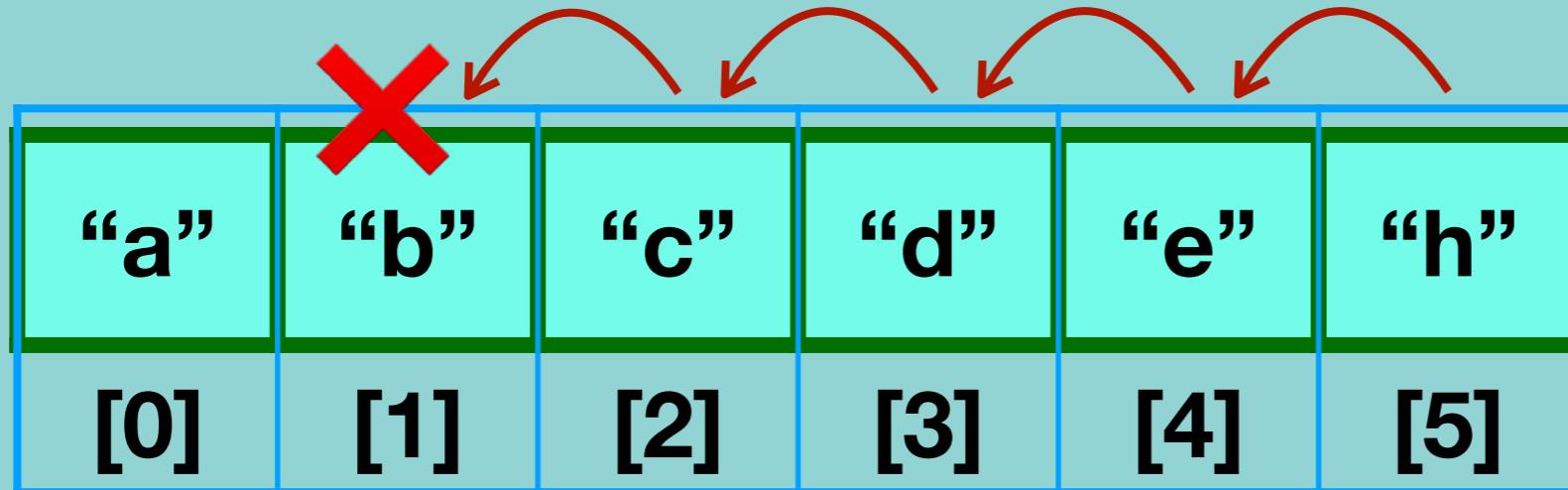
“a”		“c”	“d”	“e”	“h”
[0]	[1]	[2]	[3]	[4]	[5]



## Deletion

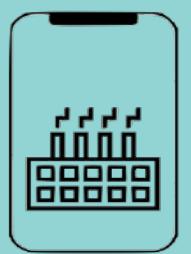


## Deletion

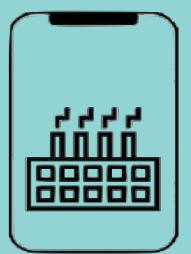


Time Complexity :  $O(n)$

Space Complexity :  $O(1)$



# Inserting a value to two dimensional array

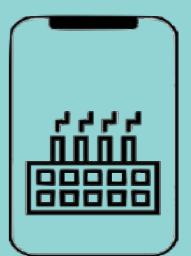
# Array type codes in Python

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	

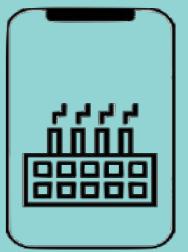
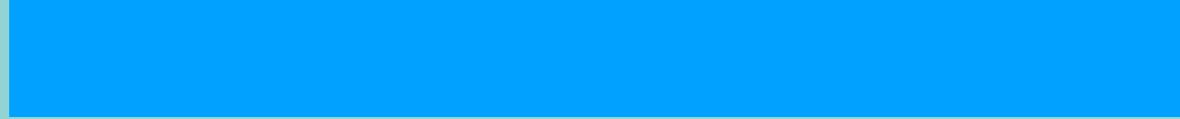
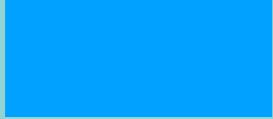
The 'u' type code corresponds to Python's obsolete unicode character ([Py\\_UNICODE](#) which is `wchar_t`). Depending on the platform, it can be 16 bits or 32 bits.

'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.

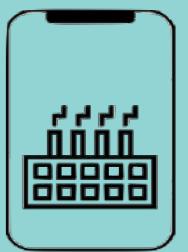


# Built in functions of arrays in Python



# 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.

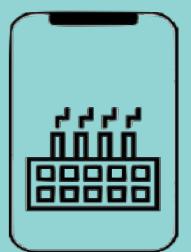
**a[i][j] —> i and j between 0 and n**

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

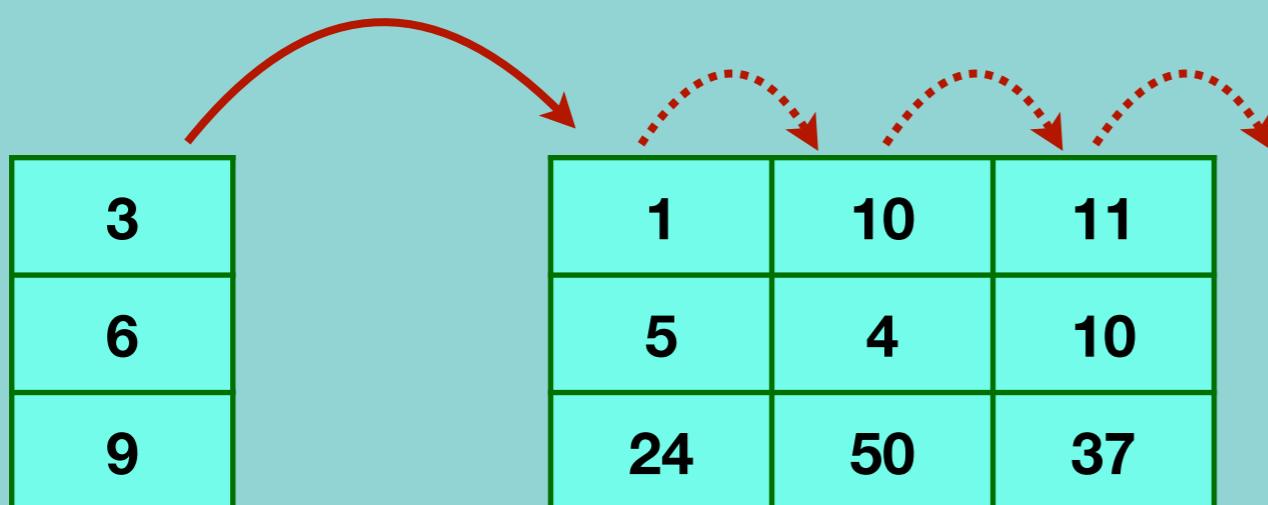
1	10	11
5	4	10
24	50	37

15

?

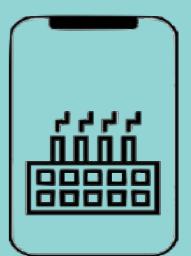
?

Adding Column



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

Time Complexity =  $O(mn)$



# Insertion - Two Dimensional array

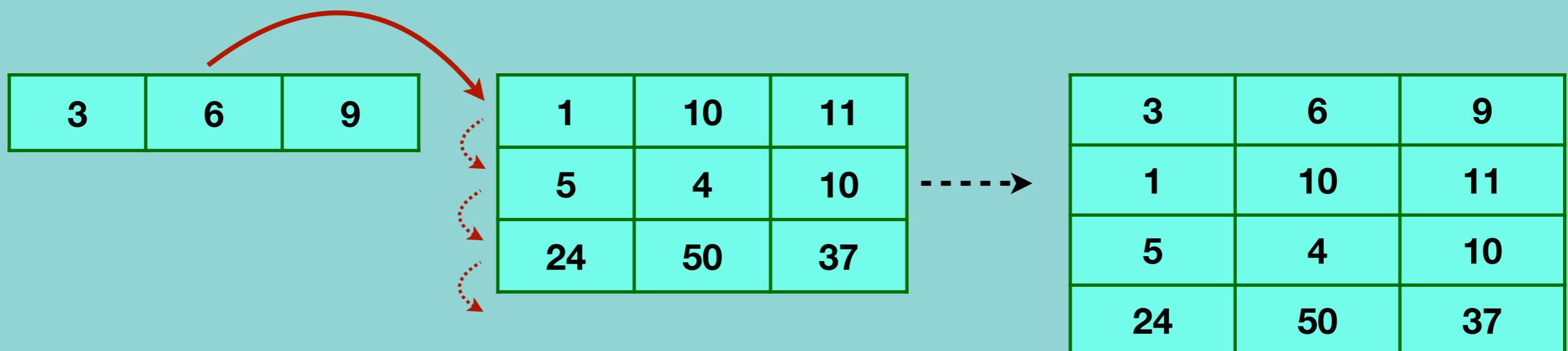
1	10	11
5	4	10
24	50	37

15

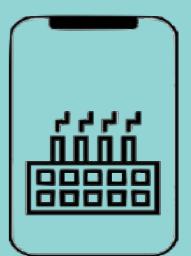
?

?

Adding Row



Time Complexity =  $O(mn)$



# Access an element of Two Dimensional Array

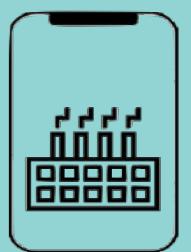
One dimensional array

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



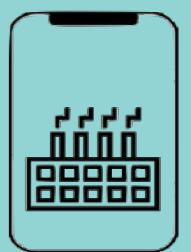
# Traversing Two Dimensional Array

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

1 33 55 91

5 4 10 11

24 50 37 40



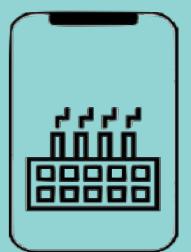
# Traversing Two Dimensional Array



Is this the element?

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

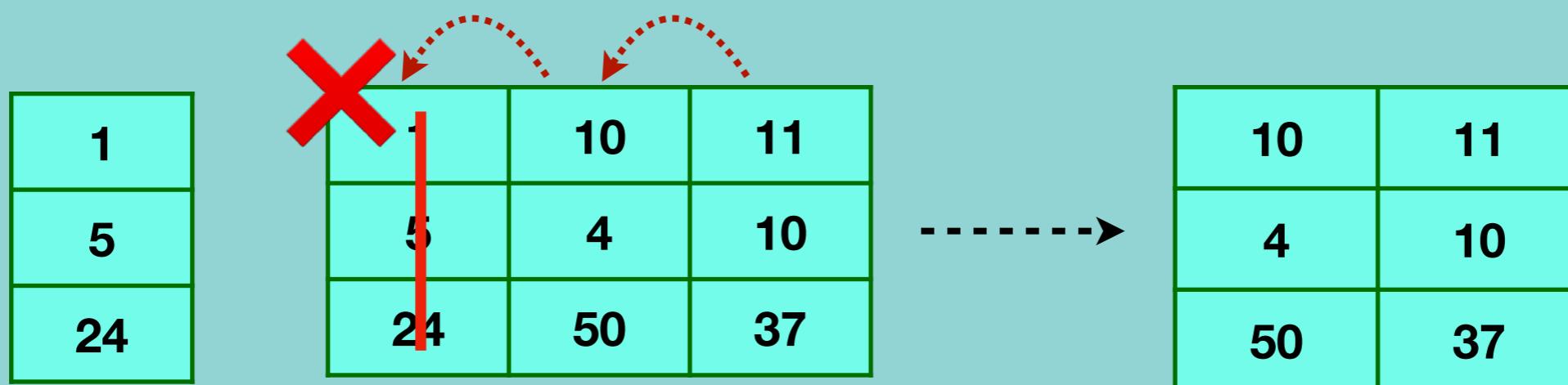
The element is found



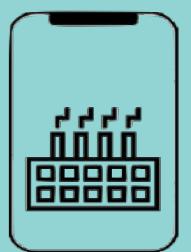
# Deletion - Two Dimensional array

1	10	11
5	4	10
24	50	37

## Deleting Column



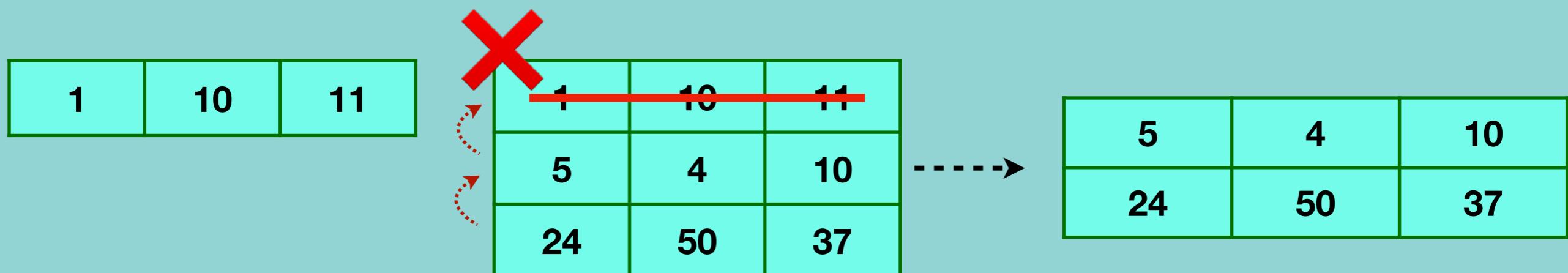
Time Complexity =  $O(mn)$



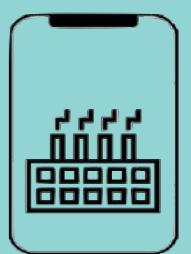
# Deletion - Two Dimensional array

1	10	11
5	4	10
24	50	37

Deleting Row

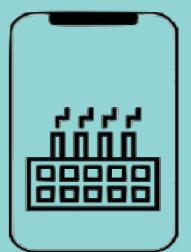


Time Complexity =  $O(mn)$



# 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)



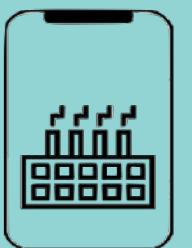
# When to use/avoid Arrays

## When to use

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

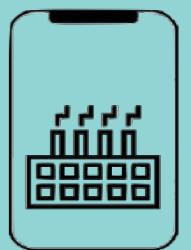
## When to avoid

- 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.

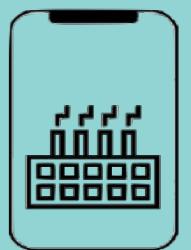


[10, 20, 30, 40] → Integers

['Edy', 'John', 'Jane'] → Strings

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

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



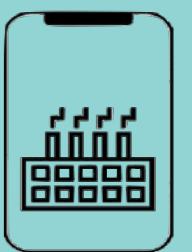
# Arrays vs Lists

## Similarities

[10, 20, 30, 40]

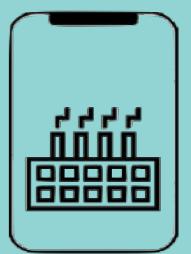
## Differences

[10, 20, 30, 40]



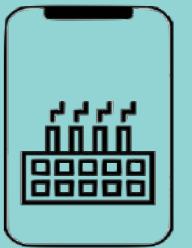
# Time and Space Complexity in Python Lists

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](http://www.appmillers.com)



# Find Number of Days Above Average Temperature

How many day's temperature?

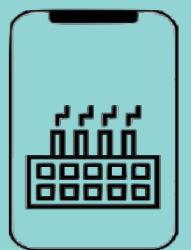
2

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:

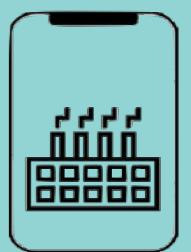
```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Output: Because nums[0] + nums[1] == 9, we return [0, 1].
```

### Example 2:

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

### Example 3:

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

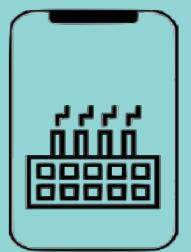


# Array/List Interview Questions - 2

*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?

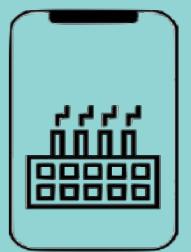


# Array/List Interview Questions -7

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

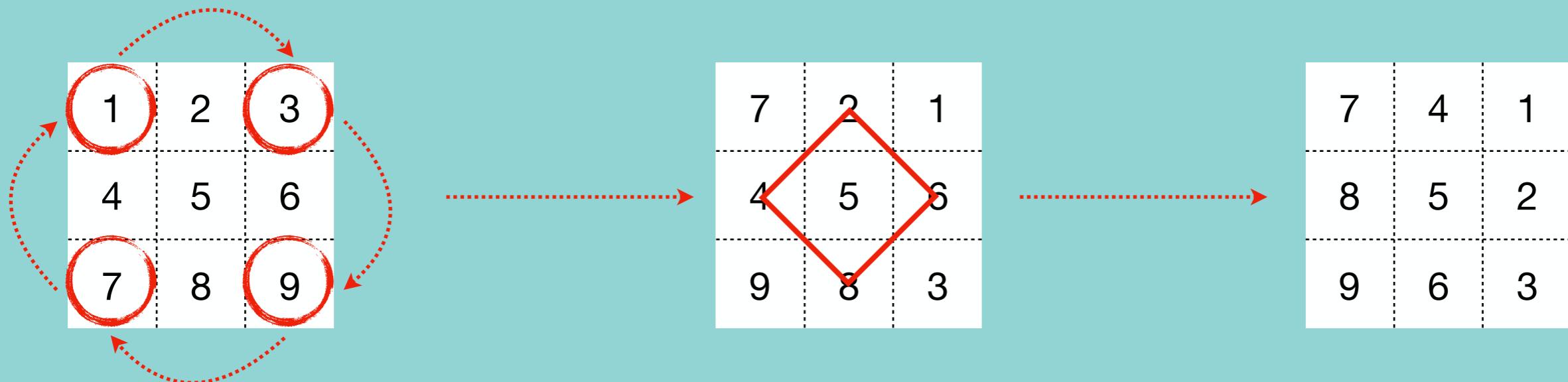
1	2	3
4	5	6
7	8	9

Rotate 90 degrees 

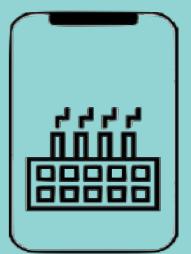


# Array/List Interview Questions -7

**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

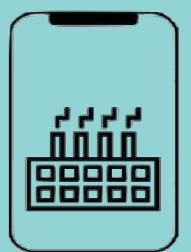


```
myDict = {"Miller": "a person who owns or works in a corn mill",
          "Programmer": "a person who writes computer programs"}
```

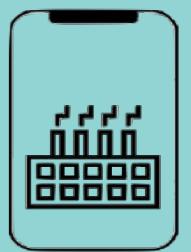
	0	1	2
myArray =	"Miller"	"Programmer"	"App Miller"

myArray[0] → Miller

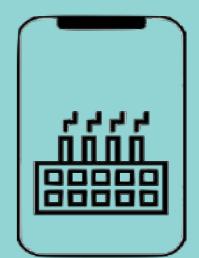
myDict["Miller"] → a person who owns or works in a corn mill



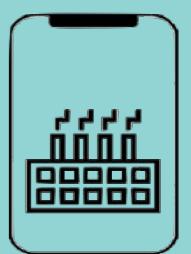
# Dictionaries



# Dictionaries

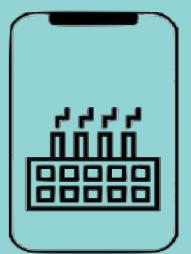
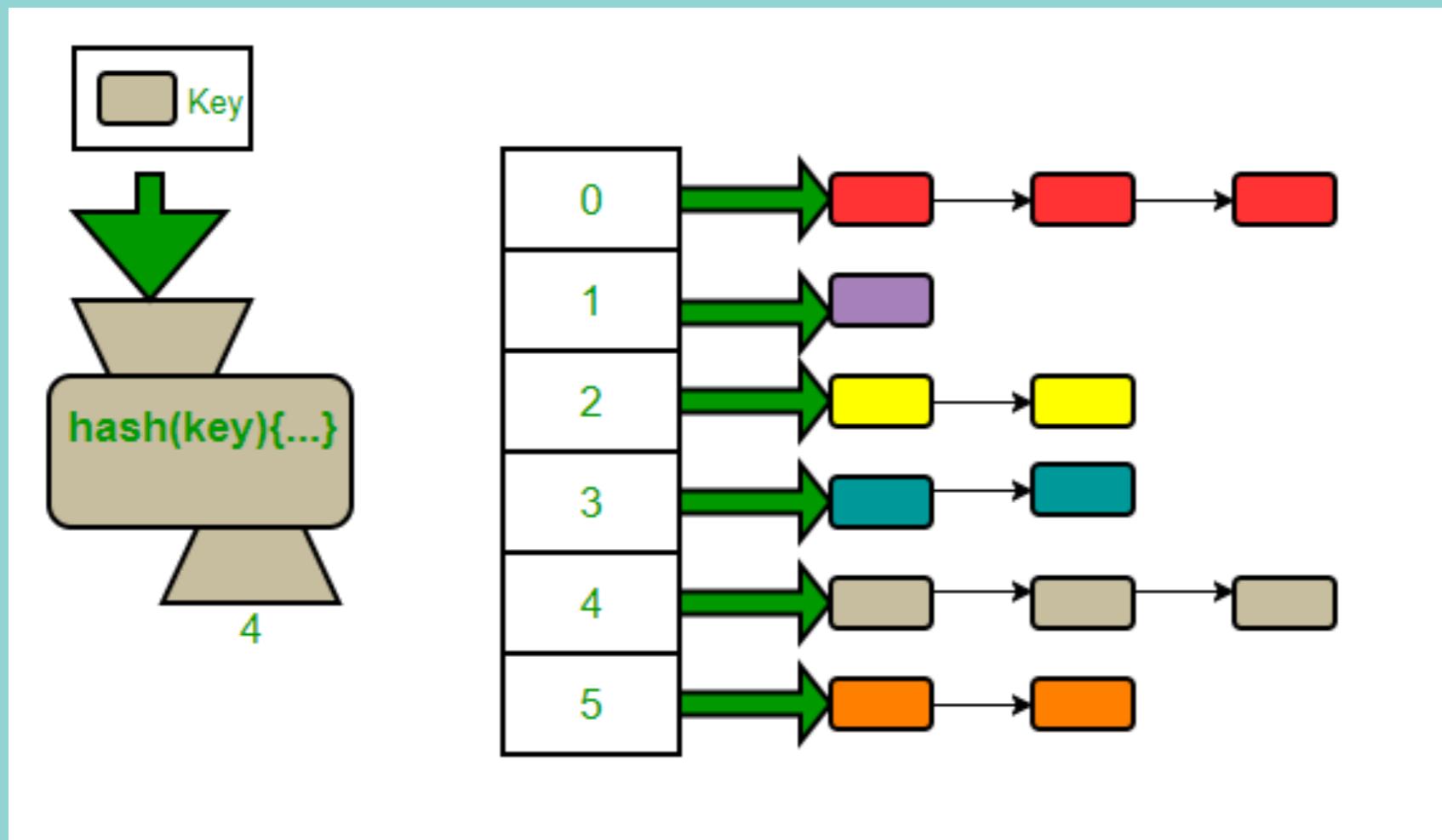


# Dictionaries



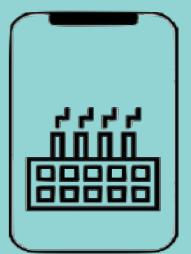
# Dictionary in Memory

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.



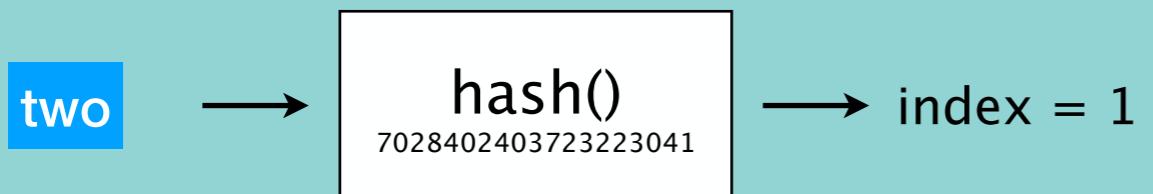
# Dictionary in Memory

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

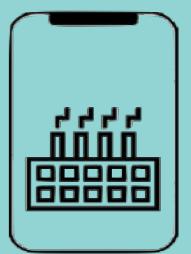


# Dictionary in Memory

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

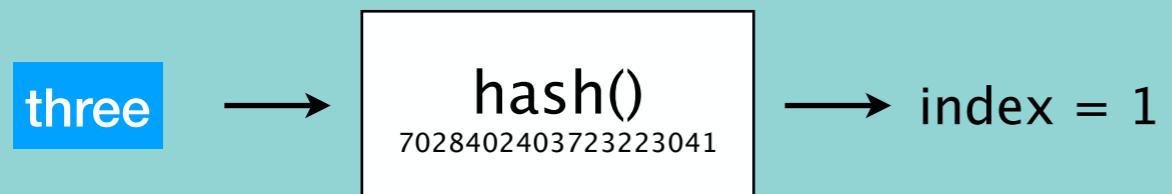


0	
1	two dos
2	
3	one uno
4	

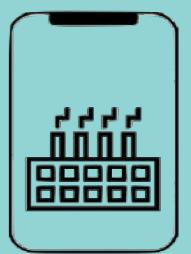


# Dictionary in Memory

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

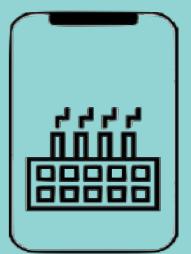
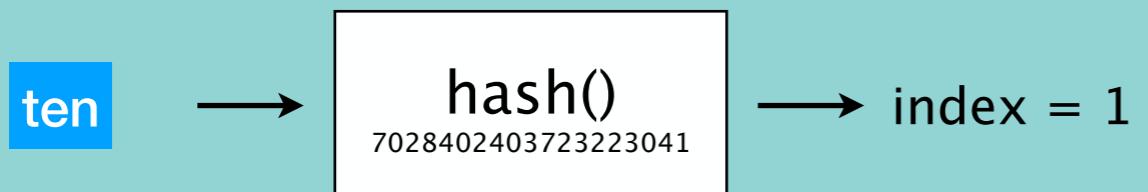


0		
1	two	dos
2		
3	one	uno
4	three	tres



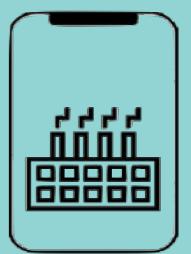
# Dictionary in Memory

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



# Dictionary all method

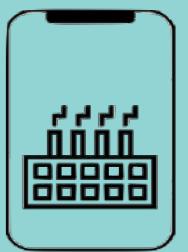
Cases	Return Value
All values are true	TRUE
All values are false	FALSE
One value is true (others are false)	FALSE
One value is false (others are true)	FALSE
Empty Iterable	TRUE



# Dictionary any method

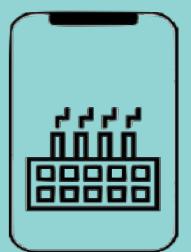
Cases	Return Value
All values are true	TRUE
All values are false	FALSE
One value is true (others are false)	TRUE
One value is false (others are true)	TRUE
Empty Iterable	FALSE

•



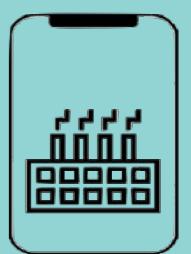
# Dictionary vs List

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



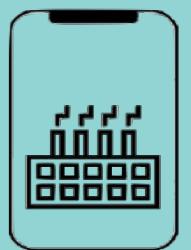
# Time and Space Complexity in Python Dictionary

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**



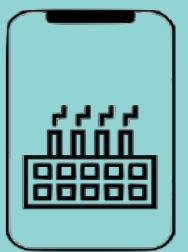
# What is a Tuple?

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')
```

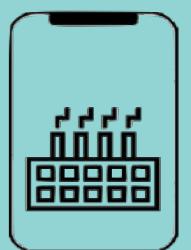
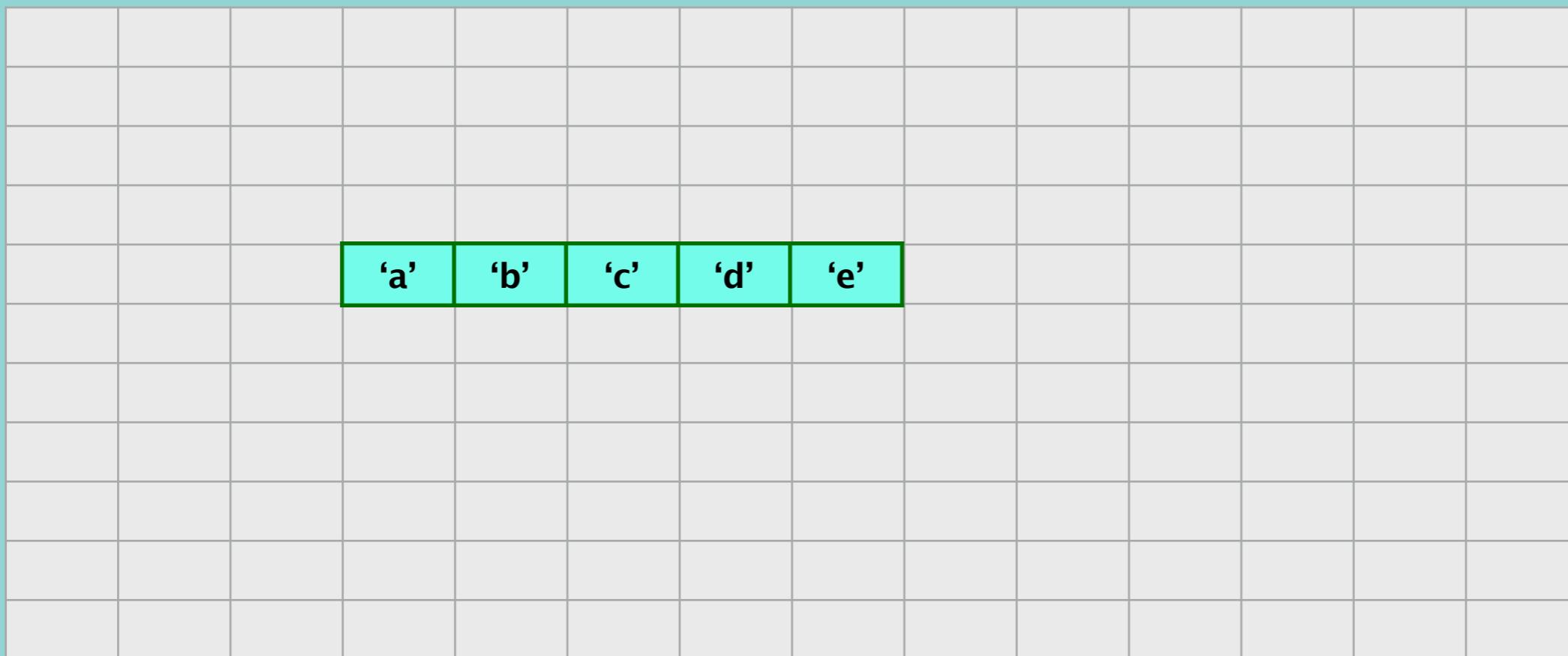


# Tuples in Memory

# Sample Tuple

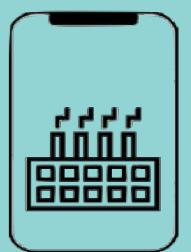
'a'	'b'	'c'	'd'	'e'
-----	-----	-----	-----	-----

## Memory



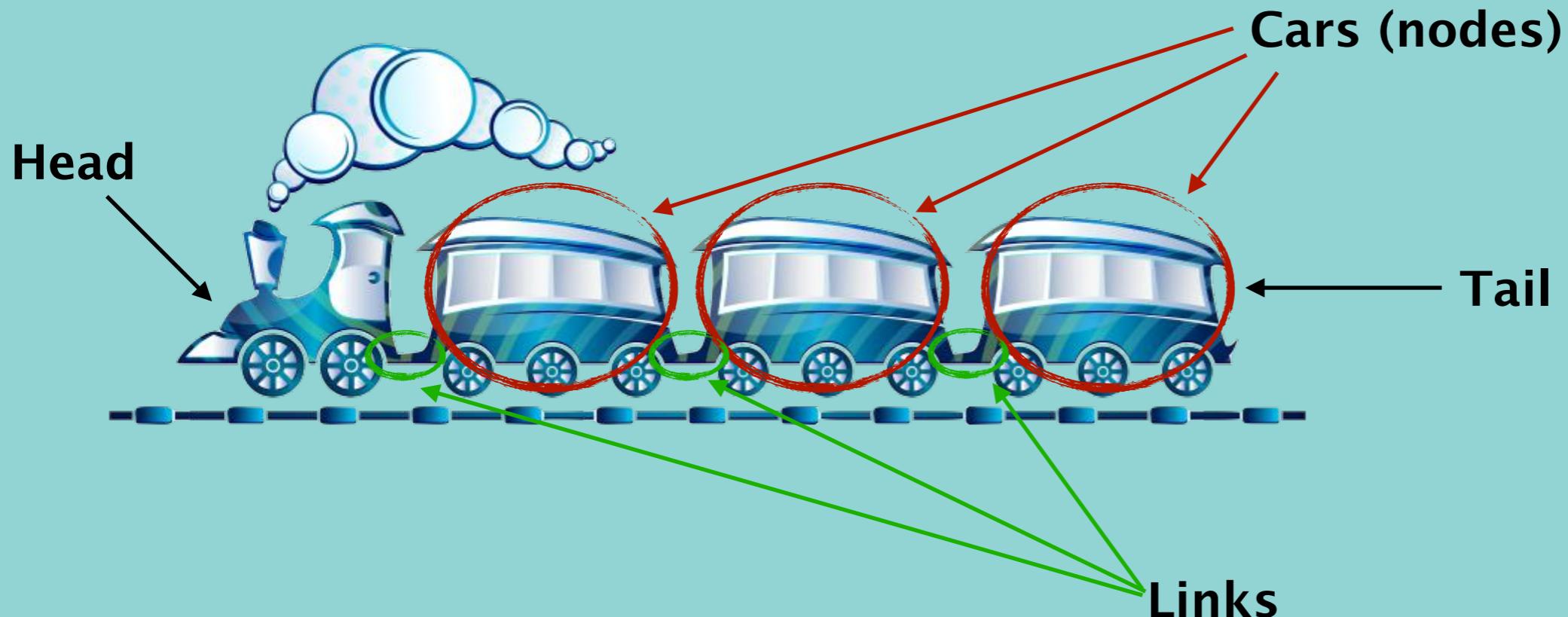
# Time and Space Complexity in Python Tuples

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)

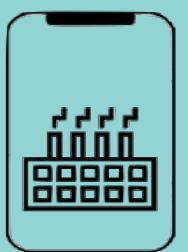


# What is a Linked List?

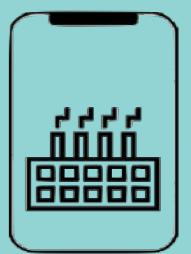
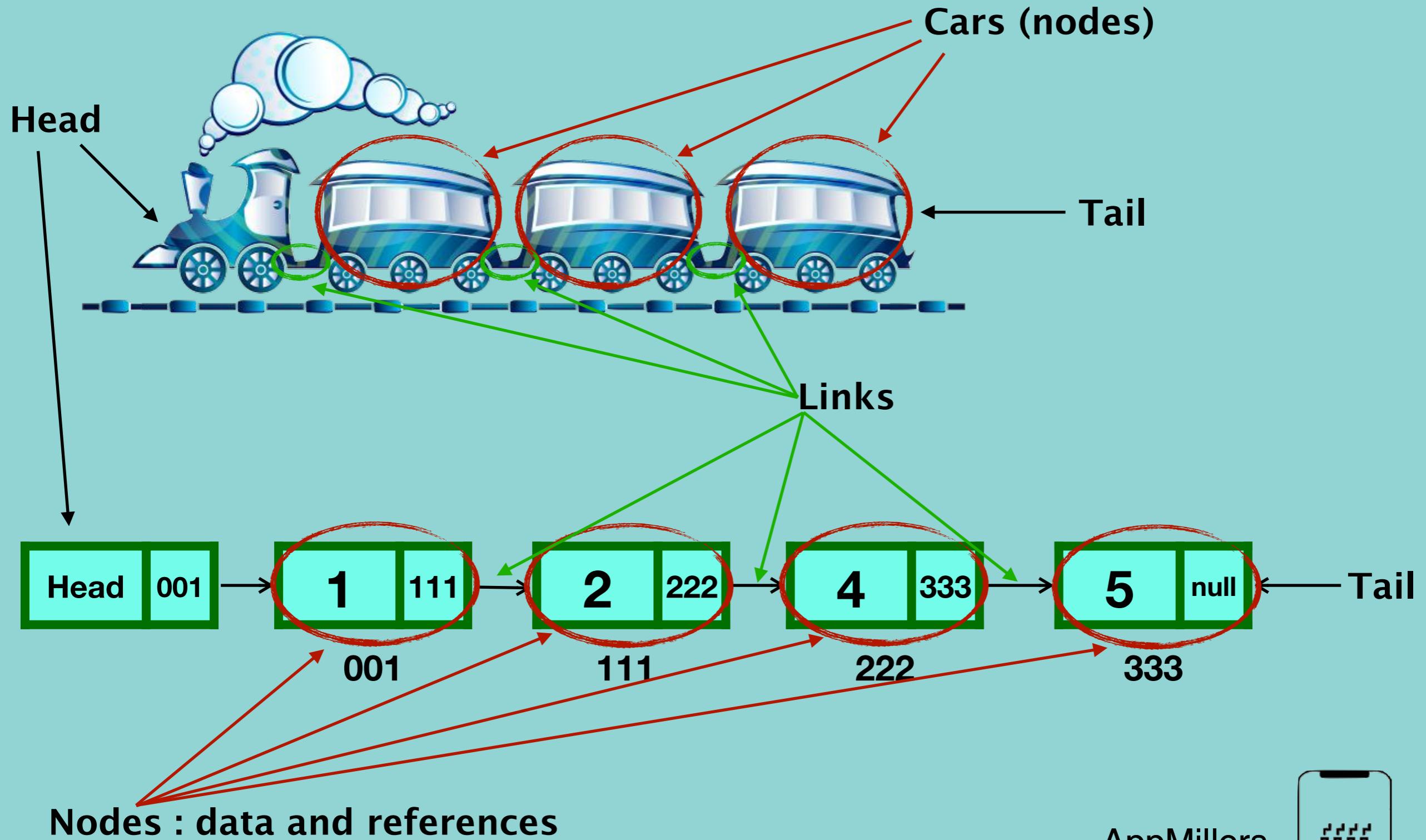
Linked List is a form of a sequential collection and it does not have to be in order. A Linked list is made up of independent nodes that may contain any type of data and each node has a reference to the next node in the link.



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

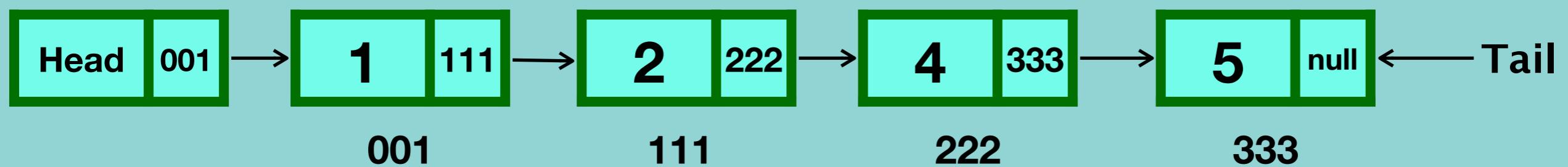
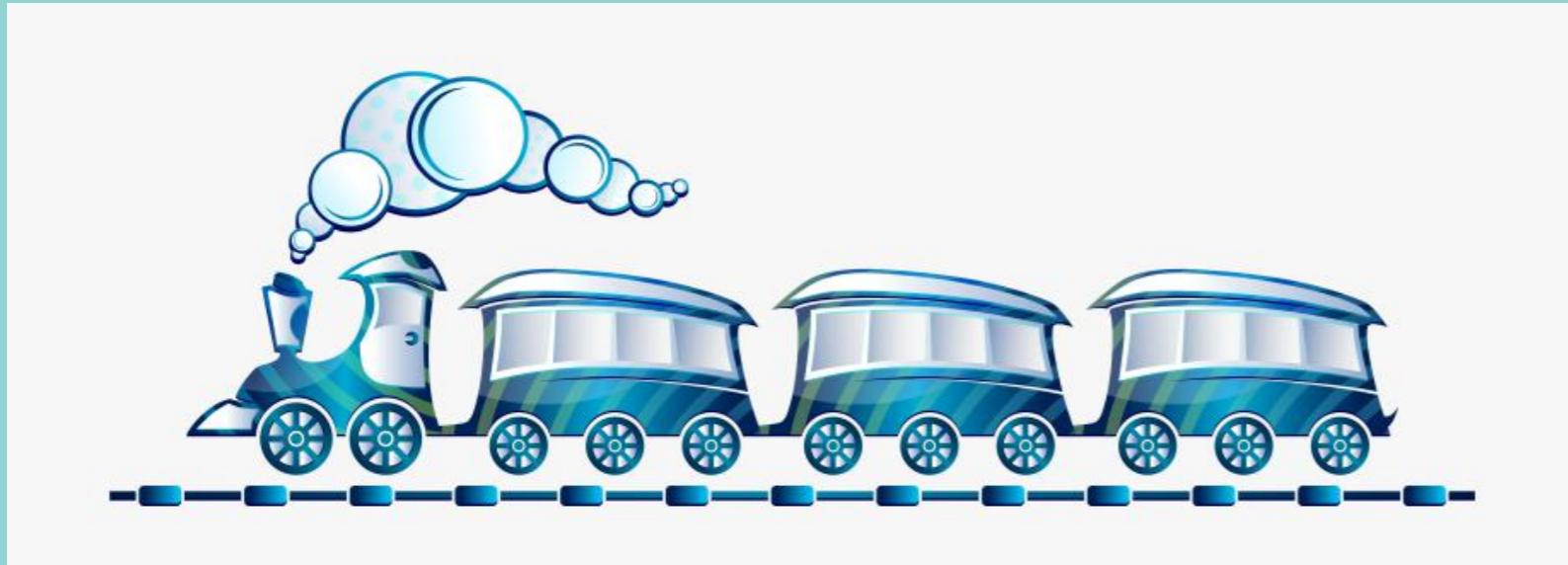


# What is a Linked List?



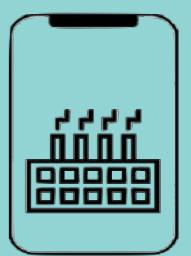
# What is a Linked List?

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.

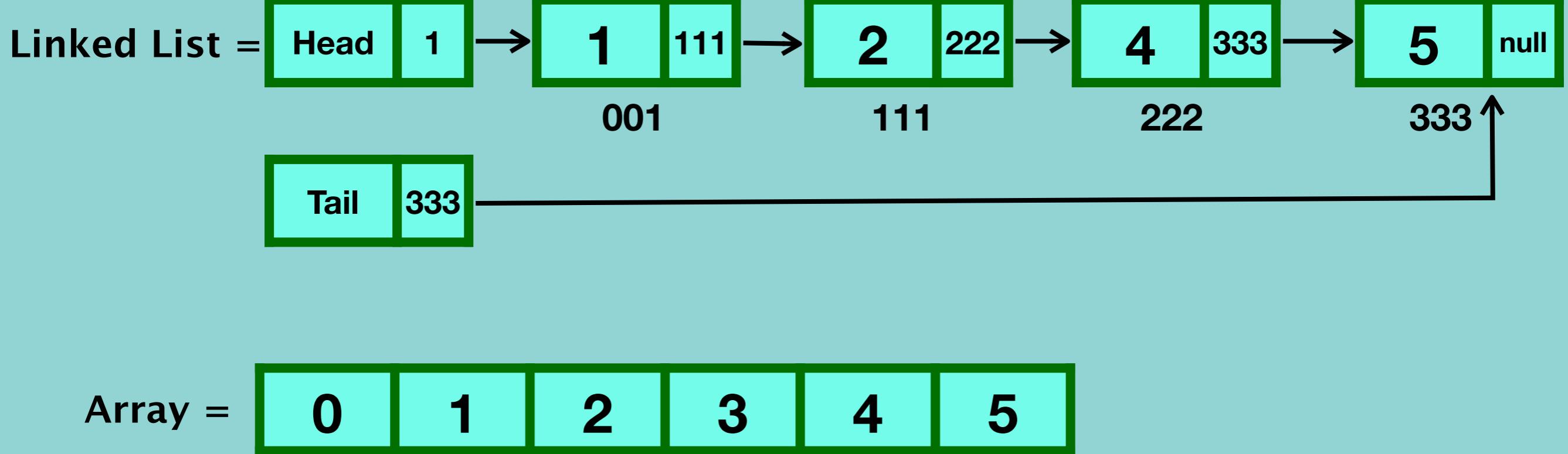


Nodes

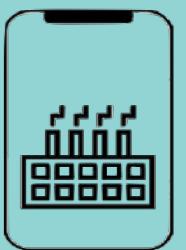
Pointers



# 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

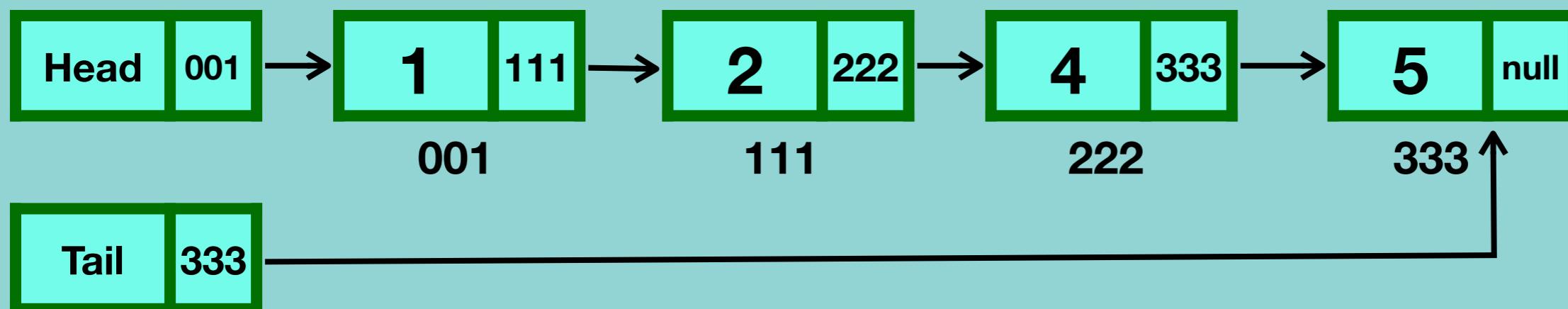


# Types of Linked Lists

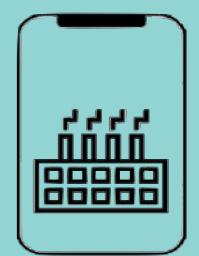
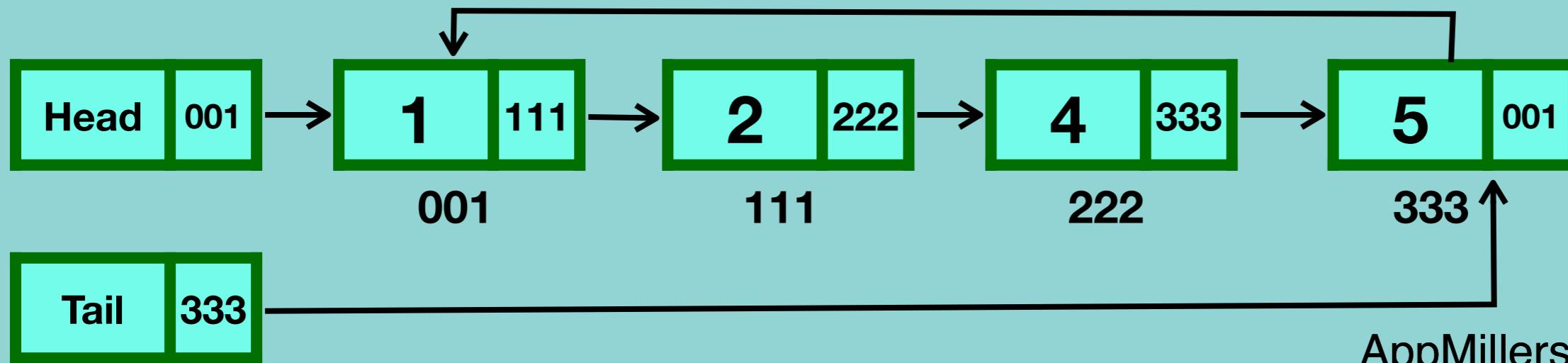
- Singly Linked List
- Circular Singly Linked List

## Singly Linked List

- Circular Doubly Linked List

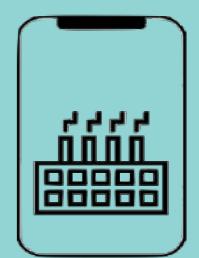
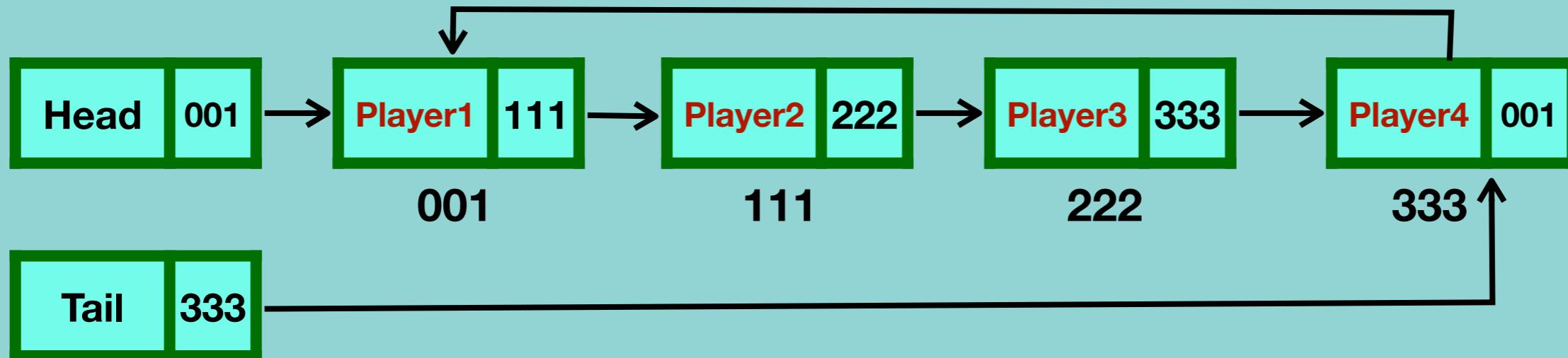


## Circular Singly Linked List



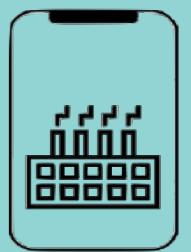
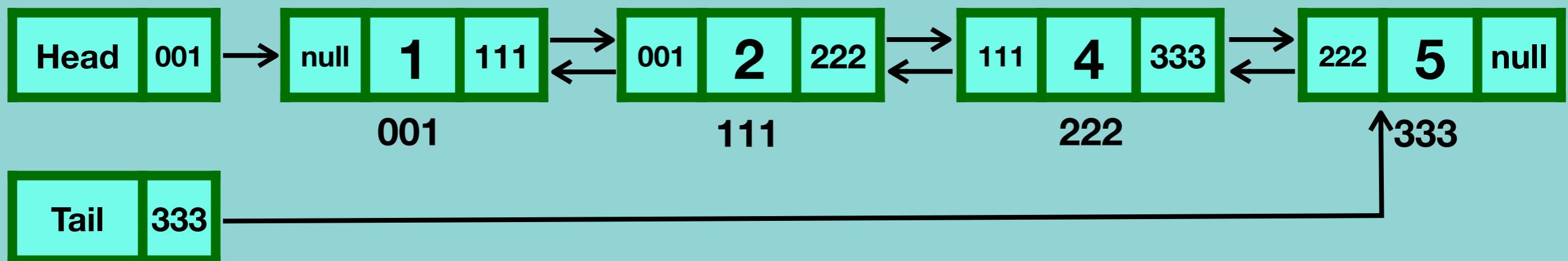


## Circular Singly Linked List

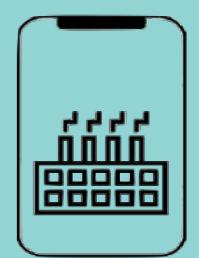
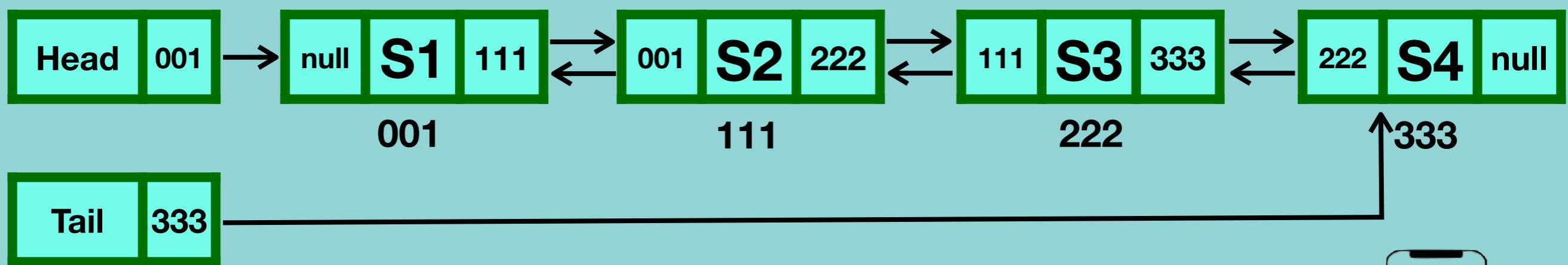
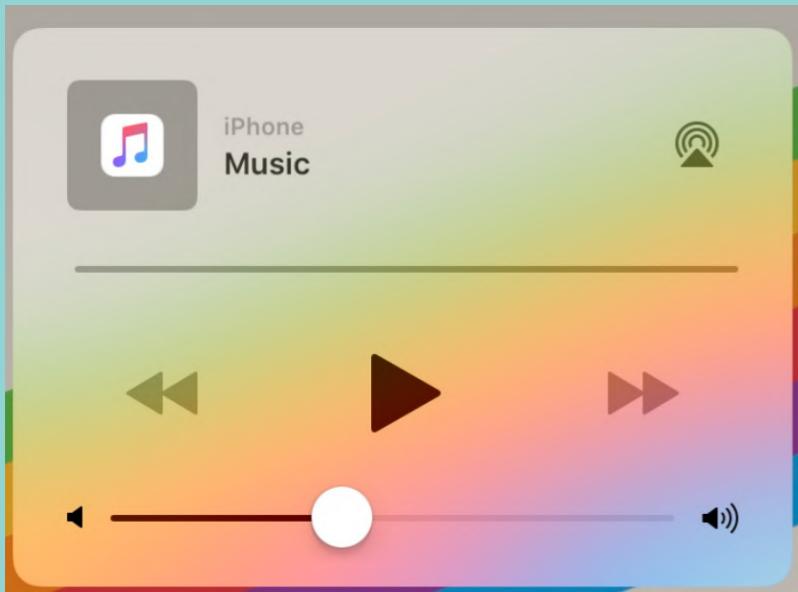


# Types of Linked Lists

## Doubly Linked List

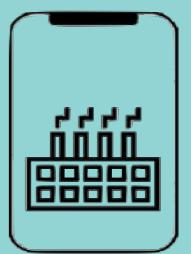
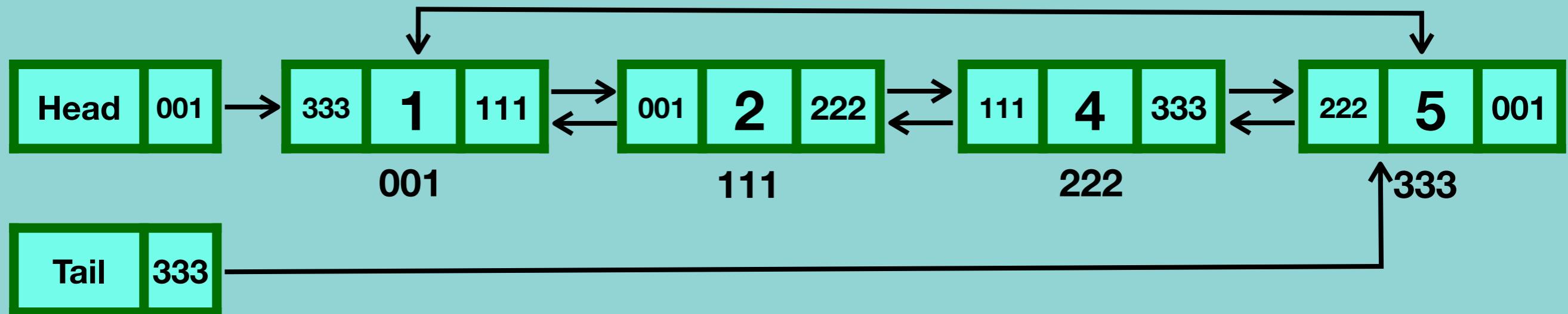


# Types of Linked Lists



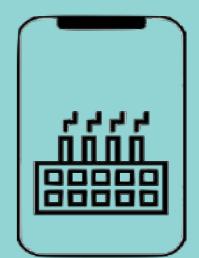
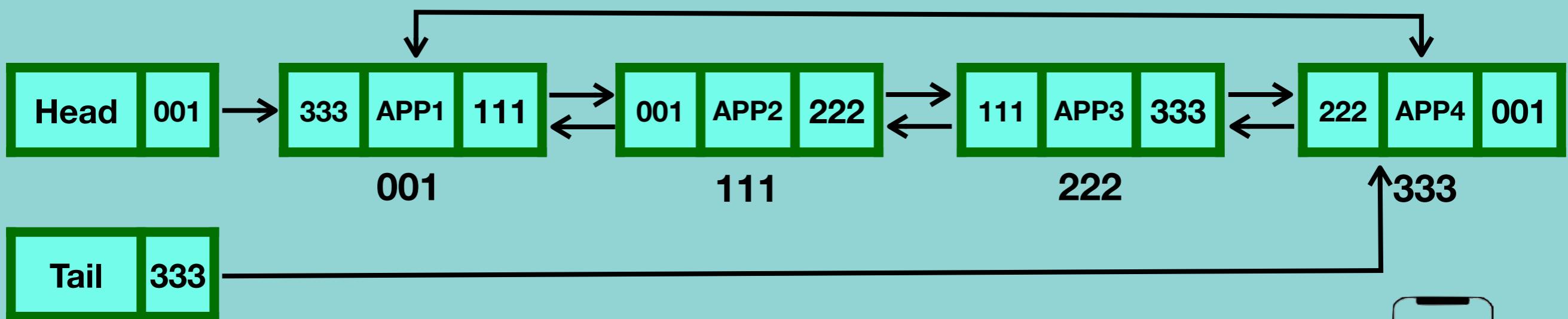
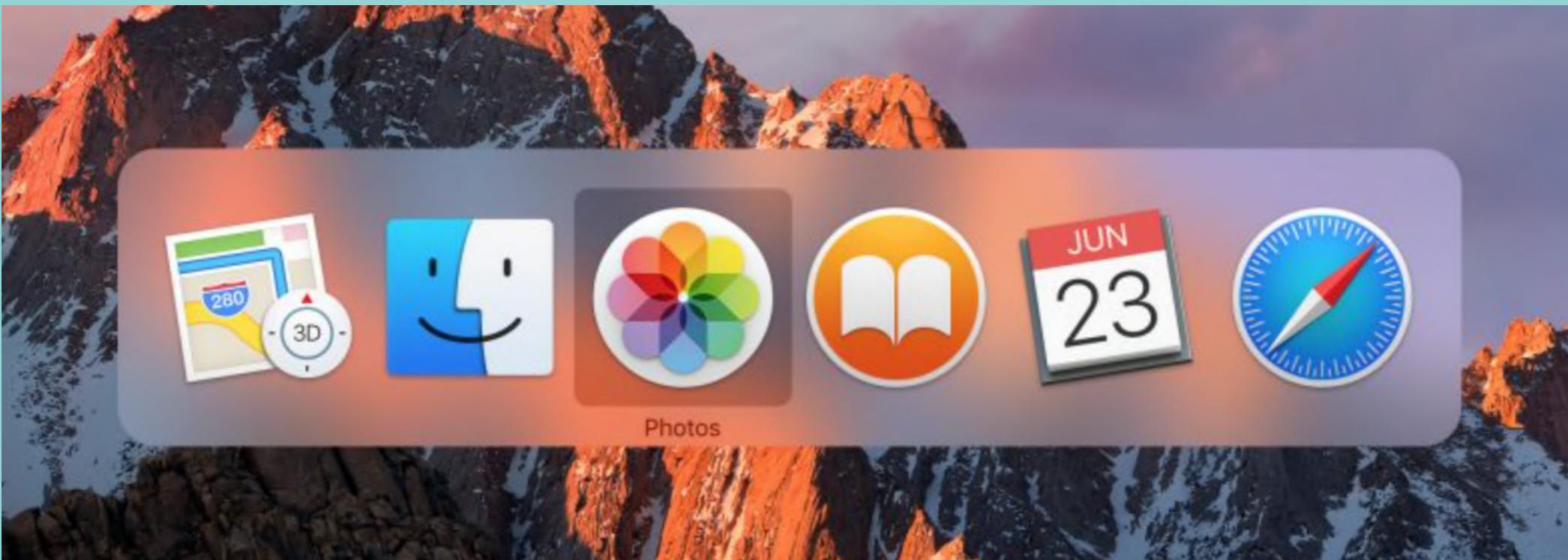
# Types of Linked Lists

## Circular Doubly Linked List



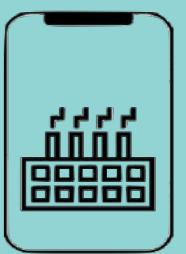
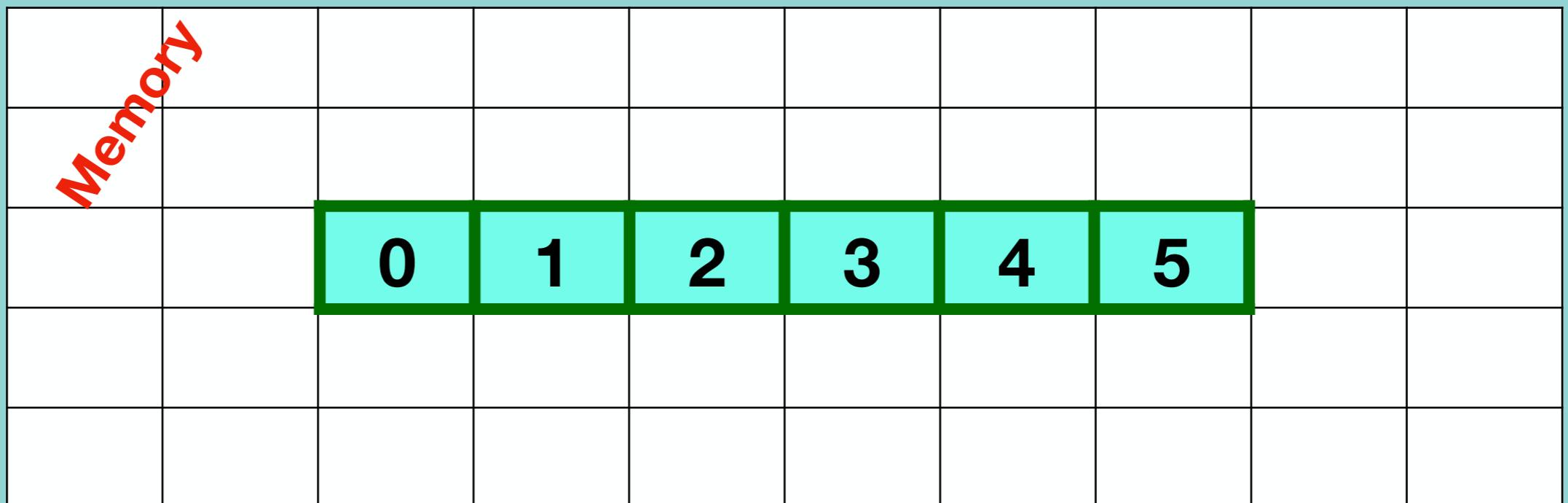
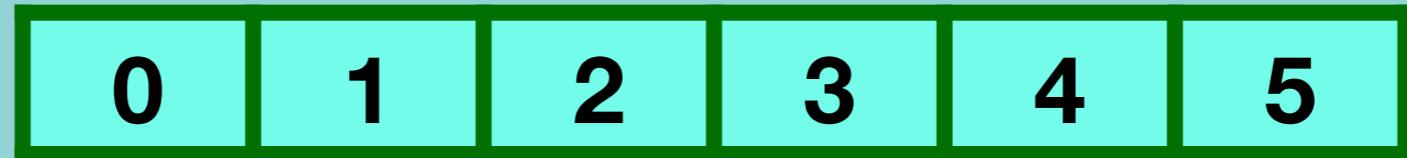
# Types of Linked Lists

Cmd+shift+tab



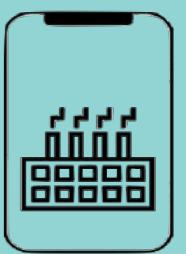
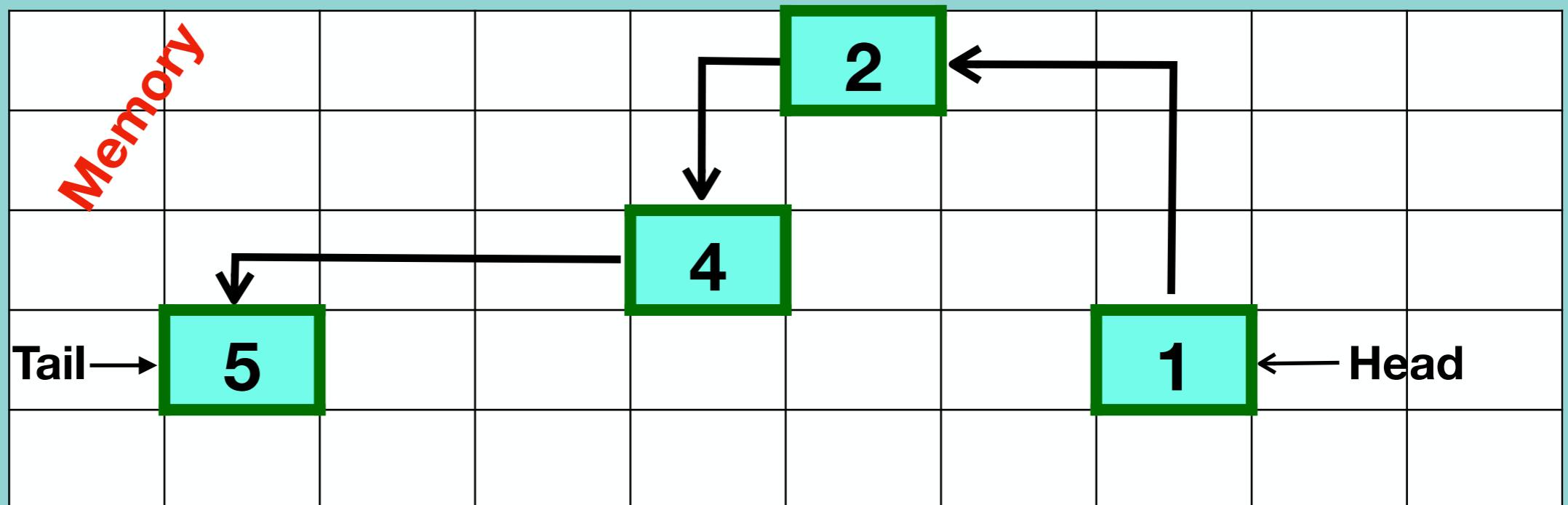
# Linked List in Memory

Arrays in memory :

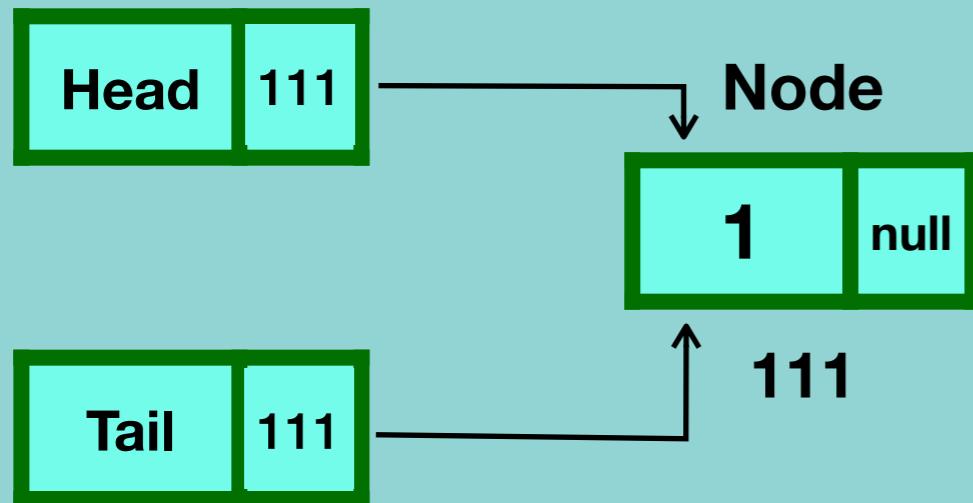


# Linked List in Memory

Linked list:



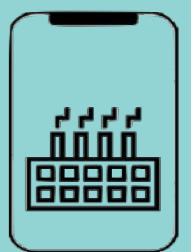
# Creation of Singly Linked List



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)**



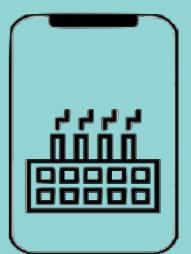
Create a blank Node and assign a value to it and reference to null.

.....  
→ **O(1)**



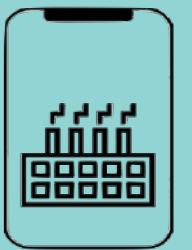
Link Head and Tail with these Node

.....  
→ **O(1)**



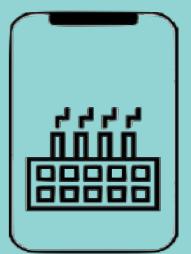
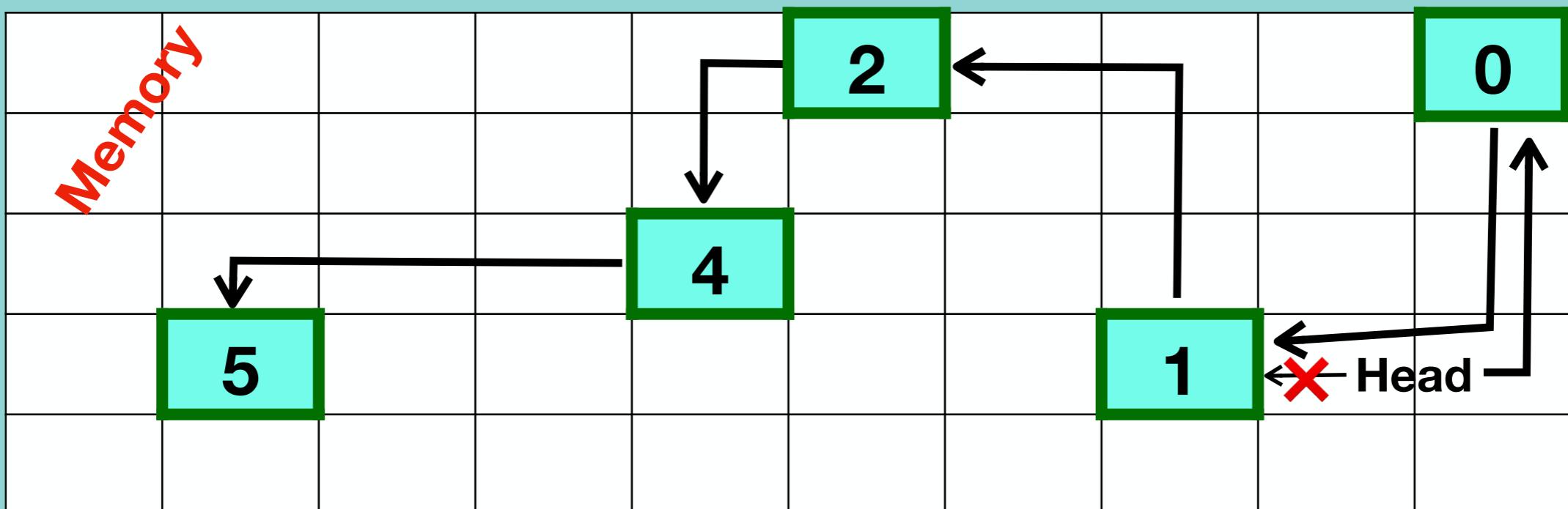
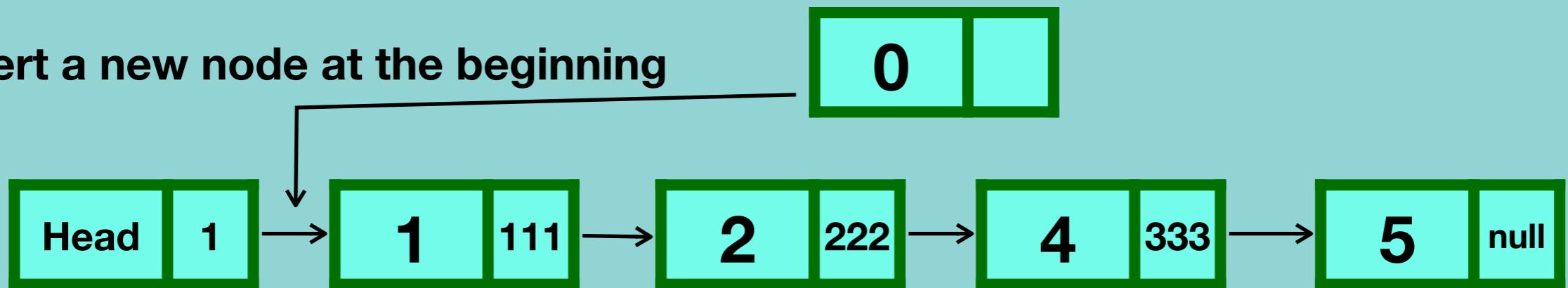
# Insertion to Linked List in Memory

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.



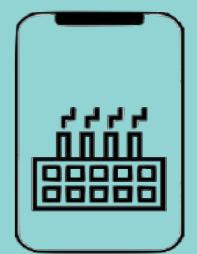
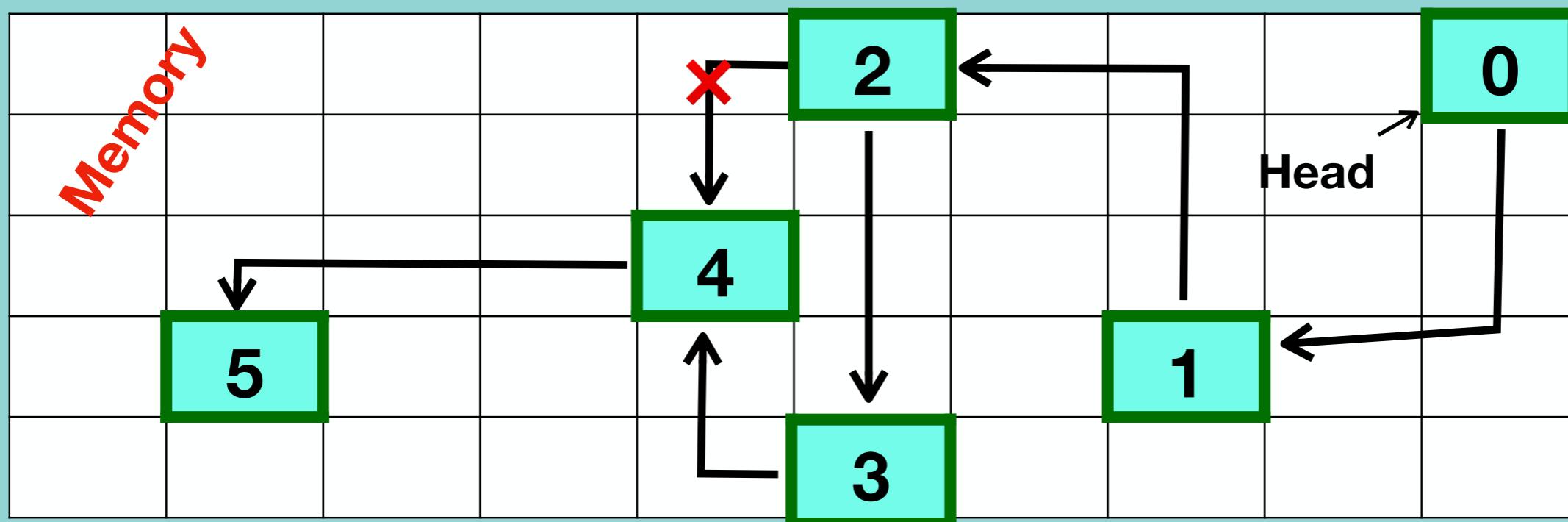
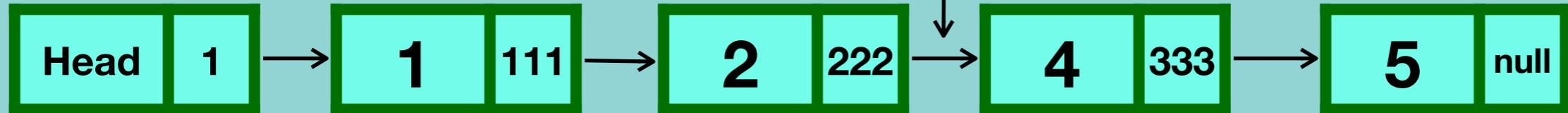
# Insertion to Linked List in Memory

Insert a new node at the beginning



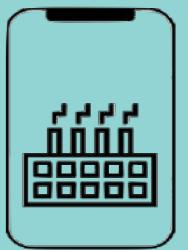
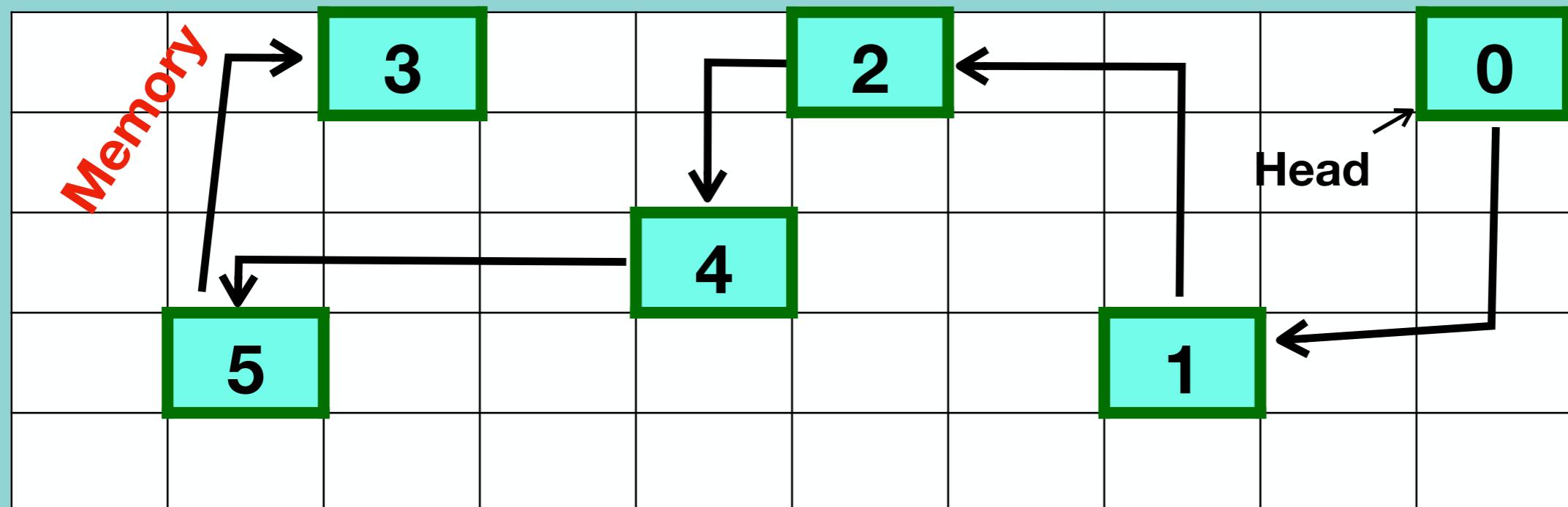
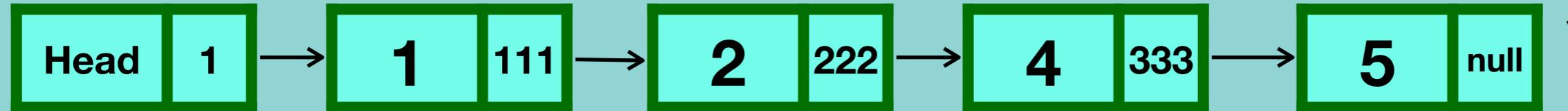
# Insertion to Linked List in Memory

Insert a new node after a node

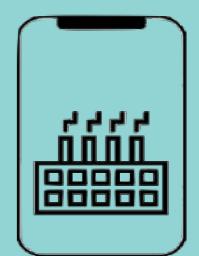
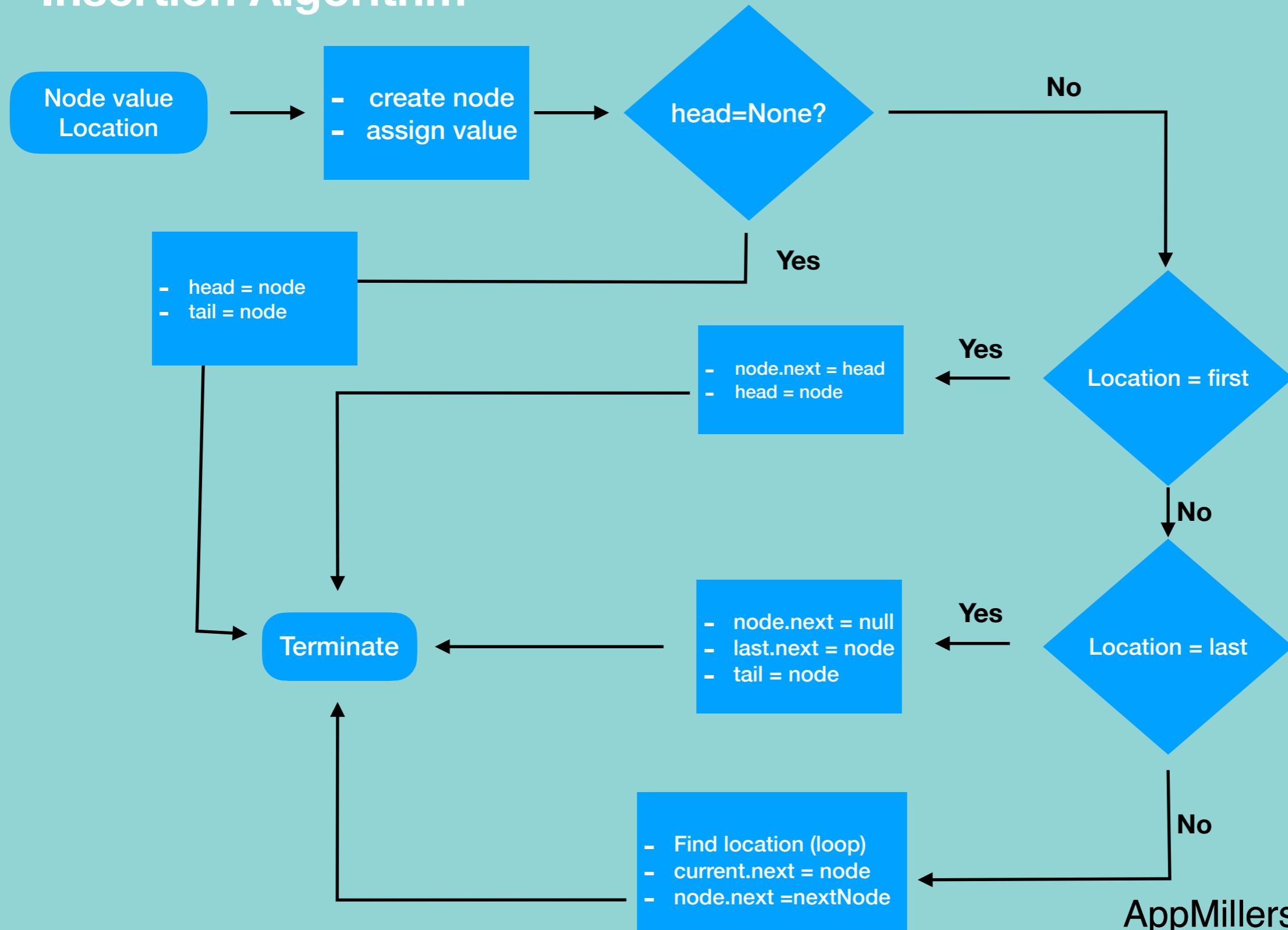


# Insertion to Linked List in Memory

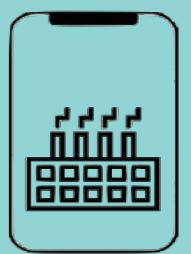
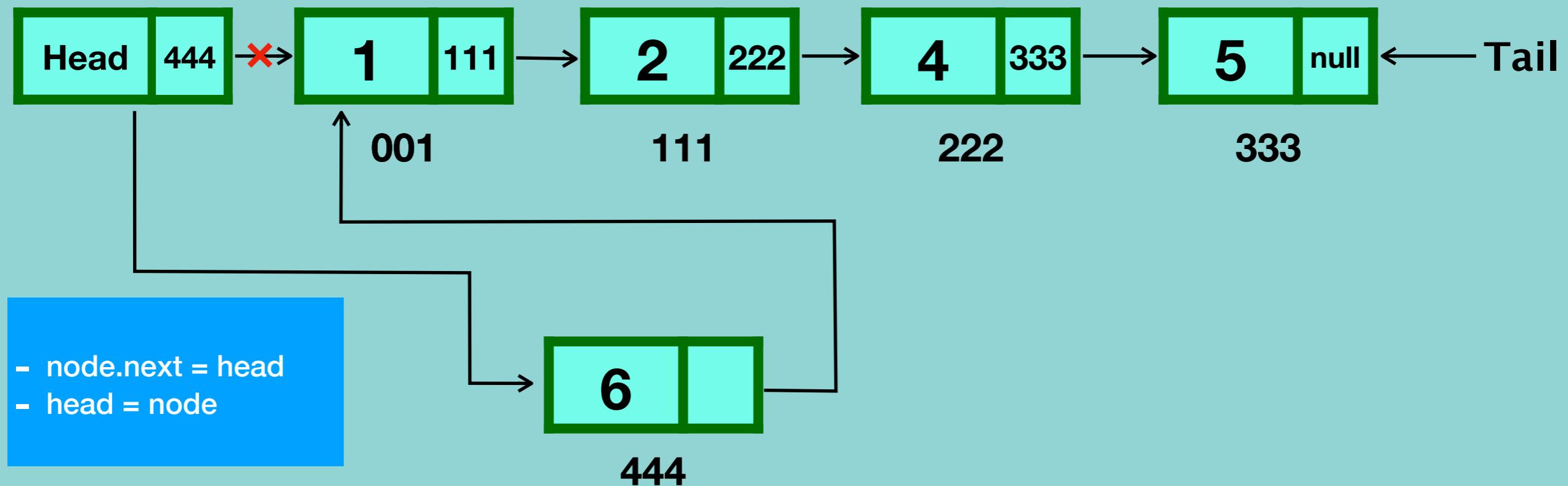
Insert a new node at the end of linked list



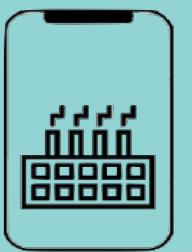
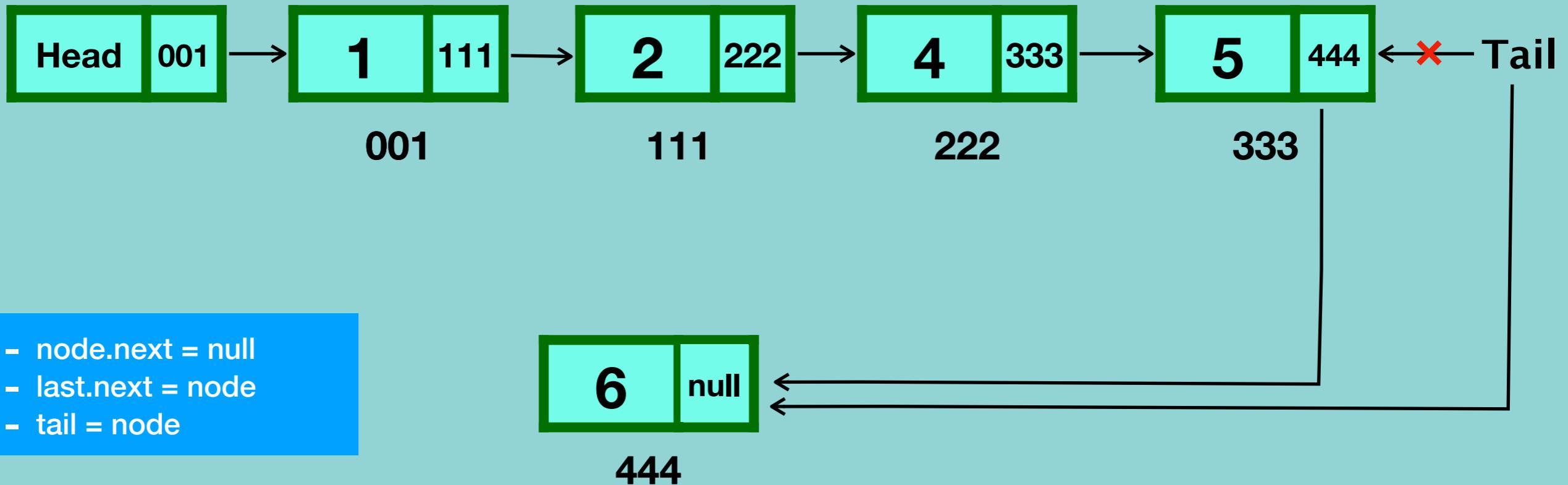
# Insertion Algorithm



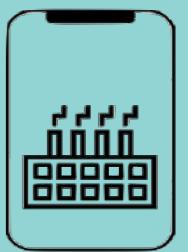
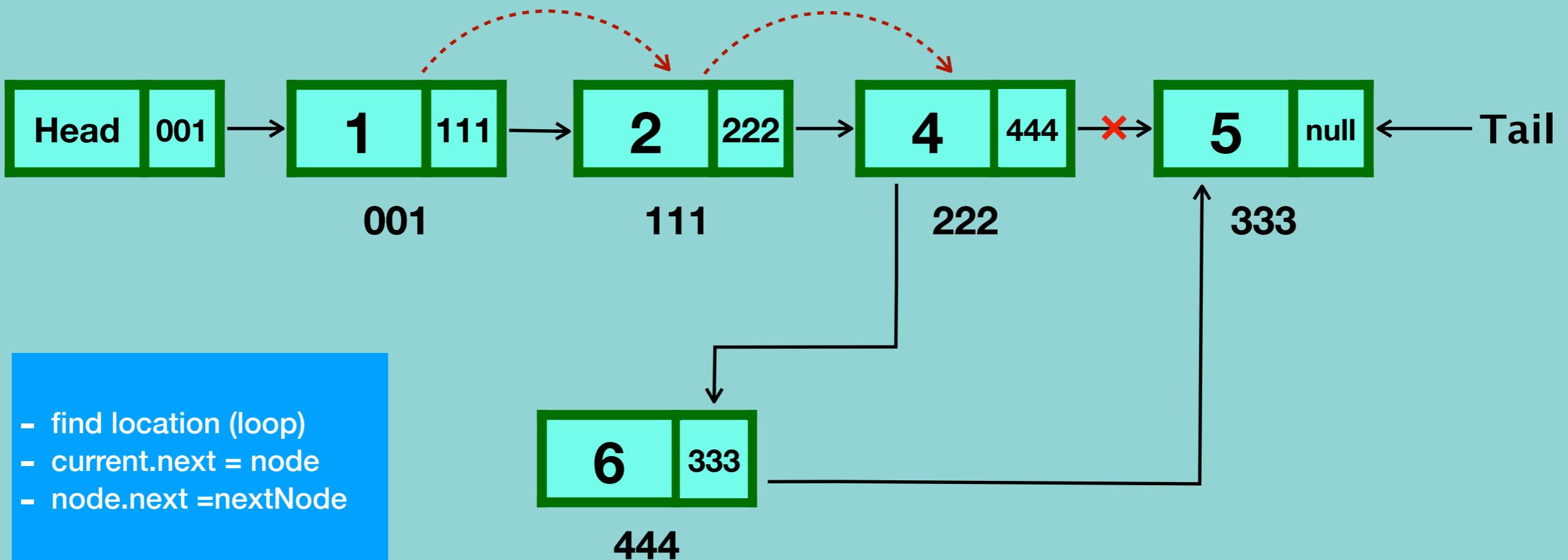
# Singly Linked List Insertion at the beginning



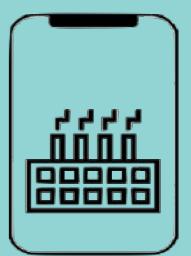
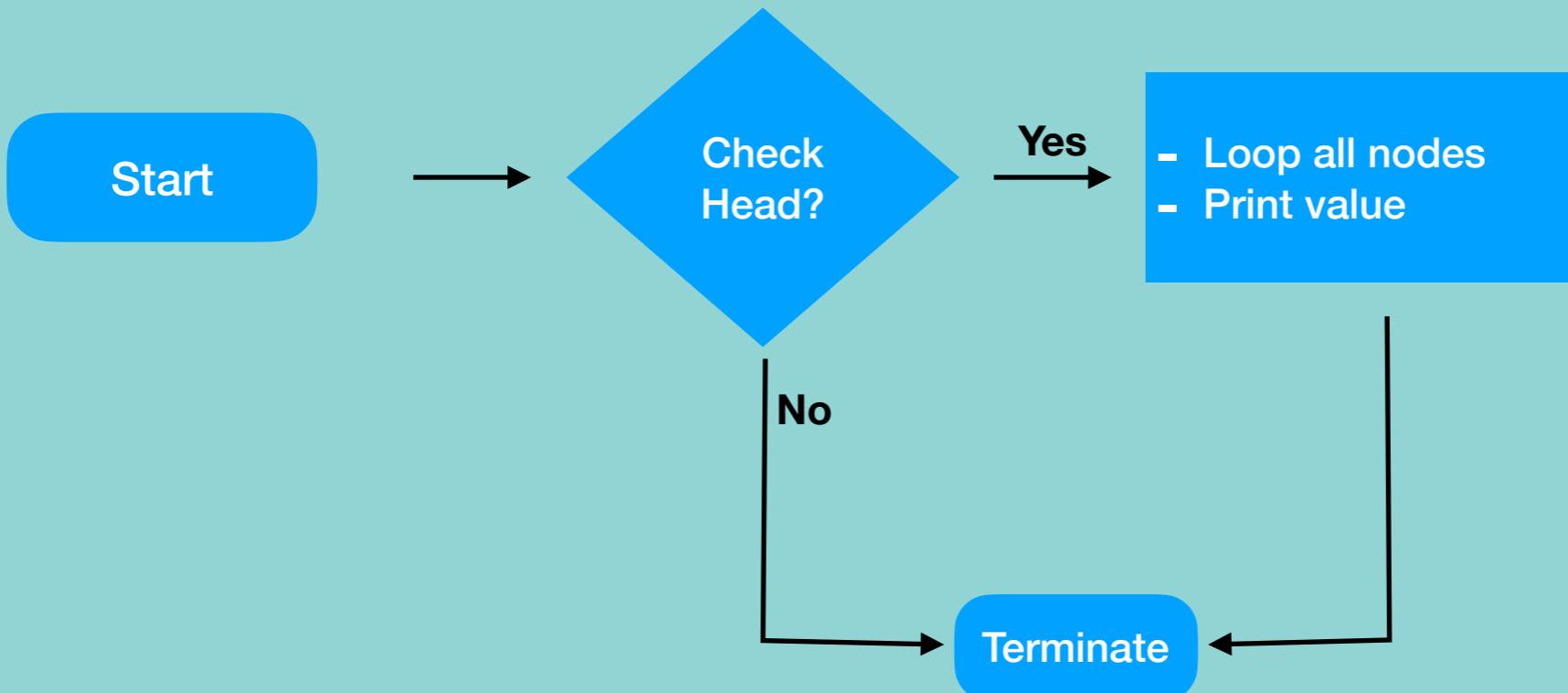
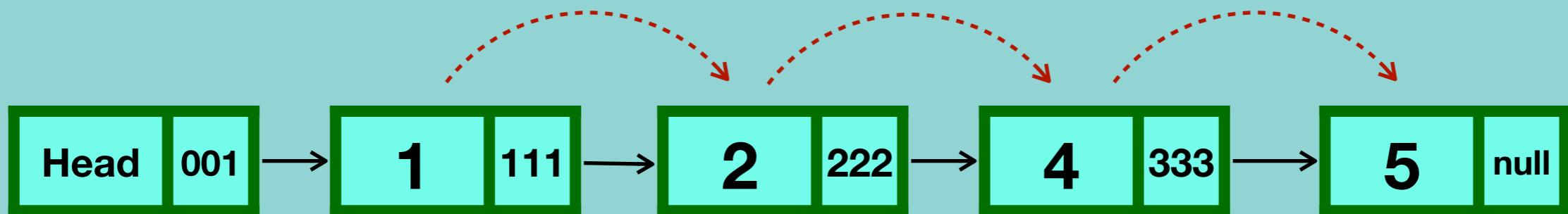
## Singly Linked List Insertion at the end



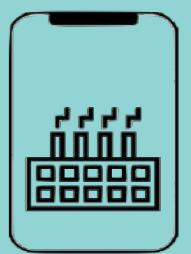
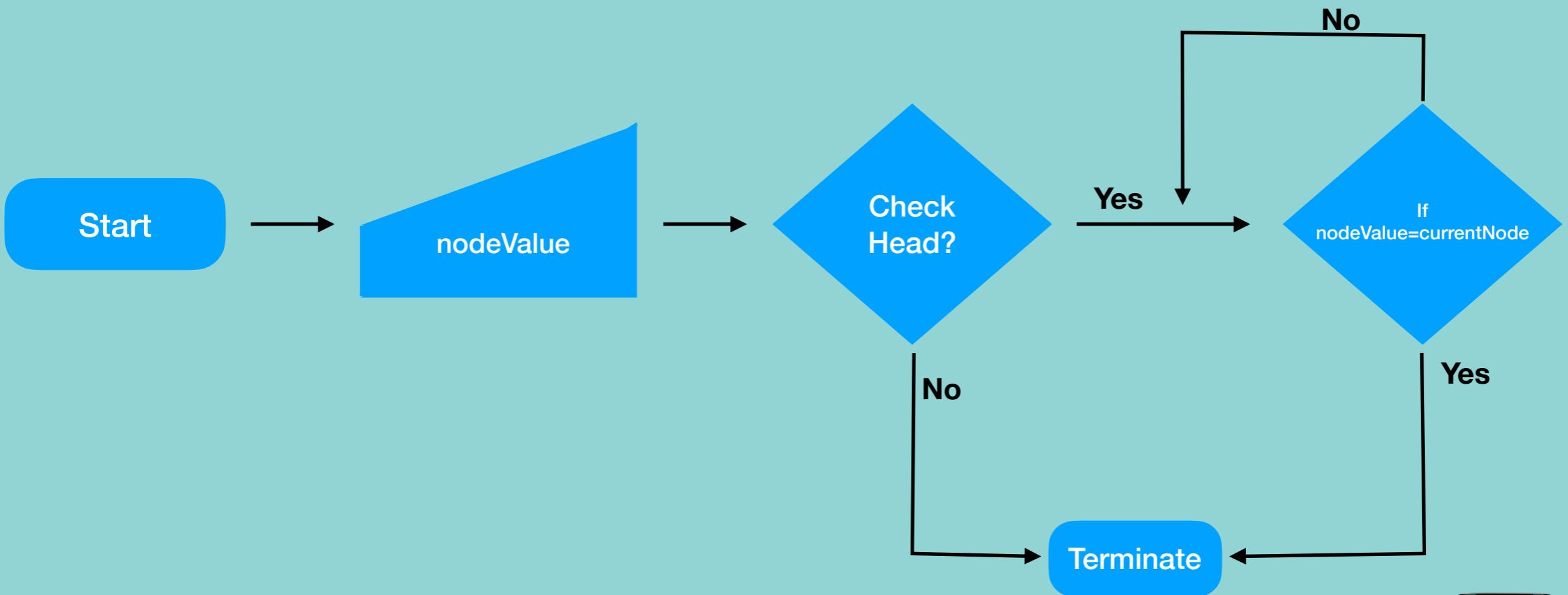
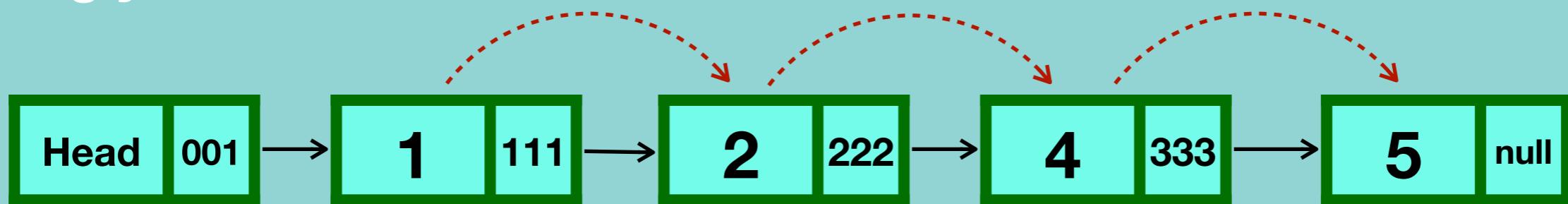
# Singly Linked List Insertion in the middle



# Singly Linked list Traversal



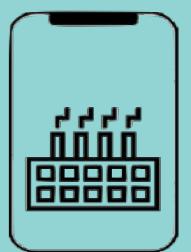
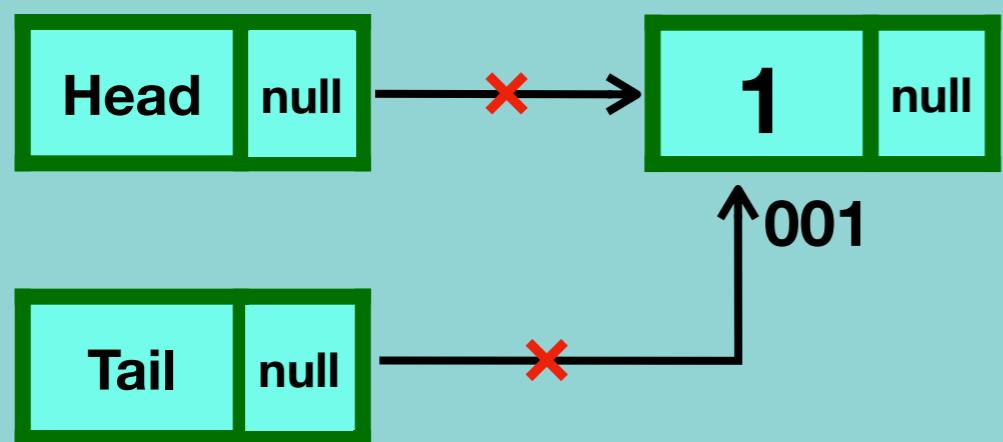
# Singly Linked list Search



# Singly Linked list Deletion

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

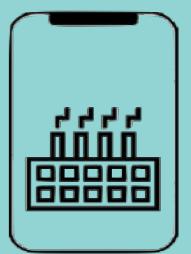
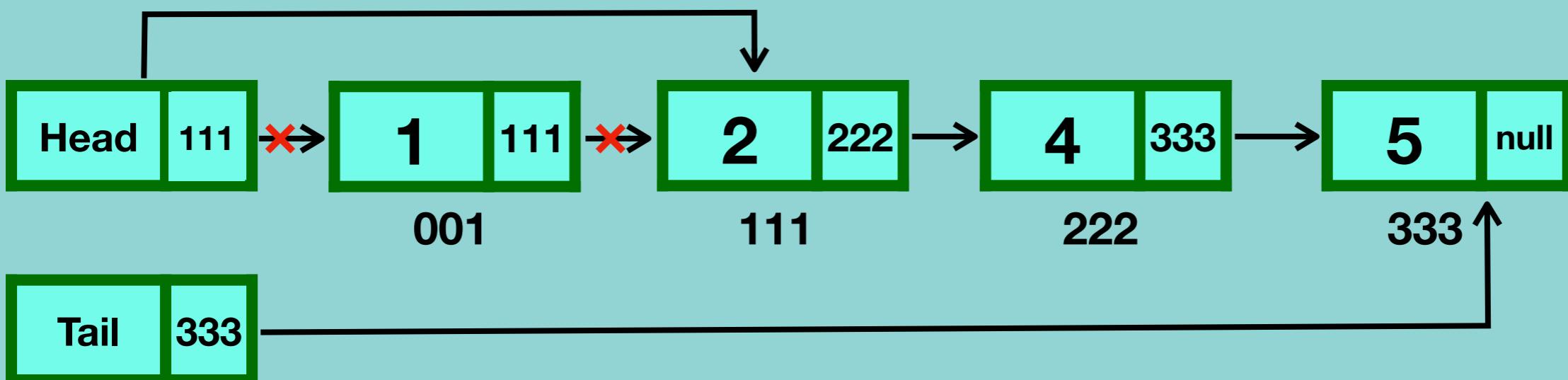
## Case 1 - one node



# Singly Linked list Deletion

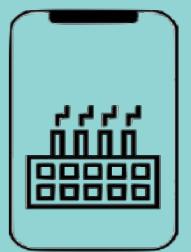
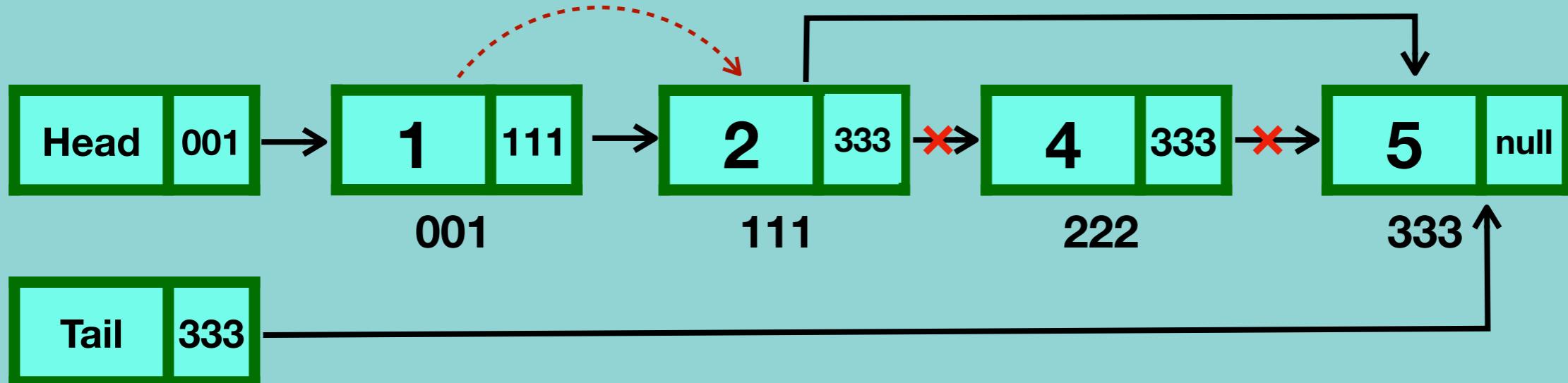
- Deleting the first node
- Deleting any given node
- Deleting the last node

## Case 2 - more than one node



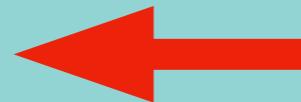
# Singly Linked list Deletion

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

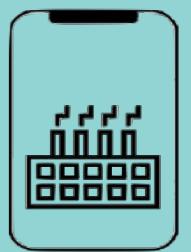
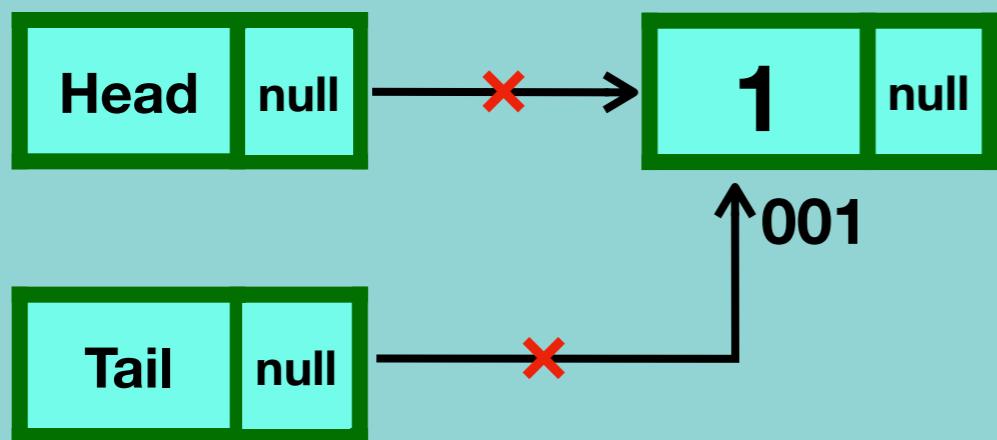


# Singly Linked list Deletion

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

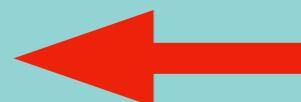


## Case 1 - one node

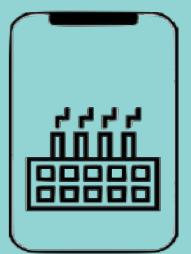
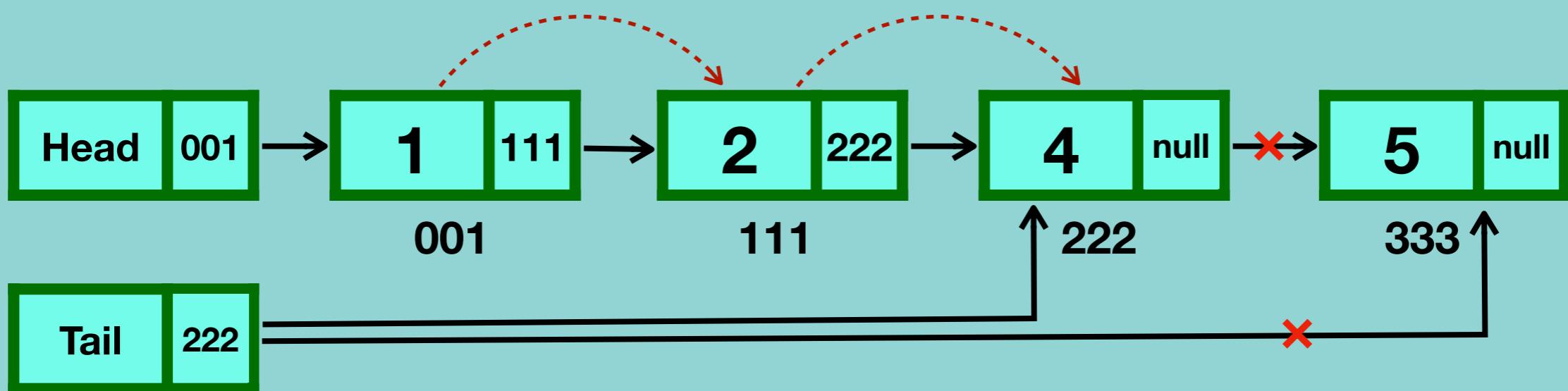


# Singly Linked list Deletion

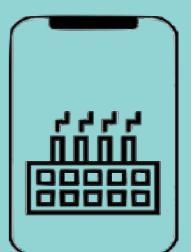
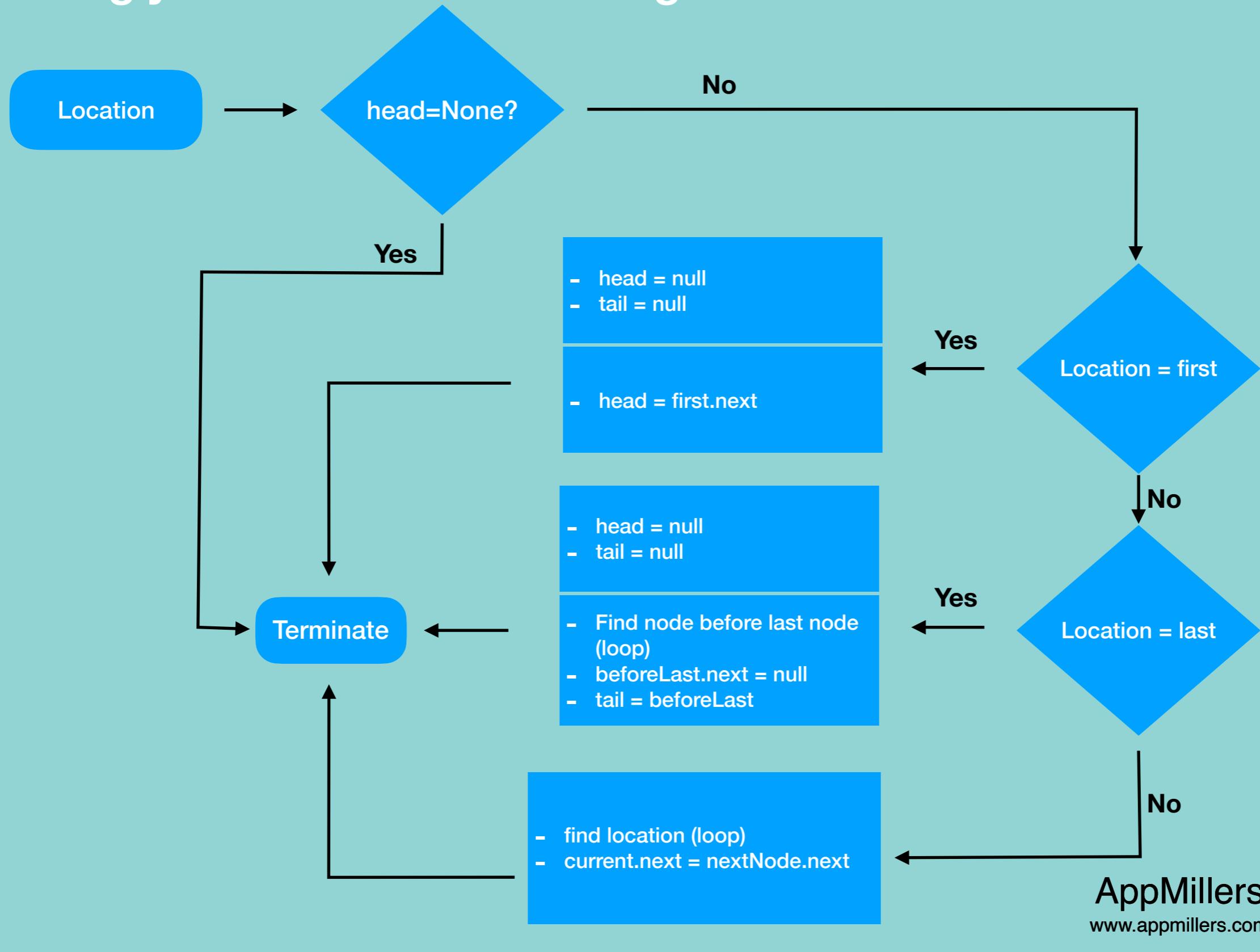
- Deleting the first node
- Deleting any given node
- Deleting the last node



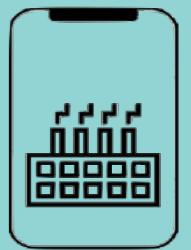
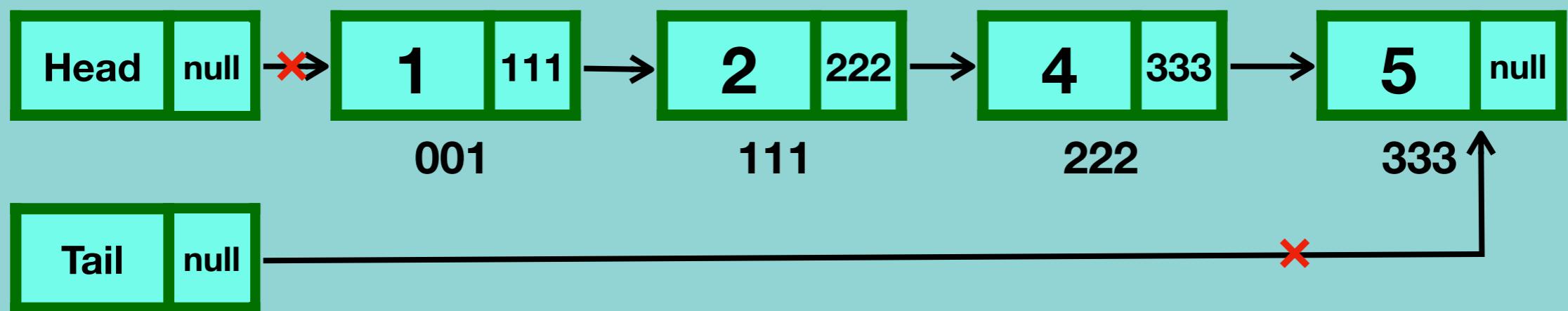
## Case 2 - more than one node



# Singly Linked list Deletion Algorithm

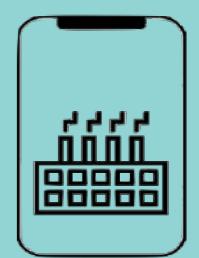
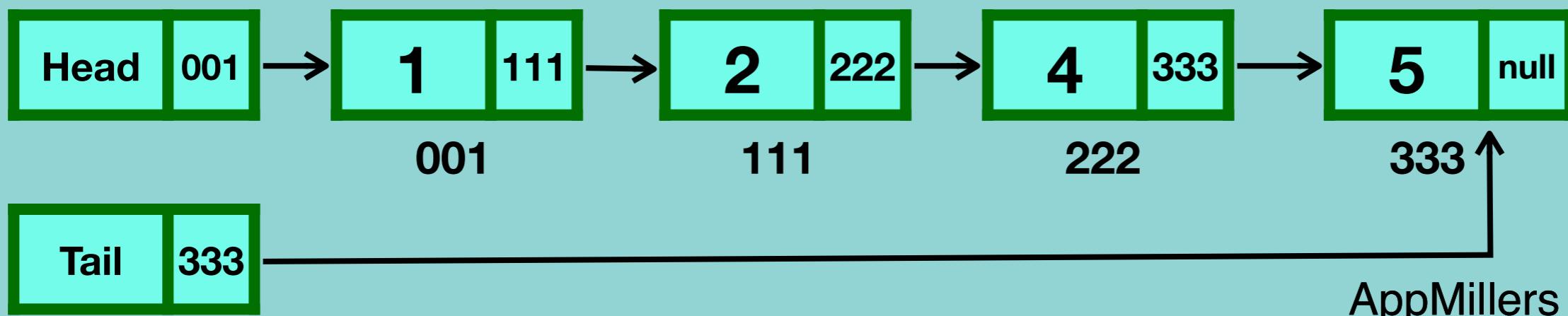


# Delete entire Singly Linked list



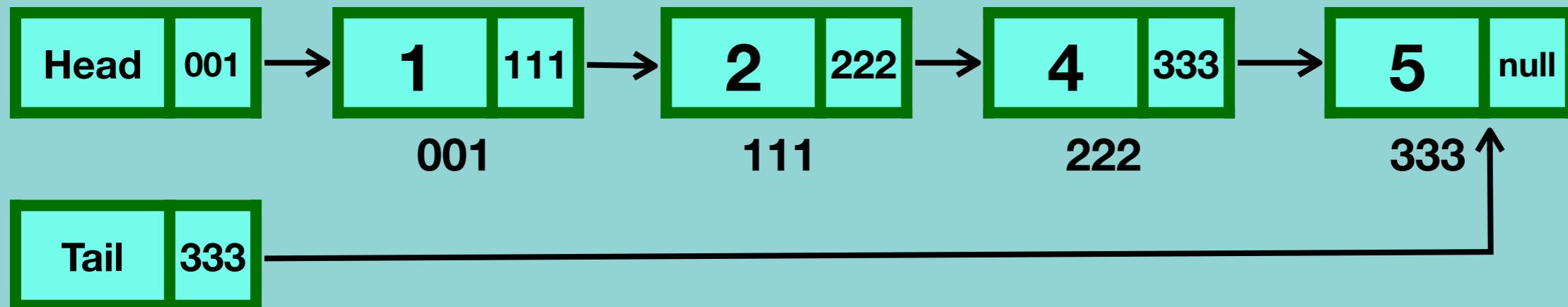
# Time and Space complexity of Singly Linked List

	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)

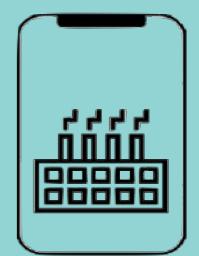
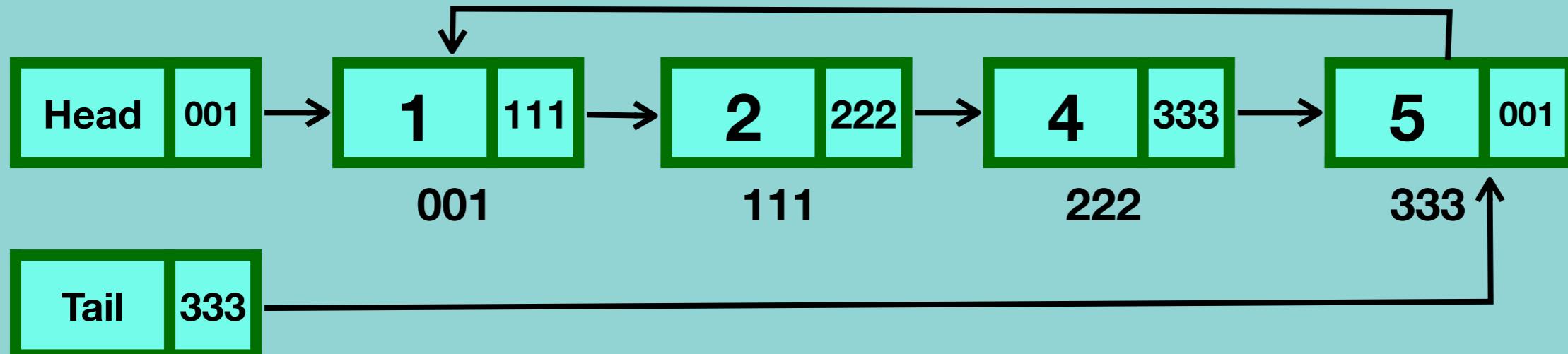


# Circular Singly Linked List

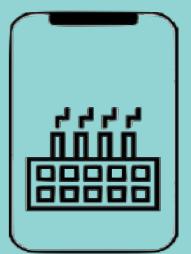
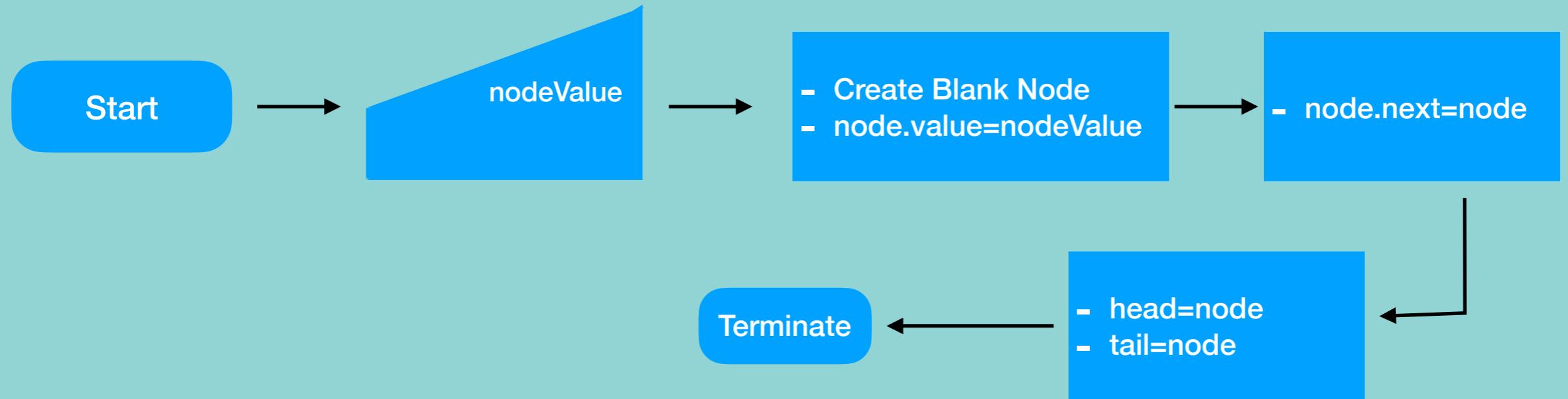
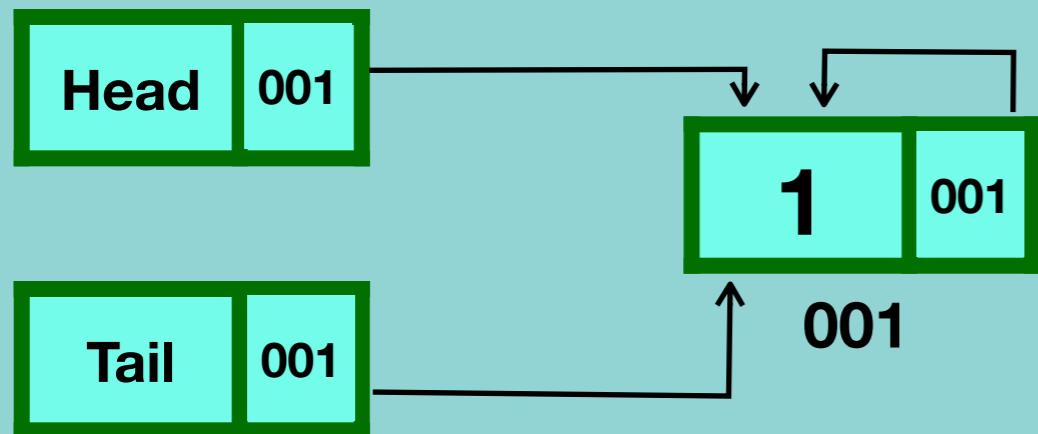
## Singly Linked List



## Circular Singly Linked List

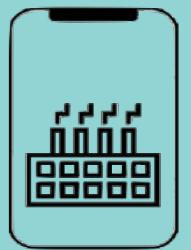
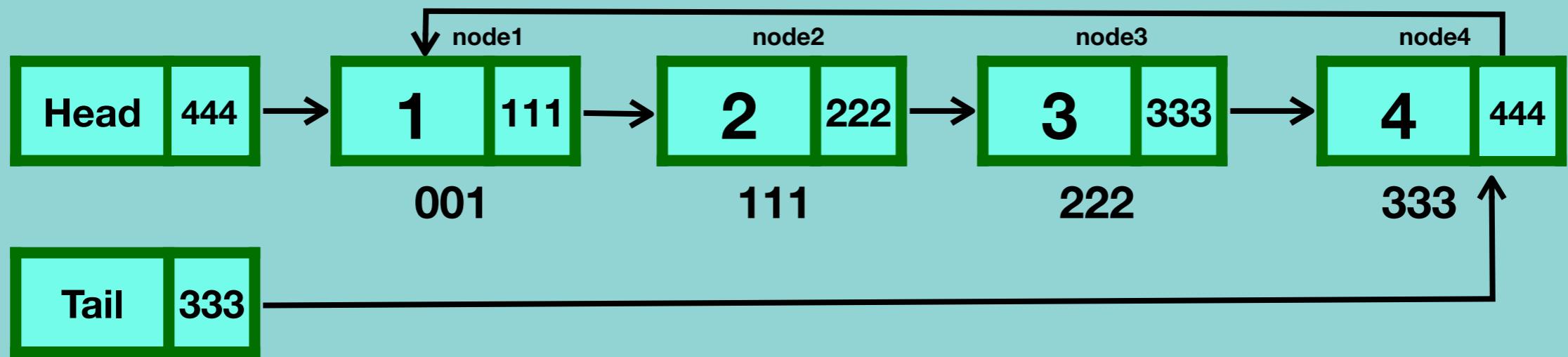


# Creation of Circular Singly Linked List



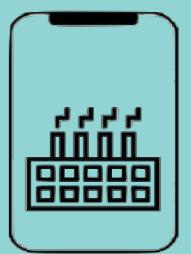
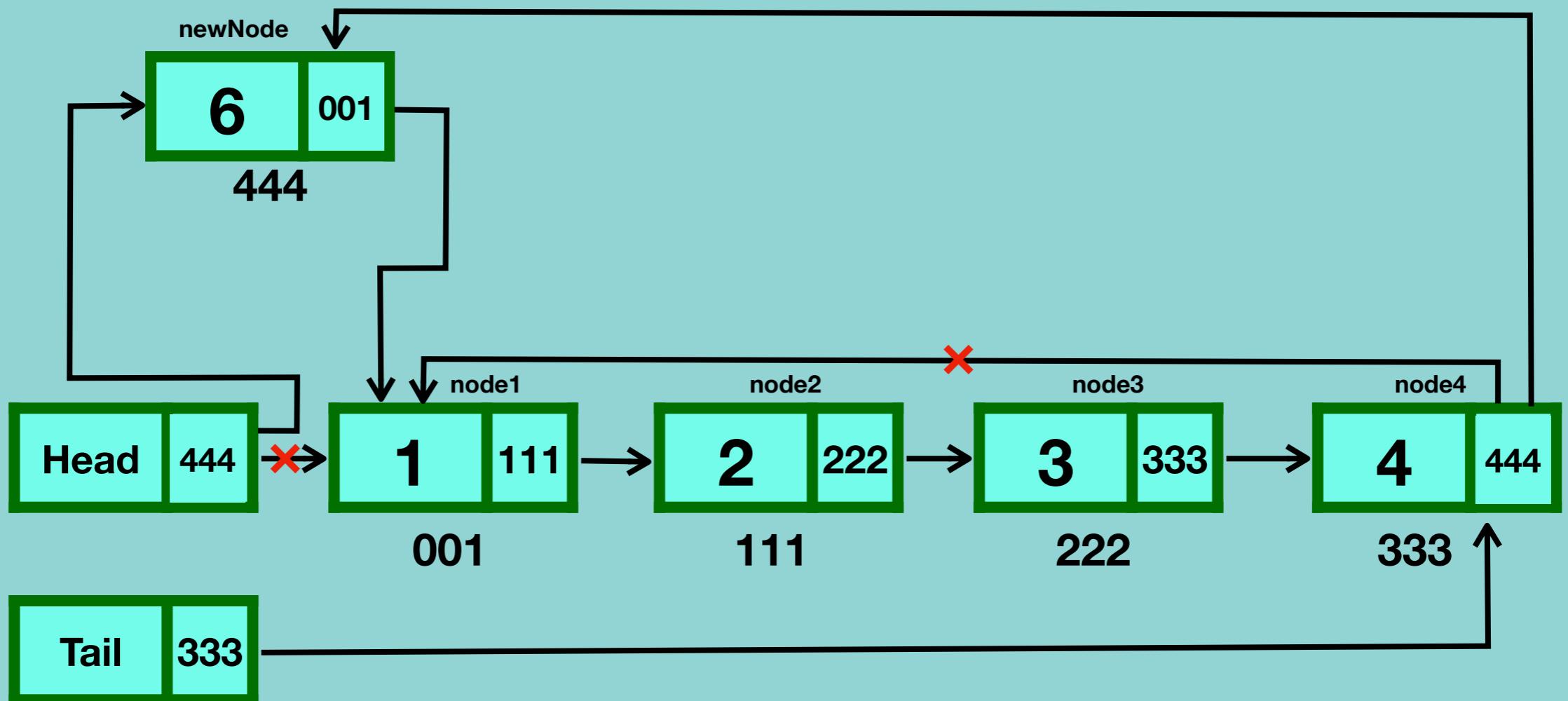
# Circular Singly Linked List - Insertion

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



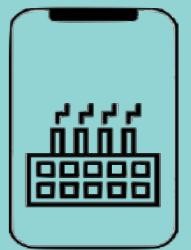
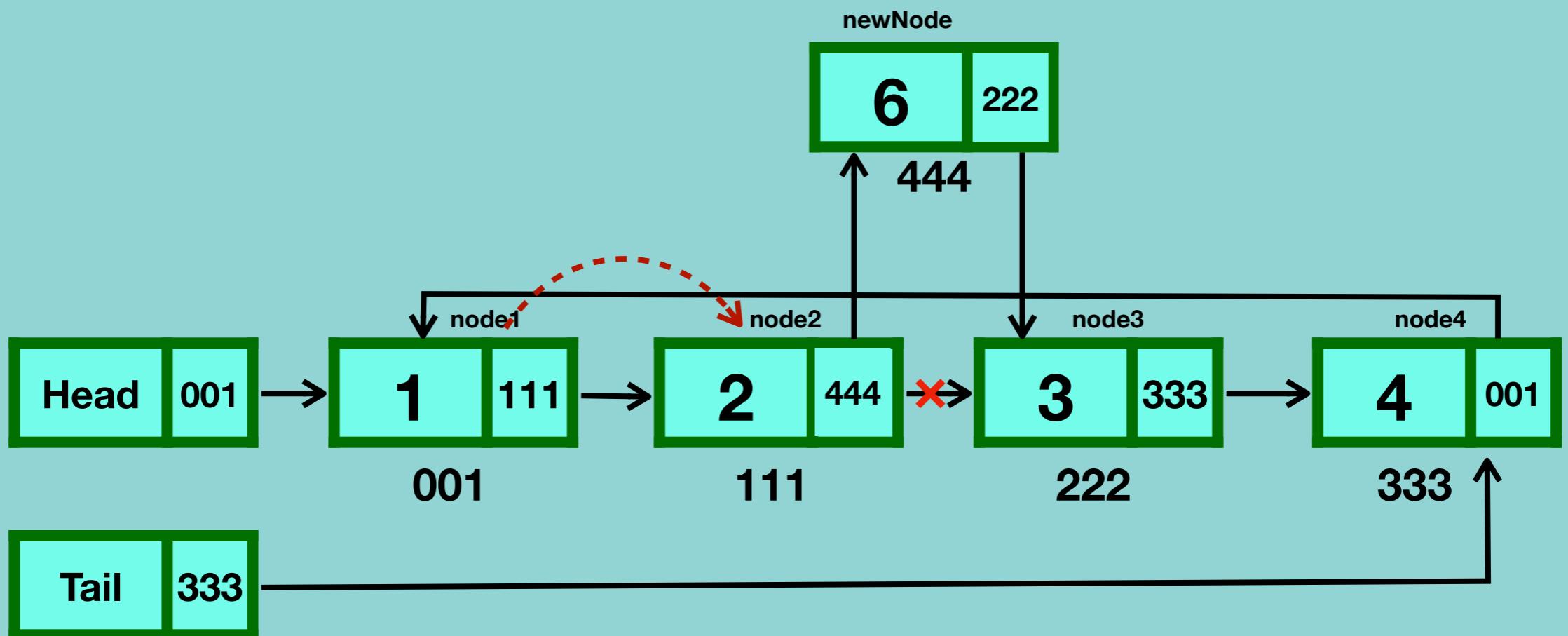
# Circular Singly Linked List - Insertion

- Insert at the beginning of linked list



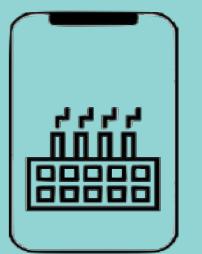
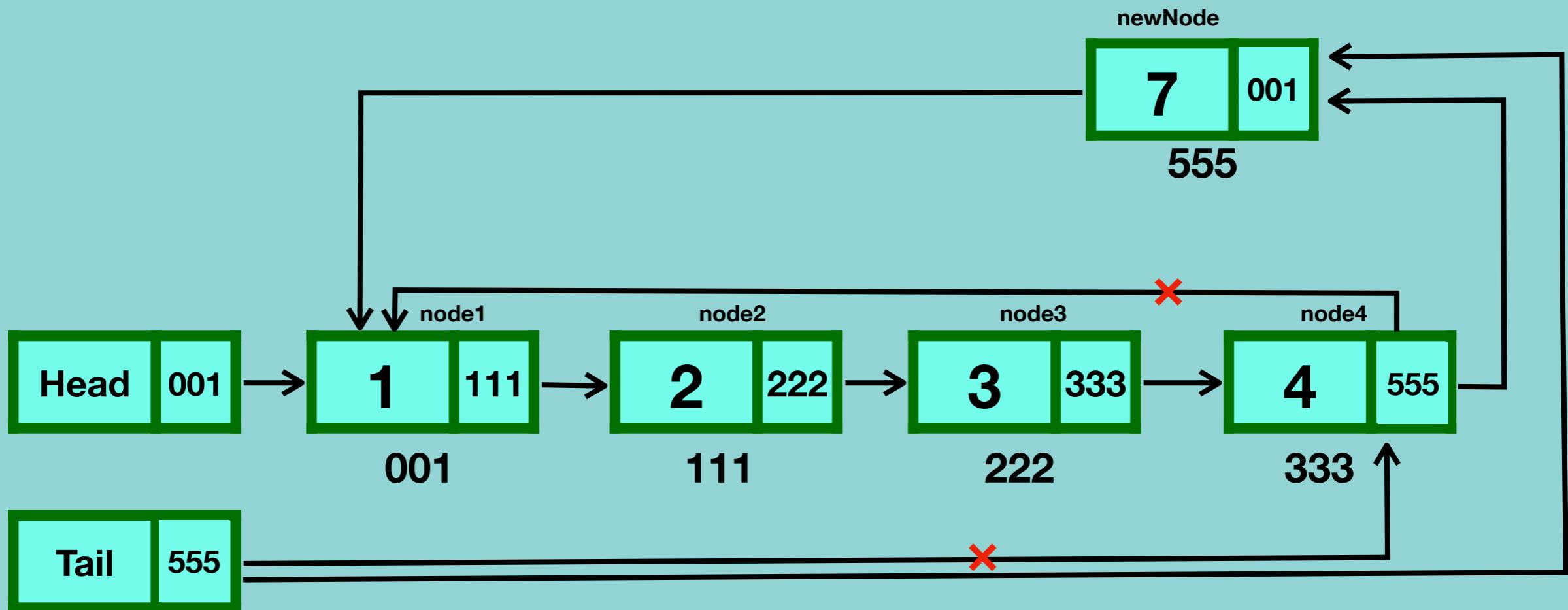
# Circular Singly Linked List - Insertion

- Insert at the specified location of linked list

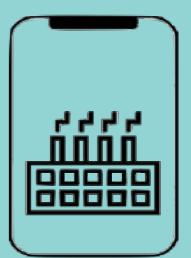
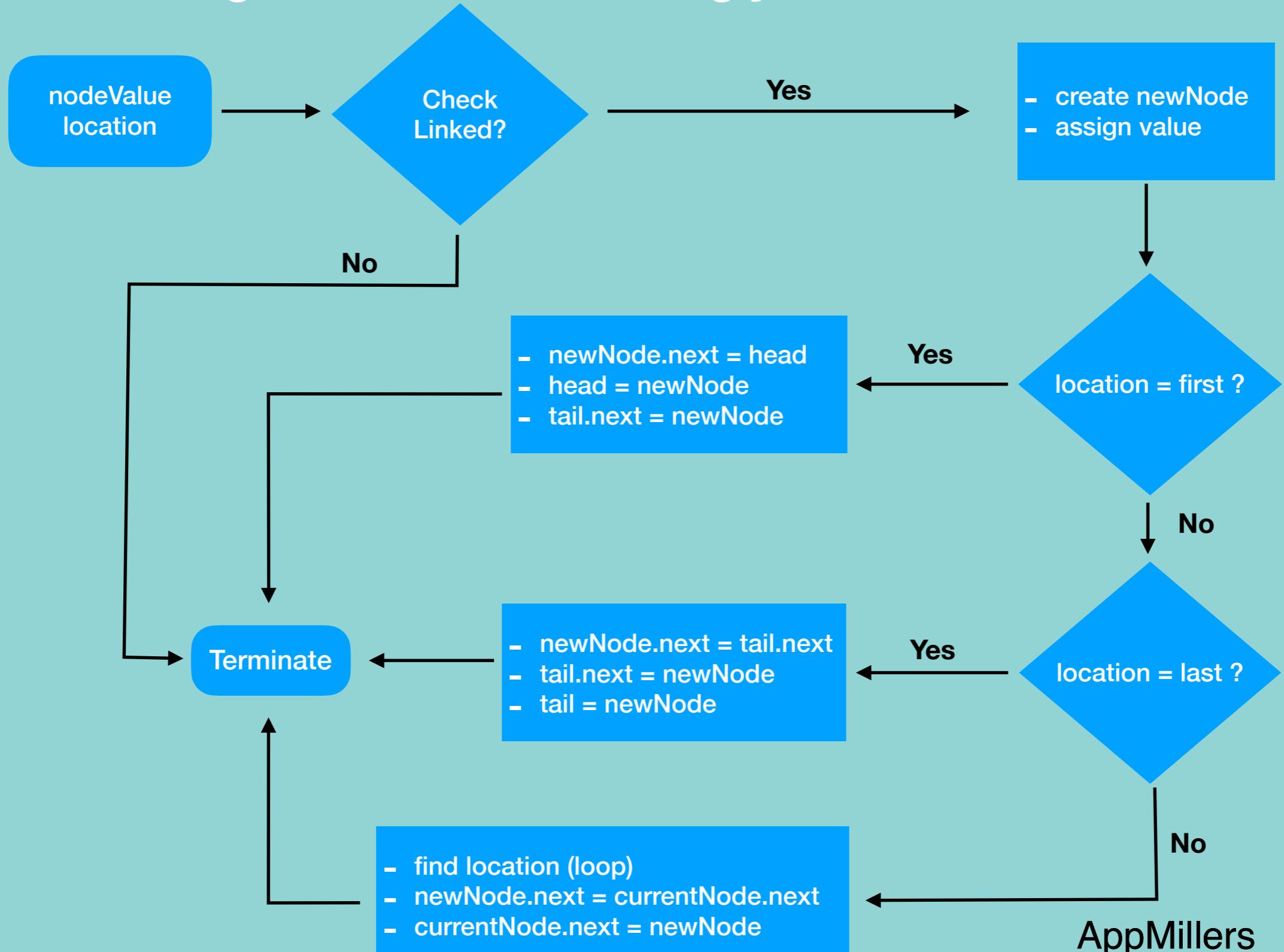


# Circular Singly Linked List - Insertion

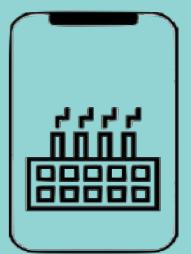
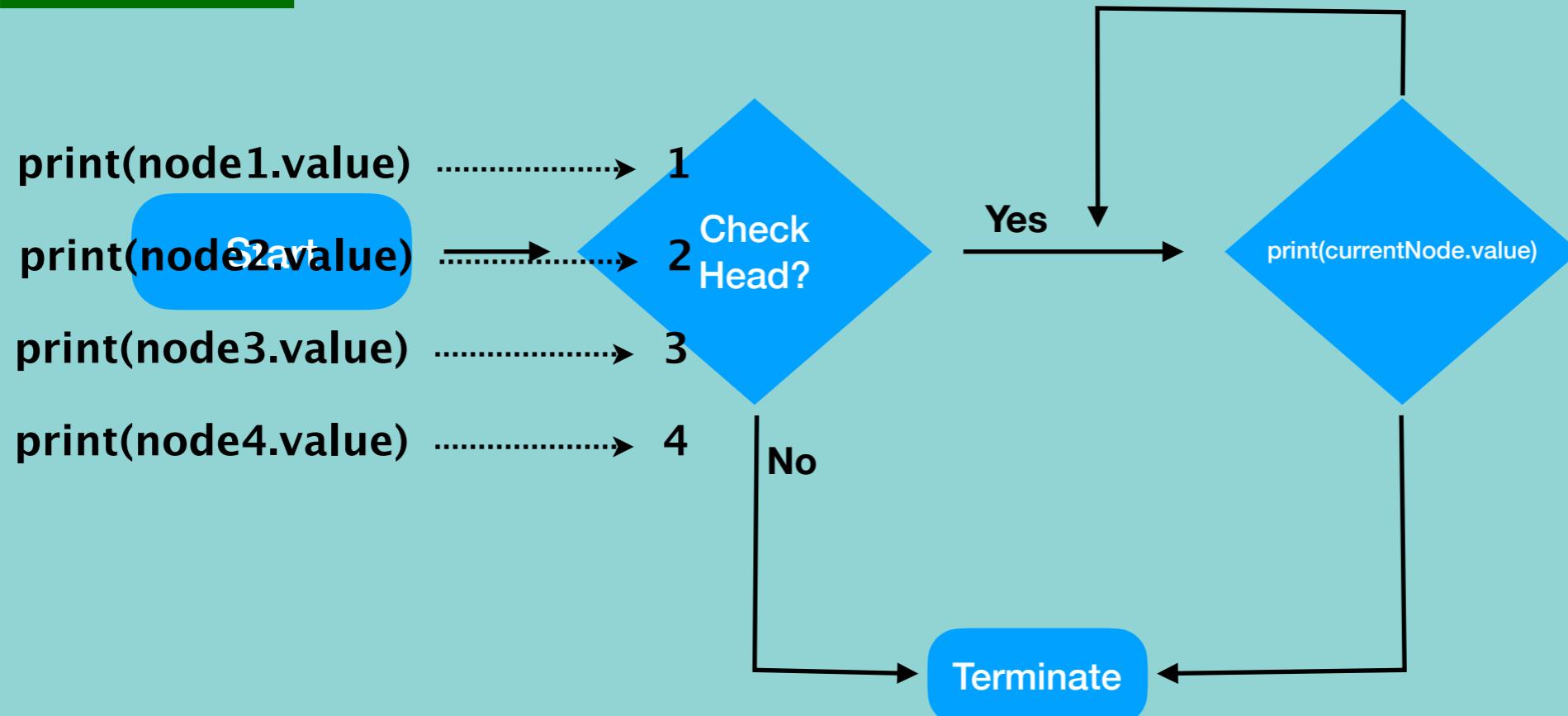
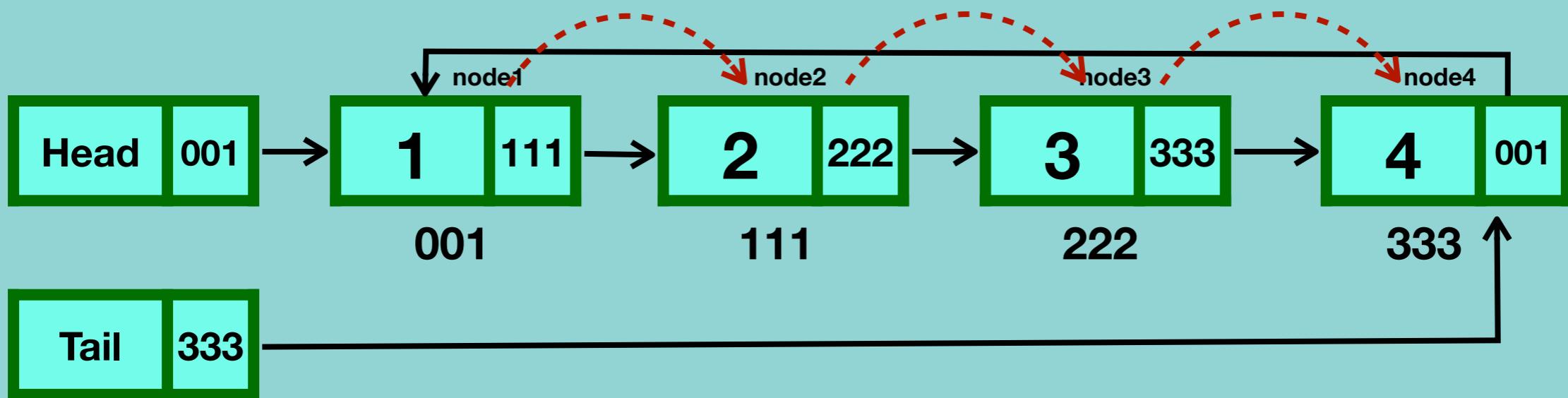
- Insert at the end of linked list



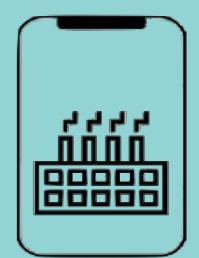
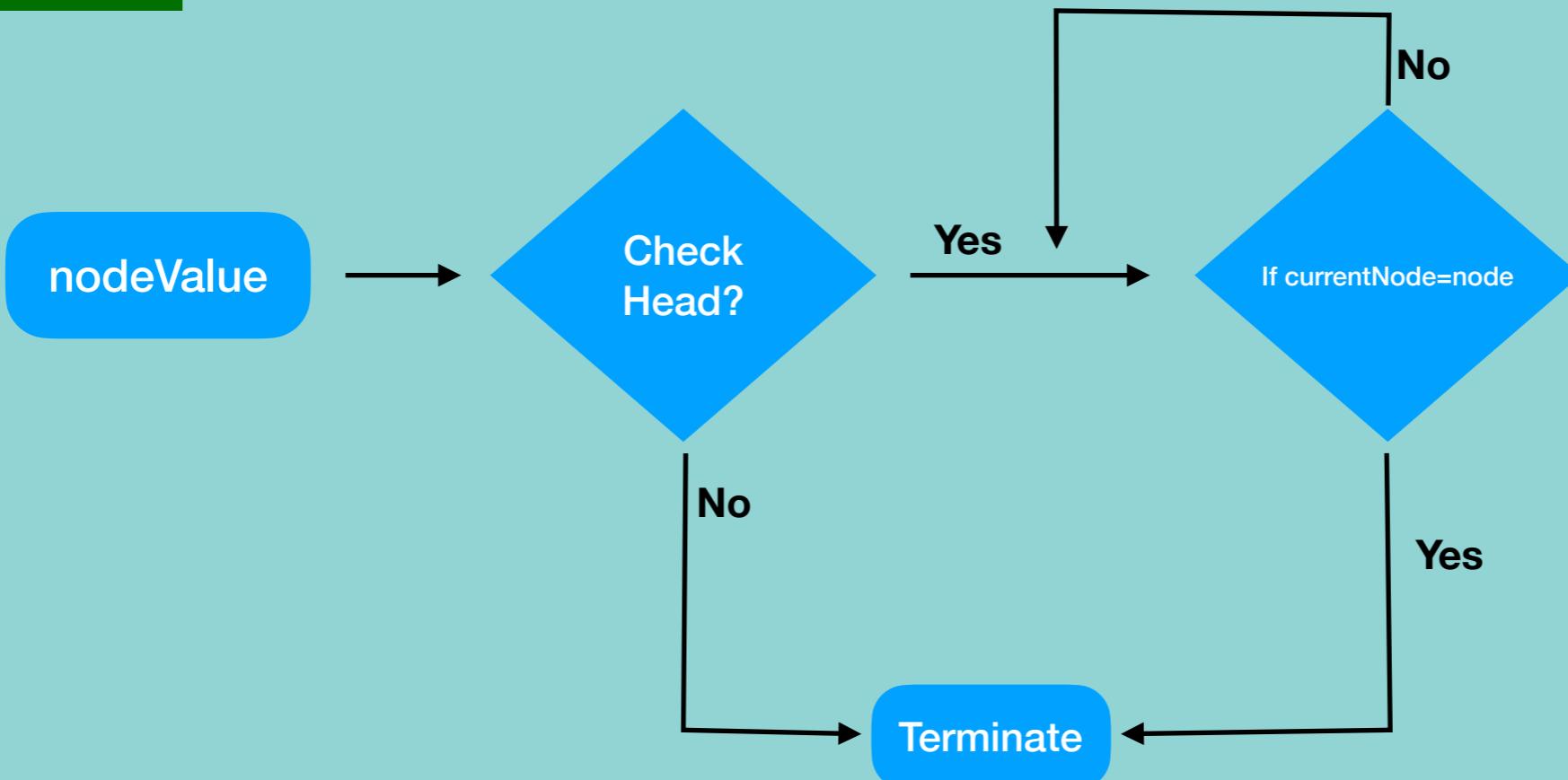
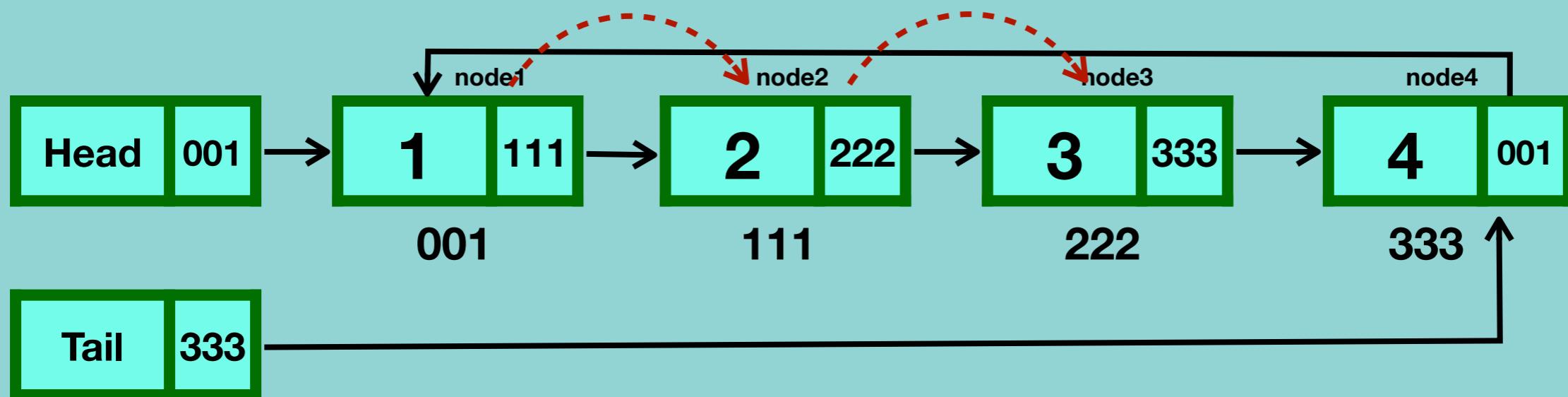
# Insertion Algorithm - Circular singly linked list



# Circular Singly Linked List - Traversal



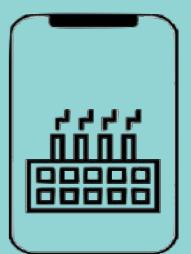
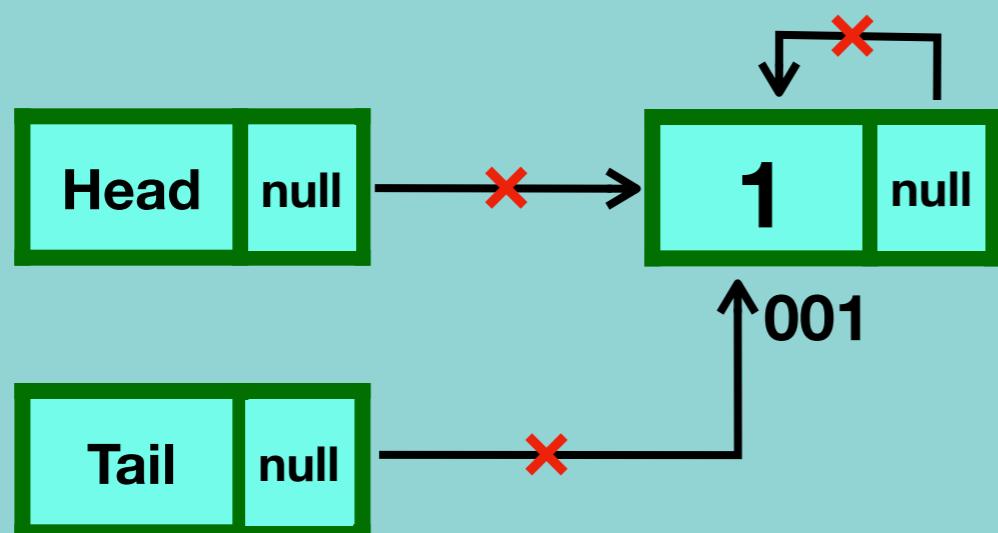
# Circular Singly Linked List - Searching



# Circular Singly Linked list - Deletion

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

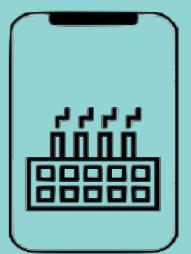
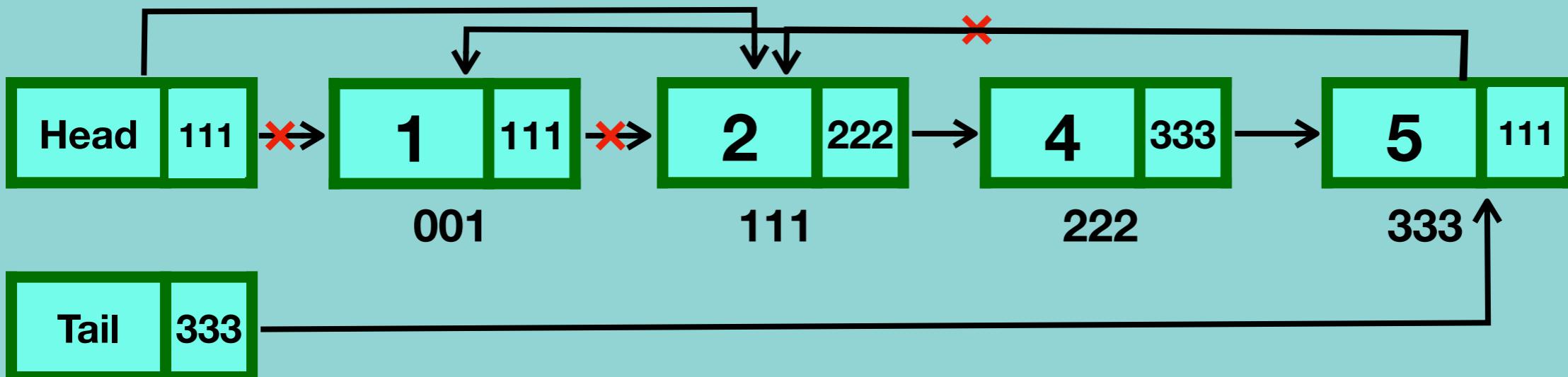
## Case 1 - one node



## Circular Singly Linked list - Deletion

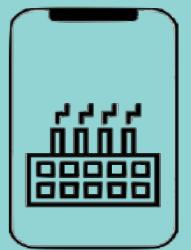
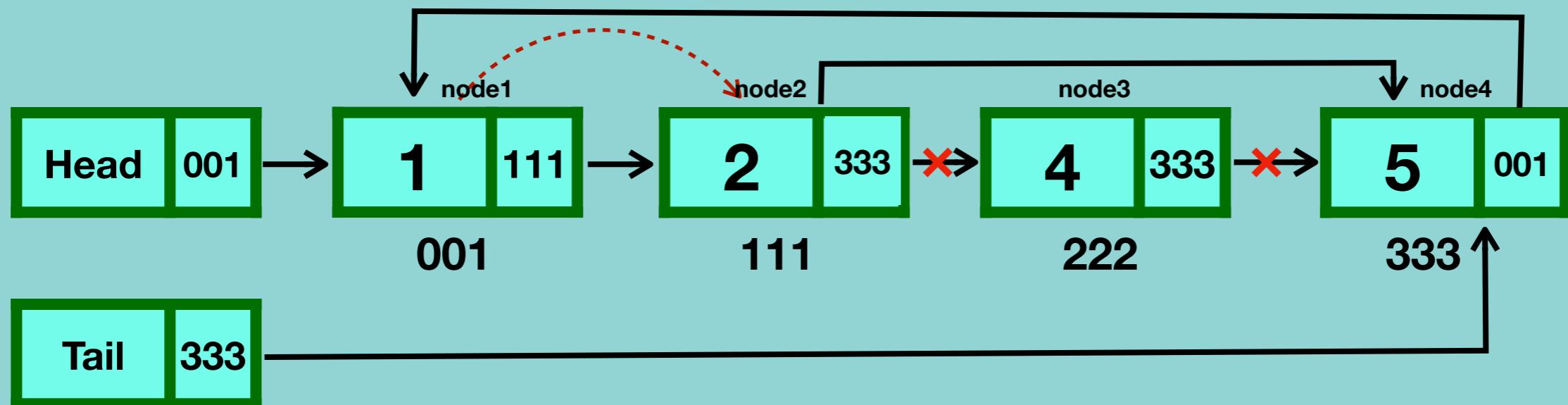
- Deleting the first node
- Deleting any given node
- Deleting the last node

### Case 2 - more than one node



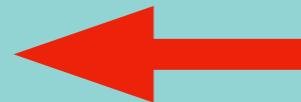
# Circular Singly Linked list - Deletion

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

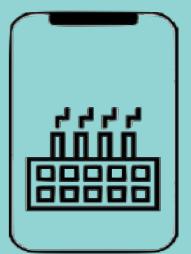
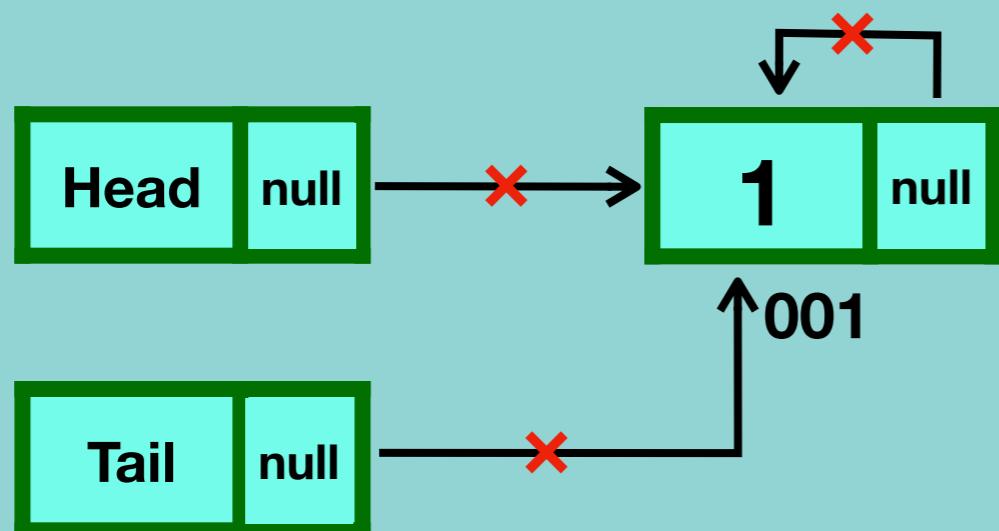


# Circular Singly Linked list - Deletion

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

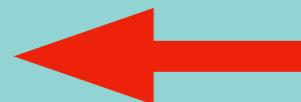


Case 1 - one node

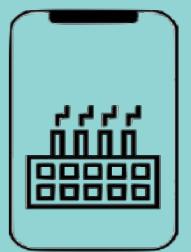
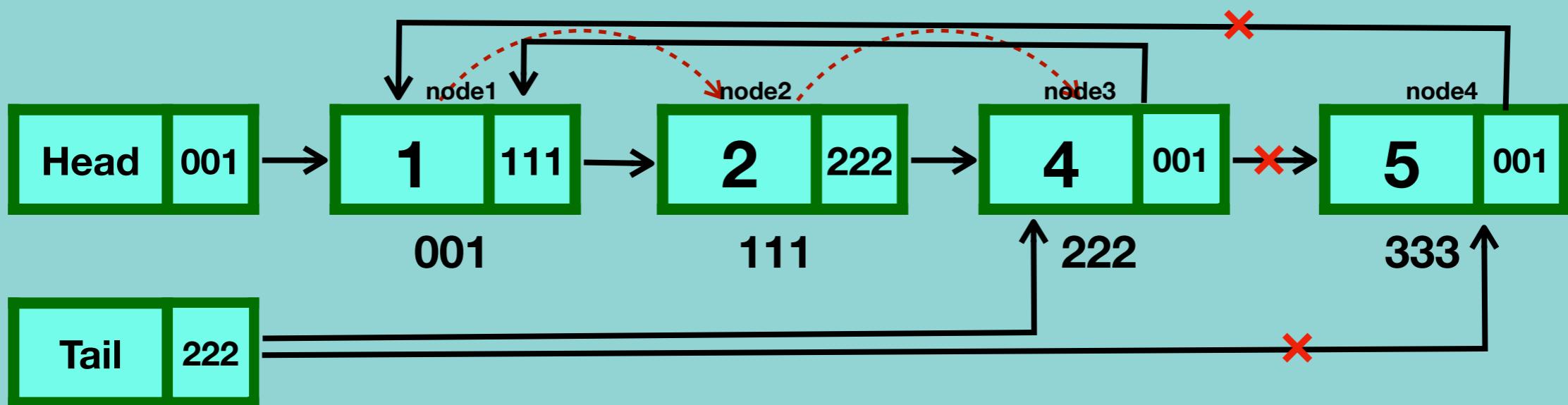


# Circular Singly Linked list - Deletion

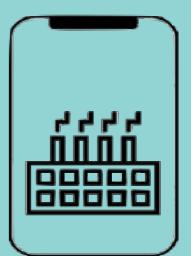
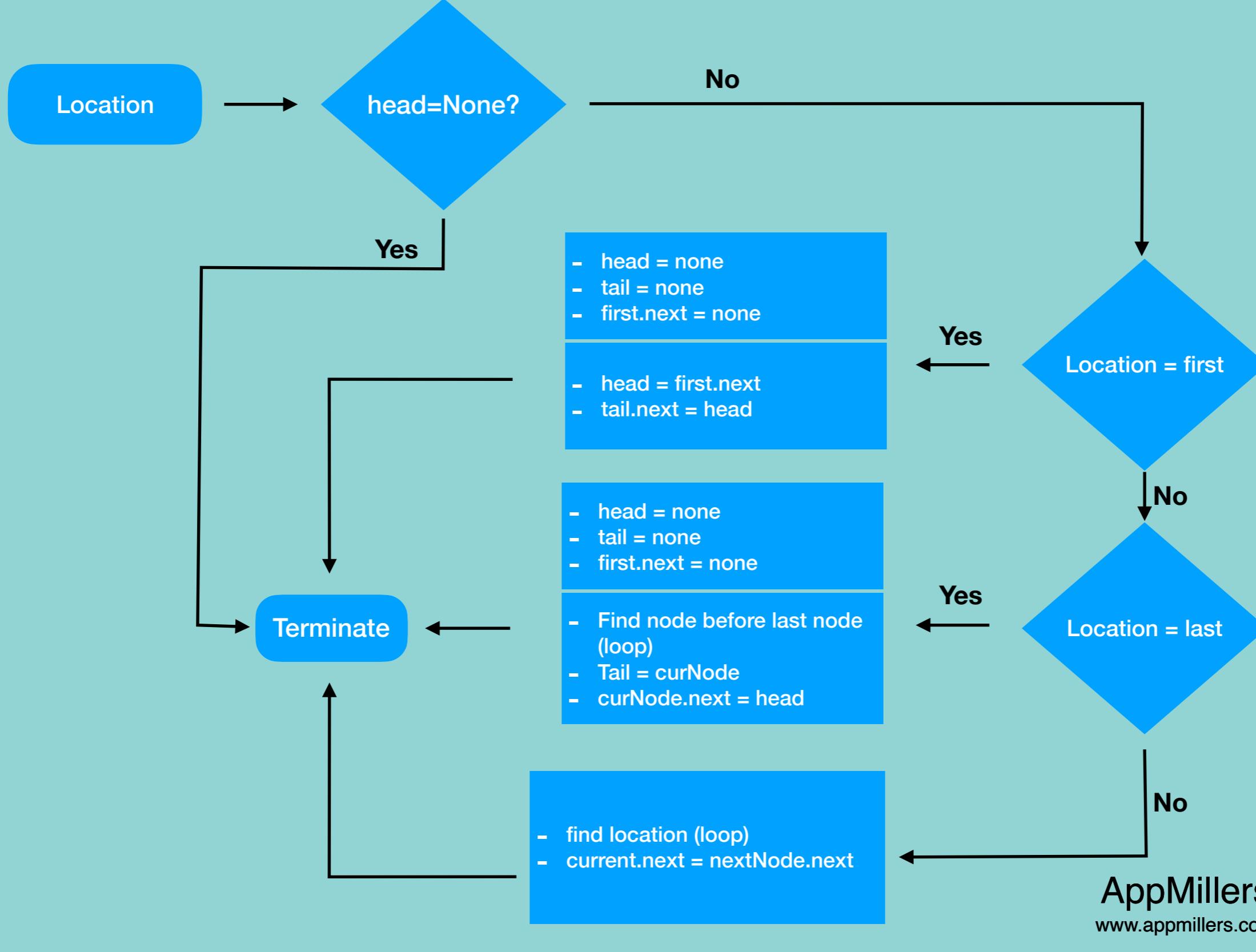
- Deleting the first node
- Deleting any given node
- Deleting the last node



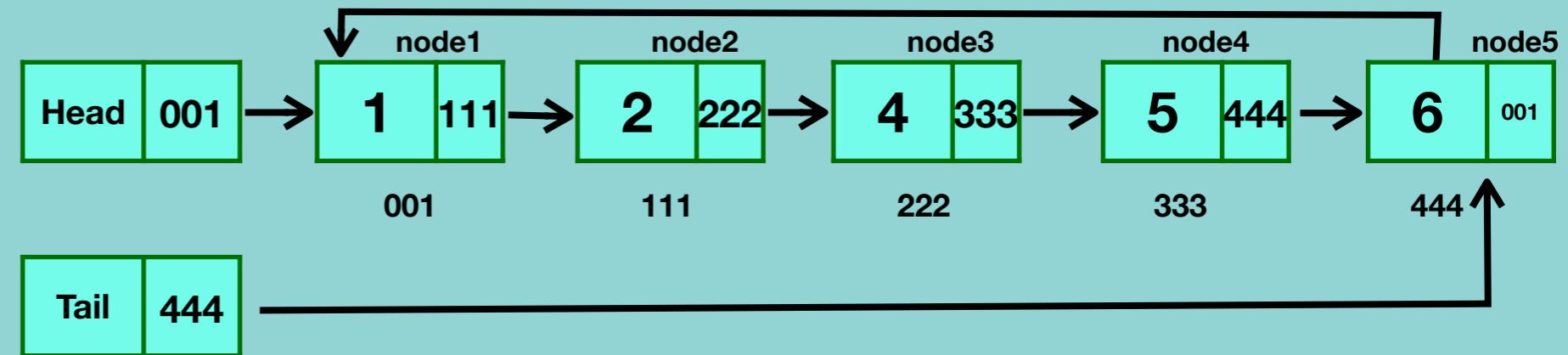
## Case 2 - more than one node



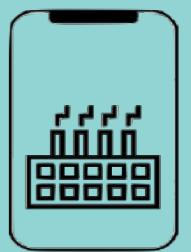
# Circular Singly Linked list Deletion Algorithm



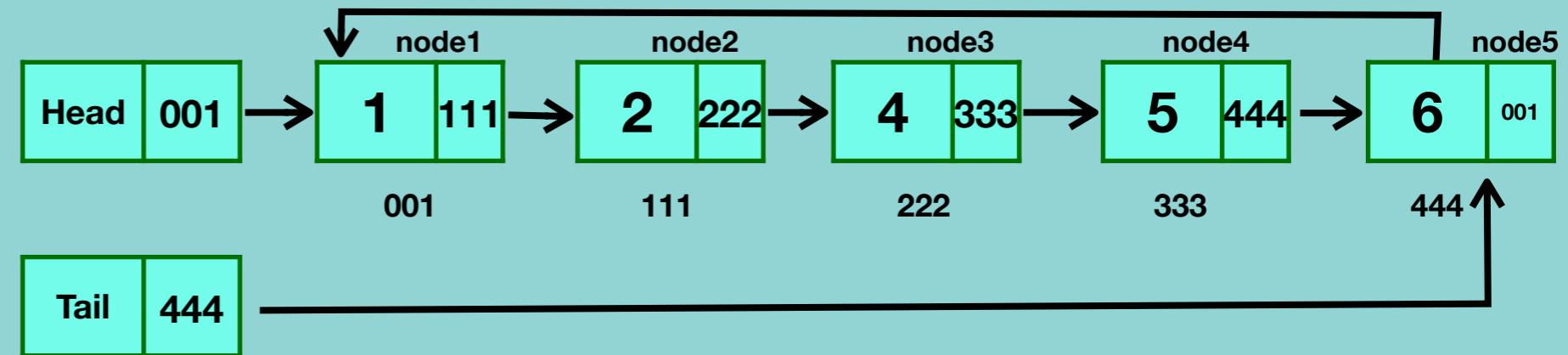
# Circular Singly Linked list - Deletion Algorithm



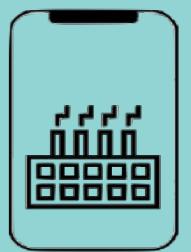
```
deleteNode(head, location):
    if head does not exist:
        return error //Linked list does not exist
    if location = firstNode's location
        if this is the only node in the list
            head=tail=node.next=null
        else
            head=head.next
            tail.next = head
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



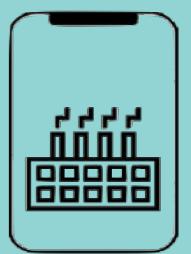
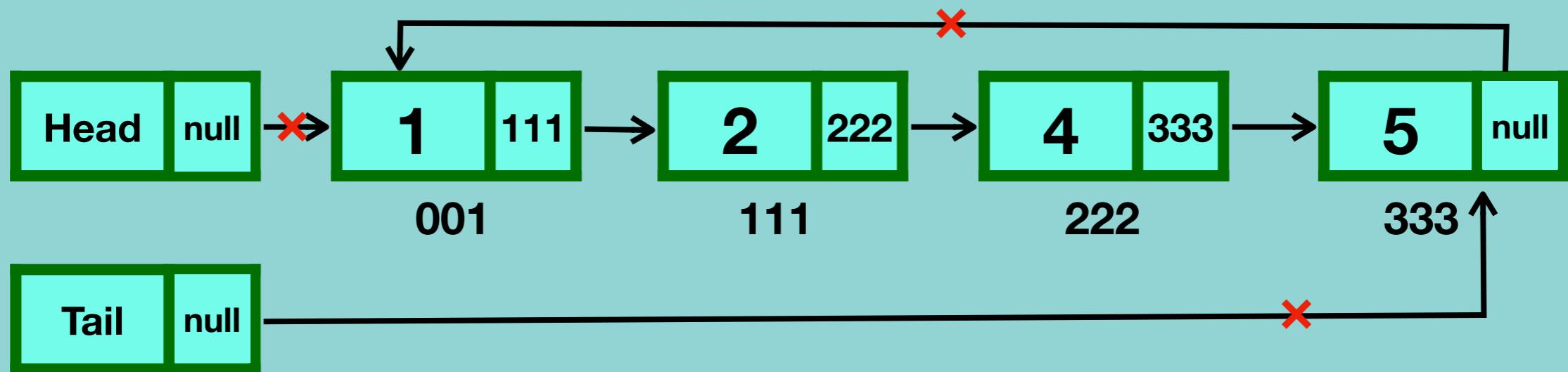
# Circular Singly Linked list - Deletion Algorithm



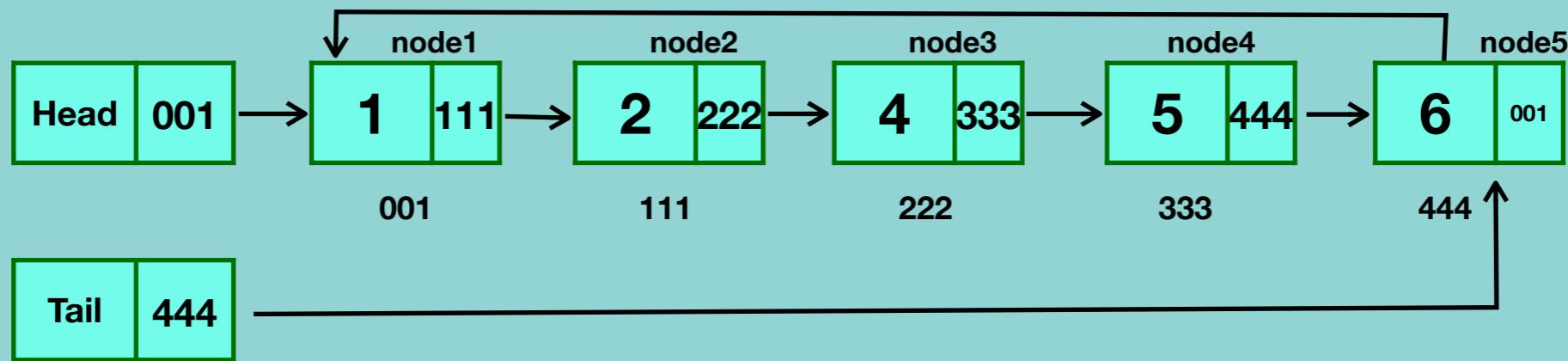
```
deleteNode(head, location):
    if head does not exist:
        return error //Linked list does not exist
    if location = firstNode's location
        if this is the only node in the list
            head=tail=node.next=null
        else
            head=head.next
            tail.next = head
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



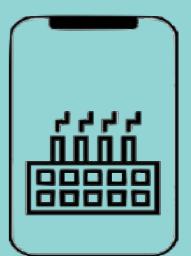
# Delete entire Circular Singly Linked list



# Delete entire Circular Singly Linked List

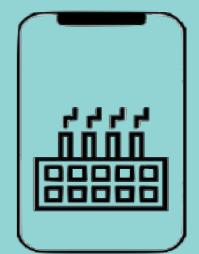
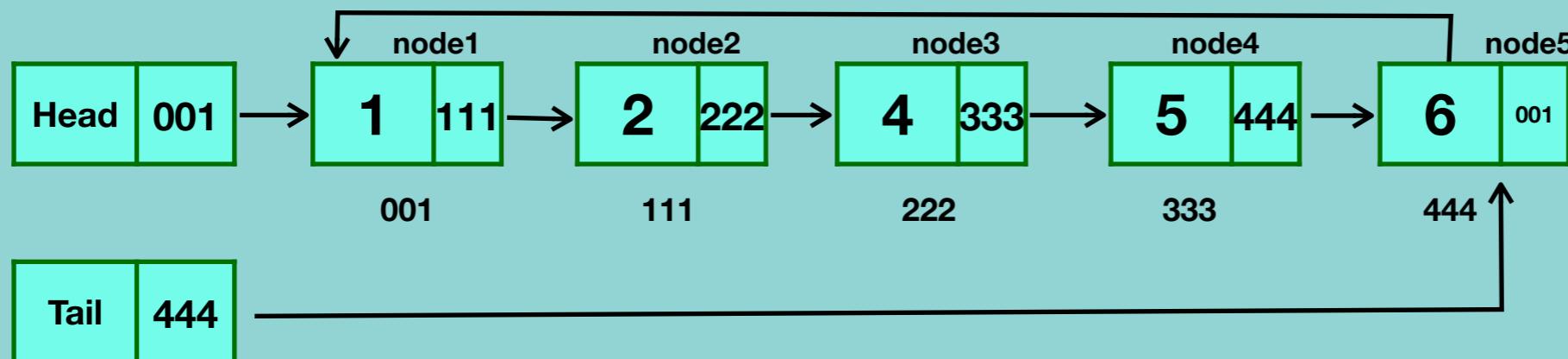


```
deleteLinkedList(head, tail):  
    head = null  
    tail.next = null  
    Tail = null
```



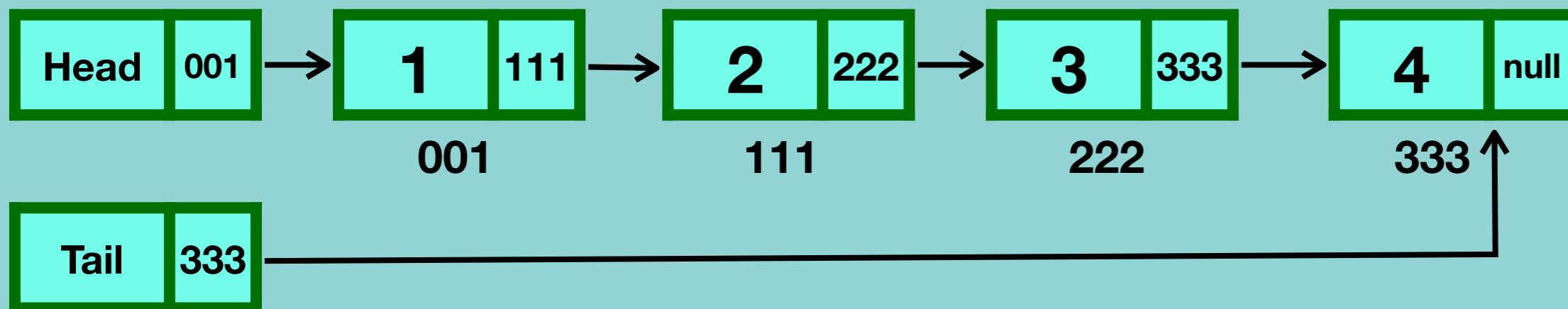
# Time and Space complexity of Circular Singly Linked List

	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)

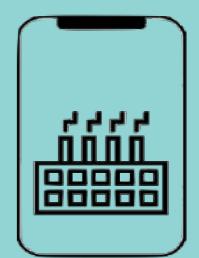
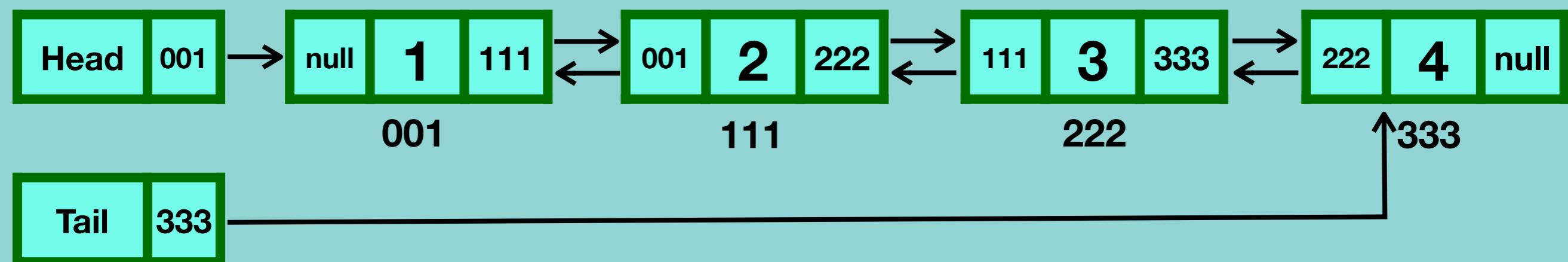


# Doubly Linked List

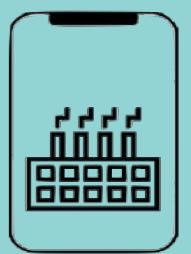
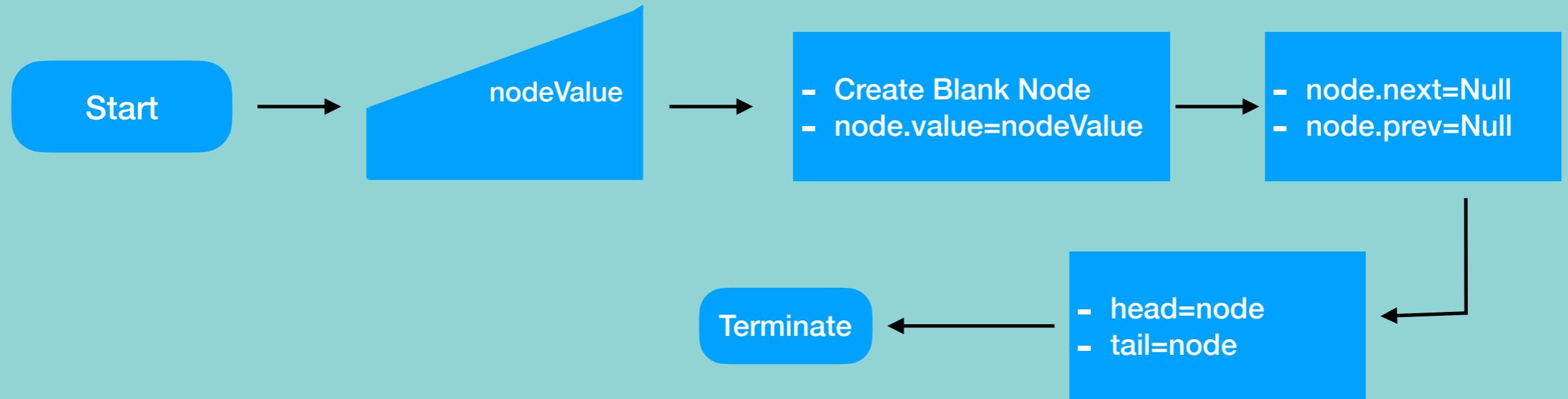
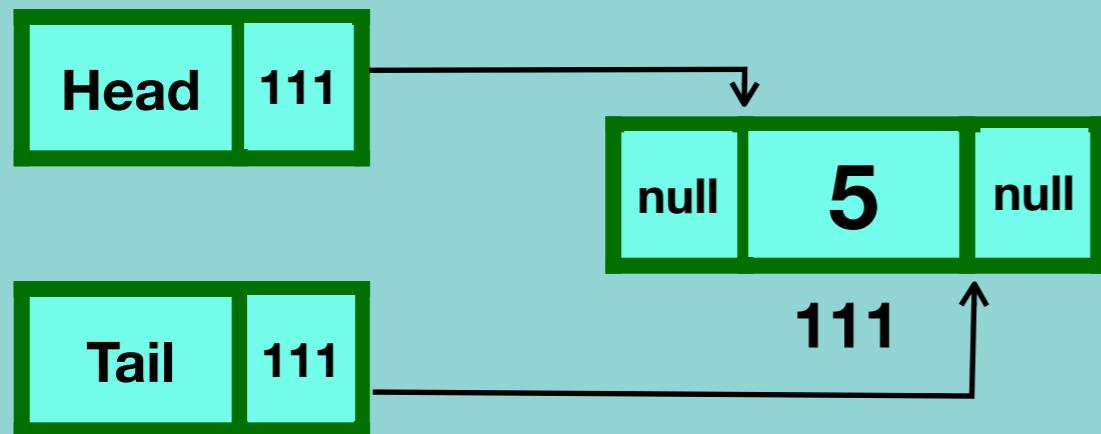
## Singly Linked List



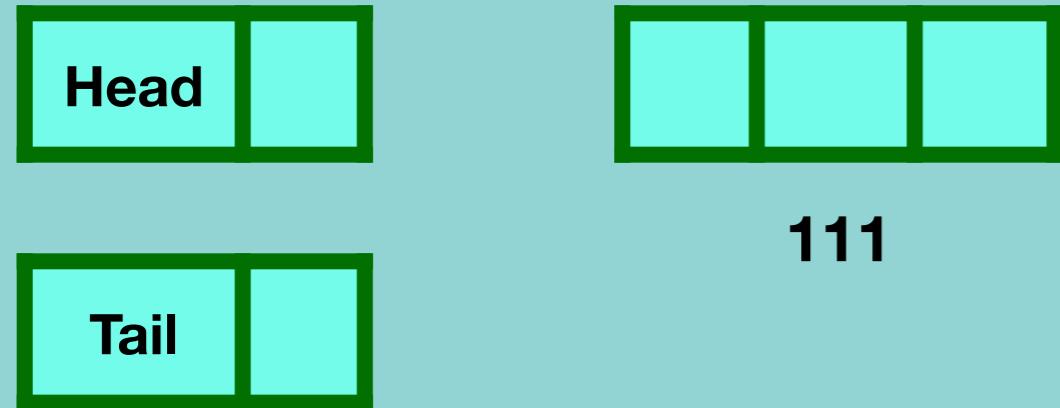
## Doubly Linked List



# Creation of Doubly Linked List



# Doubly Linked List

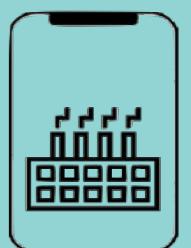


```
createDoublyLinkedList(nodeValue):
```

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

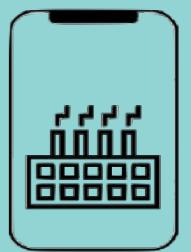
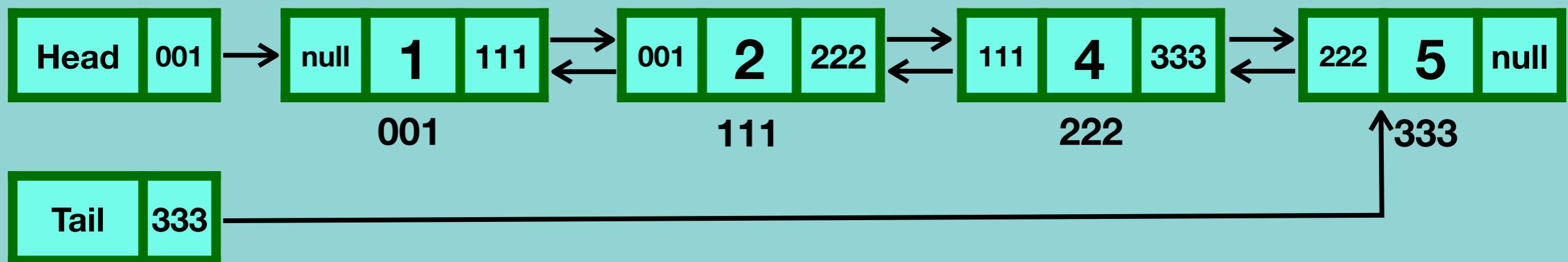
**Time complexity : O(1)**

**Space complexity : O(1)**



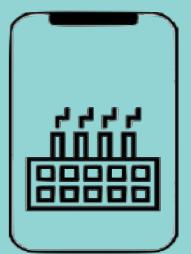
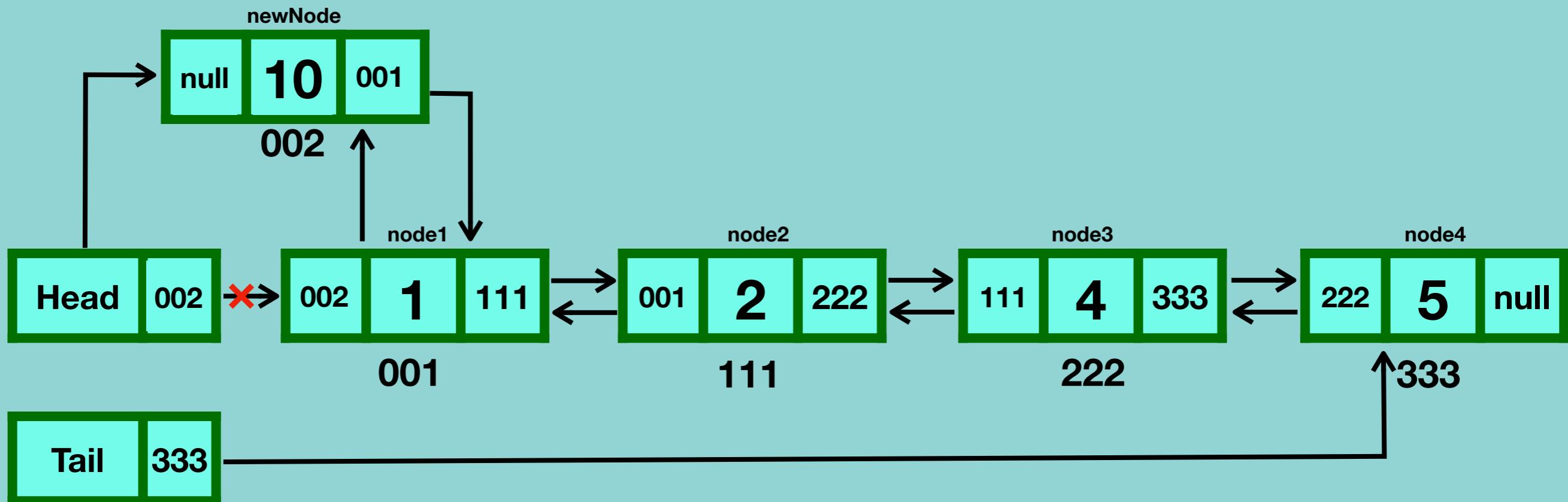
# Doubly Linked List - Insertion

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



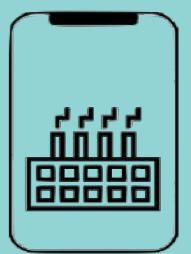
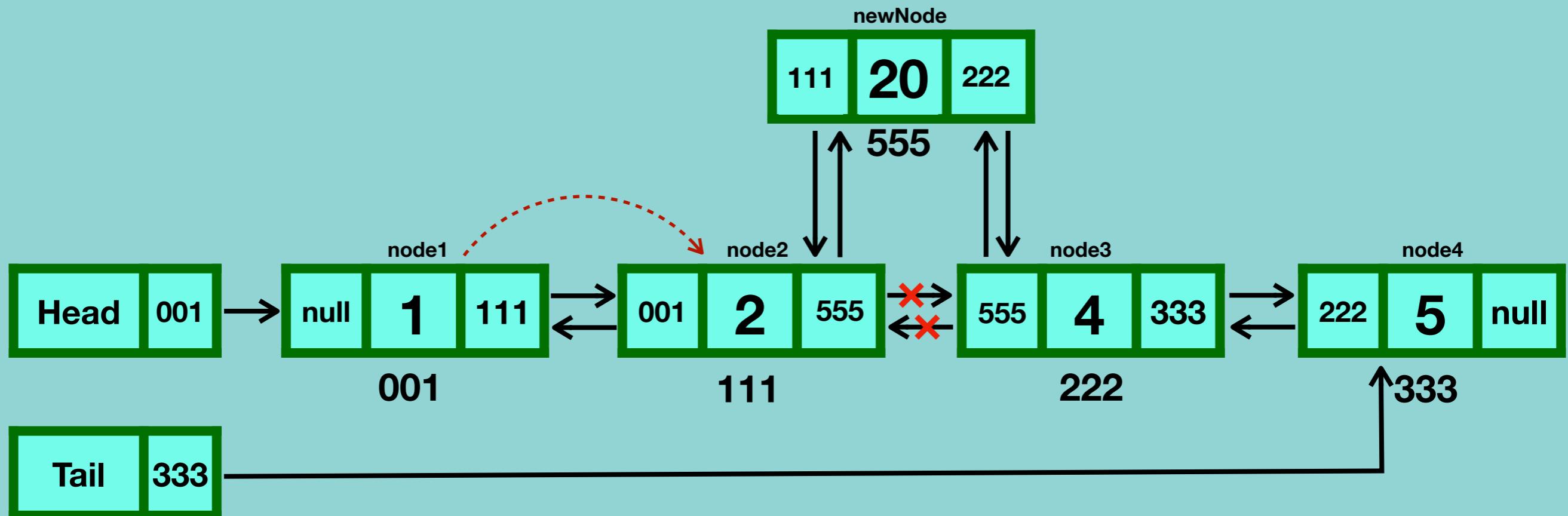
# Doubly Linked List - Insertion

- Insert at the beginning of linked list



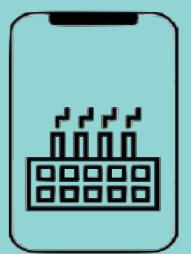
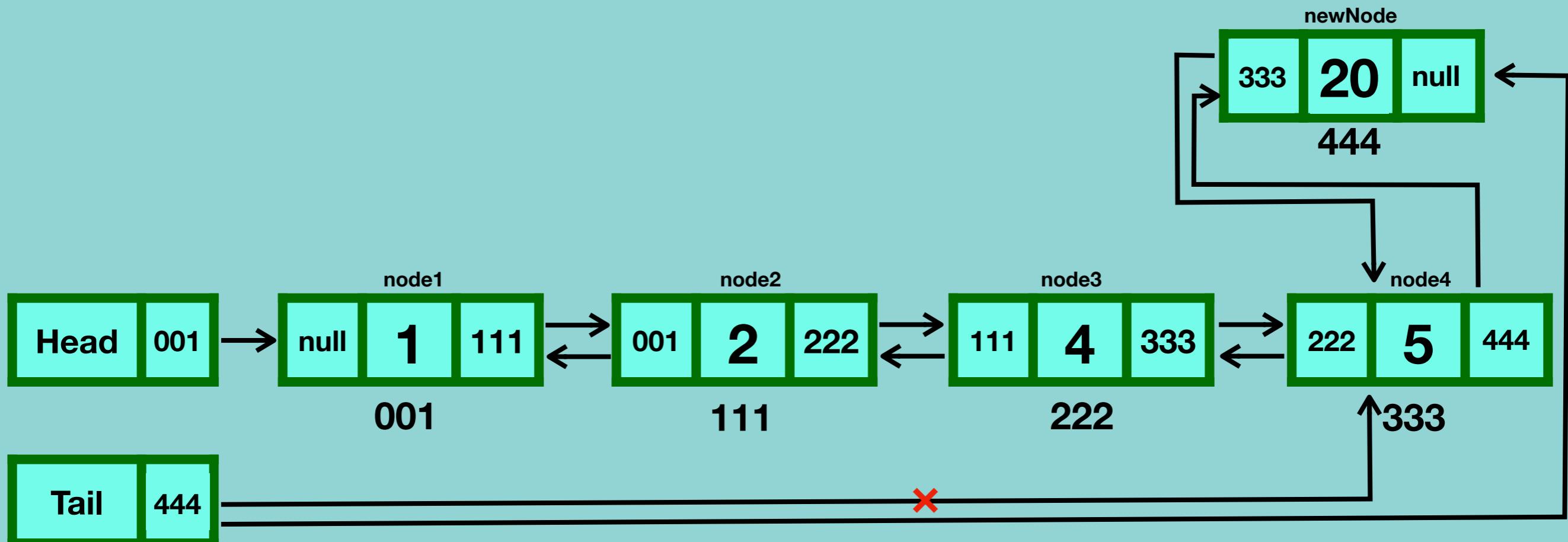
# Doubly Linked List - Insertion

- Insert at the specified location of linked list

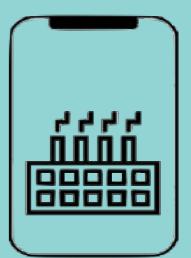
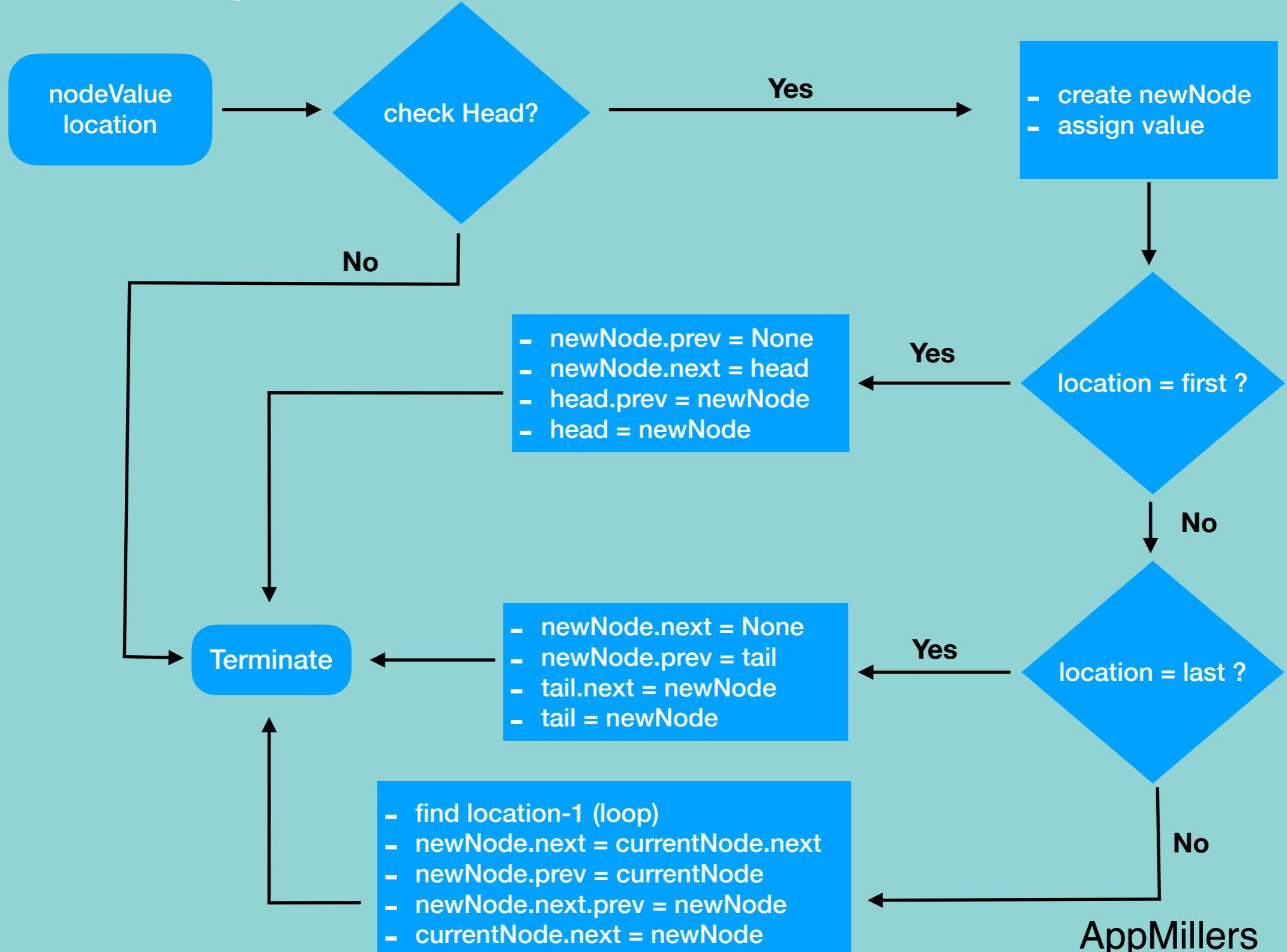


# Doubly Linked List - Insertion

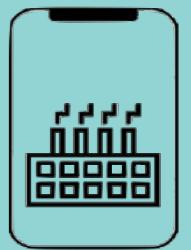
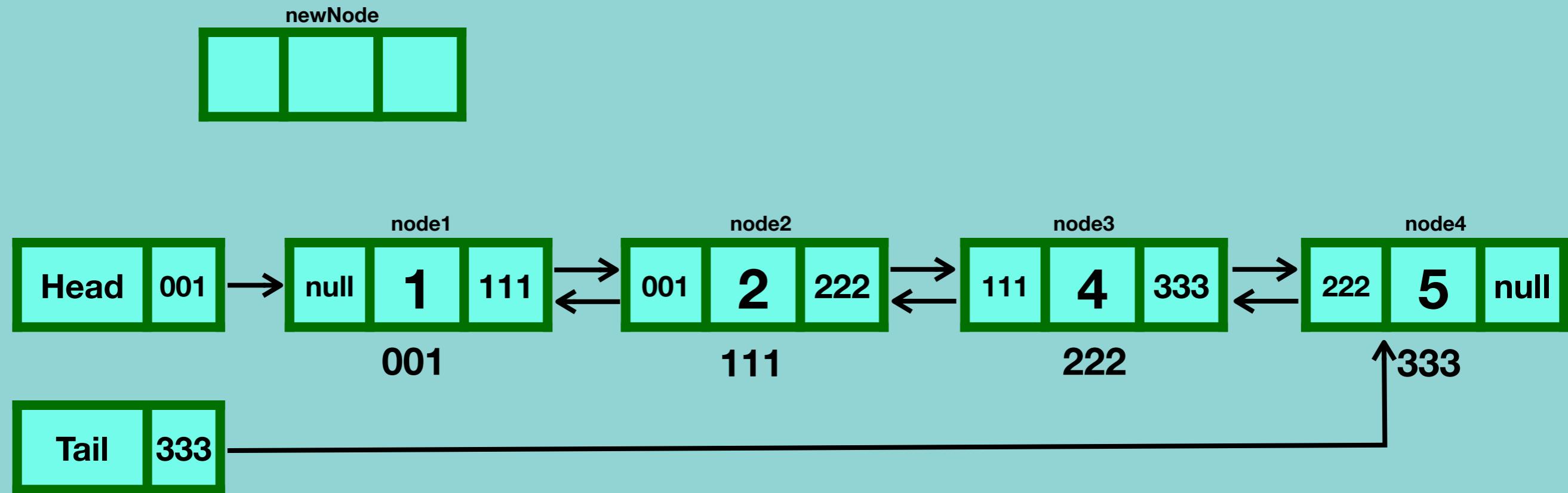
- Insert at the end of linked list



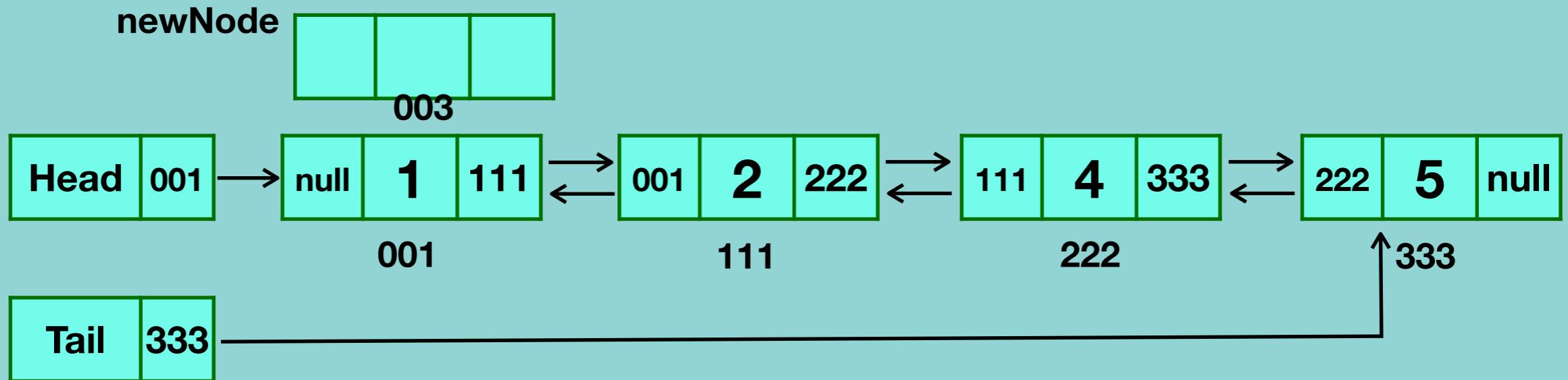
# Insertion Algorithm - Doubly linked list



# Insertion in Doubly Linked List



# Insertion Algorithm - Doubly Linked List

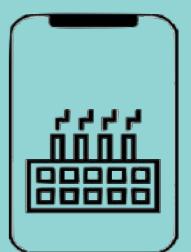


```
insertInDoublyLinkedList(head, nodeValue, location):
```

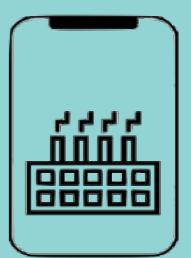
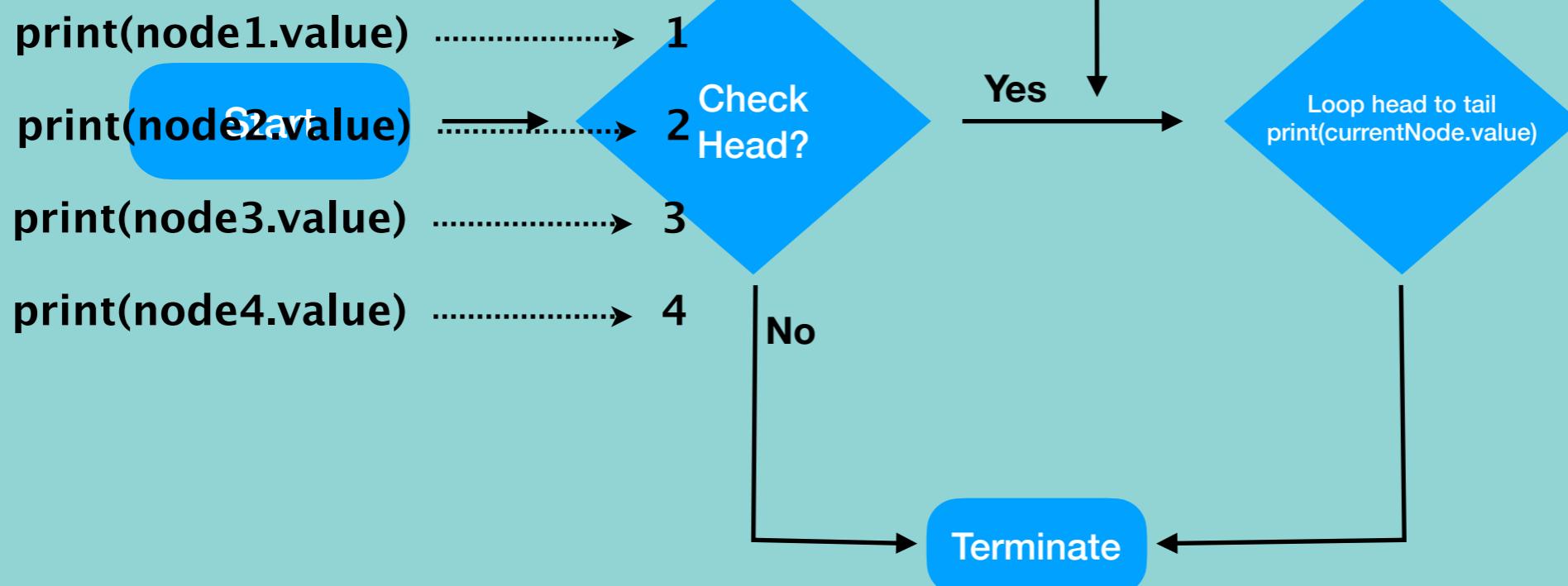
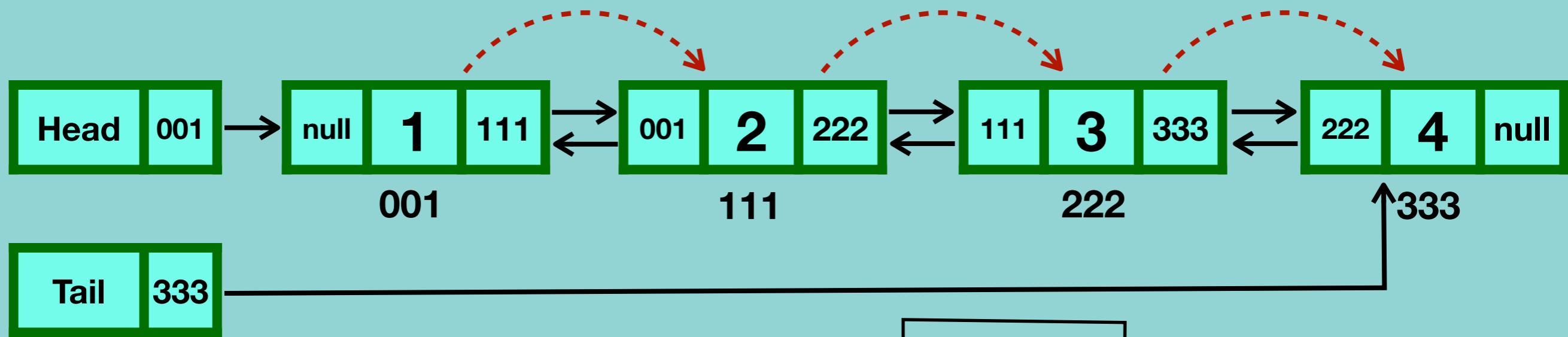
```
if head does not exist:  
    return error //Linked list does not exist  
else  
    create blank node (newNode)  
    newNode.value = nodeValue  
if location = firstNode's location  
    newNode.prev = null  
    newNode.next = head  
    head.prev = newNode  
    head = newNode  
else if location = 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)

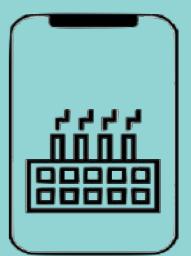
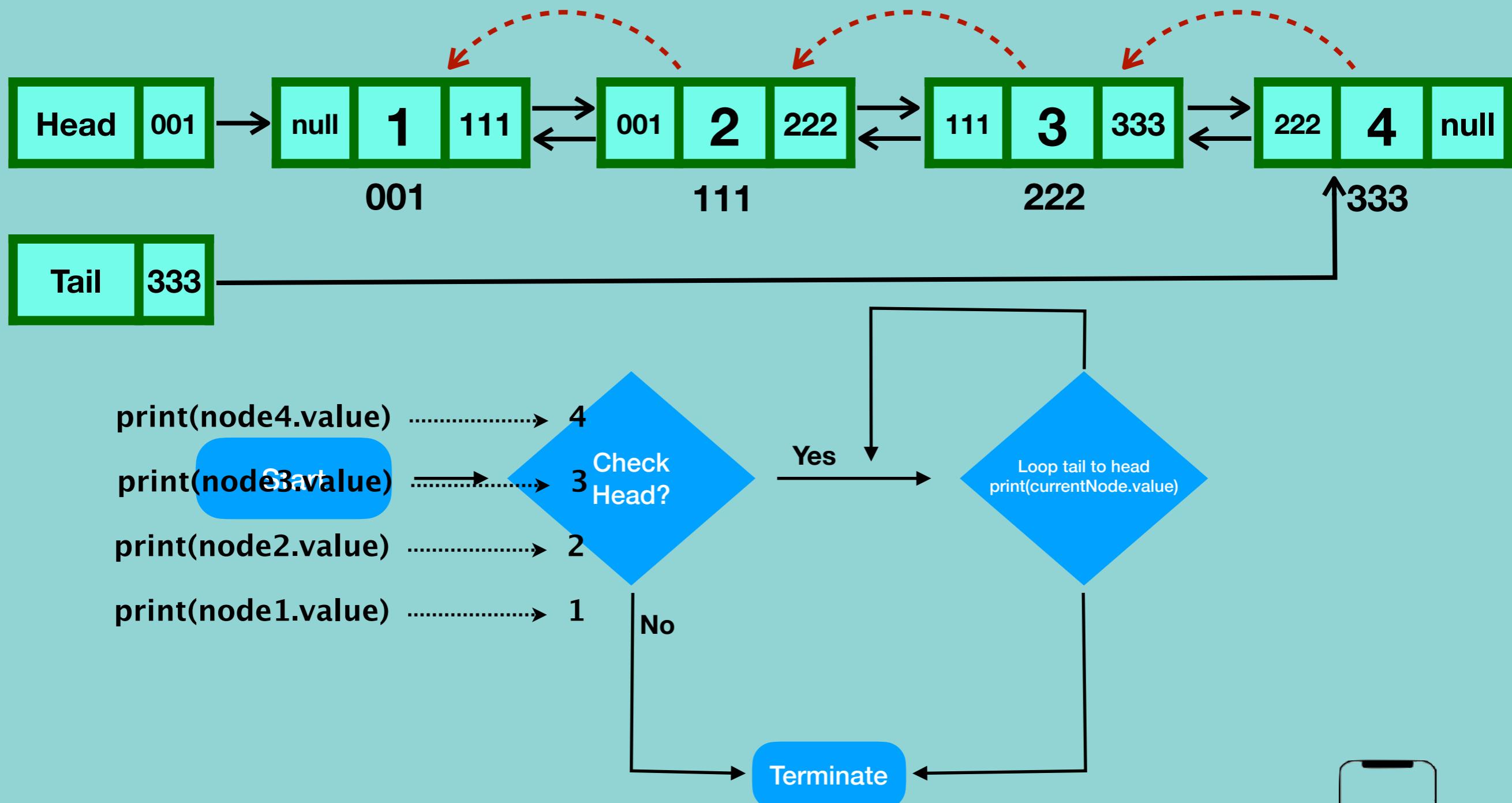
Space complexity : O(1)



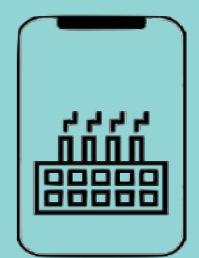
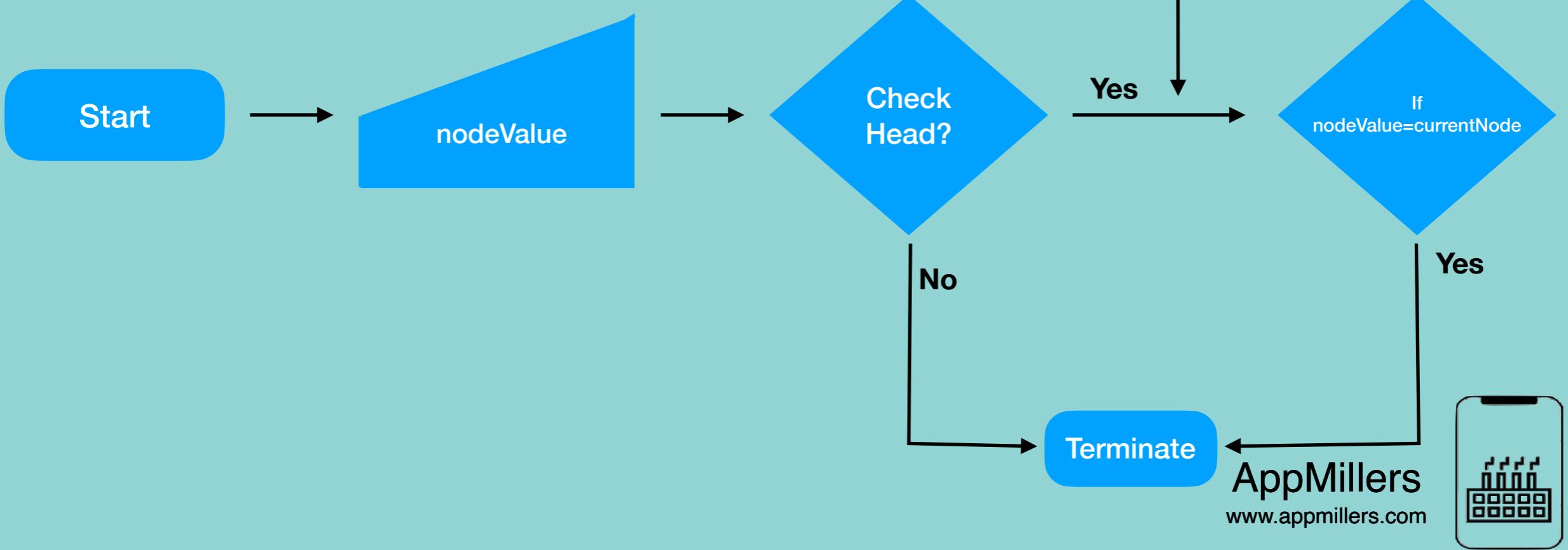
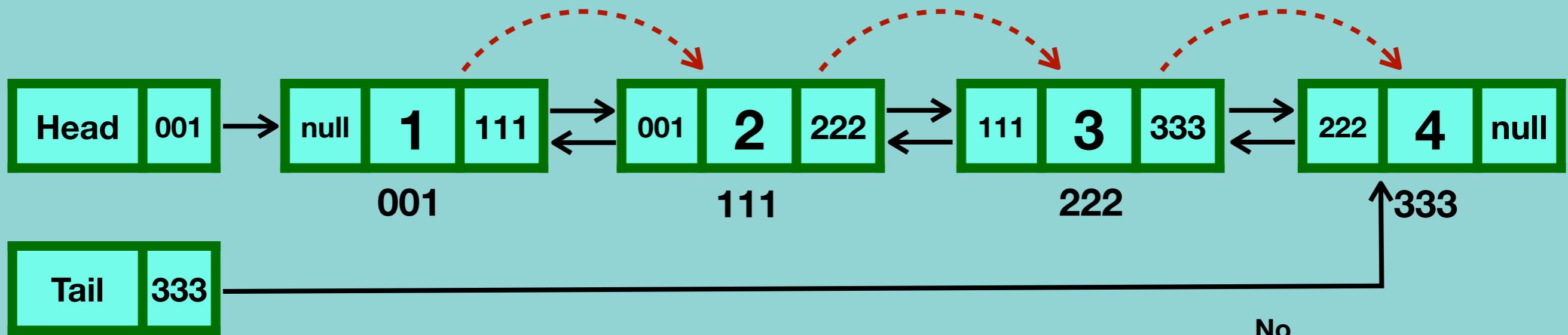
# Doubly Linked List - Traversal



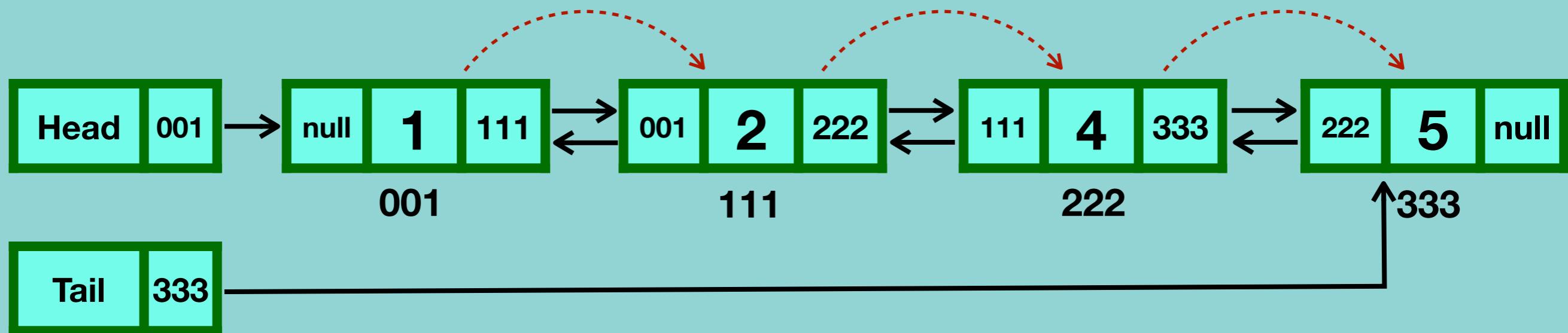
# Doubly Linked List - Reverse Traversal



# Doubly Linked List - Search



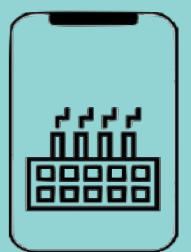
# Traversal in doubly linked list



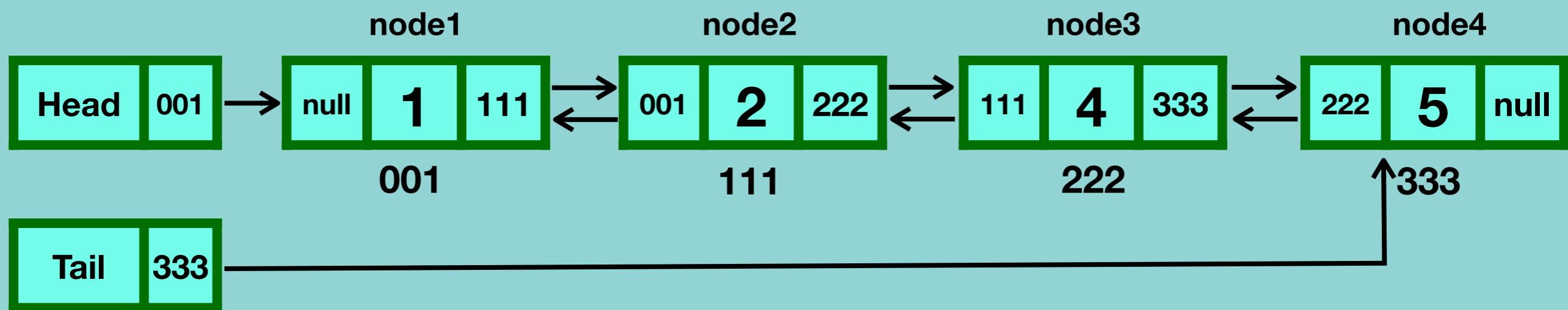
```
traversalDoublyLinkedList():
    if head == null:
        return //There is not any node in this list
    loop head to tail:
        print(currentNode.value)
```

Time complexity : O(n)

Space complexity : O(1)



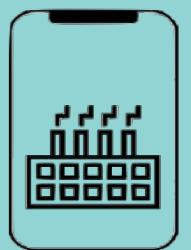
# Reverse Traversal in doubly linked list



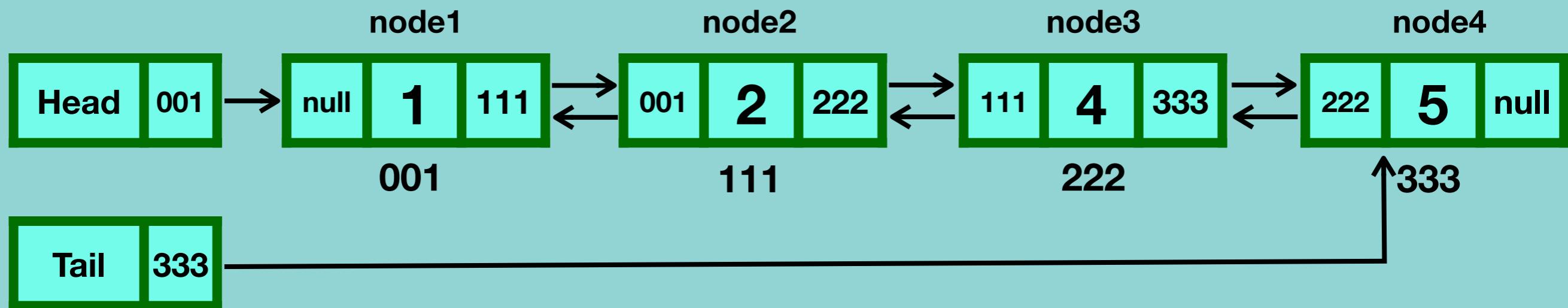
```
reverseTraversalDoublyLinkedList(head):
    if head == null:
        return //There is not any node in this list
    loop tail to head:
        print(currentNode.value)
```

Time complexity : O(1)

Space complexity : O(1)



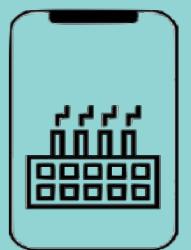
# Searching a node in doubly linked list



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

Time complexity :  $O(n)$

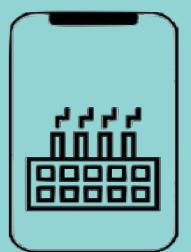
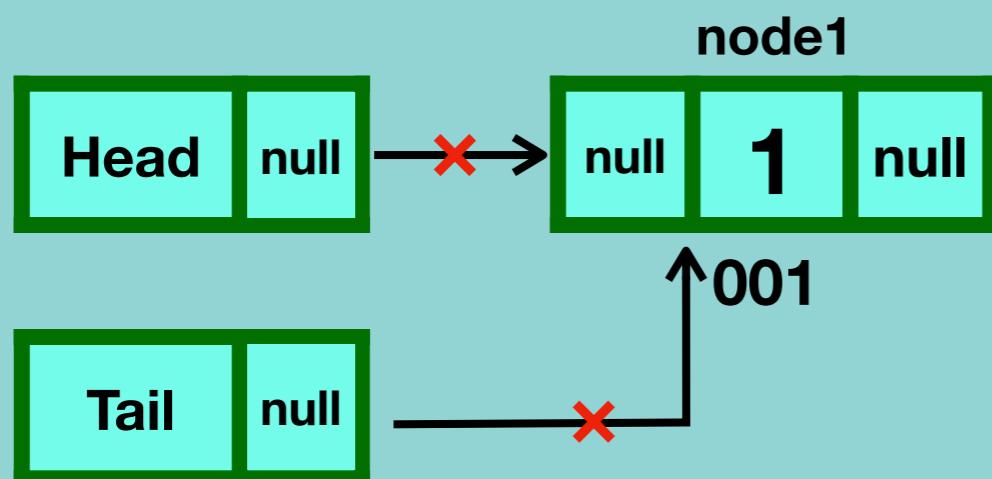
Space complexity :  $O(1)$



## Doubly Linked list - Deletion

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

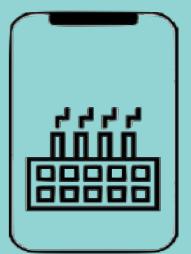
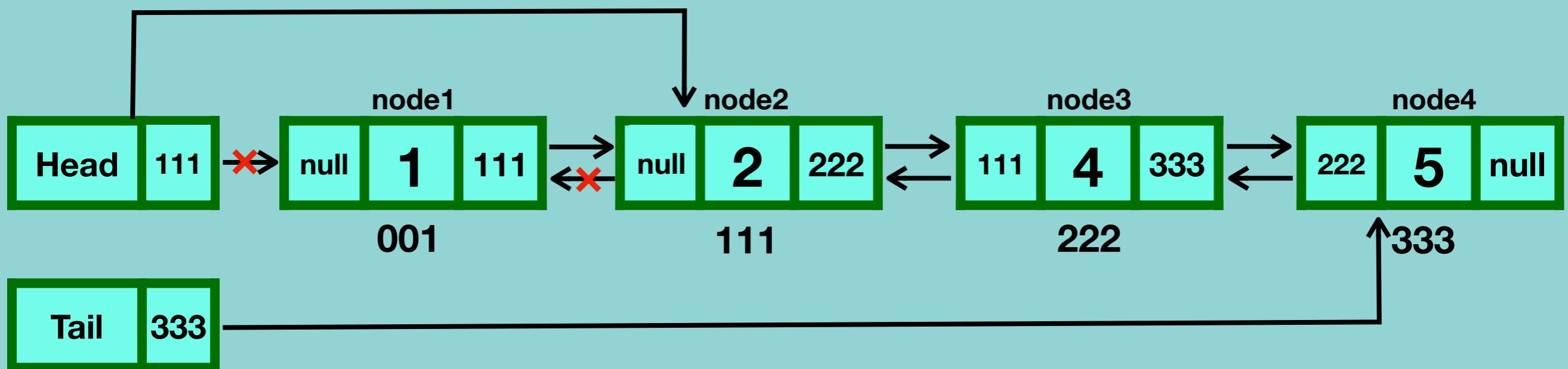
### Case 1 - one node



## Doubly Linked list - Deletion

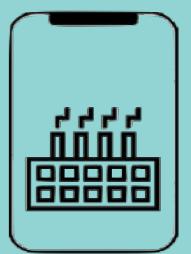
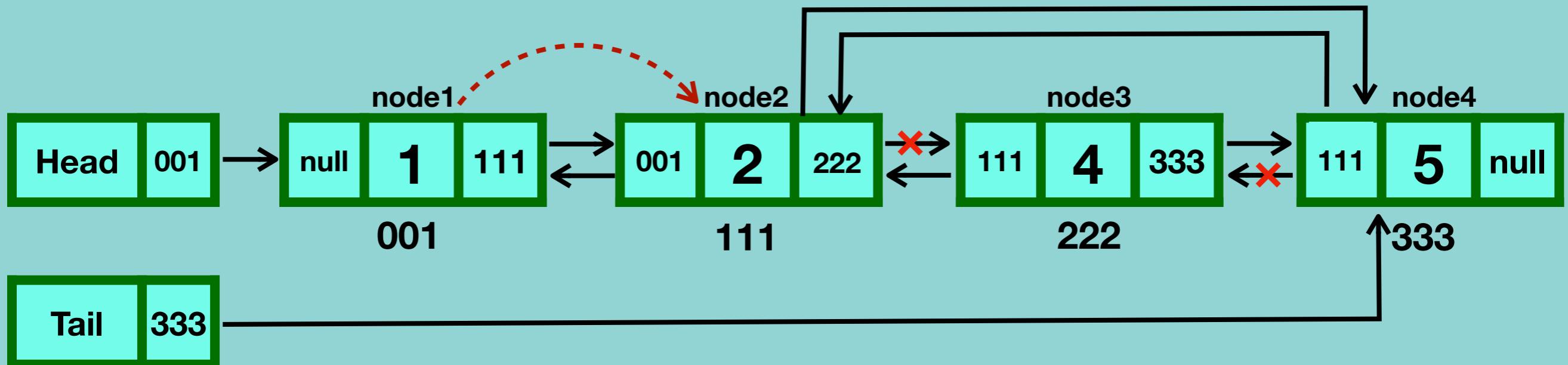
- Deleting the first node
- Deleting any given node
- Deleting the last node

### Case 2 - more than one node



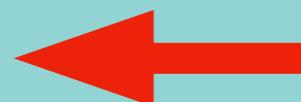
## Doubly Linked list - Deletion

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

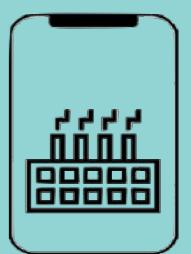
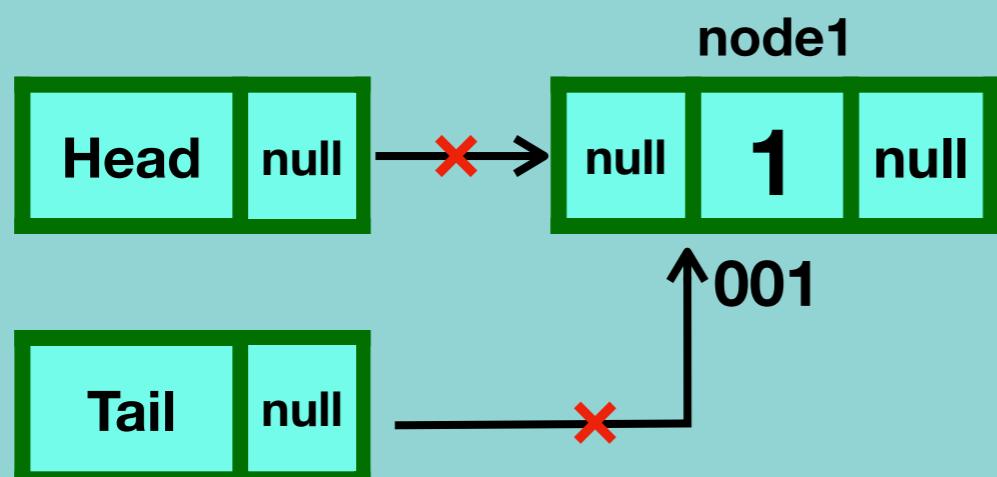


## Doubly Linked list - Deletion

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

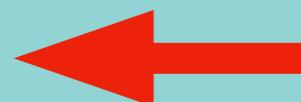


### Case 1 - one node

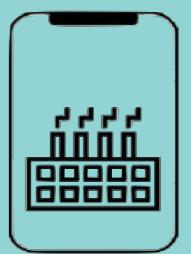
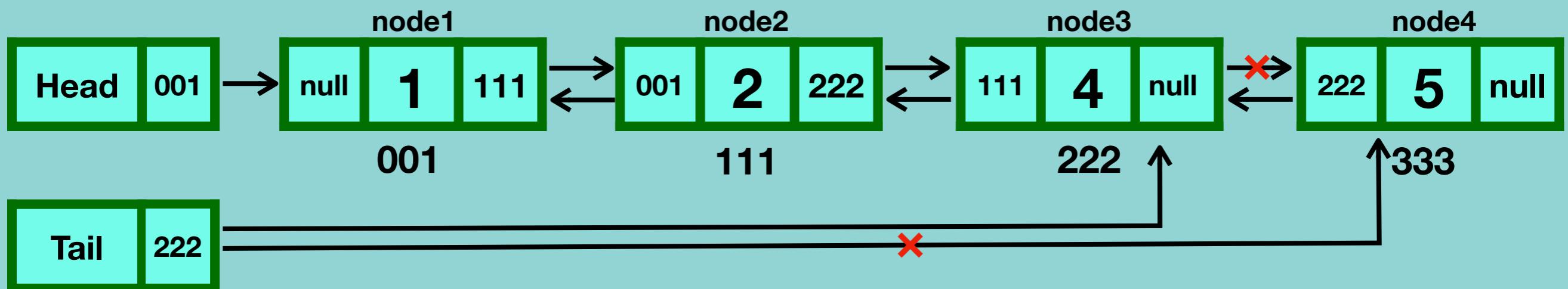


# Doubly Linked list - Deletion

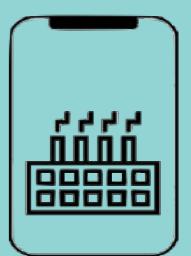
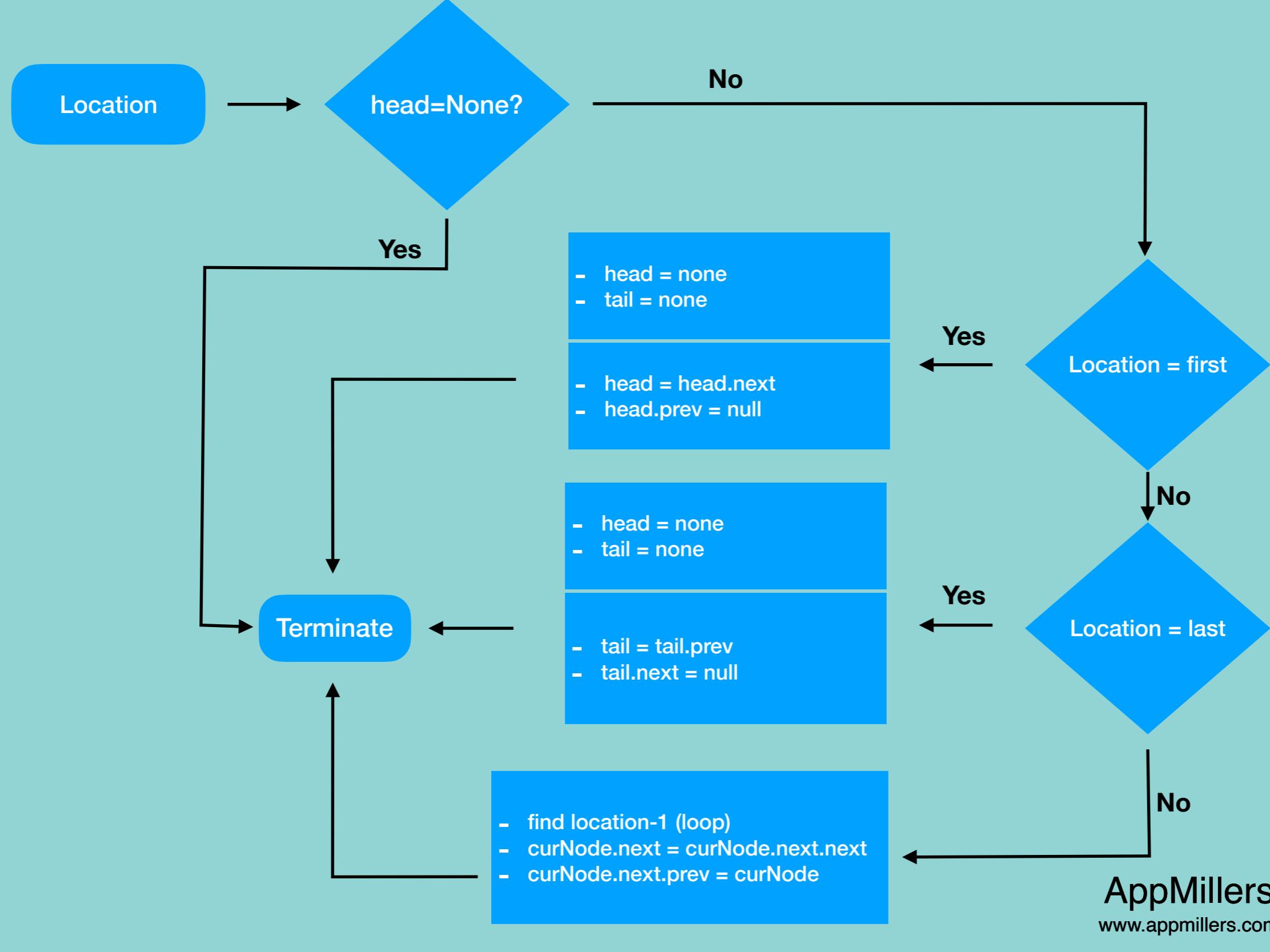
- Deleting the first node
- Deleting any given node
- Deleting the last node



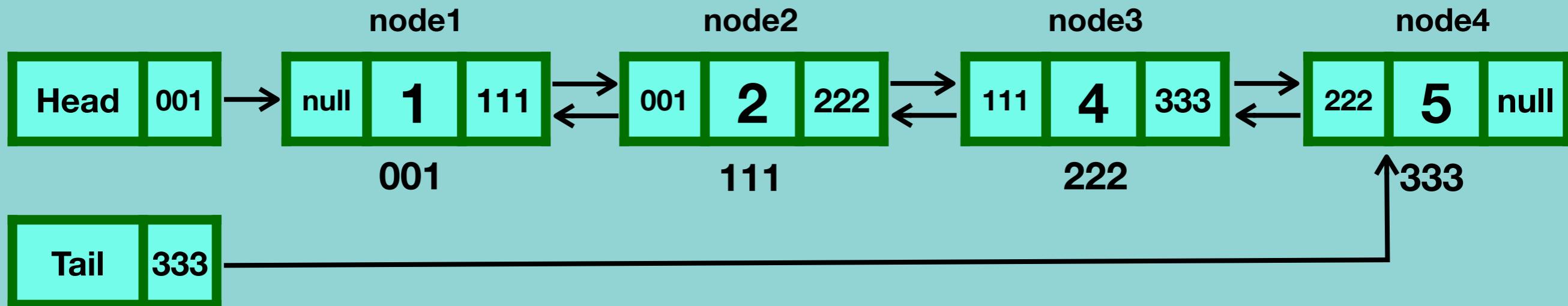
## Case 2 - more than one node



# Doubly Linked list Deletion Algorithm



# Deleting a node in doubly linked list- Algorithm

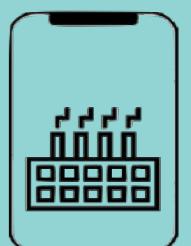


```
deleteNode(head, location):
```

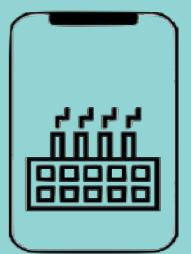
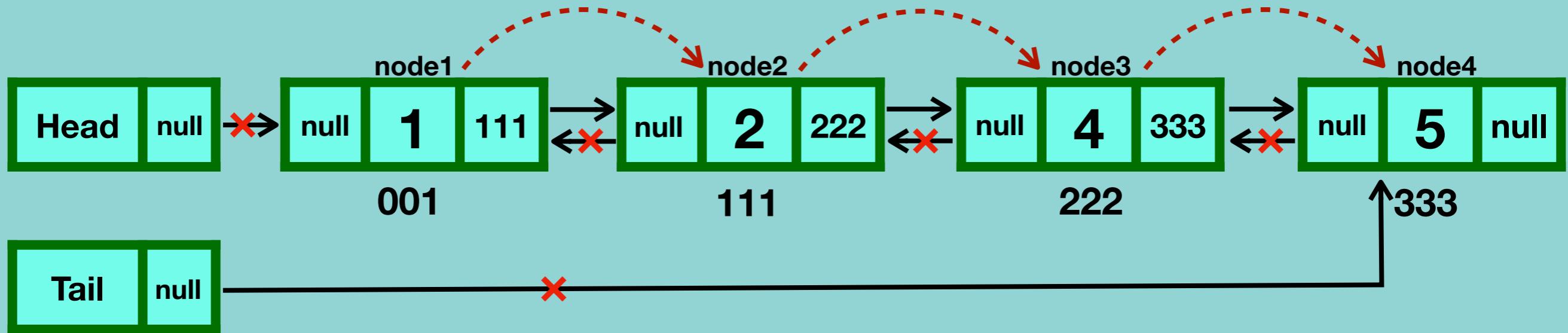
```
if head does not exist:  
    return error //Linked list does not exist  
if location = firstNode's location  
    if this is the only node in the list  
        head=tail=null  
    else  
        head=head.next  
        head.prev = null  
else if location = lastNode's location  
    if this is 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)

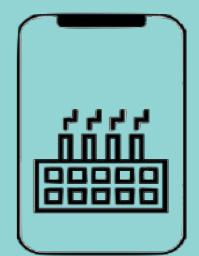
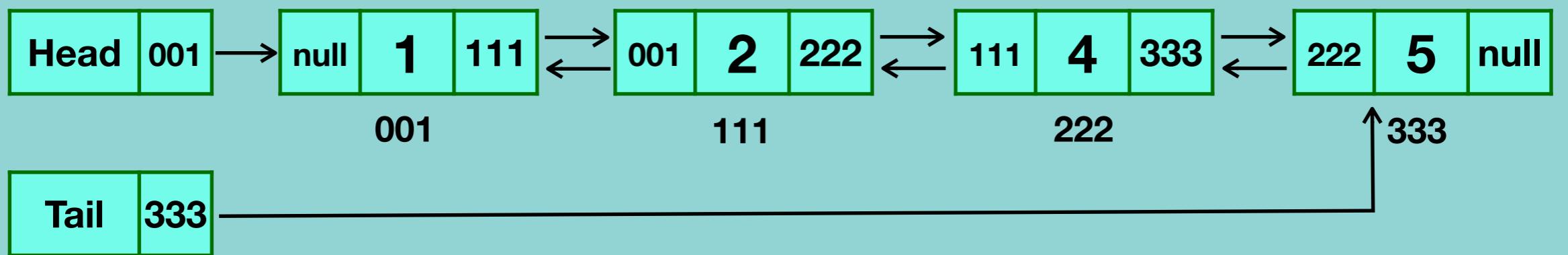


# Deleting entire doubly linked list- Algorithm

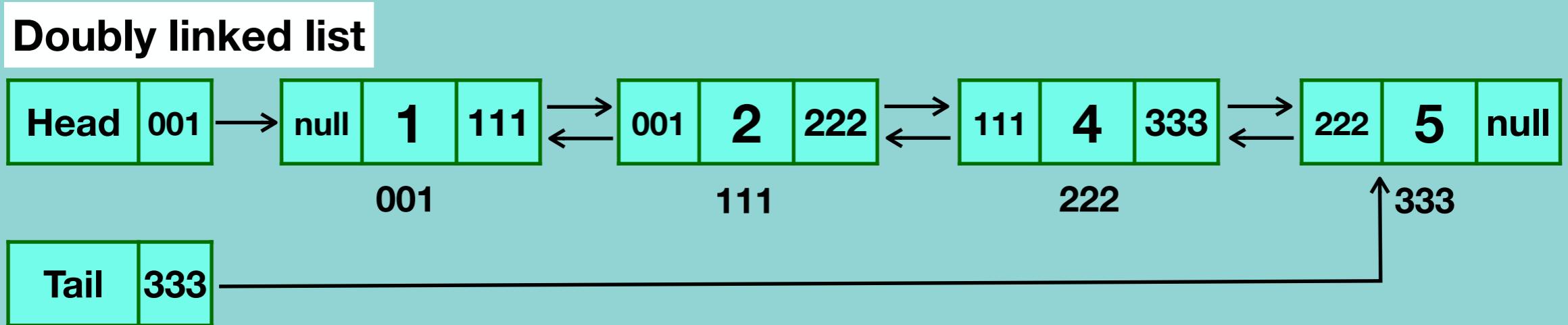


# Time and Space complexity of doubly linked list

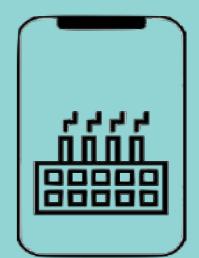
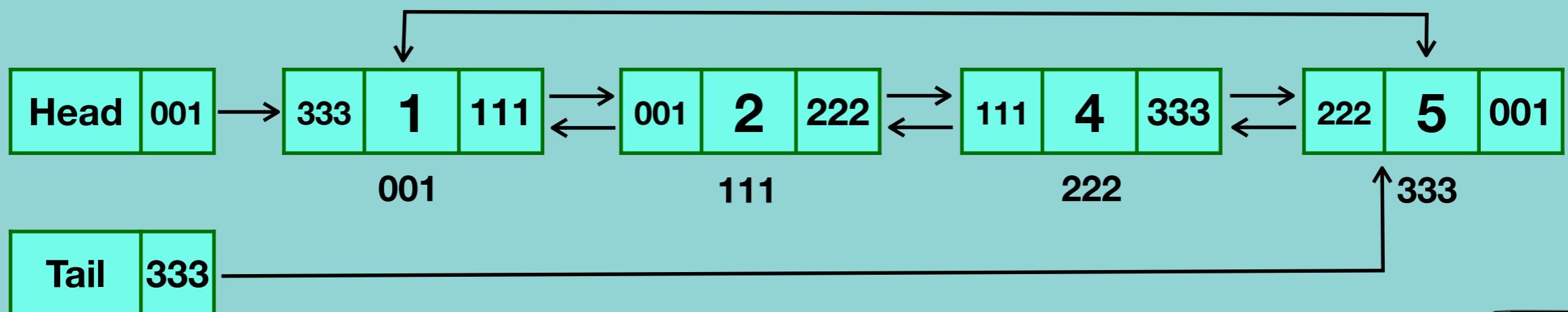
	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing (forward, backward)	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(n)	O(1)



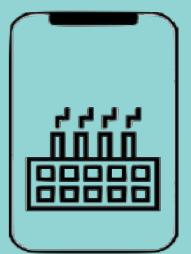
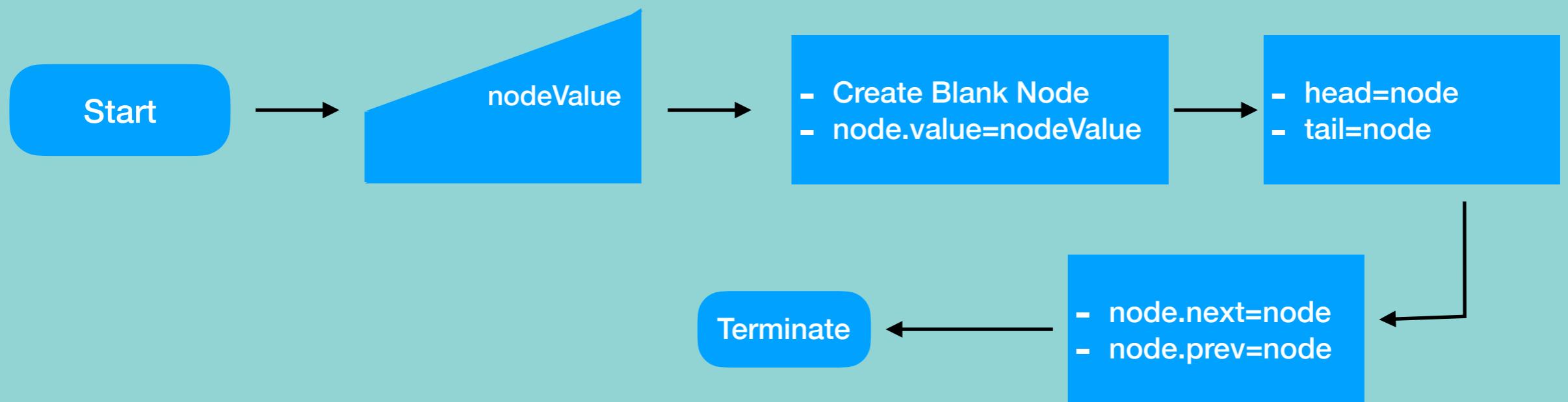
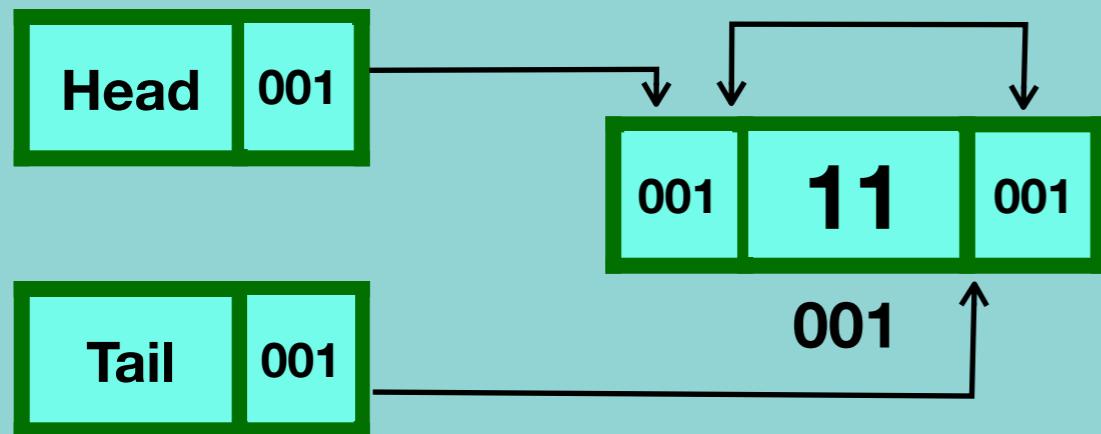
# Circular doubly linked list



## Circular Doubly linked list

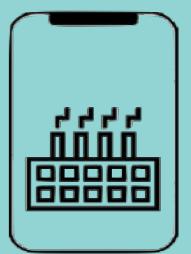
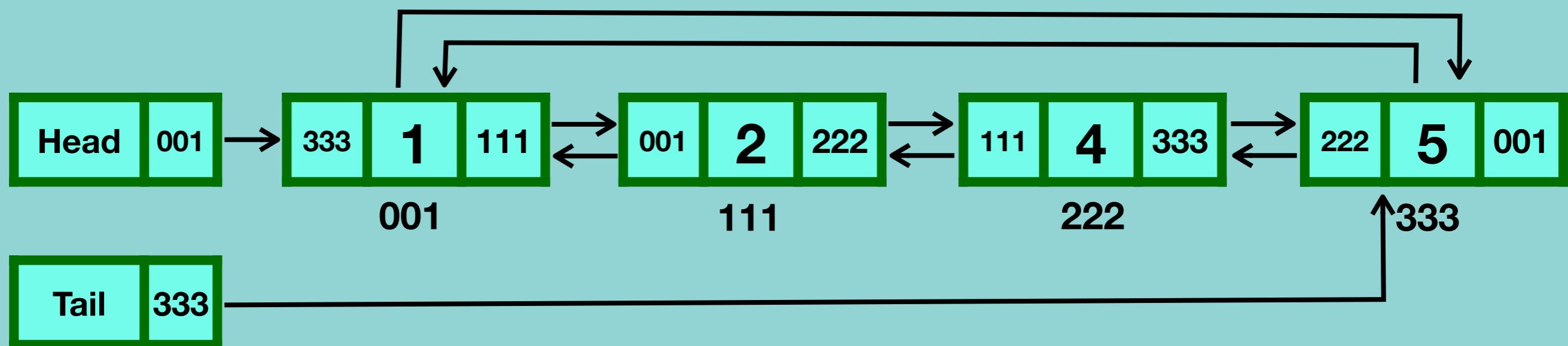


# Creation of Circular Doubly Linked List



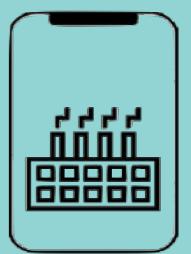
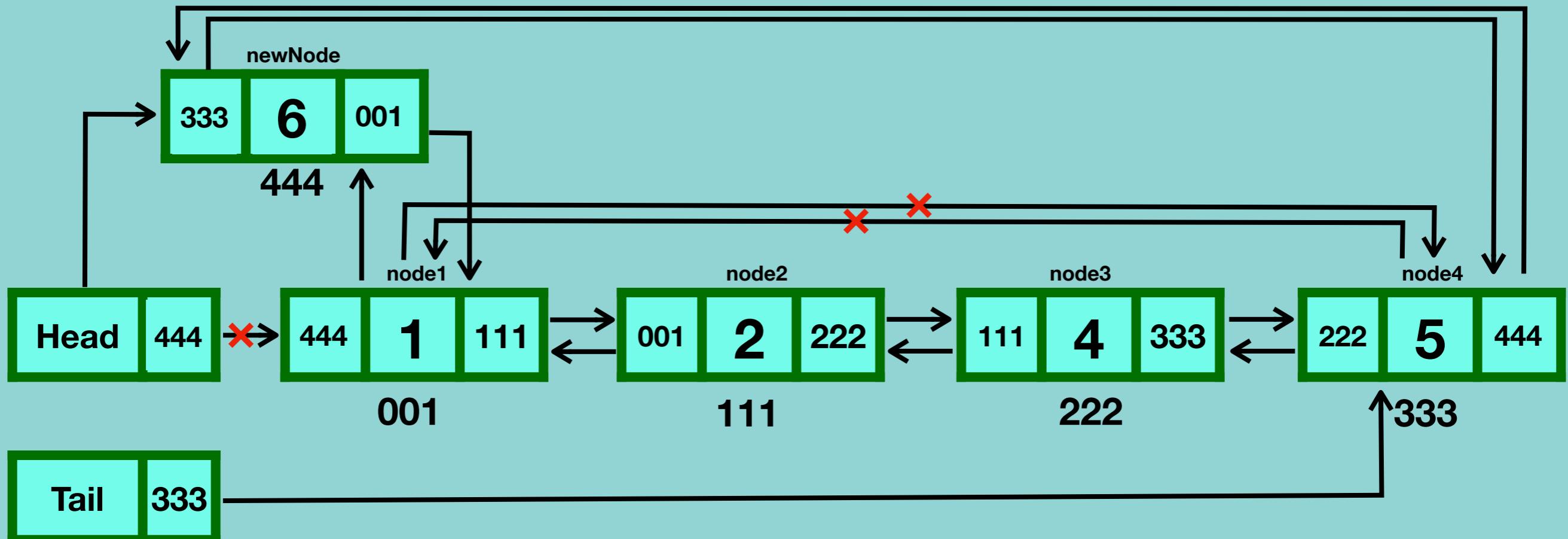
# Circular Doubly Linked List - Insertion

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



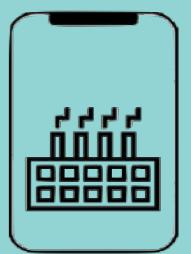
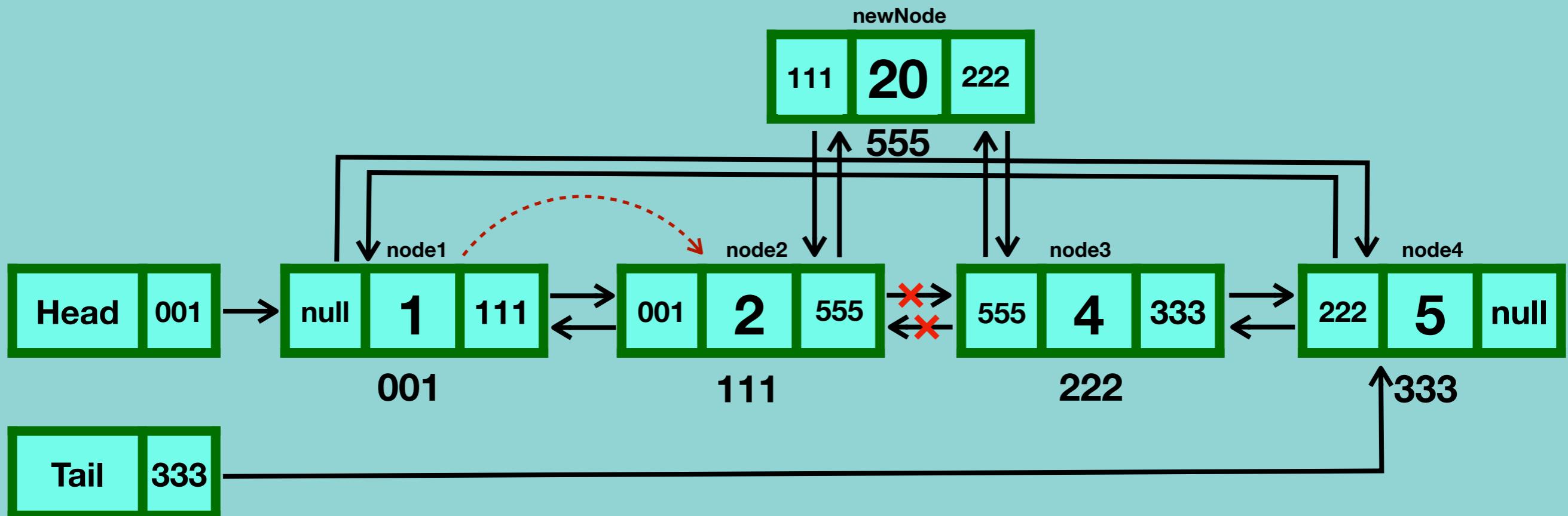
# Circular Doubly Linked List - Insertion

- Insert at the beginning of linked list



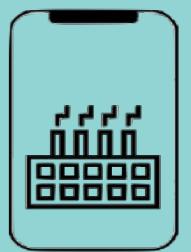
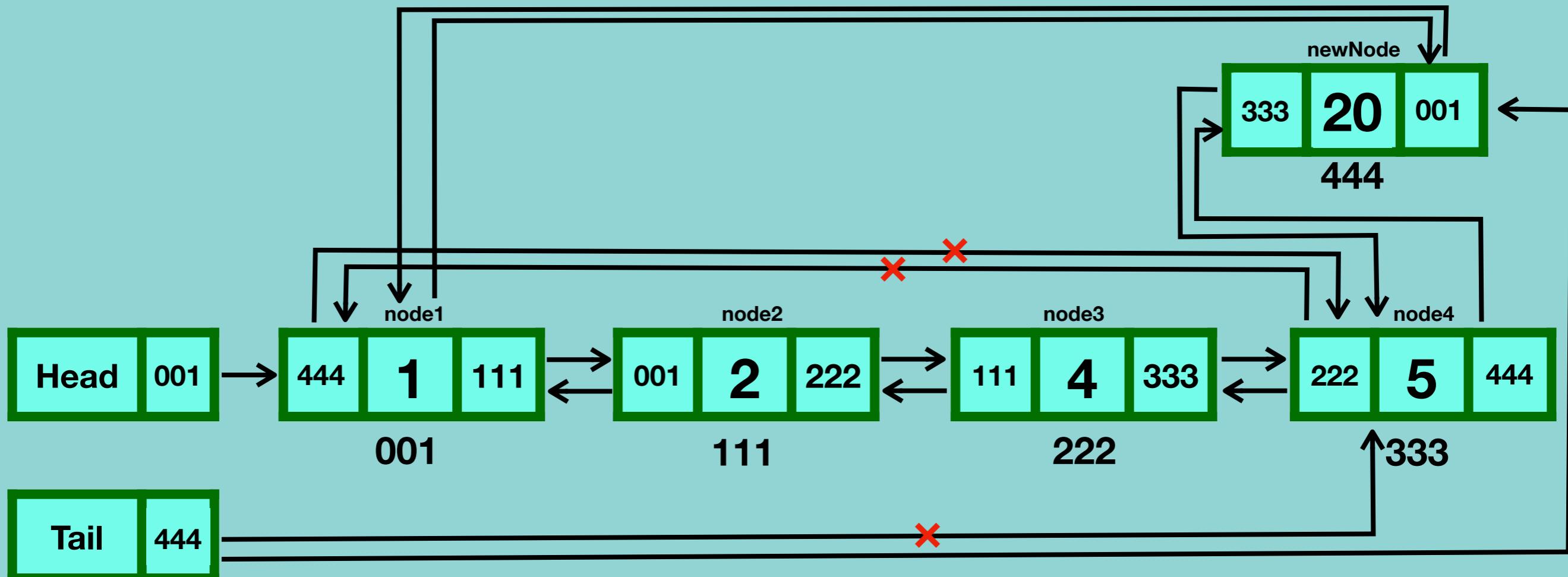
# Circular Doubly Linked List - Insertion

- Insert at the specified location of linked list

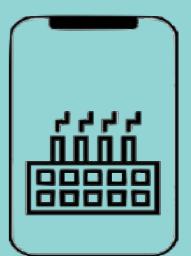
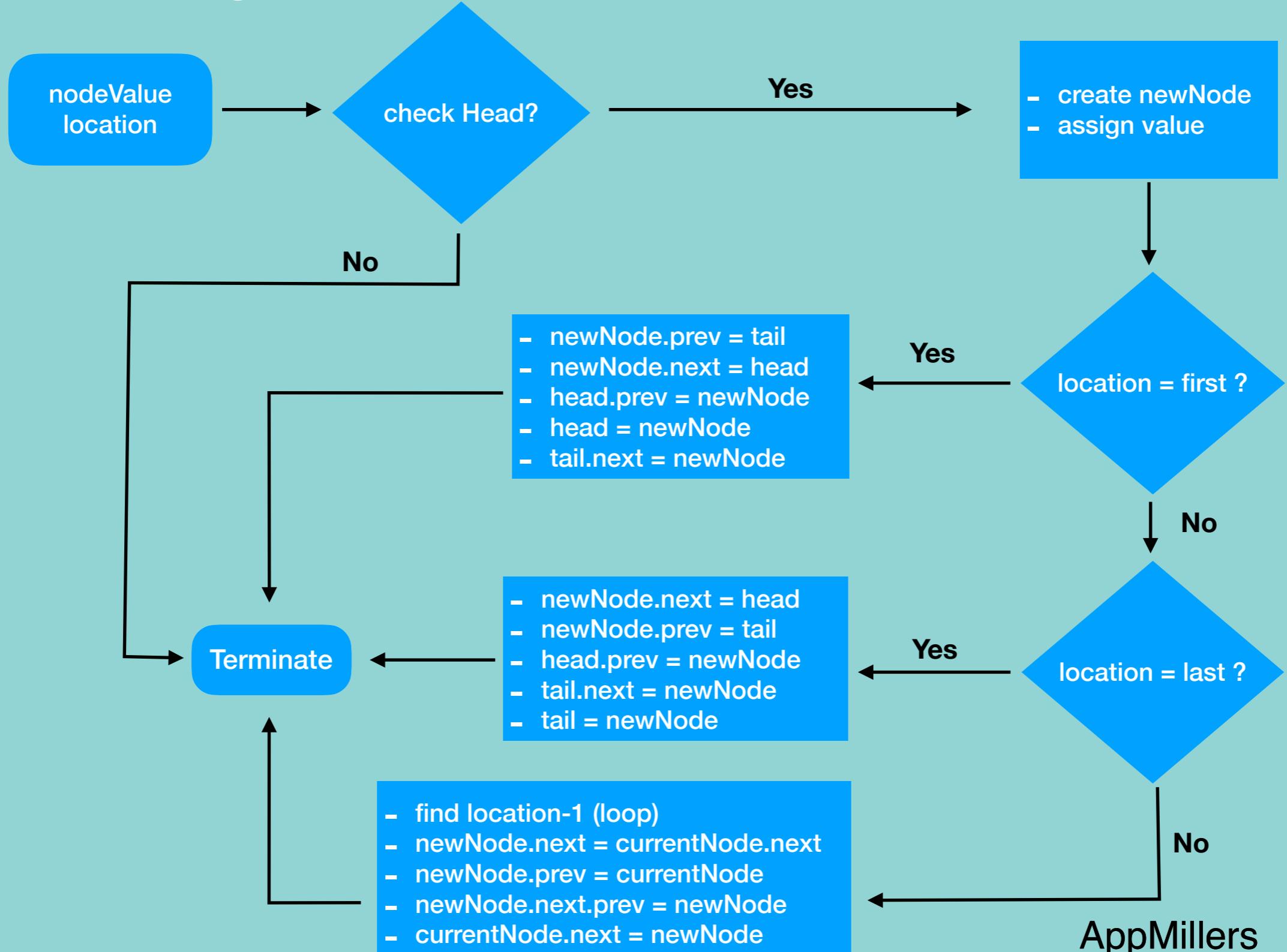


# Circular Doubly Linked List - Insertion

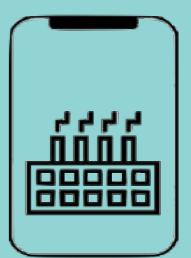
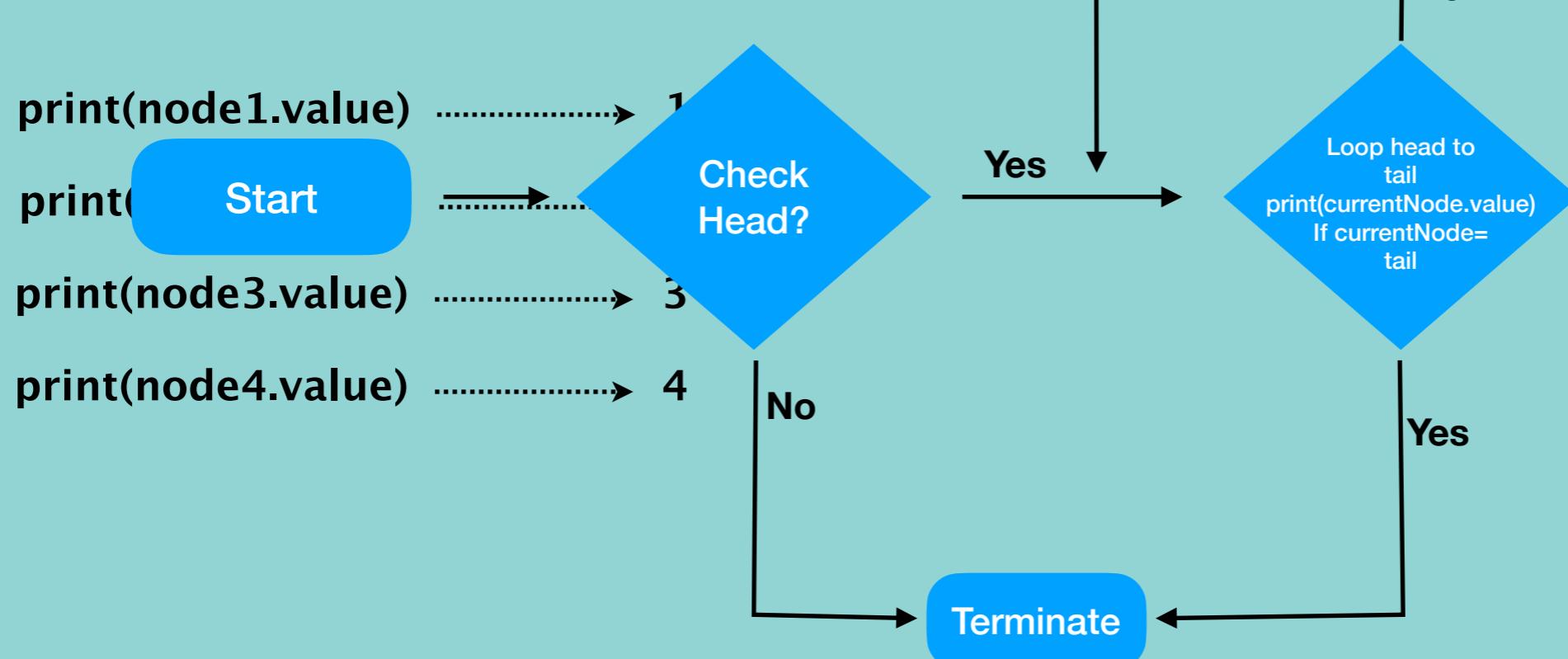
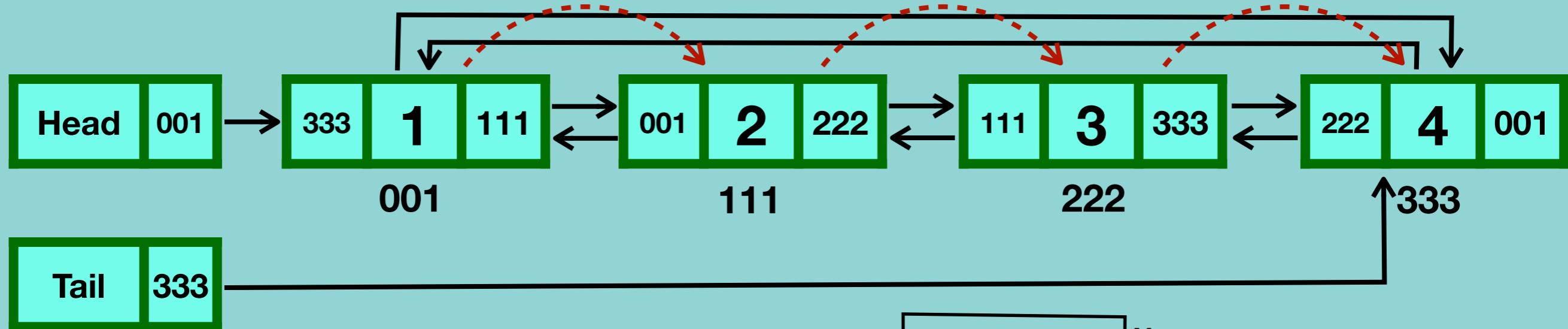
- Insert at the end of linked list



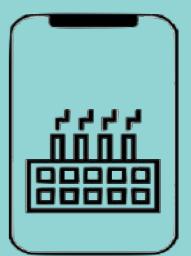
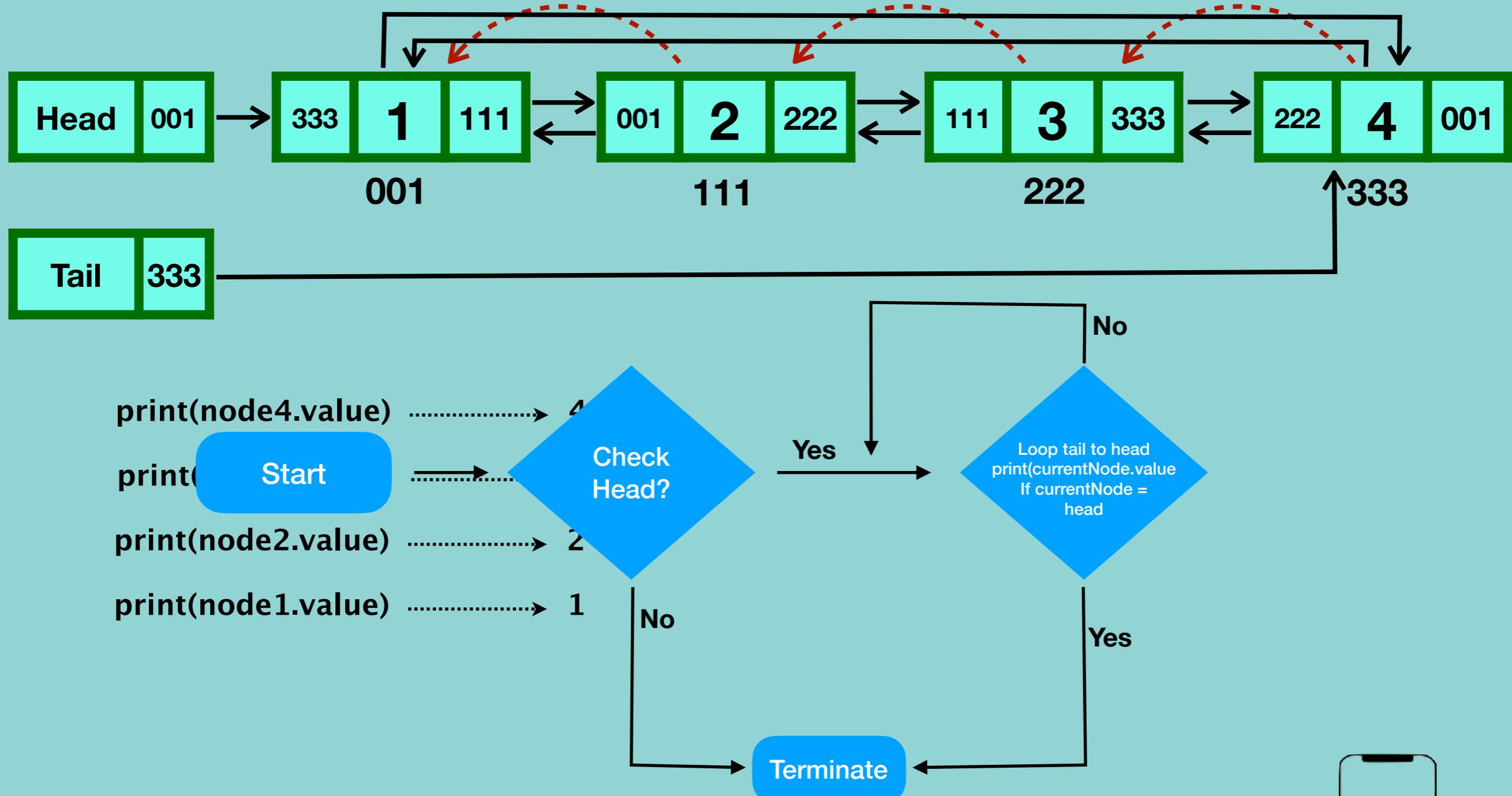
# Insertion Algorithm - Circular Doubly linked list



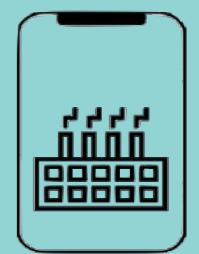
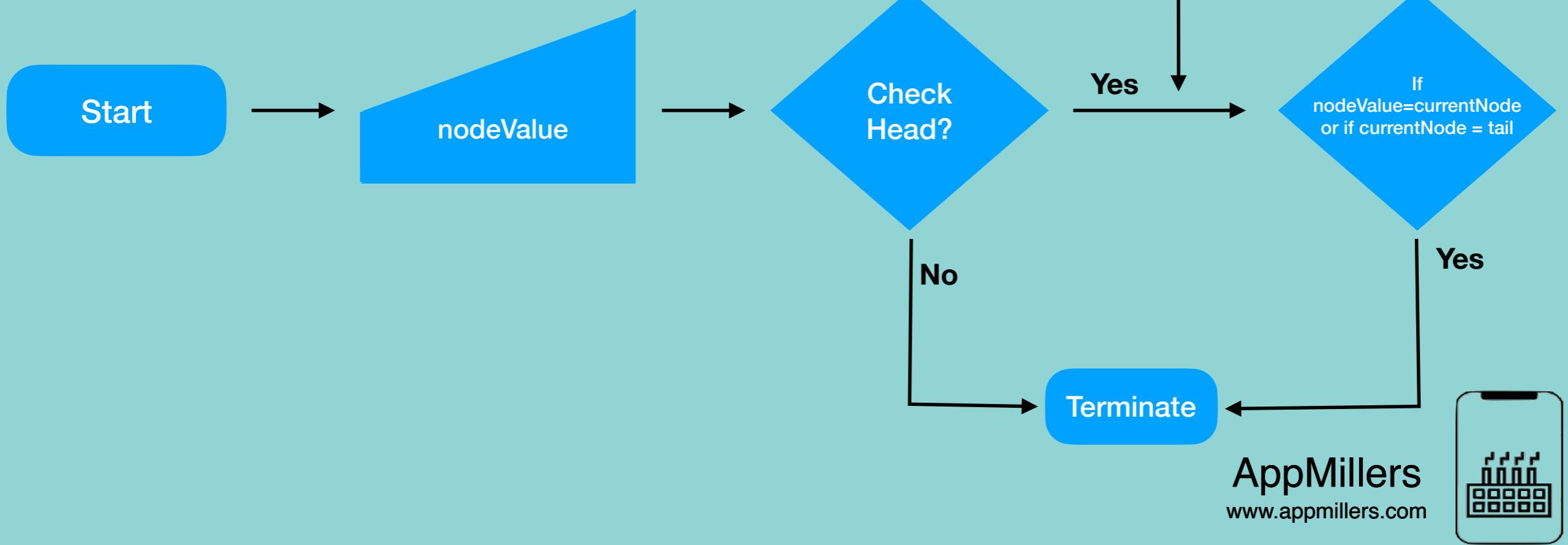
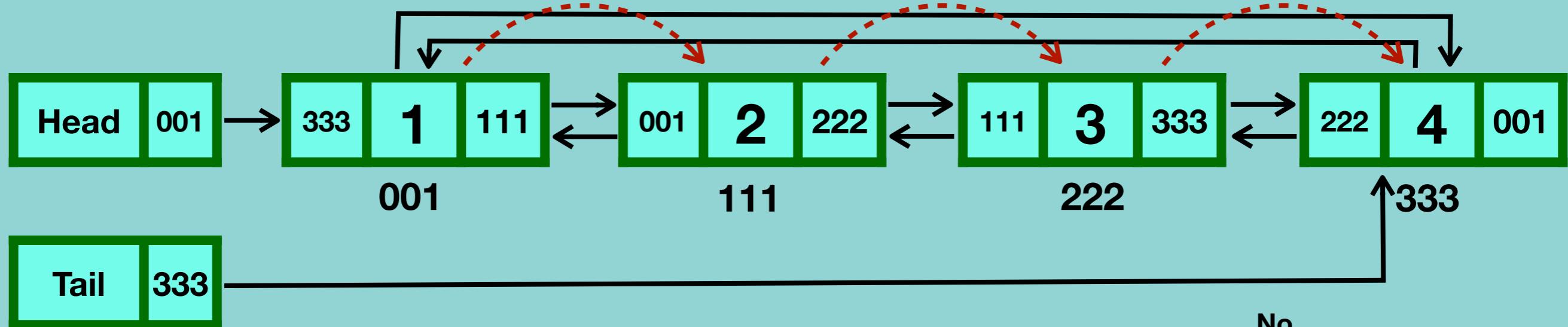
# Circular Doubly Linked List - Traversal



# Circular Doubly Linked List - Reverse Traversal

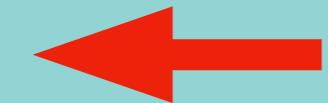


# Circular Doubly Linked List - Search

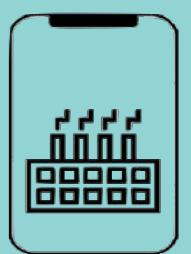
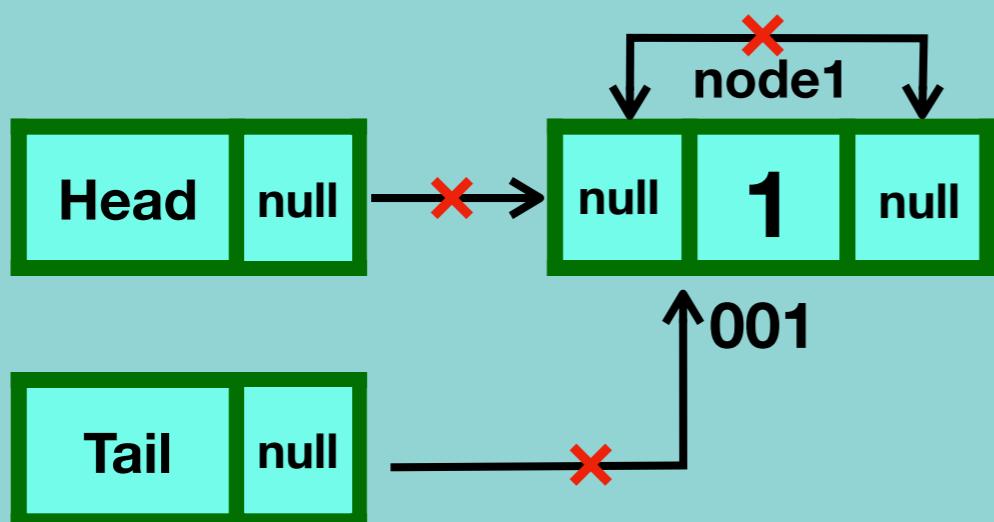


# Circular Doubly Linked list - Deletion

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



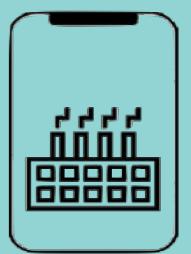
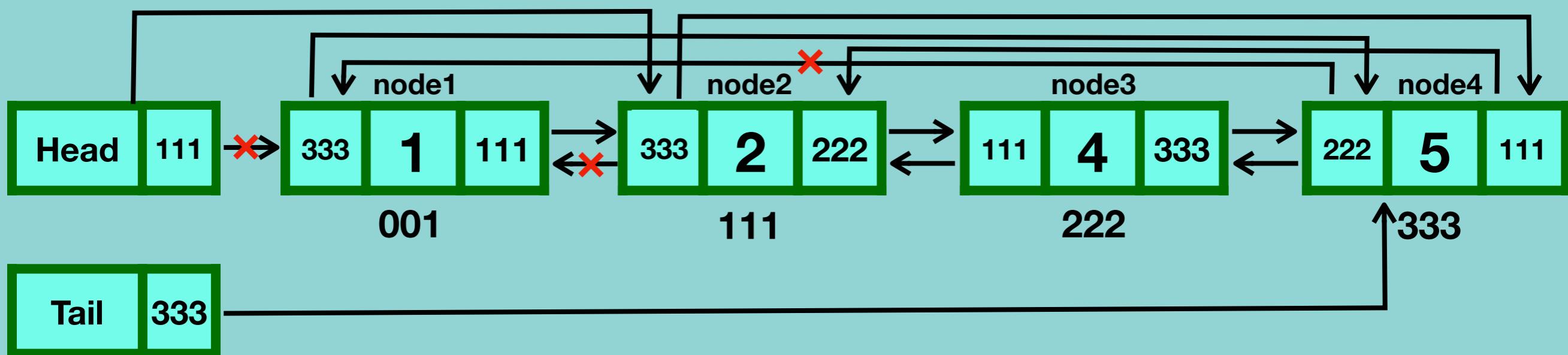
## Case 1 - one node



# Circular Doubly Linked list - Deletion

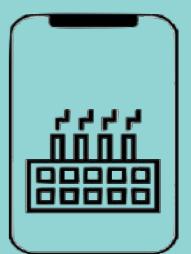
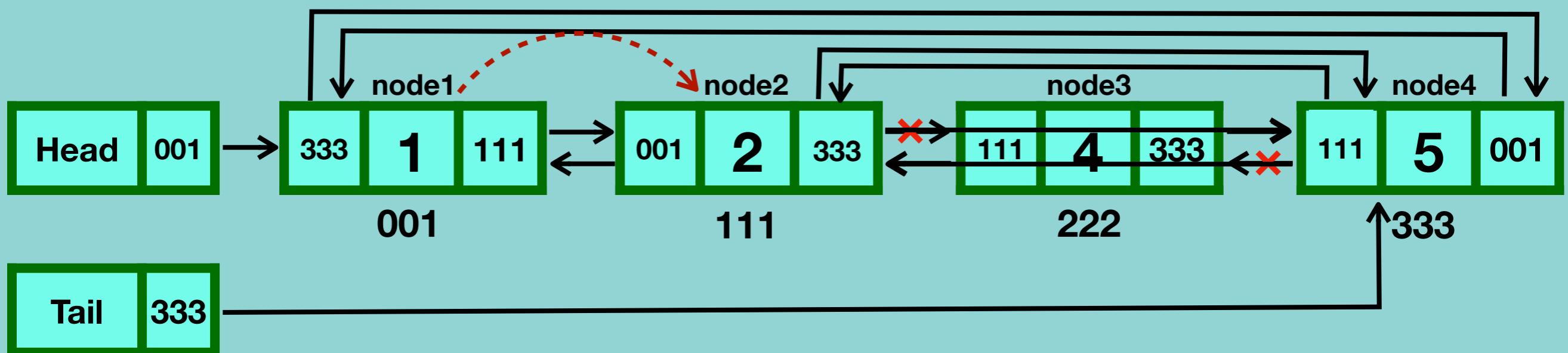
- Deleting the first node
- Deleting any given node
- Deleting the last node

## Case 2 - more than one node



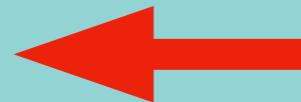
# Circular Doubly Linked list - Deletion

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

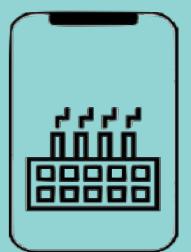
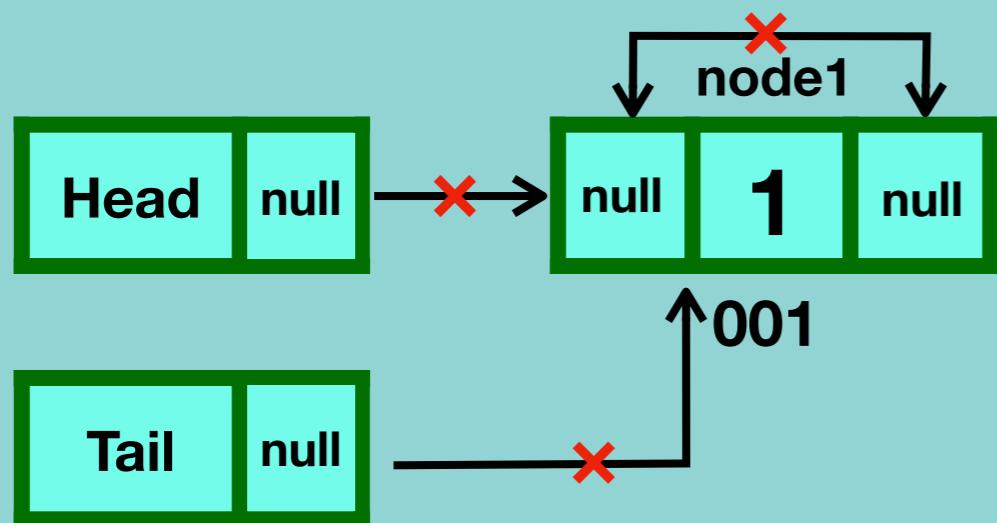


# Circular Doubly Linked list - Deletion

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

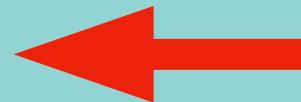


## Case 1 - one node

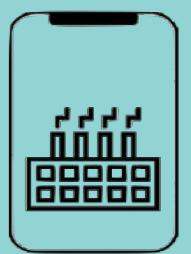
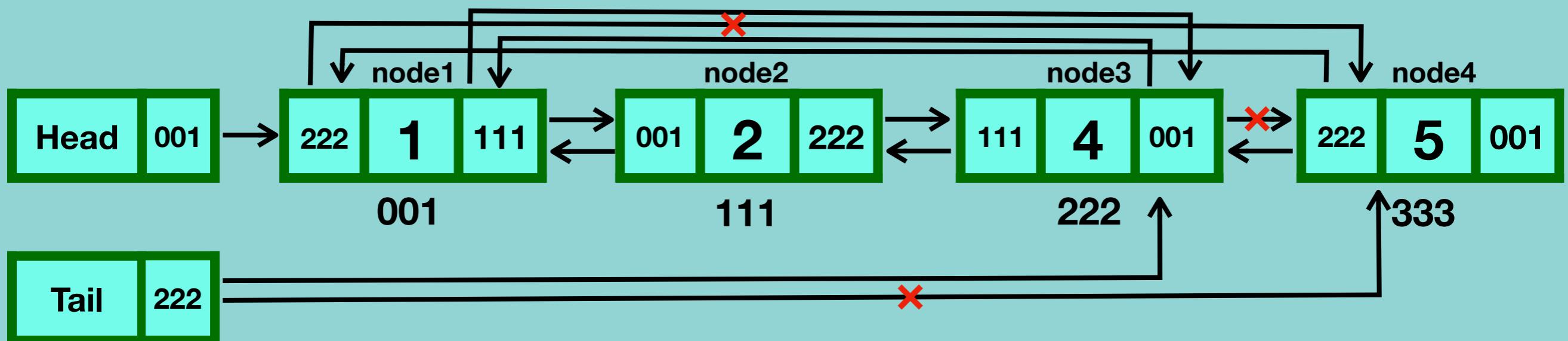


# Circular Doubly Linked list - Deletion

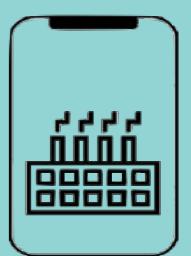
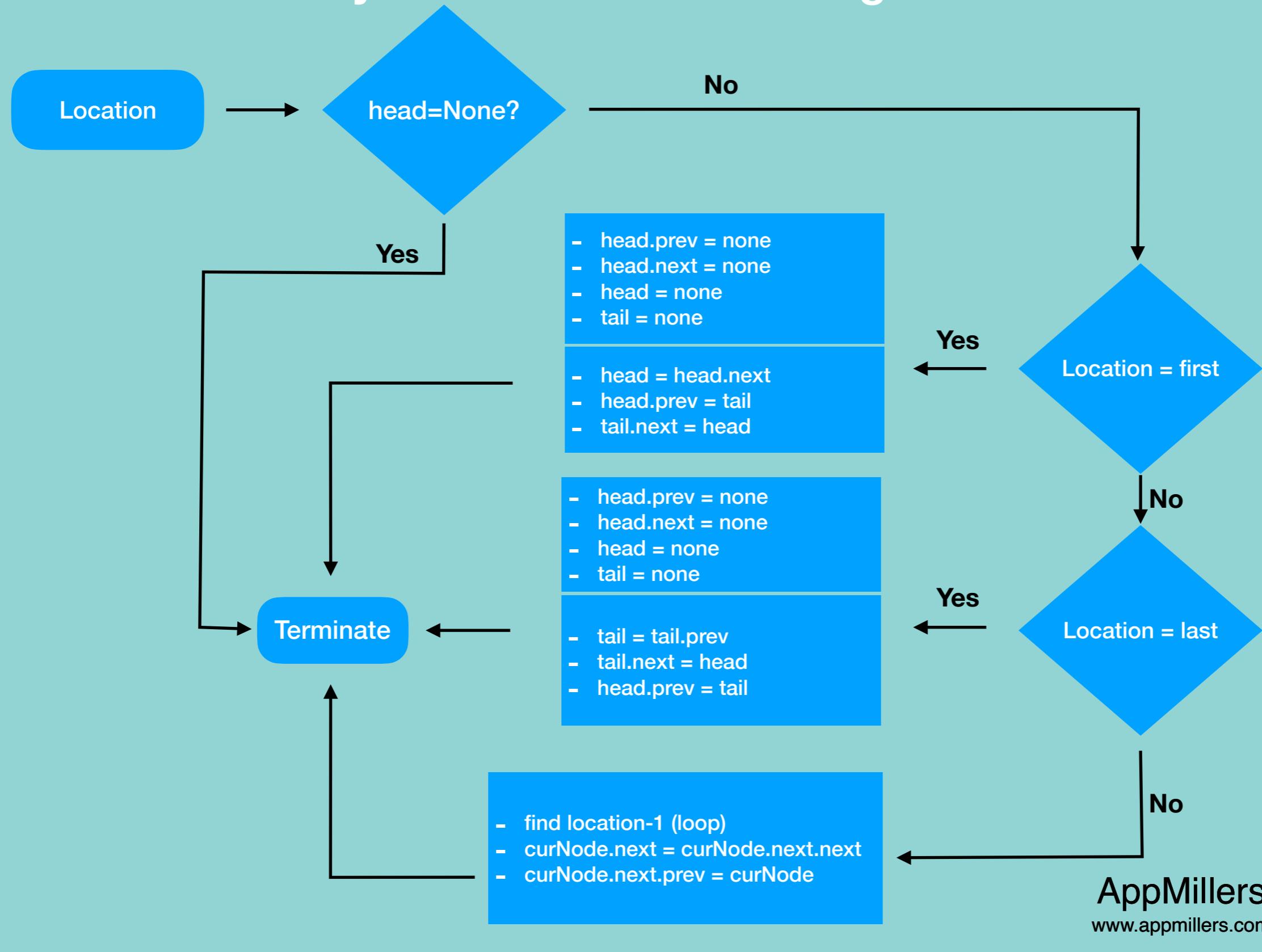
- Deleting the first node
- Deleting any given node
- Deleting the last node



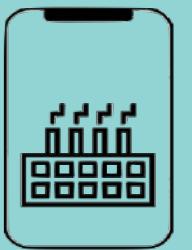
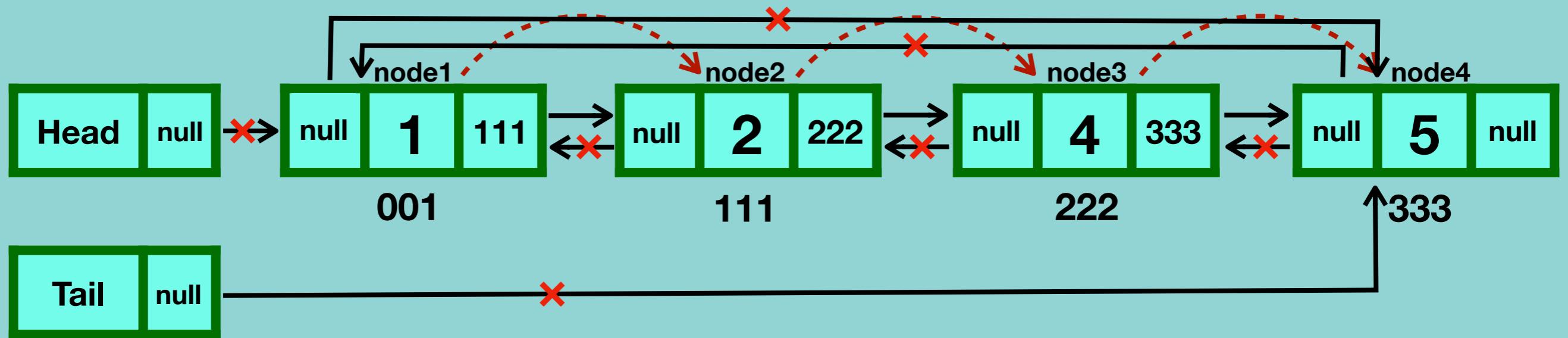
## Case 2 - more than one node



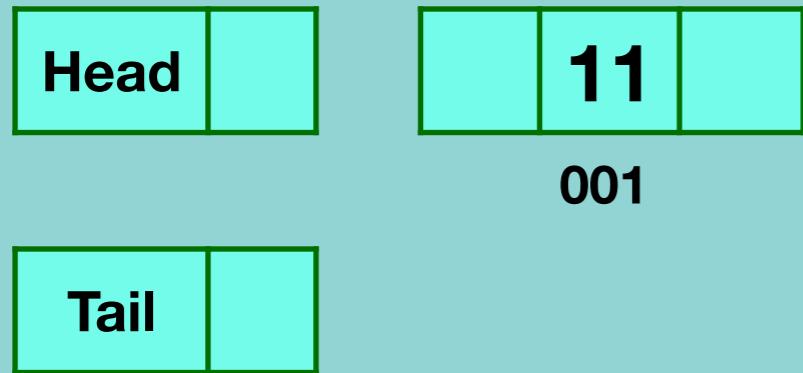
# Circular Doubly Linked list Deletion Algorithm



# Deleting entire circular doubly linked list

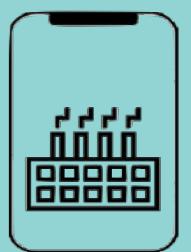


# Circular doubly linked list - creation

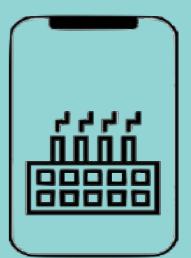
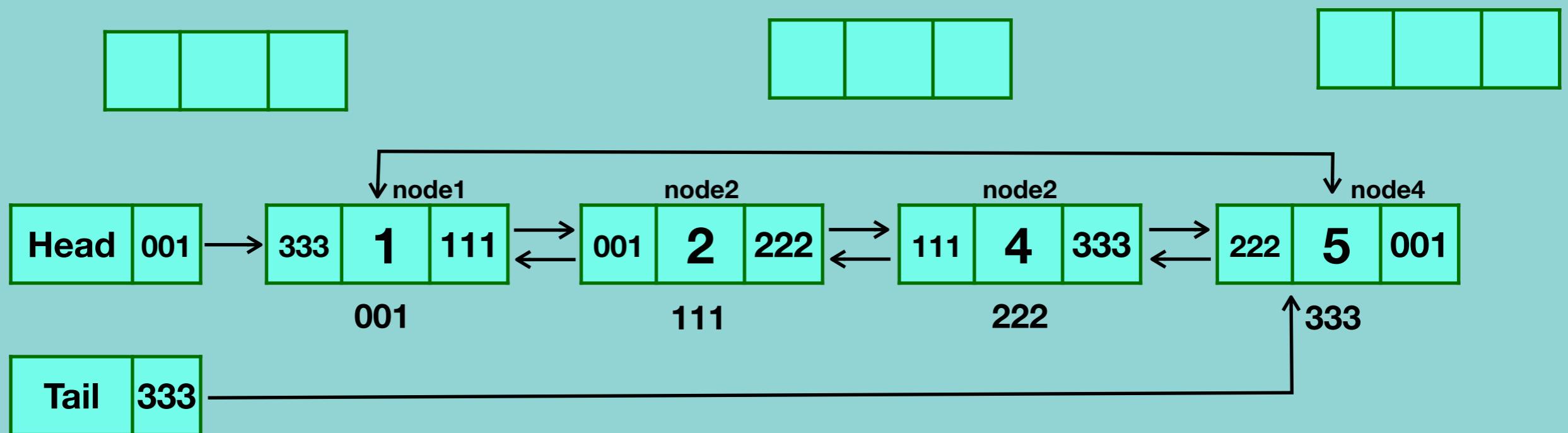


```
createCircularDoublyLinkedList(nodeValue):  
    create a blank newNode  
    newNode.value = nodeValue  
    head = newNode  
    tail = newNode  
    newNode.next = newNode.prev = newNode
```

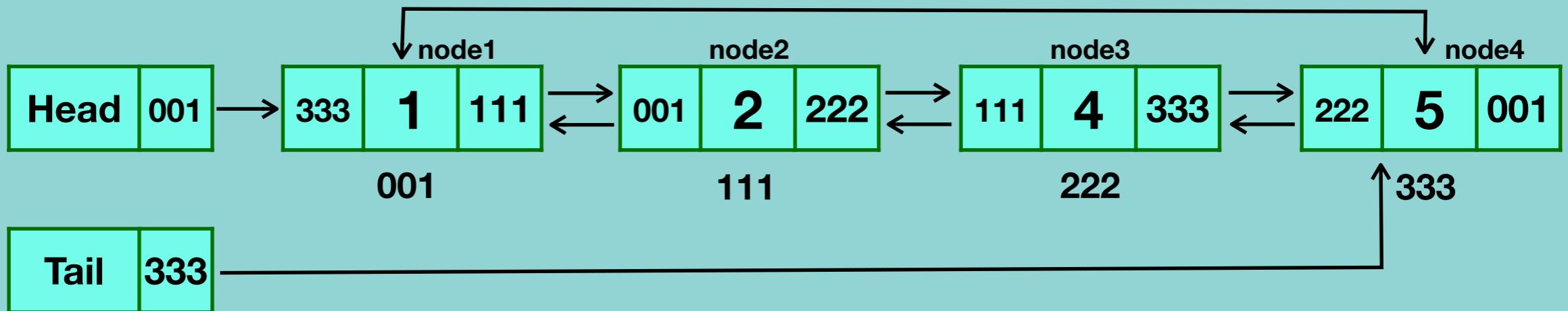
**Time complexity : O(1)**  
**Space complexity : O(1)**



# Circular doubly linked list - insertion



# Circular doubly linked list - traversal

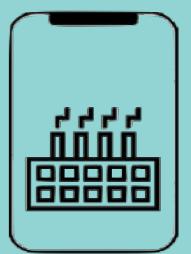


```
traversalCircularDoublyLinkedList():
```

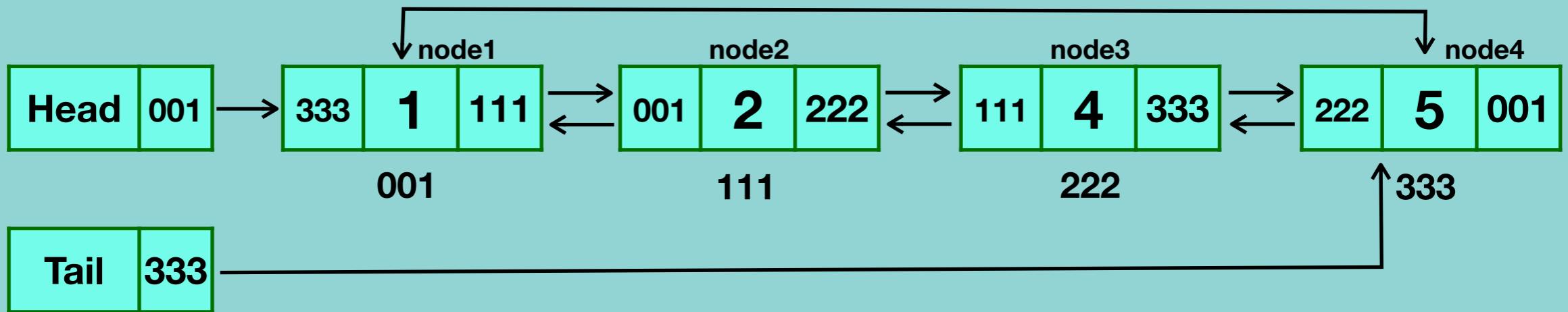
```
if head == null:  
    return //There is not any node in this list  
loop head to tail:  
    print(currentNode.value)  
    if currentNode.value = tail:  
        terminate
```

Time complexity : O(n)

Space complexity : O(1)



# Circular doubly linked list - reverse traversal

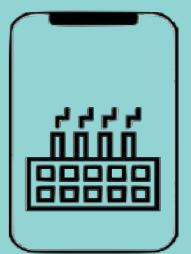


```
reverseTraversalCircularDoublyLinkedList():
```

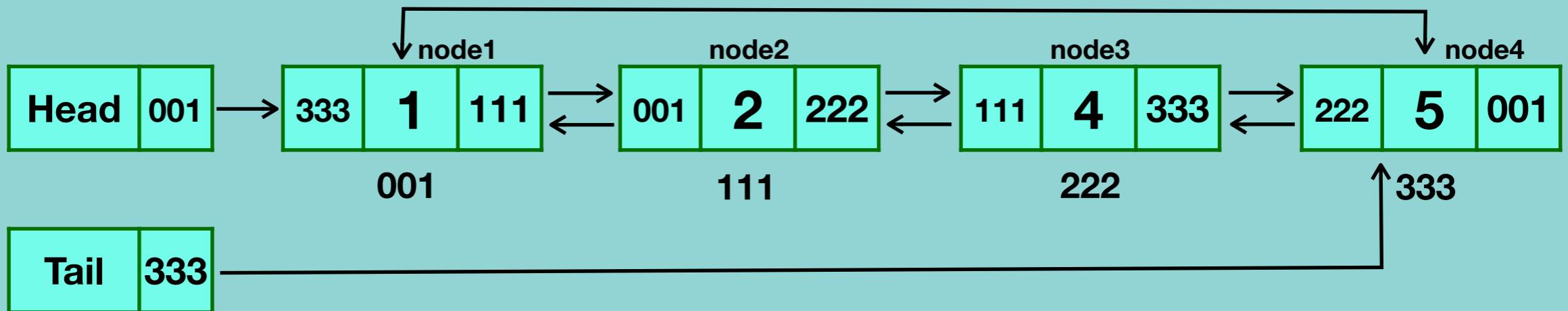
```
if head == null:  
    return //There is not any node in this list  
loop tail to head:  
    print(currentNode.value)  
    if currentNode.value = head:  
        terminate
```

Time complexity : O(n)

Space complexity : O(1)



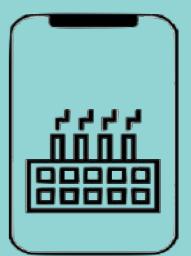
# Circular doubly linked list - searching for a node



```
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)

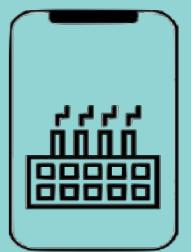
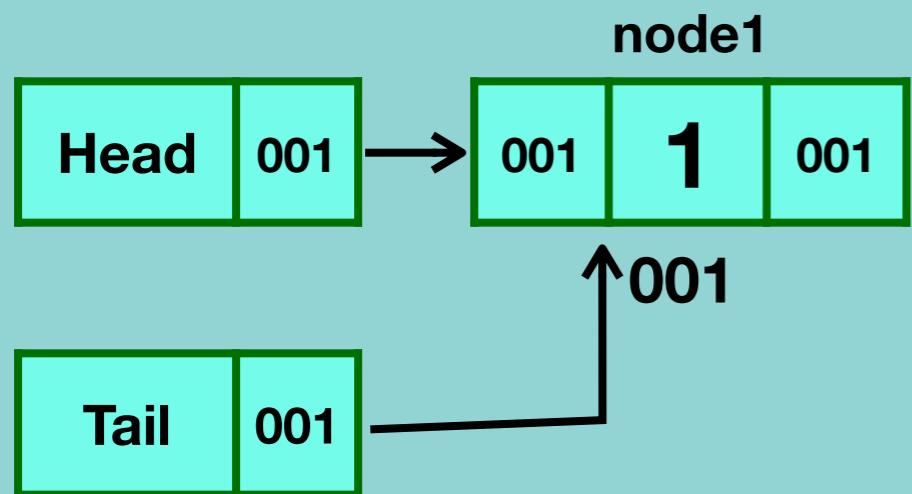


# Deleting a node in doubly linked list

- 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

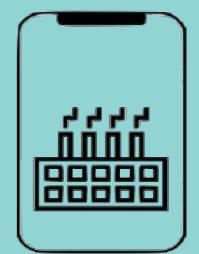
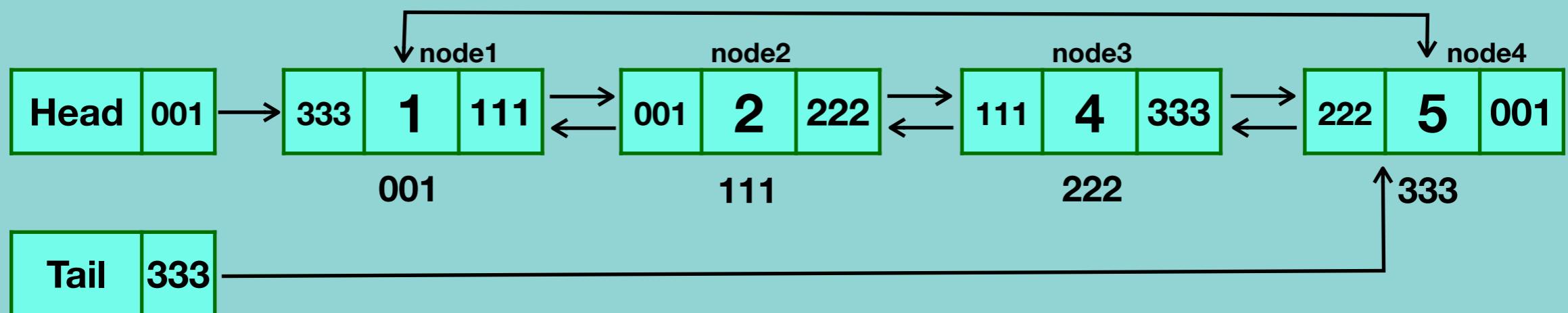


# Deleting a node in circular doubly linked list

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

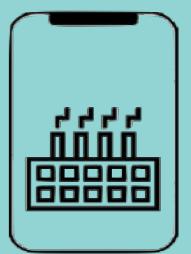
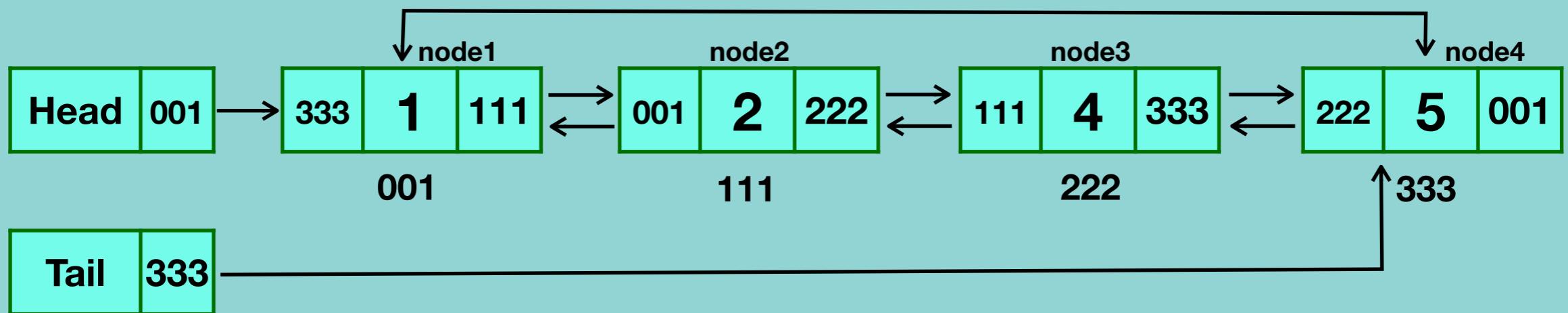
## Deleting the first node

### Case 1- More than one node in the list



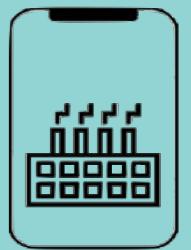
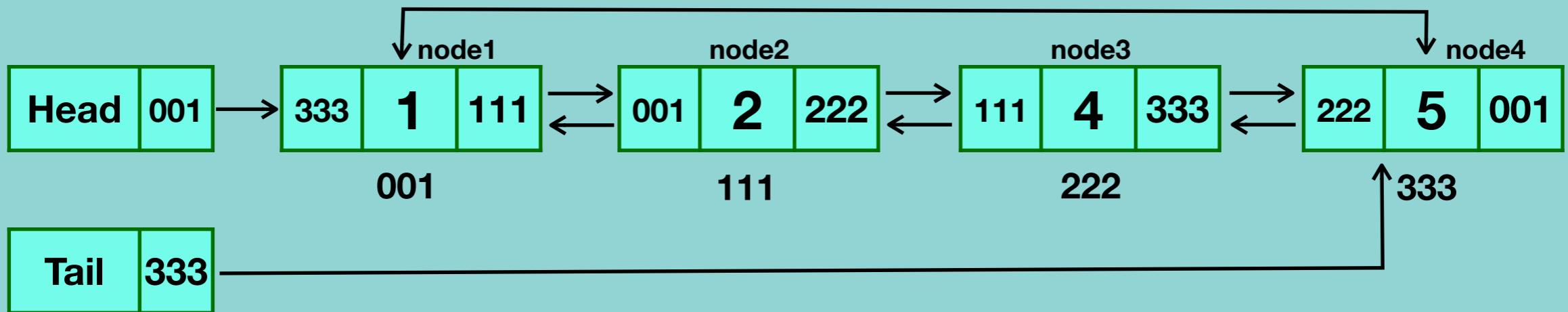
# Deleting a node in circular doubly linked list

## Deleting the last node node

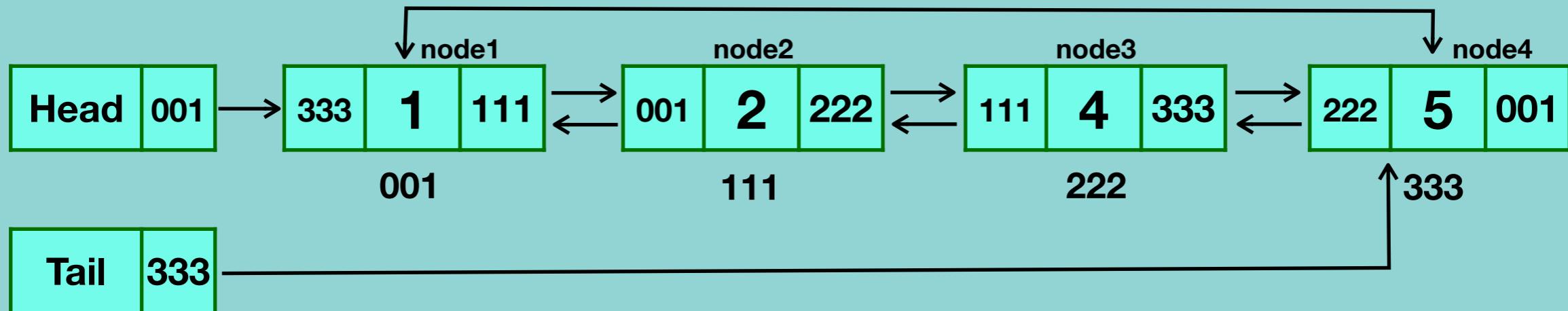


# Deleting a node in circular doubly linked list

Deleting the node from the middle



# Deleting a node in circular doubly linked list- Algorithm

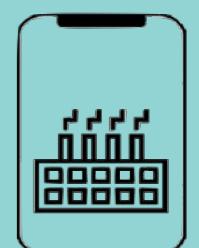


```
deleteNode(head, location):
```

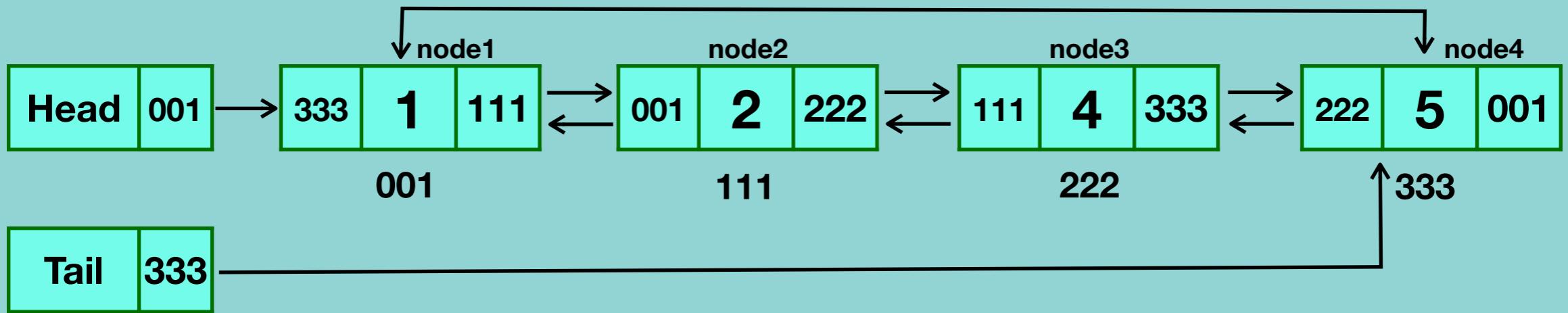
```
if head does not exist:  
    return error //Linked list does not exist  
if location = firstNode's location  
    if this is the only node in the list  
        head.prev=head.next=head=tail=null  
    else  
        head=head.next  
        head.prev = tail  
        tail.next = head  
else if 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)

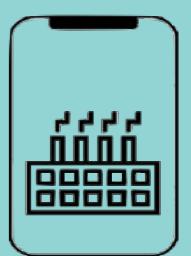


# Deleting entire circular doubly linked list- Algorithm



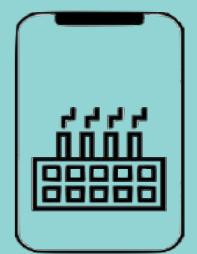
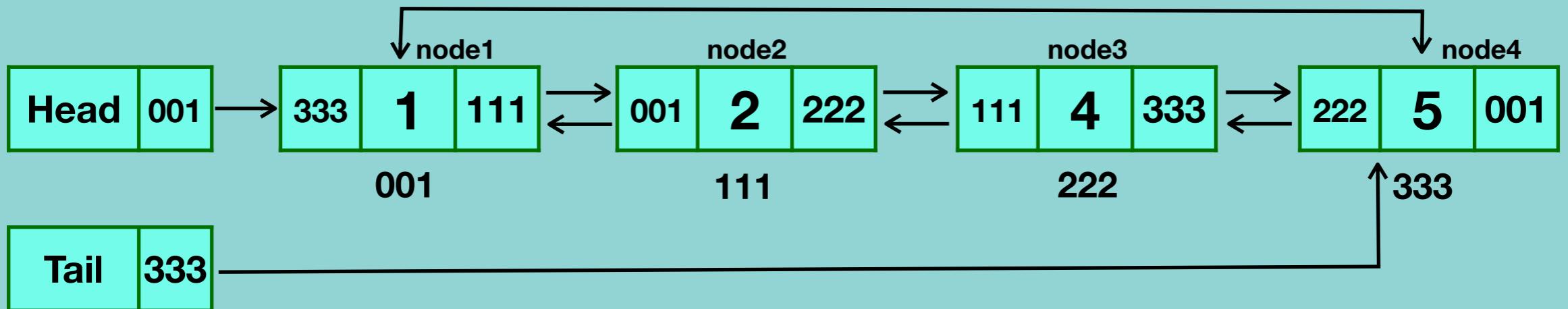
```
deleteLinkedList(head, tail):  
    tail.next = null  
    loop head to tail:  
        curNode.prev=null  
    head=tail=null
```

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



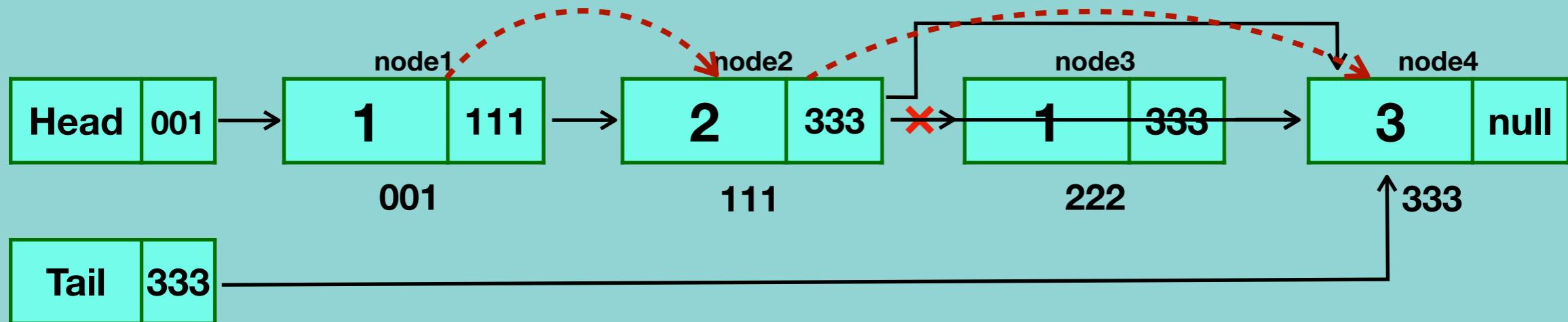
# Time and Space complexity of circular doubly linked list

	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
```

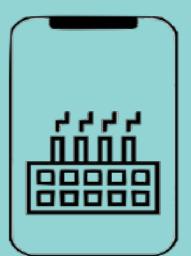
```
tempSet = {1} → {1,2} → {1,2,3}
```

```
while currentNode.next is not None:
```

```
    If next node's value is in tempSet
```

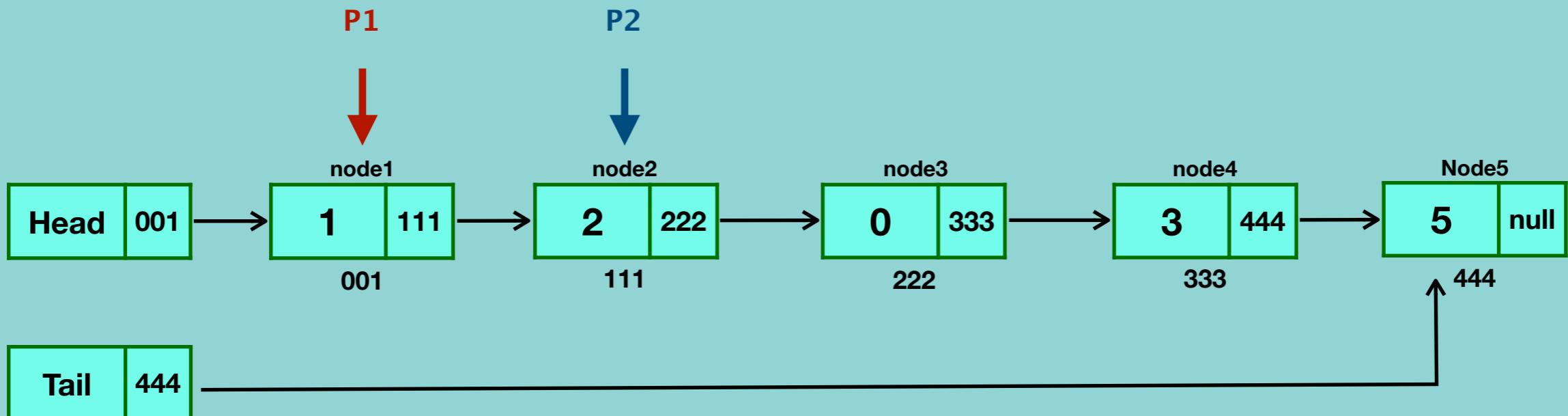
```
        Delete next node
```

```
    Otherwise add it to tempSet
```



# Interview Questions - 2 : Return Nth to Last

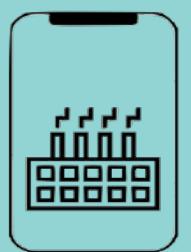
Implement and algorithm to find the nth to last element of a singly linked list.



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

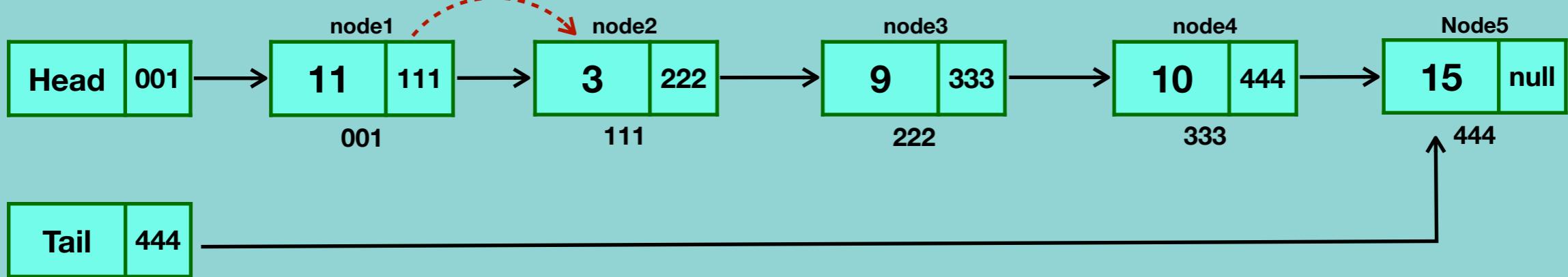
pointer1 = node1  
= node2  
= node3  
**= node4**

pointer2 = node2  
= node3  
= node4  
= node5



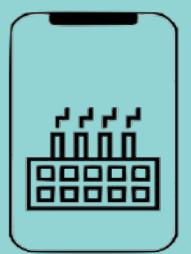
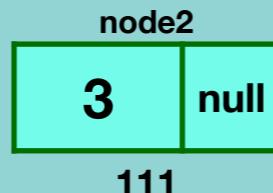
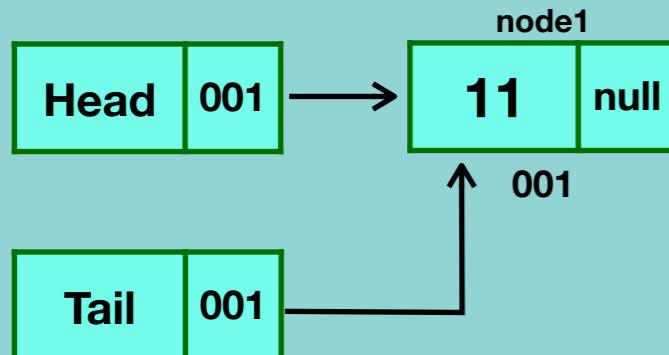
# Interview Questions - 3 : Partition

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.



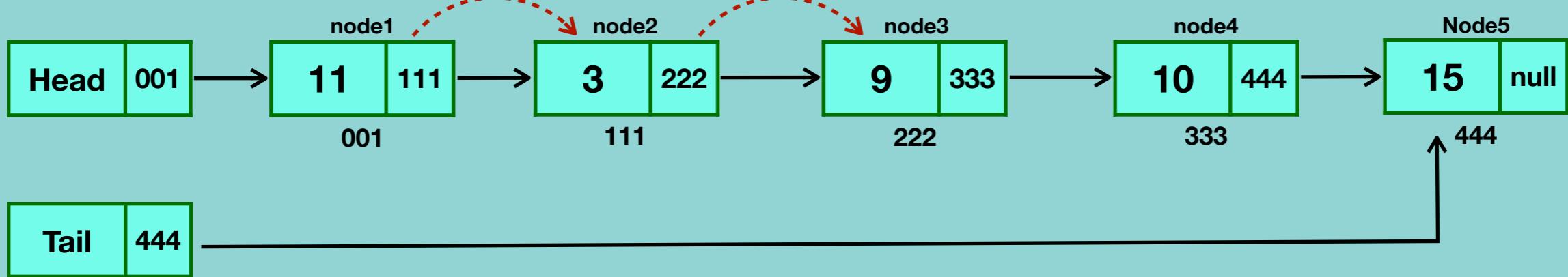
$x = 10$

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



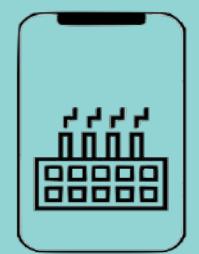
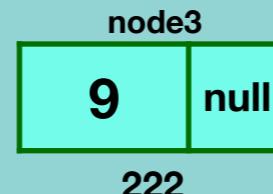
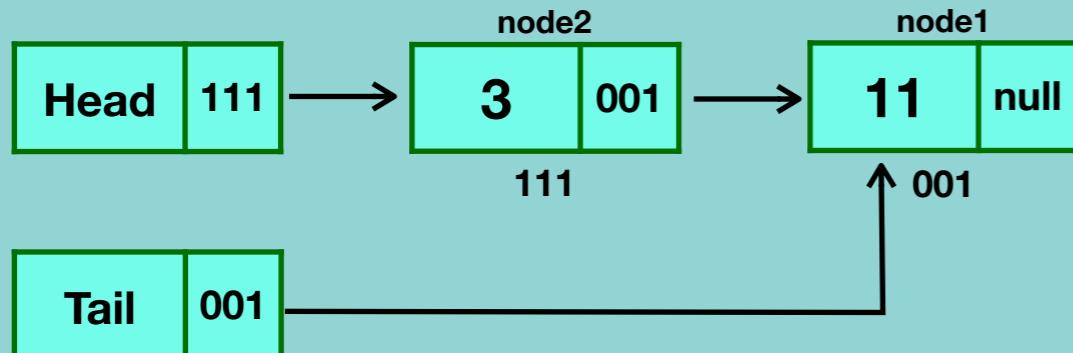
# Interview Questions - 3 : Partition

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.



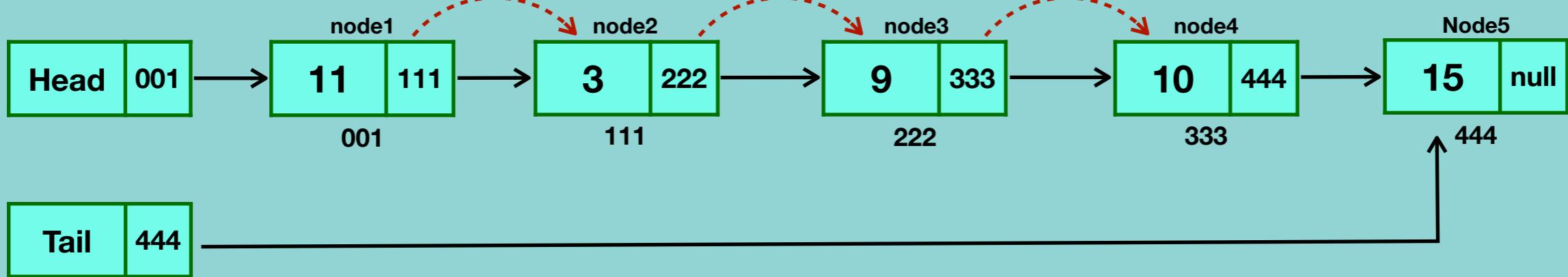
$x = 10$

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



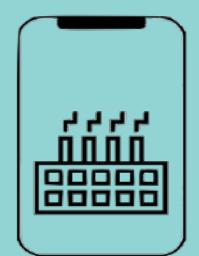
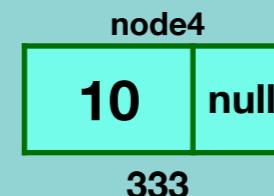
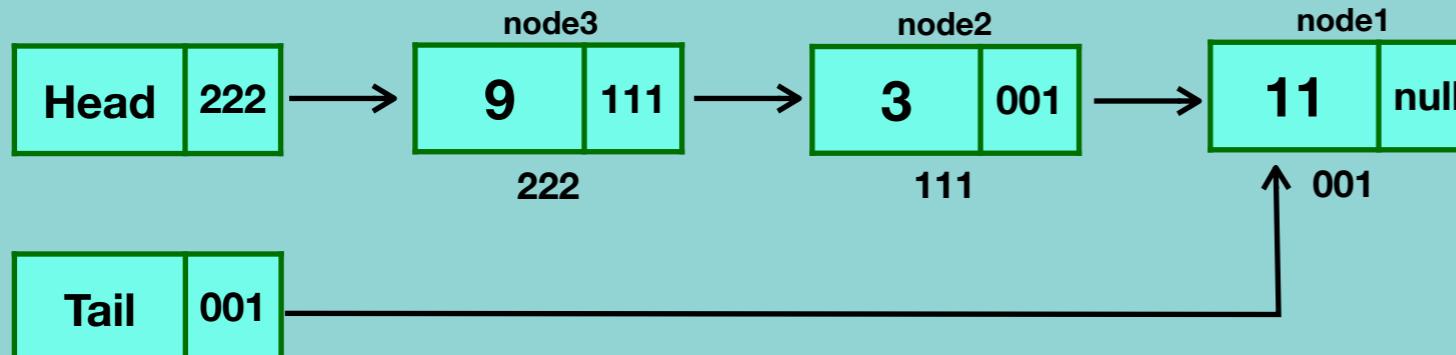
# Interview Questions - 3 : Partition

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.



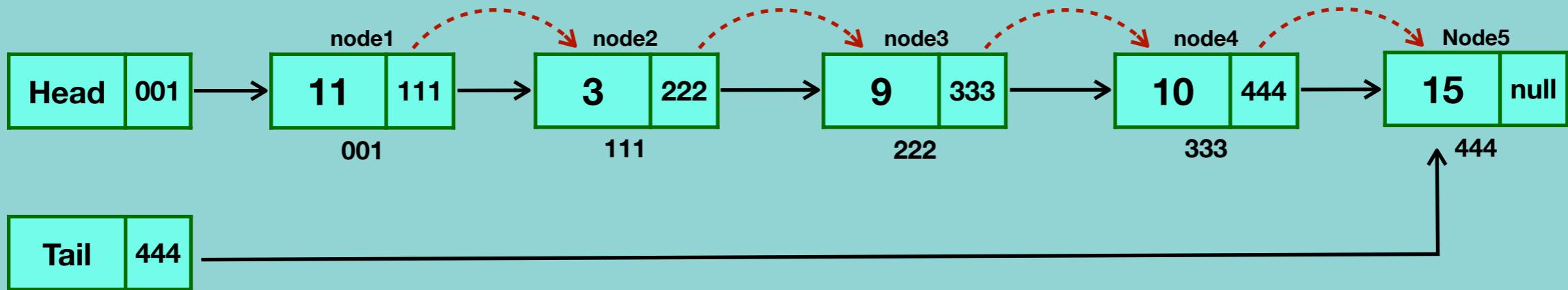
$x = 10$

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



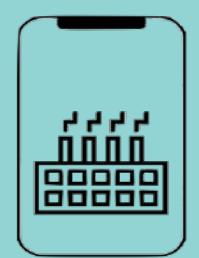
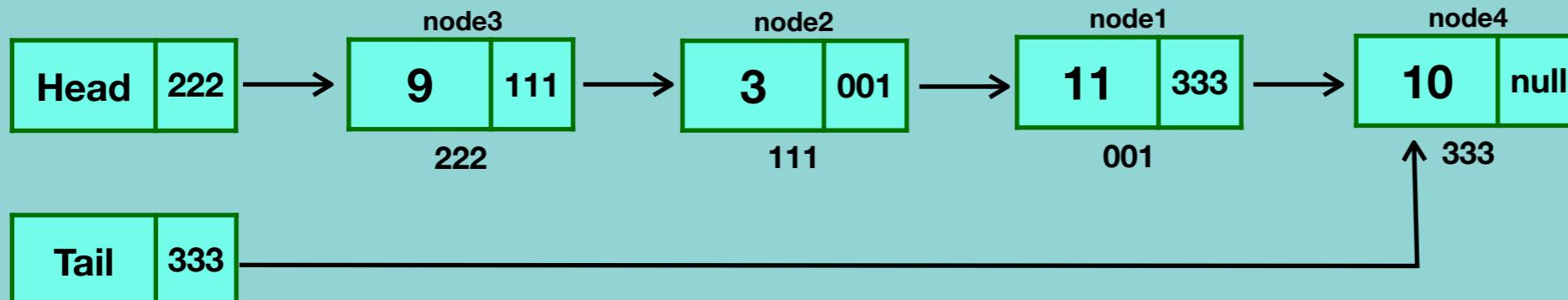
# Interview Questions - 3 : Partition

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.



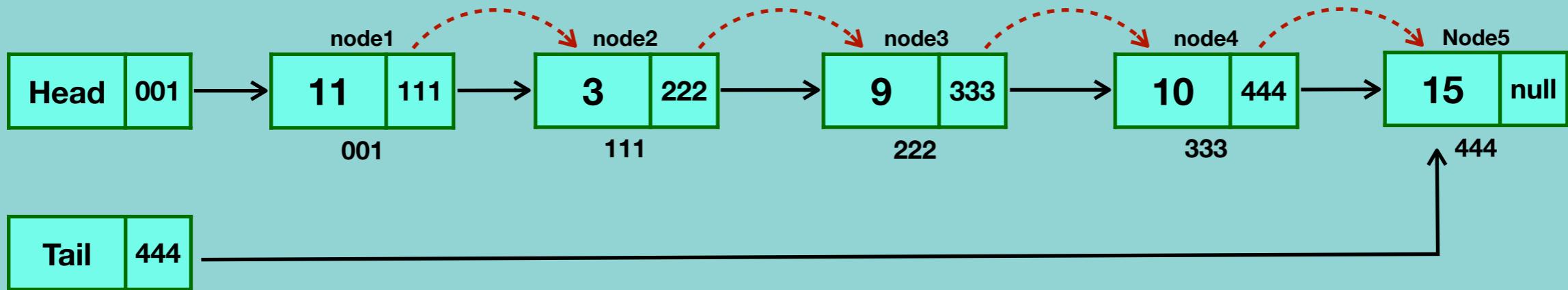
$x = 10$

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



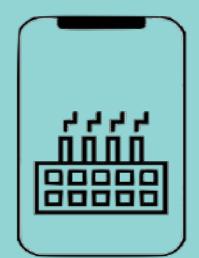
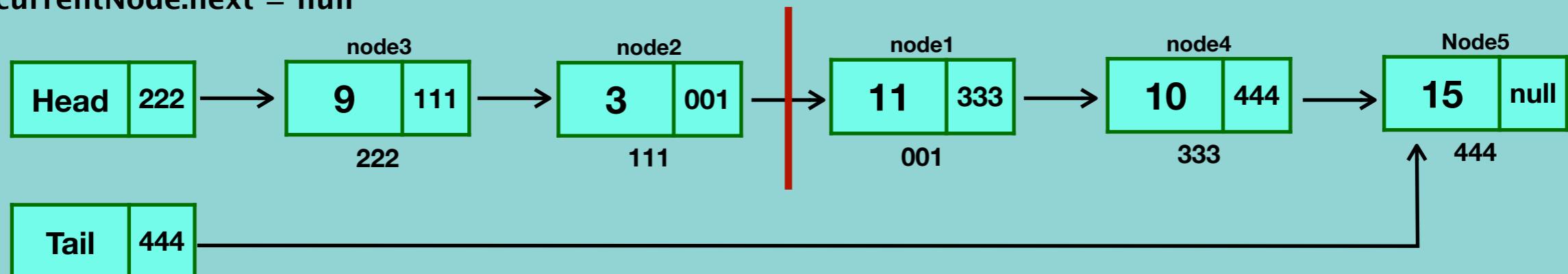
# Interview Questions - 3 : Partition

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
```



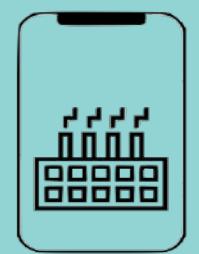
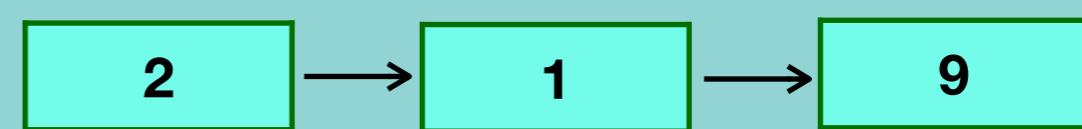
# Interview Questions - 4 : Sum Lists

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

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

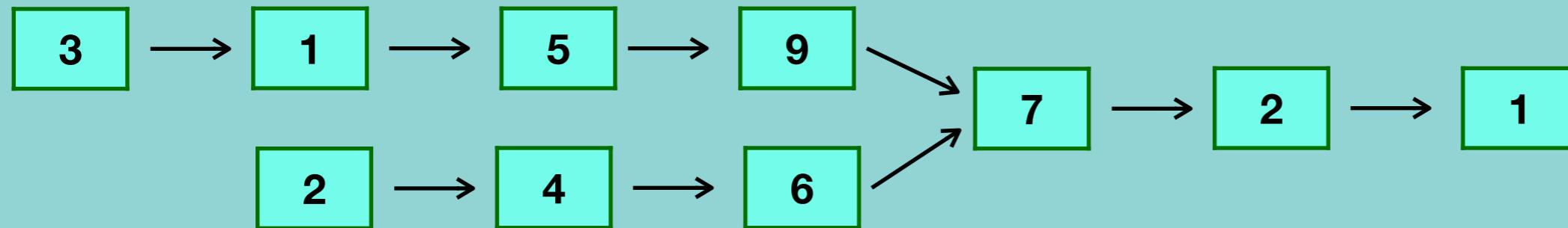
7 + 5 = 12  
1+9+1 = 11  
6+2+1 = 9



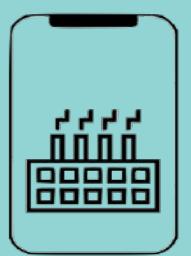
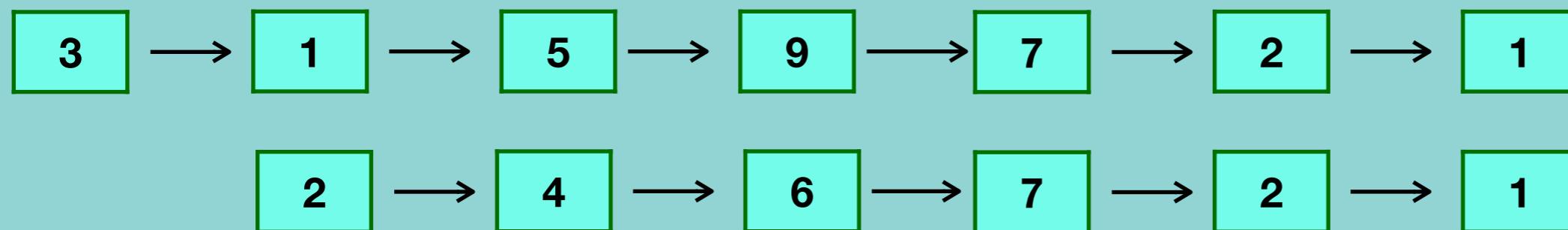
# Interview Questions - 5 : Intersection

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



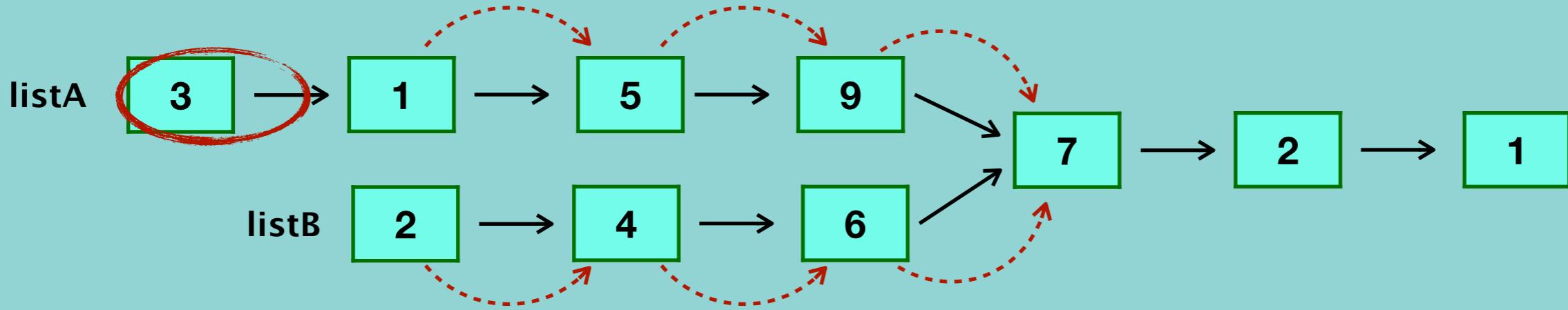
## Non – intersecting linked lists



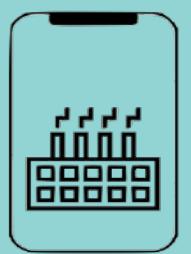
# Interview Questions - 5 : Intersection

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

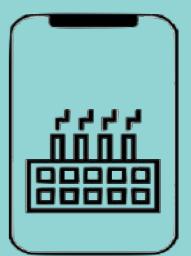
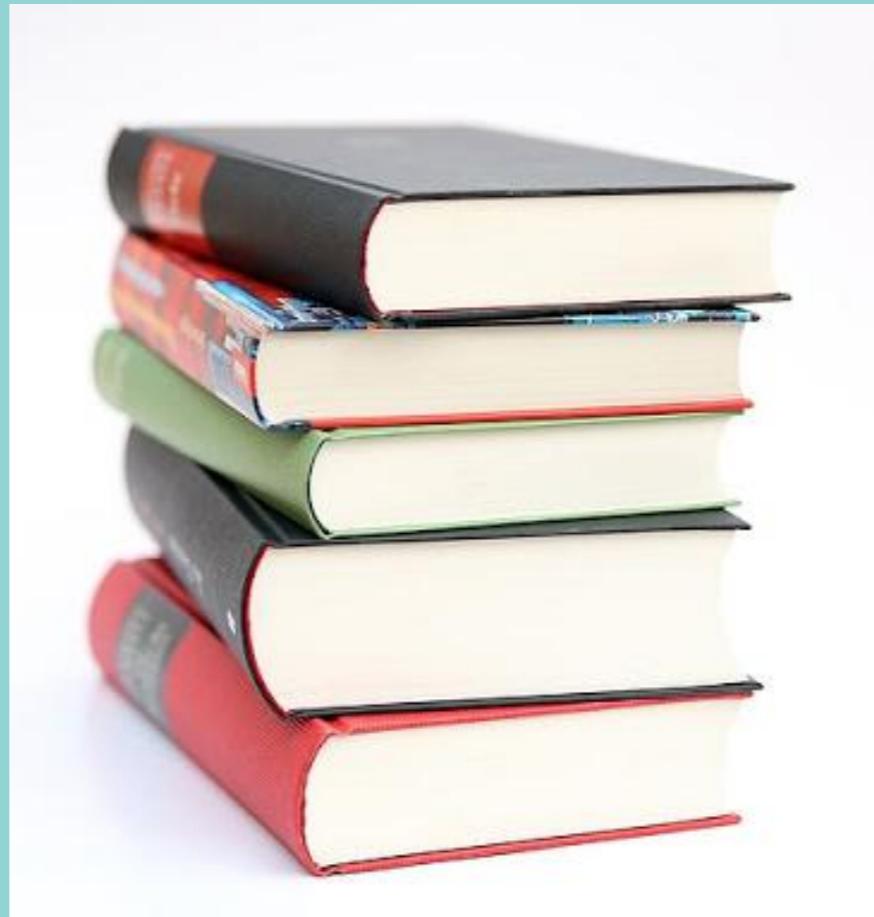


$$\begin{aligned} \text{len(listA)} &= 7 \\ \text{len(listB)} &= 6 \end{aligned} \longrightarrow 7 - 6 = 1$$



# What is a Stack?

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

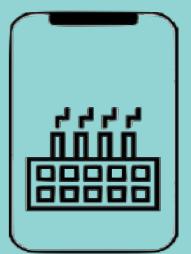


# What is a Stack?

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

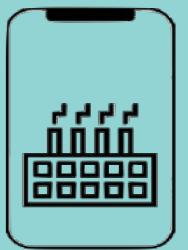
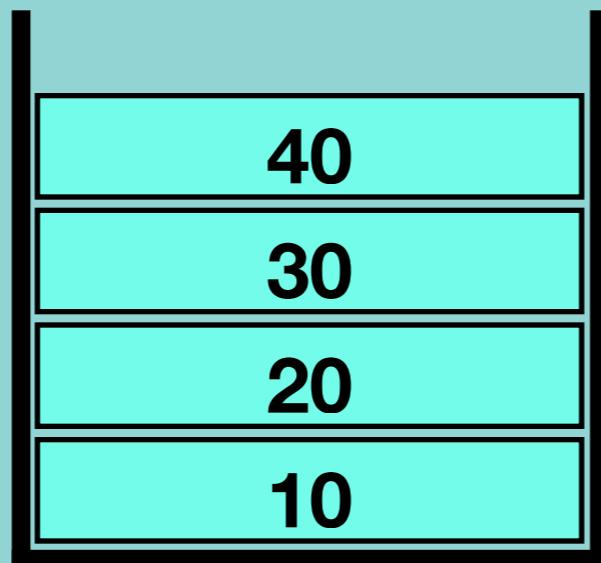


**LIFO method**

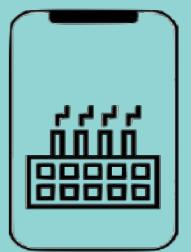
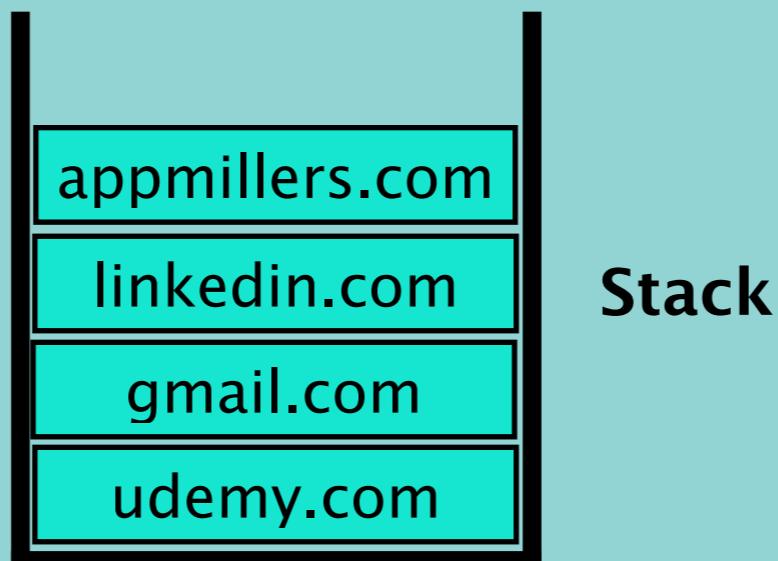
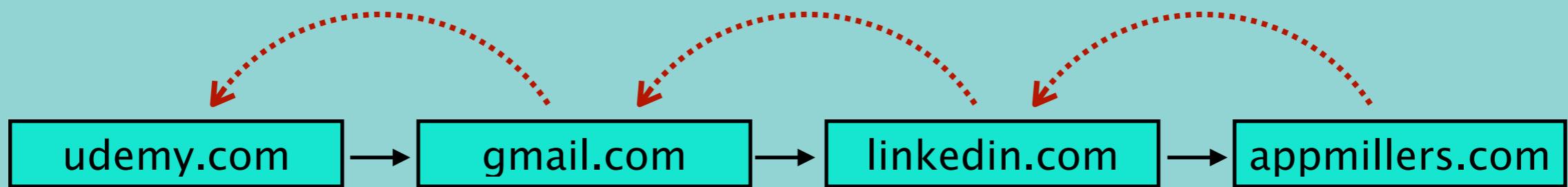
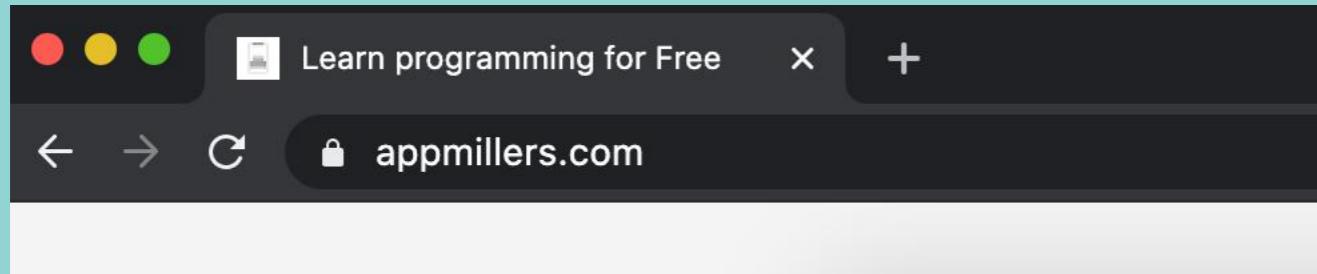


# What is a Stack?

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



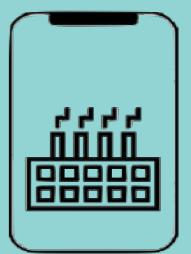
# Why do we need a Stack?



# Stack operations

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

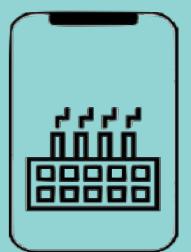
**customStack()**



# Push() method

```
customStack = [ ]
```

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

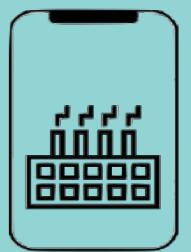



# Push() method

**customStack = [1]**

**customStack.push(2)**

				1					

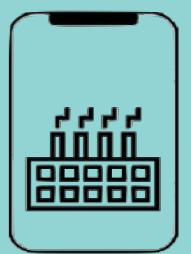


# Push() method

**customStack = [1,2]**

**customStack.push(3)**

				2						
				1						

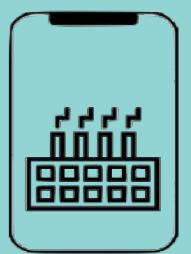


# Push() method

**customStack = [1,2,3]**

**customStack.push(3)**

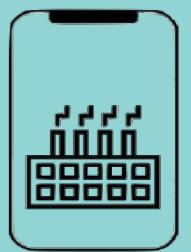
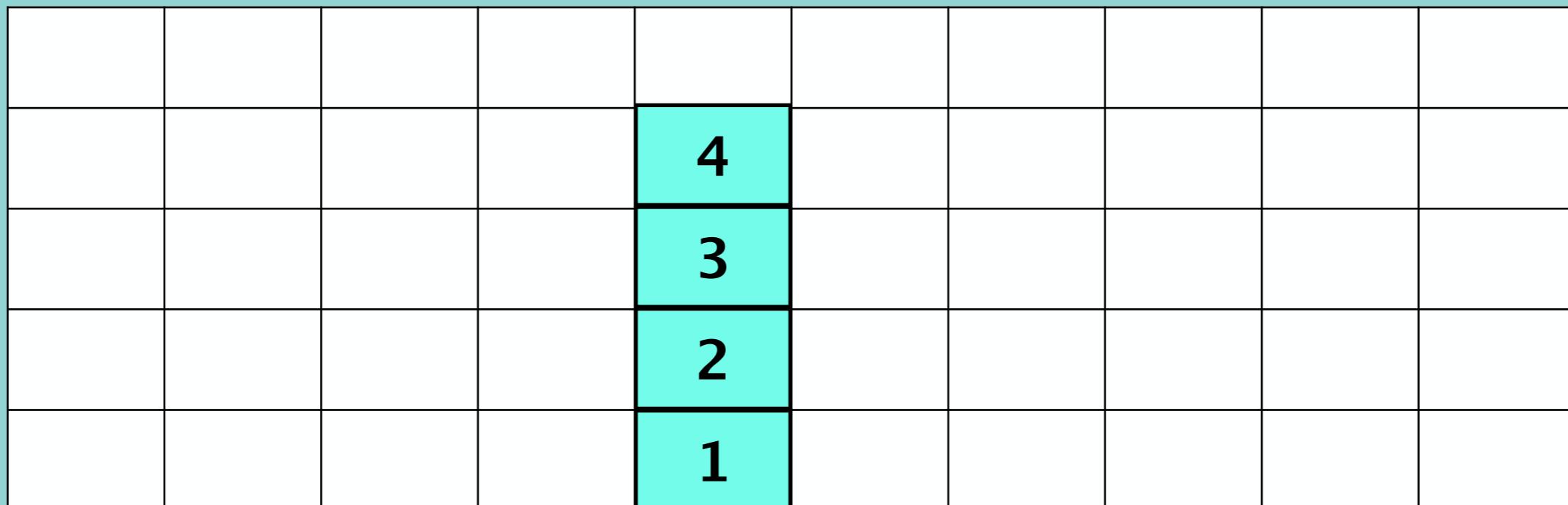
					3					
					2					
					1					



# Push() method

**customStack = [1,2,3,4]**

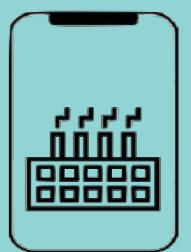
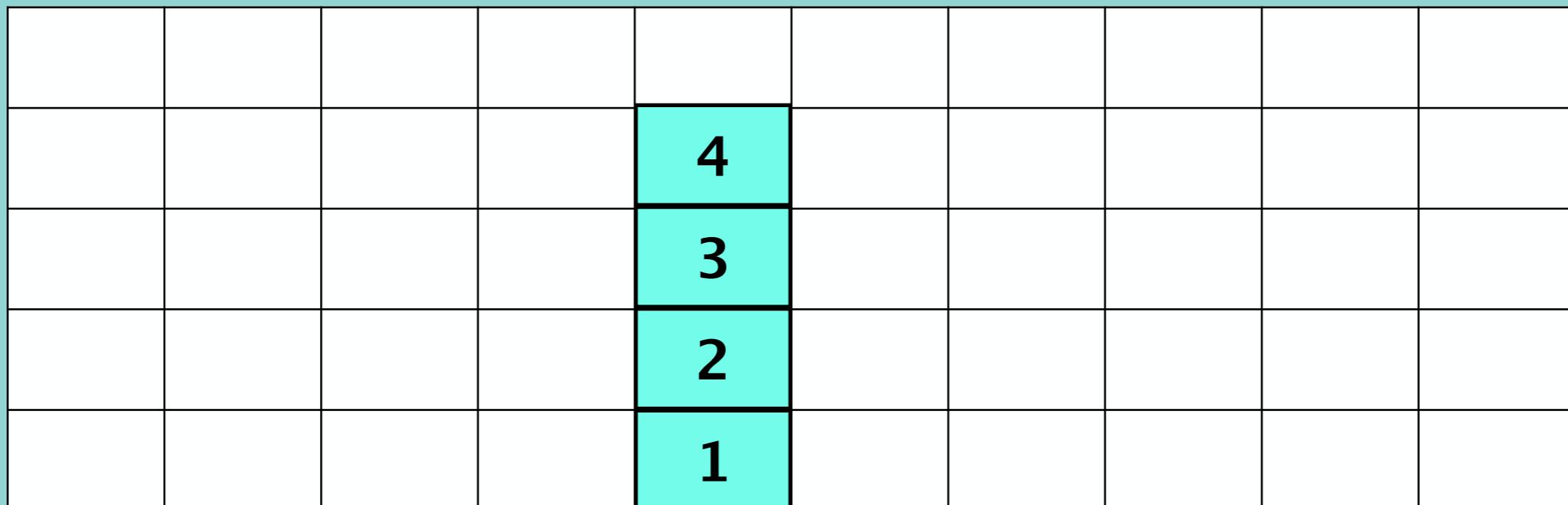
**customStack.push(4)**



## Pop() method

**customStack = [1,2,3]4]**

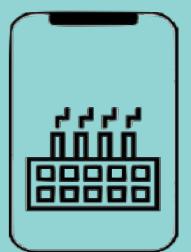
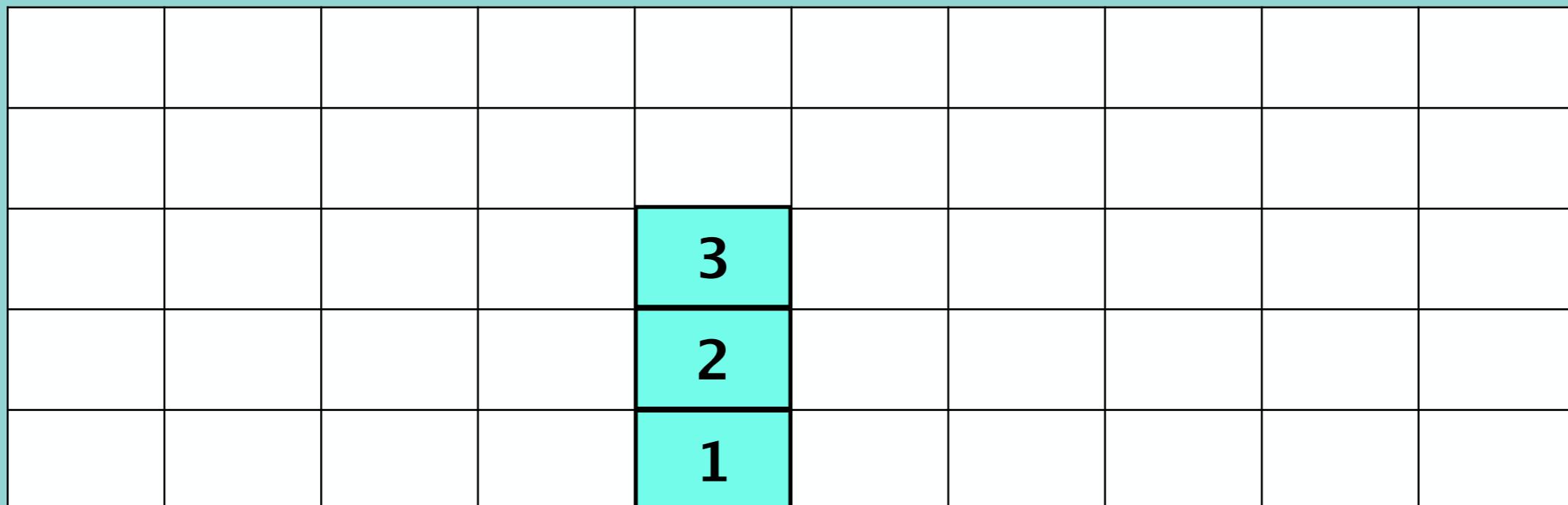
**customStack.pop () —→ 4**



## Pop() method

**customStack = [1,2]3]**

**customStack.pop () —→ 3**

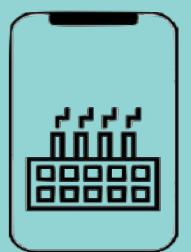


## Pop() method

**customStack = [1]2**

**customStack.pop () → 2**

				2						
				1						

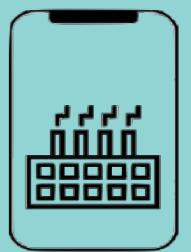


## Pop() method

**customStack = [1]**

**customStack.pop() —→ 1**

				1					



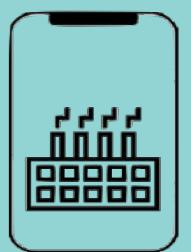
## Pop() method

```
customStack = []
```

```
customStack.pop
```



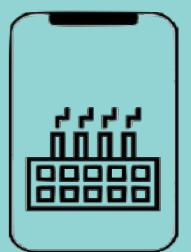
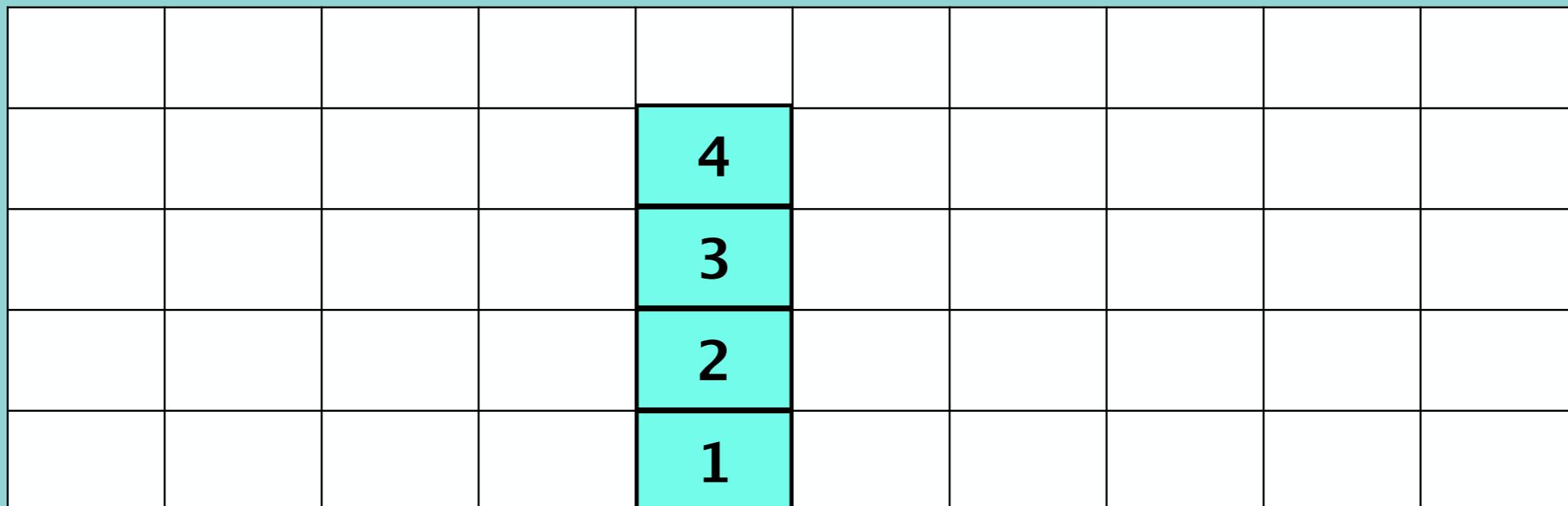
The stack is Empty

## Peek() method

**customStack = [1,2,3,4]**

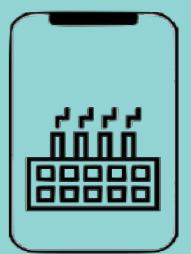
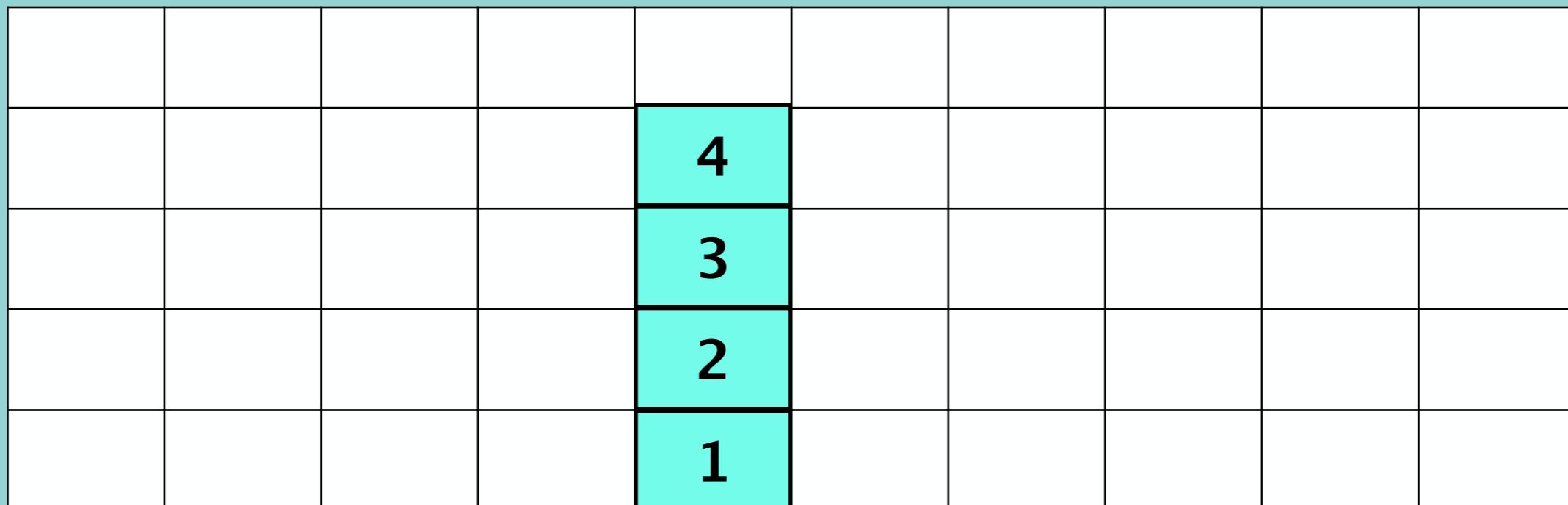
**customStack.peek()** → 4



## **isEmpty() method**

**customStack = [1,2,3,4]**

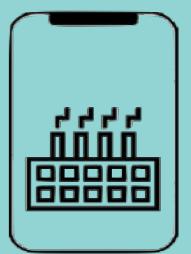
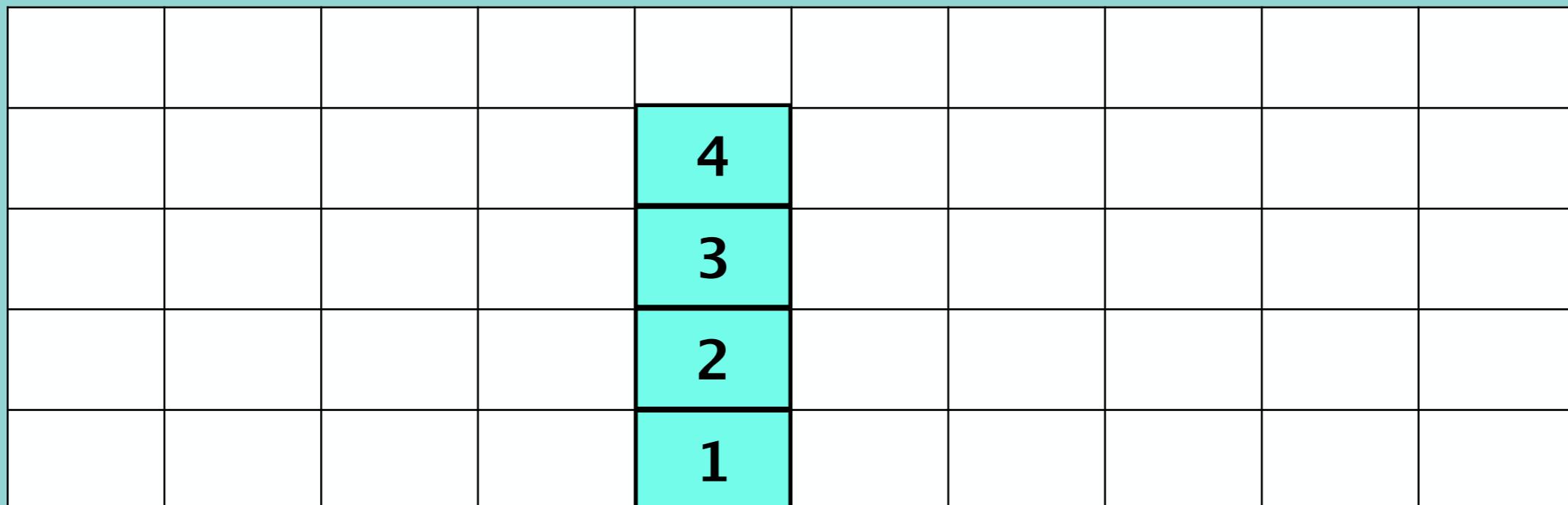
**customStack.isEmpty()** → **False**



## isFull() method

**customStack = [1,2,3,4]**

**customStack.isFull()** → **False**

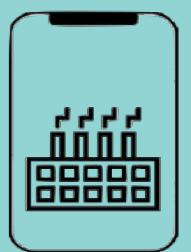


## **delete() method**

**customStack = [1,2,3,4]**

**customStack.delete()**

					4						
					3						
					2						
					1						



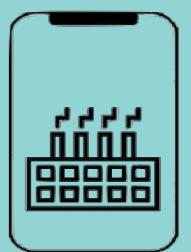
# Stack creation

## Stack using List

- Easy to implement
- Speed problem when it grows

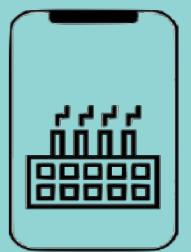
## Stack using Linked List

- Fast Performance
- Implementation is not easy



# Time and Space complexity of Stack operations with List

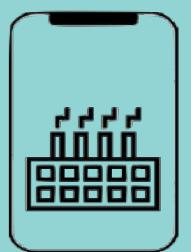
	Time complexity	Space complexity
Create Stack	O(1)	O(1)
Push	O(1) / O( $n^2$ )	O(1)
Pop	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
Delete Entire Stack	O(1)	O(1)



# Stack using Linked List

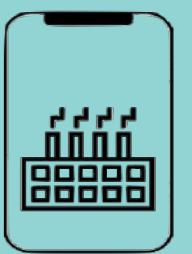
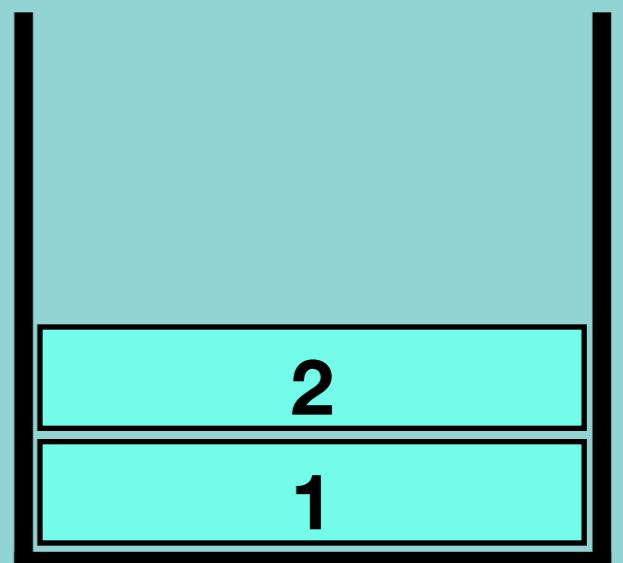
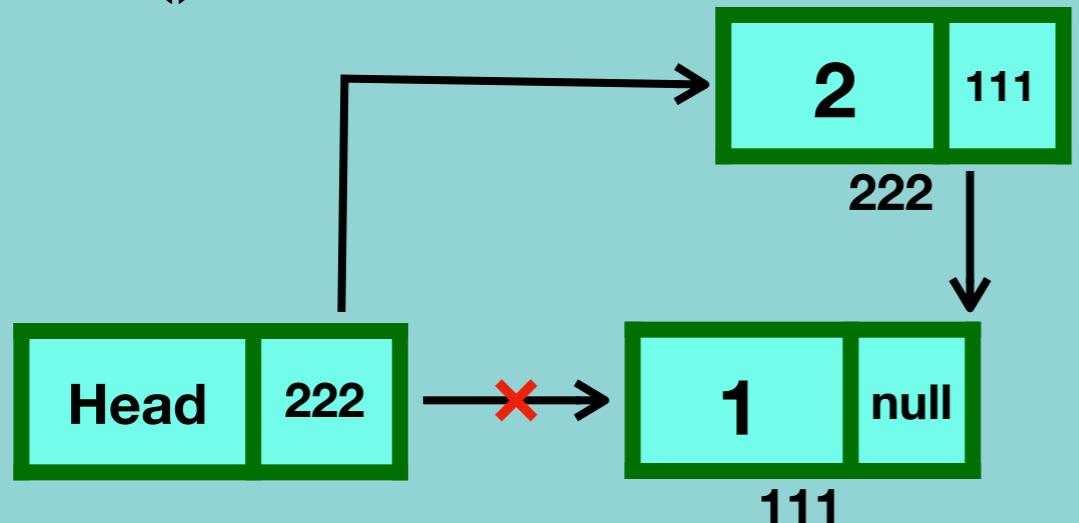
## Create a Stack

Create an object of Linked List class



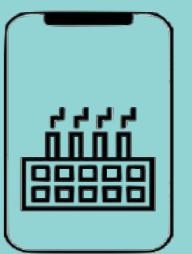
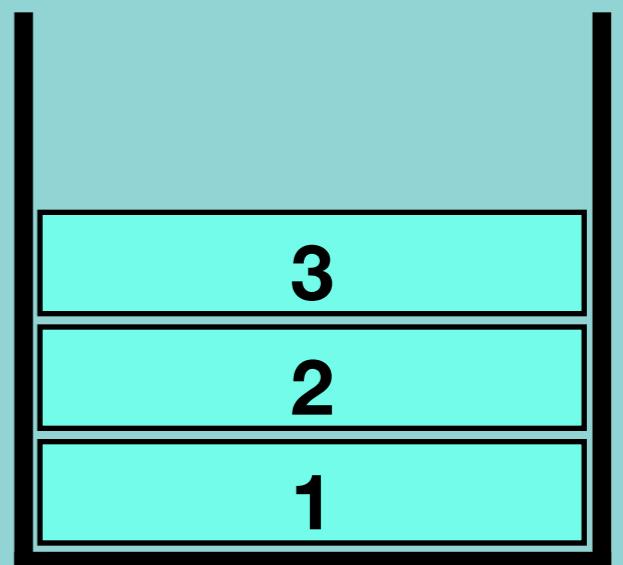
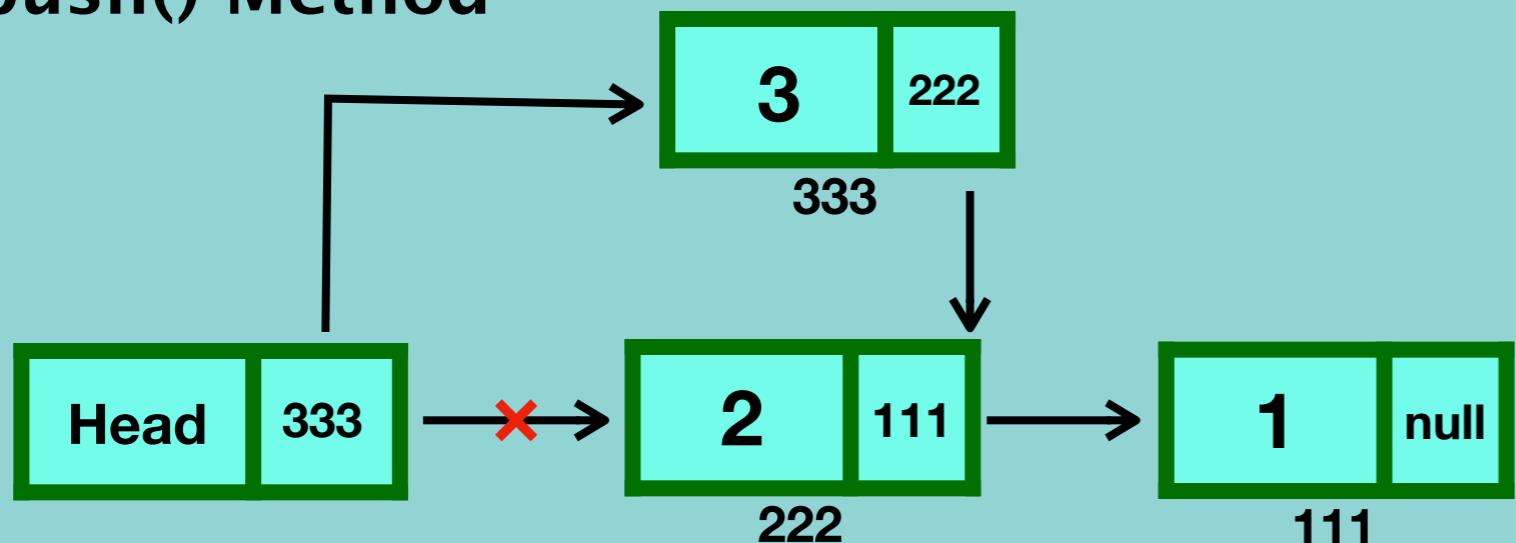
# Stack using Linked List

## push() Method



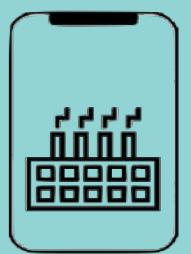
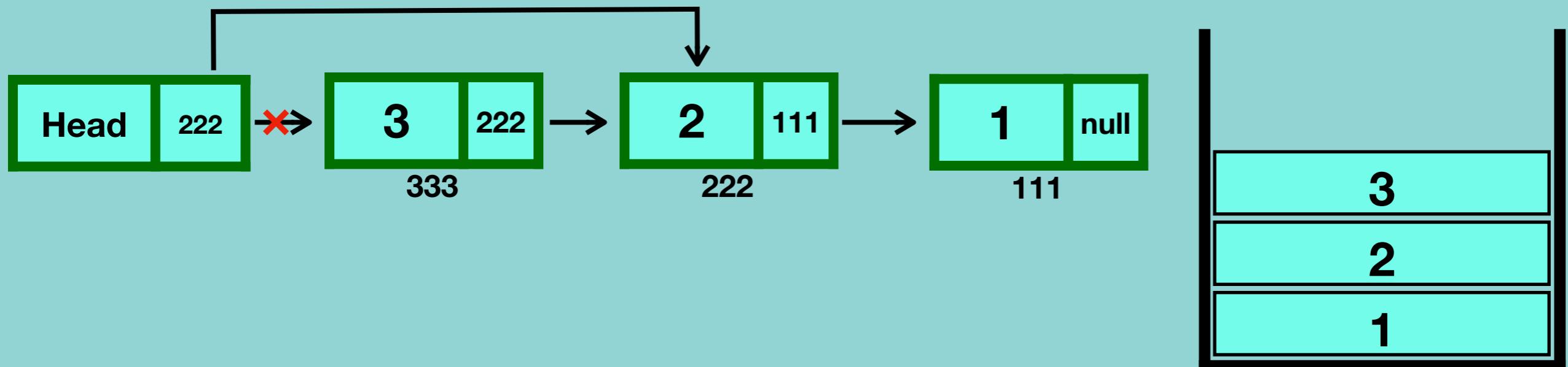
# Stack using Linked List

## push() Method



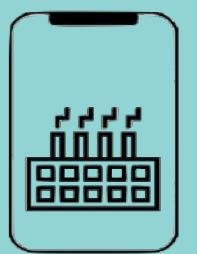
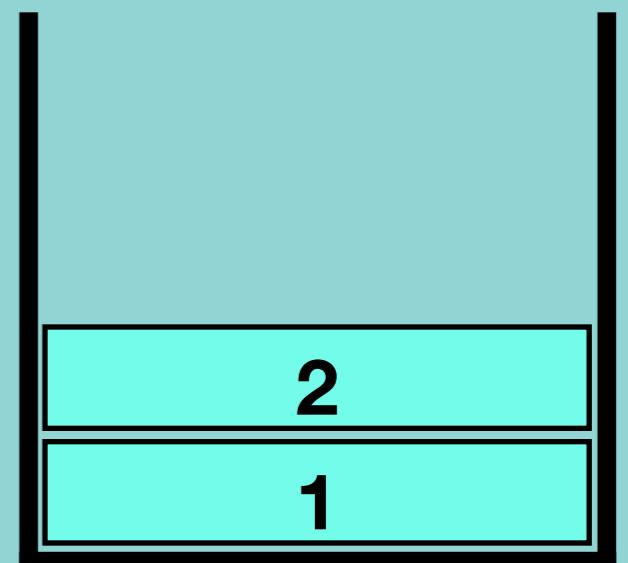
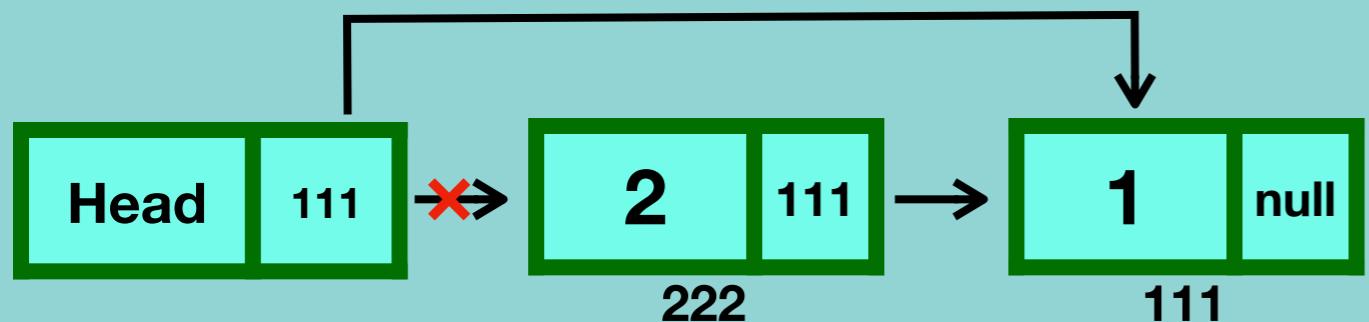
# Stack using Linked List

## pop() Method



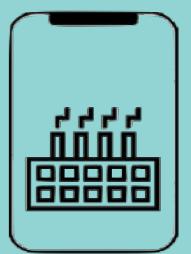
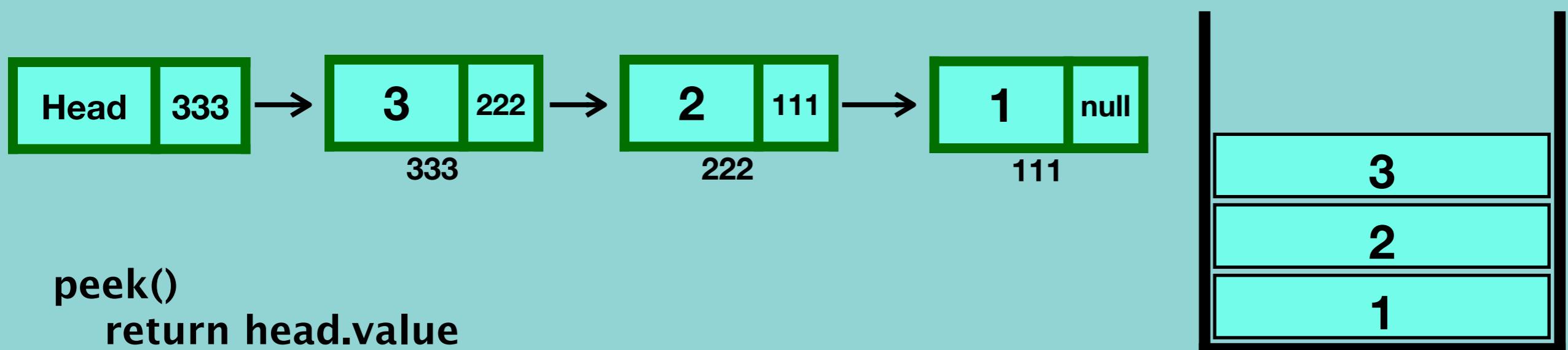
# Stack using Linked List

## pop() Method



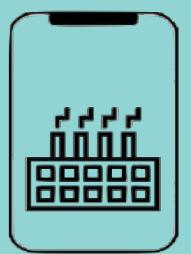
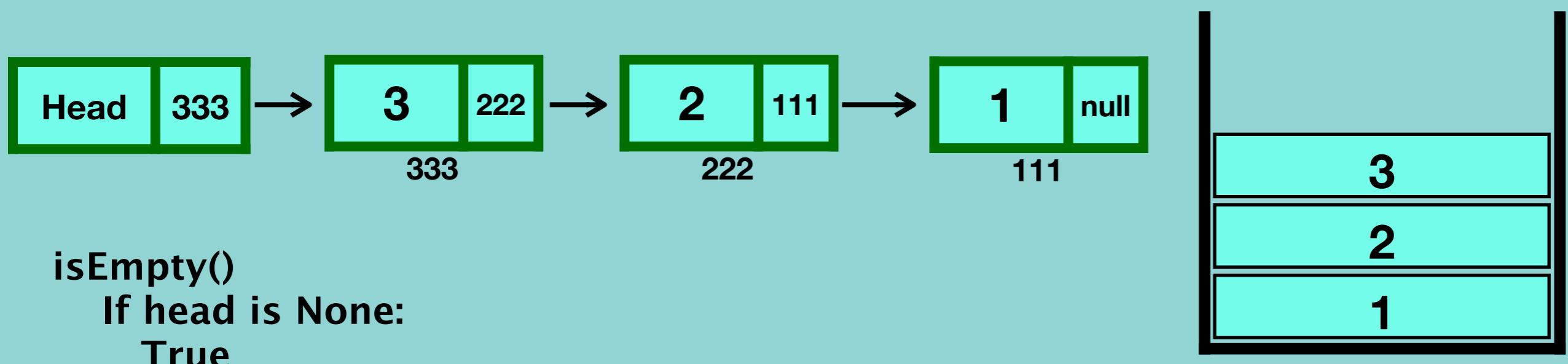
# Stack using Linked List

## peek() Method



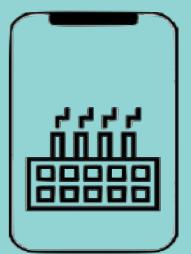
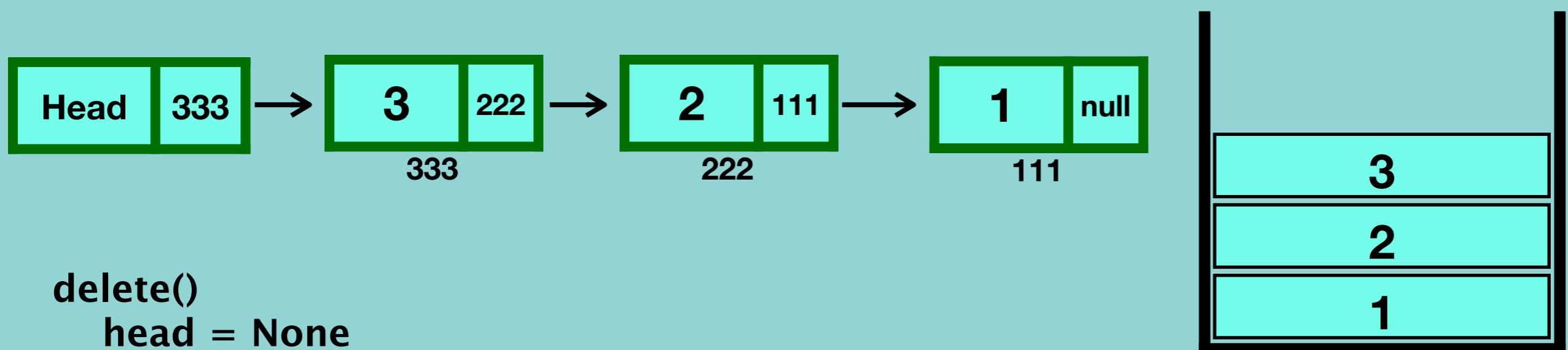
# Stack using Linked List

## isEmpty() Method



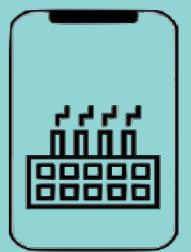
# Stack using Linked List

## 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)



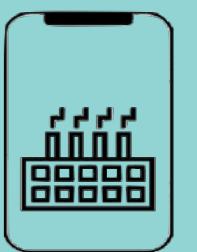
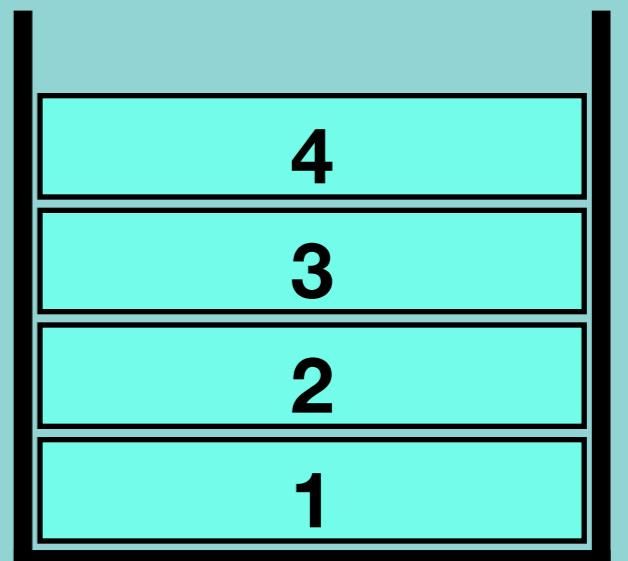
# When to use / avoid Stack

## Use:

- LIFO functionality
- The chance of data corruption is minimum

## Avoid:

- Random access is not possible



# What is a Queue?

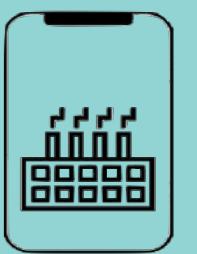
**Queue is a data structure that stores items in a First-In/First-Out manner.**



A new addition to this queue happens at the end of the queue.

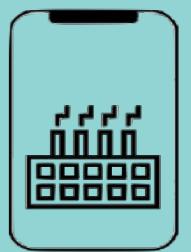
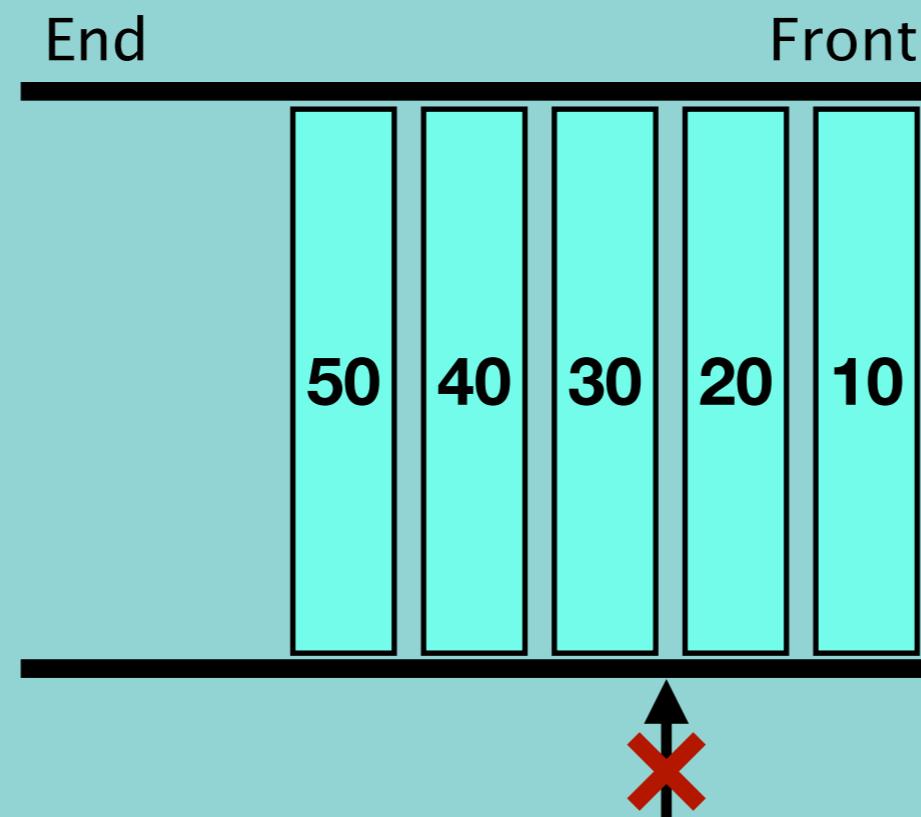
First person in the queue will be served first

FIFO method - First in First Out



# What is a Queue?

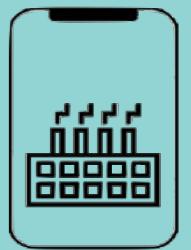
Queue is a data structure that stores items in a First-In/First-Out manner.



# What we need Queue Data Structure?

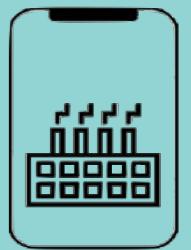
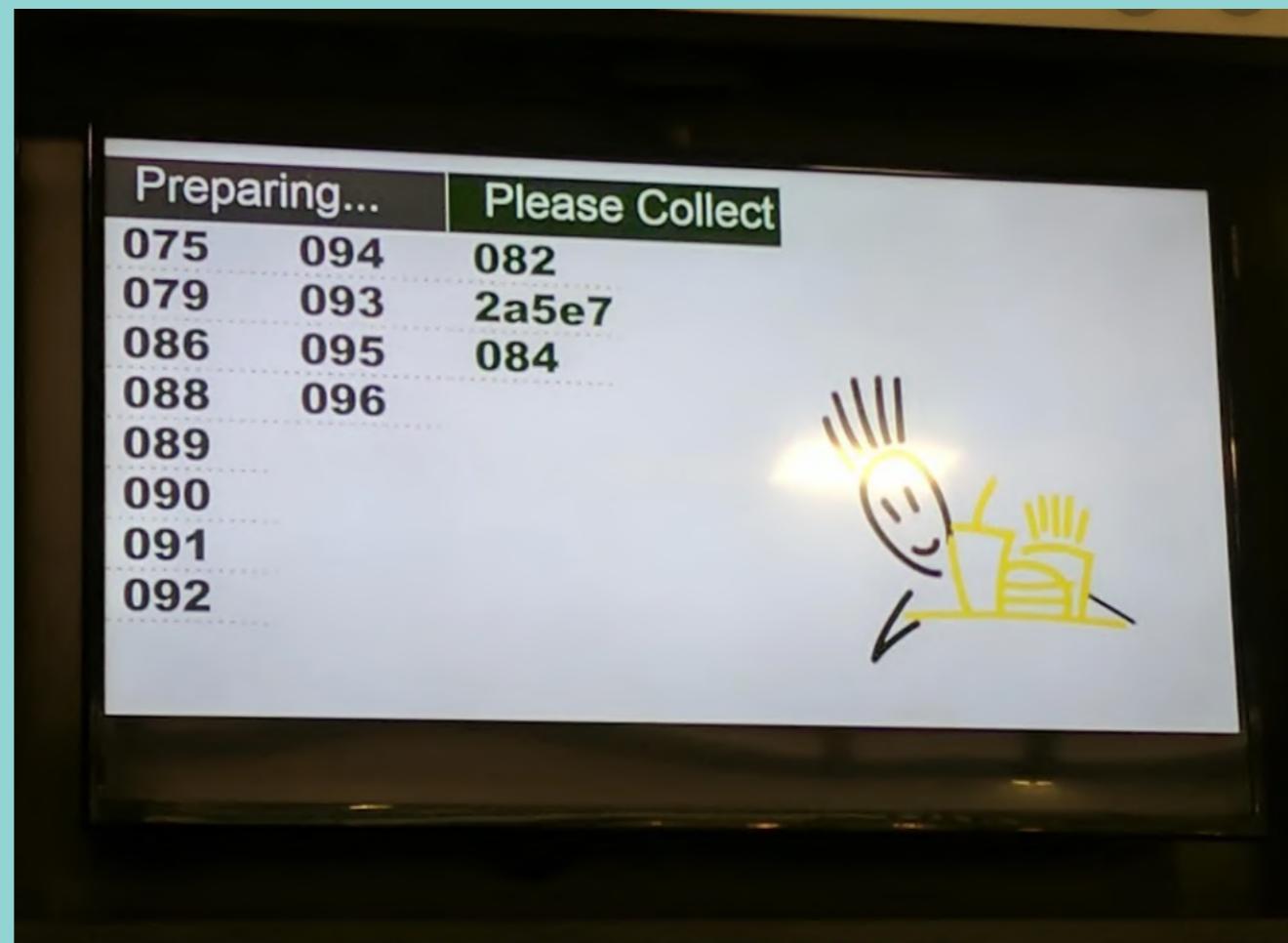
**Utilize first coming data first , while others wait for their turn.**

**FIFO method - First In First Out**



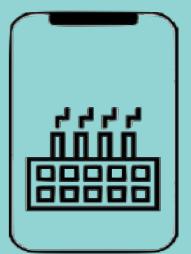
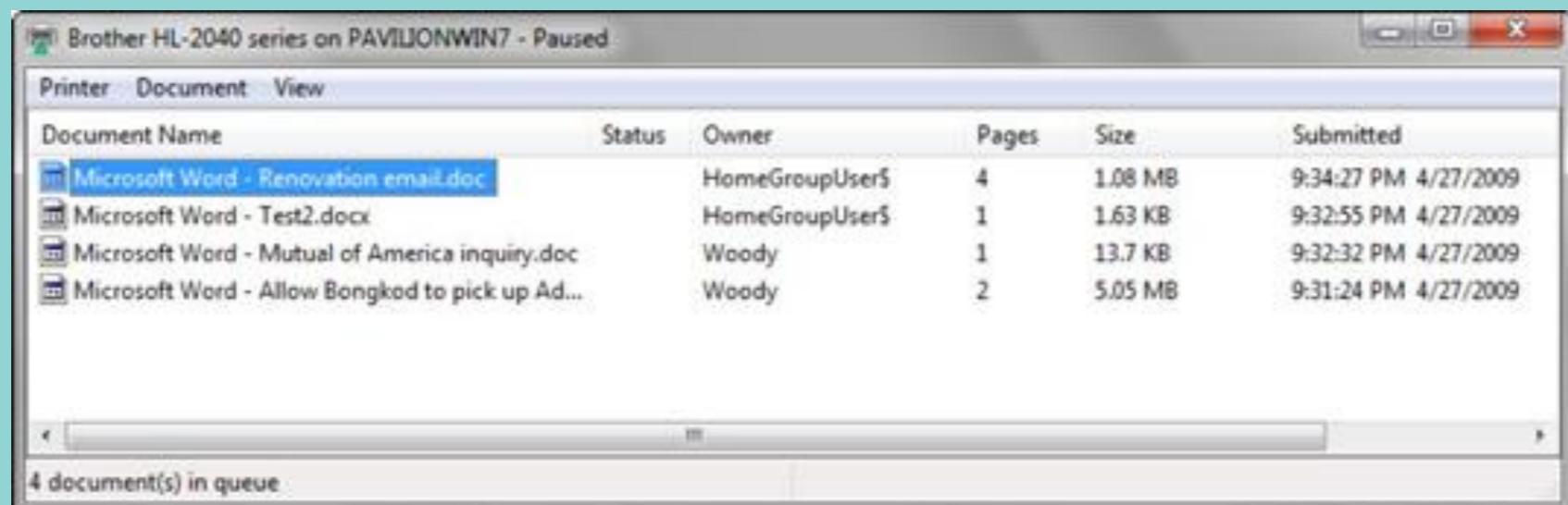
# What we need Queue Data Structure?

## Point sale system of a restaurant



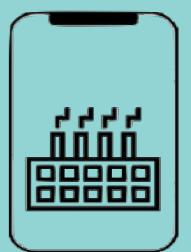
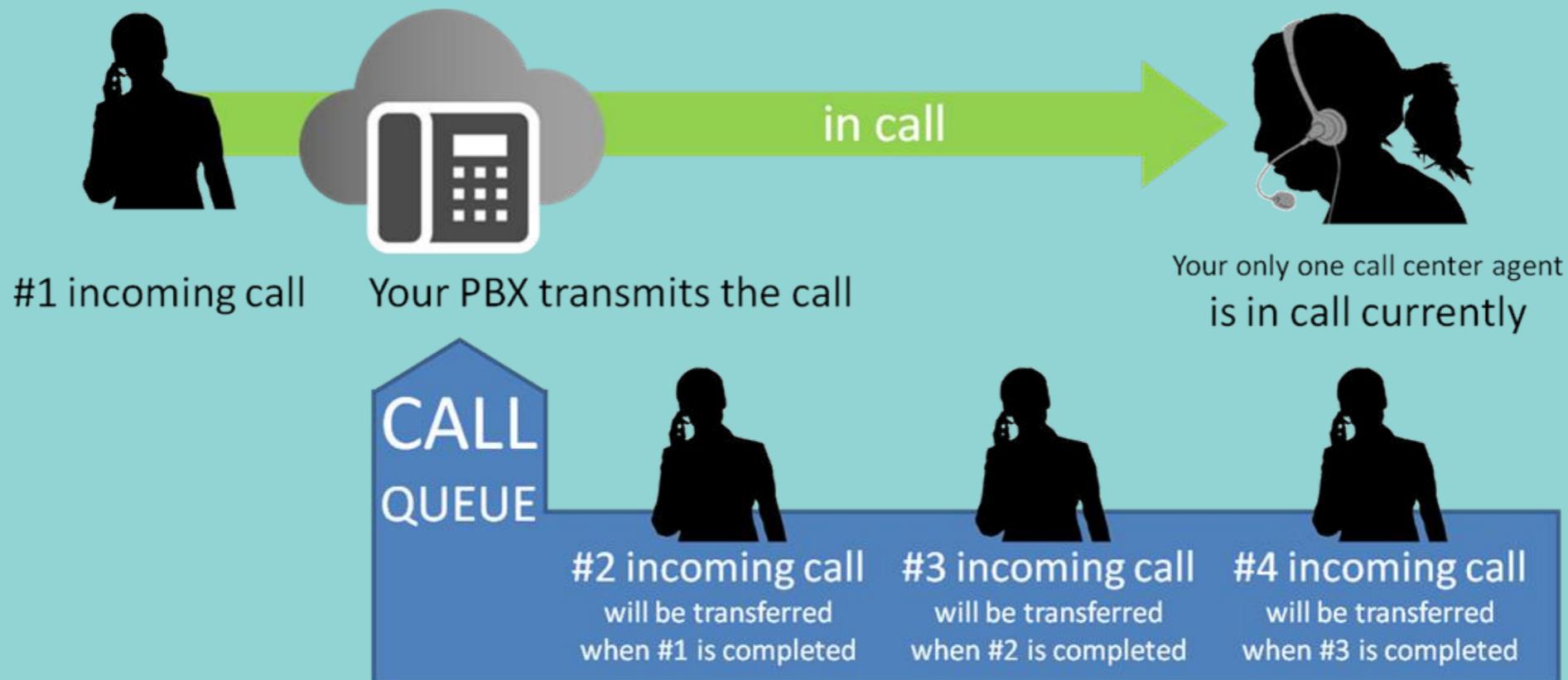
# What we need Queue Data Structure?

## Printer queue



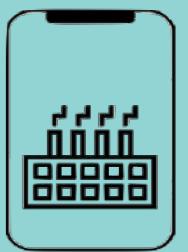
# What we need Queue Data Structure?

## Call center phone systems



# Queue Operations

- Create Queue
- Enqueue
- Dequeue
- Peek
- isEmpty
- isFull
- deleteQueue



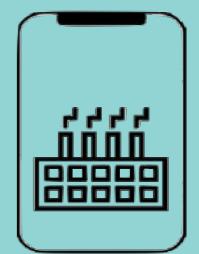
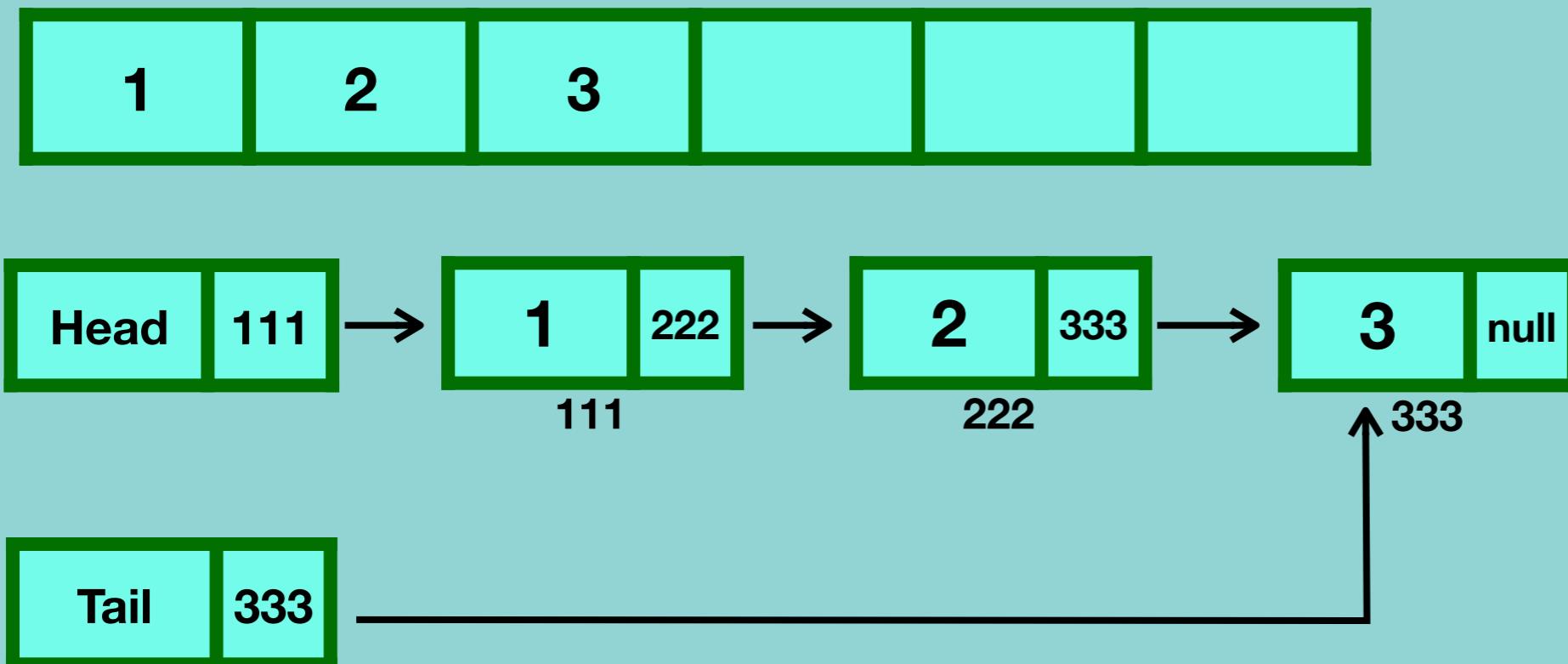
# Queue Operations

## Implementation

### 1. Python List

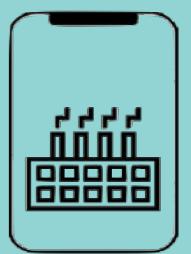
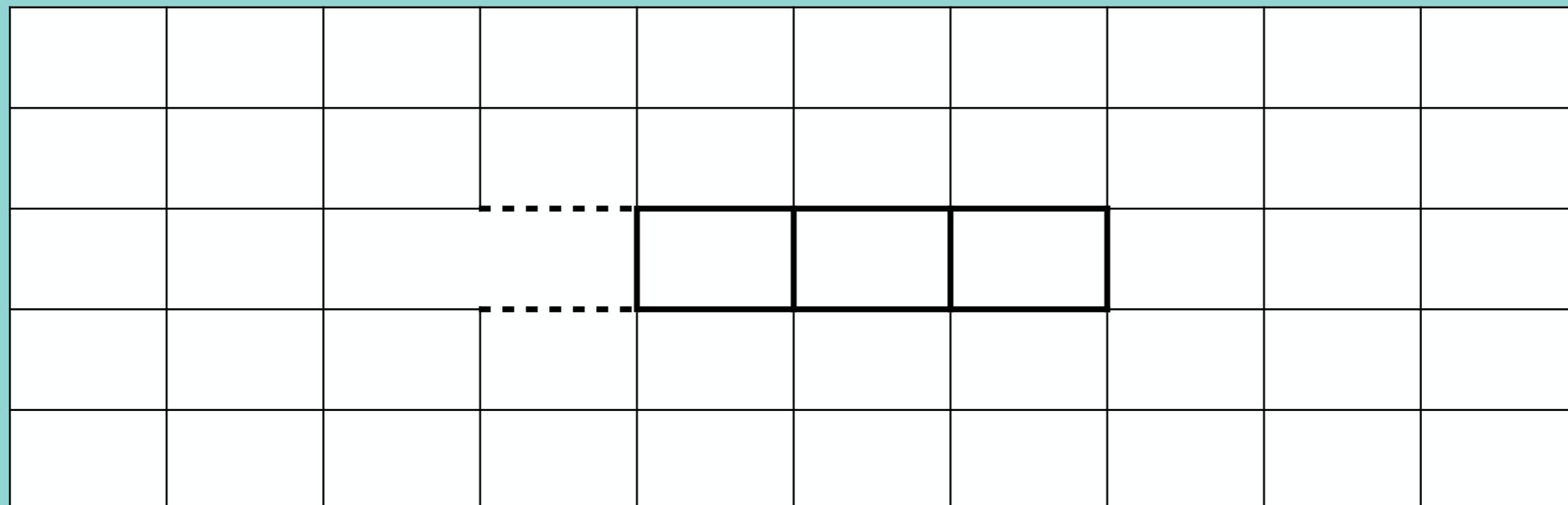
- Queue without capacity
- Queue with capacity (Circular Queue)

### 2. Linked List



# Create Queue using Python List no capacity

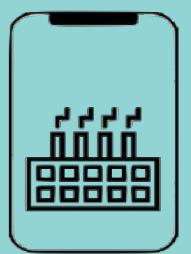
```
customQueue = [ ]
```



# Enqueue() method

```
customQueue = [ ]
```

```
customQueue.enqueue(1)
```

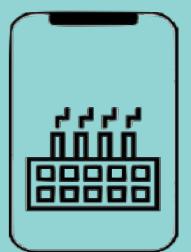



# Enqueue() method

**customQueue = [1]**

**customQueue.enqueue(2)**

						1			

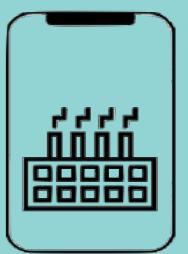


# Enqueue() method

**customQueue = [1,2]**

**customQueue.enqueue(3)**

					2	1				

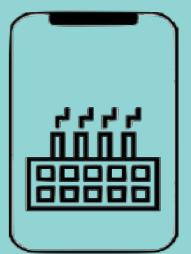


# Enqueue() method

**customQueue = [1,2,3]**

**customQueue.enqueue(4)**

				3	2	1				

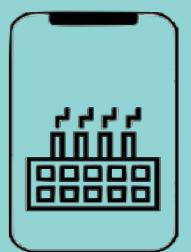


# Enqueue() method

**customQueue = [1,2,3,4]**

**customQueue.enqueue(4)**

			4	3	2	1				

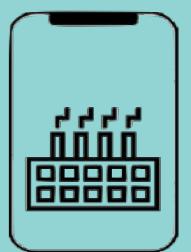


## Dequeue() method

**customQueue = [1,2,3]4]**

**customQueue.dequeue () → 1**

			4	3	2	1				

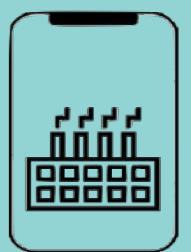


## Dequeue() method

**customQueue = [3,3]4]**

**customQueue.dequeue () → 2**

			4	3	2					

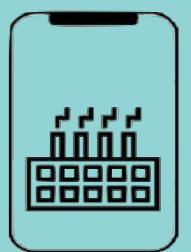


## Dequeue() method

**customQueue = [3]4**

**customQueue.dequeue () → 3**

			4	3						

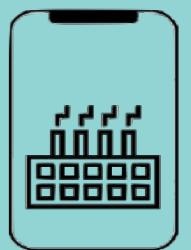


## Dequeue() method

**customQueue = [4]**

**customQueue.dequeue () → 4**

			4						



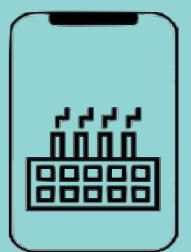
# Dequeue() method

```
customQueue = []
```

```
customQueue.deq
```



The queue is Empty

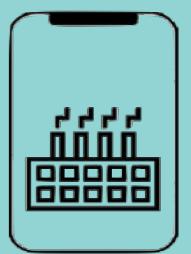



## Peek() method

**customQueue = [1,2,3,4]**

**customQueue.peek()** → 1

			4	3	2	1				

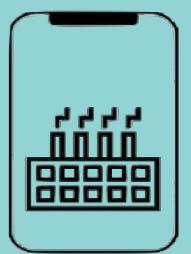


## **isEmpty() method**

**customQueue =[1,2,3,4]**

**customQueue.isEmpty()** → **False**

			4	3	2	1			

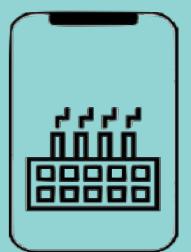


## isFull() method

**customQueue = [1,2,3,4]**

**customQueue.isFull()**

			4	3	2	1			

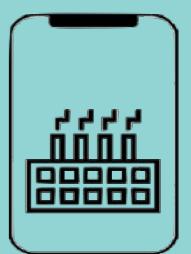


## **delete() method**

**customQueue =[1,2,3,4]**

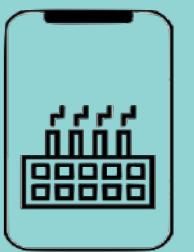
**customQueue.delete()**

			4	3	2	1				



# Time and Space complexity of Queue operations with Python List

	Time complexity	Space complexity
Create Queue	O(1)	O(1)
Enqueue	O(n)	O(1)
Dequeue	O(n)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
Delete Entire Queue	O(1)	O(1)



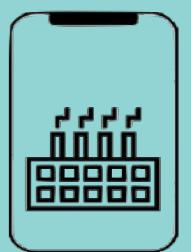
# Queue creation

## Queue using List

- Easy to implement
- Speed problem when it grows

## Queue using Linked List

- Fast Performance
- Implementation is not easy



# Queue with fixed capacity (Circular Queue)

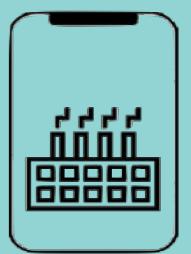
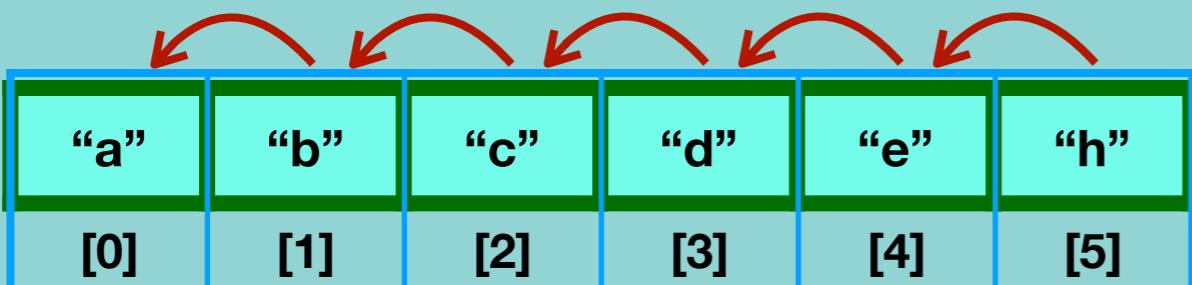
## List relocation

“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]

## Shifting cells left



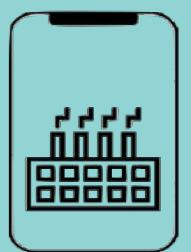
# Queue with fixed capacity (Circular Queue)

## Create Queue

**Size = 6**

**Start = -1**

**Top = -1**



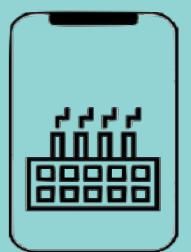
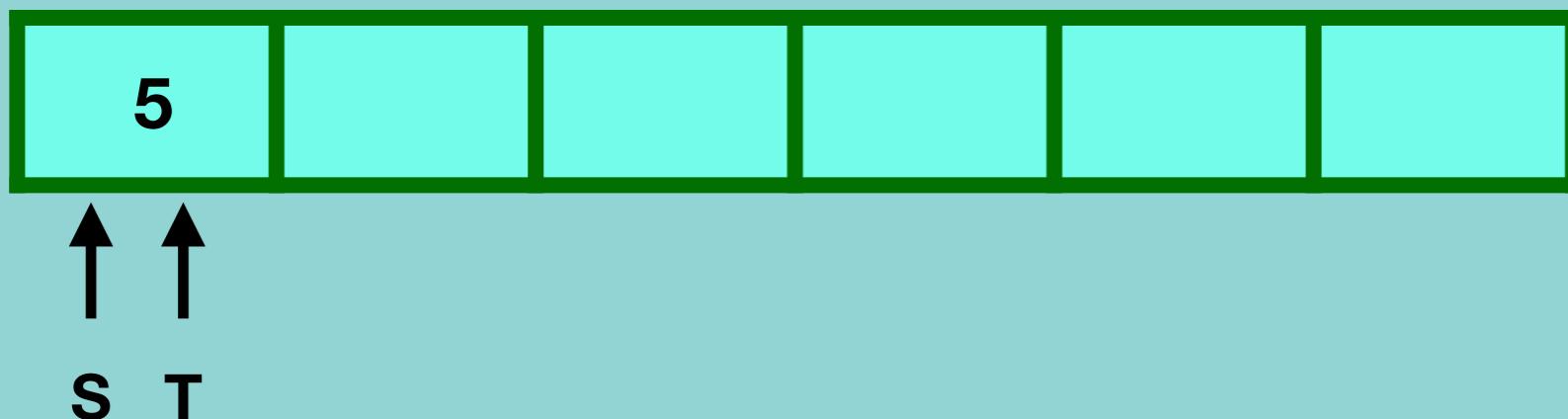
# Queue with fixed capacity (Circular Queue)

## Enqueue and Dequeue Methods

**enqueue(5)**

**Start = 0**

**Top = 0**



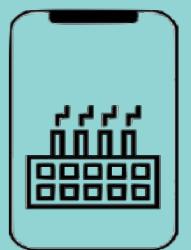
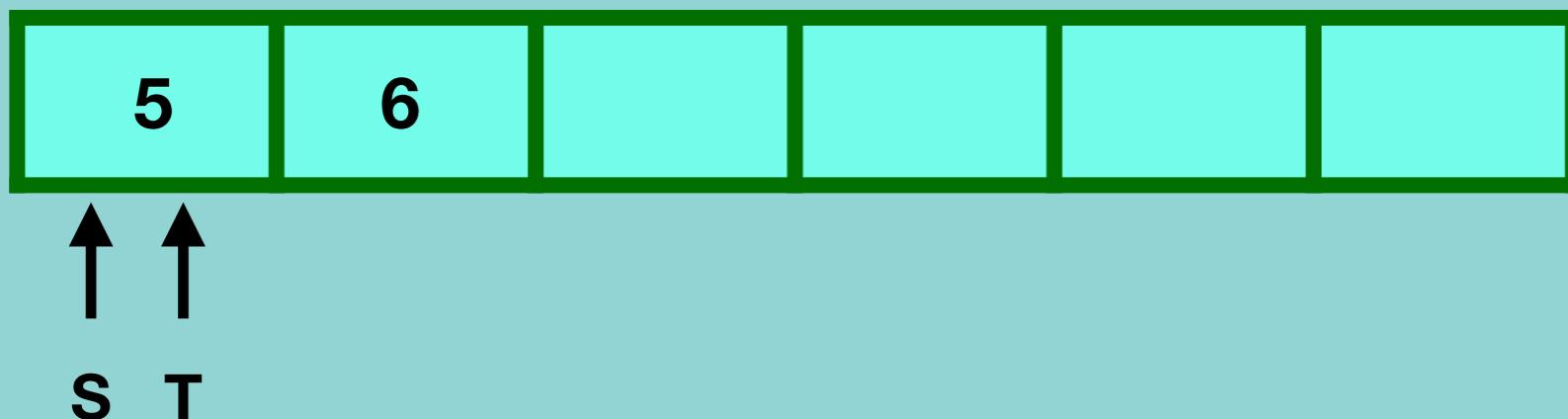
# Queue with fixed capacity (Circular Queue)

## Enqueue and Dequeue Methods

**enqueue(6)**

**Start = 0**

**Top = 0**



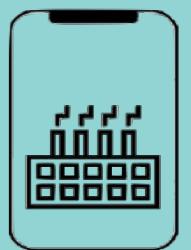
# Queue with fixed capacity (Circular Queue)

## Enqueue and Dequeue Methods

**enqueue(7)**

**Start = 0**

**Top = 2**



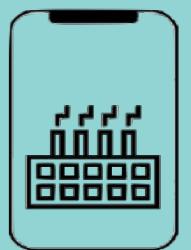
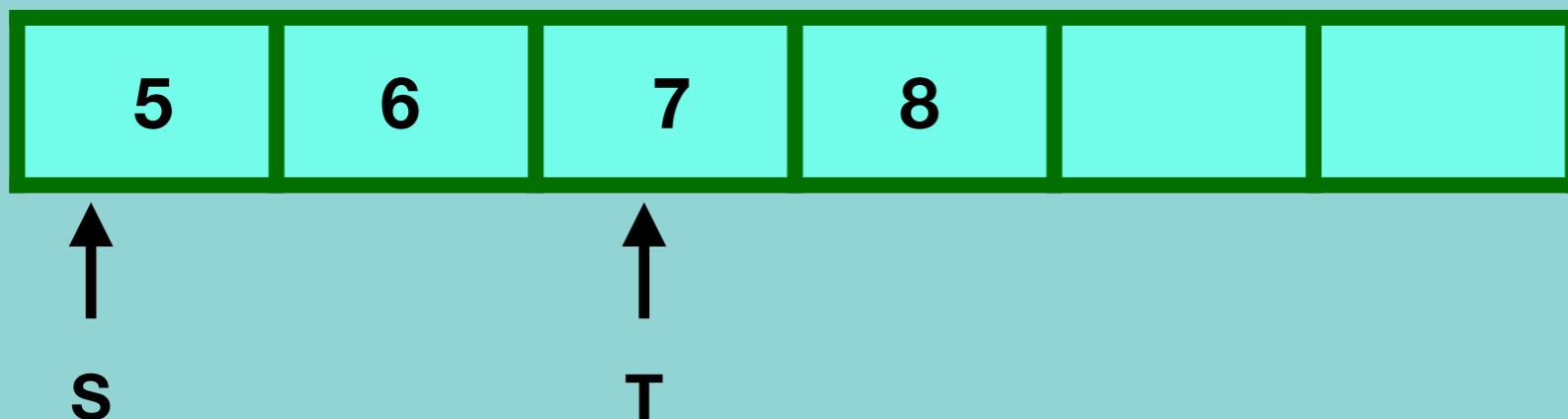
# Queue with fixed capacity (Circular Queue)

## Enqueue and Dequeue Methods

**enqueue(8)**

**Start = 0**

**Top = 3**



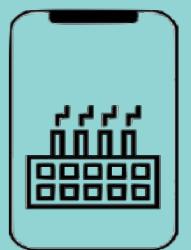
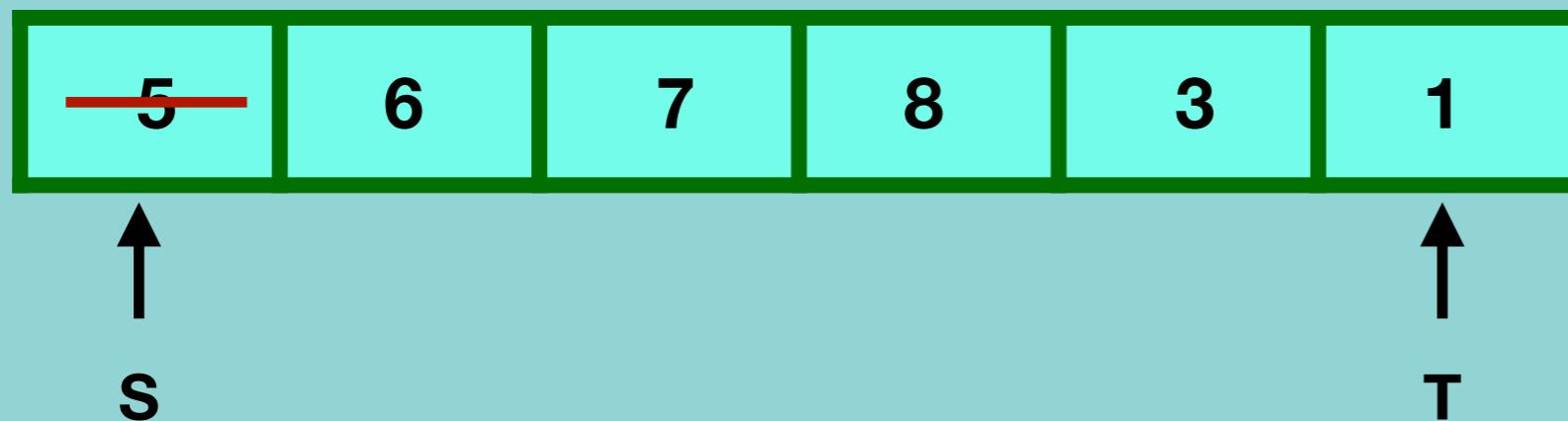
# Queue with fixed capacity (Circular Queue)

## Enqueue and Dequeue Methods

`dequeue() —→ 5`

`Start = 0`

`Top = 5`



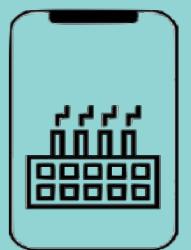
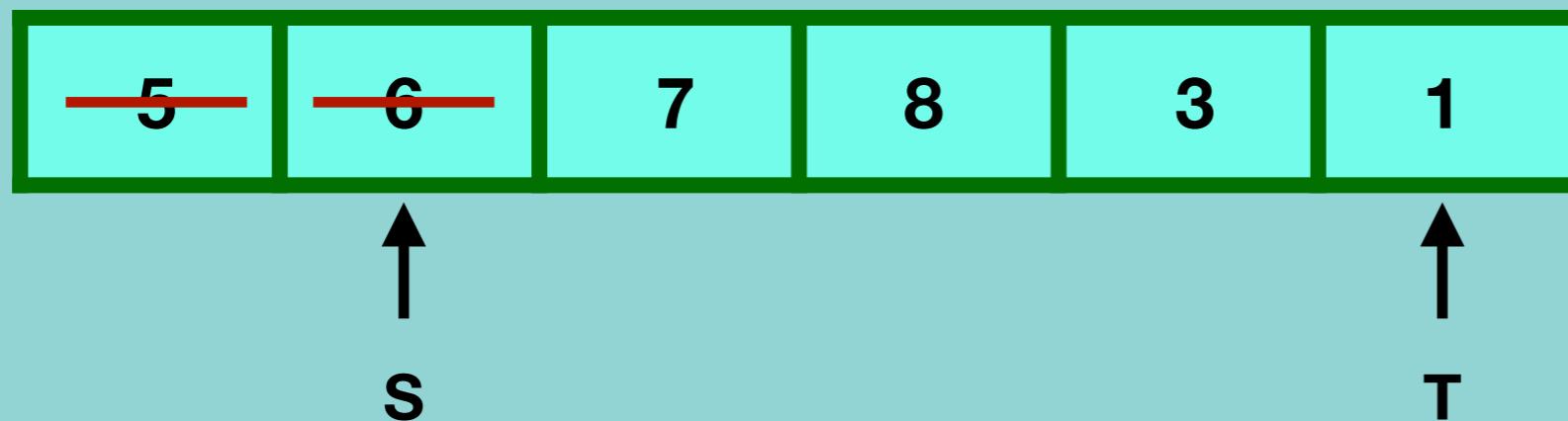
# Queue with fixed capacity (Circular Queue)

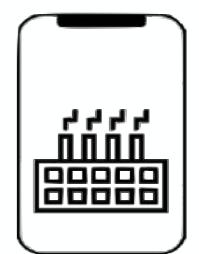
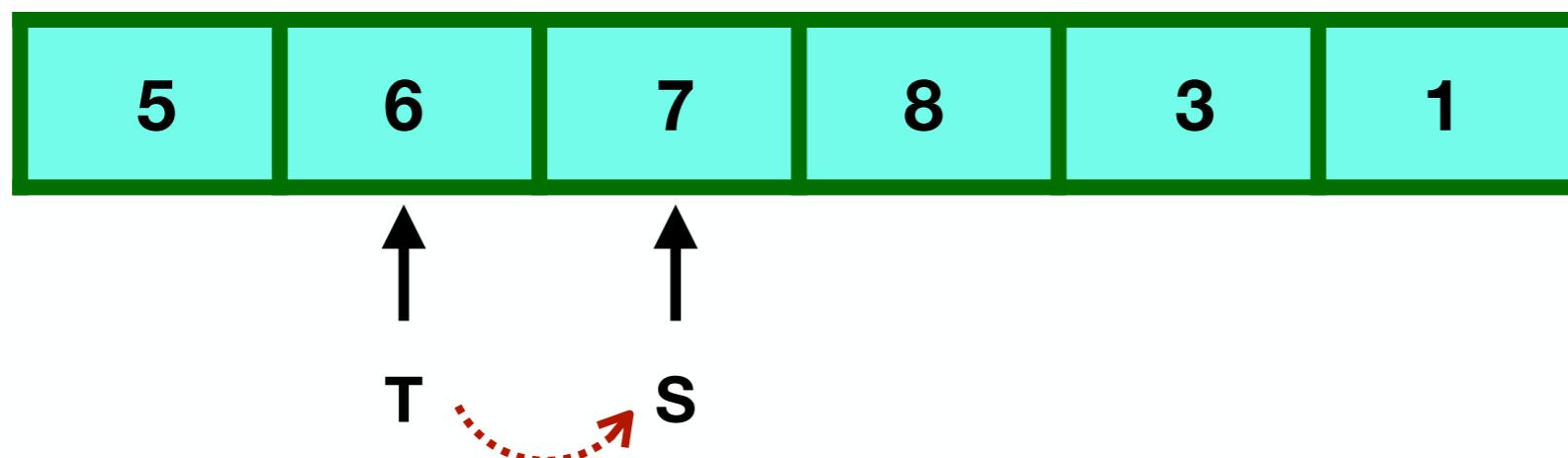
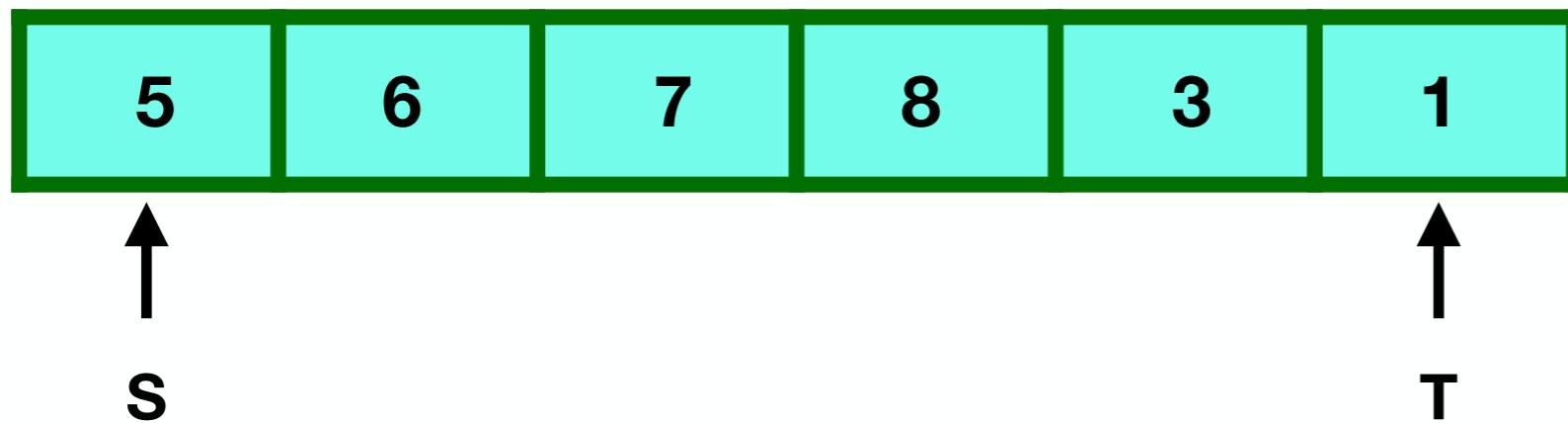
## Enqueue and Dequeue Methods

`dequeue() —→ 6`

`Start = 1`

`Top = 5`





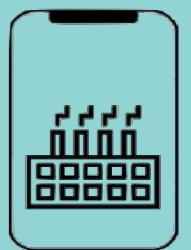
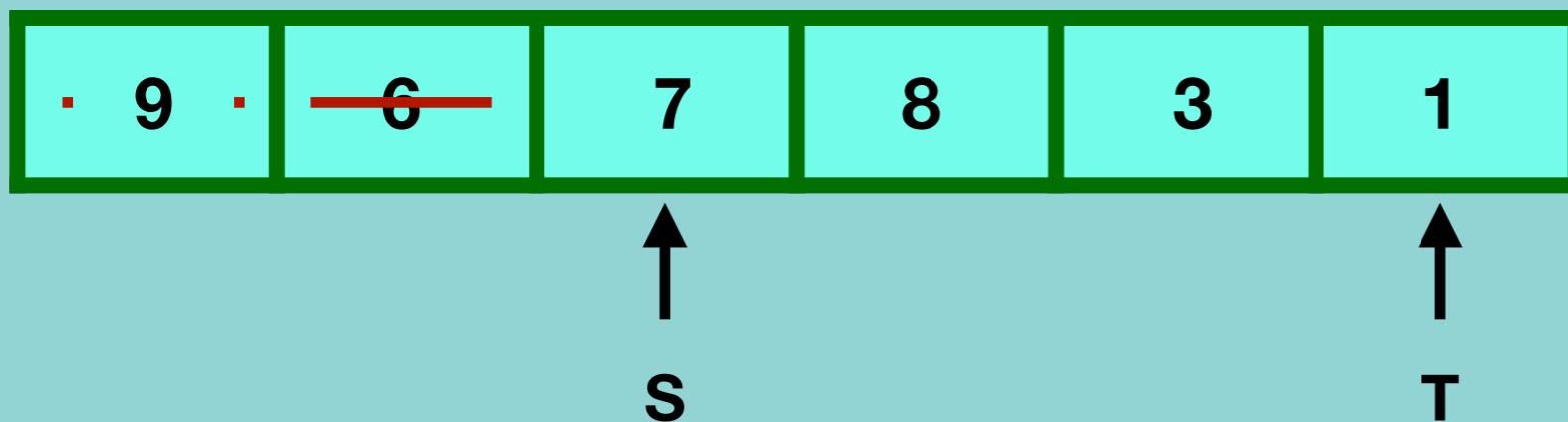
# Queue with fixed capacity (Circular Queue)

## Enqueue and Dequeue Methods

**enqueue(9)**

**Start = 2**

**Top = 0**



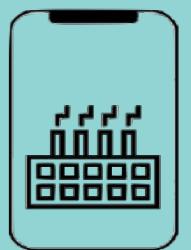
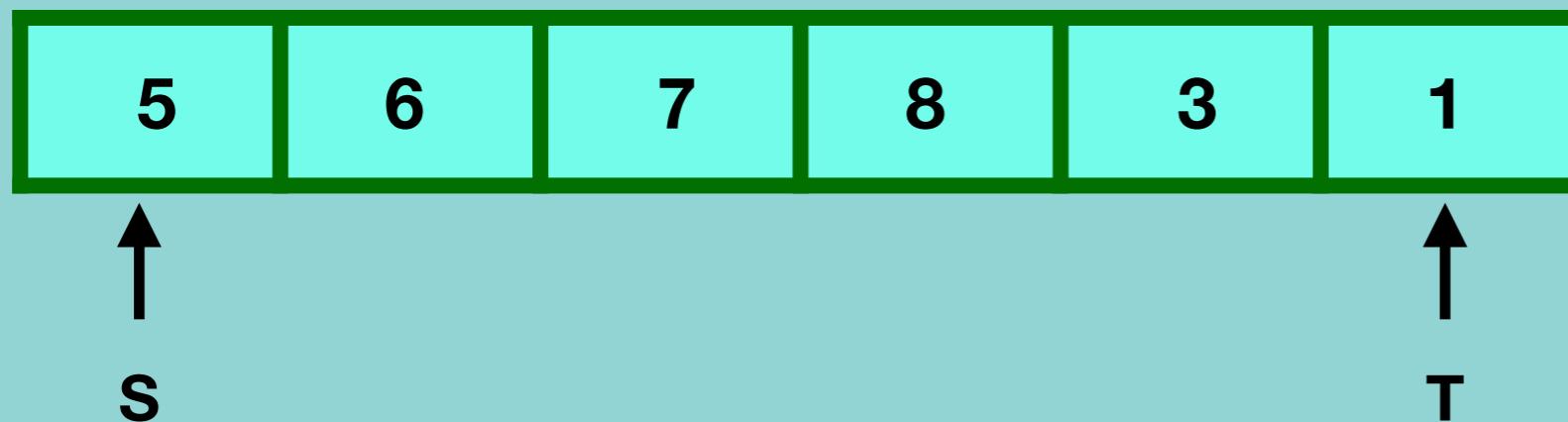
# Queue with fixed capacity (Circular Queue)

## Peek method

`peek()` → 5

`Start = 0`

`Top = 5`



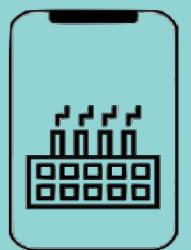
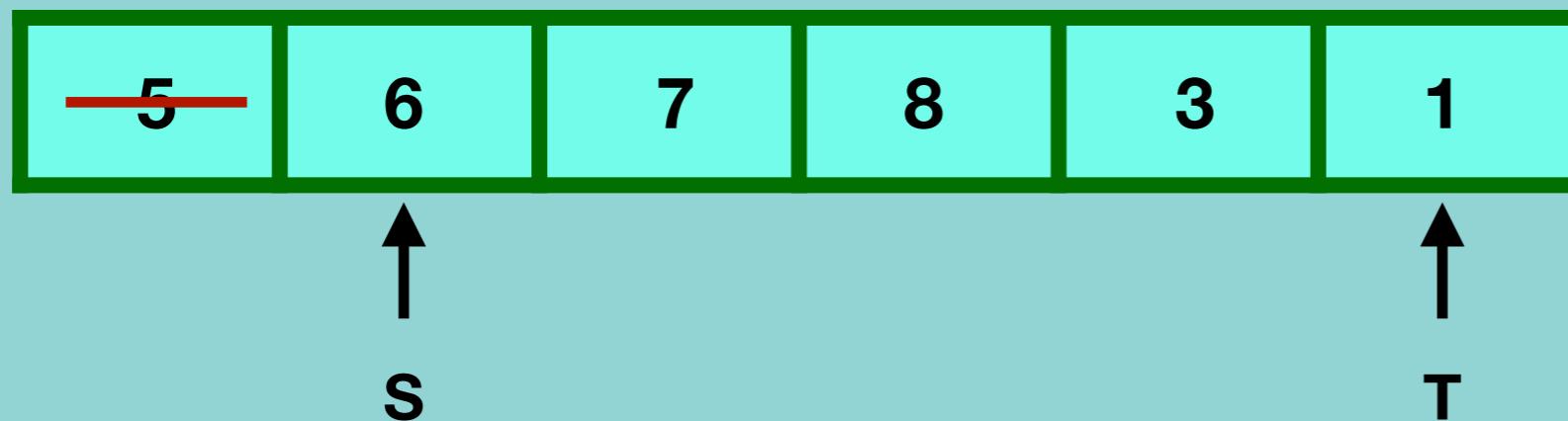
# Queue with fixed capacity (Circular Queue)

## Peek method

peek() —→ 6

Start = 1

Top = 5



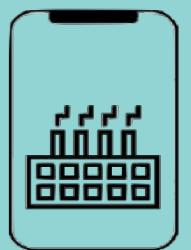
# Queue with fixed capacity (Circular Queue)

## isFull method

isFull() —→ True

Start = 0

Top = 5



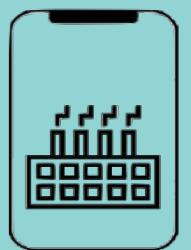
# Queue with fixed capacity (Circular Queue)

## isFull method

isFull() → False

Start = 1

Top = 5



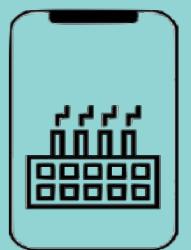
# Queue with fixed capacity (Circular Queue)

## isEmpty method

isEmpty() —→ False

Start = 1

Top = 5



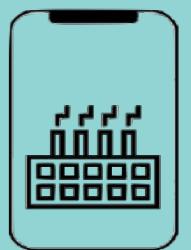
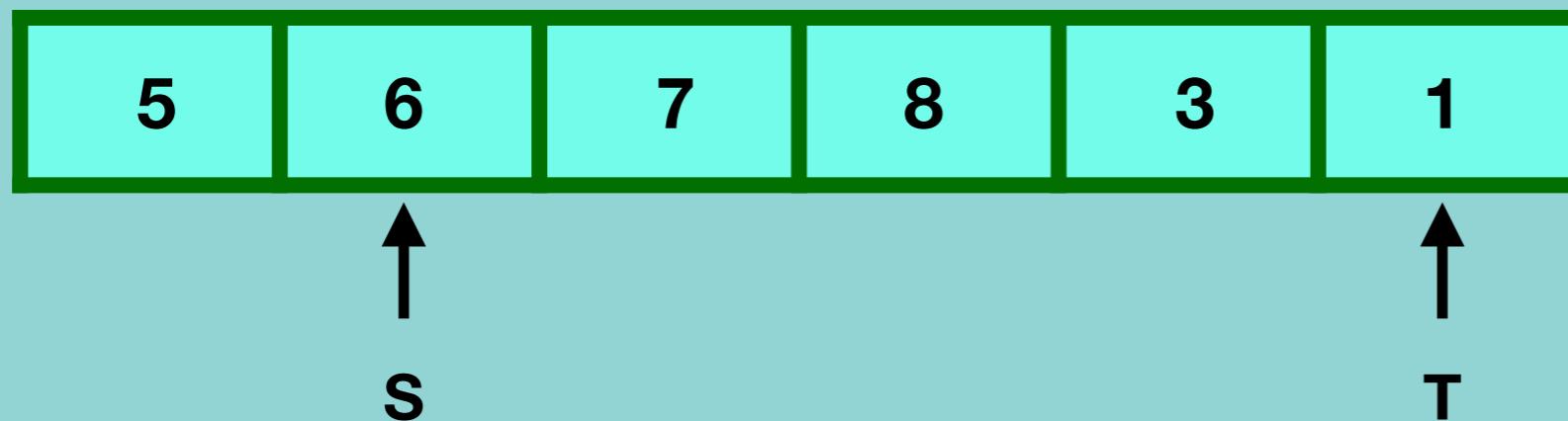
# Queue with fixed capacity (Circular Queue)

## Delete method

**delete()**

**Start = 1**

**Top = 5**



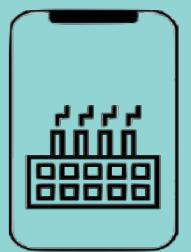
# Queue with fixed capacity (Circular Queue)

## Delete method

**delete()**

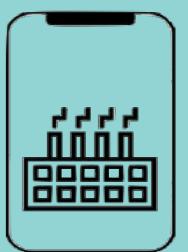
**Start = -1**

**Top = -1**



# Time and Space complexity of Circular Queue operations

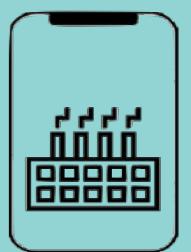
	Time complexity	Space complexity
Create Queue	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
isFull	O(1)	O(1)
Delete Entire Queue	O(1)	O(1)



# Queue using Linked List

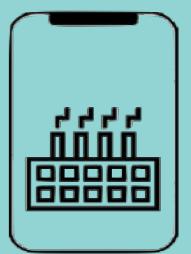
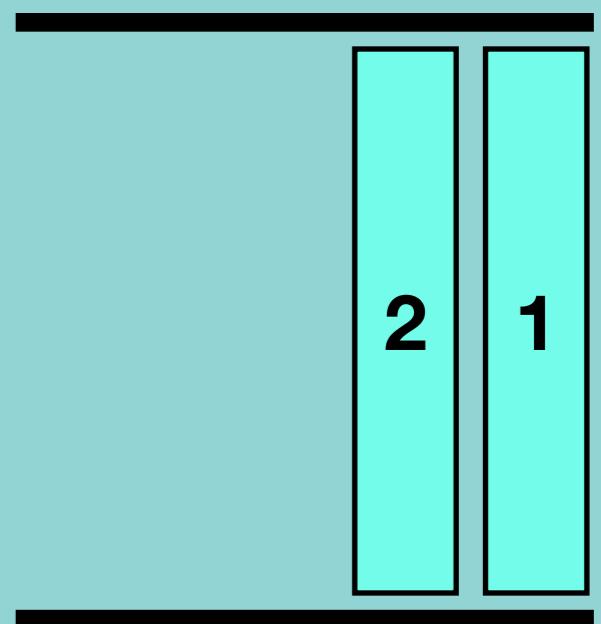
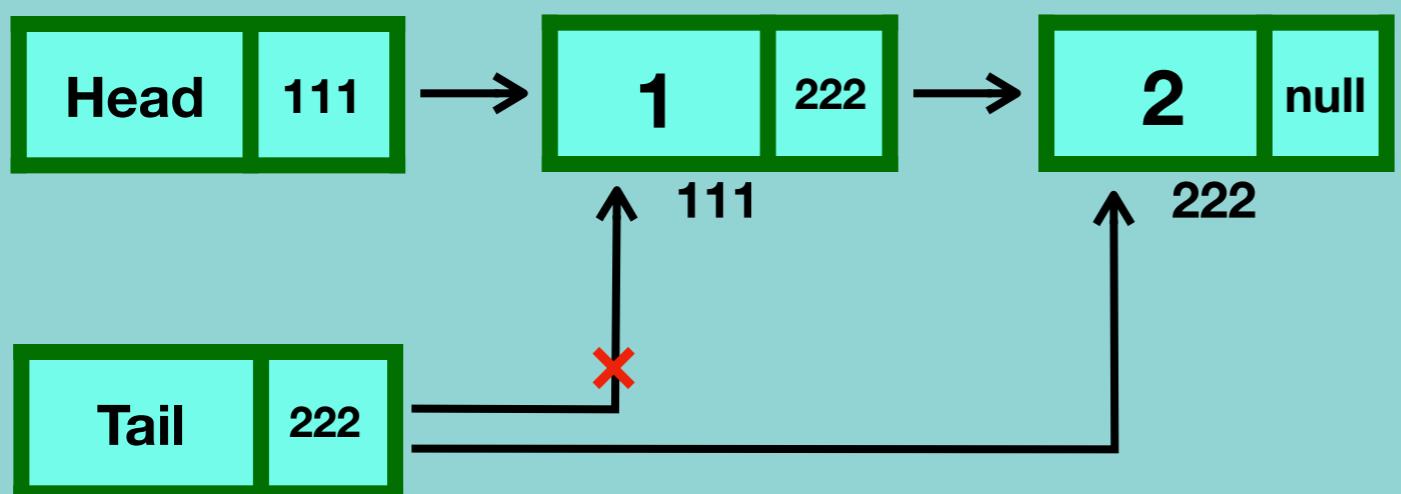
## Create a Queue

Create an object of Linked List class



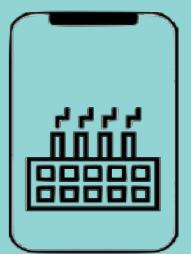
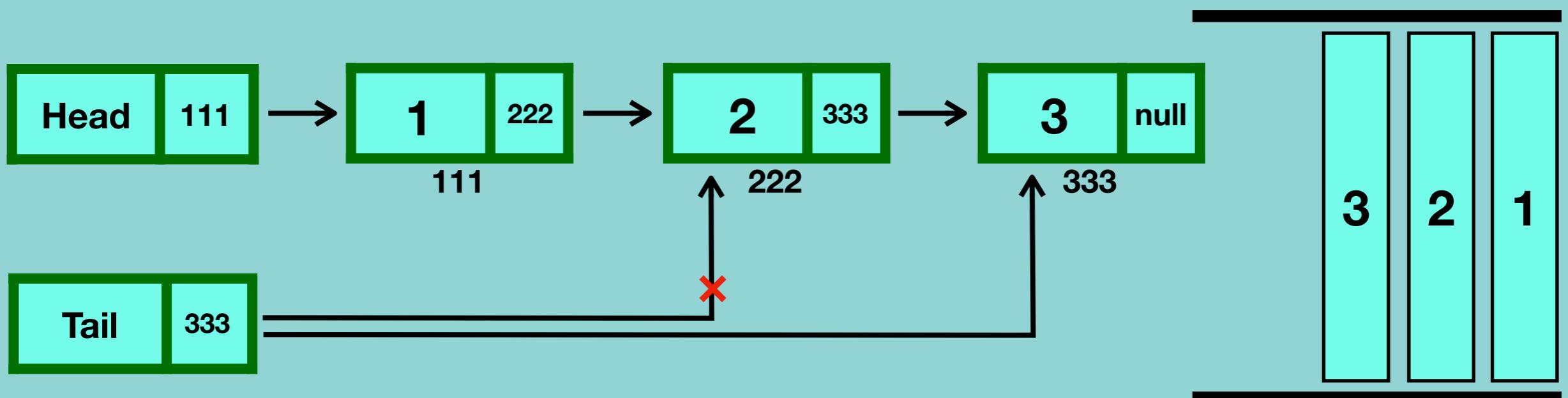
# Queue using Linked List

## enQueue() Method



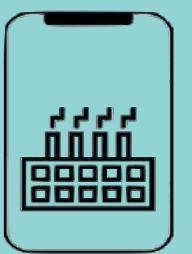
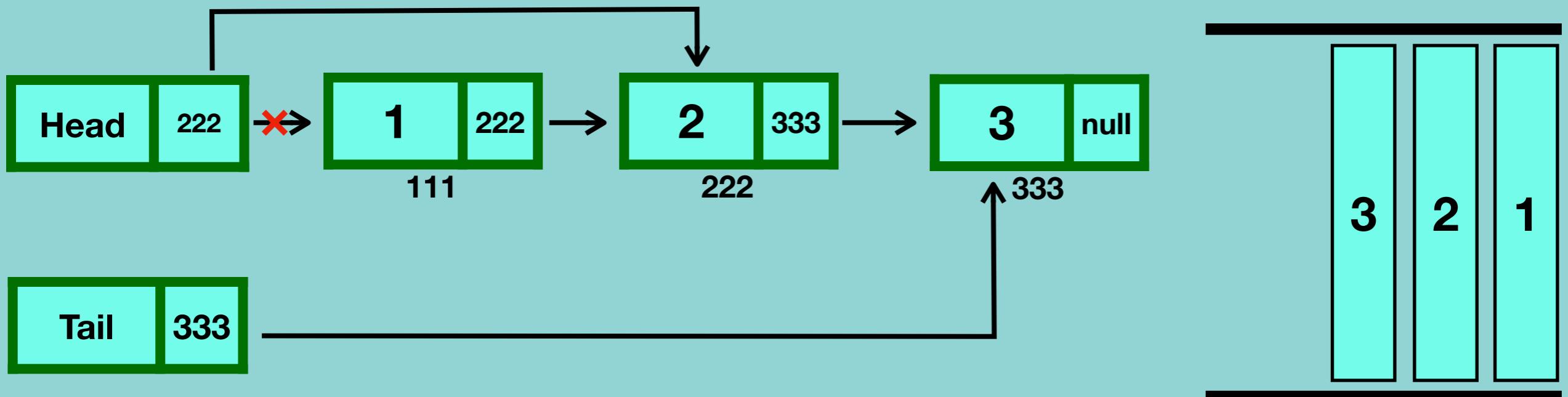
# Queue using Linked List

## enQueue() Method



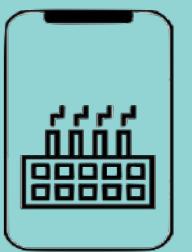
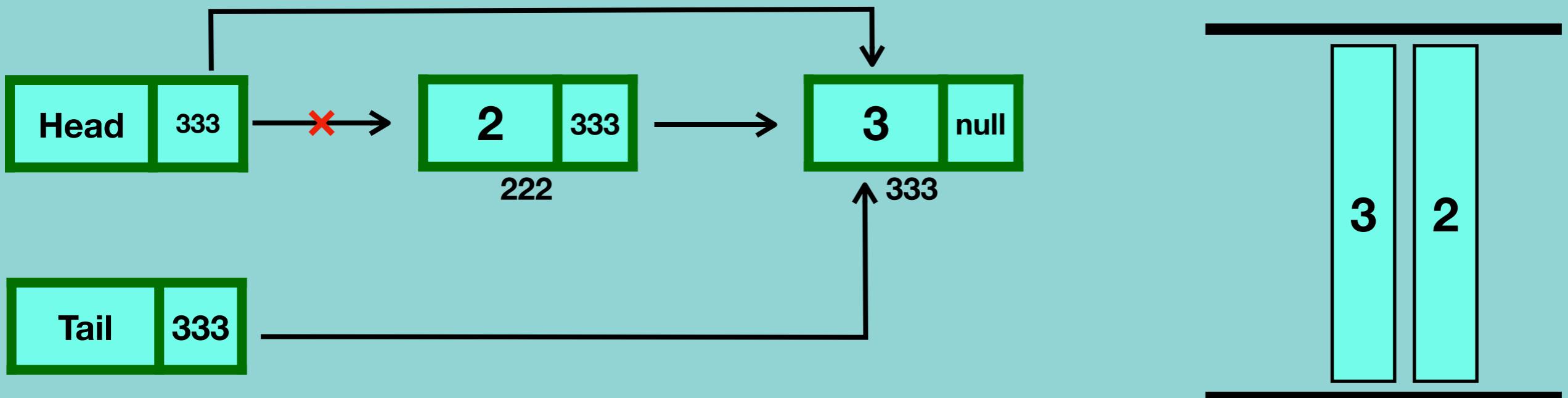
# Queue using Linked List

## deQueue() Method



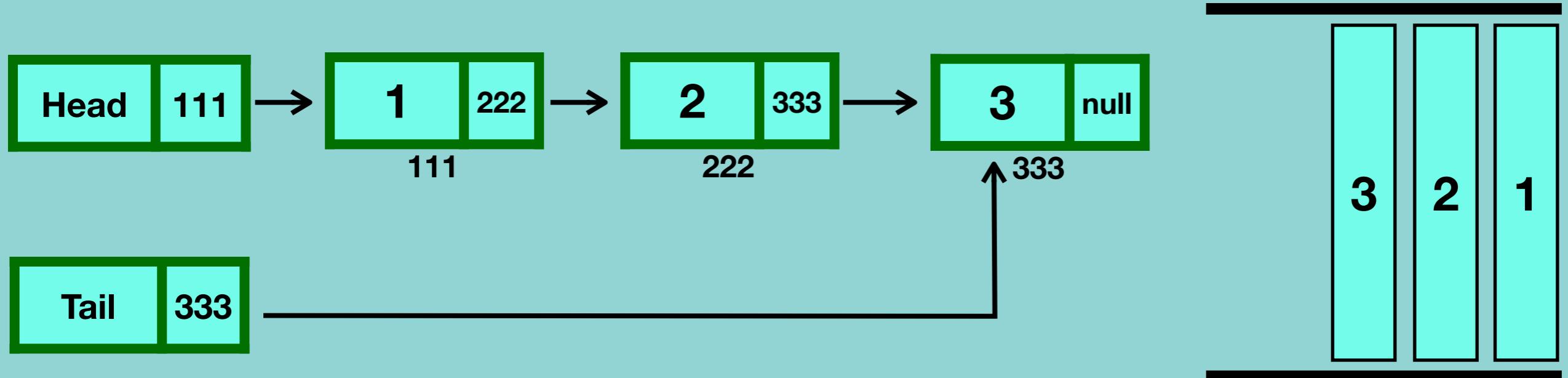
# Queue using Linked List

## deQueue() Method

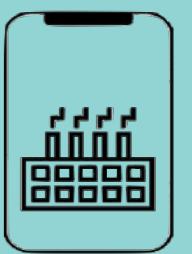


# Queue using Linked List

## peek() Method

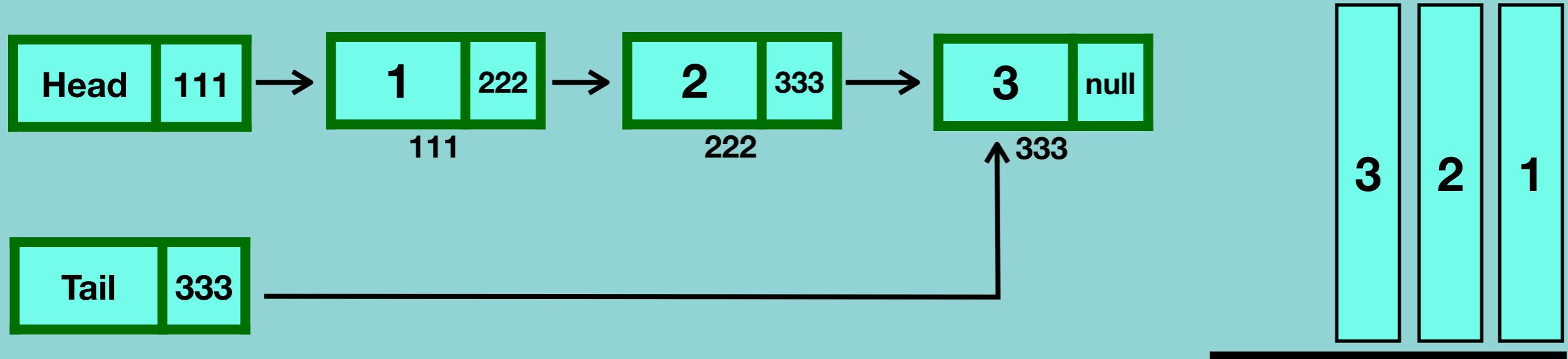


**peek()**  
return head.value

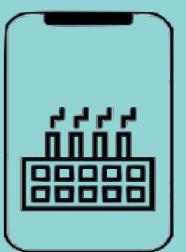


# Queue using Linked List

## isEmpty() Method

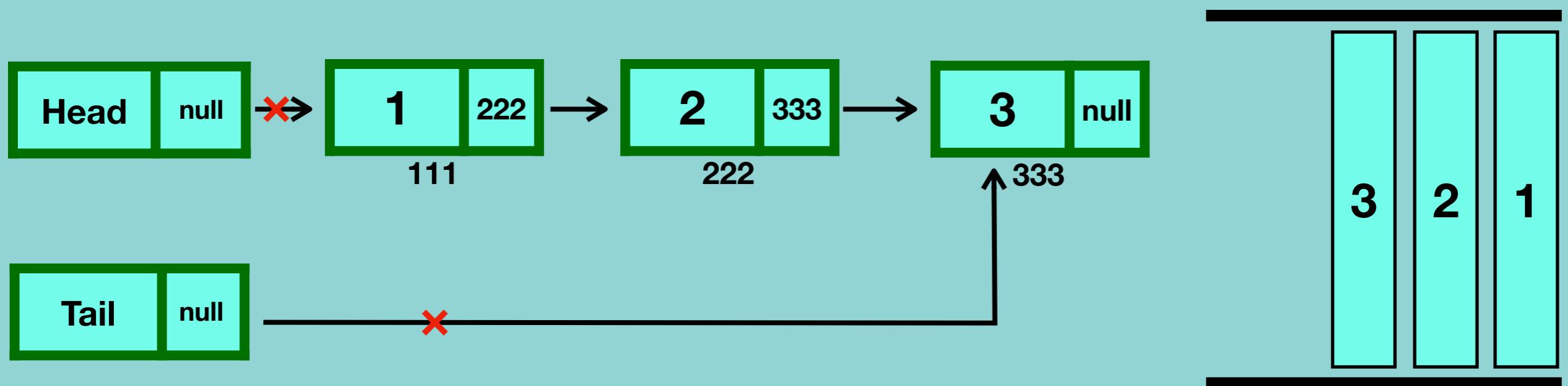


**isEmpty()**  
If head is None:  
True

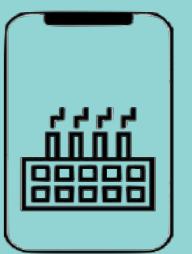


# Queue using Linked List

## delete() Method

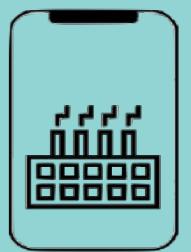


**delete()**  
head = None  
tail = None



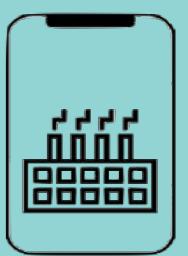
# Time and Space complexity of Queue operations using Linked List

	Time complexity	Space complexity
Create Queue	O(1)	O(1)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
Delete Entire Queue	O(1)	O(1)



# Time and Space complexity Queue : List vs Linked List

	List no capacity limit		List with capacity (Circular Queue)		Linked List	
	Time complexity	Space complexity	Time complexity	Space complexity	Time complexity	Space complexity
Create Queue	O(1)	O(1)	O(1)	O(n)	O(1)	O(1)
Enqueue	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Dequeue	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
isFull	-	-	O(1)	O(1)	-	-
Delete Entire Queue	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

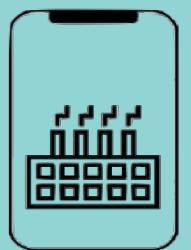
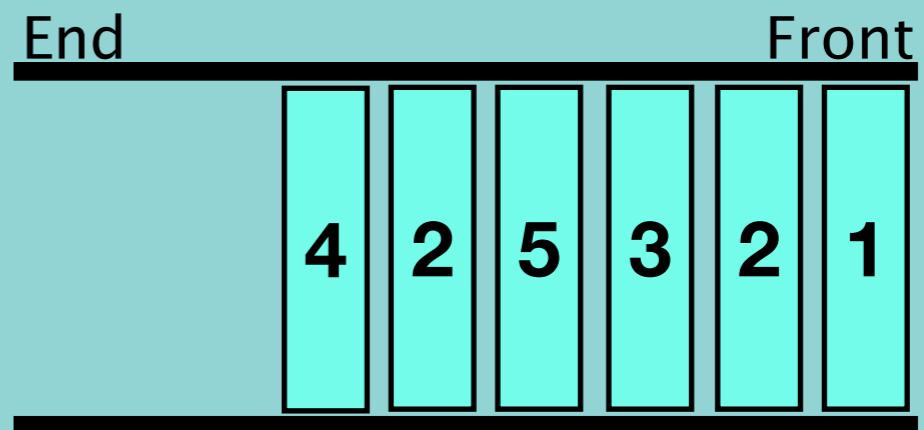


# Python Queue Modules

Collections module

Queue Module

Multiprocessing Module

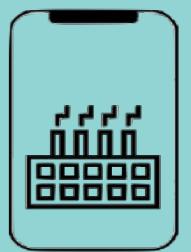
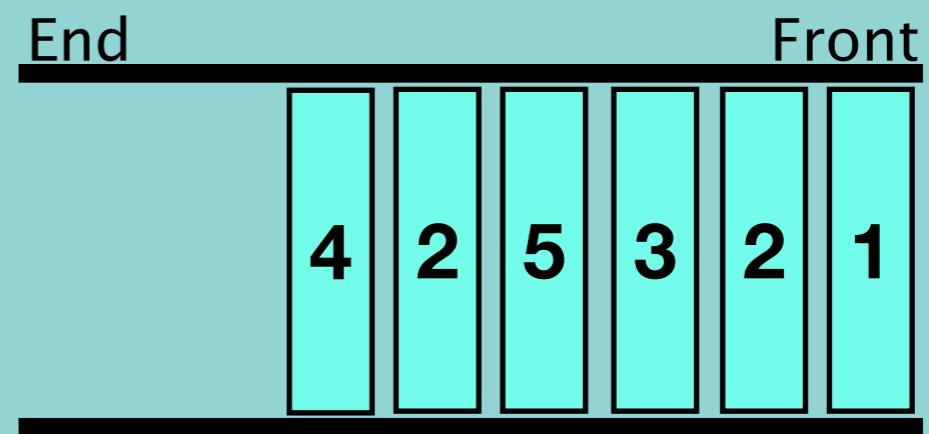


# Python Collections Module

## The `collections.deque` Class

### Methods

- `deque()`
- `append()`
- `popleft()`
- `clear()`

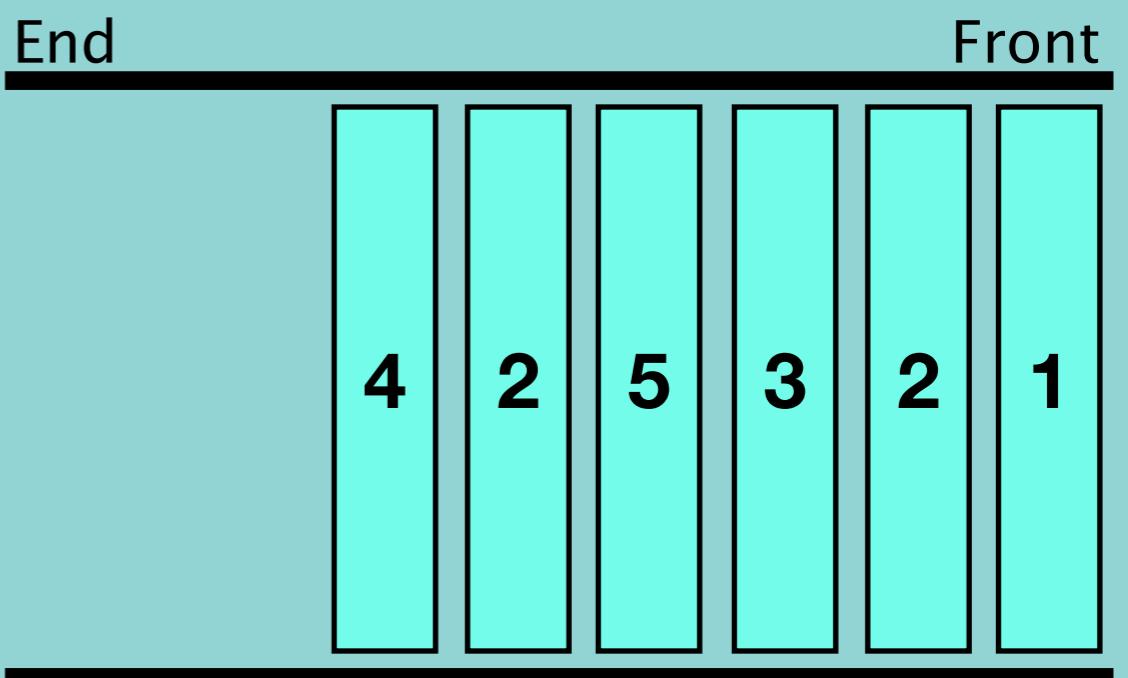


# Python Queue Module

FIFO queue ✓ → Queue(maxsize=0)

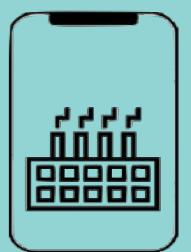
LIFO queue – Stack

Priority queue



## Methods

- qsize()
- empty()
- full()
- put()
- get()
- task\_done()
- join()

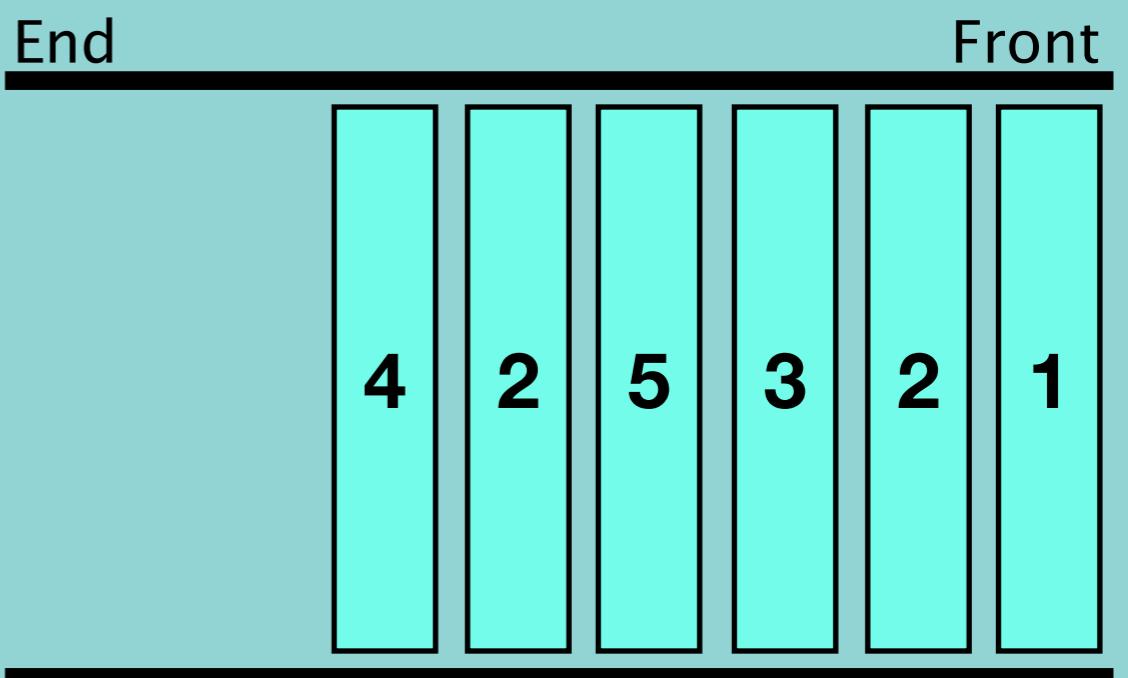


# Python Queue Module

FIFO queue ✓ → Queue(maxsize=0)

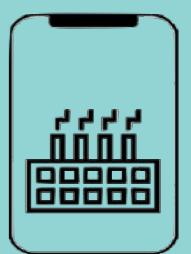
LIFO queue – Stack

Priority queue



## Methods

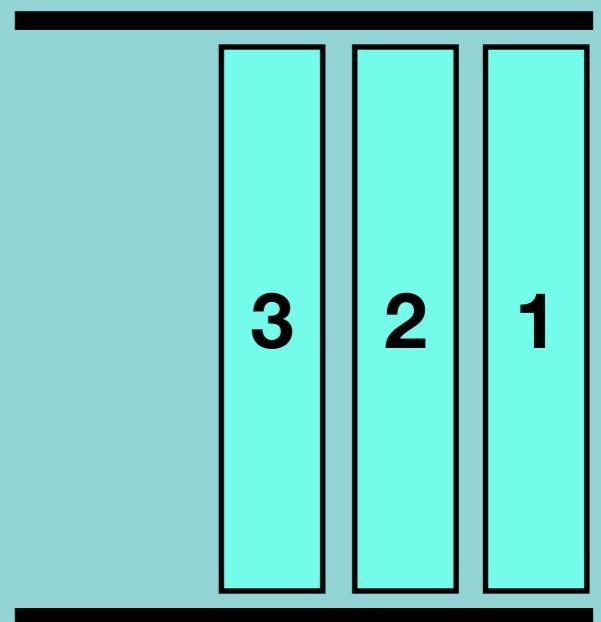
- qsize()
- empty()
- full()
- put()
- get()
- task\_done()
- join()



# When to use / avoid Queue

## Use:

- FIFO functionality
- The chance of data corruption is minimum

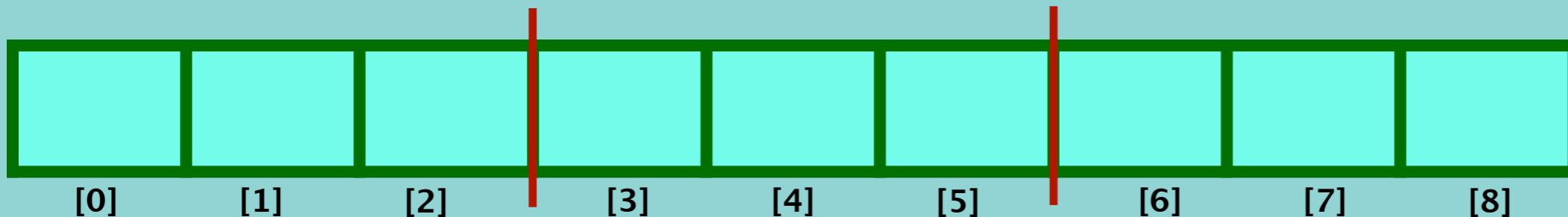


## Avoid:

- Random access is not possible

# Interview Questions - 1 : Three in One

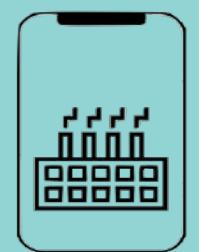
Describe how you could use a single Python list to implement three stacks.



For Stack 1 — [0], [1], [2] → [0,  $n/3$ )

For Stack 2 — [3], [4], [6] → [ $n/3$ ,  $2n/3$ )

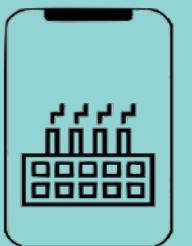
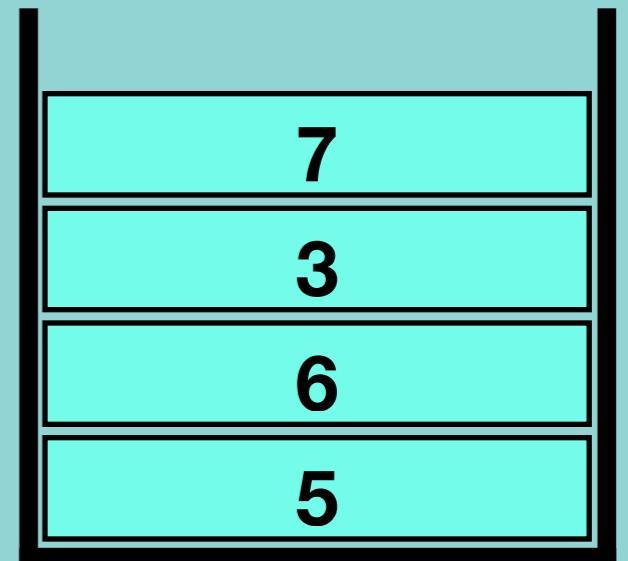
For Stack 3 — [7], [8], [9] → [ $2n/3$ ,  $n$ )



# Interview Questions - 2 : Stack Min

How would you design a stack which, in addition to push and pop, has a function min which returns the minimum element? Push, pop and min should all operate in O(1).

push(5)	min() → 5
push(6)	min() → 5
push(3)	min() → 3
push(7)	min() → 3
pop()	min() → 3
pop()	min() → 5



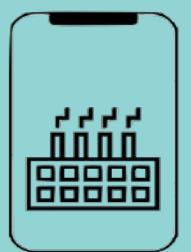
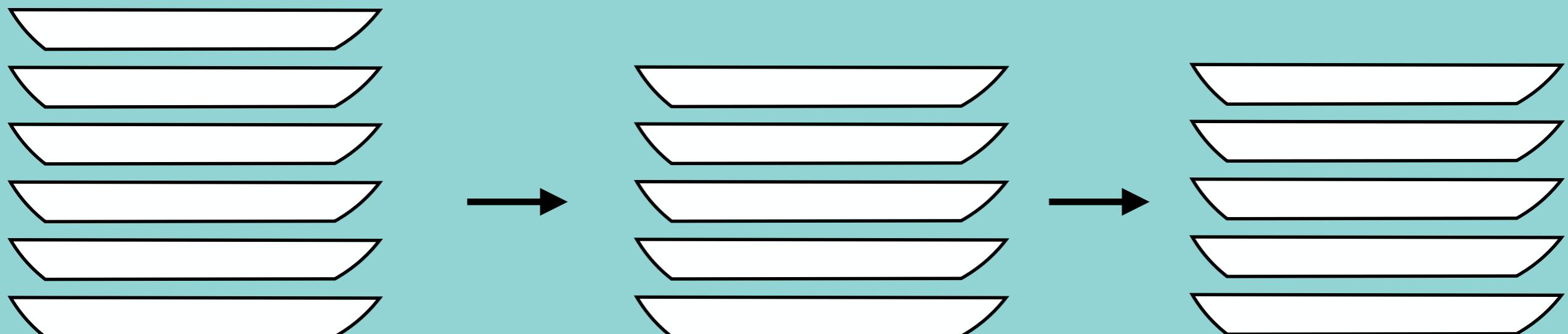
# Interview Questions - 3 : Stack of Plates

Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold.

Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity, `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

Follow Up:

Implement a function `popAt (int index)` which performs a pop operation on a specific sub - stack.



# Interview Questions - 3 : Stack of Plates

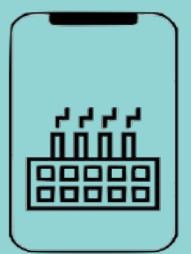
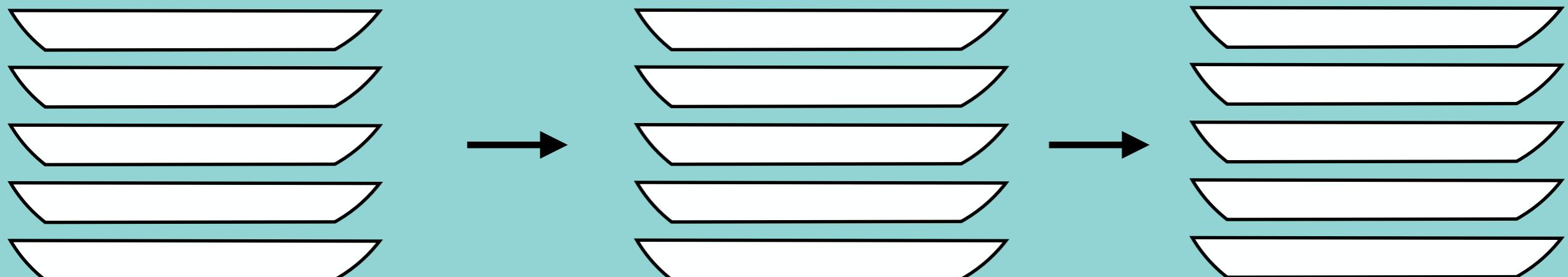
Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold.

Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity, `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

**Follow Up:**

Implement a function `popAt (int index)` which performs a pop operation on a specific sub - stack.

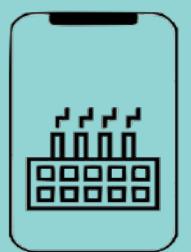
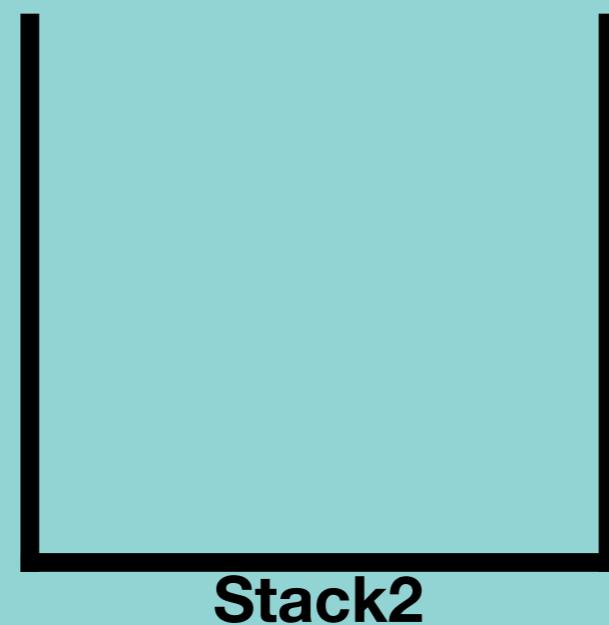
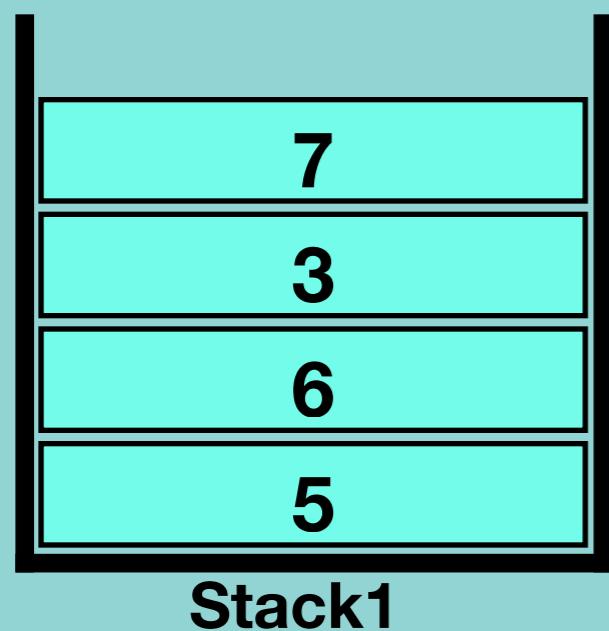
`pop( )`



# Interview Questions - 4 : Queue via Stacks

Implement Queue class which implements a queue using two stacks.

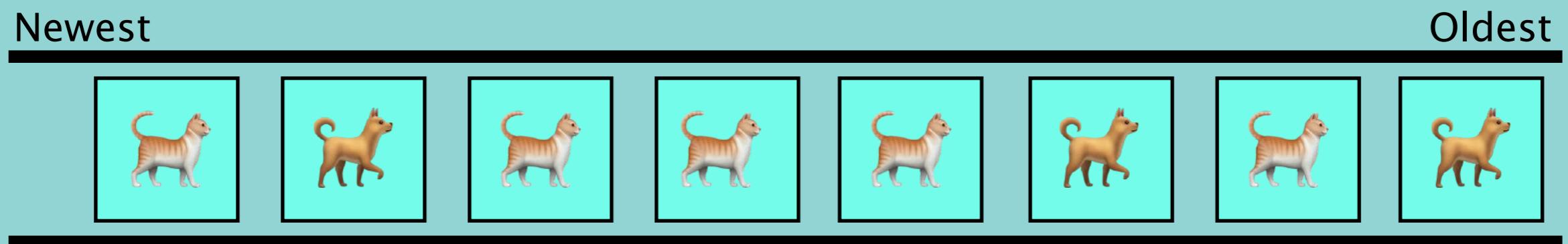
**Dequeue()**



# Interview Questions - 5 : Animal Shelter

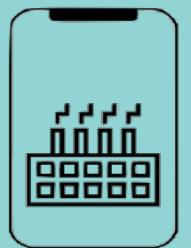
An animal shelter, which holds only dogs and cats, operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog, and dequeueCat.

First In First Out



Enqueue(Dog)

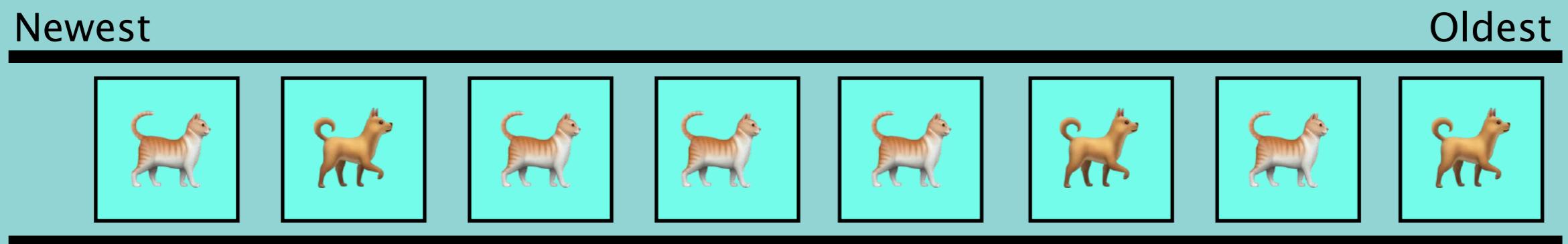
Enqueue(Cat)



# Interview Questions - 5 : Animal Shelter

An animal shelter, which holds only dogs and cats, operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog, and dequeueCat.

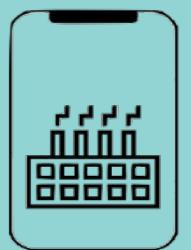
First In First Out



DequeueAny()

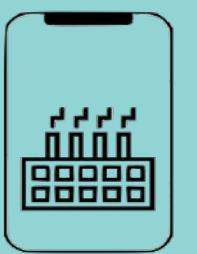
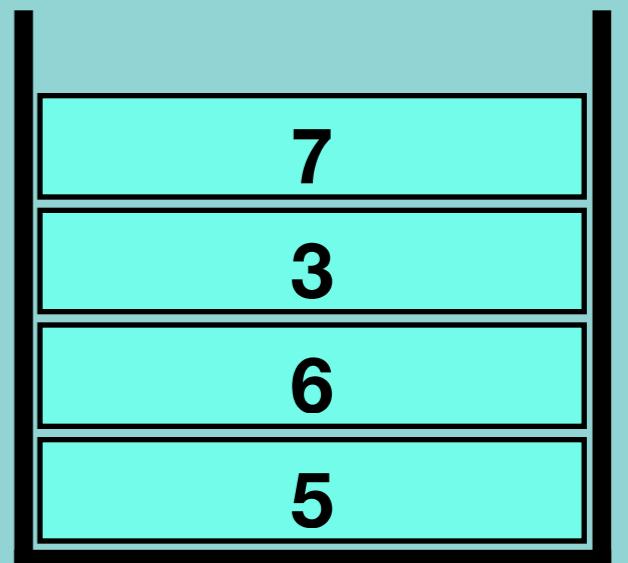
DequeueDog()

DequeueCat()



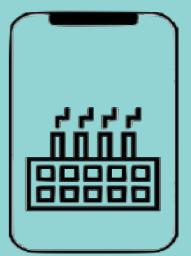
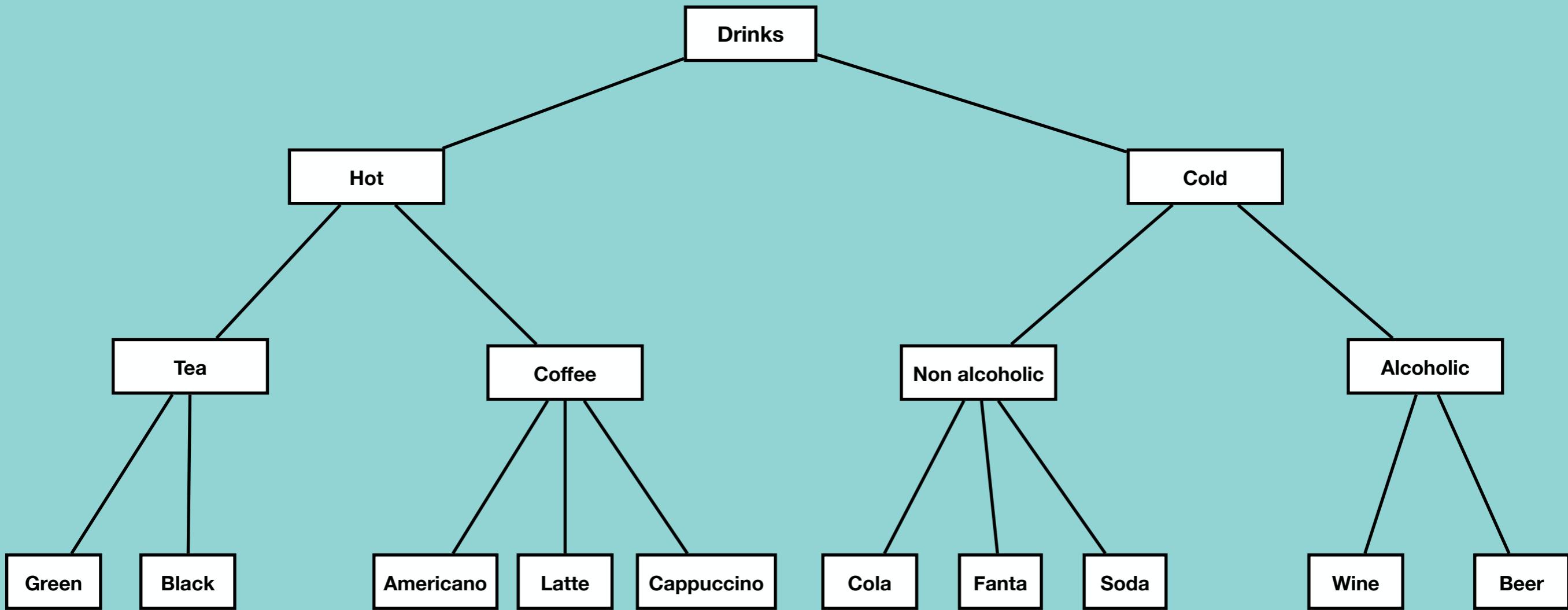
# Interview Questions - 6 : Animal Shelter

Implement MyQueue class which implements a queue using two stacks.



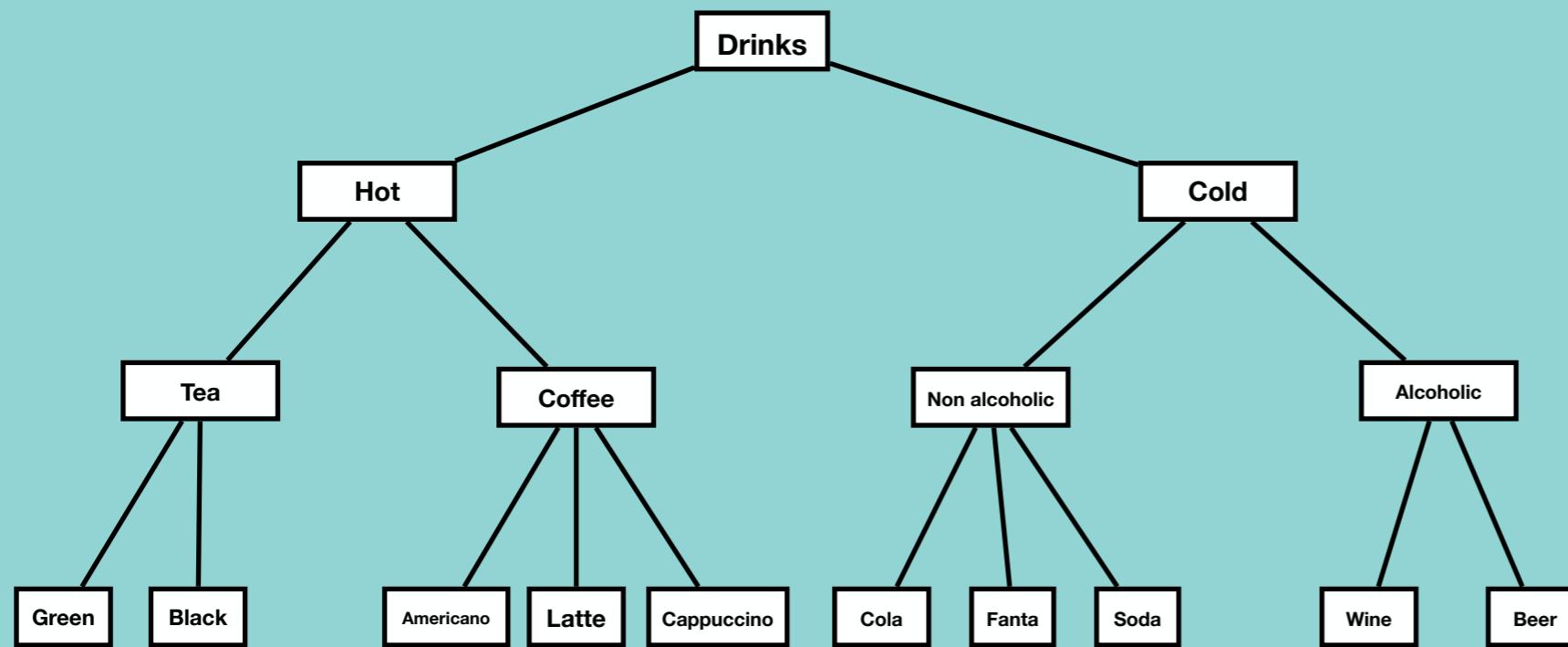
# What is a Tree?

A tree is a nonlinear data structure with hierarchical relationships between its elements without having any cycle, it is basically reversed from a real life tree.



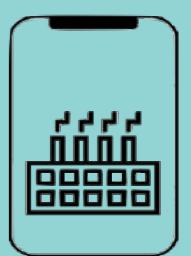
# What is a Tree?

A tree is a nonlinear data structure with hierarchical relationships between its elements without having any cycle, it is basically reversed from a real life tree.



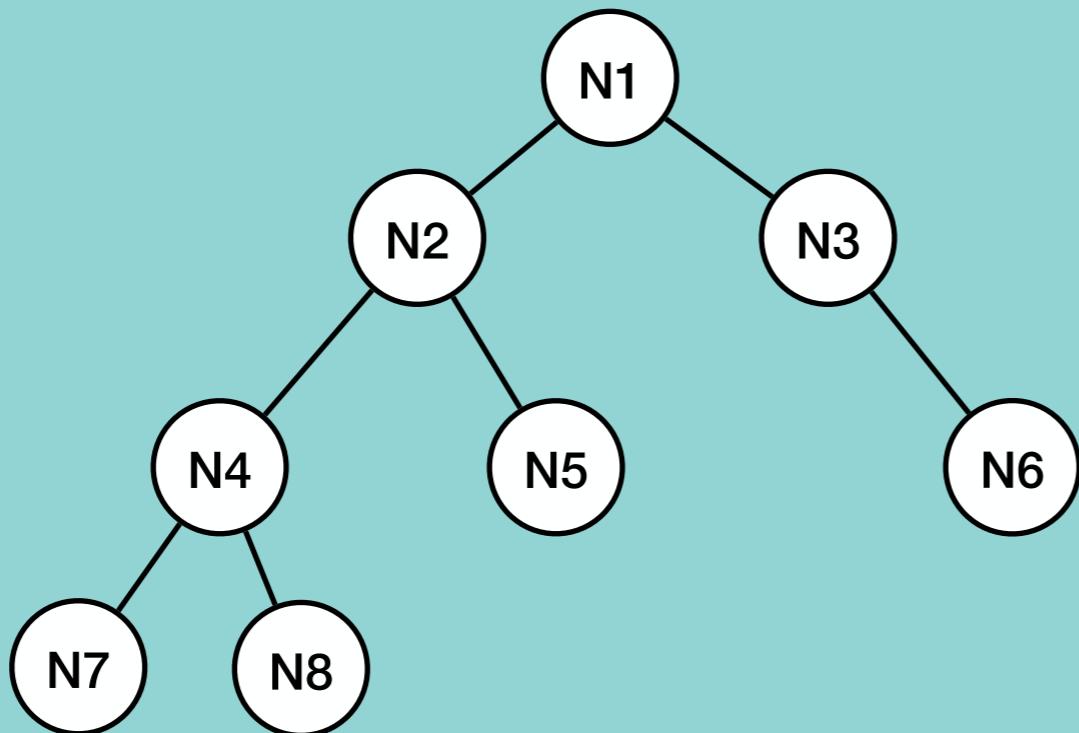
## Properties:

- Represent hierarchical data
- Each node has two components : data and a link to its sub category
- Base category and sub categories under it



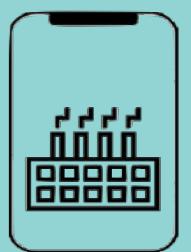
# What is a Tree?

A tree is a nonlinear data structure with hierarchical relationships between its elements without having any cycle, it is basically reversed from a real life tree.



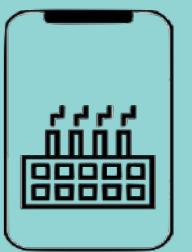
## Tree Properties:

- Represent hierarchical data
- Each node has two components : data and a link to its sub category
- Base category and sub categories under it



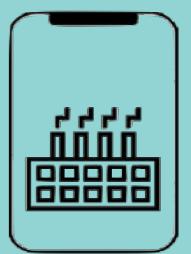
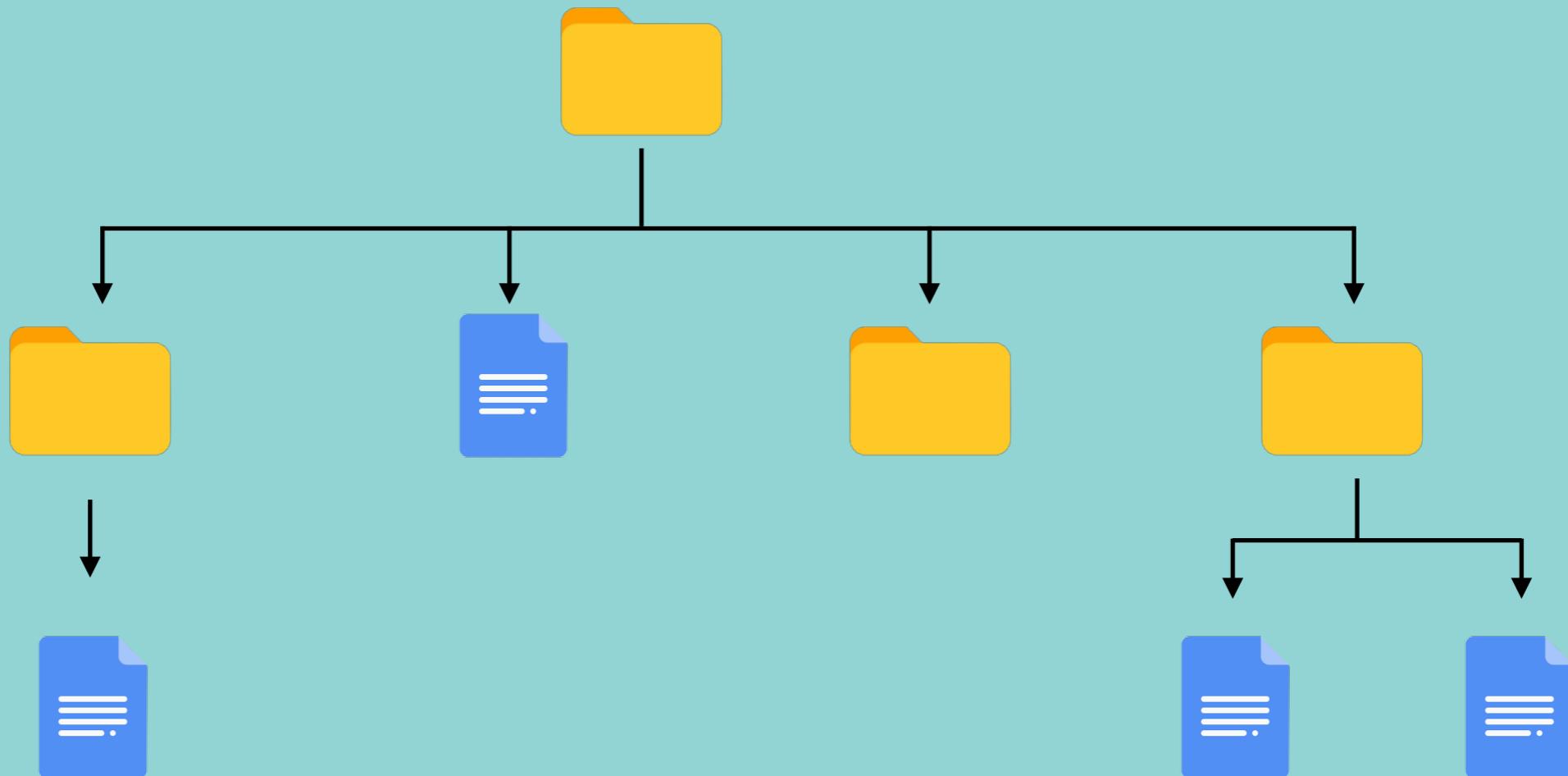
# Why a Tree?

- Quicker and Easier access to the data
- Store hierarchical data, like folder structure, organization structure, XML/HTML data.



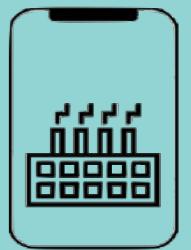
# Why a Tree?

The file system on a computer



# Why a Tree?

- Quicker and Easier access to the data
- Store hierarchical data, like folder structure, organization structure, XML/HTML data.
- There are many different types of data structures which performs better in various situations
  - Binary Search Tree, AVL, Red Black Tree, Trie



# Tree Terminology

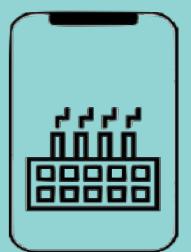
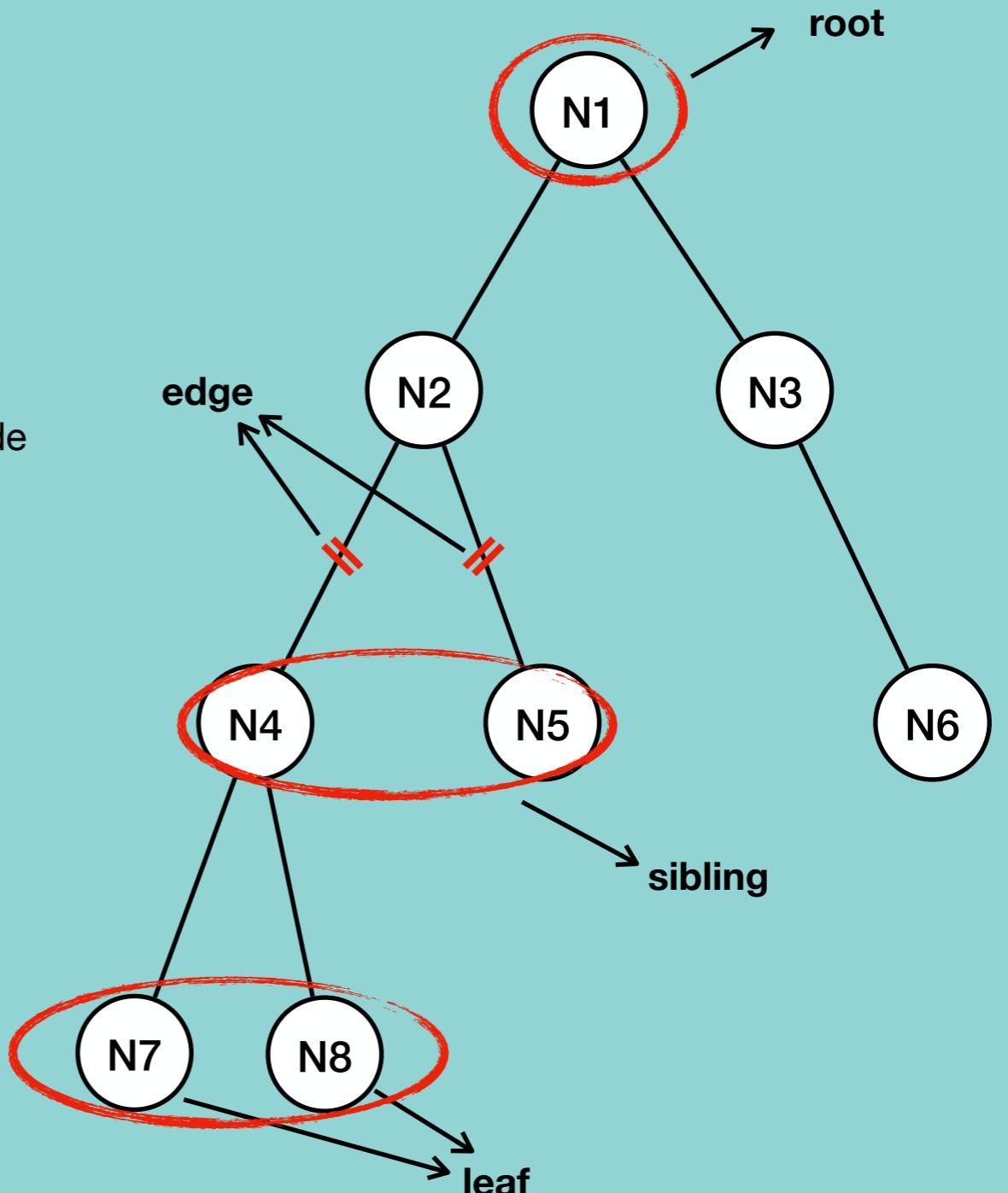
**Root** : top node without parent

**Edge** : a link between parent and child

**Leaf** : a node which does not have children

**Sibling** : children of same parent

**Ancestor** : parent, grandparent, great grandparent of a node



# Tree Terminology

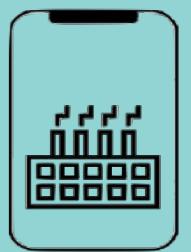
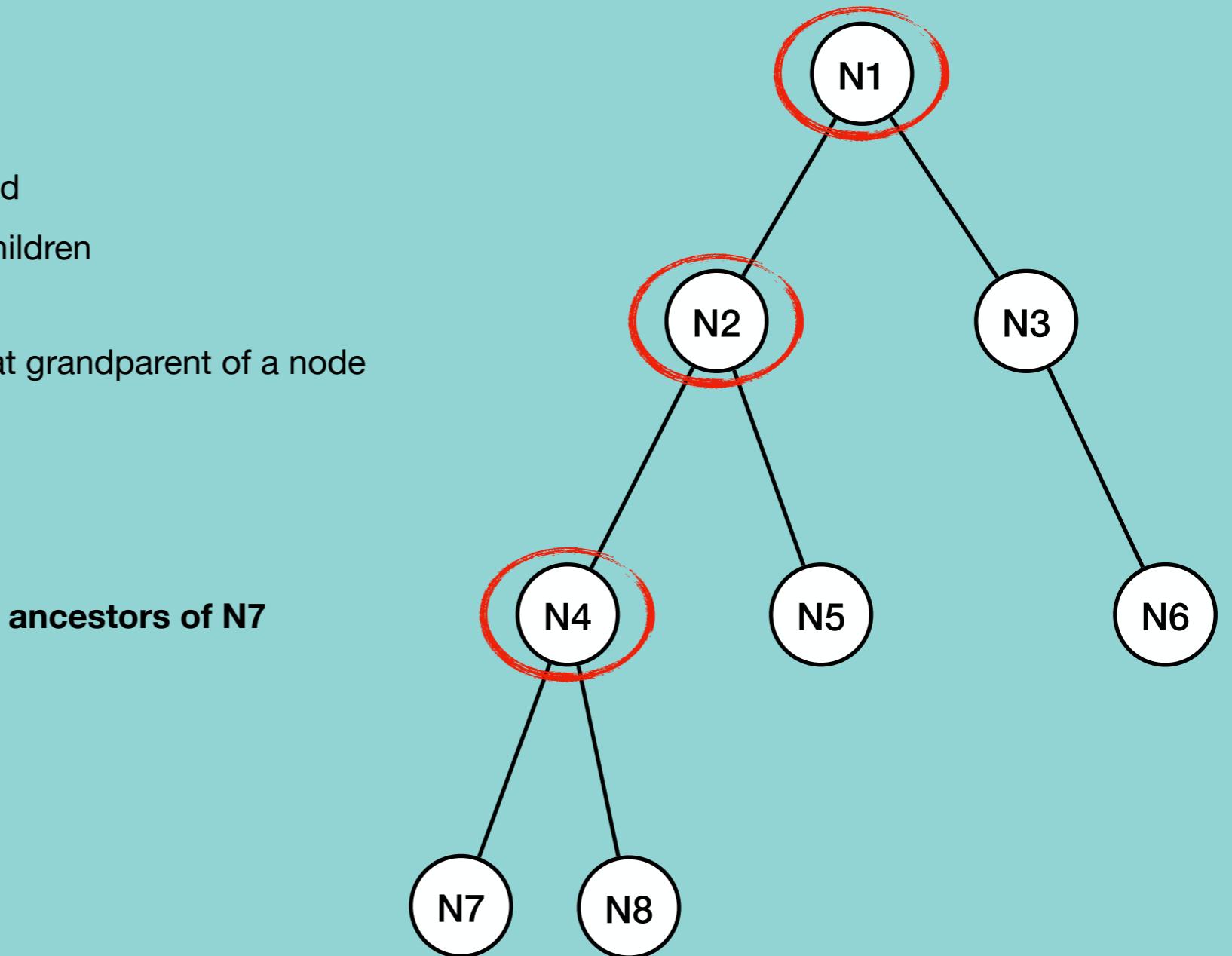
**Root** : top node without parent

**Edge** : a link between parent and child

**Leaf** : a node which does not have children

**Sibling** : children of same parent

**Ancestor** : parent, grandparent, great grandparent of a node



# Tree Terminology

**Root** : top node without parent

**Edge** : a link between parent and child

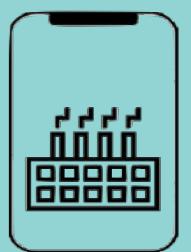
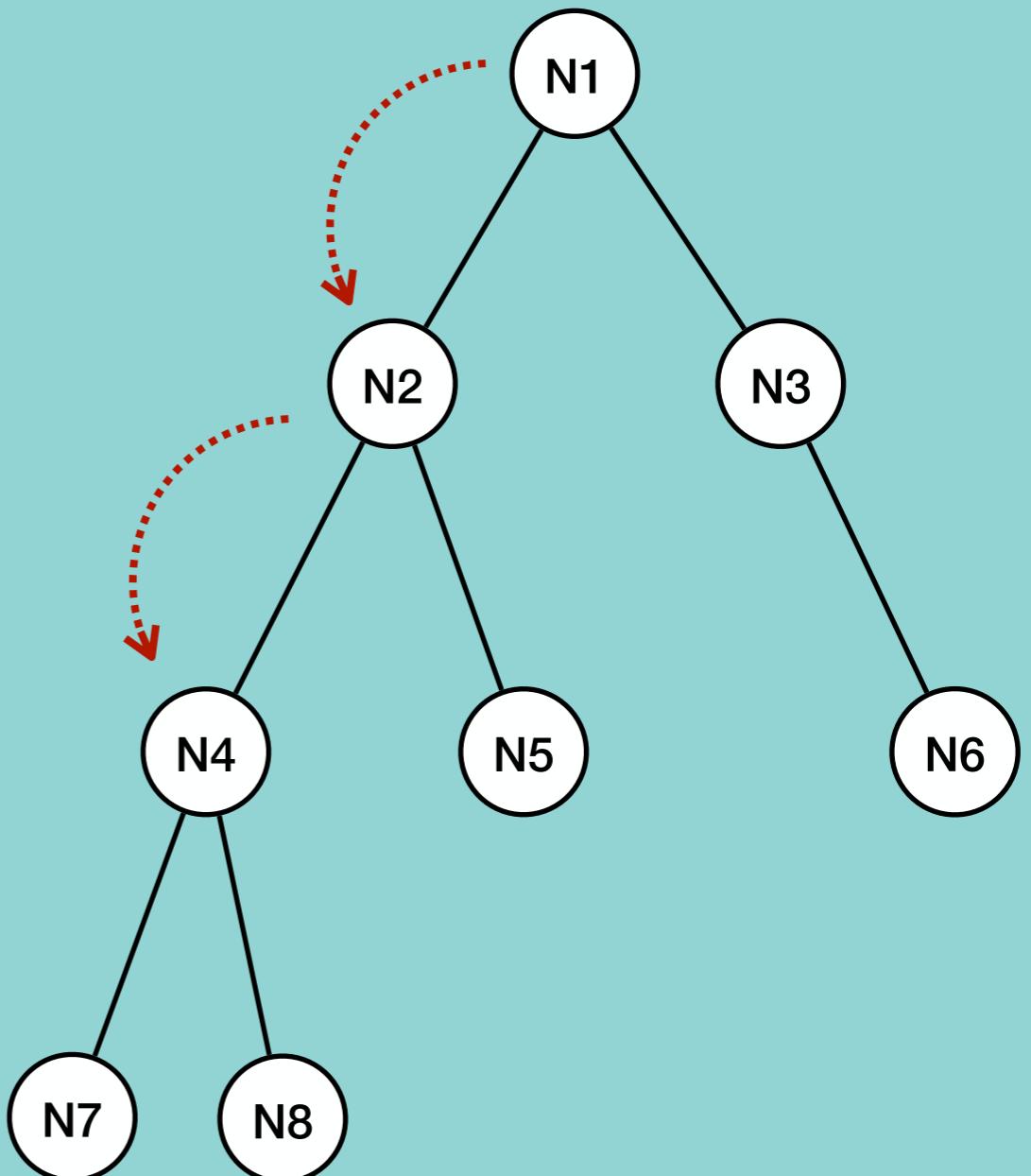
**Leaf** : a node which does not have children

**Sibling** : children of same parent

**Ancestor** : parent, grandparent, great grandparent of a node

**Depth of node** : a length of the path from root to node

**Depth of N4 = 2**



# Tree Terminology

**Root** : top node without parent

**Edge** : a link between parent and child

**Leaf** : a node which does not have children

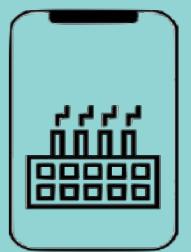
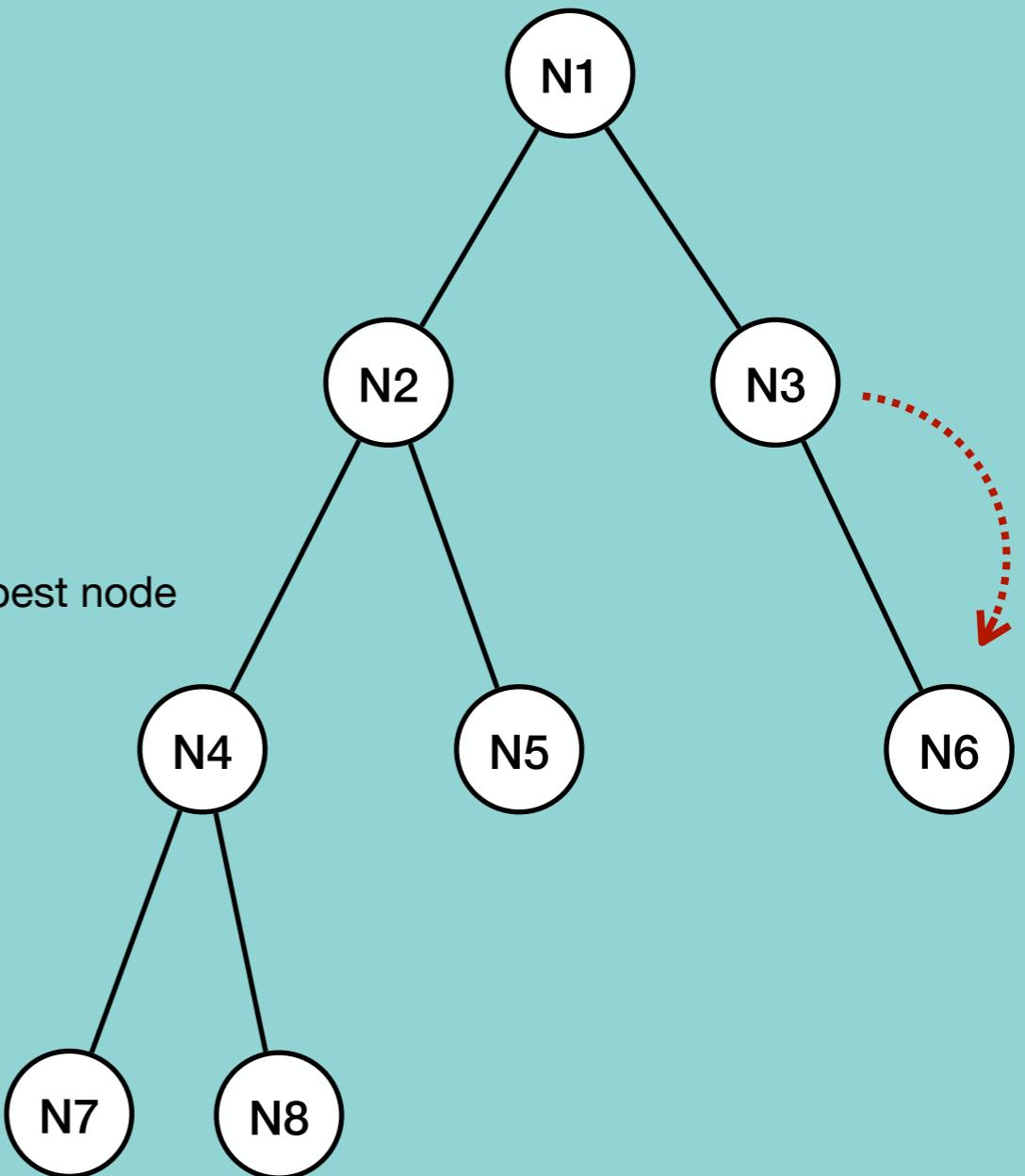
**Sibling** : children of same parent

**Ancestor** : parent, grandparent, great grandparent of a node

**Depth of node** : a length of the path from root to node

**Height of node** : a length of the path from the node to the deepest node

**Heigh of N3 = 1**



# Tree Terminology

**Root** : top node without parent

**Edge** : a link between parent and child

**Leaf** : a node which does not have children

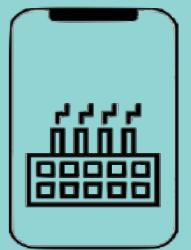
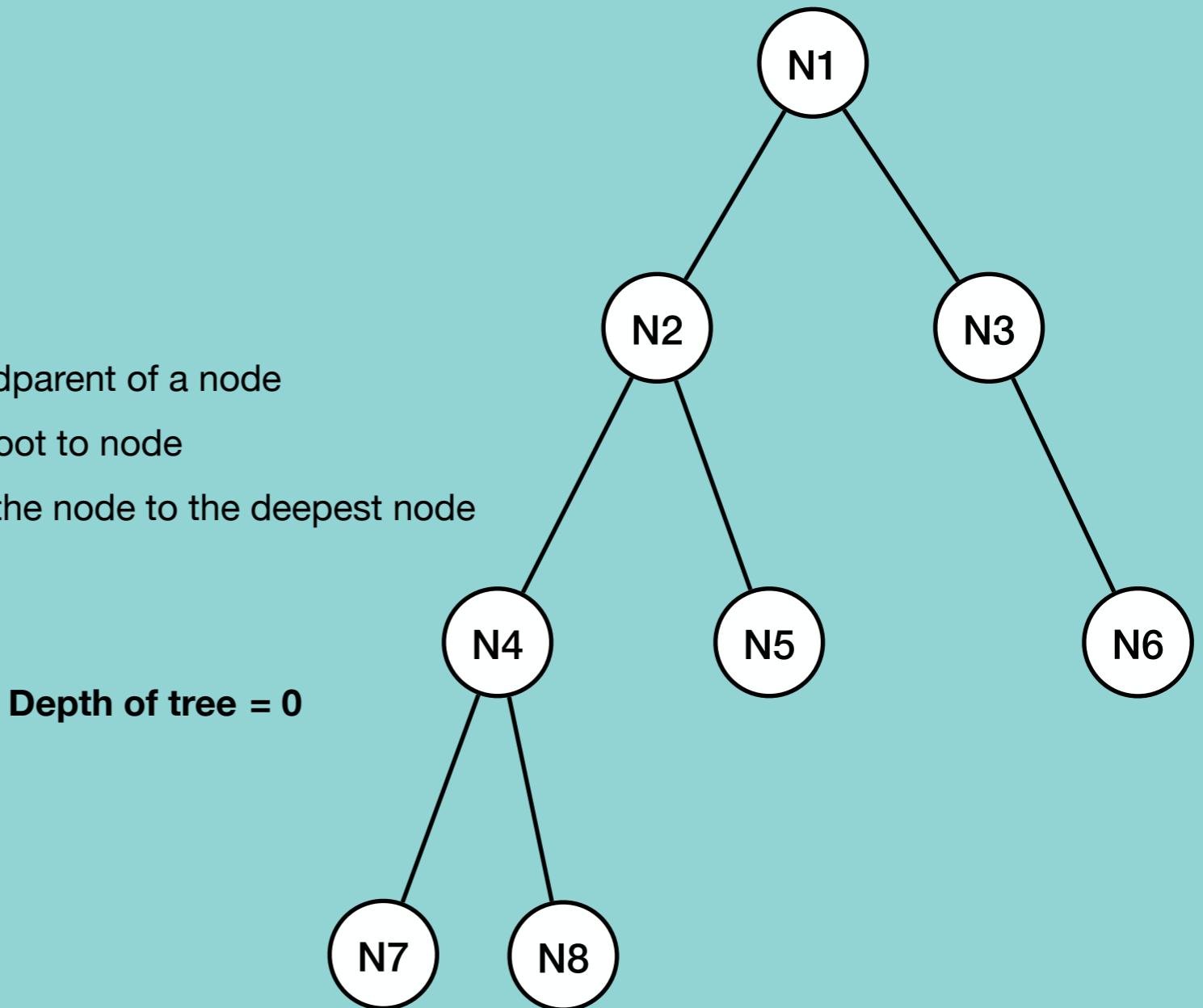
**Sibling** : children of same parent

**Ancestor** : parent, grandparent, great grandparent of a node

**Depth of node** : a length of the path from root to node

**Height of node** : a length of the path from the node to the deepest node

**Depth of tree** : depth of root node



# Tree Terminology

**Root** : top node without parent

**Edge** : a link between parent and child

**Leaf** : a node which does not have children

**Sibling** : children of same parent

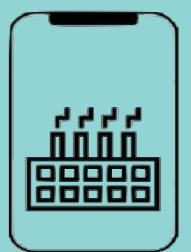
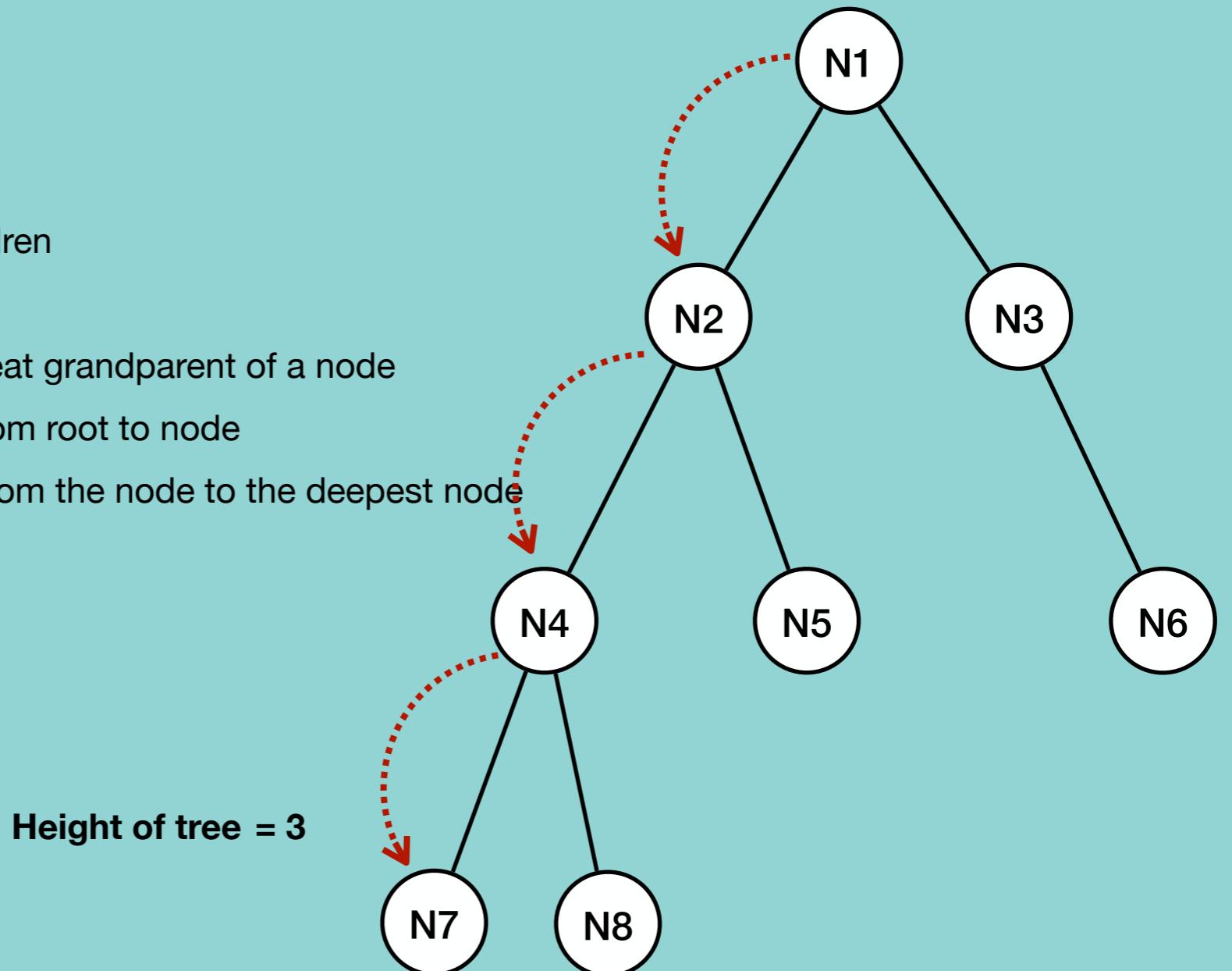
**Ancestor** : parent, grandparent and great grandparent of a node

**Depth of node** : a length of the path from root to node

**Height of node** : a length of the path from the node to the deepest node

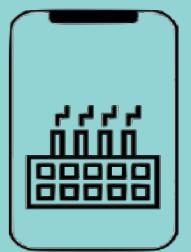
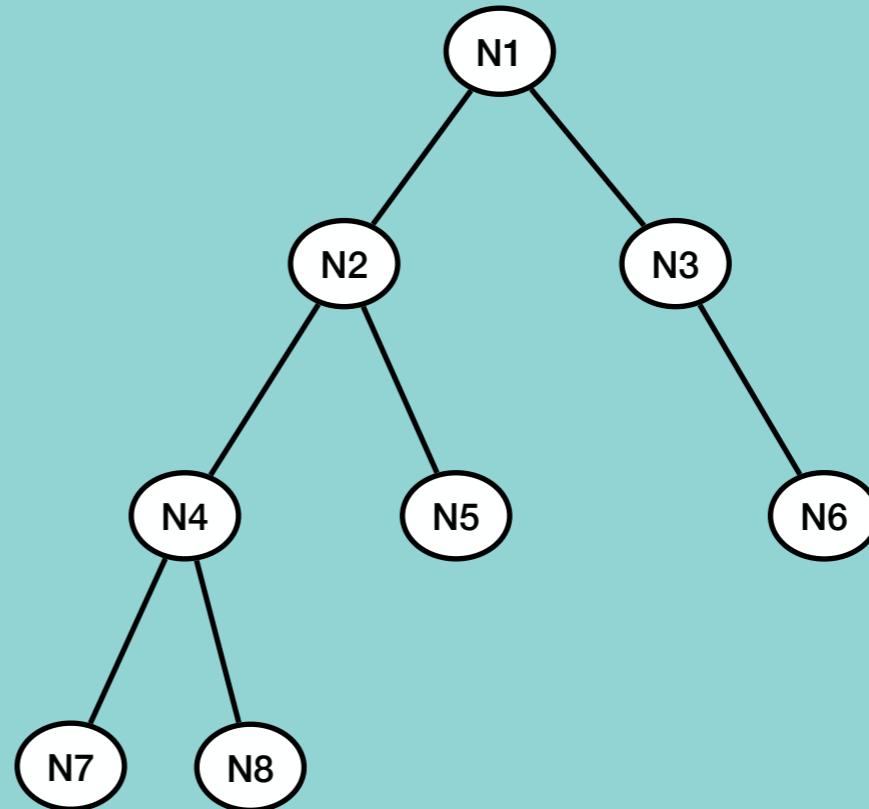
**Depth of tree** : depth of root node

**Height of tree** : height of root node



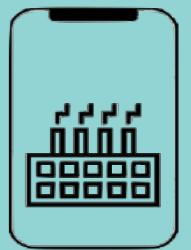
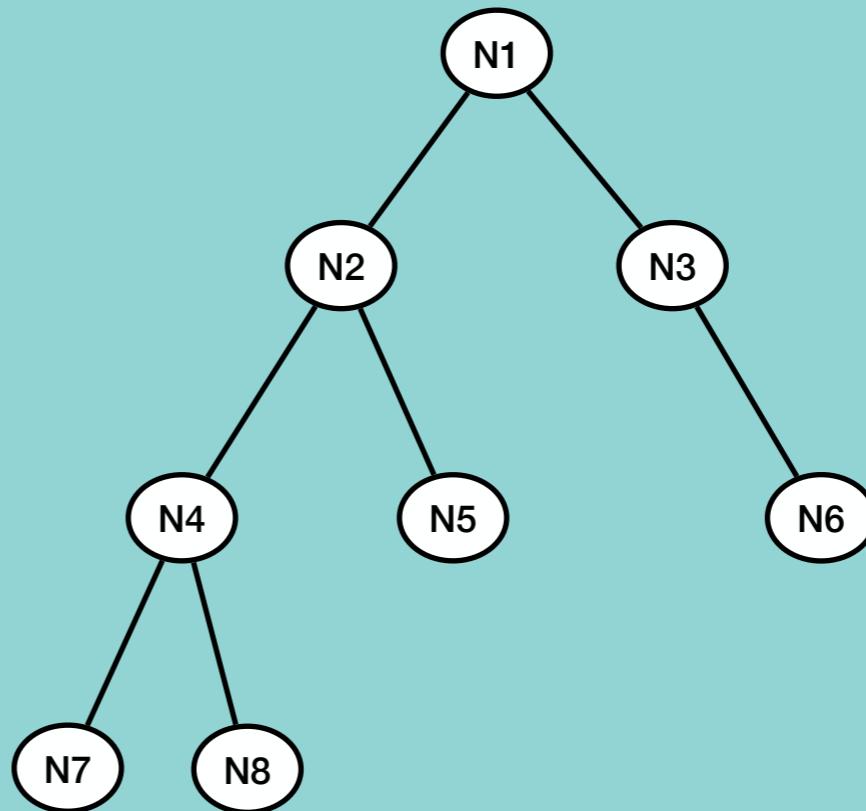
# Binary Tree

- Binary trees are the data structures in which each node has at most two children, often referred to as the left and right children
- Binary tree is a family of data structure (BST, Heap tree, AVL, red black trees, Syntax tree)



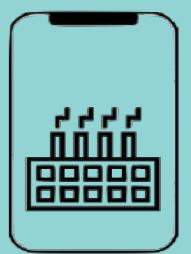
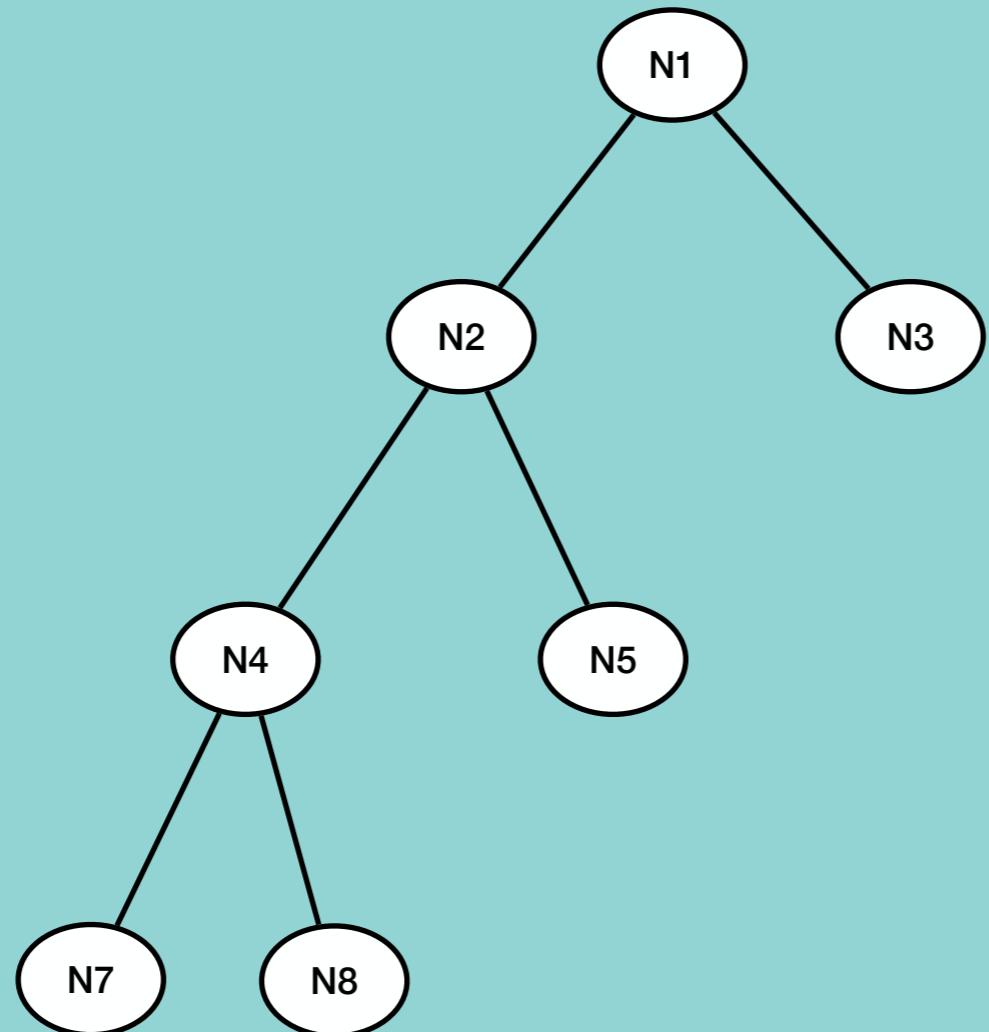
# Why Binary Tree?

- Binary trees are a prerequisite for more advanced trees like BST, AVL, Red Black Trees
- Huffman coding problem , heap priority problem and expression parsing problems can be solved efficiently using binary trees,



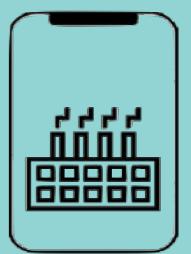
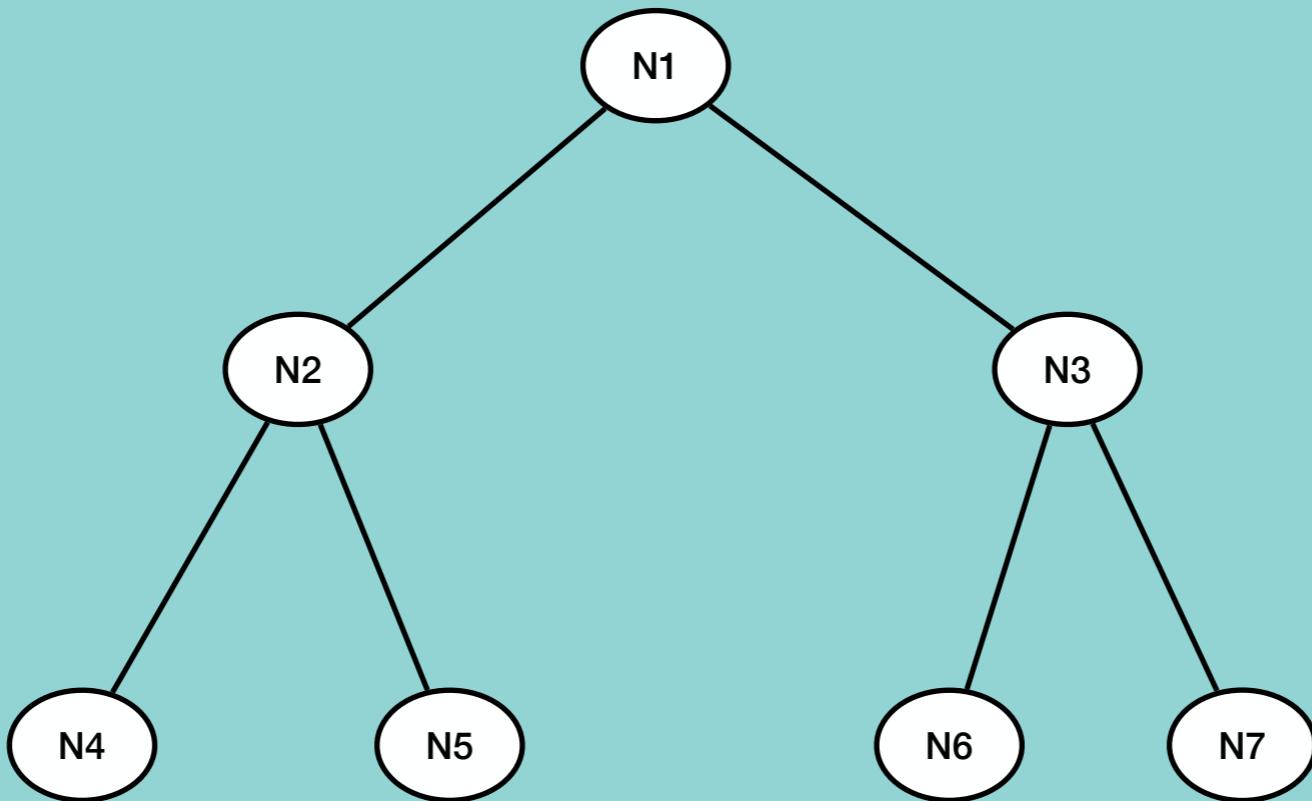
# Types of Binary Tree

## Full Binary Tree



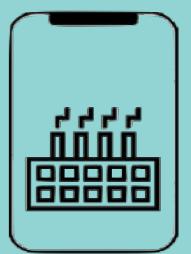
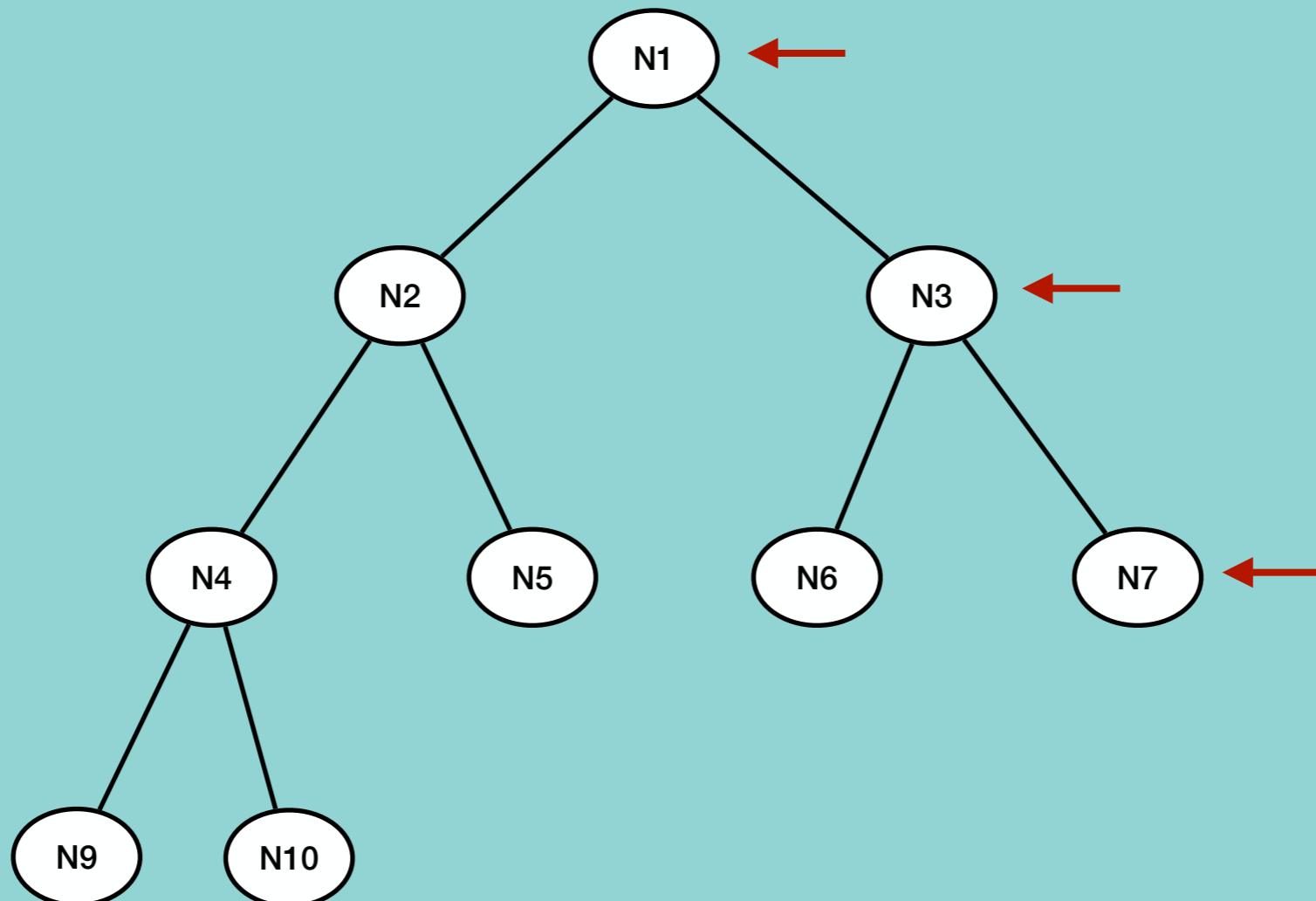
# Types of Binary Tree

## Perfect Binary Tree



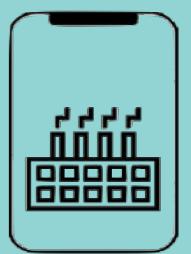
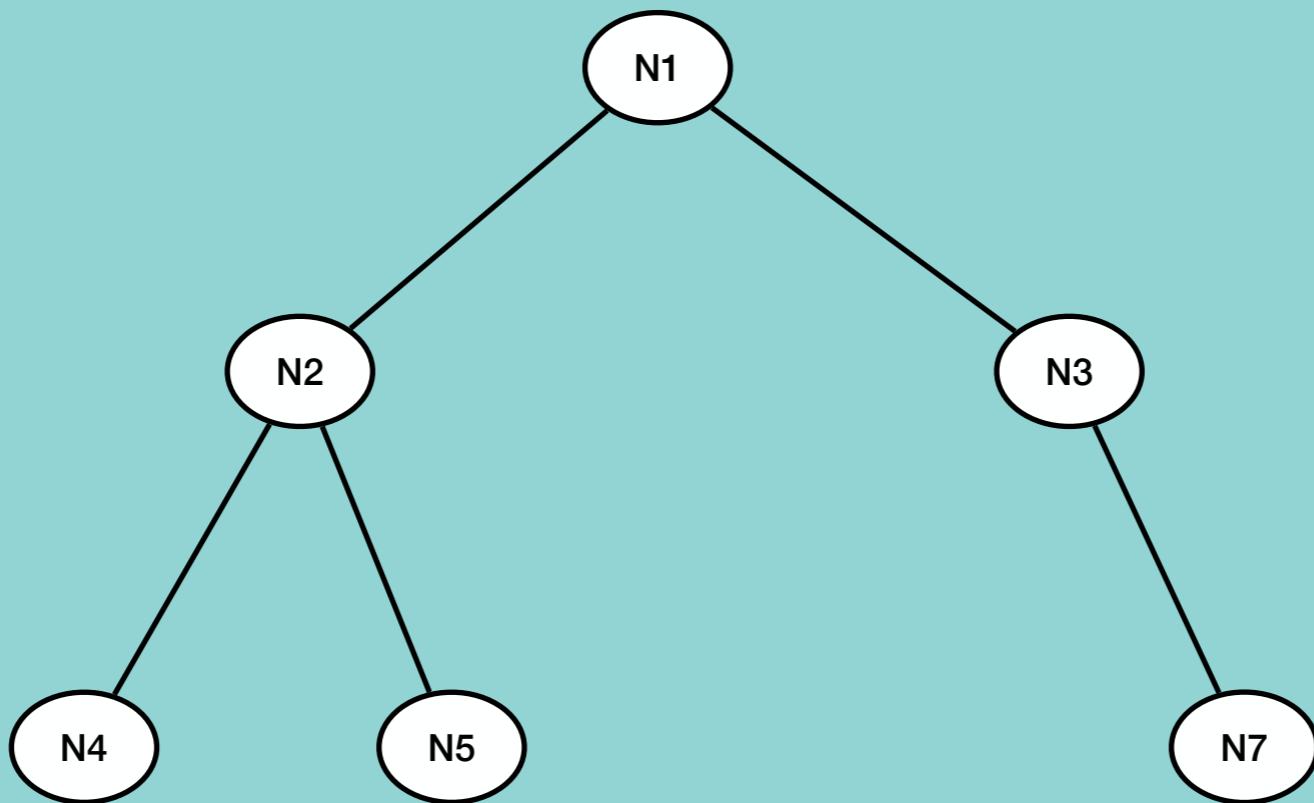
# Types of Binary Tree

## Complete Binary Tree



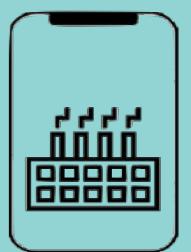
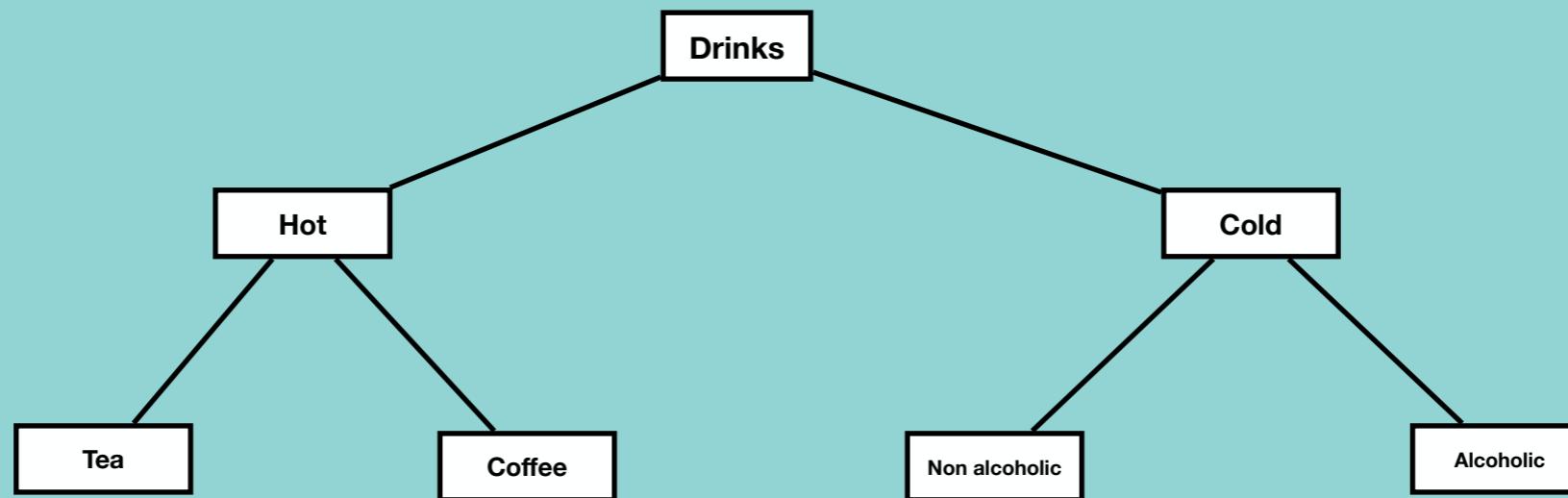
# Types of Binary Tree

## Balanced Binary Tree



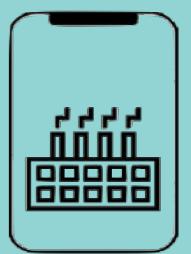
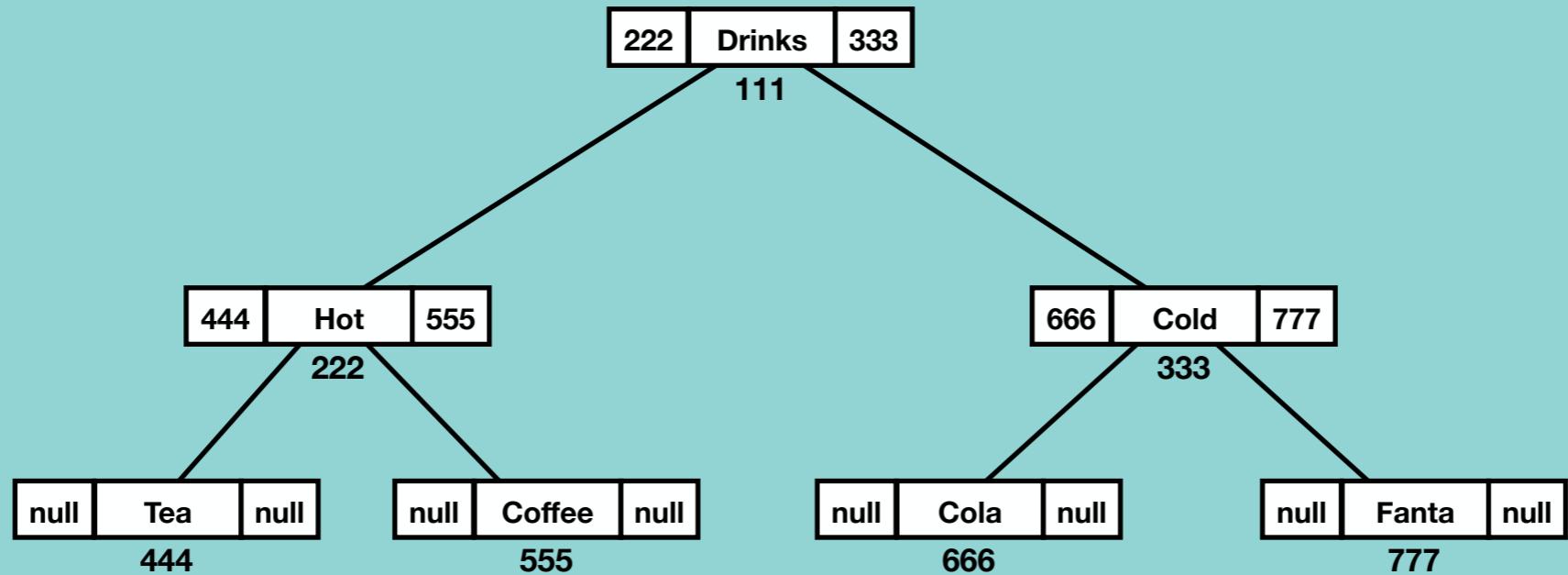
# Binary Tree

- Linked List
- Python List (array)



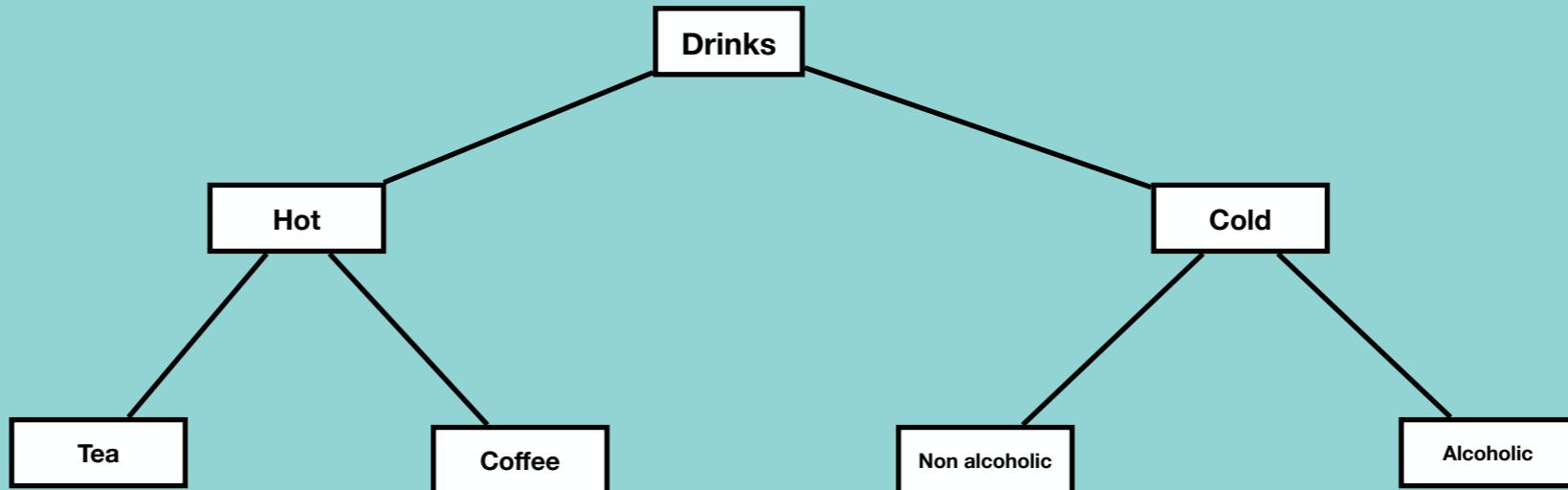
# Binary Tree

## Linked List



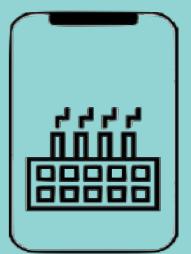
# Binary Tree

## Python List



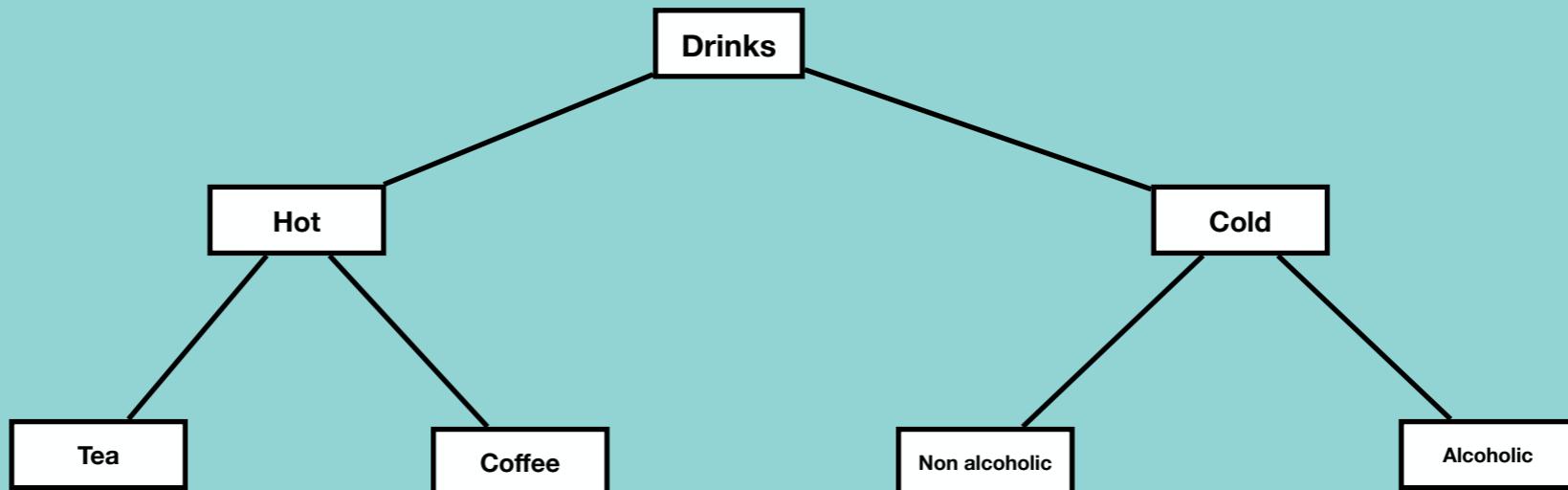
**Left child = cell[2x]**

**Right child = cell[2x+1]**



# Binary Tree

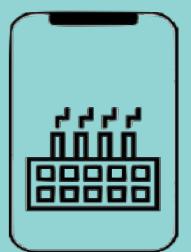
## Python List



0	1	2	3	4	5	6	7	8
✗	Drinks	Hot	Cold					

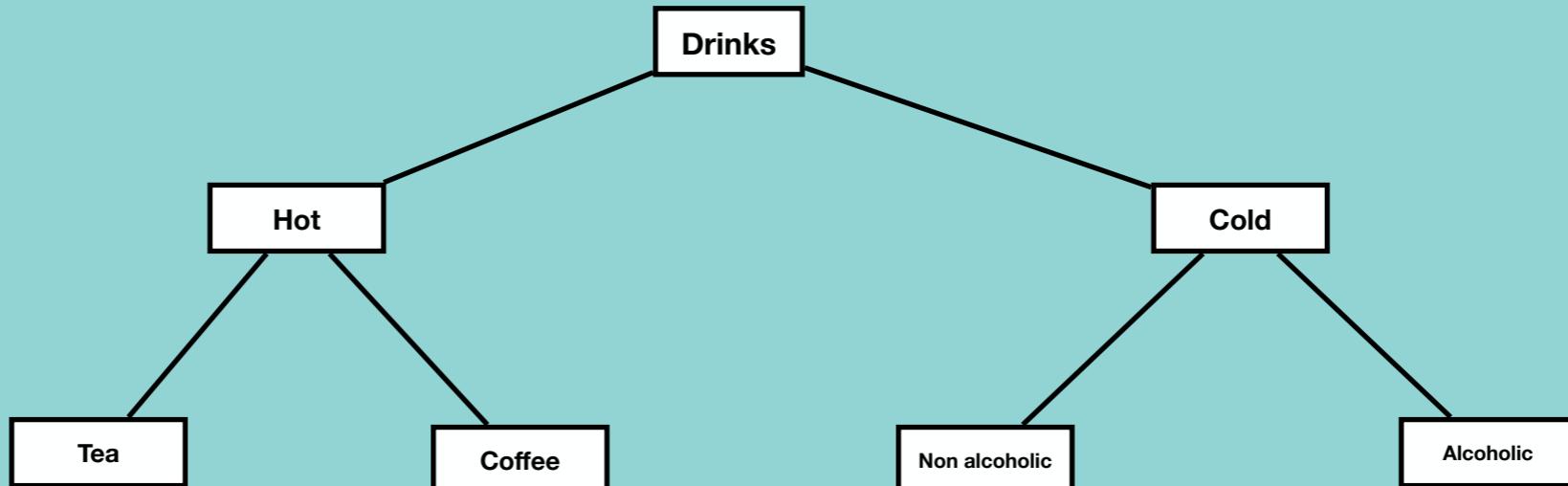
Left child = `cell[2x]` → `x = 1 , cell[2x1=2]`

Right child = `cell[2x+1]` → `x = 1 , cell[2x1+1=3]`



# Binary Tree

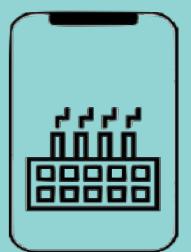
## Python List



0	1	2	3	4	5	6	7	8
✗	Drinks	Hot	Cold	Tea	Coffee			

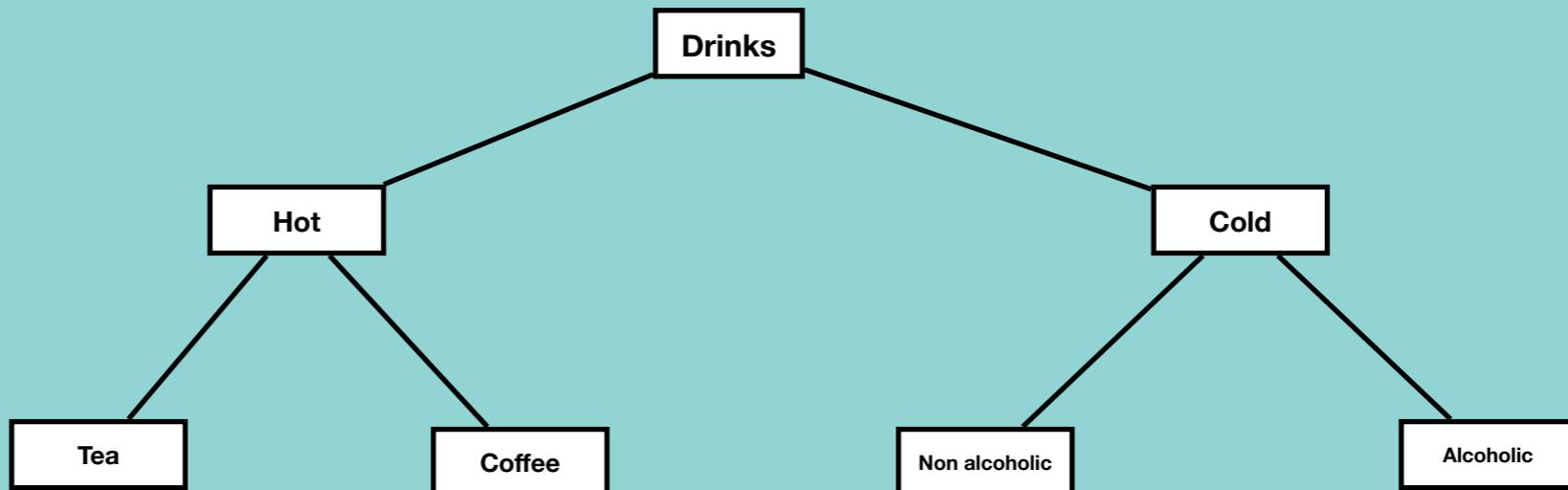
Left child = `cell[2x]` →  $x = 2, \text{cell}[2 \times 2 = 4]$

Right child = `cell[2x+1]` →  $x = 2, \text{cell}[2 \times 2 + 1 = 5]$



# Binary Tree

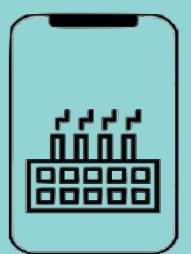
## Python List



0	1	2	3	4	5	6	7	8
✗	Drinks	Hot	Cold	Tea	Coffee	Non alcoholic	Alcoholic	

Left child = `cell[2x]` → `x = 3, cell[2x3=6]`

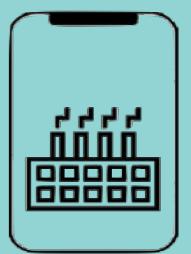
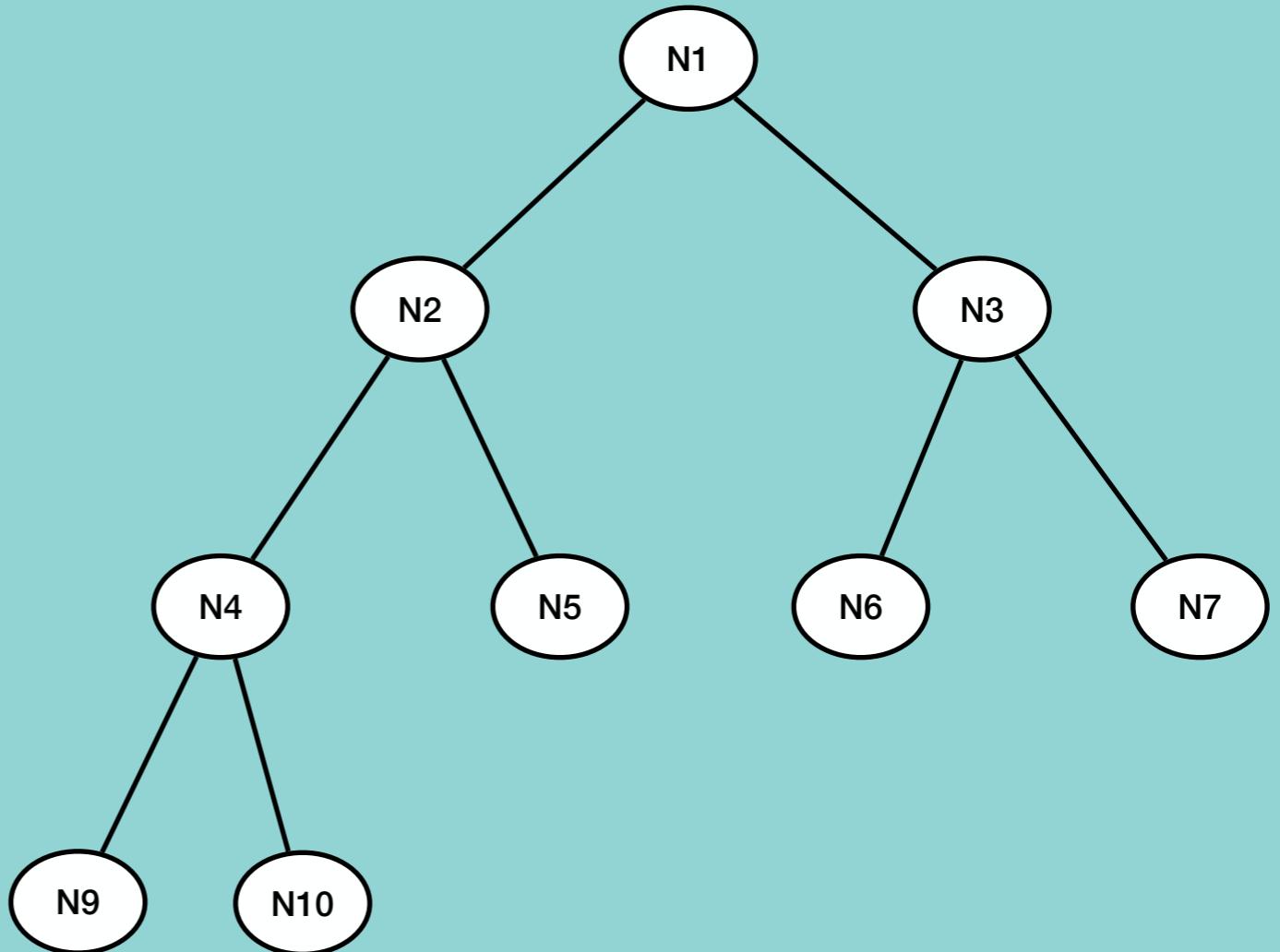
Right child = `cell[2x+1]` → `x = 3, cell[2x3+1=7]`



# Create Binary Tree using Linked List

- Creation of Tree
- Insertion of a node
- Deletion of a node
- Search for a value
- Traverse all nodes
- Deletion of tree

```
newTree = Tree()
```



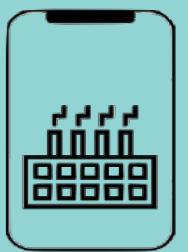
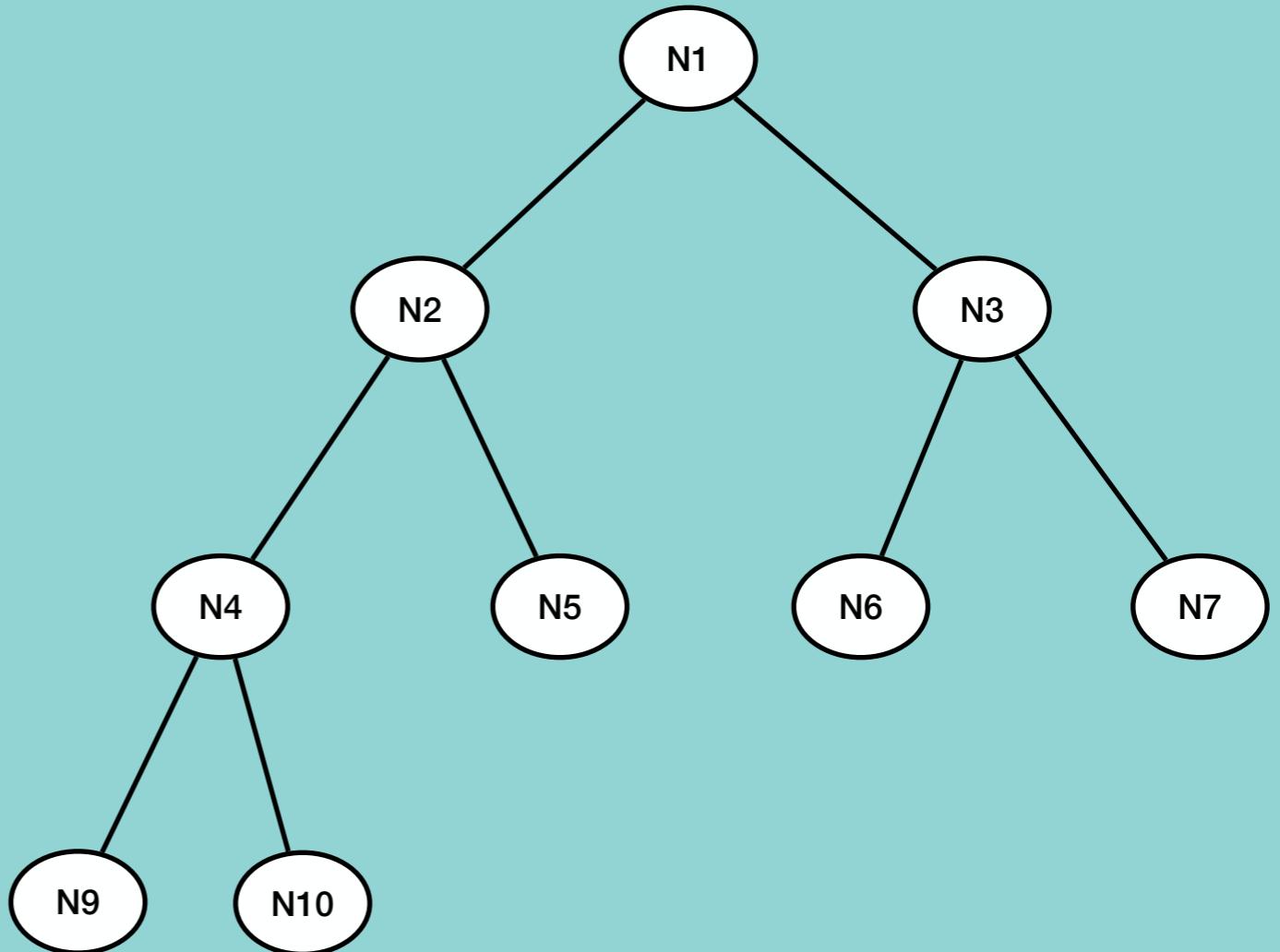
# Traversal of Binary Tree

## Depth first search

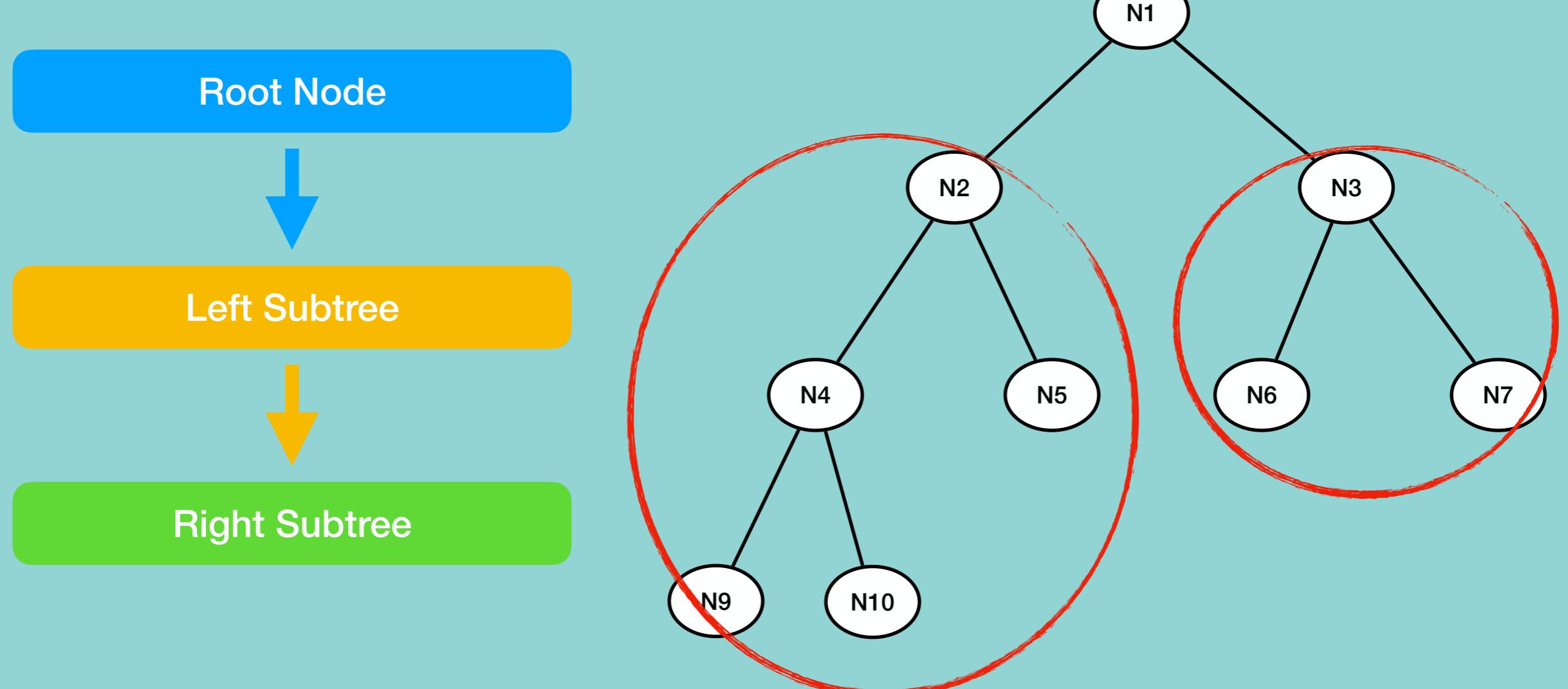
- Preorder traversal
- Inorder traversal
- Post order traversal

## Breadth first search

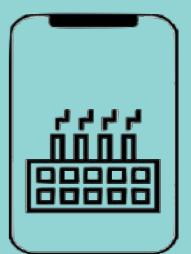
- Level order traversal



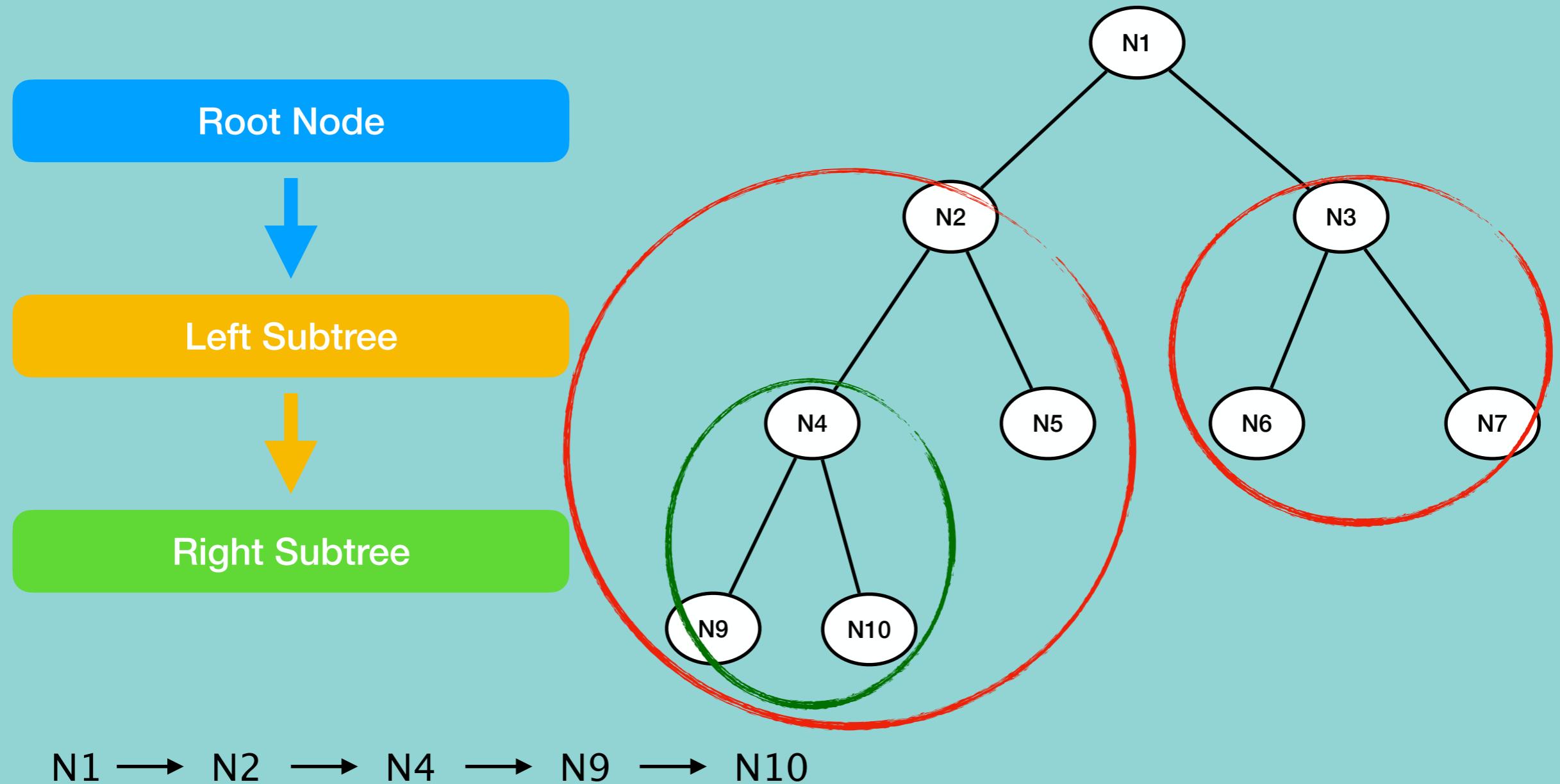
# PreOrder Traversal of Binary Tree



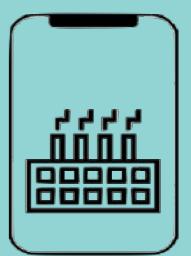
$N1 \rightarrow N2$



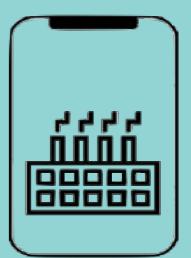
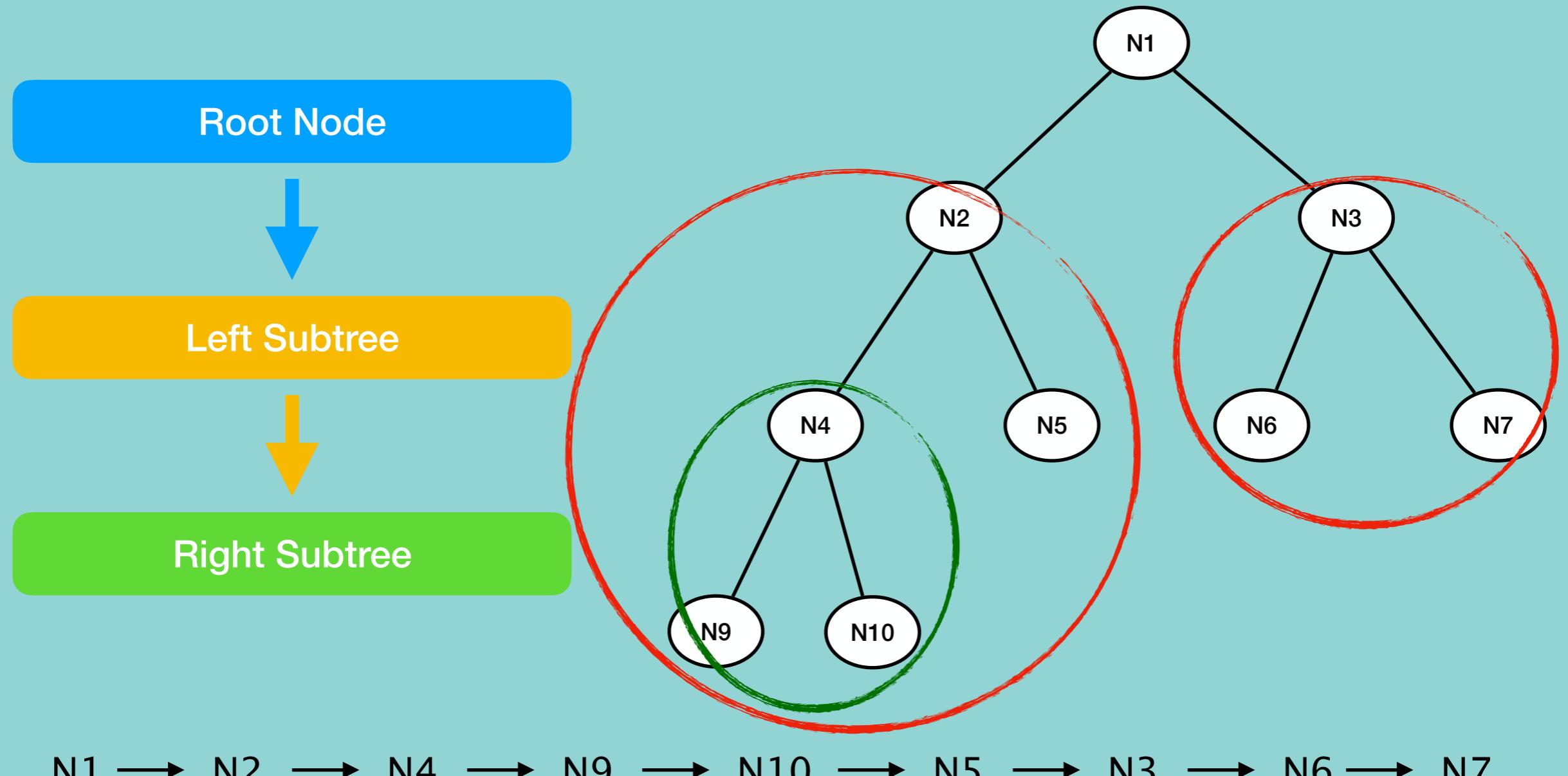
# PreOrder Traversal of Binary Tree



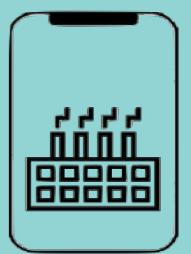
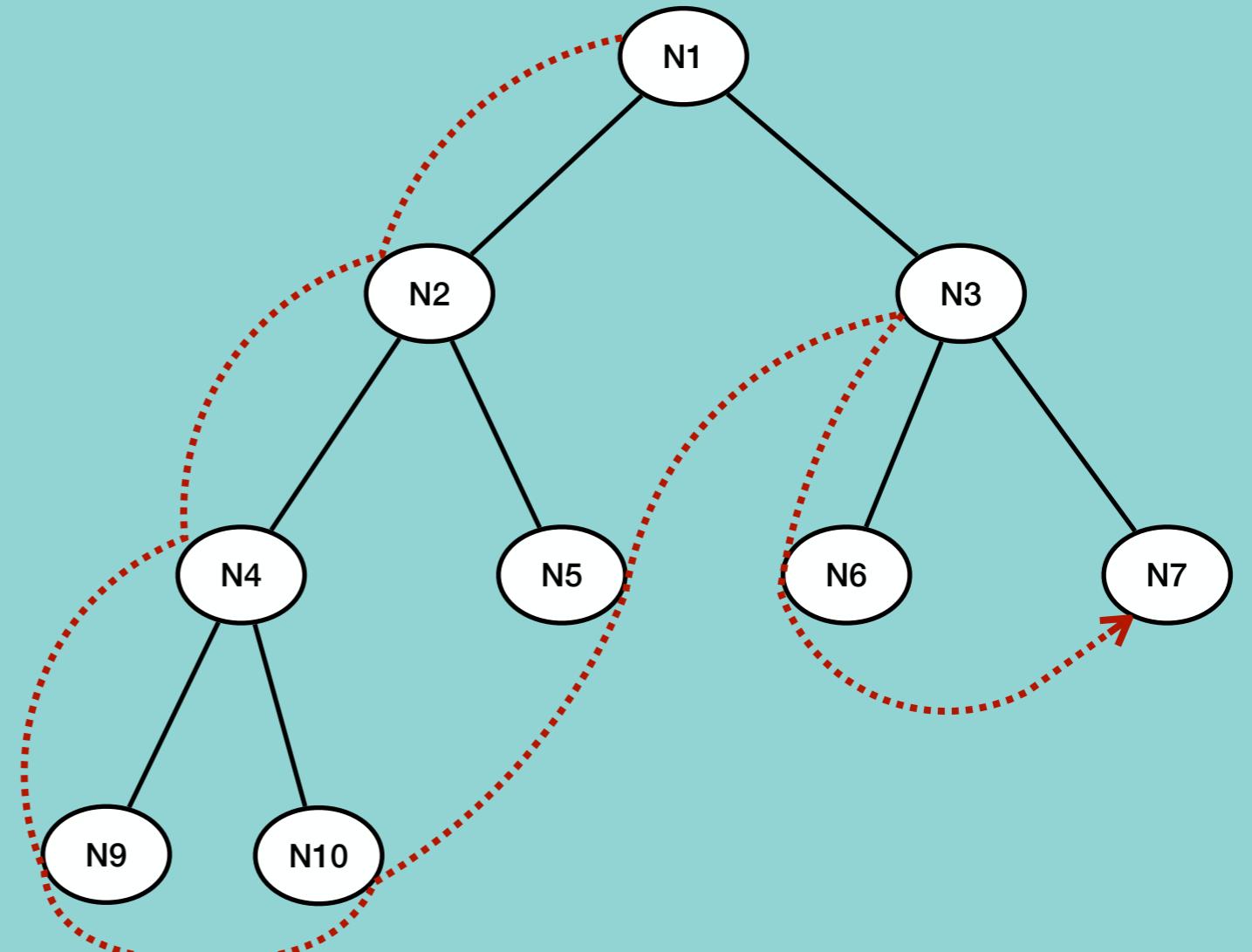
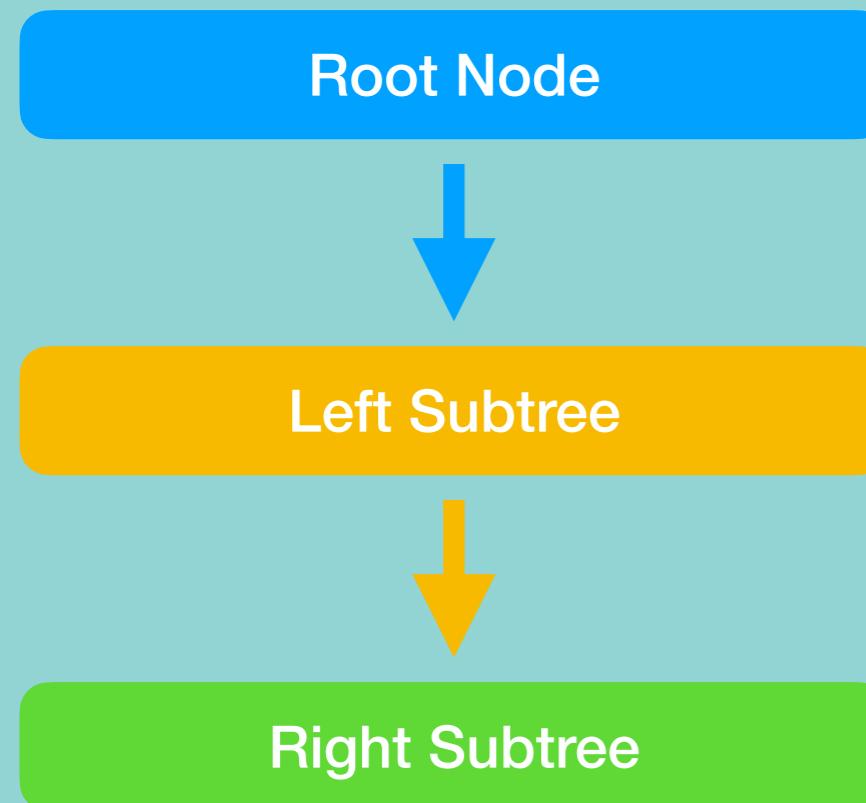
N1 → N2 → N4 → N9 → N10



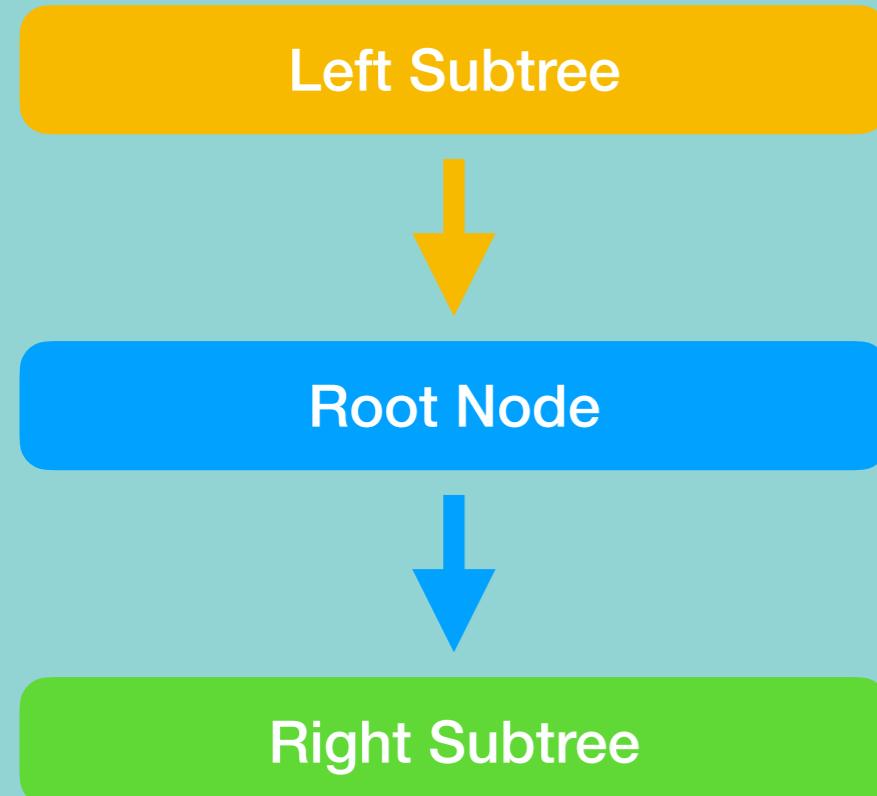
# PreOrder Traversal of Binary Tree



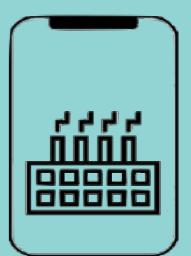
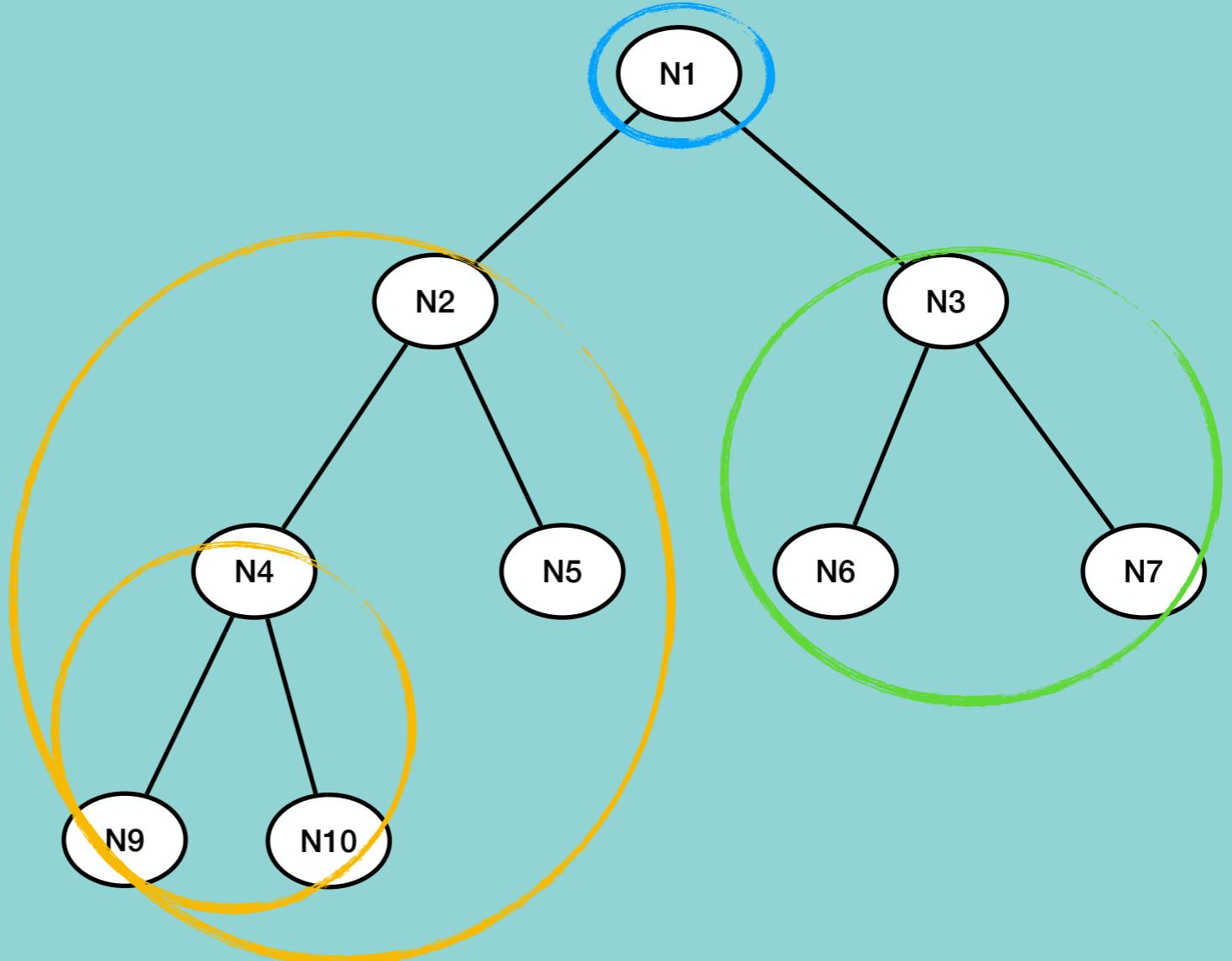
# PreOrder Traversal of Binary Tree



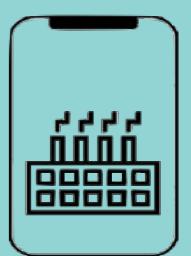
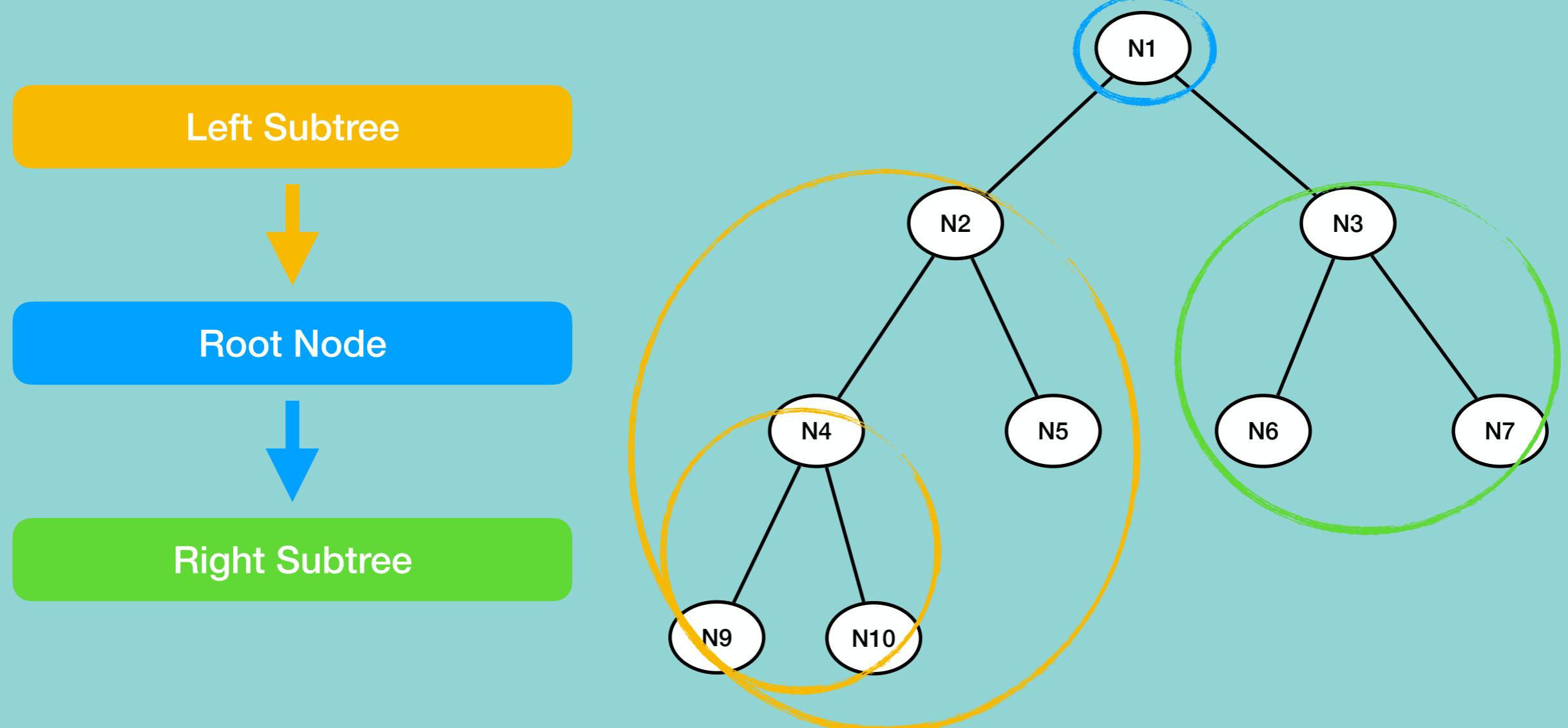
# InOrder Traversal of Binary Tree



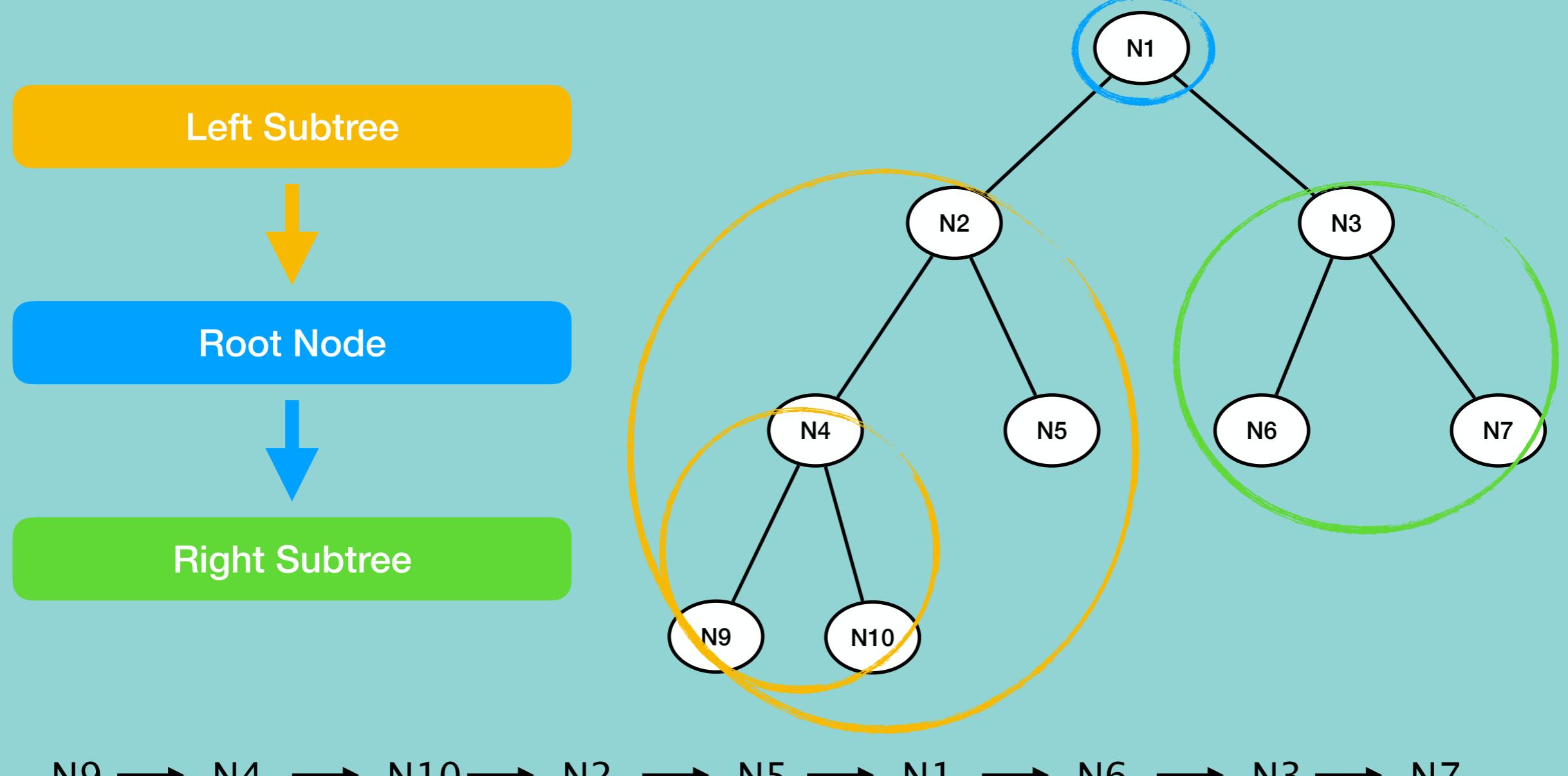
$N9 \rightarrow N4$



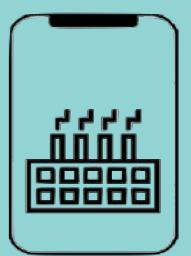
# InOrder Traversal of Binary Tree



# InOrder Traversal of Binary Tree



N9 → N4 → N10 → N2 → N5 → N1 → N6 → N3 → N7



# InOrder Traversal of Binary Tree

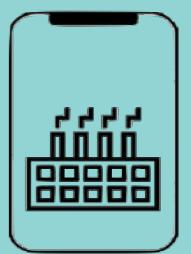
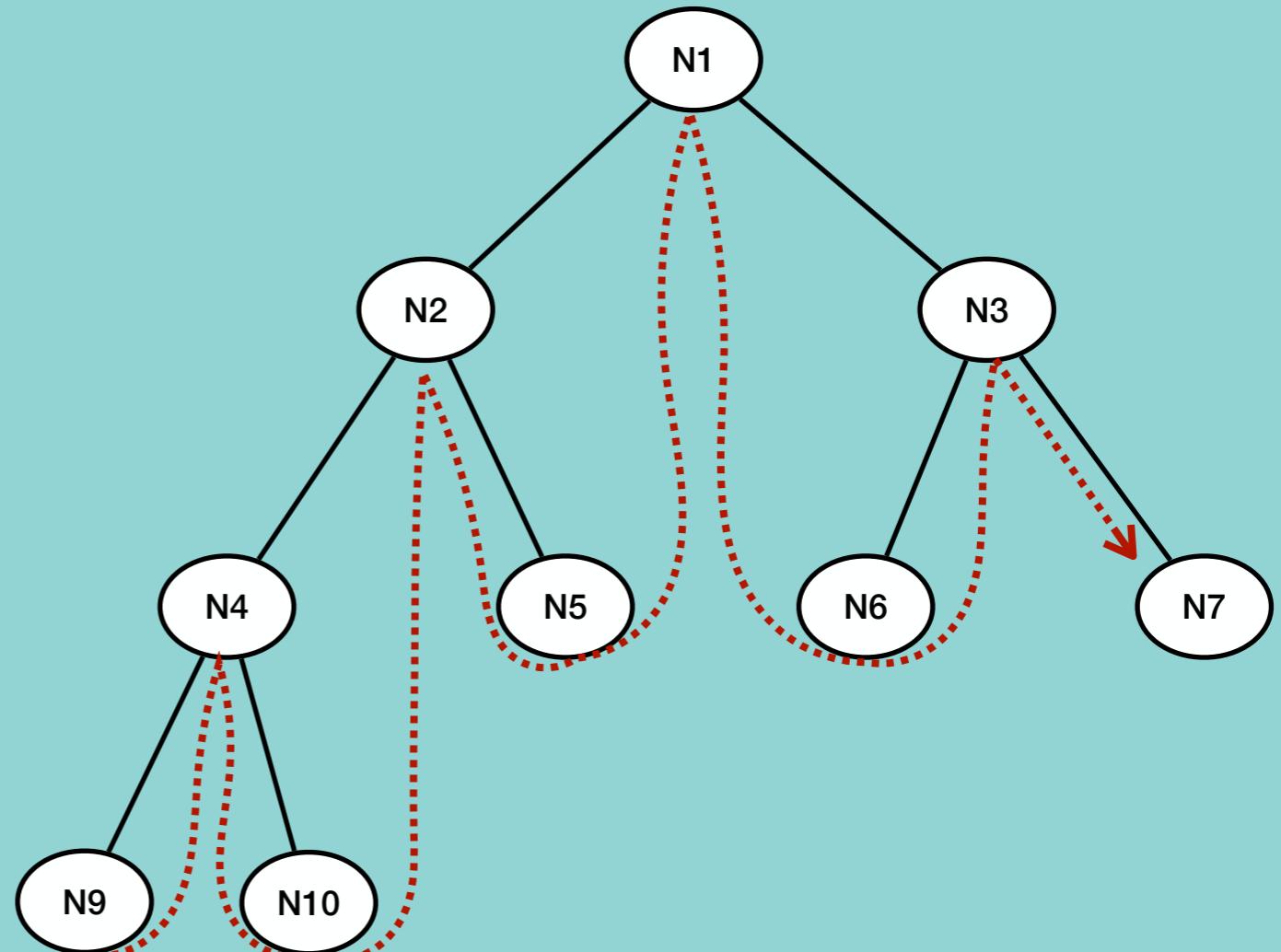
Left Subtree



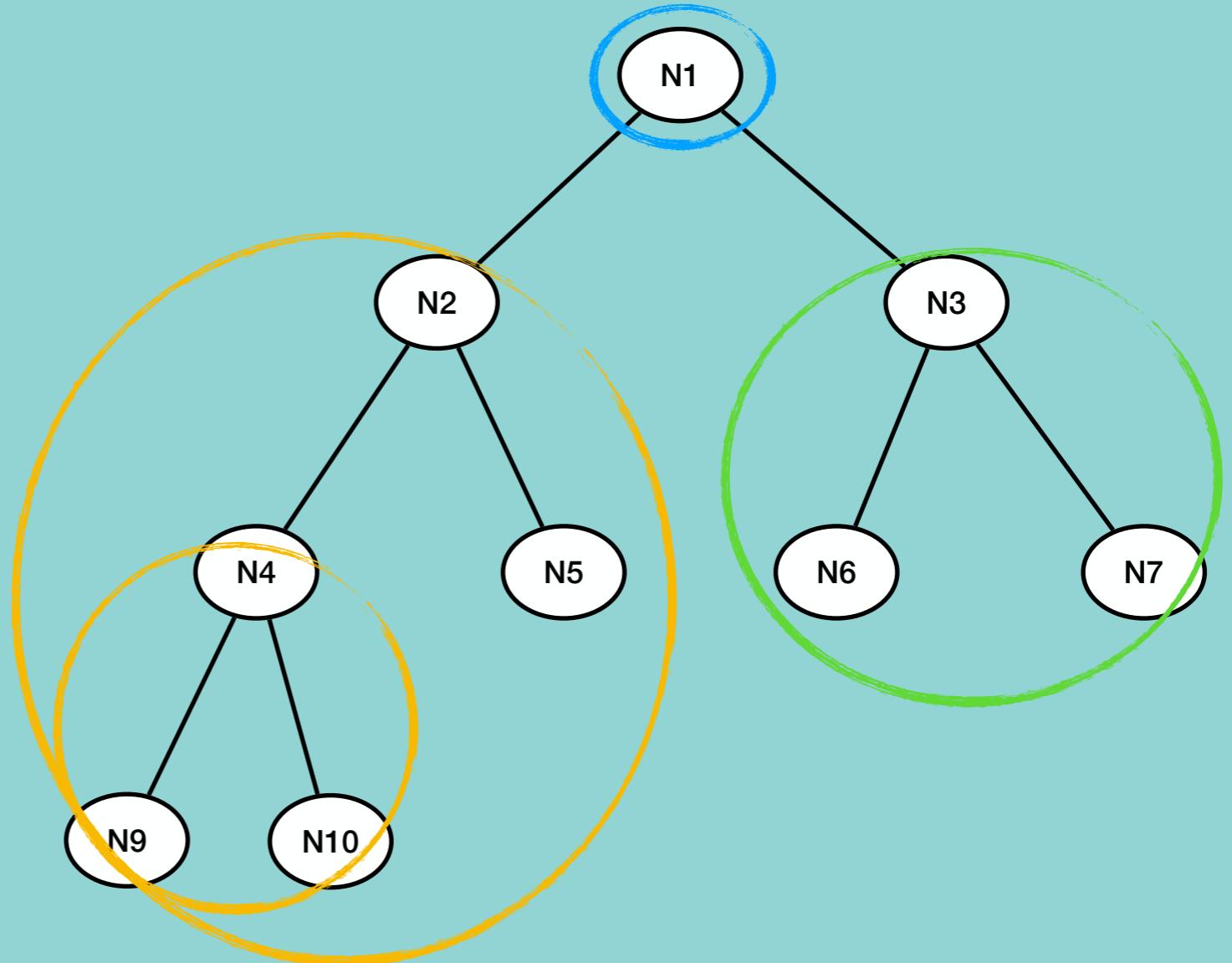
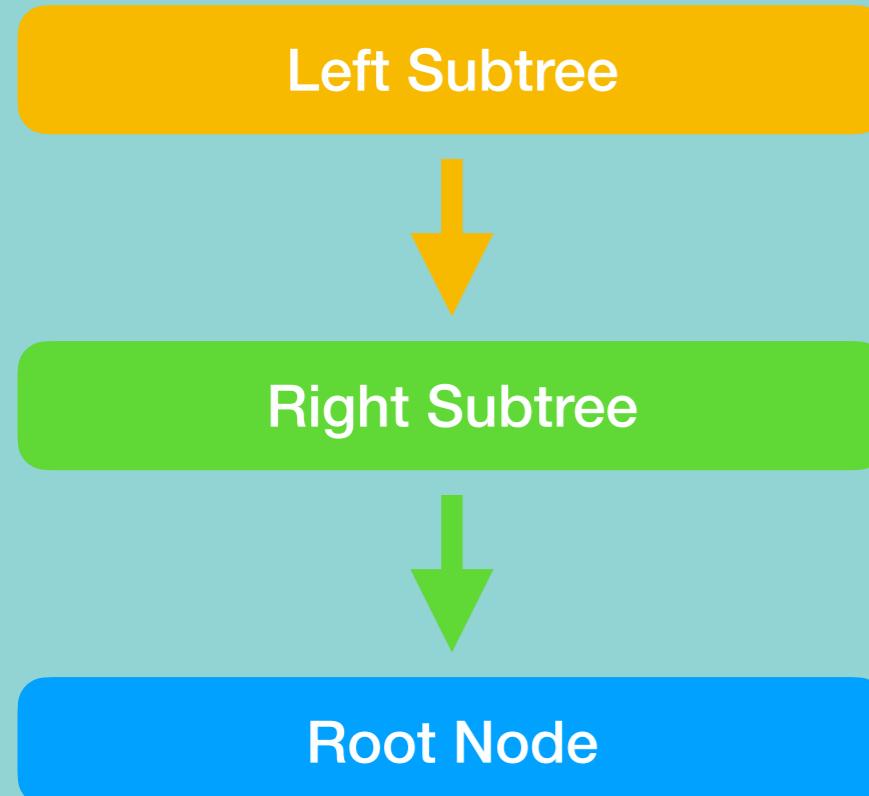
Root Node



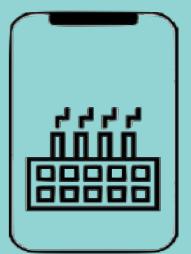
Right Subtree



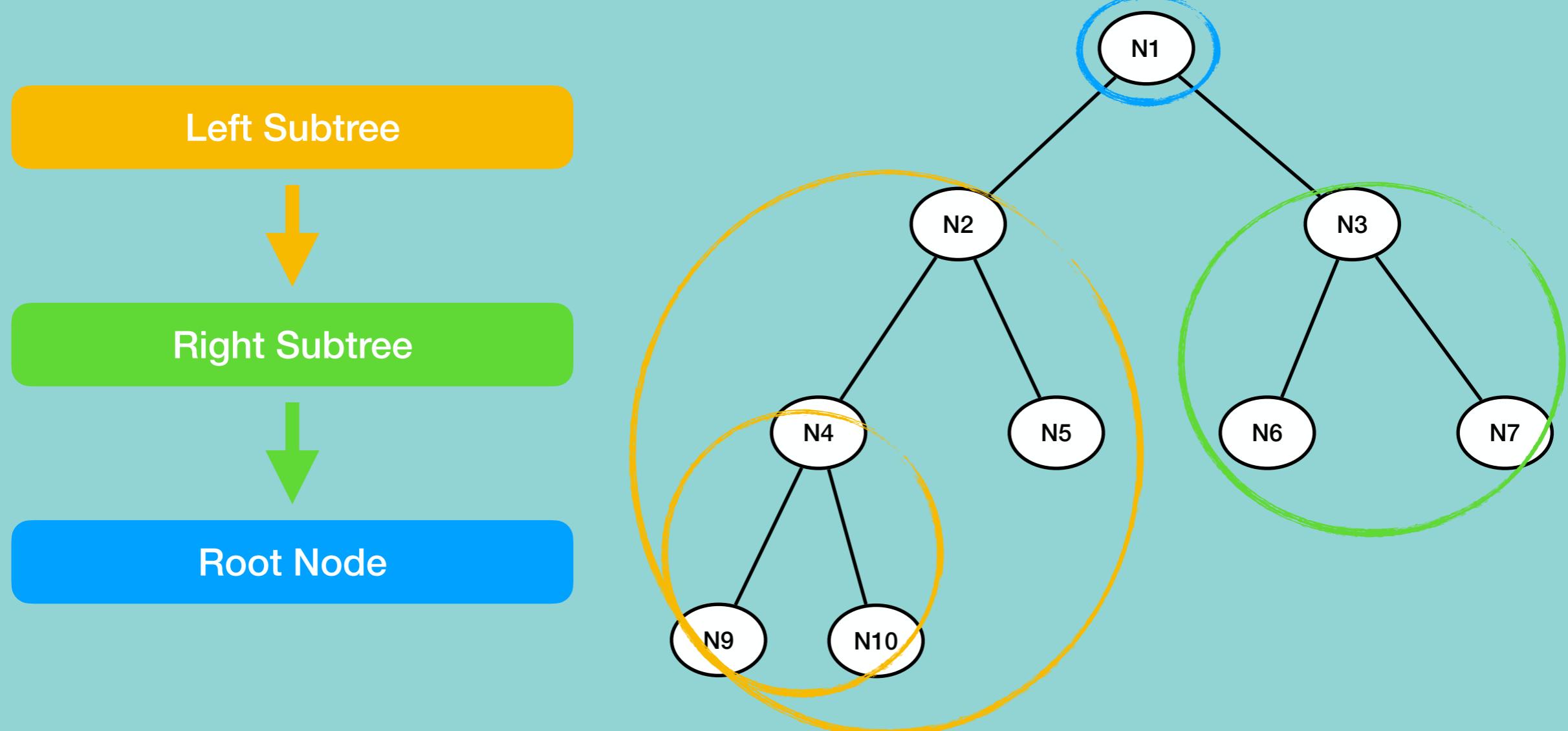
# PostOrder Traversal of Binary Tree



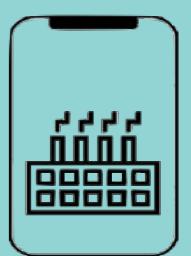
$N9 \rightarrow N10$



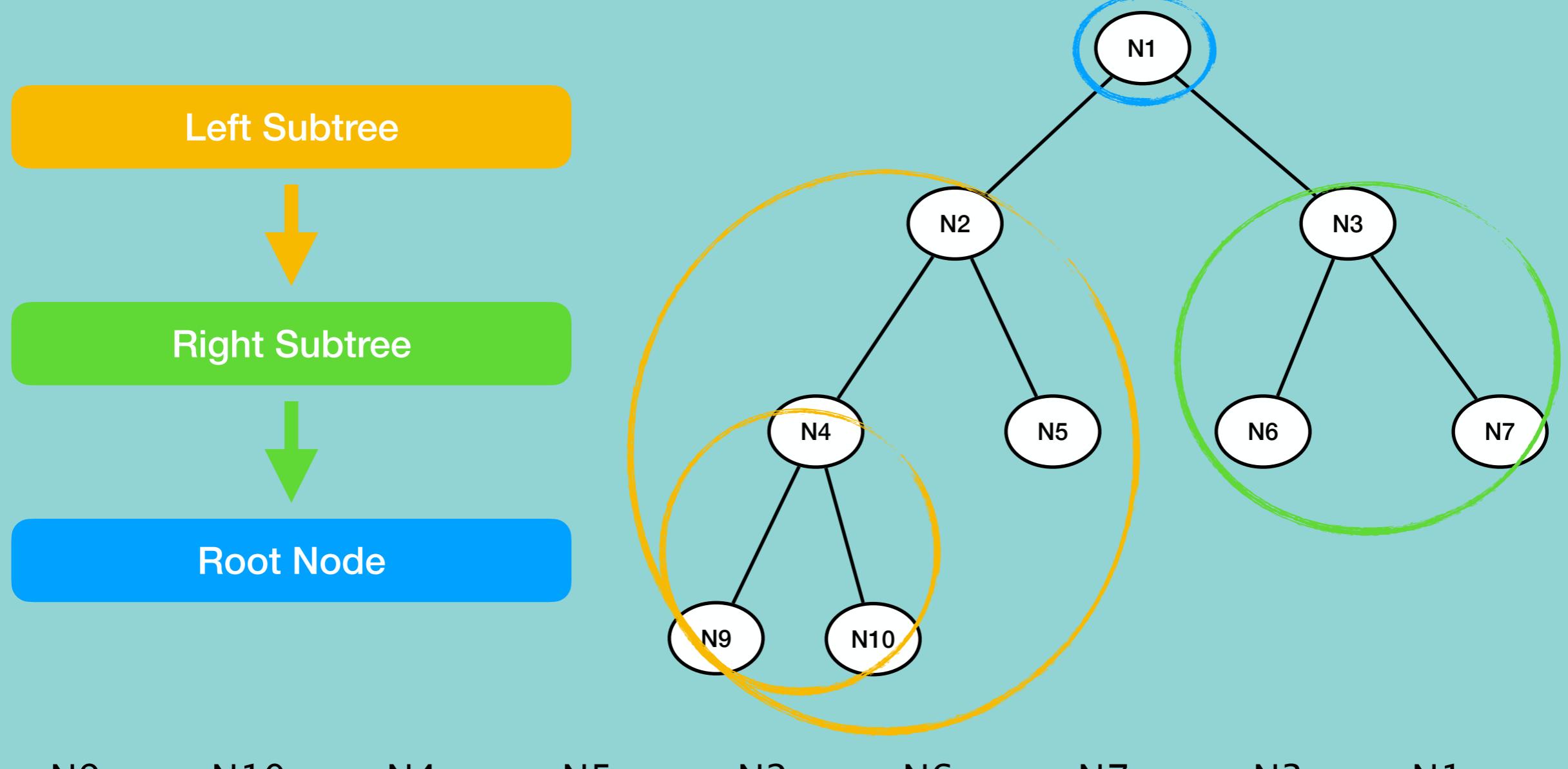
# PostOrder Traversal of Binary Tree



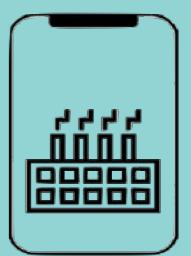
N9 → N10 → N4 → N5 → N2



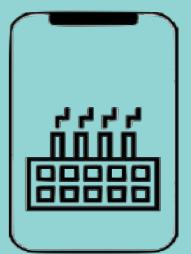
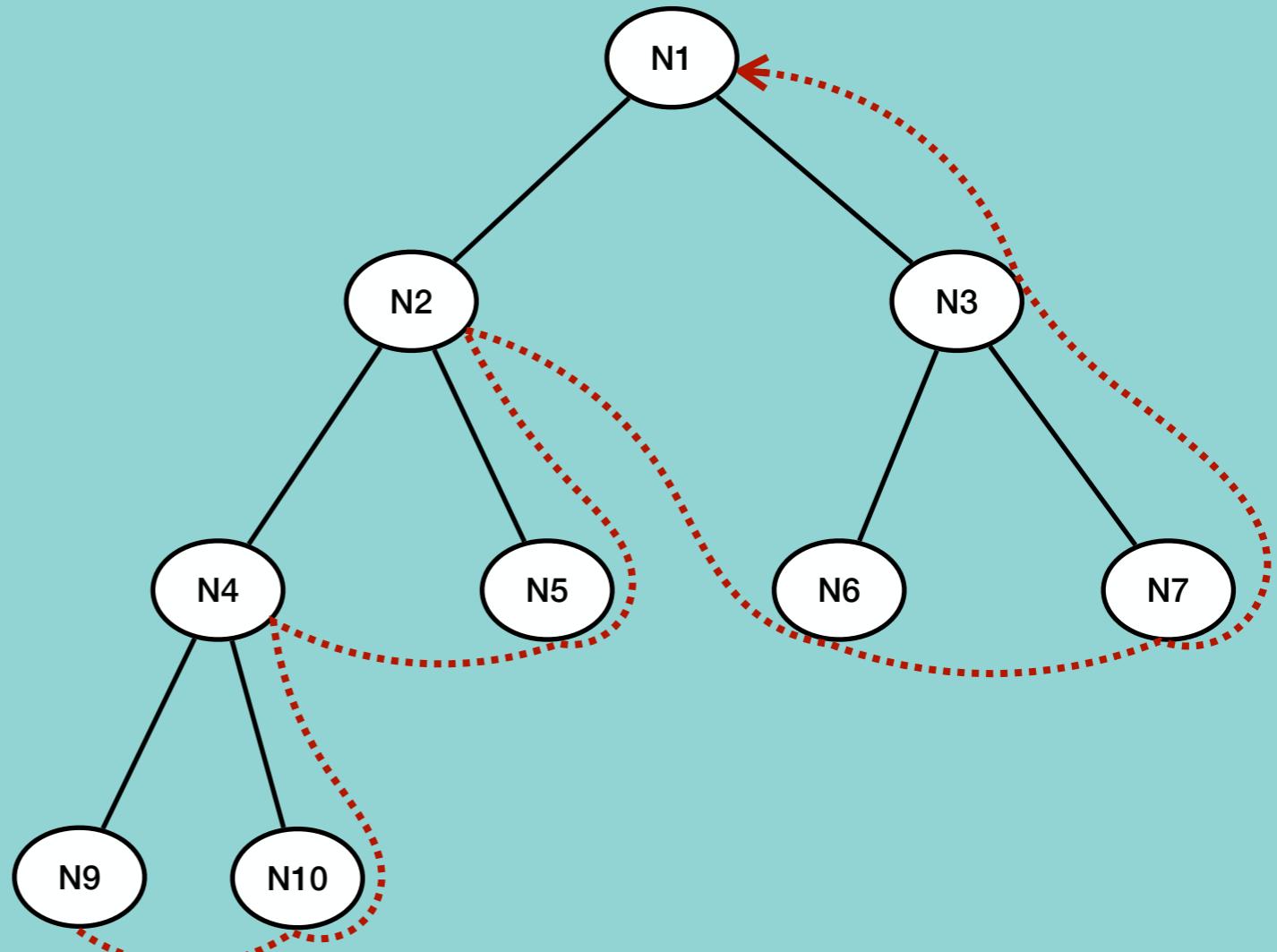
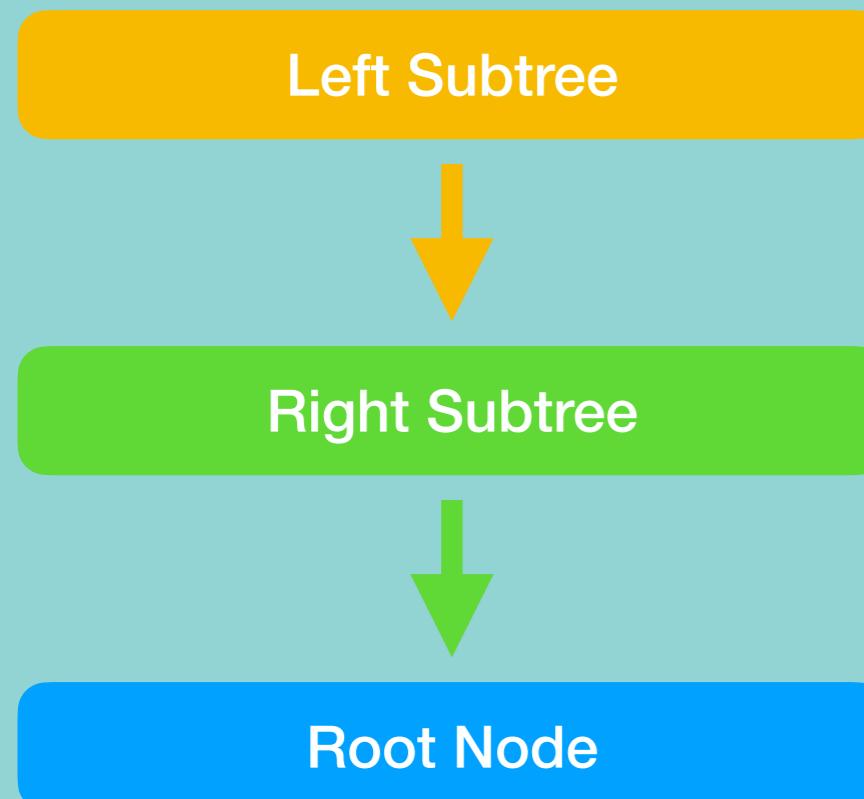
# PostOrder Traversal of Binary Tree



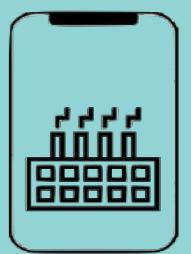
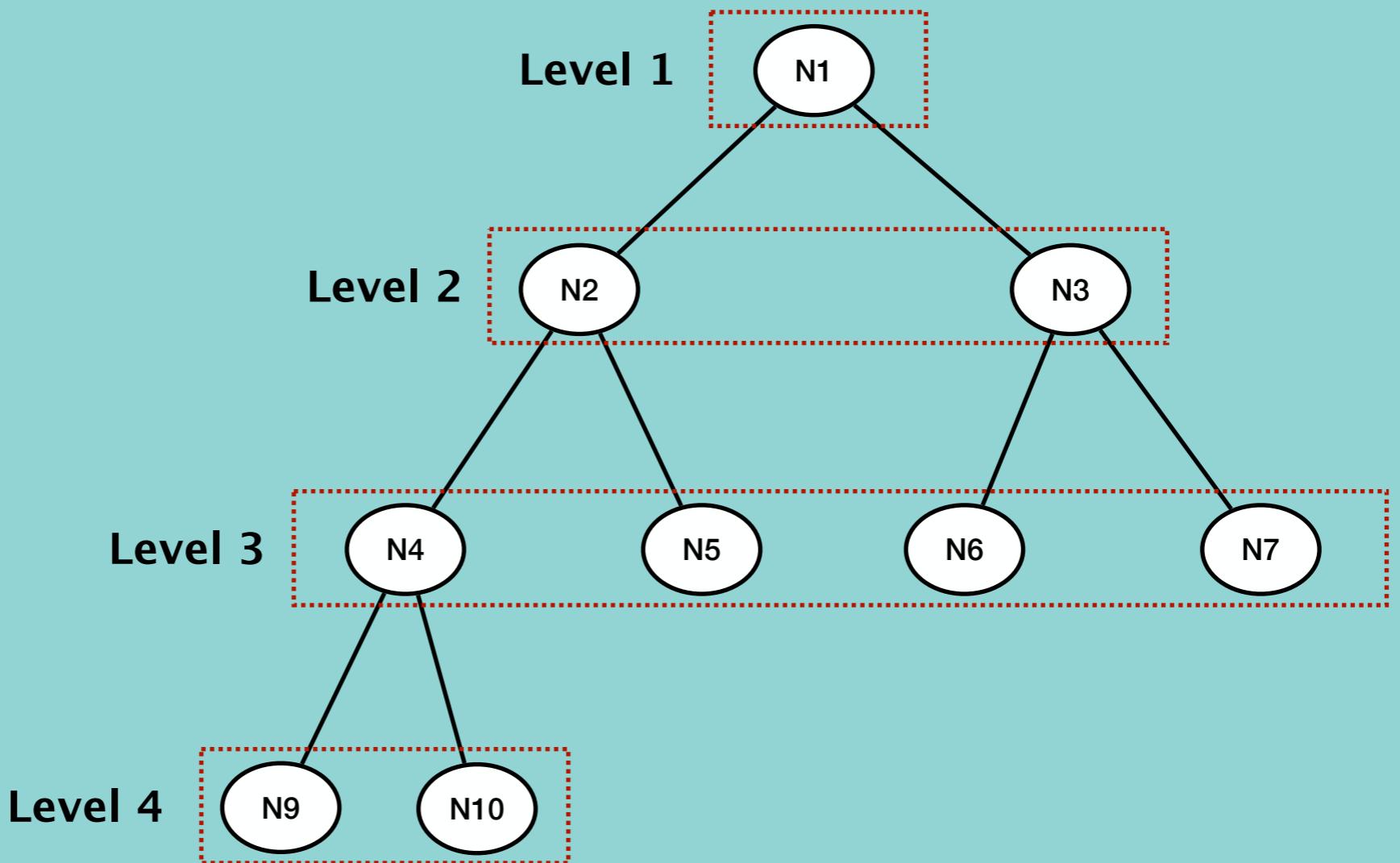
N9 → N10 → N4 → N5 → N2 → N6 → N7 → N3 → N1



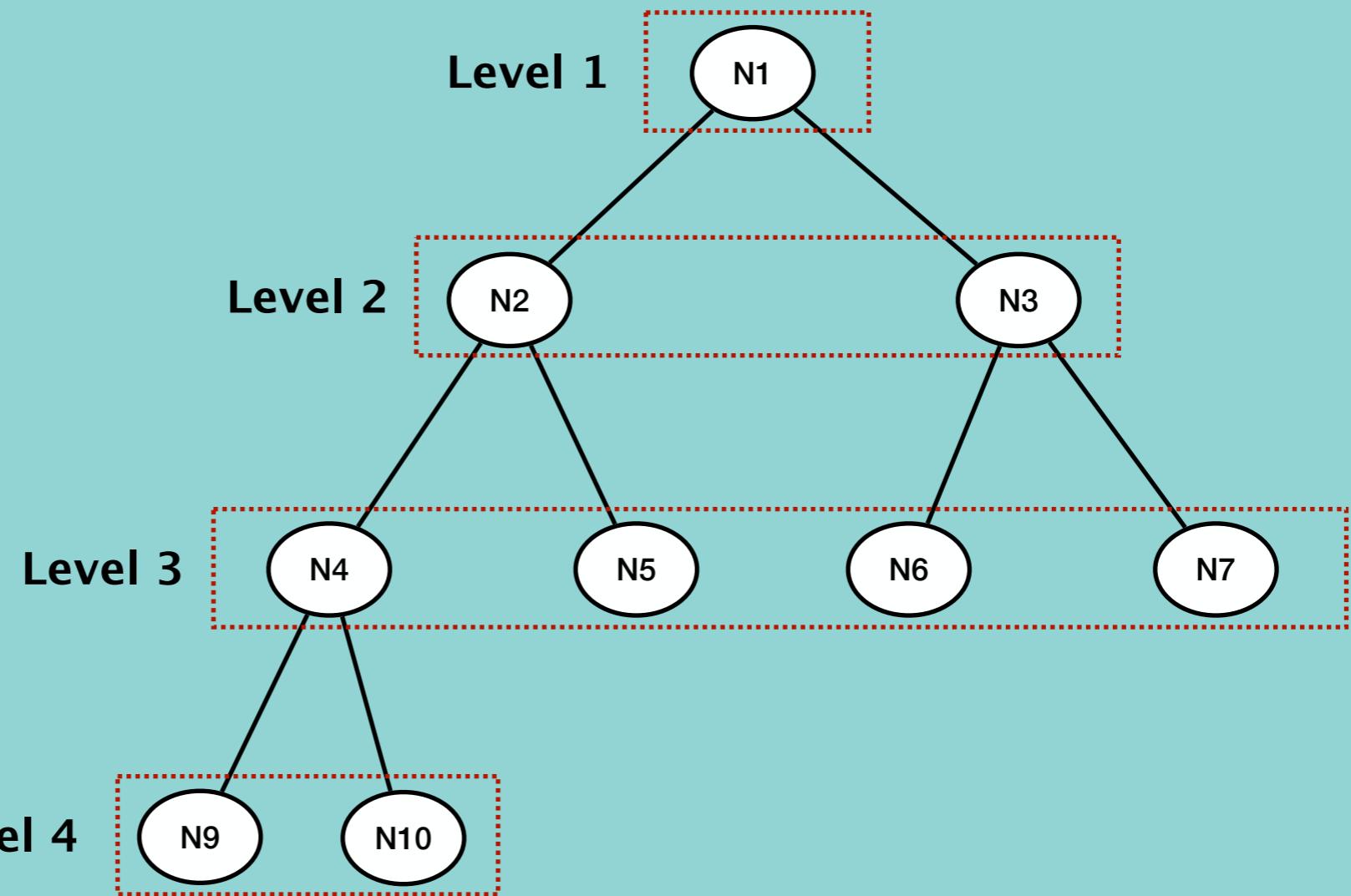
# PostOrder Traversal of Binary Tree



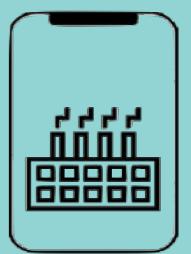
# LevelOrder Traversal of Binary Tree



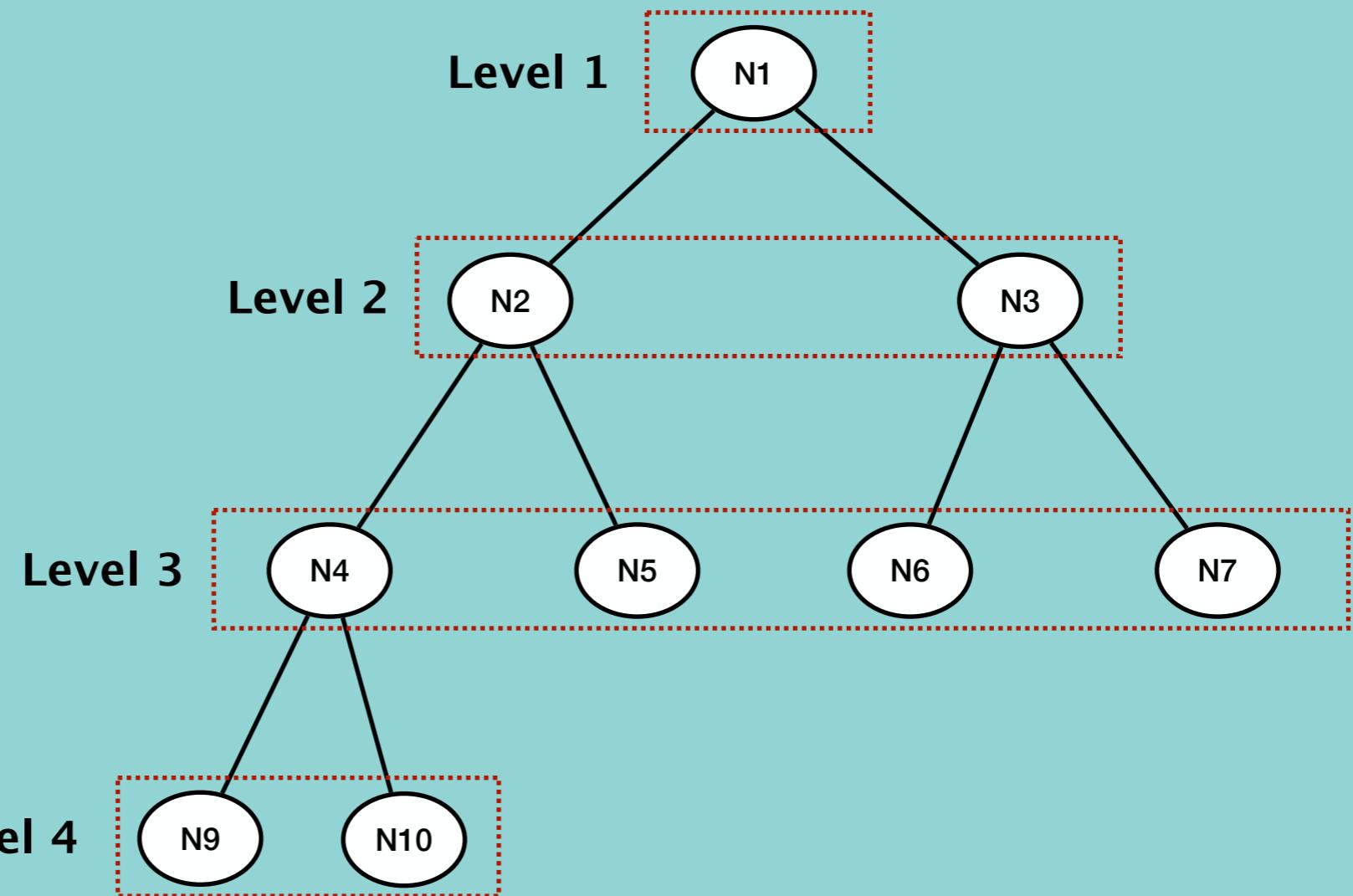
# LevelOrder Traversal of Binary Tree



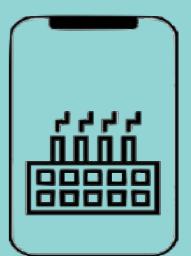
N1 → N2 → N3 → N4 → N5



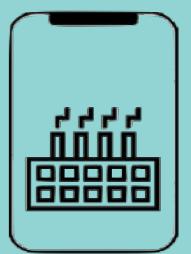
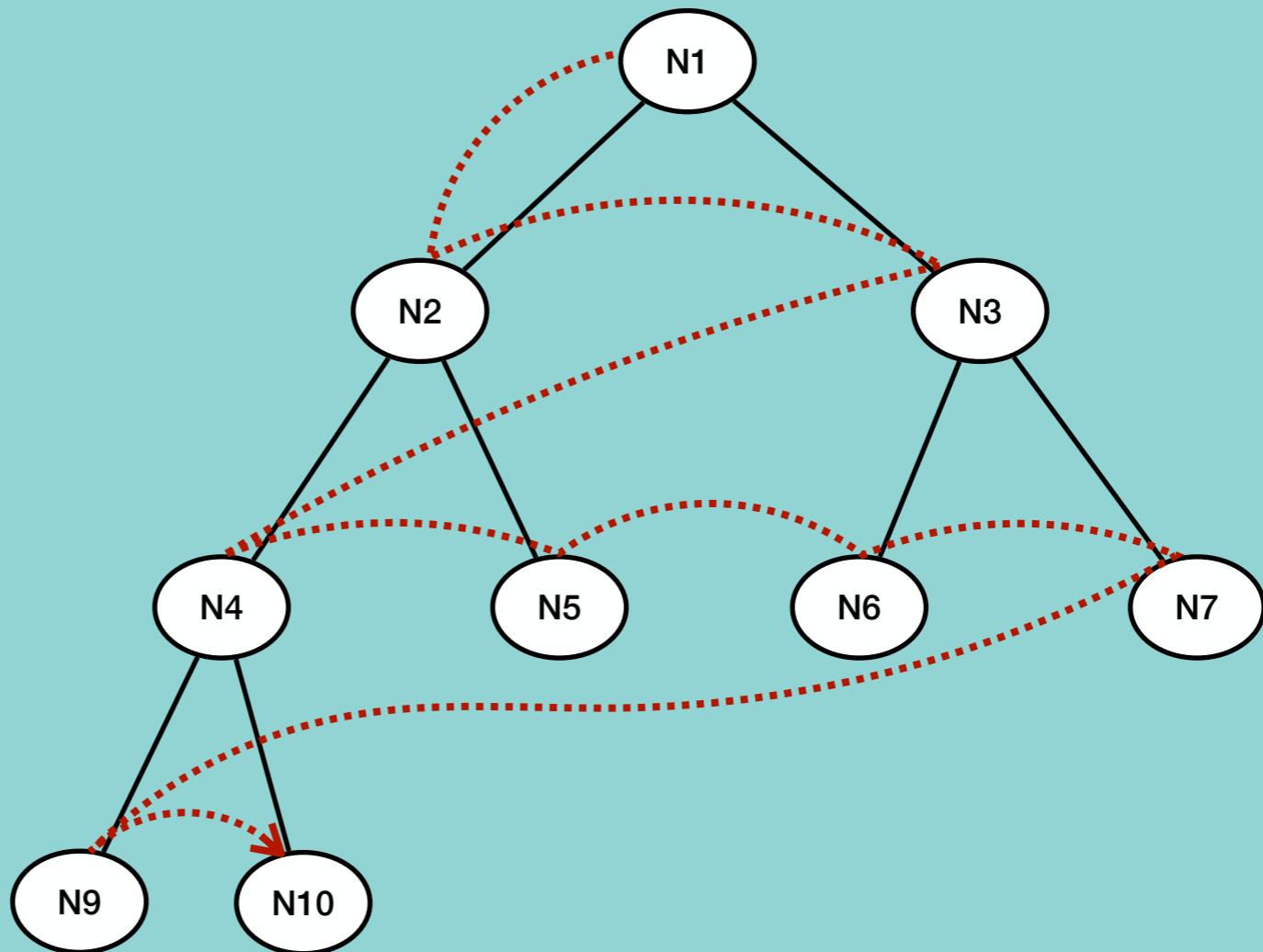
# LevelOrder Traversal of Binary Tree



N1 → N2 → N3 → N4 → N5 → N6 → N7 → N9 → N10



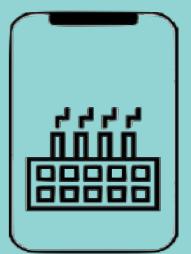
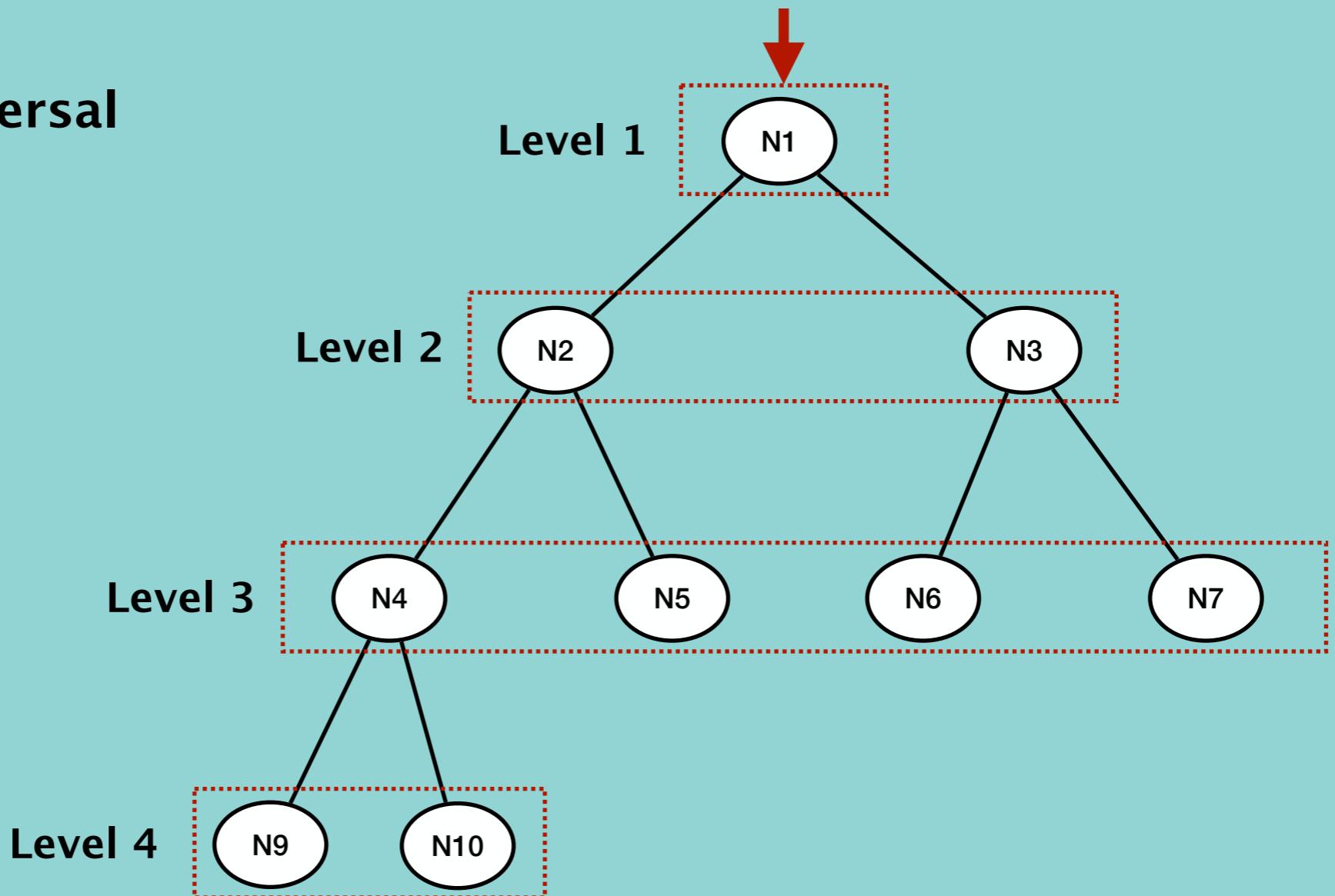
# LevelOrder Traversal of Binary Tree



# Searching for a node in Binary Tree

## Level Order Traversal

N5 → Success

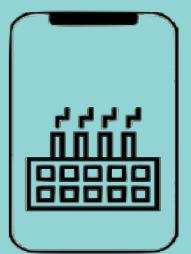
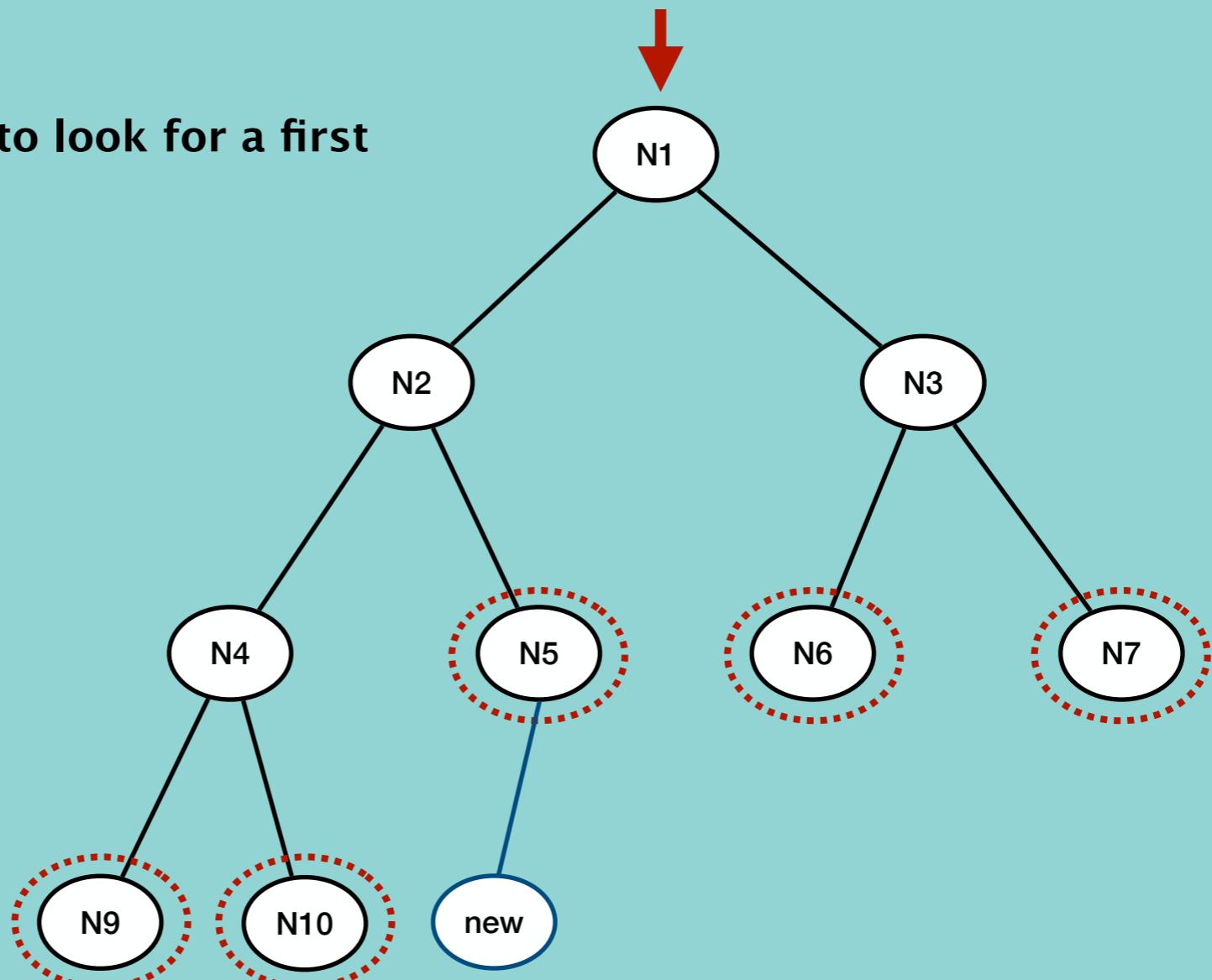


# Insert a node in Binary Tree

- A root node is blank
- The tree exists and we have to look for a first vacant place

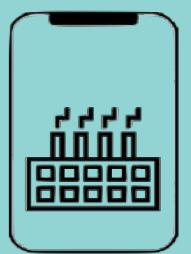
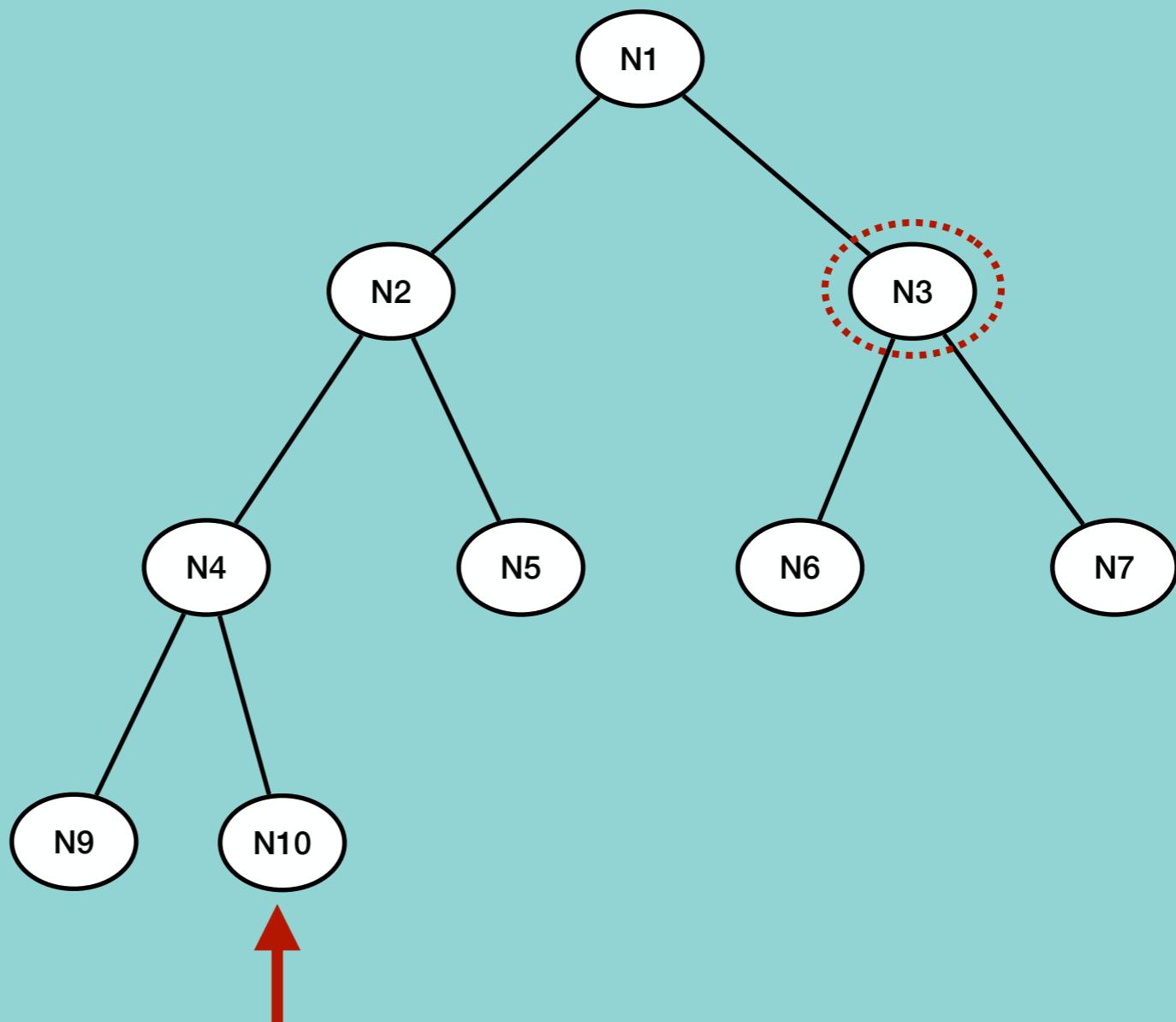
## Level Order Traversal

**newNode**



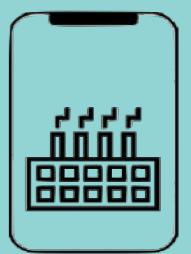
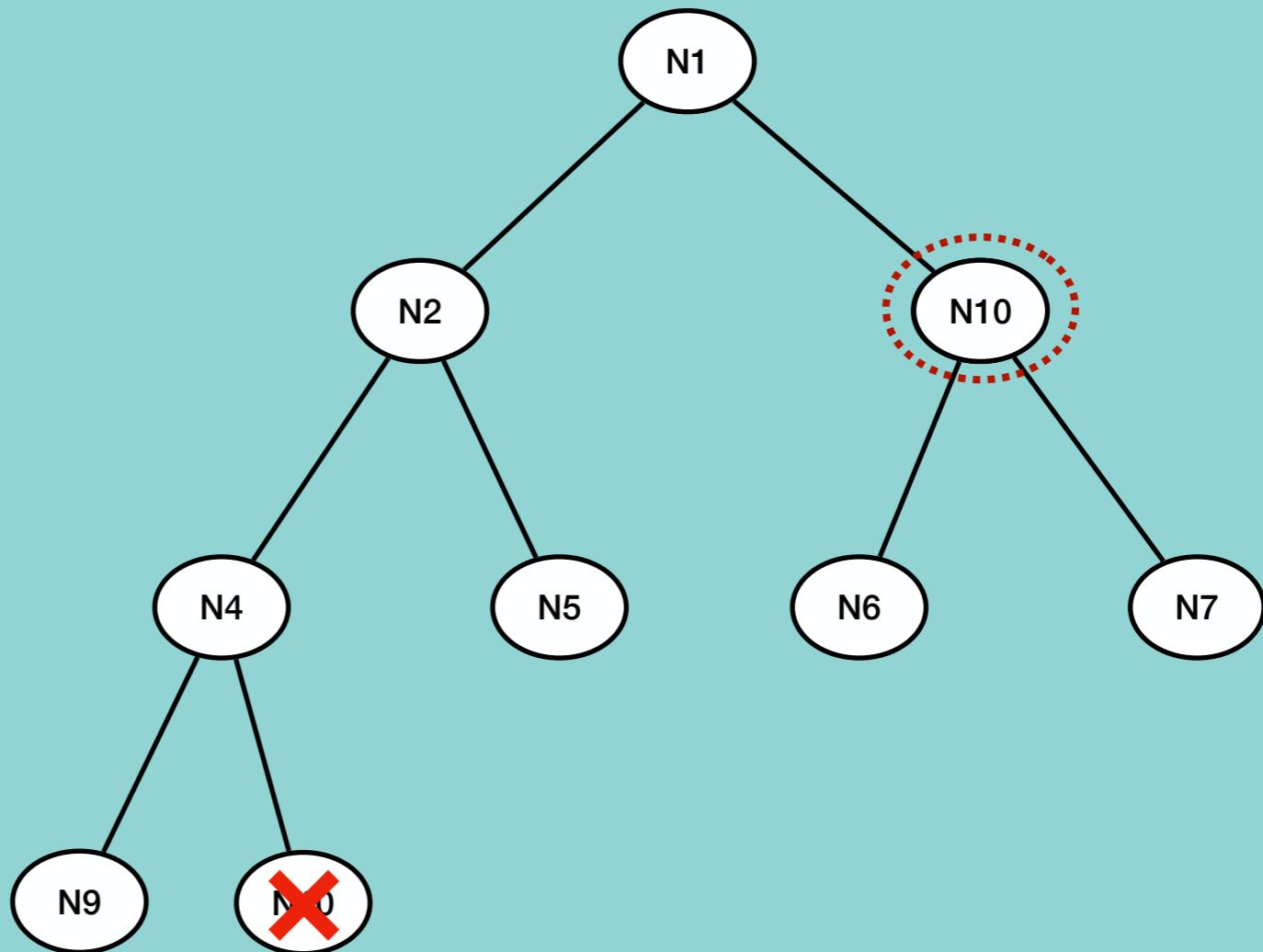
# Delete a node from Binary Tree

## Level Order Traversal



# Delete a node from Binary Tree

## Level Order Traversal

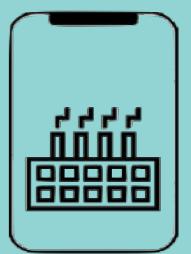
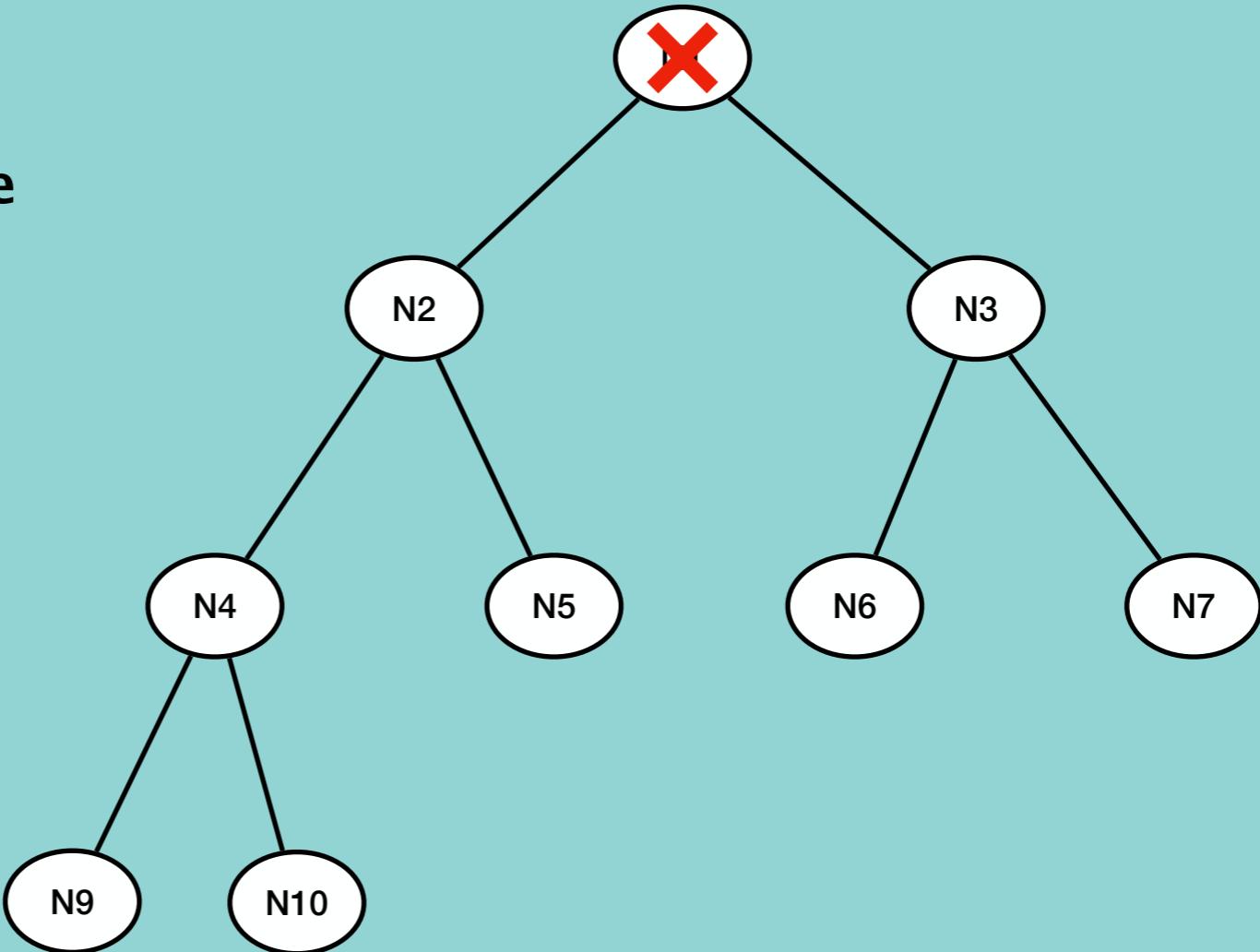


# Delete entire Binary Tree

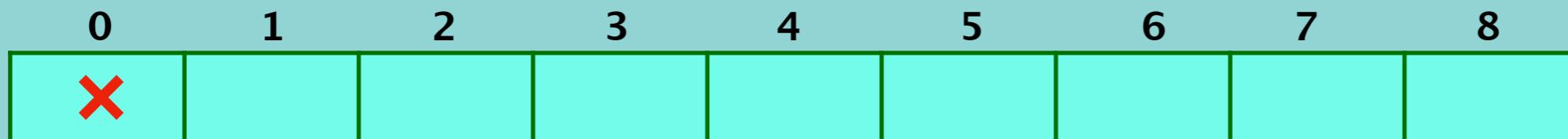
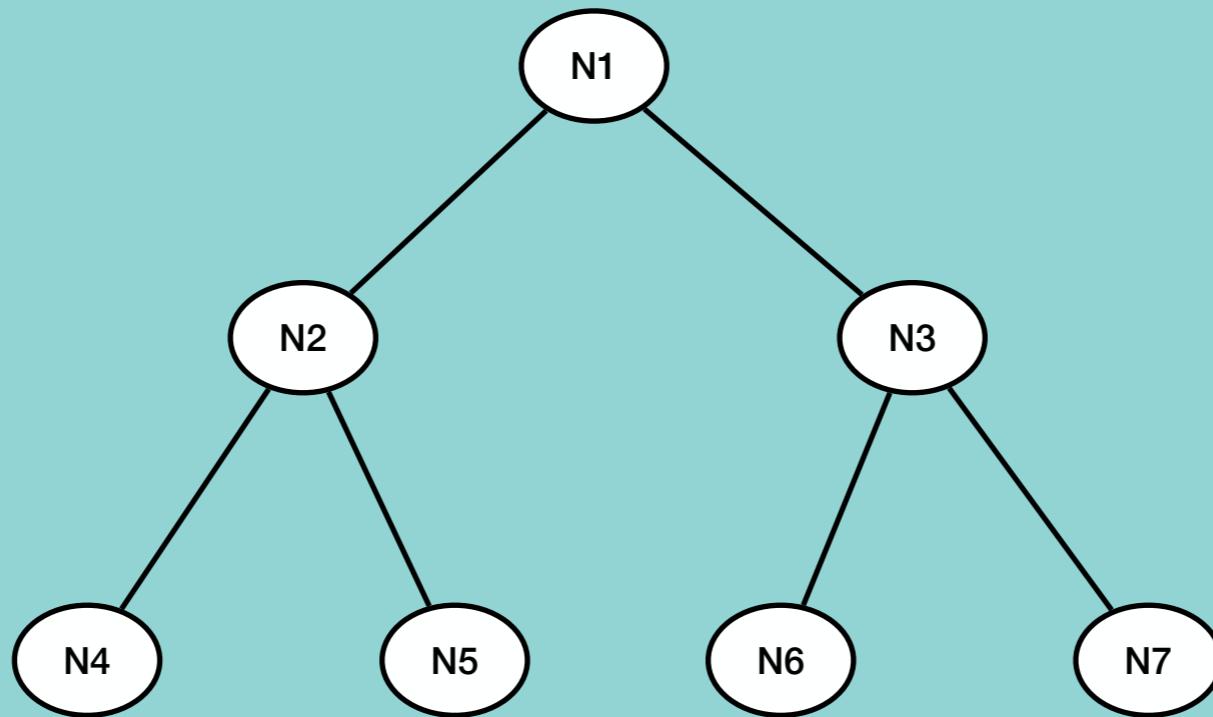
```
rootNode = None
```

```
rootNode.leftChild = None
```

```
rootNode.rightChild = None
```

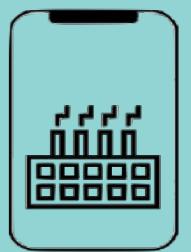


# Create Binary Tree using Python List

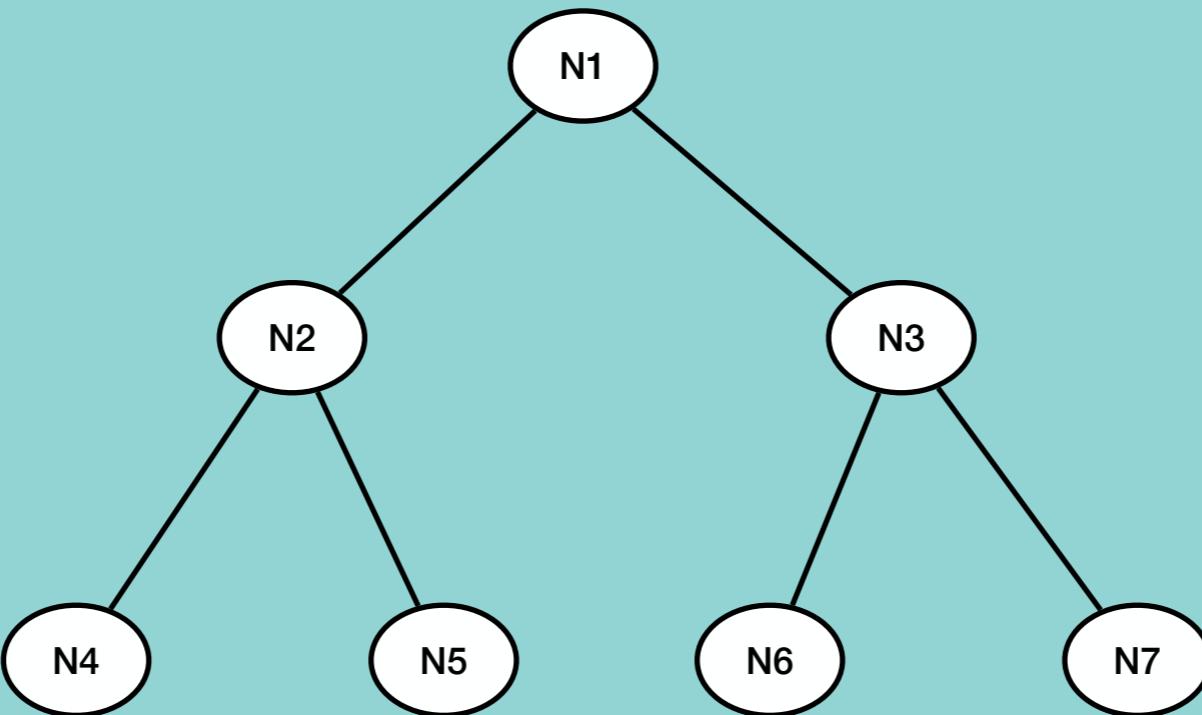


**Left child = cell[2x]**

**Right child = cell[2x+1]**



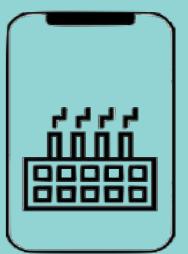
# Create Binary Tree using Python List



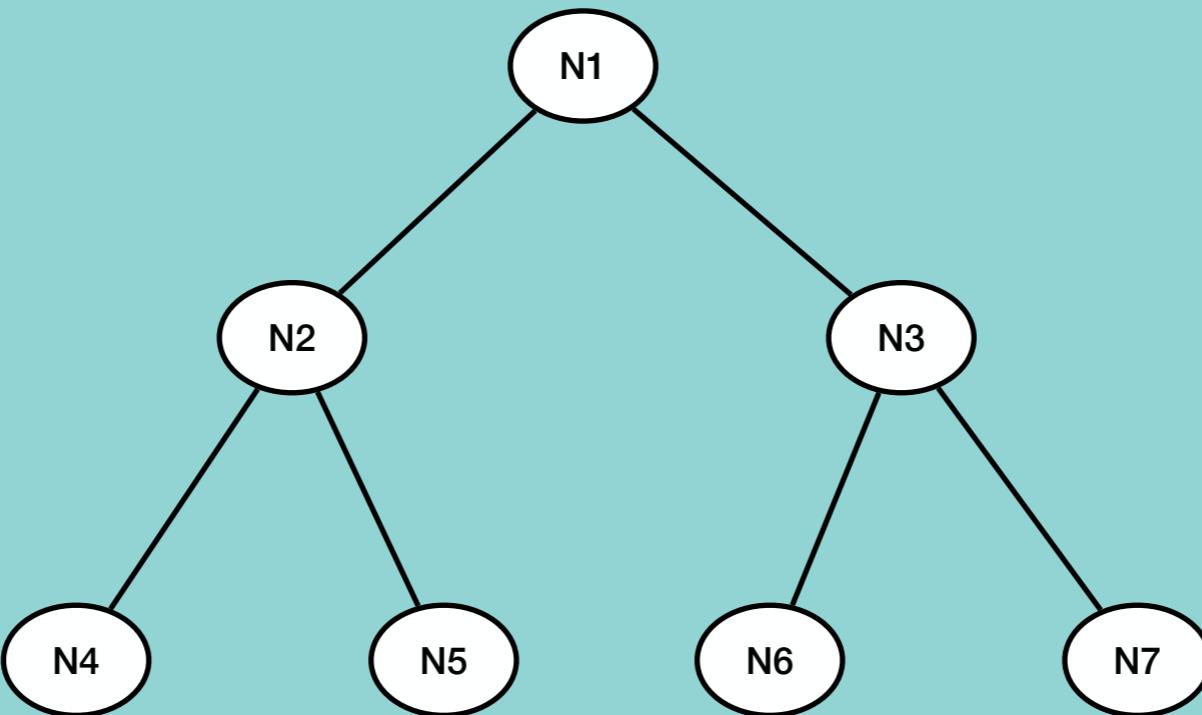
0	1	2	3	4	5	6	7	8
✗	N1	N2	N3					

Left child = `cell[2x]`  $\longrightarrow$  `x = 1 , cell[2x1=2]`

Right child = `cell[2x+1]`  $\longrightarrow$  `x = 1 , cell[2x1+1=3]`



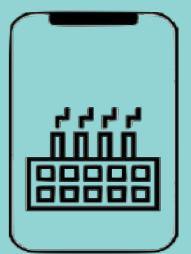
# Create Binary Tree using Python List



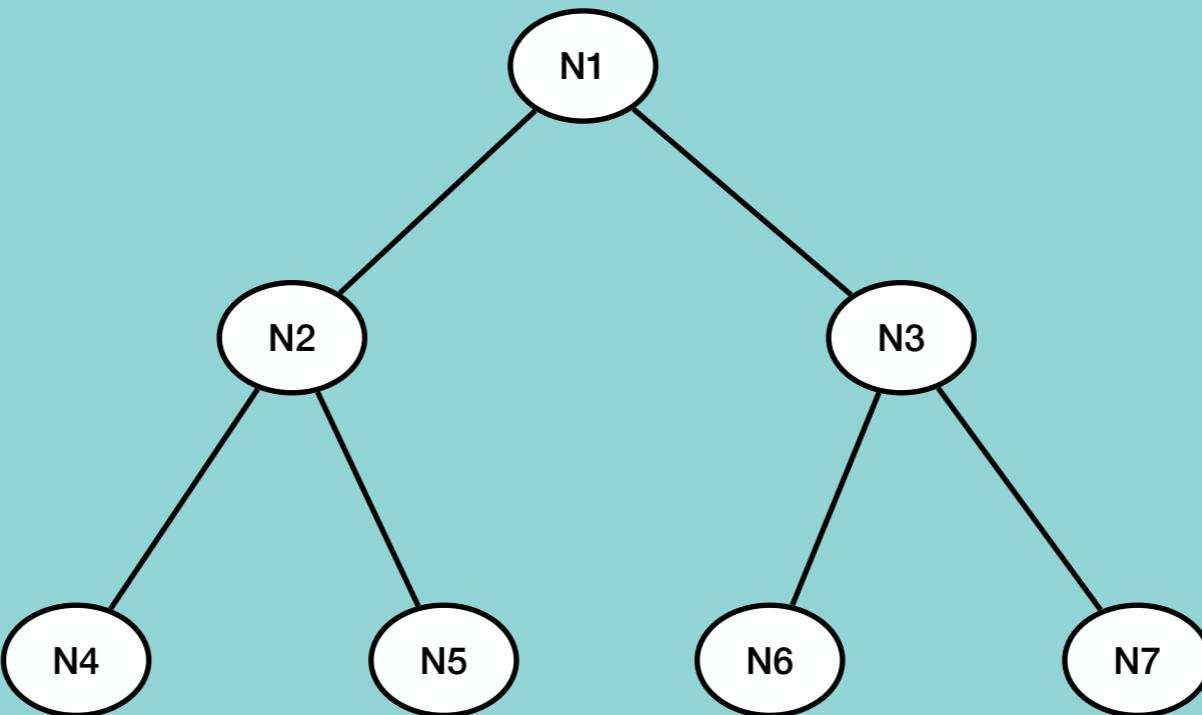
0	1	2	3	4	5	6	7	8
✗	N1	N2	N3	N4	N5			

Left child = `cell[2x]`  $\longrightarrow$  `x = 2, cell[2x2=4]`

Right child = `cell[2x+1]`  $\longrightarrow$  `x = 2, cell[2x2+1=5]`



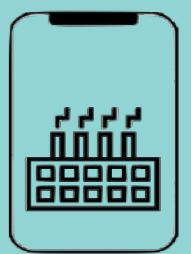
# Create Binary Tree using Python List



0	1	2	3	4	5	6	7	8
✗	N1	N2	N3	N4	N5	N6	N7	

Left child = `cell[2x]`  $\longrightarrow$  `x = 3, cell[2x3=6]`

Right child = `cell[2x+1]`  $\longrightarrow$  `x = 3, cell[2x3+1=7]`

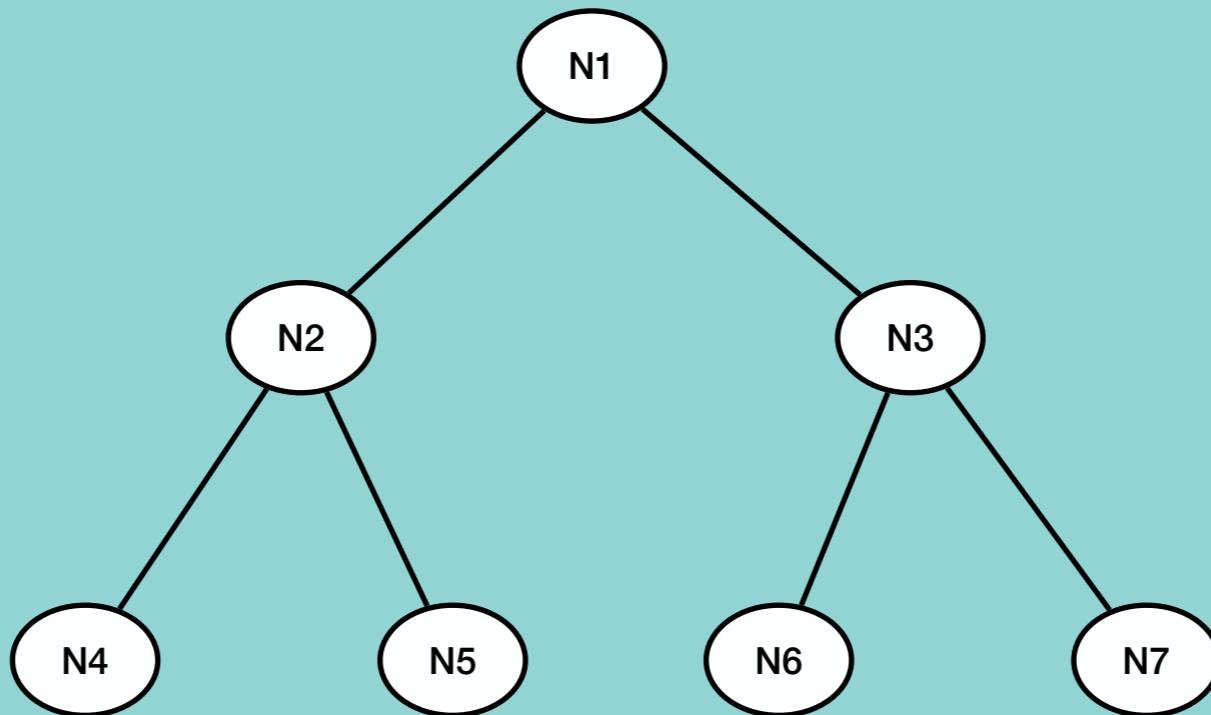


# Create Binary Tree using Python List

`newBT = BTclass()`

Fixed size Python List

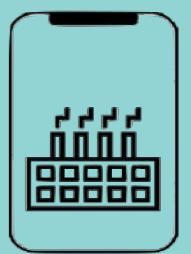
`lastUsedIndex`



0	1	2	3	4	5	6	7	8
✗	N1	N2	N3	N4	N5	N6	N7	

`Left child = cell[2x]` → `x = 3, cell[2x3=6]`

`Right child = cell[2x+1]` → `x = 3, cell[2x3+1=7]`

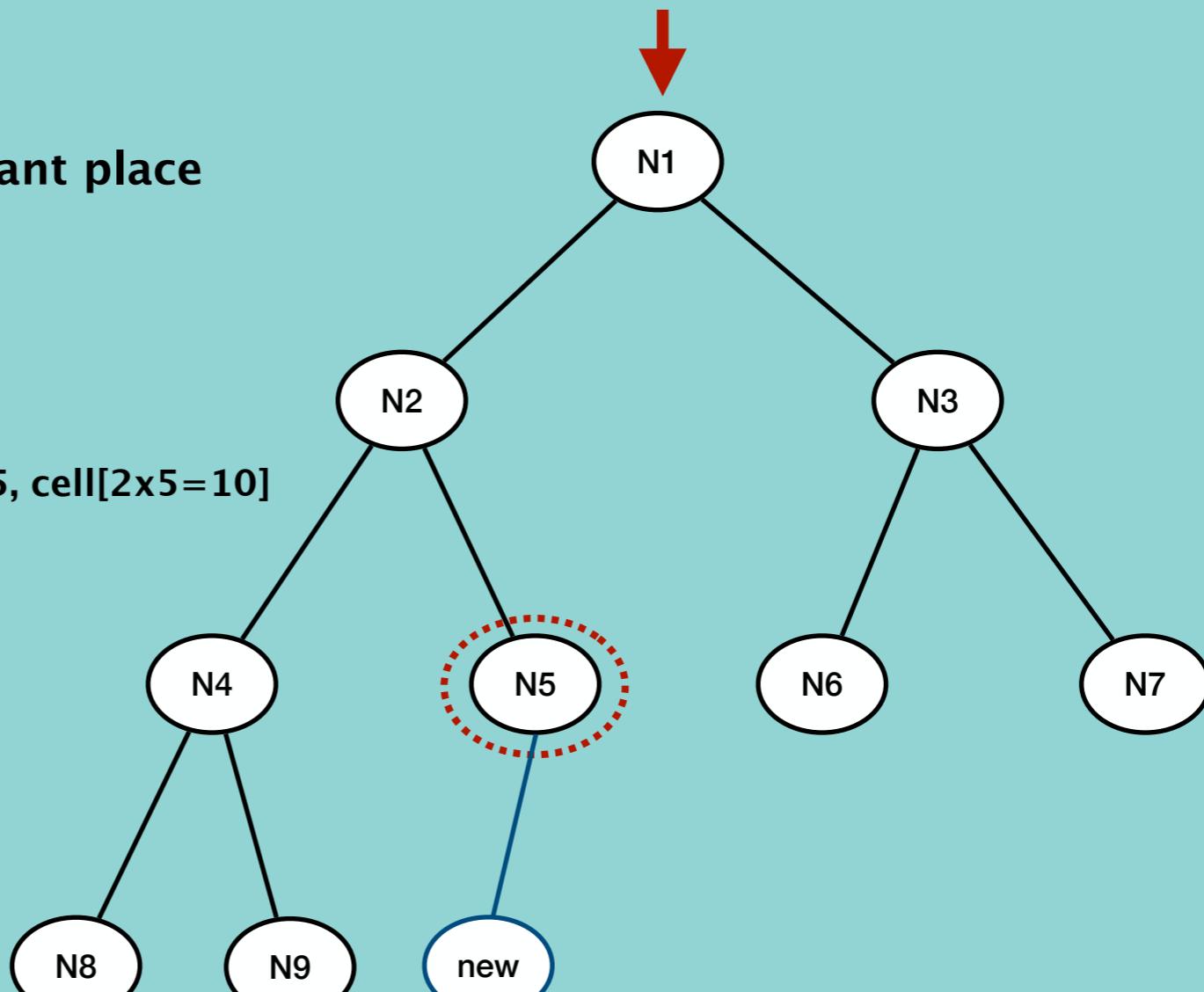


# Insert a node in Binary Tree

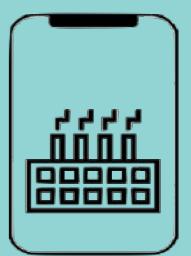
- The Binary Tree is full
- We have to look for a first vacant place

**newNode**

Left child = cell[2x]  $\longrightarrow$   $x = 5, \text{cell}[2 \times 5 = 10]$

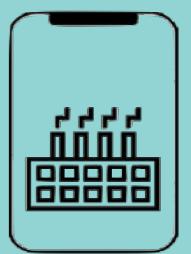
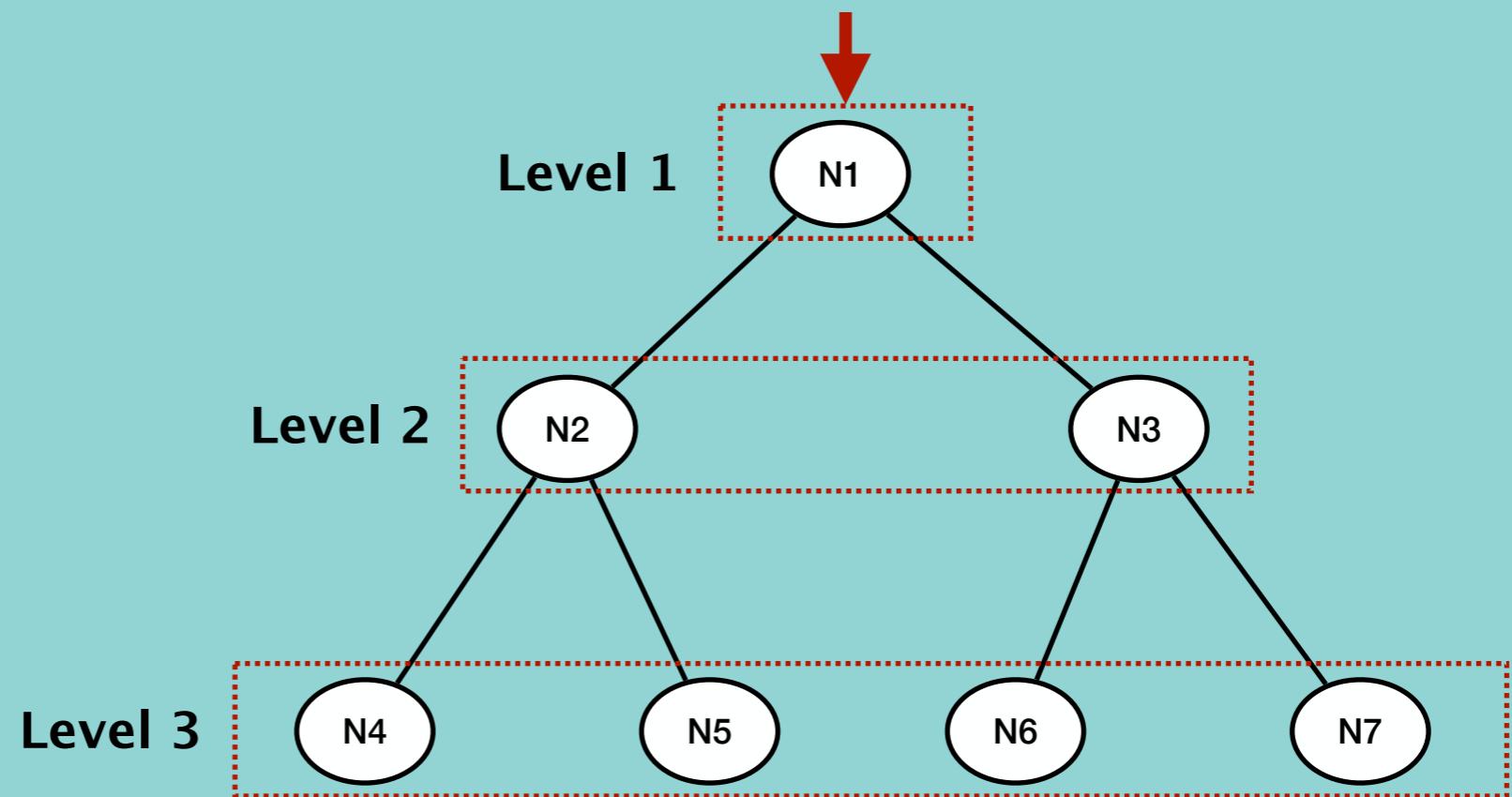


0	1	2	3	4	5	6	7	8	9	10	11
X	N1	N2	N3	N4	N5	N6	N7	N8	N9	New	

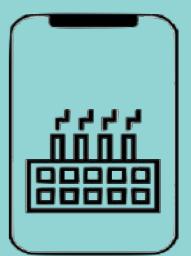
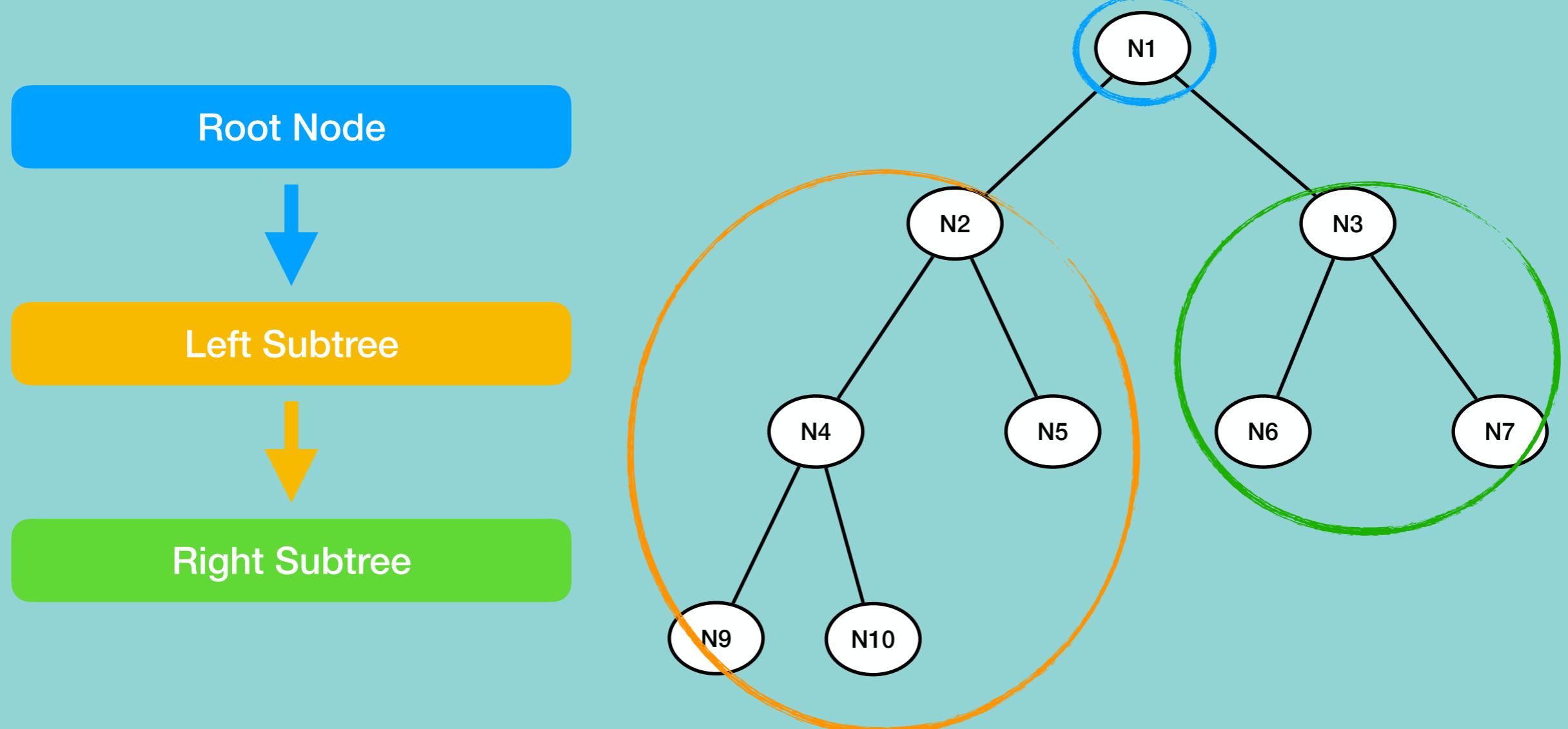


# Searching for a node in Binary Tree (python list)

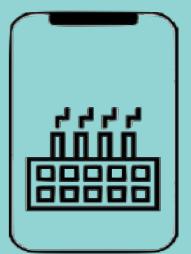
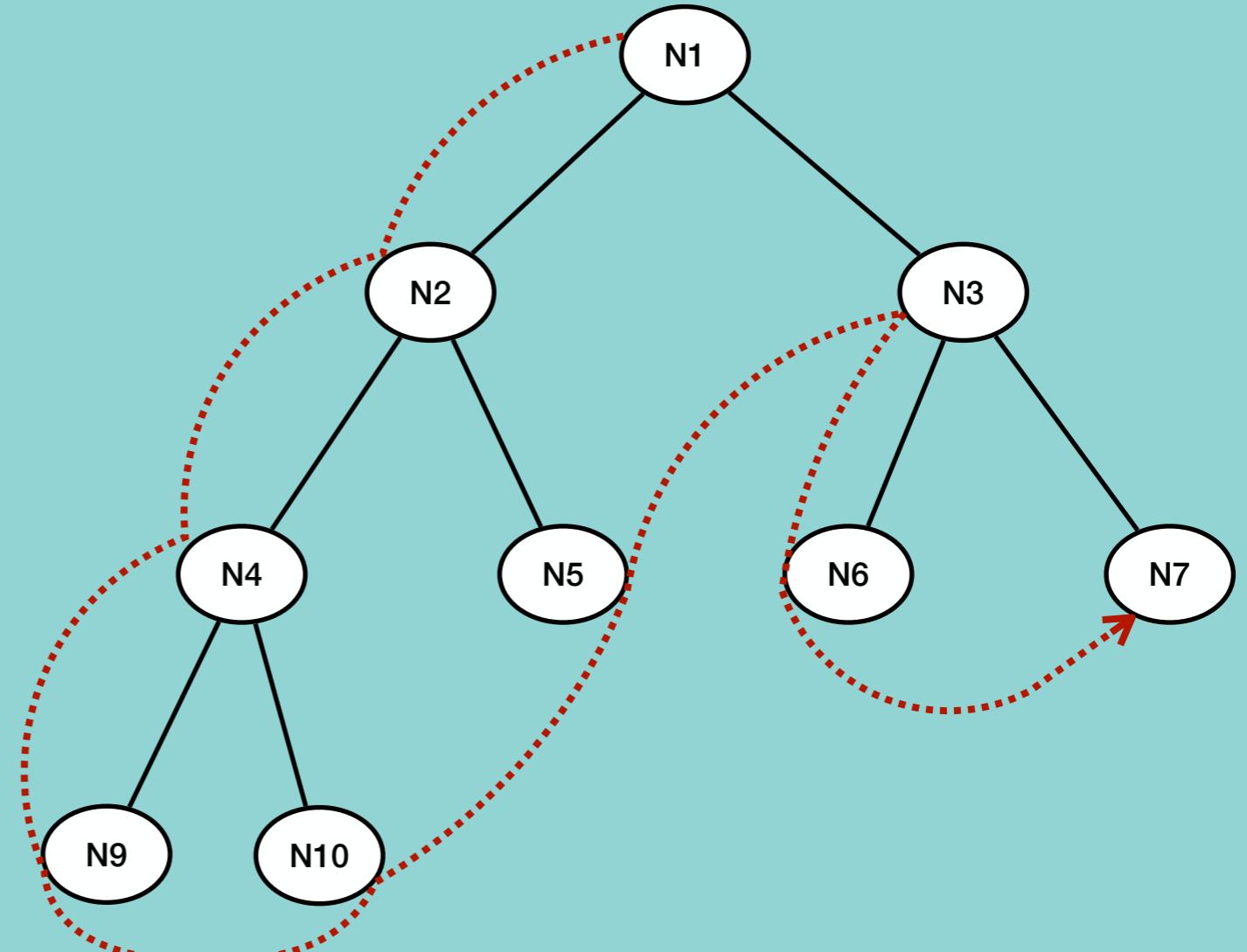
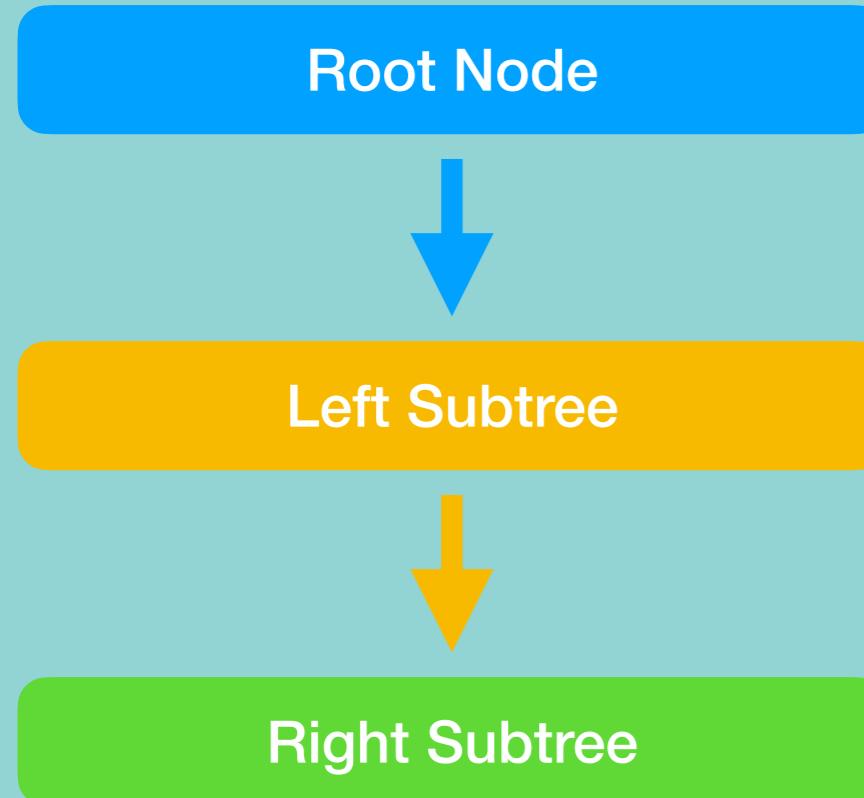
N5 → Success



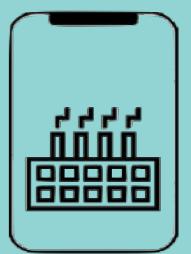
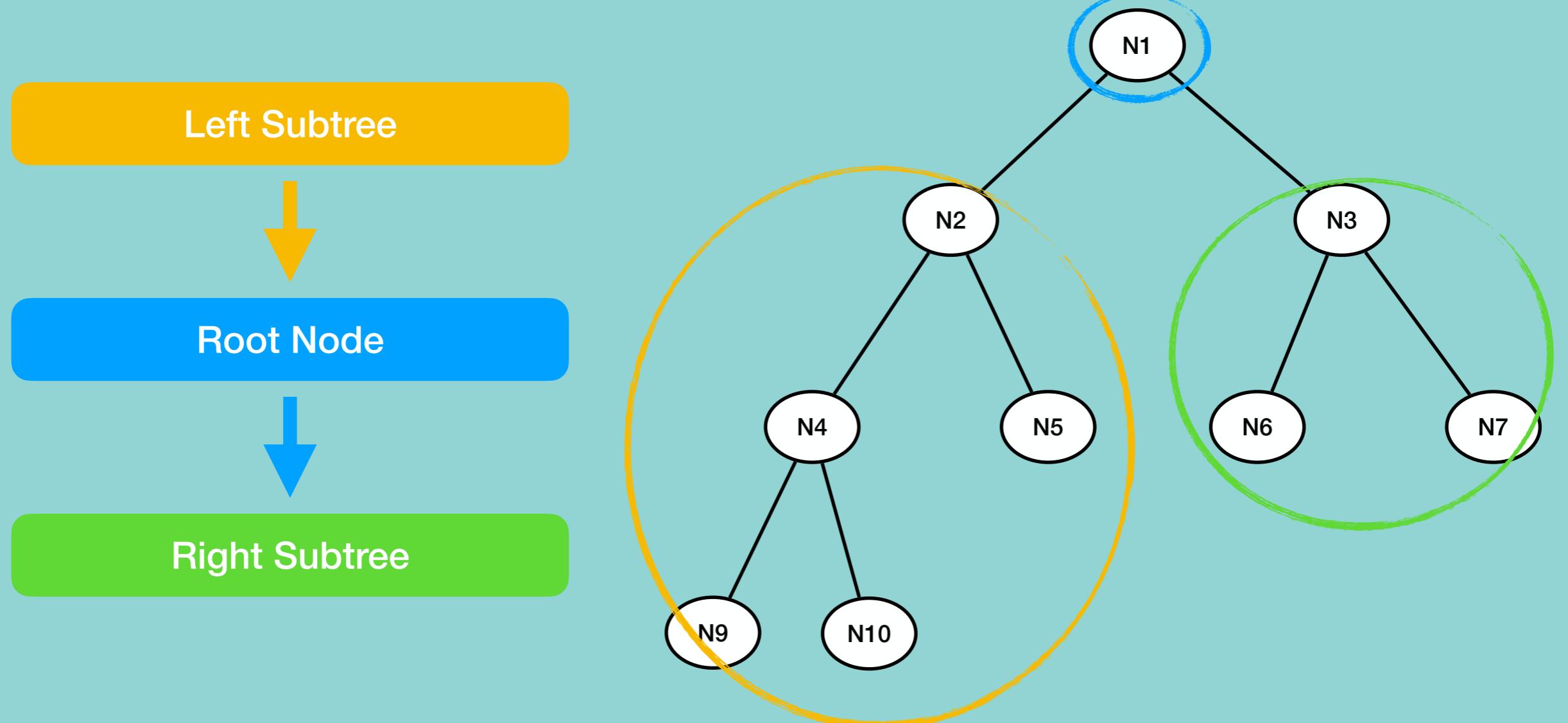
# PreOrder Traversal of Binary Tree (python list)



# PreOrder Traversal of Binary Tree (python list)



# InOrder Traversal of Binary Tree (python list)



# InOrder Traversal of Binary Tree (python list)

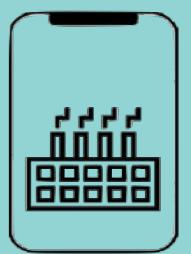
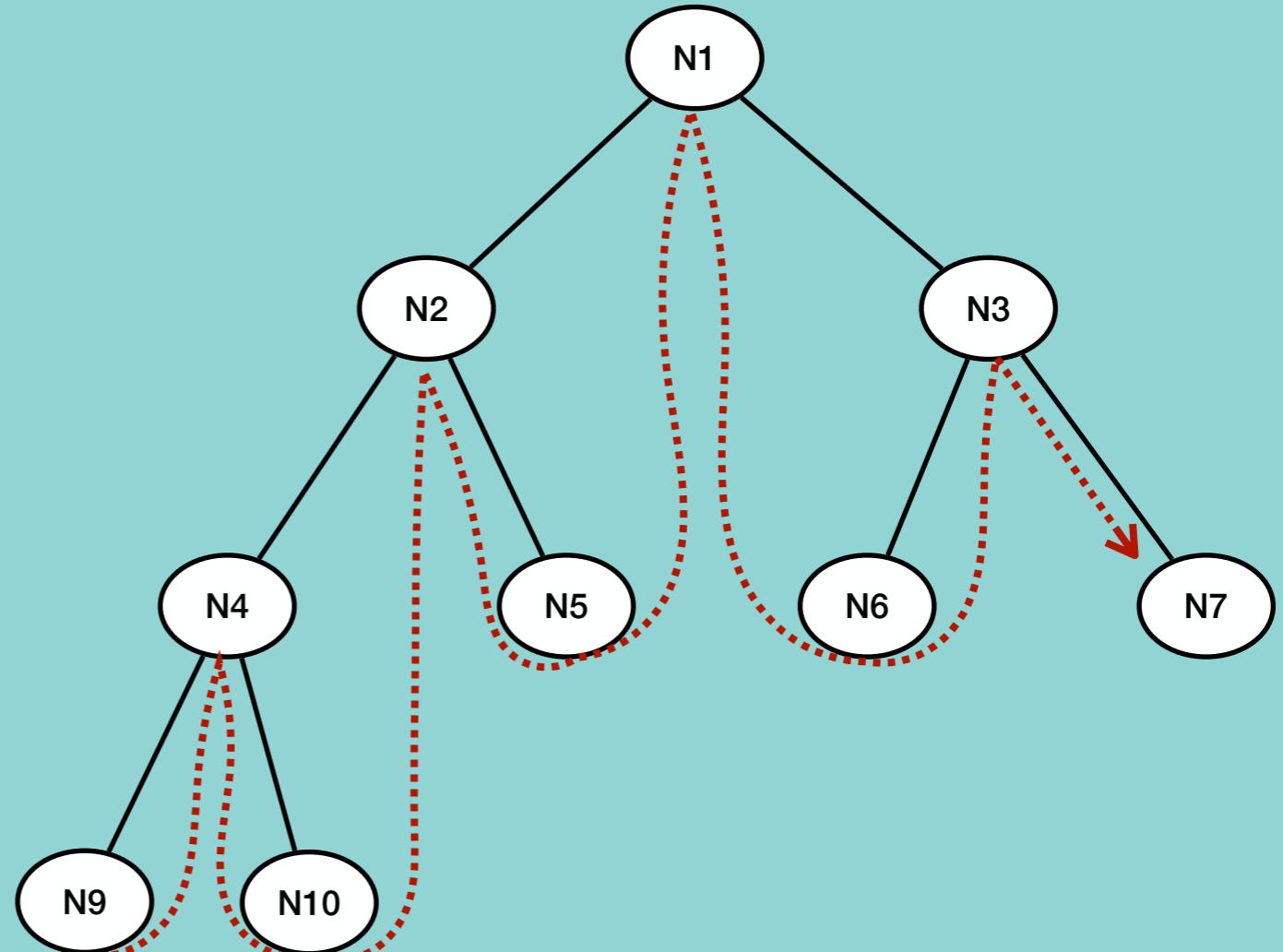
Left Subtree



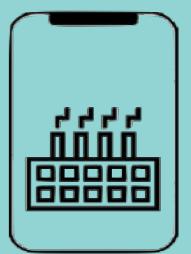
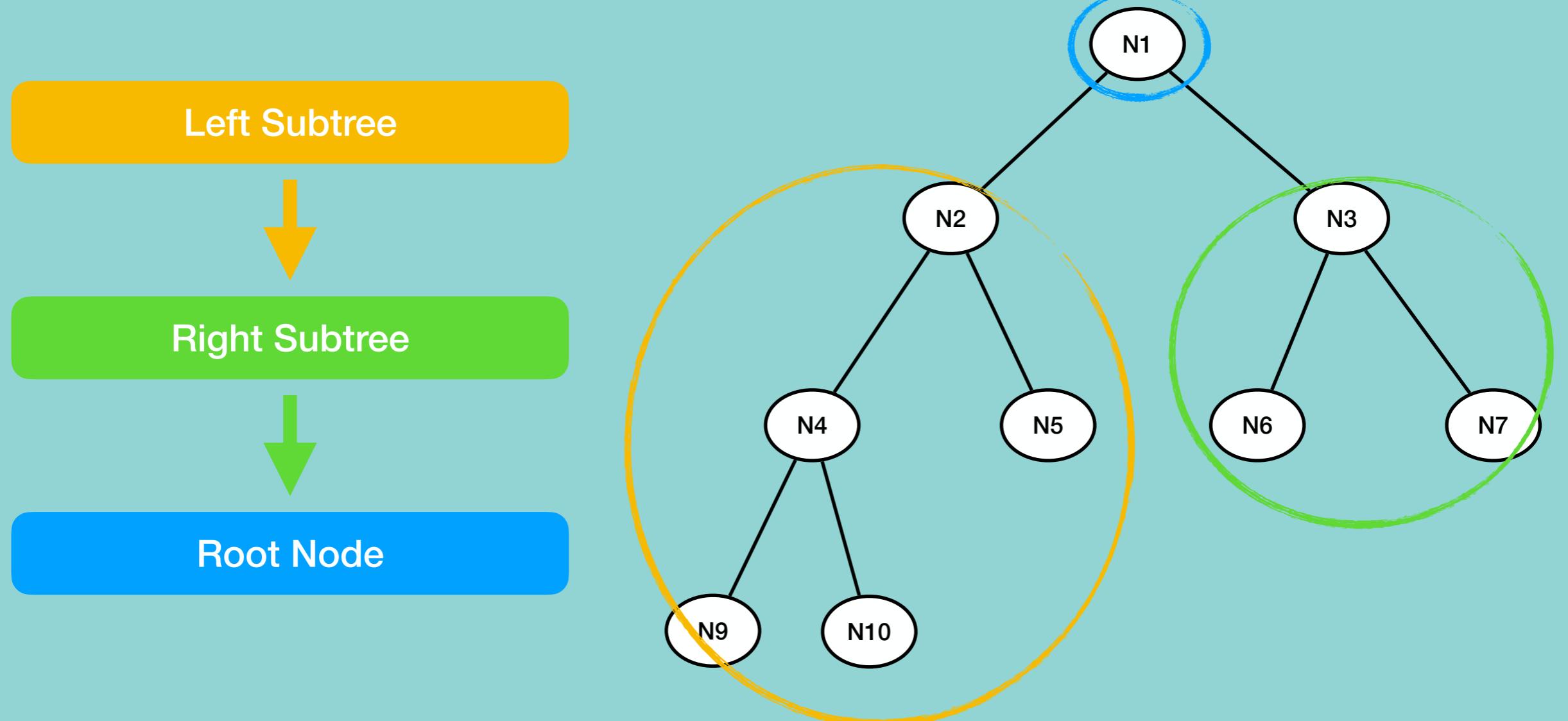
Root Node



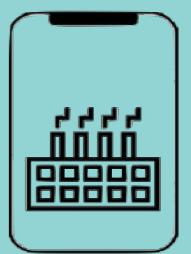
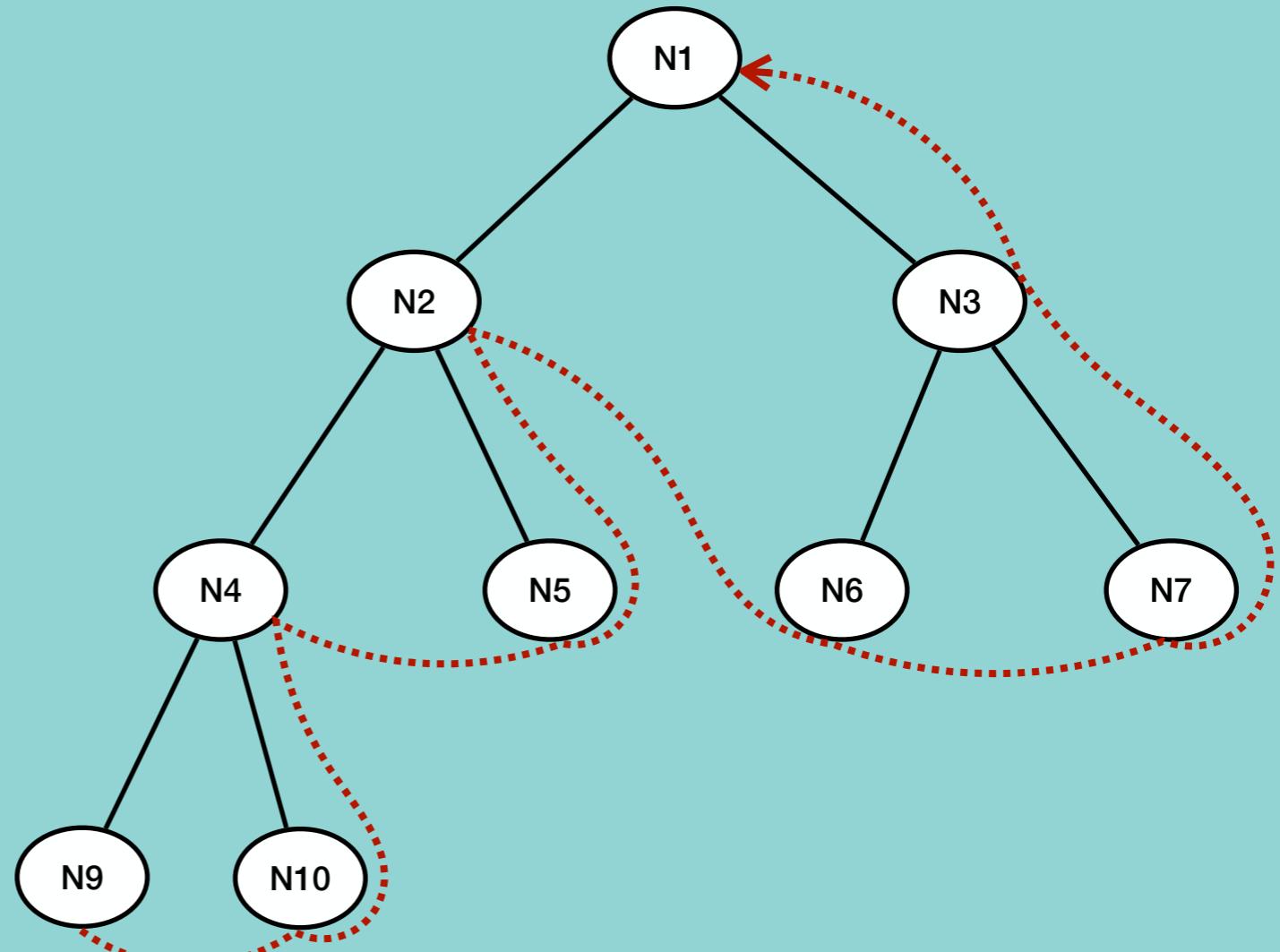
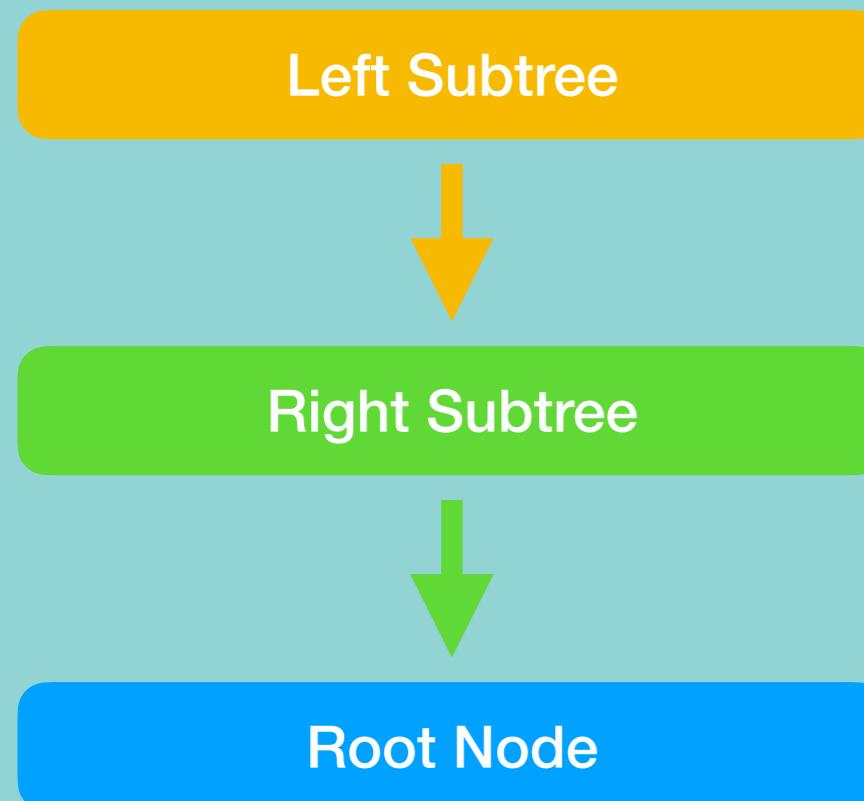
Right Subtree



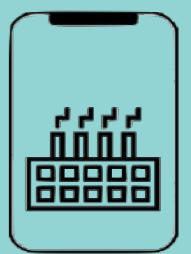
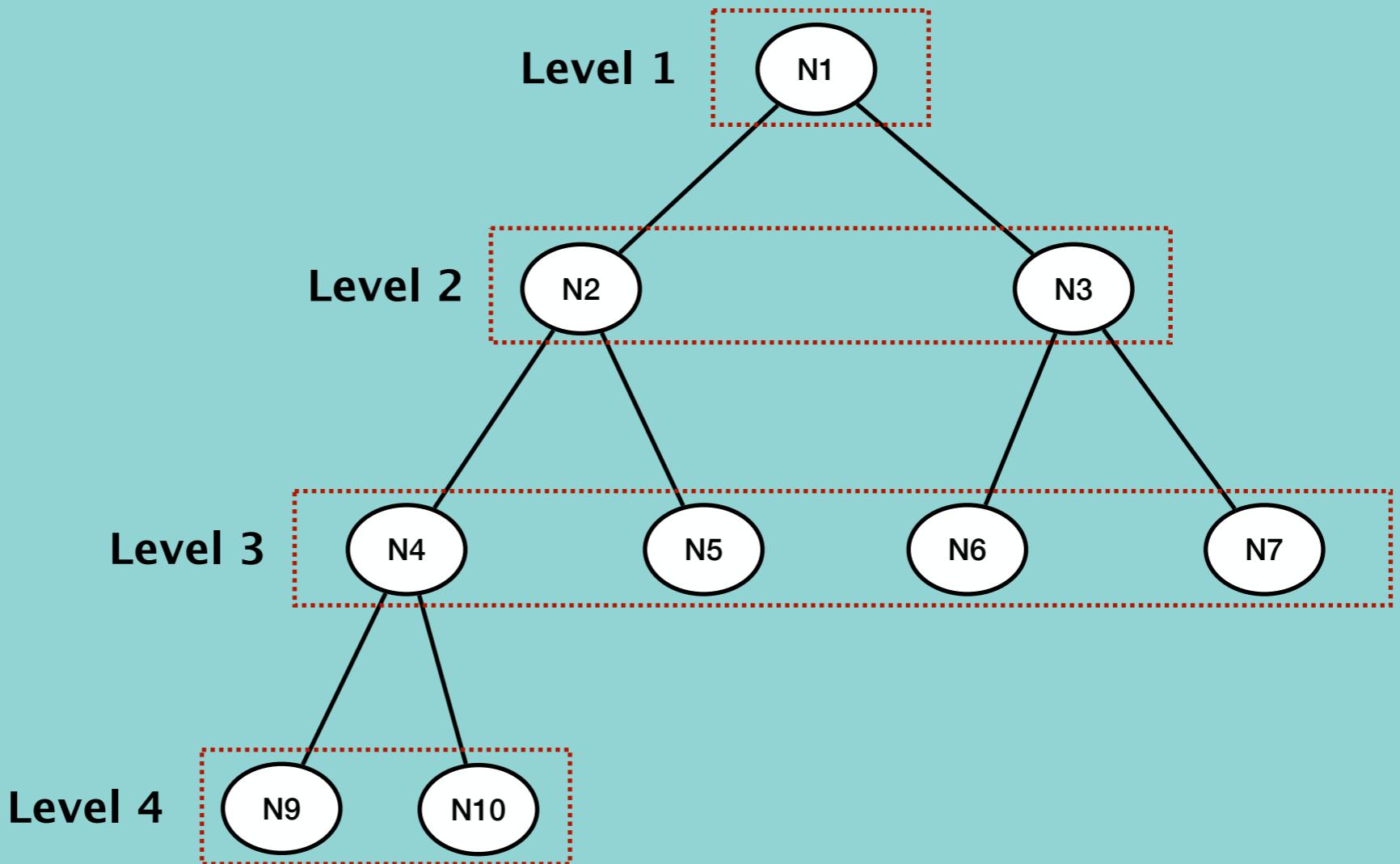
# PostOrder Traversal of Binary Tree (python list)



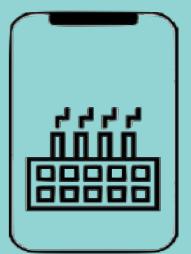
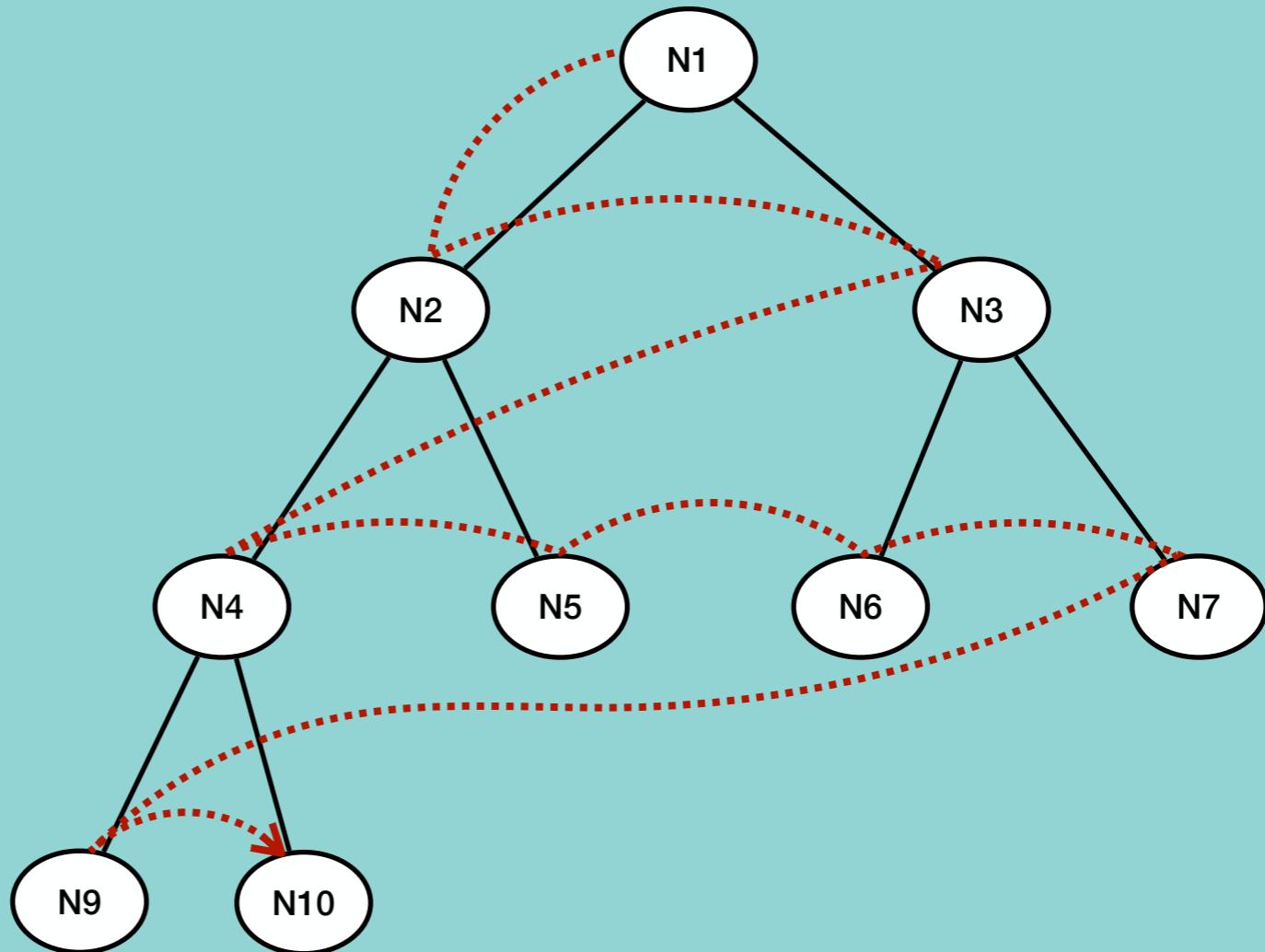
# PostOrder Traversal of Binary Tree (python list)



# LevelOrder Traversal of Binary Tree (python list)



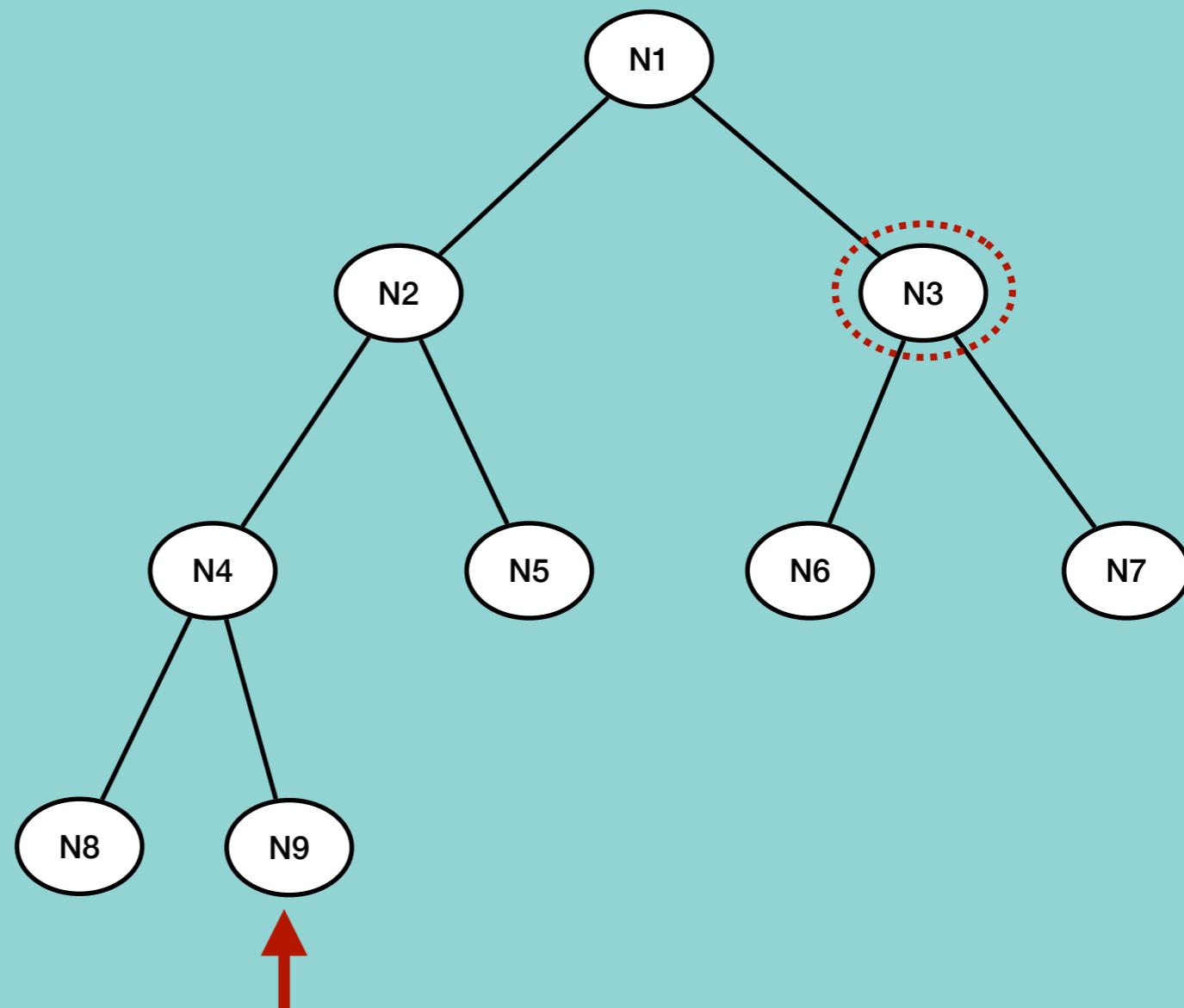
# LevelOrder Traversal of Binary Tree (python list)



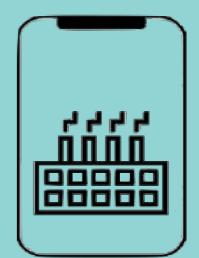
# Delete a node from Binary Tree (python list)

## Level Order Traversal

deepestNode = lastUsedIndex

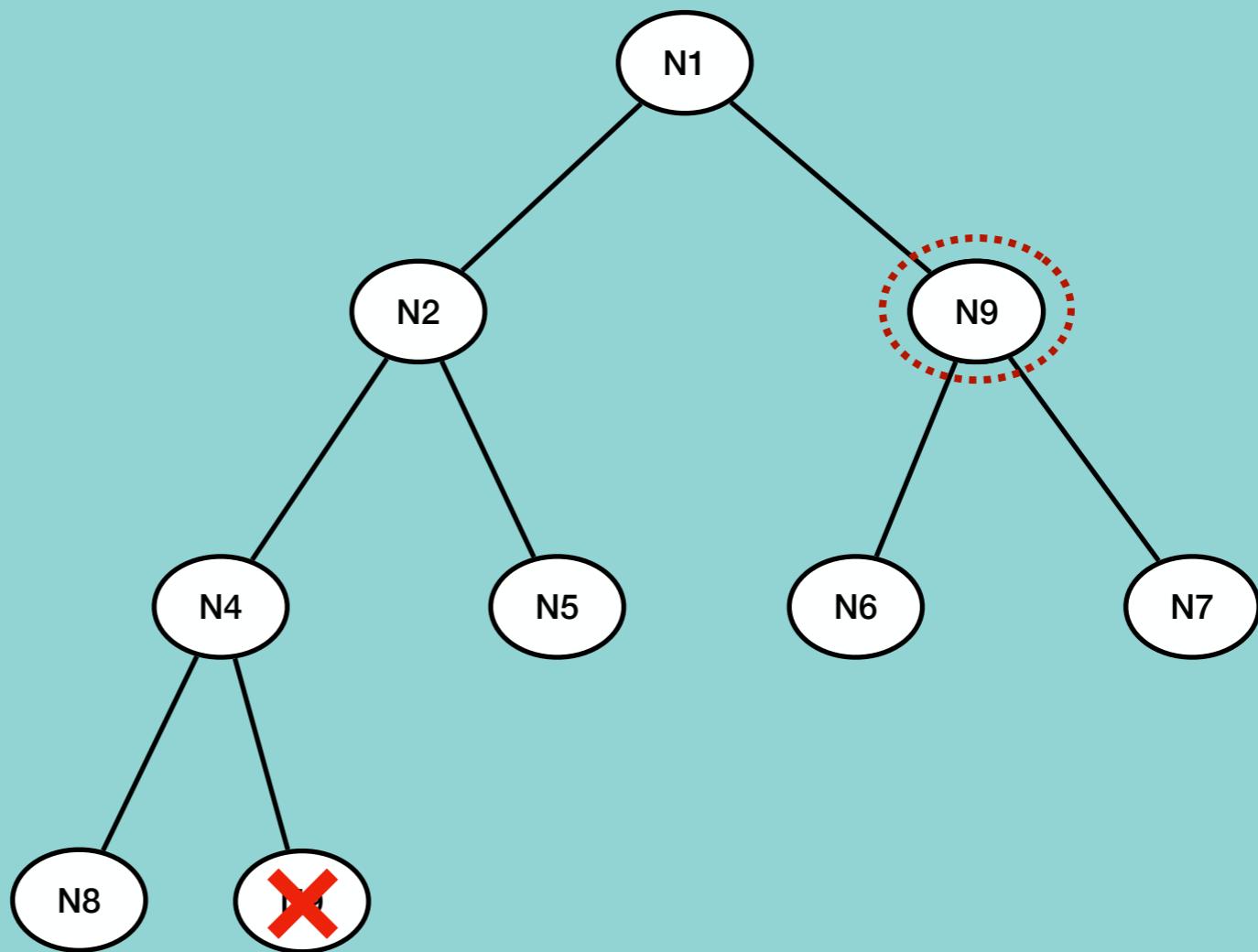


✗	N1	N2	N3	N4	N5	N6	N7	N8	N9		
0	1	2	3	4	5	6	7	8	9	10	11

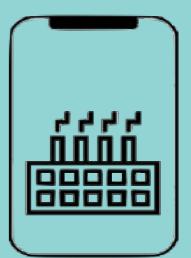


# Delete a node from Binary Tree (python list)

## Level Order Traversal

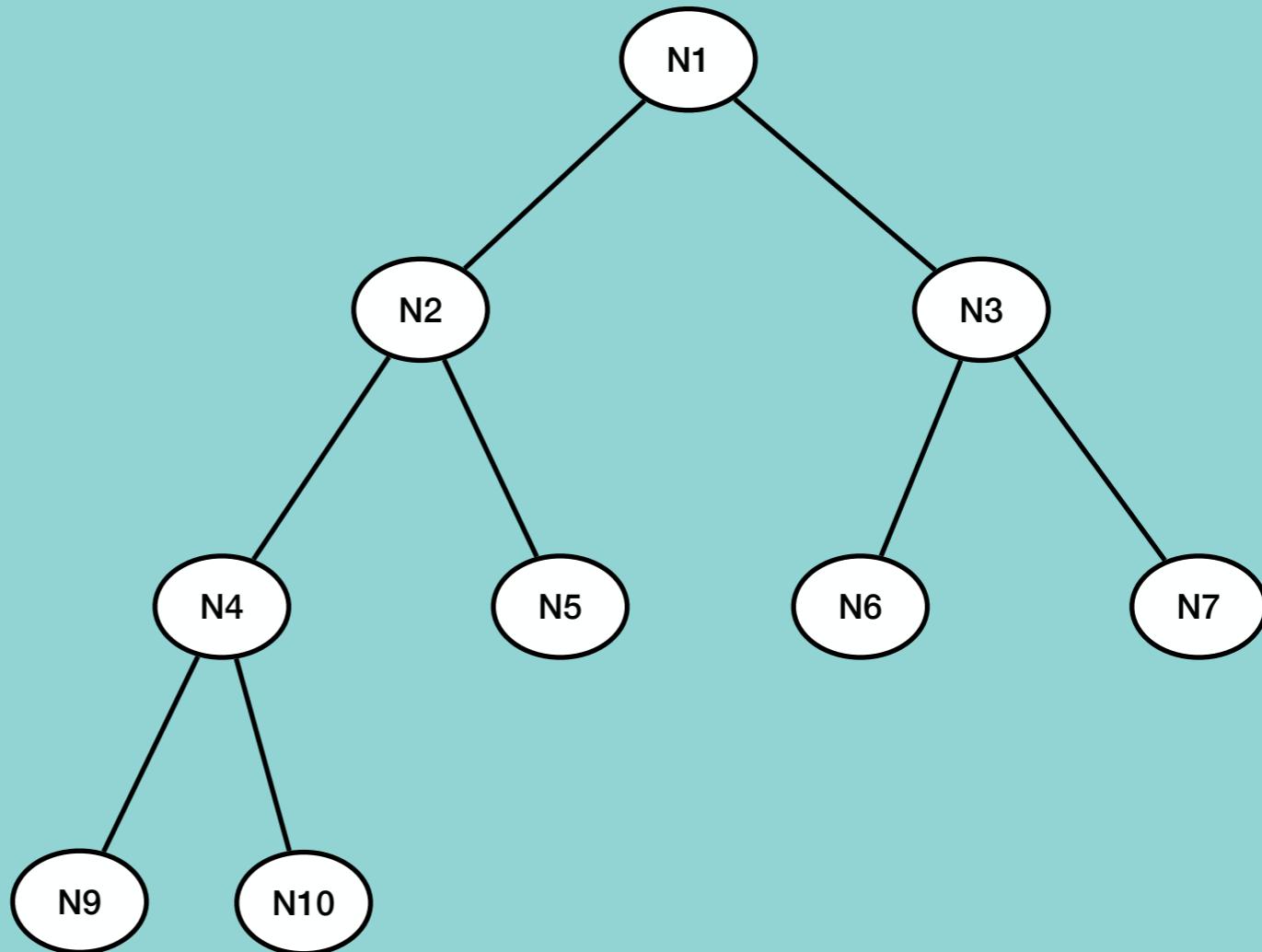


X	N1	N2	N9	N4	N5	N6	N7	N8			
0	1	2	3	4	5	6	7	8	9	10	11

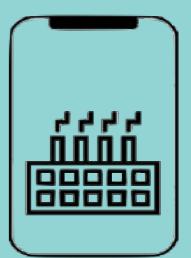


# Delete entire Binary Tree (python list)

**customList = None**

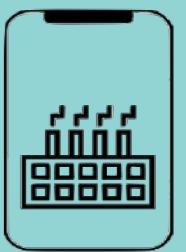


✗	N1	N2	N3	N4	N5	N6	N7	N8	N9		
0	1	2	3	4	5	6	7	8	9	10	11



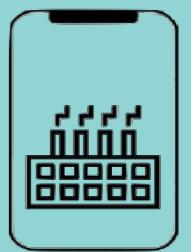
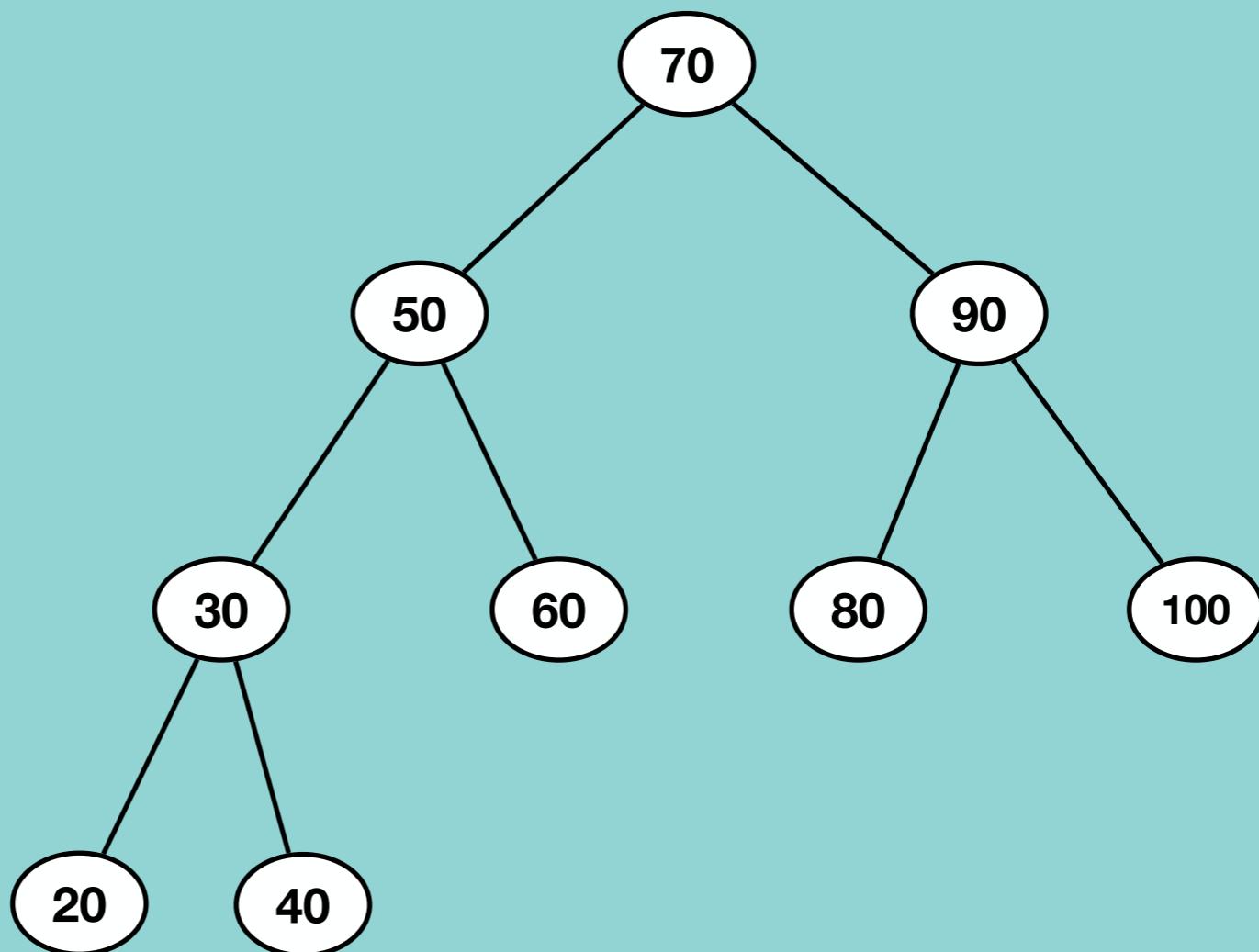
# Binary Tree (Python List vs Linked List)

	Python List with capacity		Linked List	
	Time complexity	Space complexity	Time complexity	Space complexity
Create Binary Tree	O(1)	O(n)	O(1)	O(1)
Insert a node to Binary Tree	O(1)	O(1)	O(n)	O(n)
Delete a node from Binary Tree	O(n)	O(1)	O(n)	O(n)
Search for a node in Binary Tree	O(n)	O(1)	O(n)	O(n)
Traverse Binary Tree	O(n)	O(1)	O(n)	O(n)
Delete entire Binary Tree	O(1)	O(1)	O(1)	O(1)
Space efficient?		No		Yes



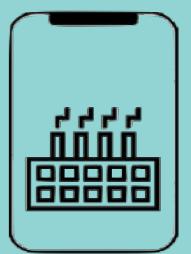
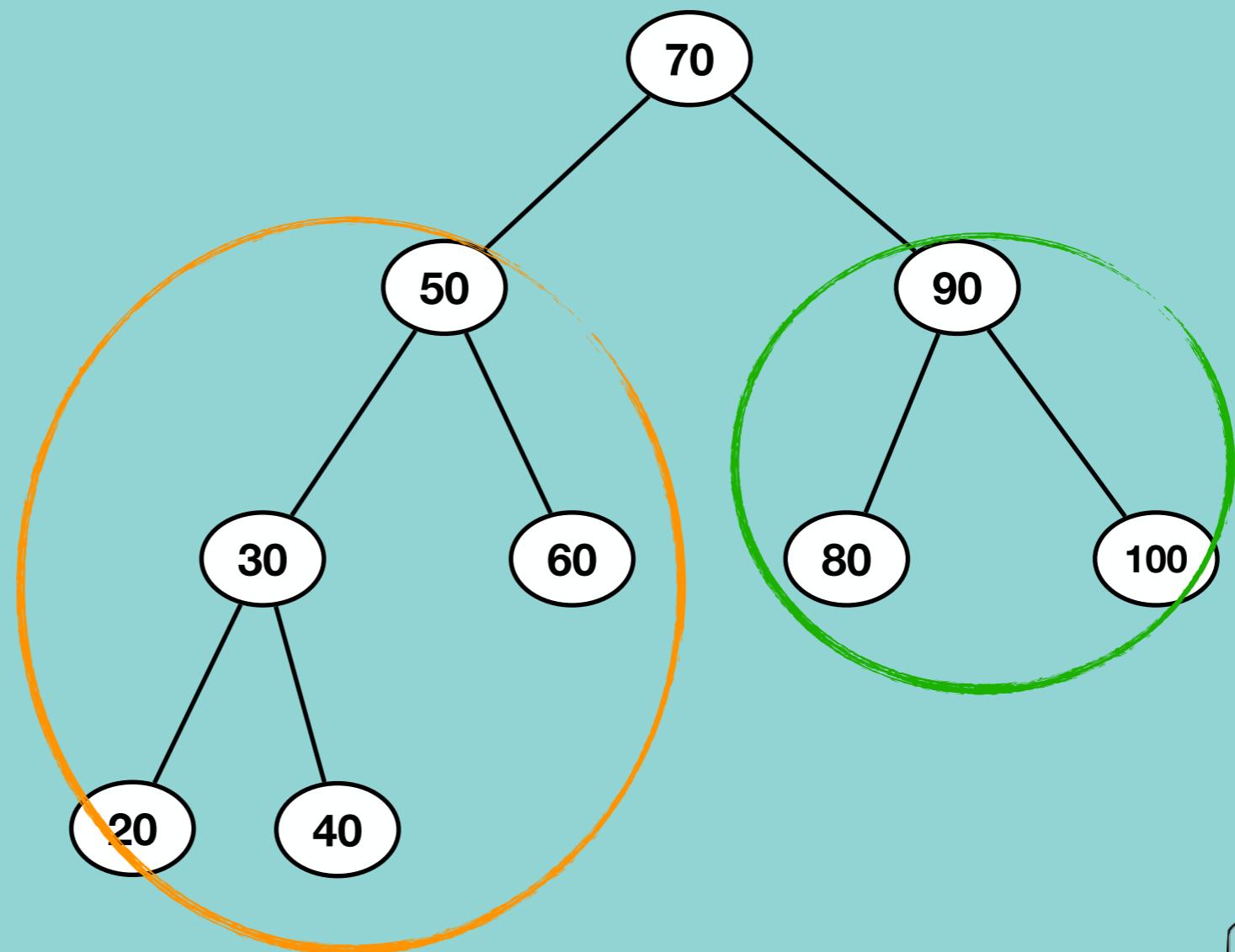
# What is a Binary Search Tree?

- In the left subtree the value of a node is less than or equal to its parent node's value.
- In the right subtree the value of a node is greater than its parent node's value



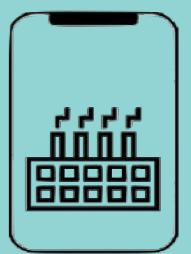
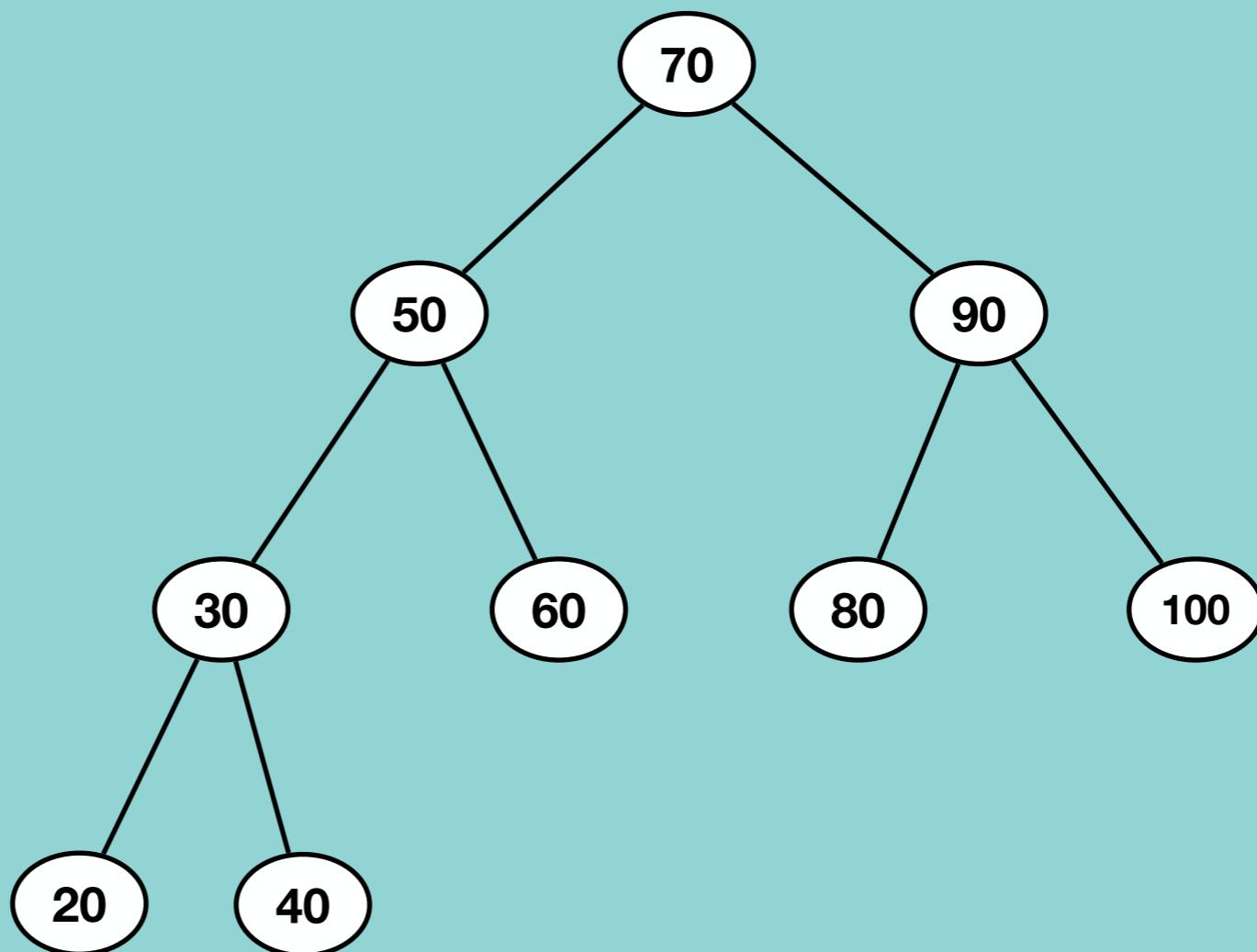
# What is a Binary Search Tree?

- In the left subtree the value of a node is less than or equal to its parent node's value.
- In the right subtree the value of a node is greater than its parent node's value



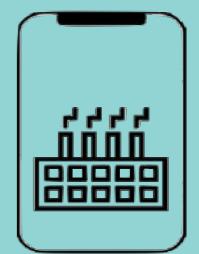
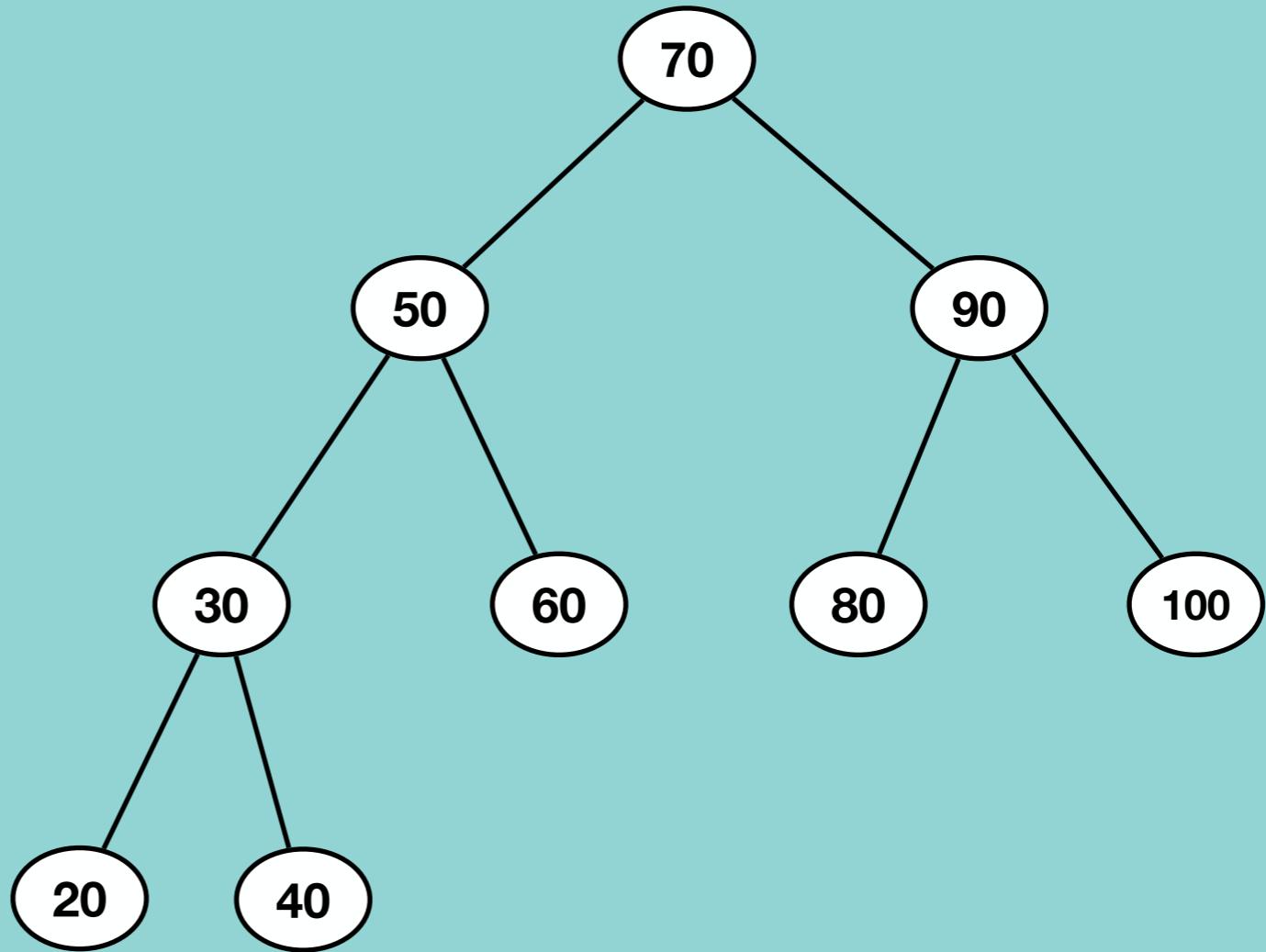
# Why Binary Search Tree?

- It performs faster than Binary Tree when inserting and deleting nodes



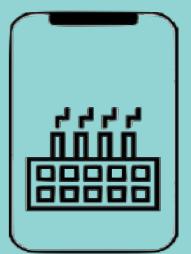
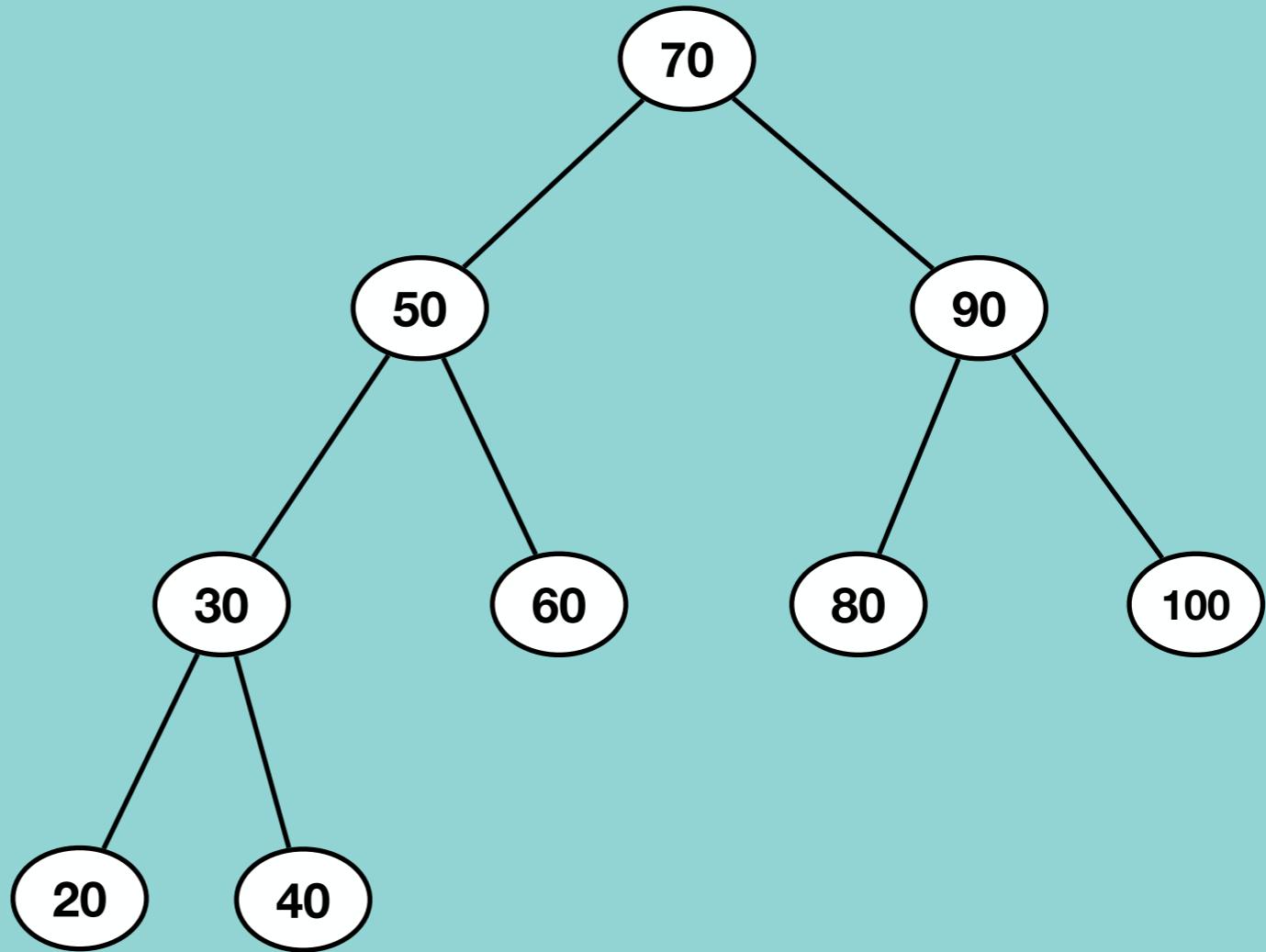
# Common Operations on Binary Search Tree

- Creation of Tree
- Insertion of a node
- Deletion of a node
- Search for a value
- Traverse all nodes
- Deletion of tree

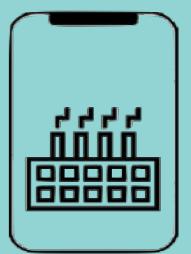
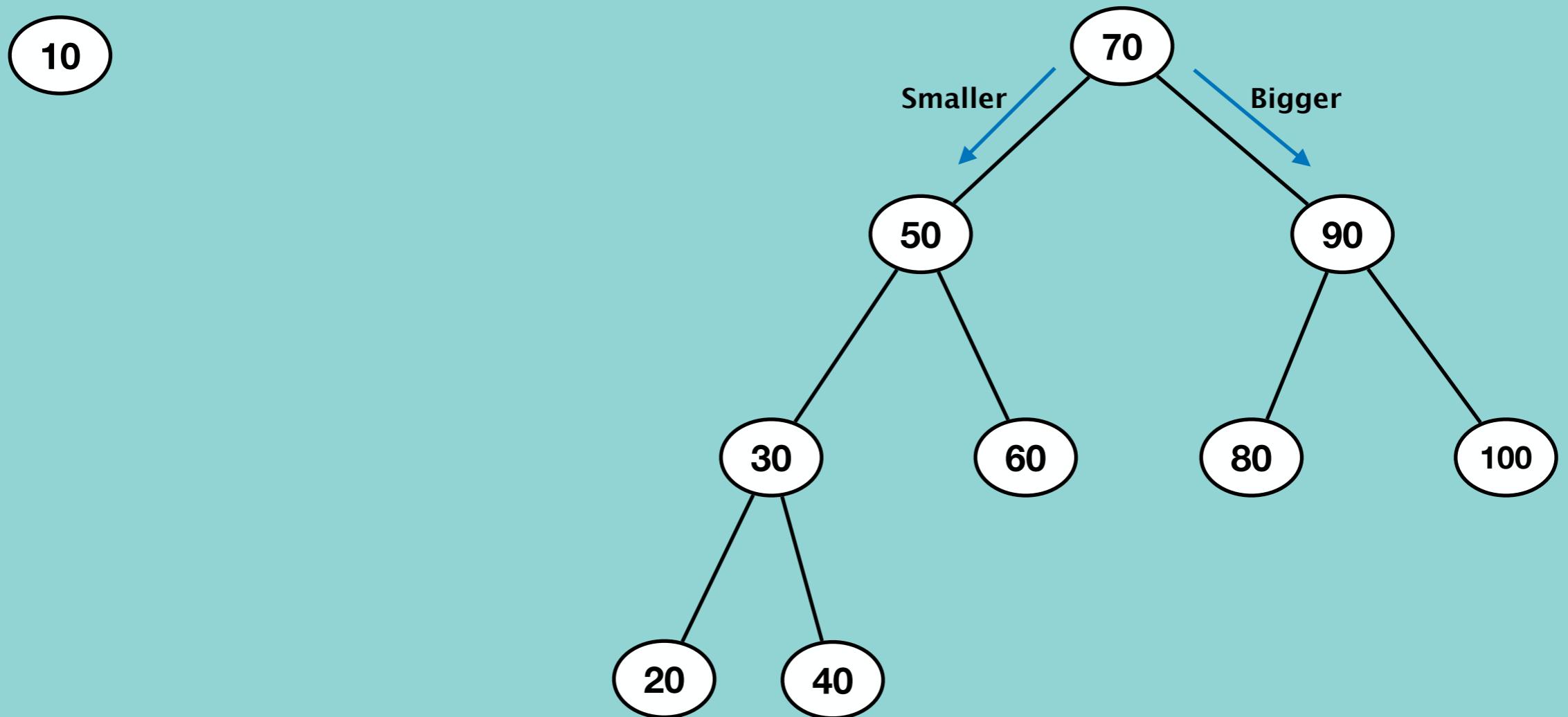


# Create Binary Search Tree

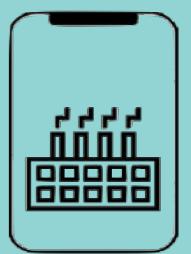
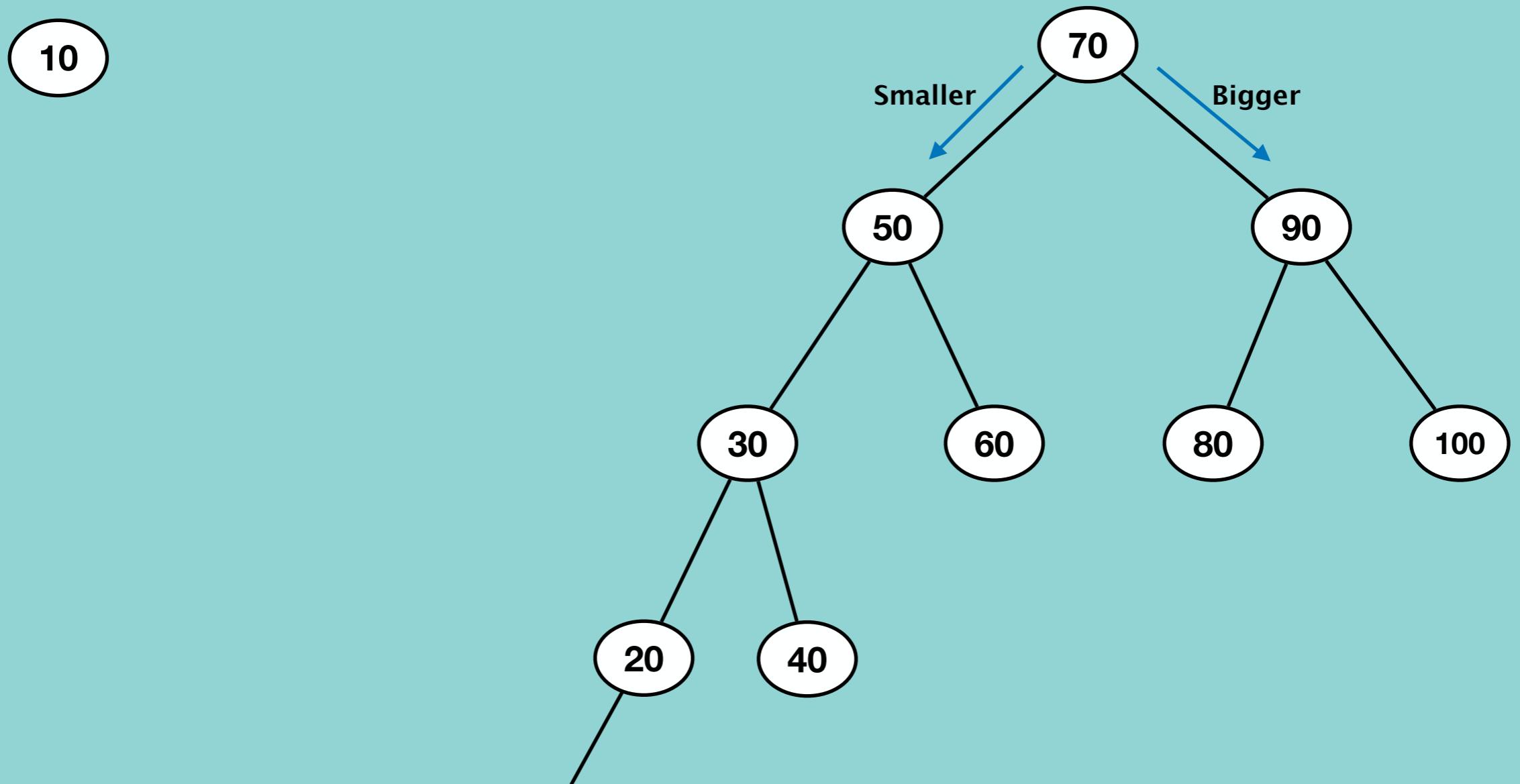
```
newTree = BST()
```



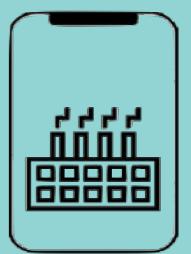
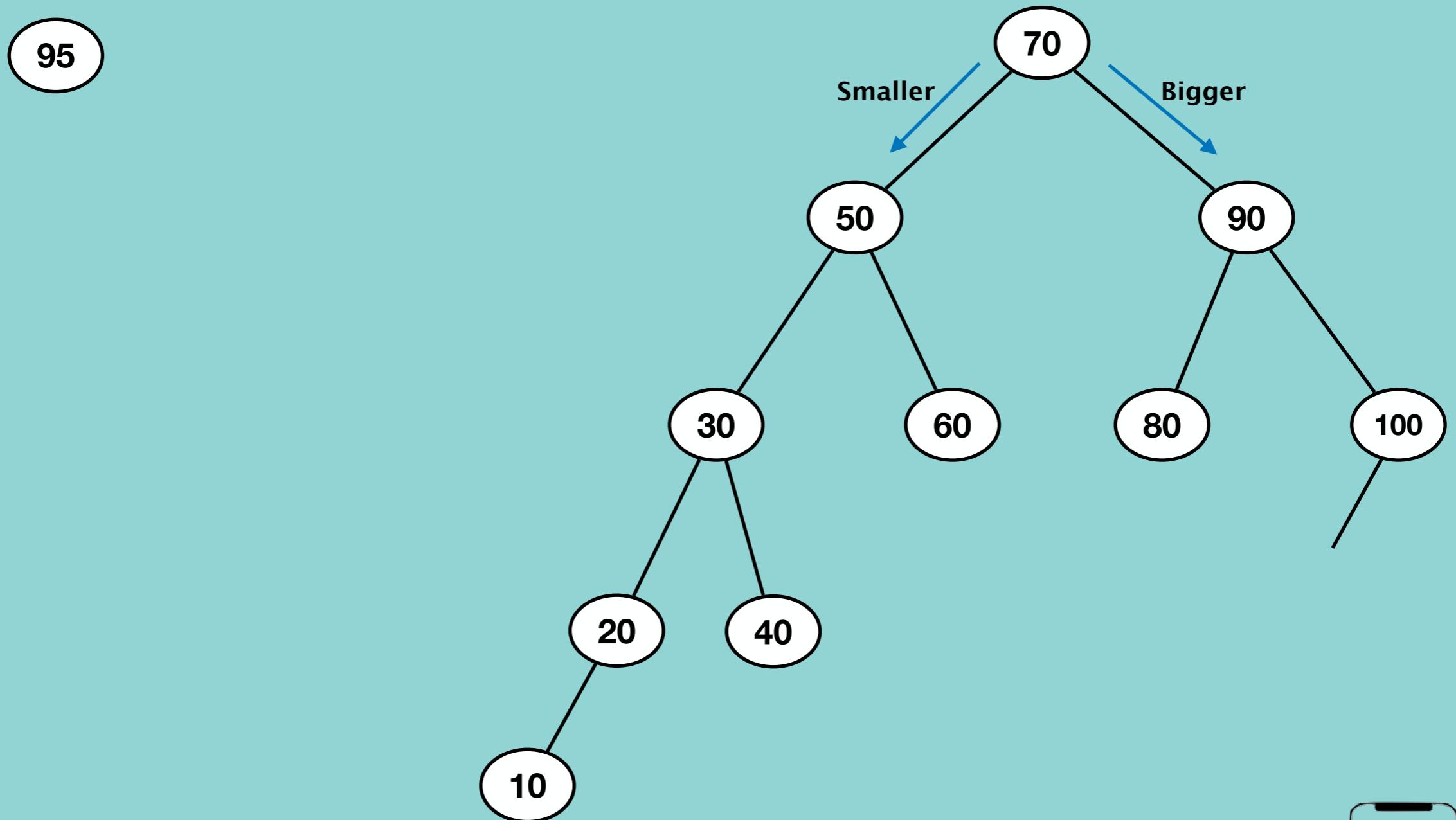
# Insert a node to Binary Search Tree



# Insert a node to Binary Search Tree



# Insert a node to Binary Search Tree



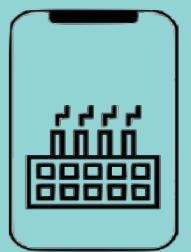
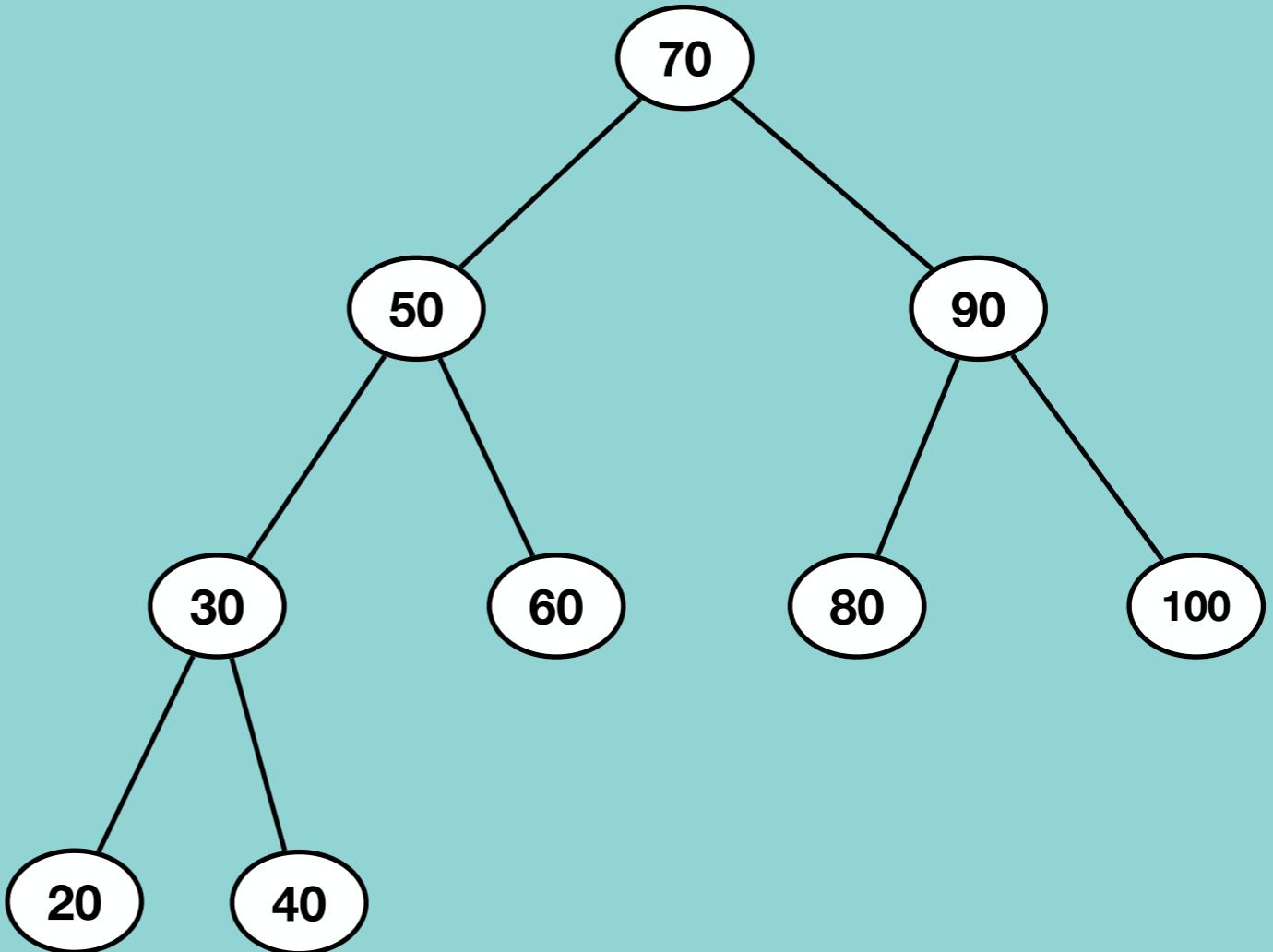
# Traversal of Binary Search Tree

## Depth first search

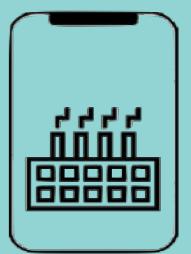
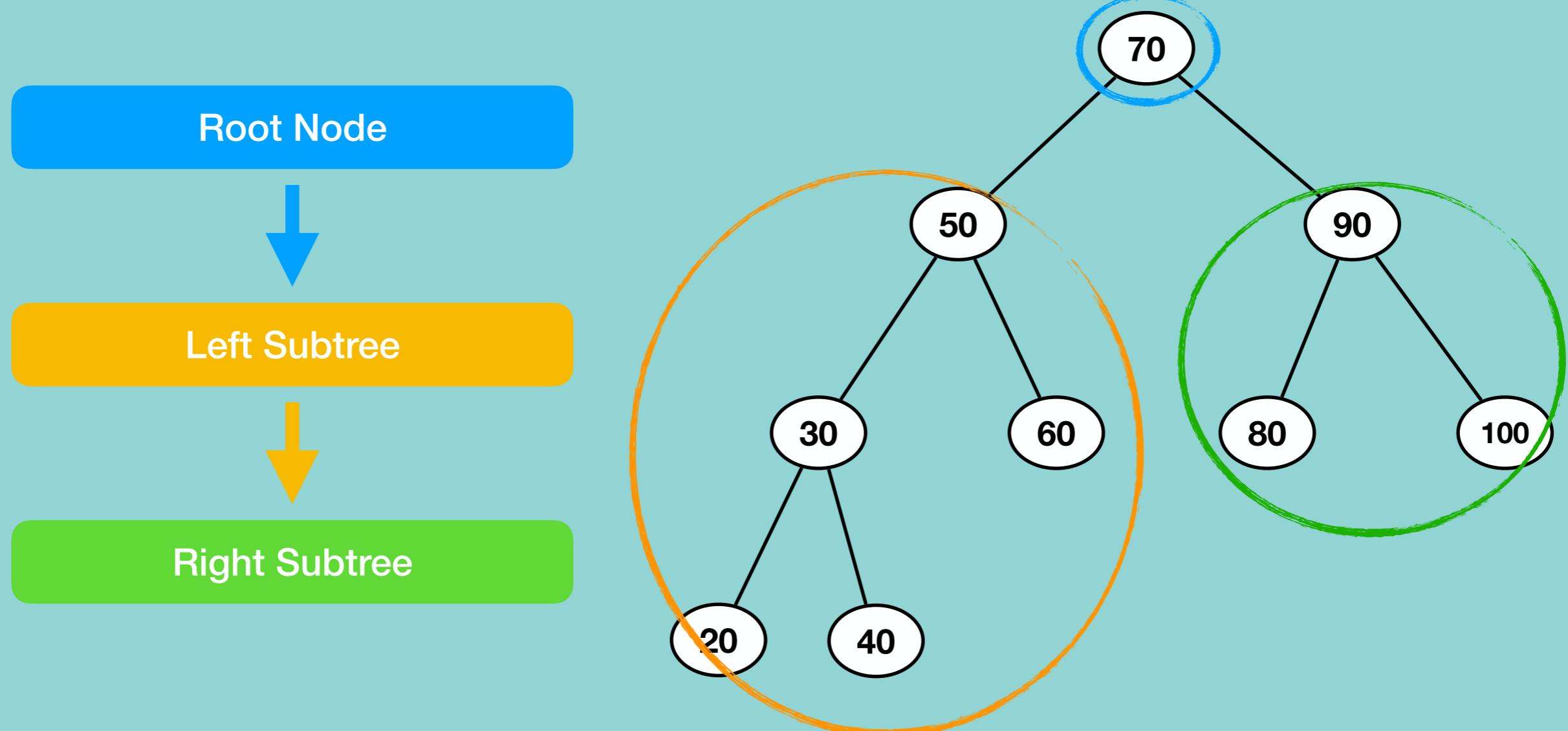
- Preorder traversal
- Inorder traversal
- Post order traversal

## Breadth first search

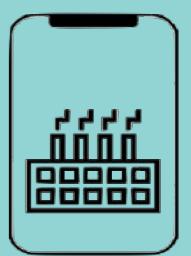
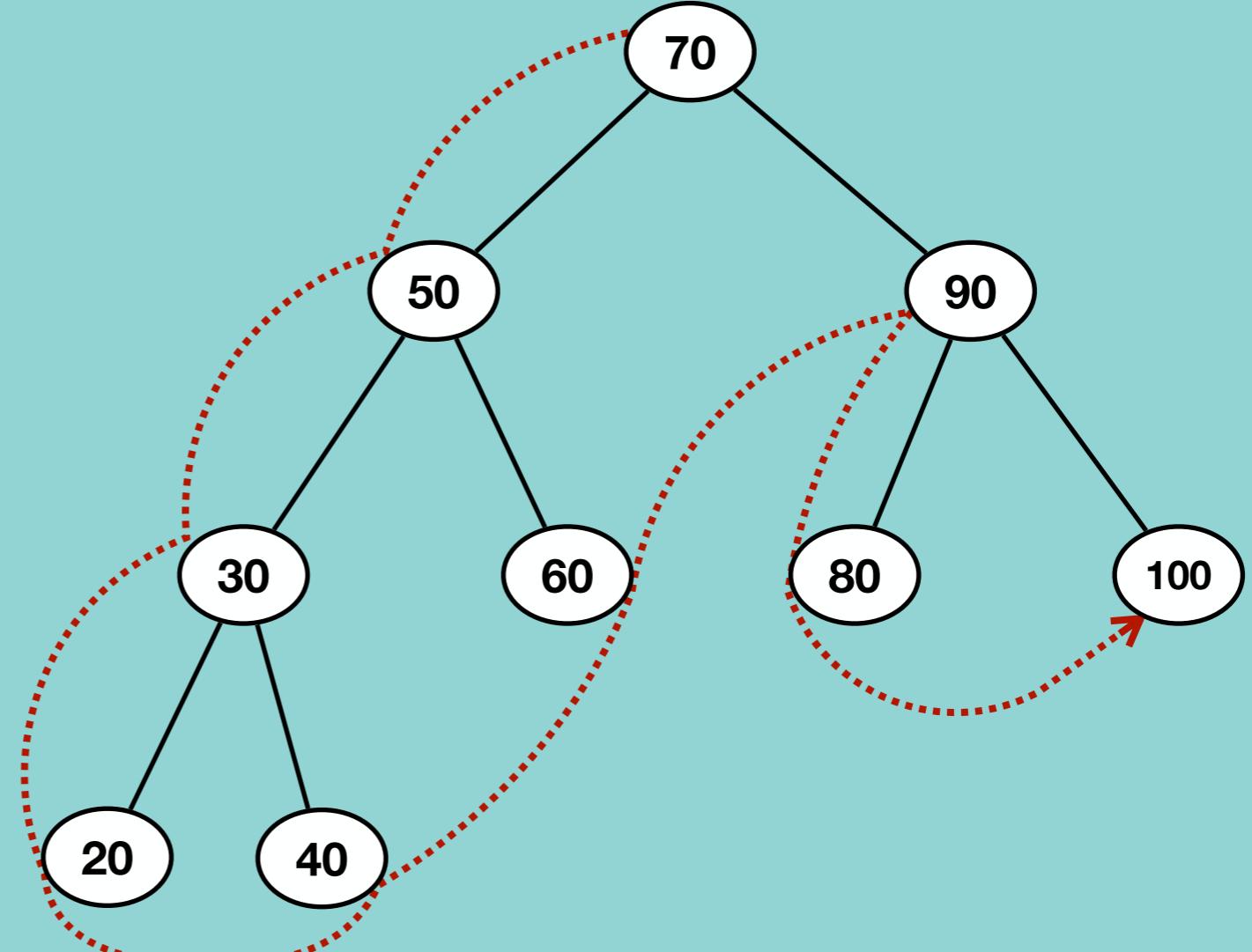
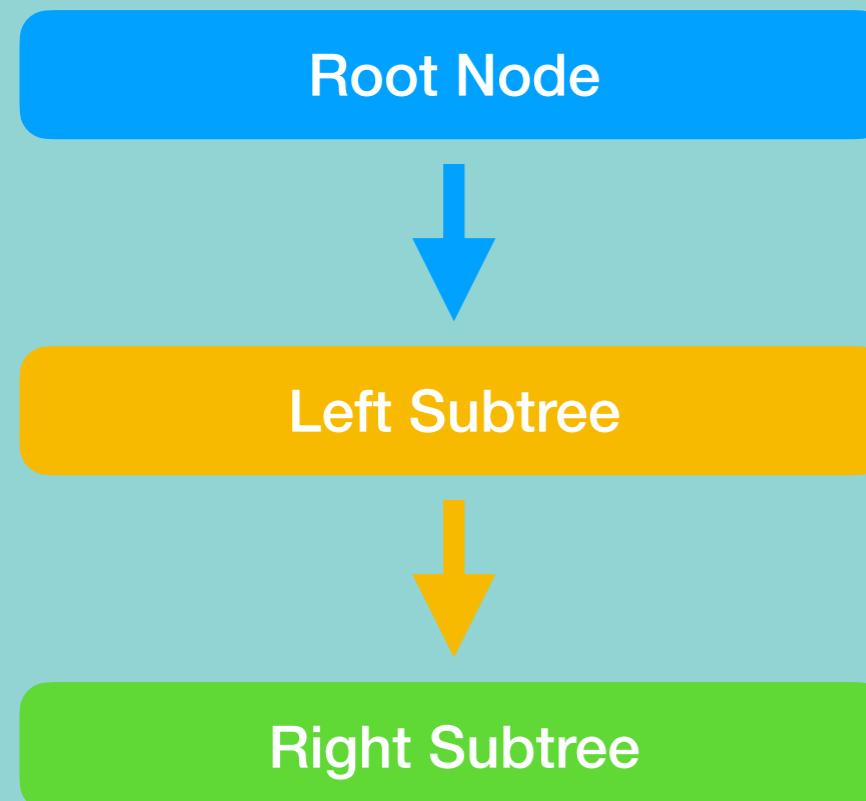
- Level order traversal



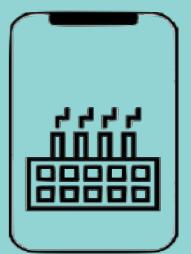
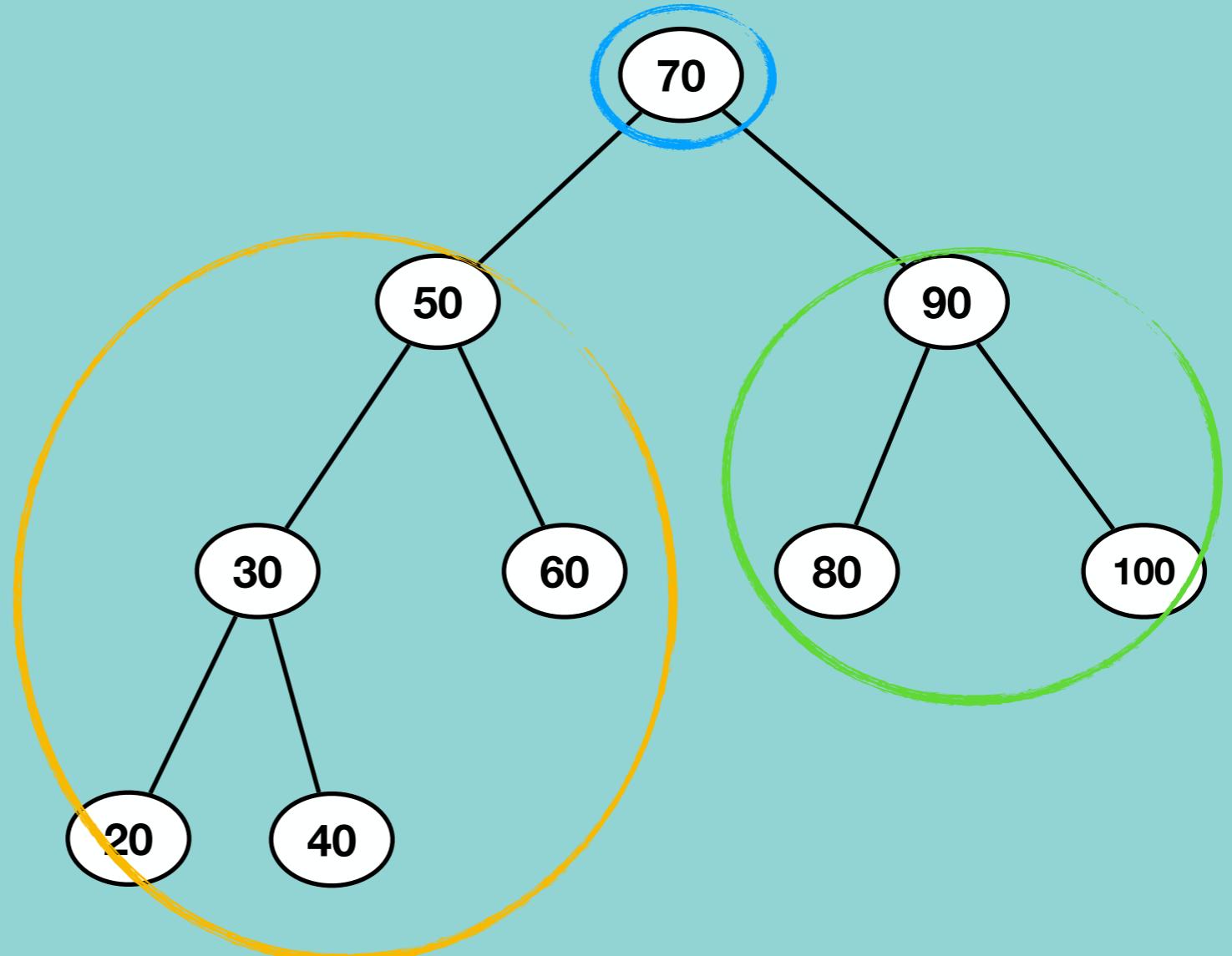
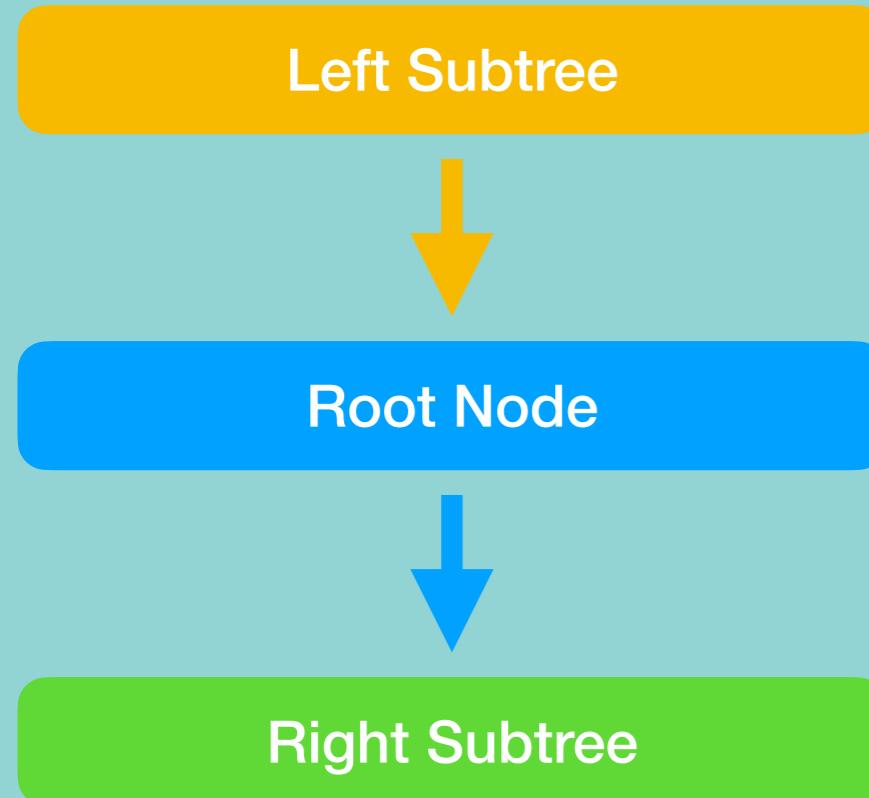
# PreOrder Traversal of Binary Search Tree



# PreOrder Traversal of Binary Search Tree



# InOrder Traversal of Binary Search Tree



# InOrder Traversal of Binary Search Tree

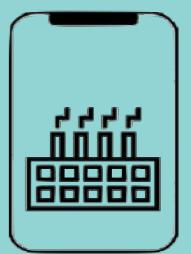
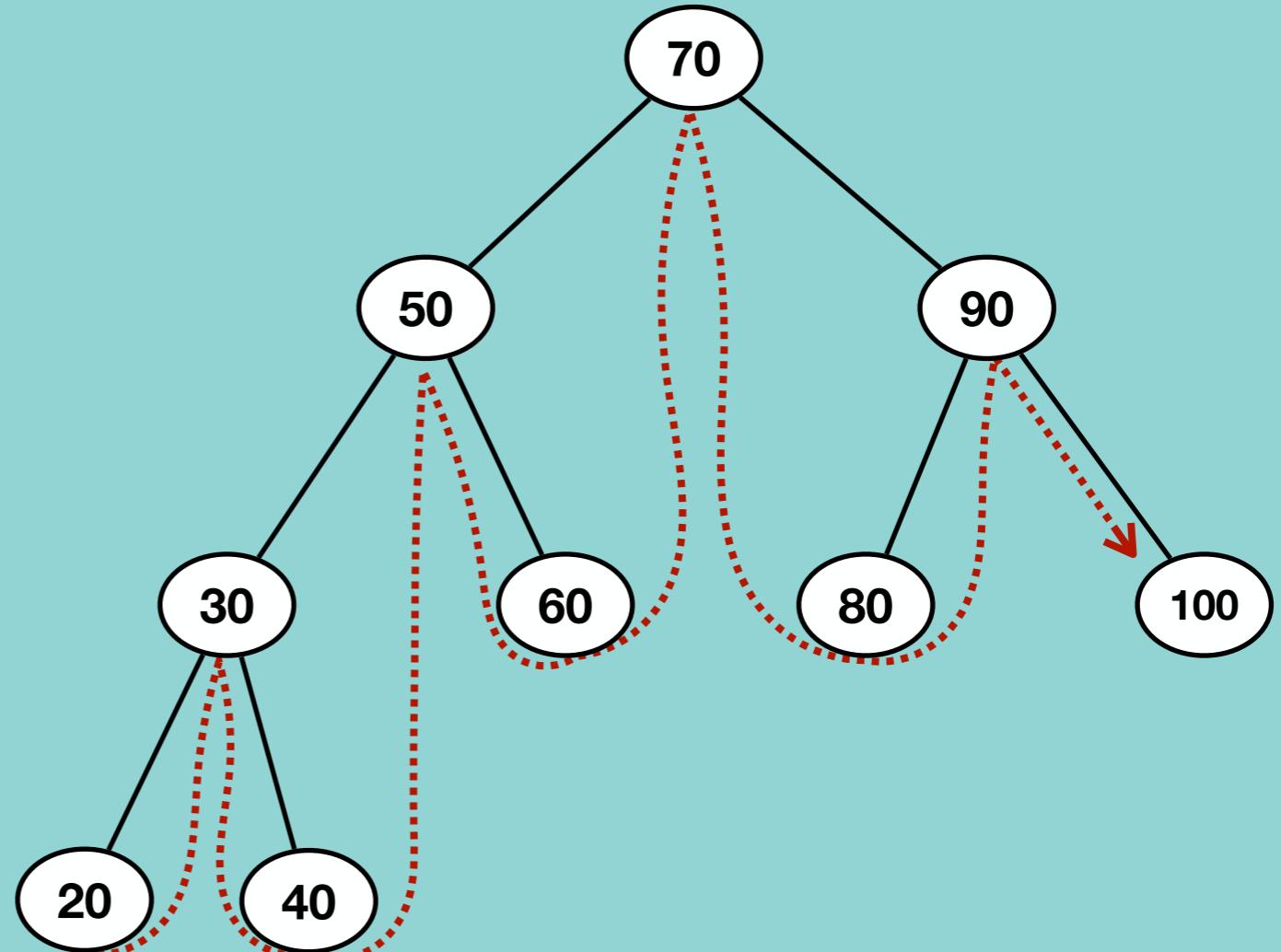
Left Subtree



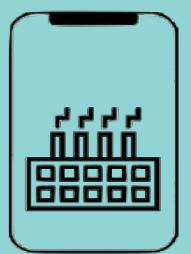
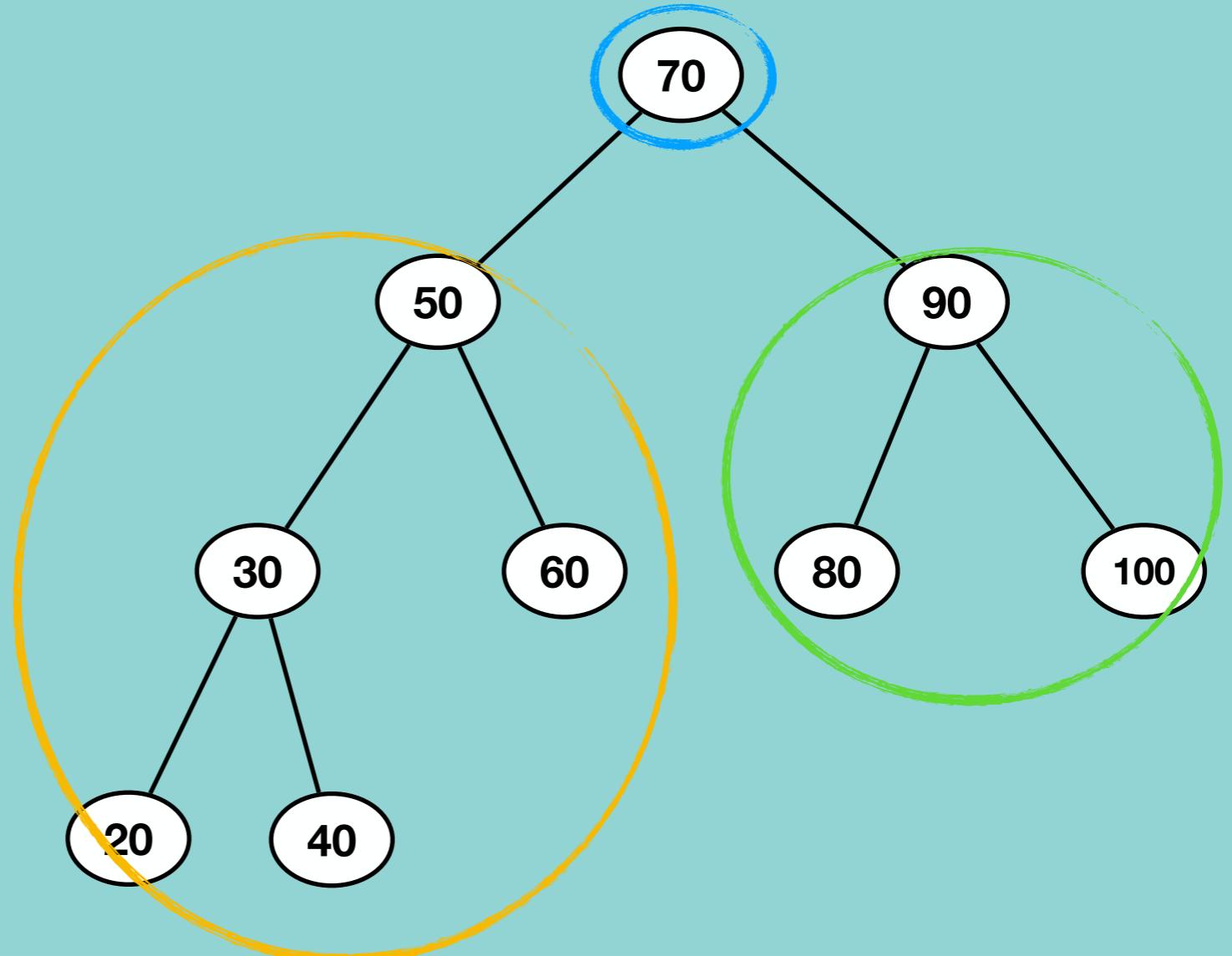
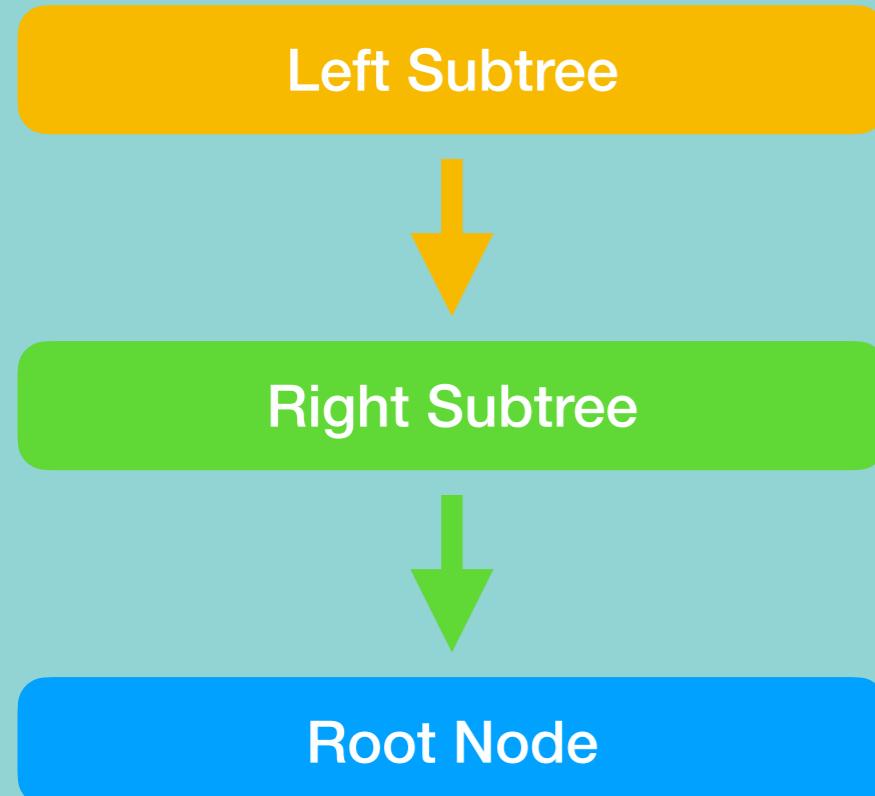
Root Node



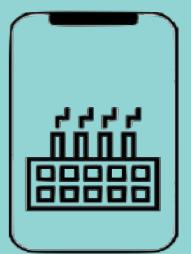
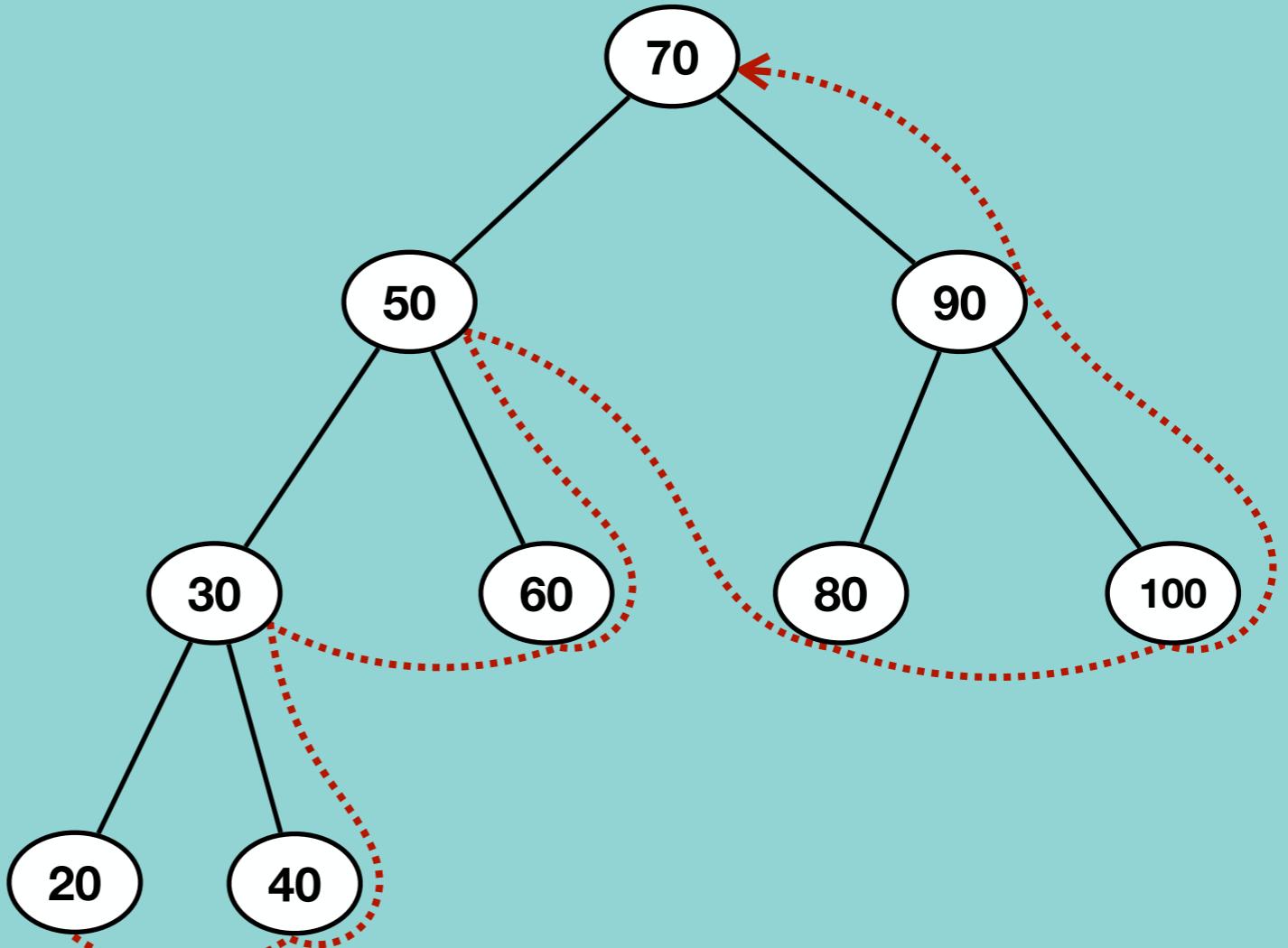
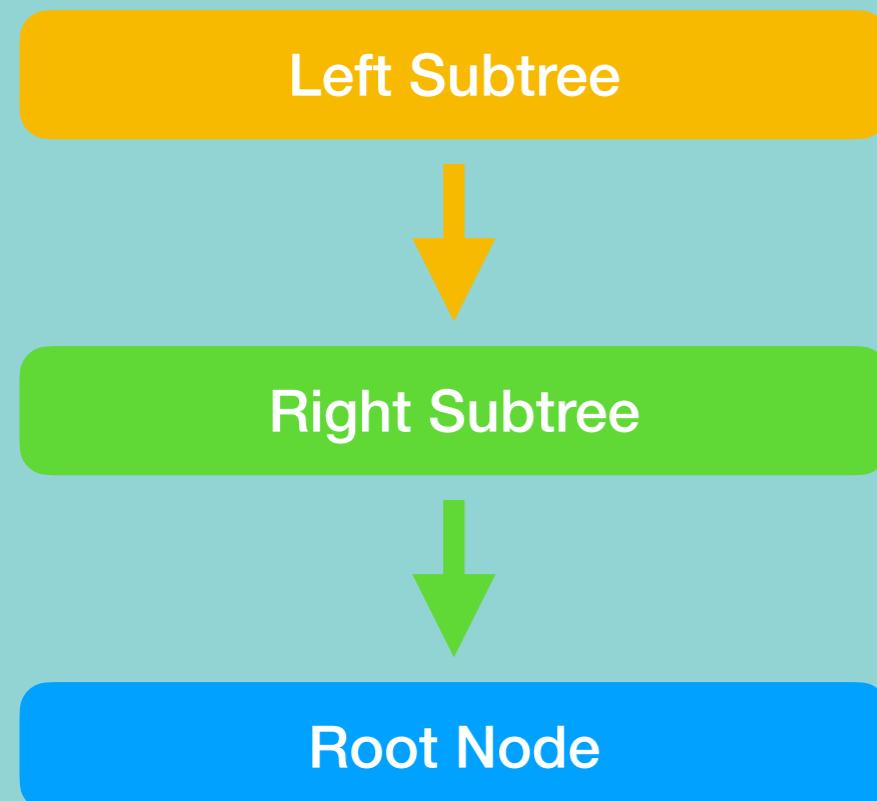
Right Subtree



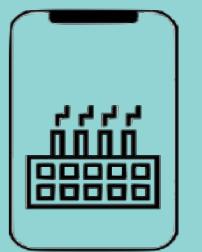
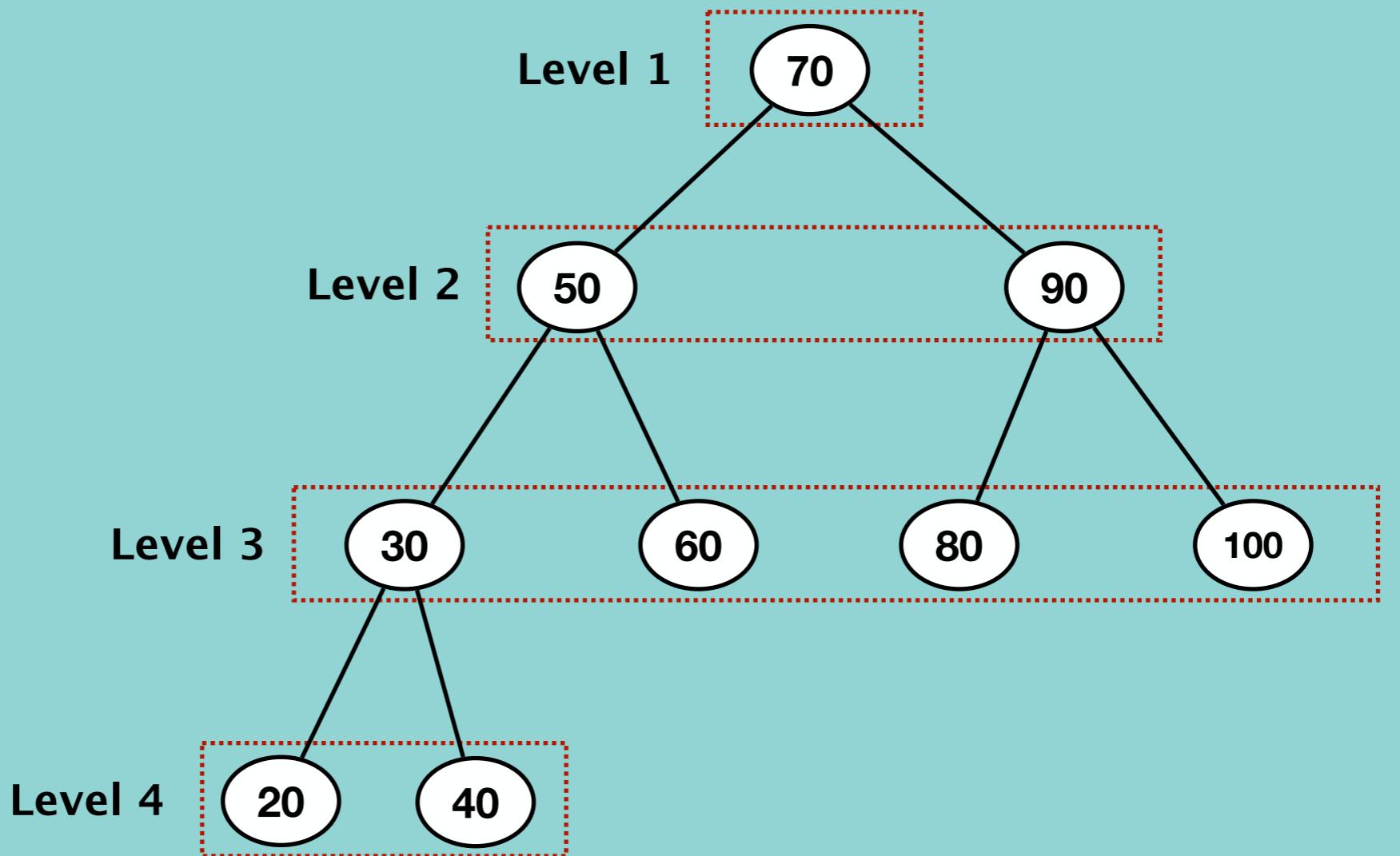
# PostOrder Traversal of Binary Search Tree



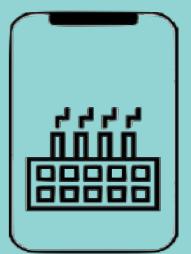
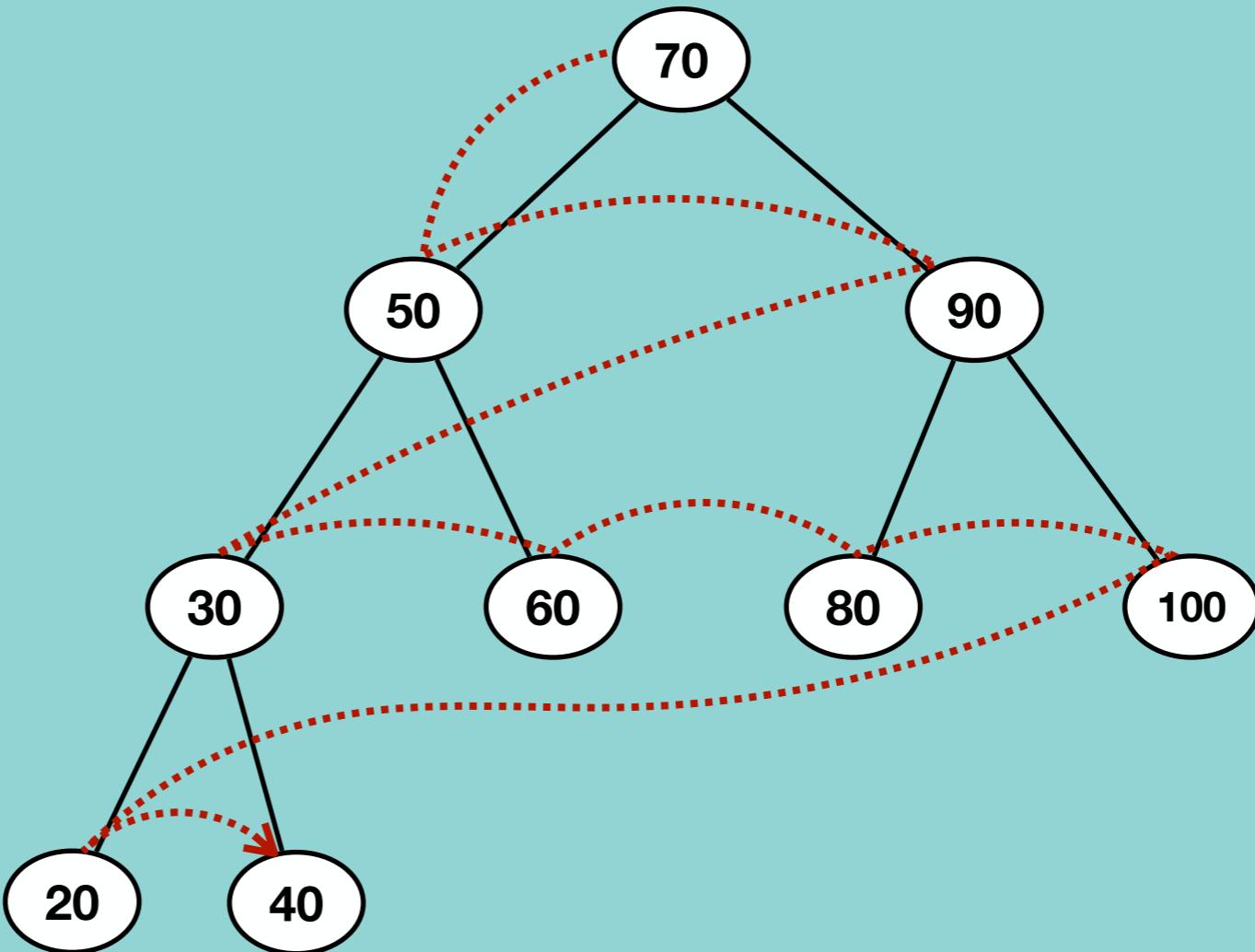
# PostOrder Traversal of Binary Search Tree



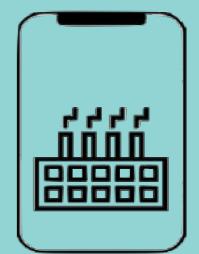
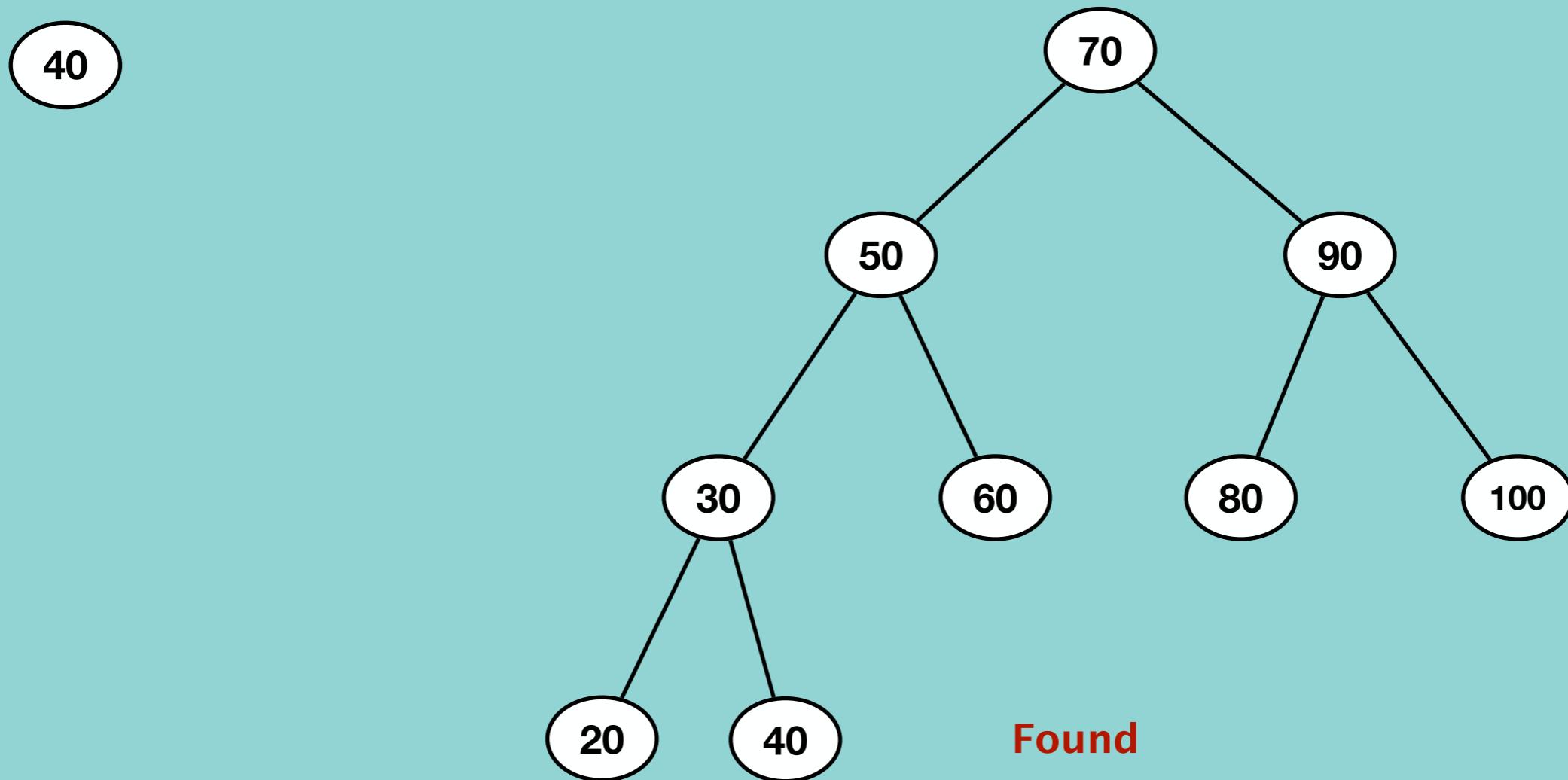
# LevelOrder Traversal of Binary Search Tree



# LevelOrder Traversal of Binary Search Tree



# Search for a node in Binary Search Tree

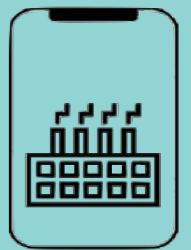
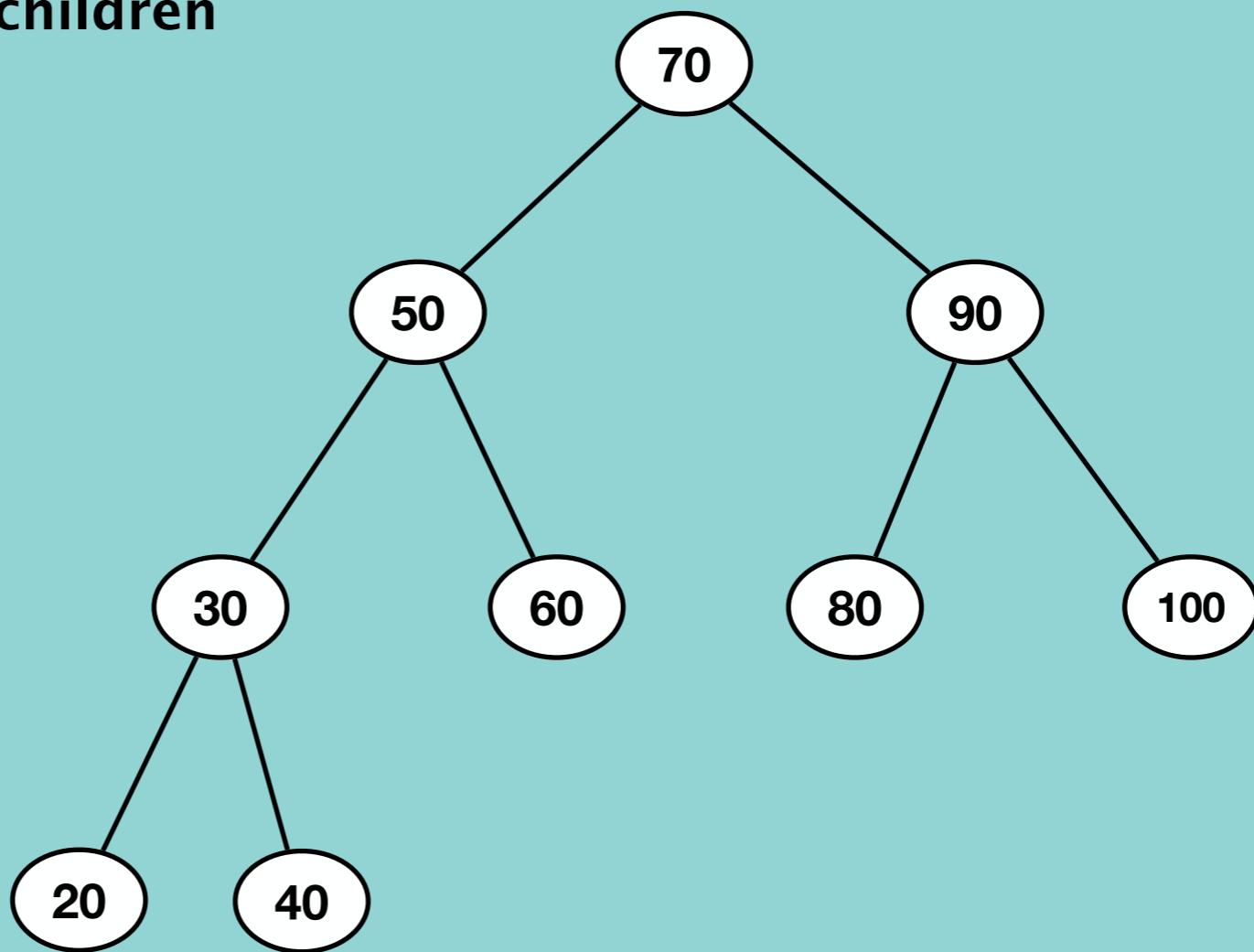


# Delete a node from Binary Search Tree

**Case 1: The node to be deleted is a leaf node**

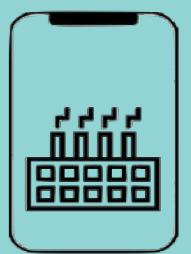
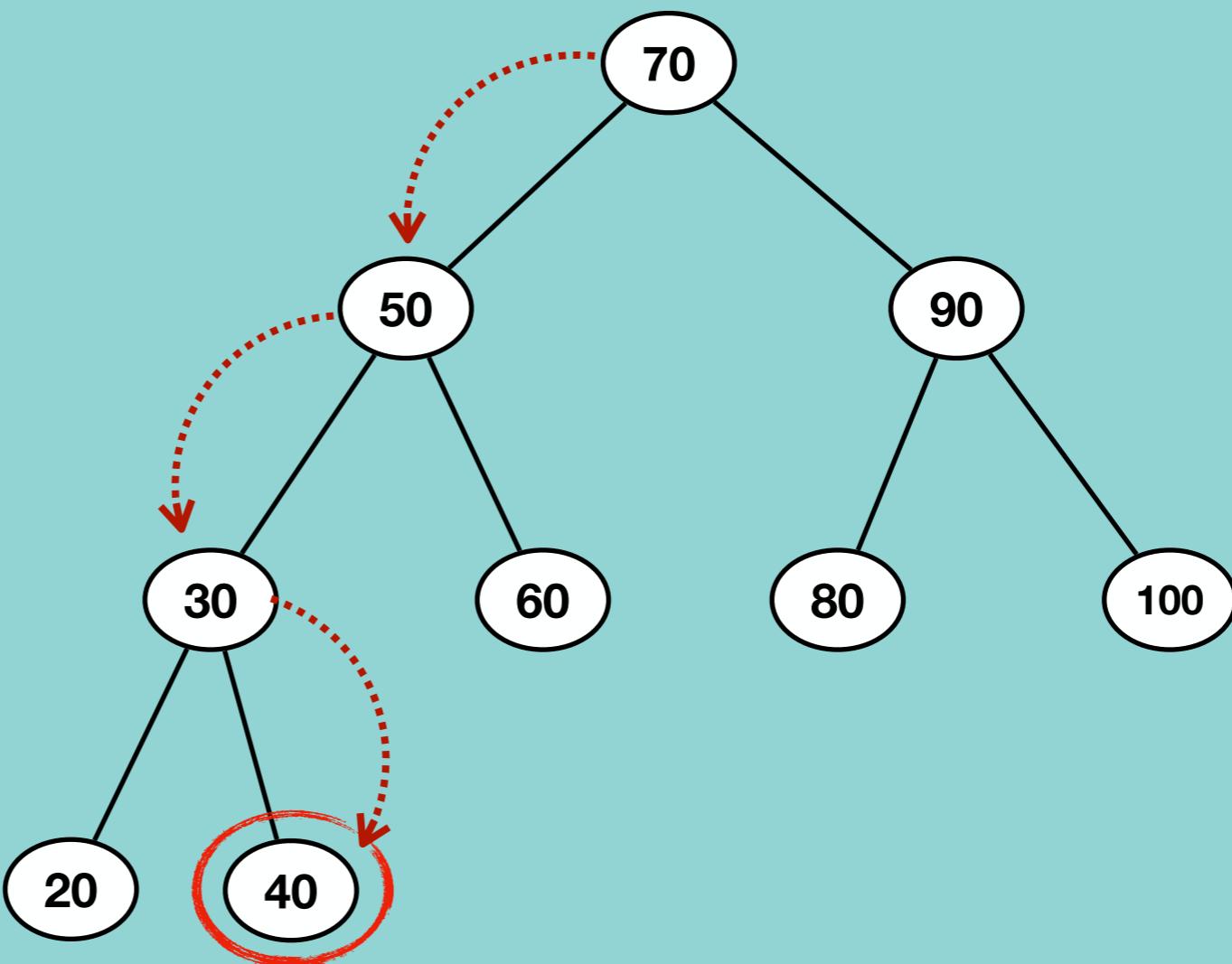
**Case 2: The node has one child**

**Case 3: The node has two children**



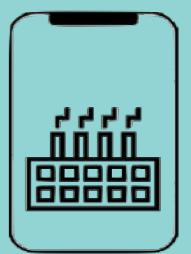
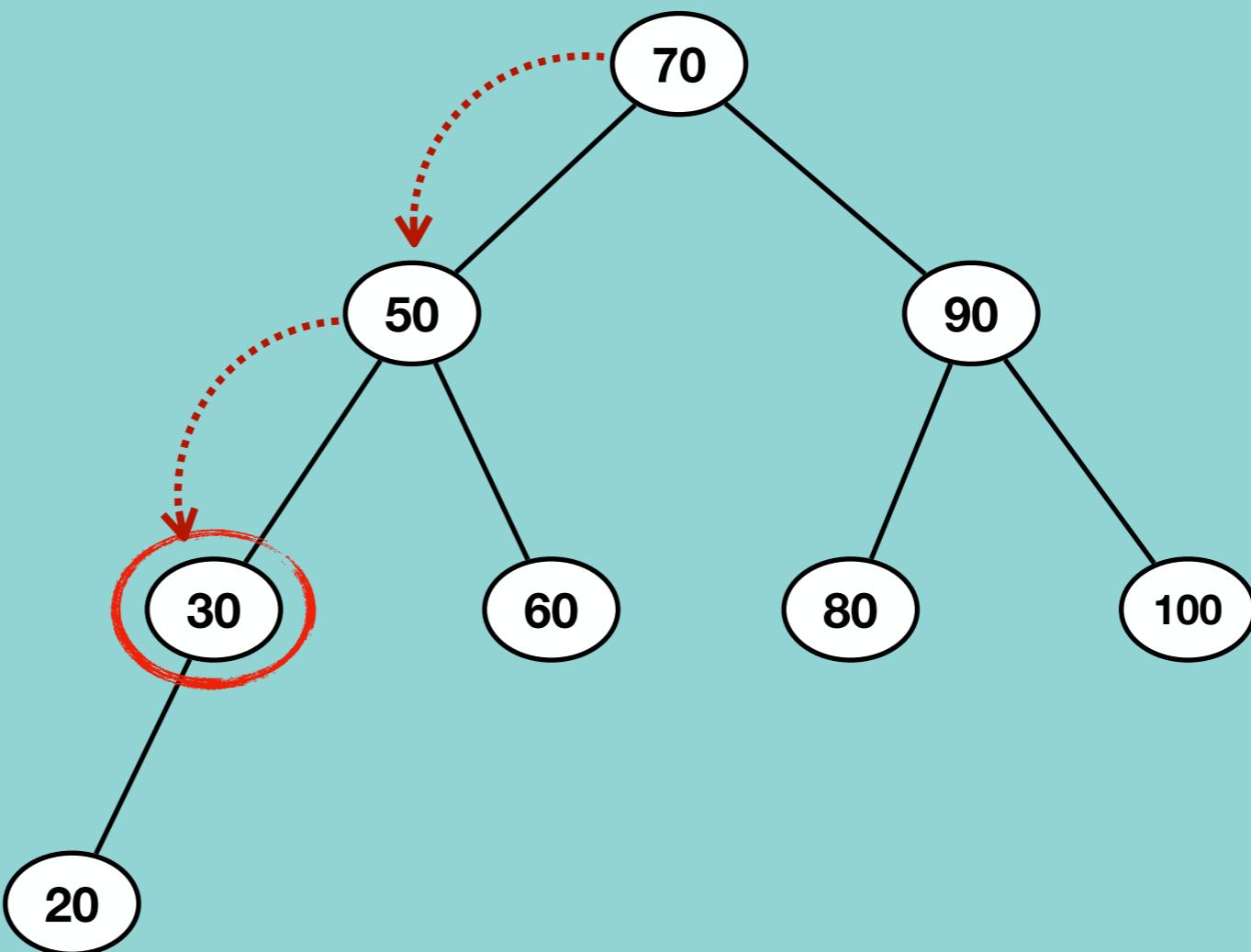
# Delete a node from Binary Search Tree

**Case 1: The node to be deleted is a leaf node**



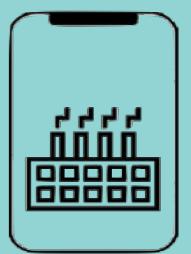
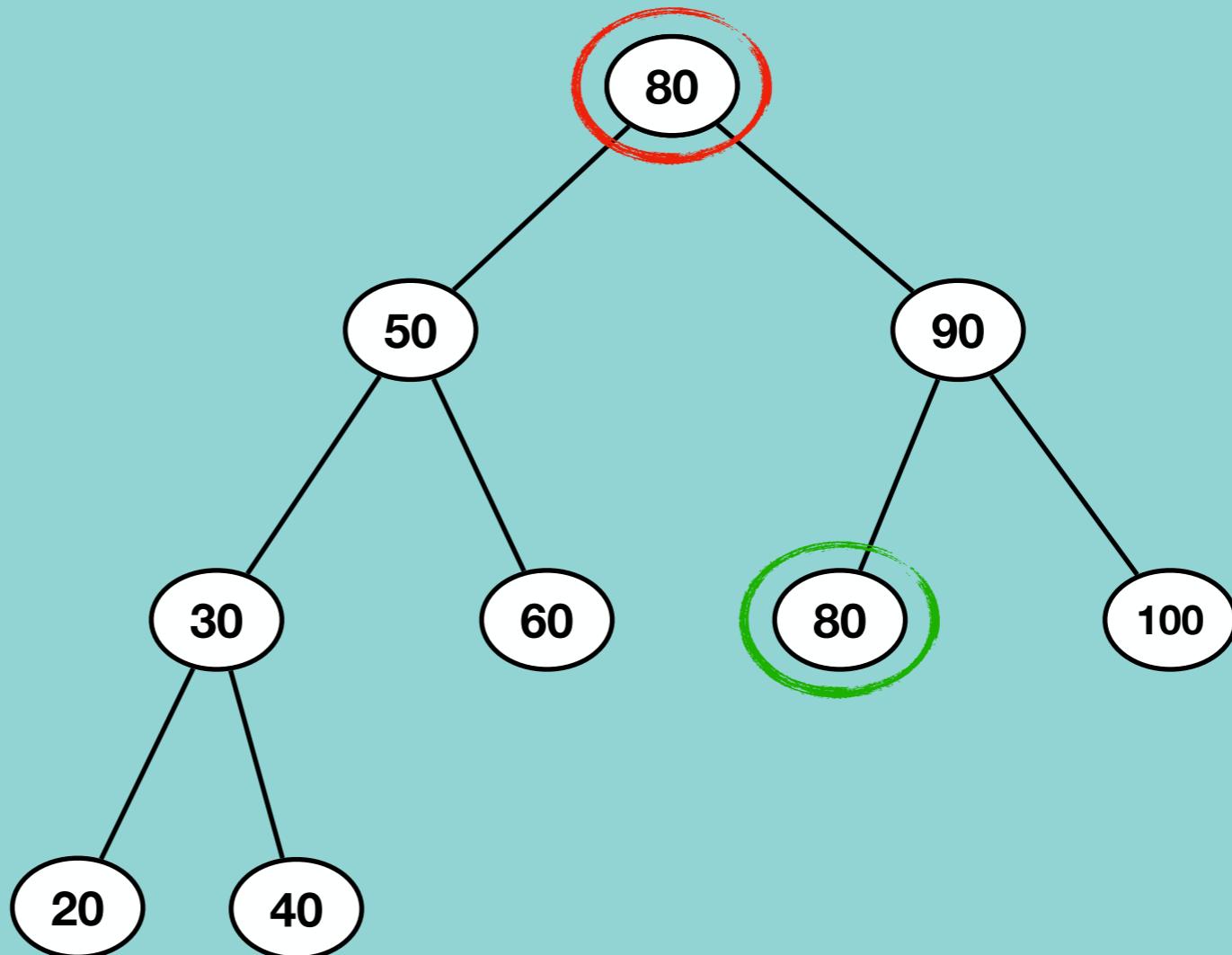
# Delete a node from Binary Search Tree

Case 2: The node to be deleted has a child node



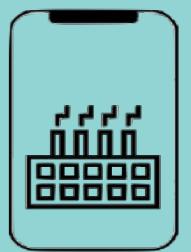
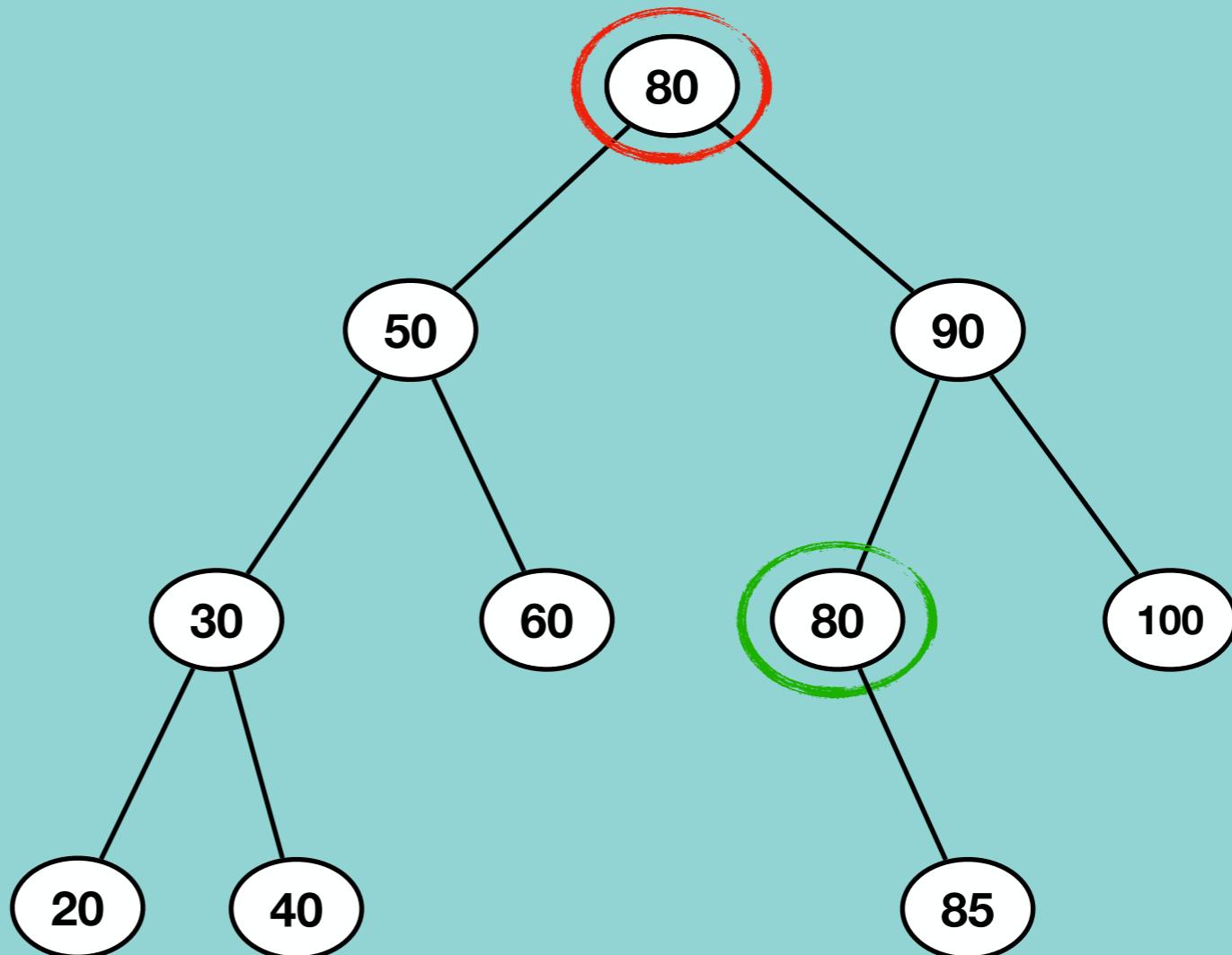
# Delete a node from Binary Search Tree

**Case 3: The node to be deleted has two children**



# Delete a node from Binary Search Tree

Case 3: The node to be deleted has two children

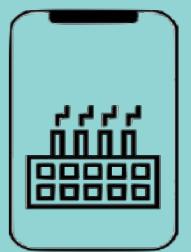
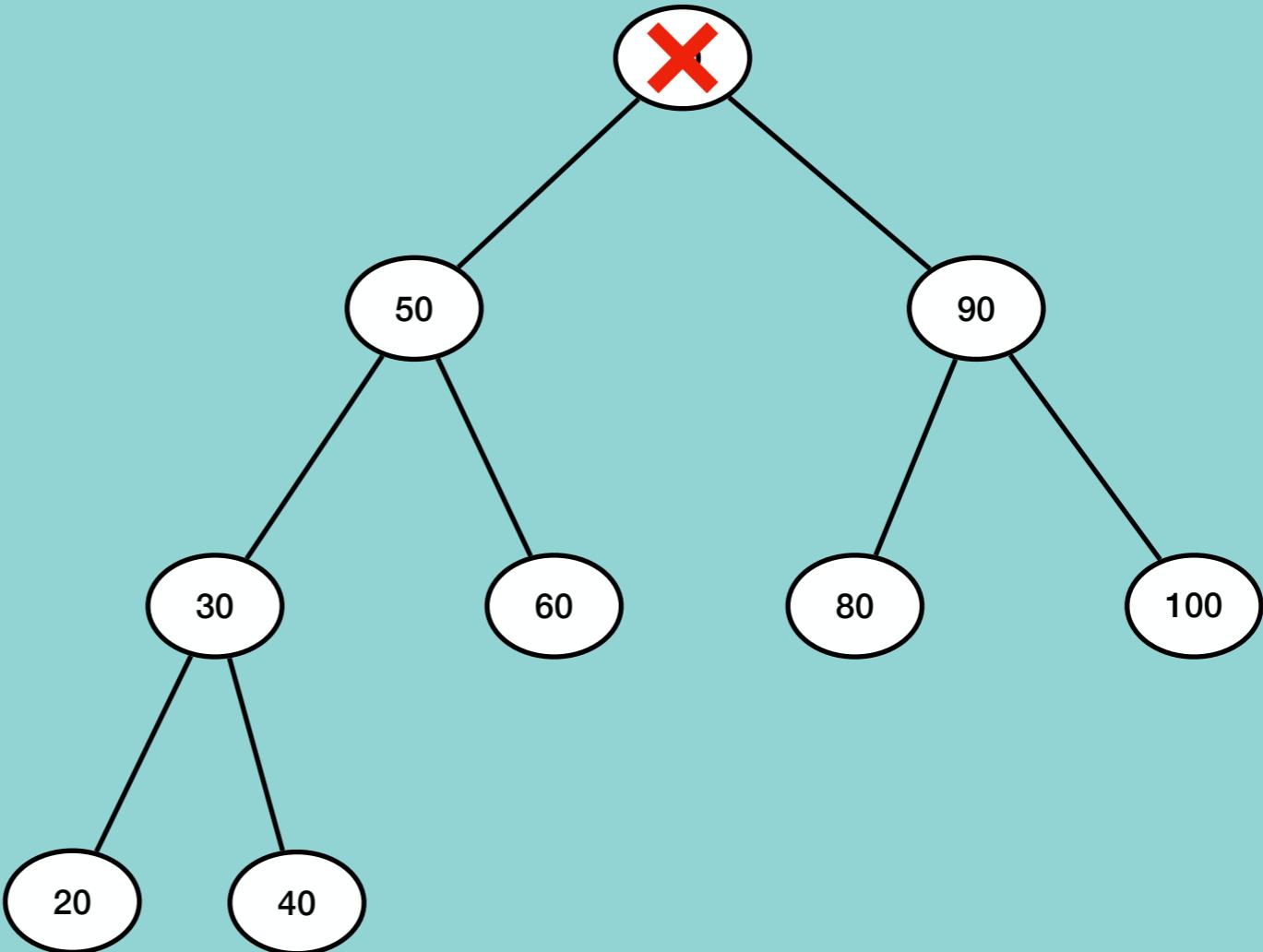


# Delete entire Binary Search Tree

```
rootNode = None
```

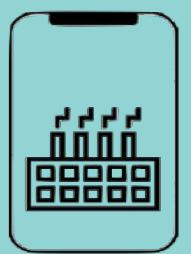
```
rootNode.leftChild = None
```

```
rootNode.rightChild = None
```



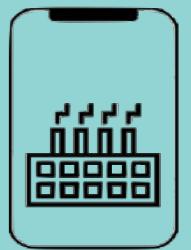
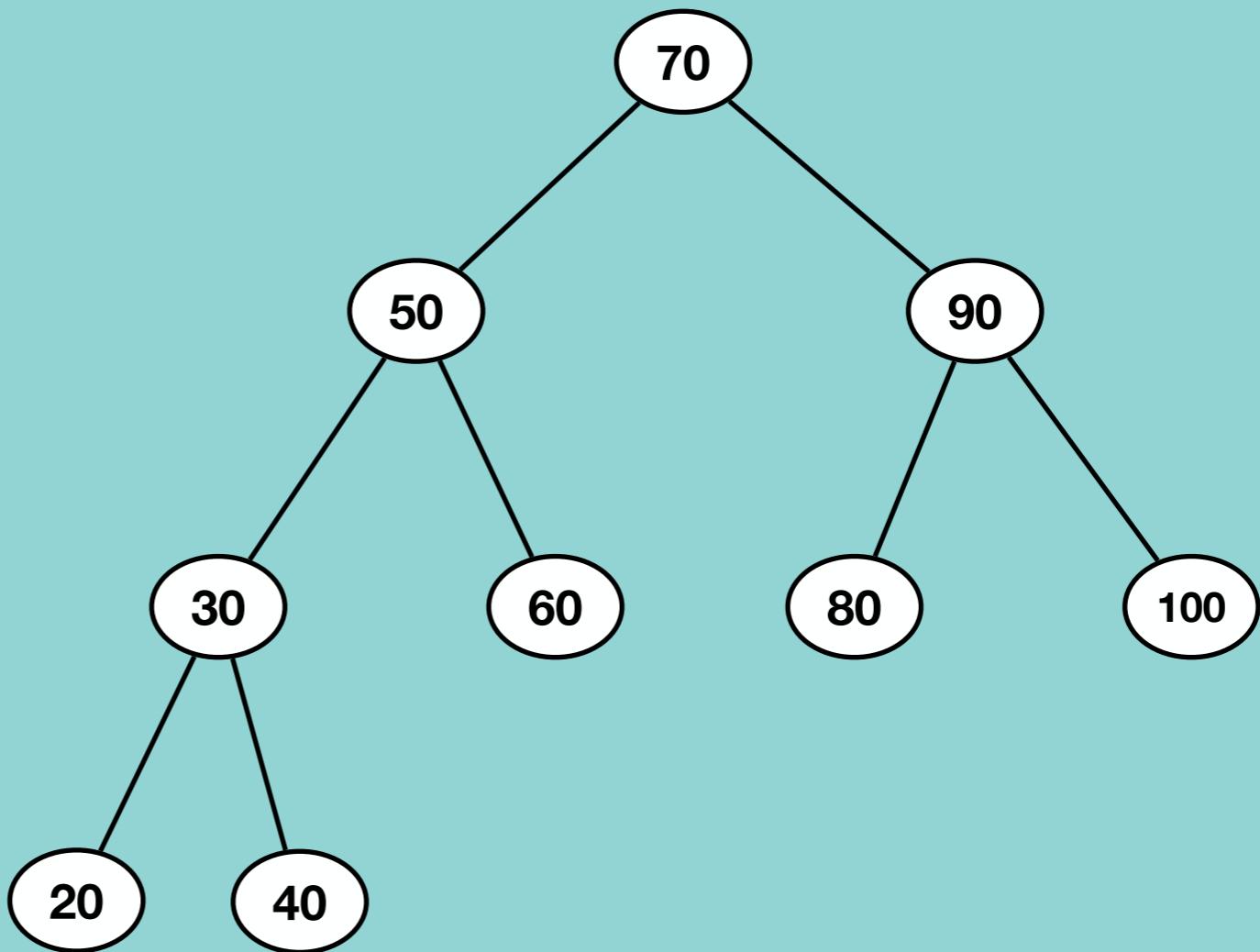
# Time and Space complexity of Binary Search Tree

	Time complexity	Space complexity
Create BST	O(1)	O(1)
Insert a node BST	O(logN)	O(logN)
Traverse BST	O(N)	O(N)
Search for a node BST	O(logN)	O(logN)
Delete node from BST	O(logN)	O(logN)
Delete Entire BST	O(1)	O(1)



# What is an AVL Tree?

An AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.



# What is an AVL Tree?

An AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Height of leftSubtree = 3

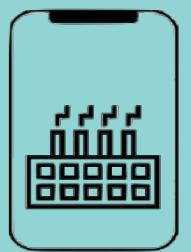
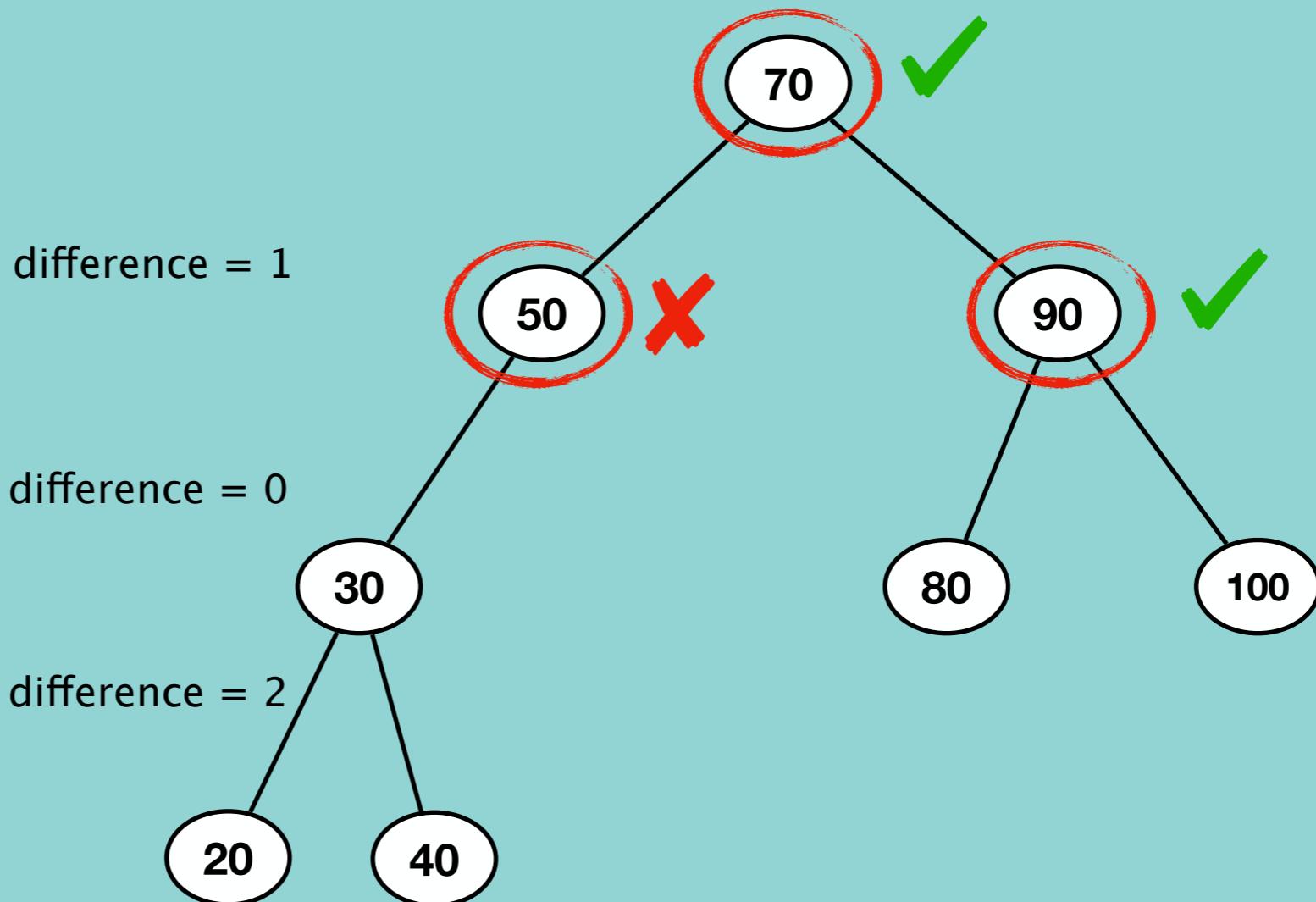
Height of rightSubtree = 2

Height of leftSubtree = 1

Height of rightSubtree = 1

Height of leftSubtree = 2

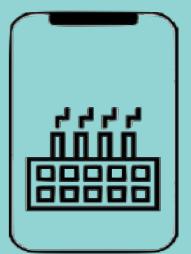
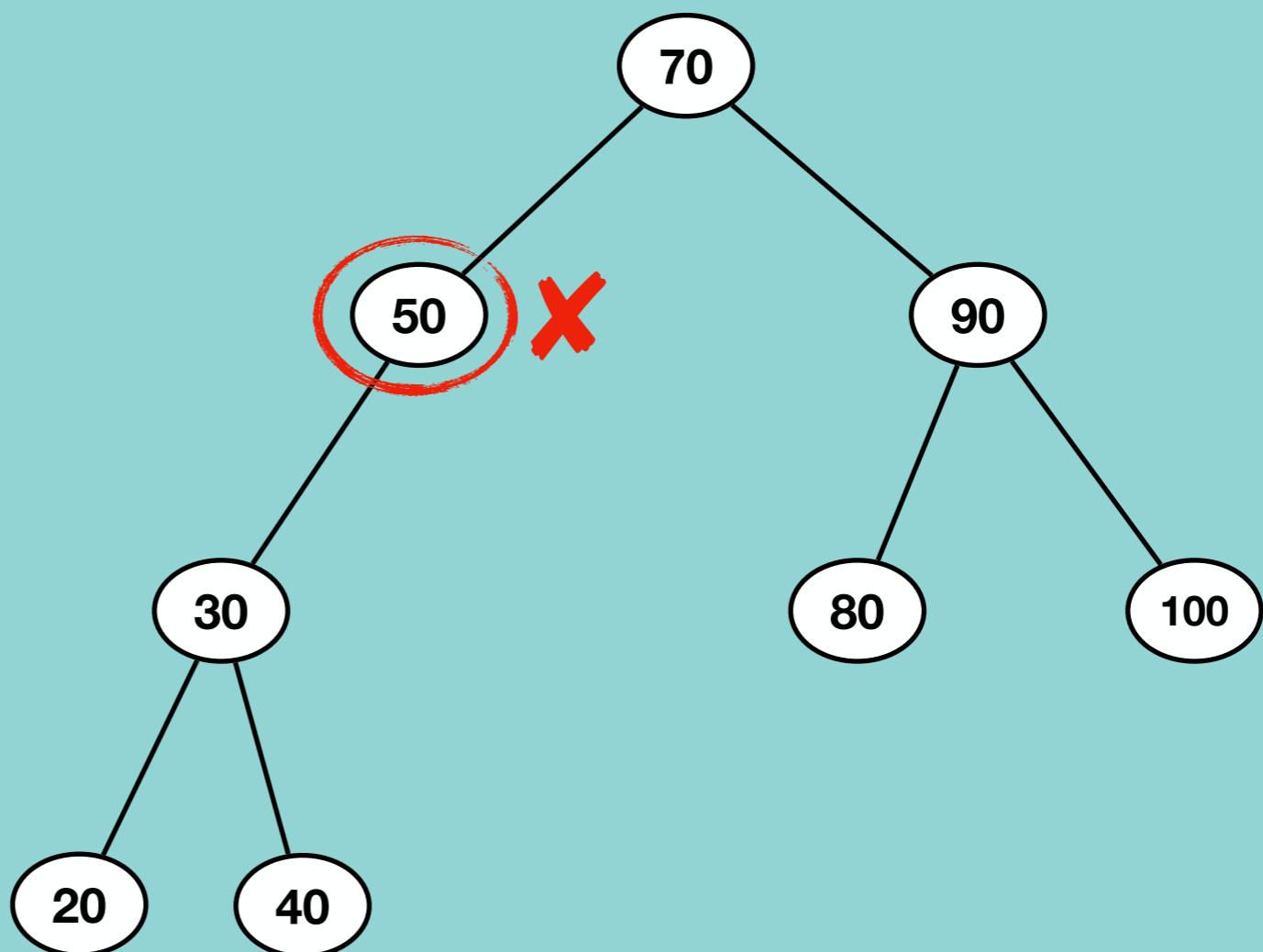
Height of rightSubtree = 0



# What is an AVL Tree?

An AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

If at any time heights of left and right subtrees differ by more than one, then rebalancing is done to restore AVL property, this process is called **rotation**



# Examples

Height of leftSubtree = 2

Height of rightSubtree = 2

Height of leftSubtree = 1

Height of rightSubtree = 1

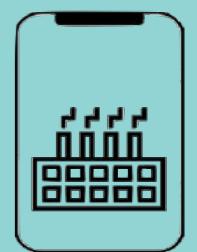
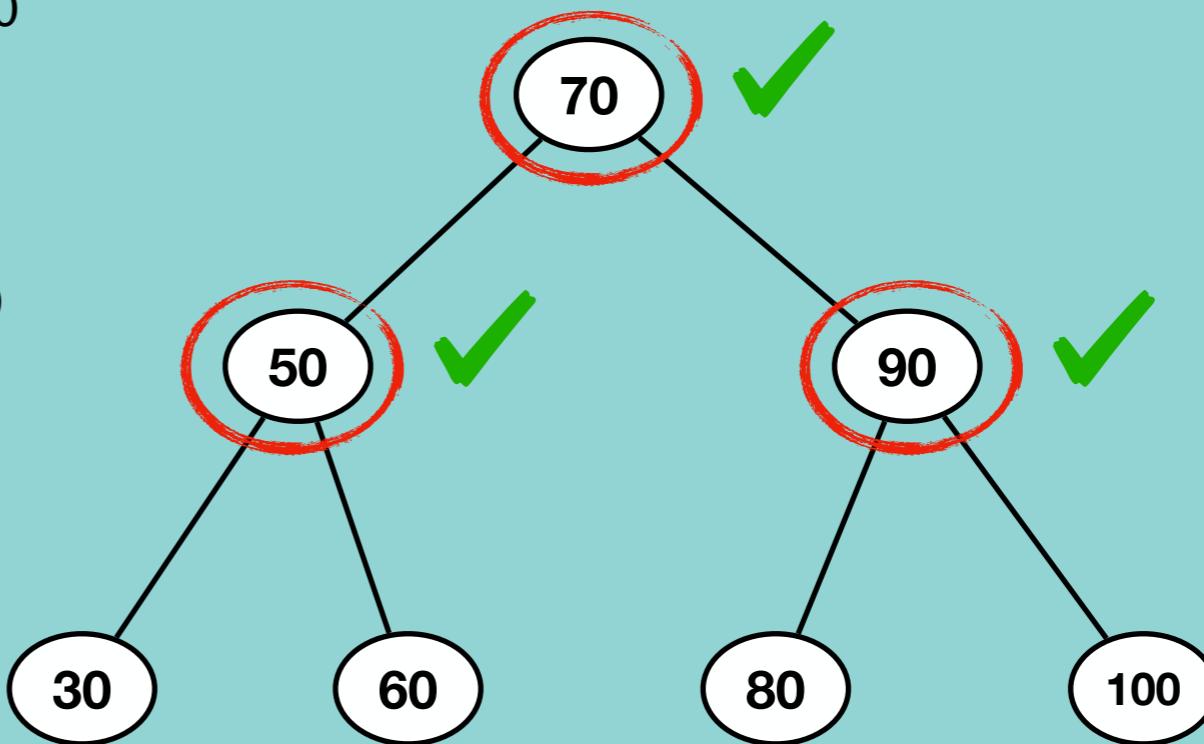
Height of leftSubtree = 1

Height of rightSubtree = 1

difference = 0

difference = 0

difference = 0



# Examples

Height of leftSubtree = 3

Height of rightSubtree = 2

Height of leftSubtree = 2

Height of rightSubtree = 1

Height of leftSubtree = 1

Height of rightSubtree = 1

Height of leftSubtree = 1

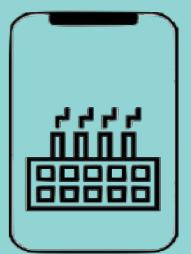
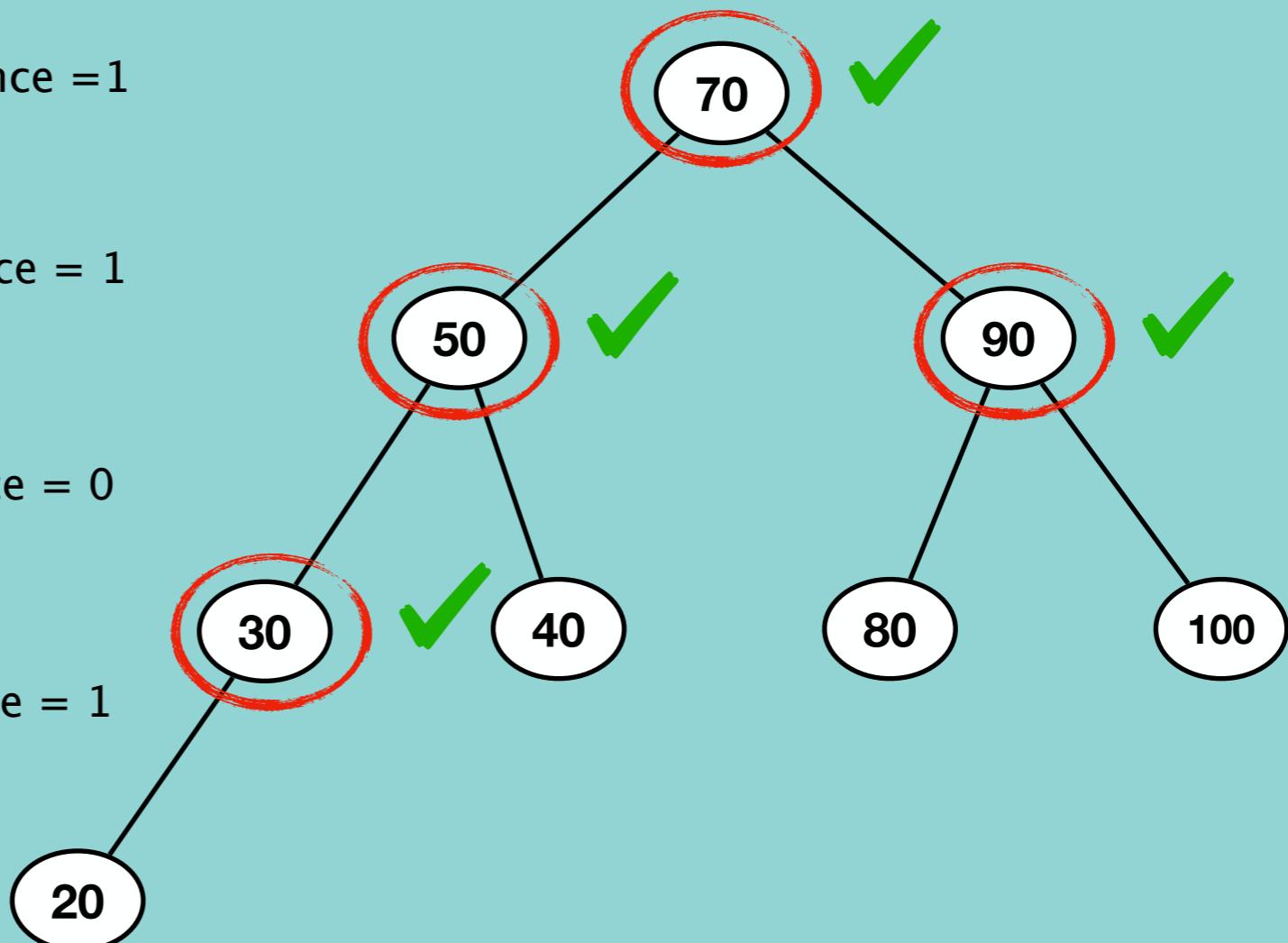
Height of rightSubtree = 0

difference = 1

difference = 1

difference = 0

difference = 1

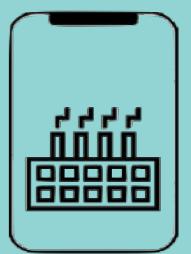
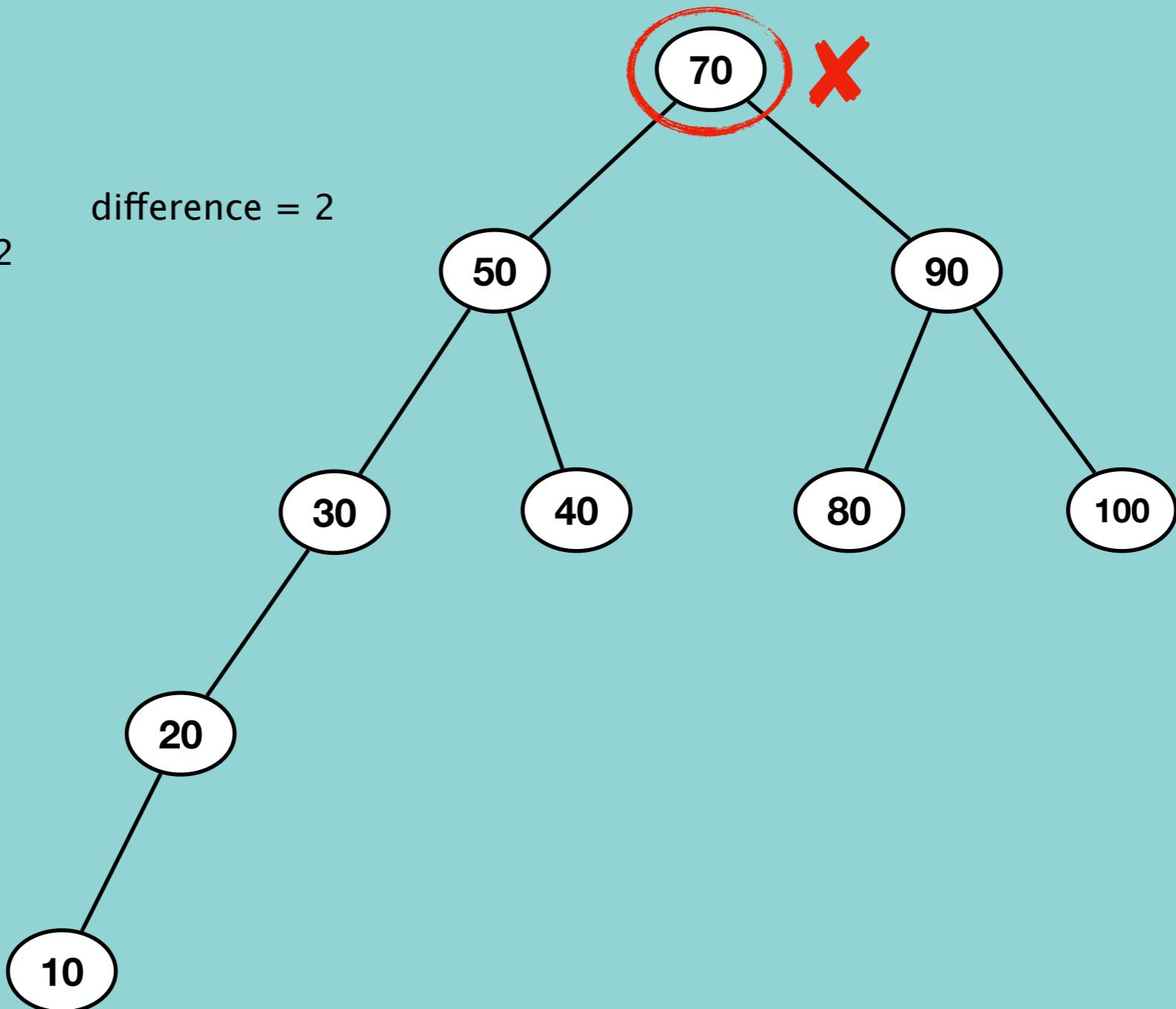


## Examples

Height of leftSubtree = 4

Height of rightSubtree = 2

difference = 2

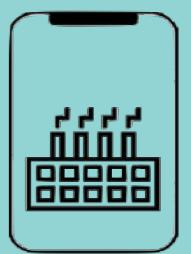
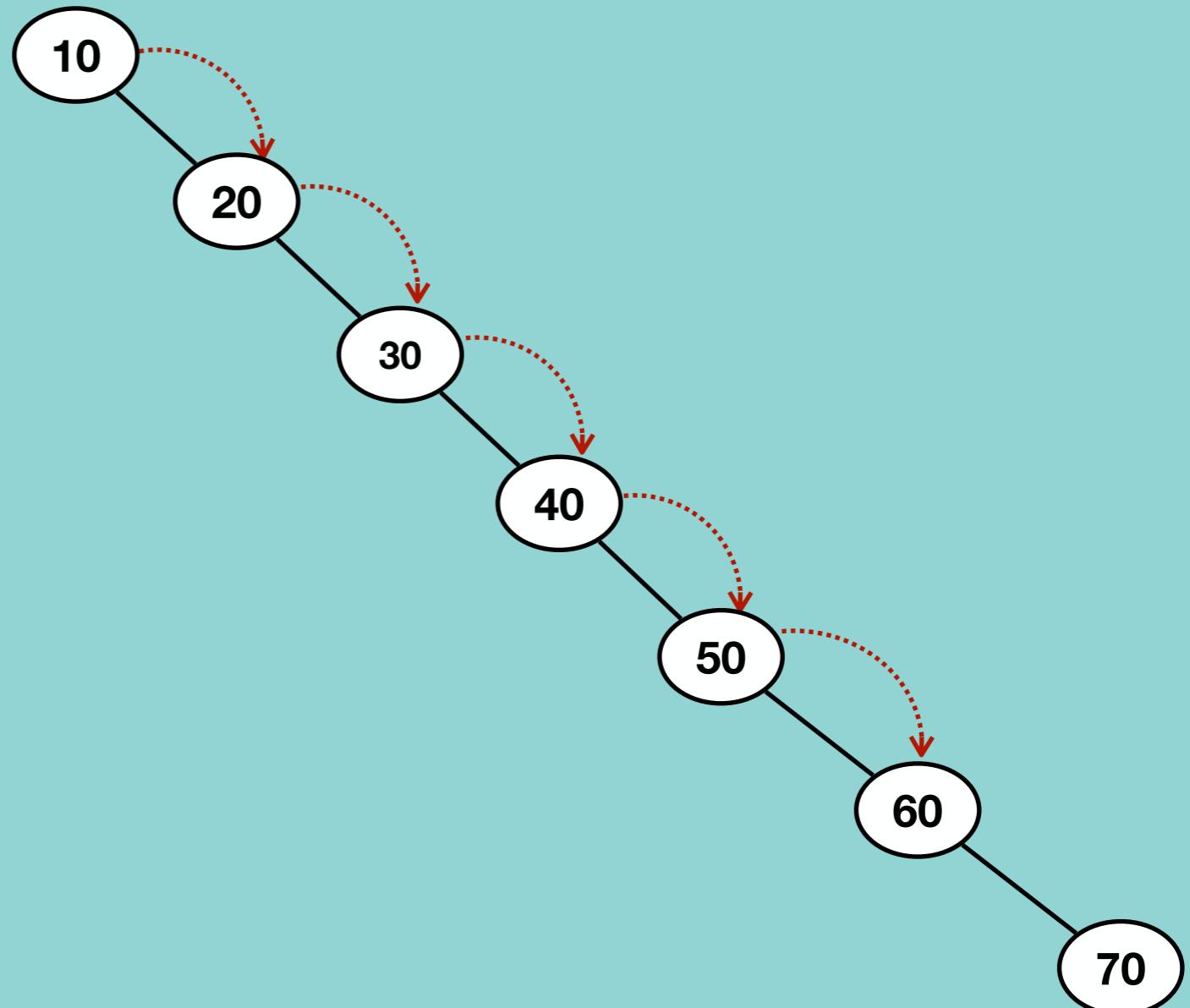


# Why do we need AVL Tree?

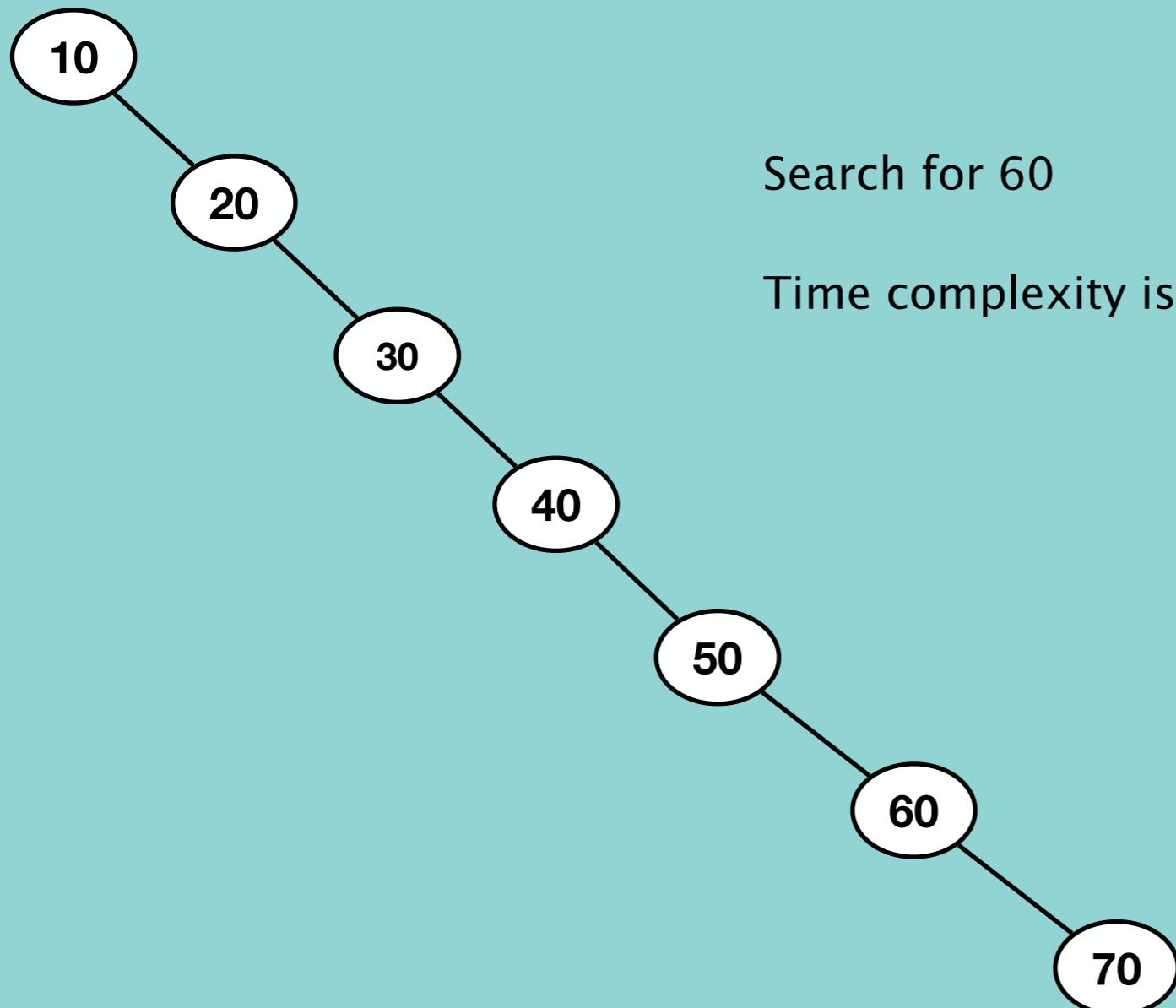
10, 20, 30, 40, 50, 60, 70

Search for 60

Time complexity is O(N)

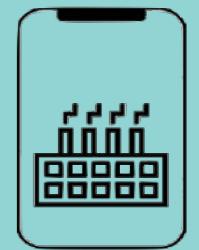
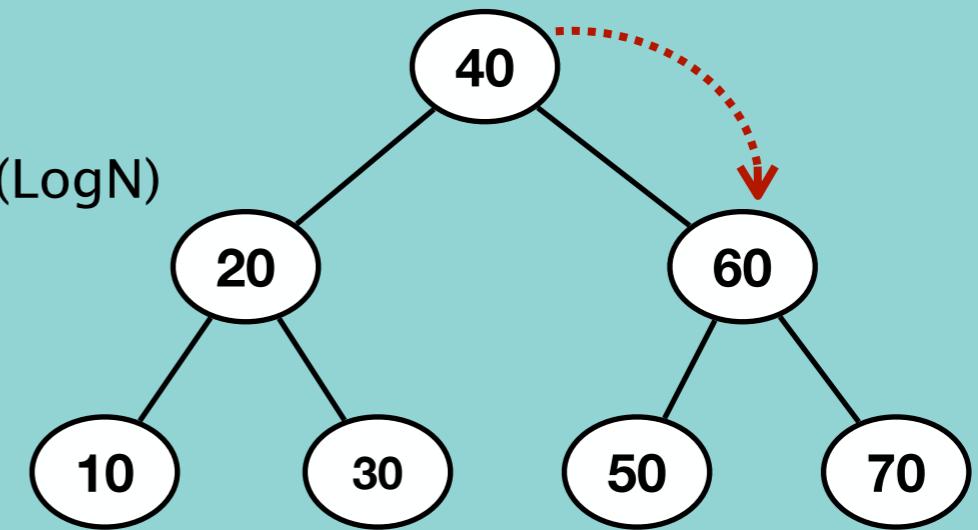


# Why do we need AVL Tree?



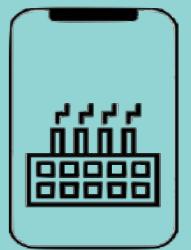
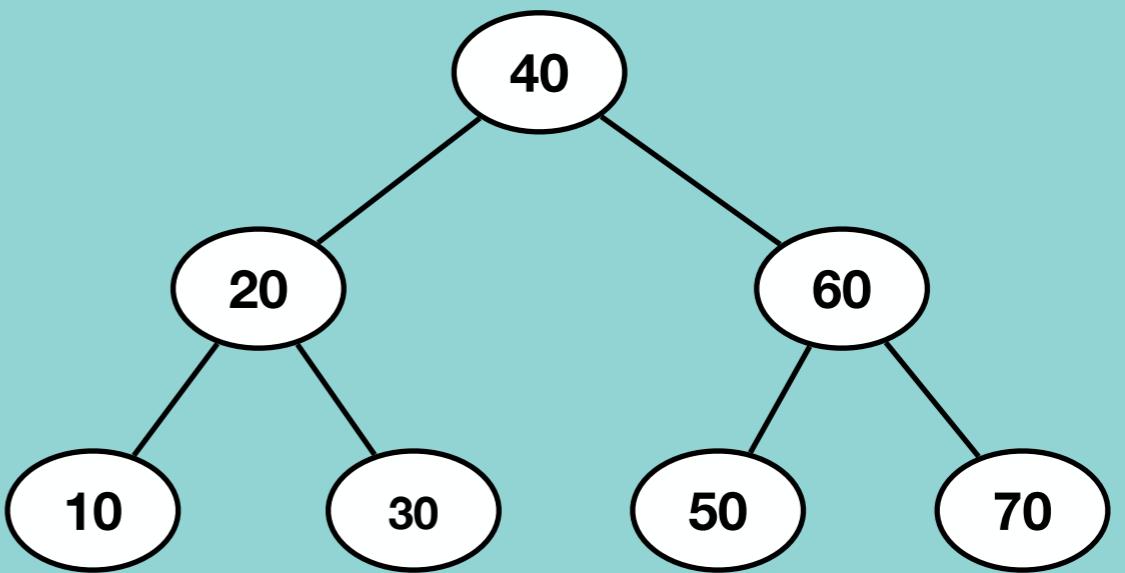
Search for 60

Time complexity is  $O(\text{Log}N)$



# Common operations on AVL Trees

- Creation of AVL trees,
- Search for a node in AVL trees
- Traverse all nodes in AVL trees
- Insert a node in AVL trees
- Delete a node from AVL trees
- Delete the entire AVL trees



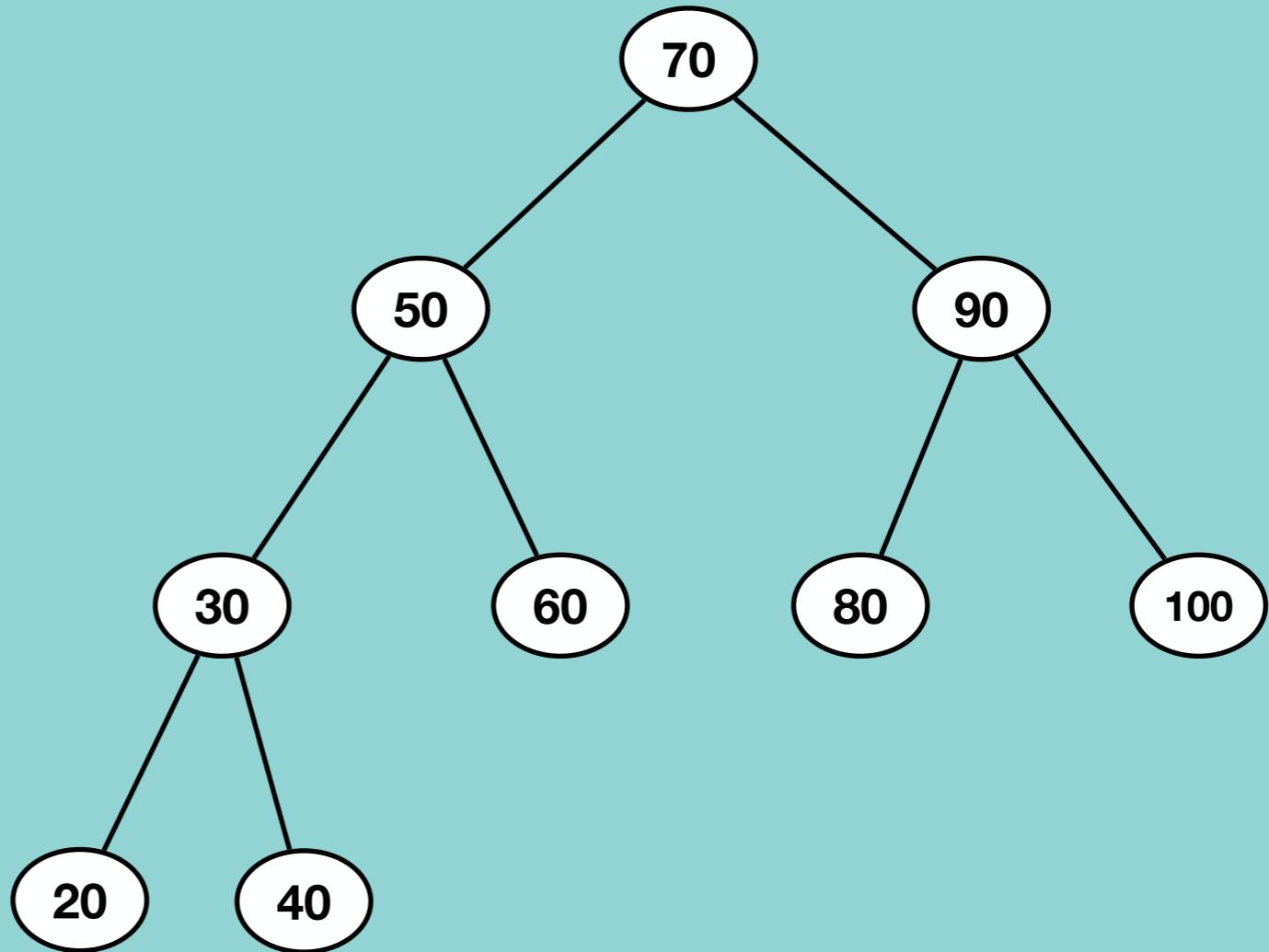
# Create AVL Tree

```
newAVL = AVL()
```

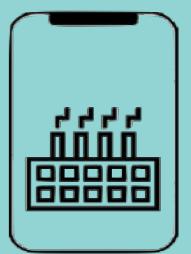
```
rootNode = None
```

```
leftChild = None
```

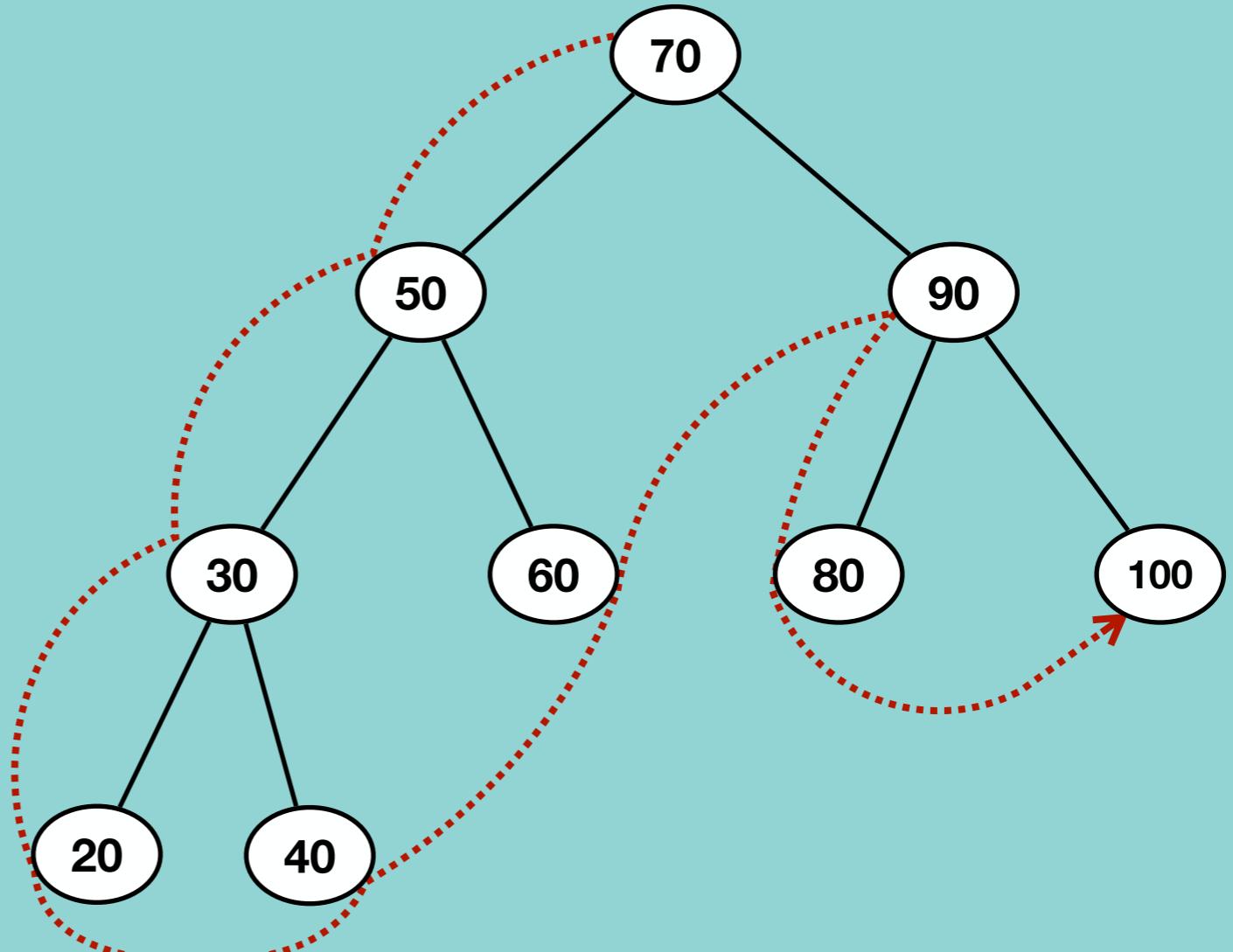
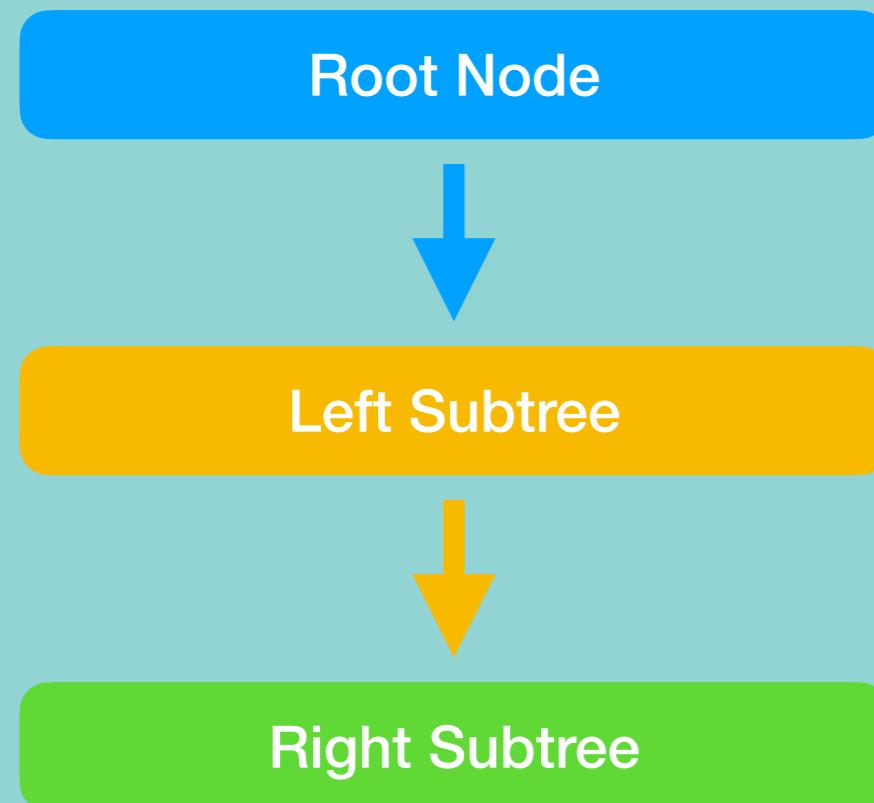
```
rightChild = None
```



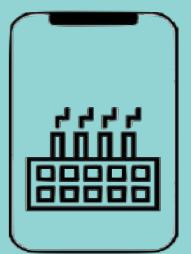
Time complexity : O(1)  
Space complexity : O(1)



# PreOrder Traversal of AVL Tree



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



# InOrder Traversal of AVL Tree

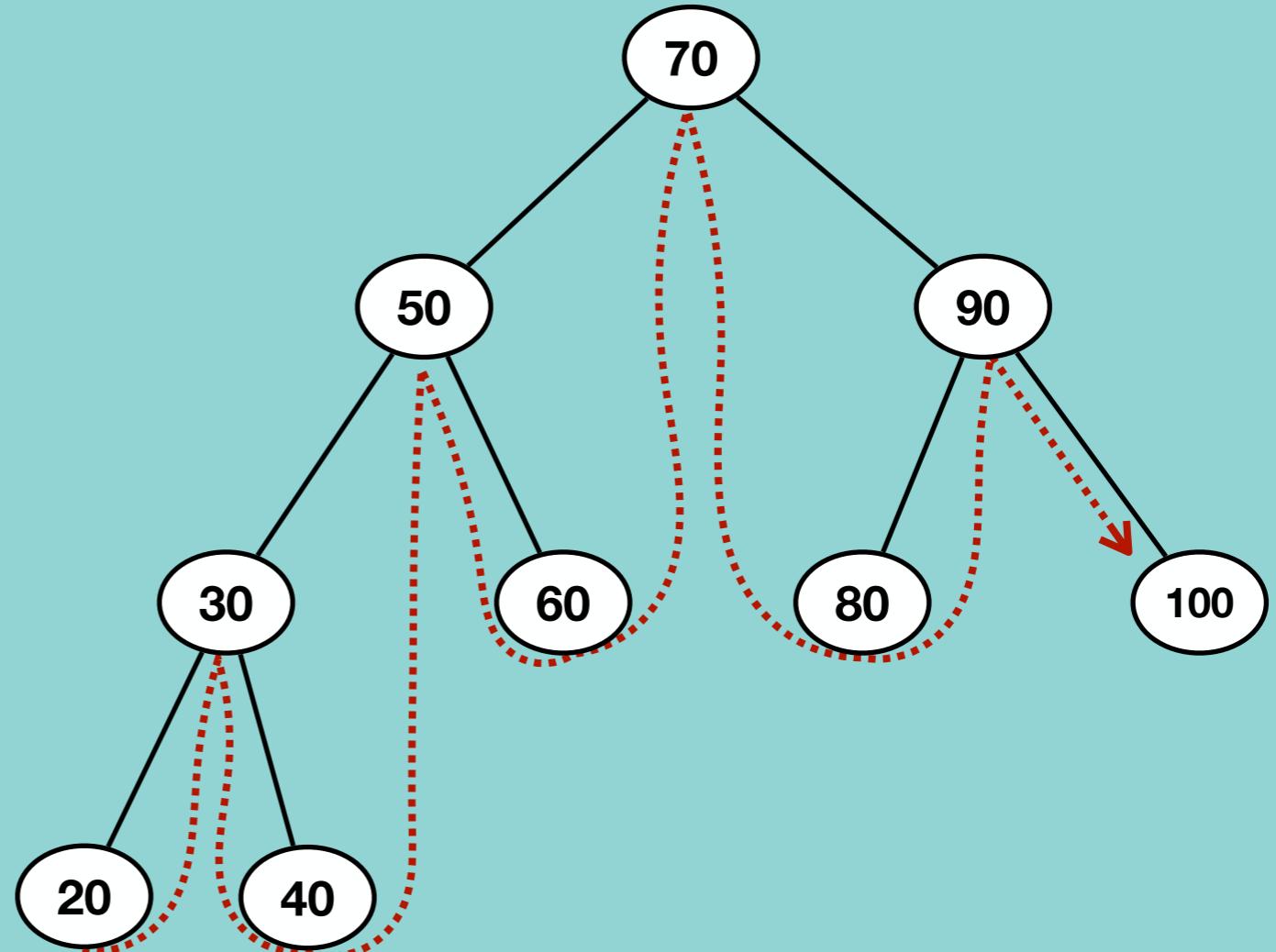
Left Subtree



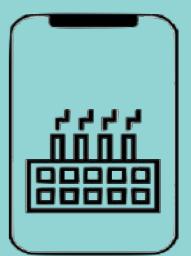
Root Node



Right Subtree



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



# PostOrder Traversal of AVL Tree

Left Subtree

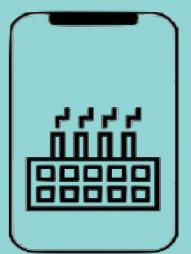
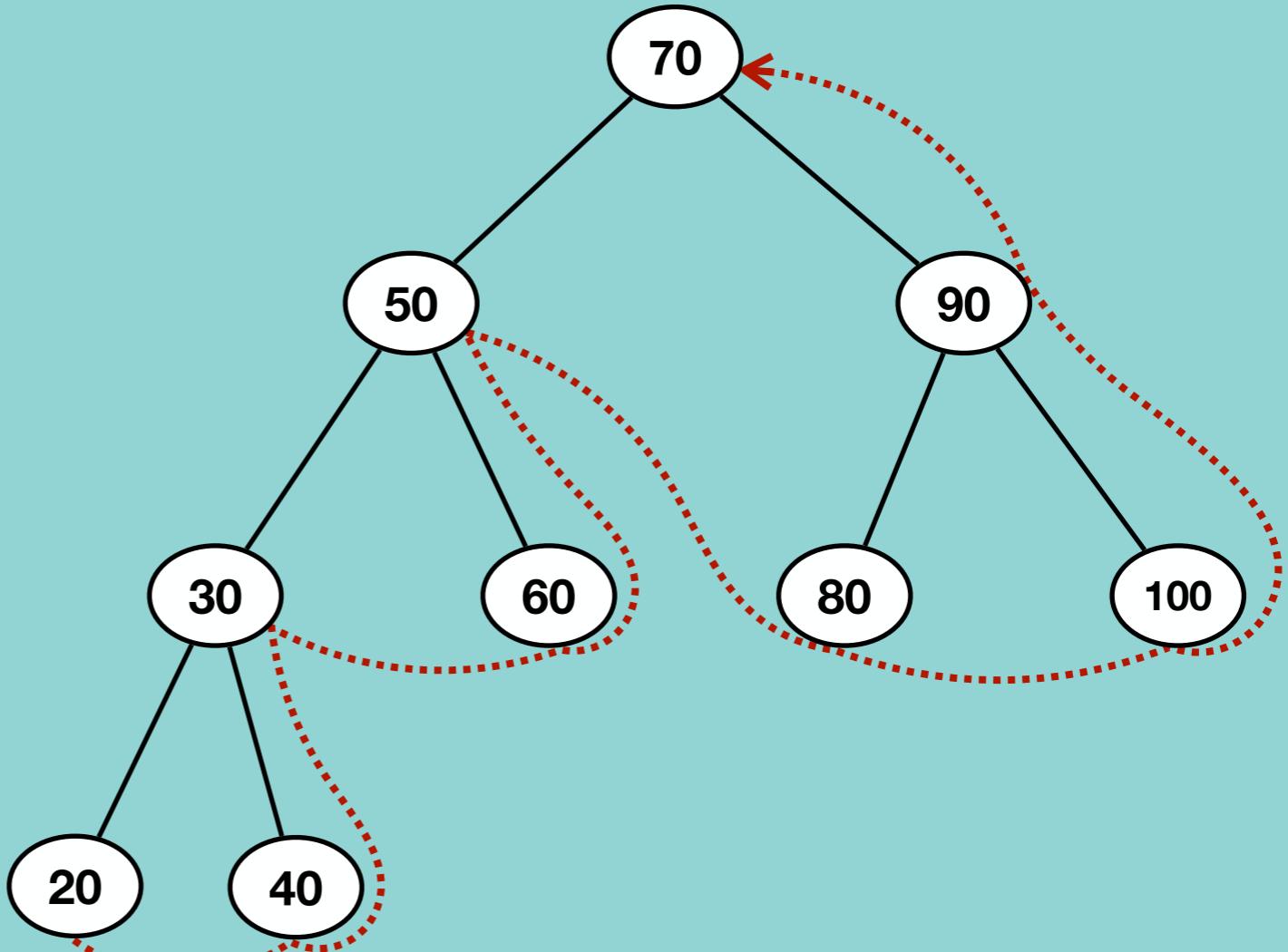


Right Subtree

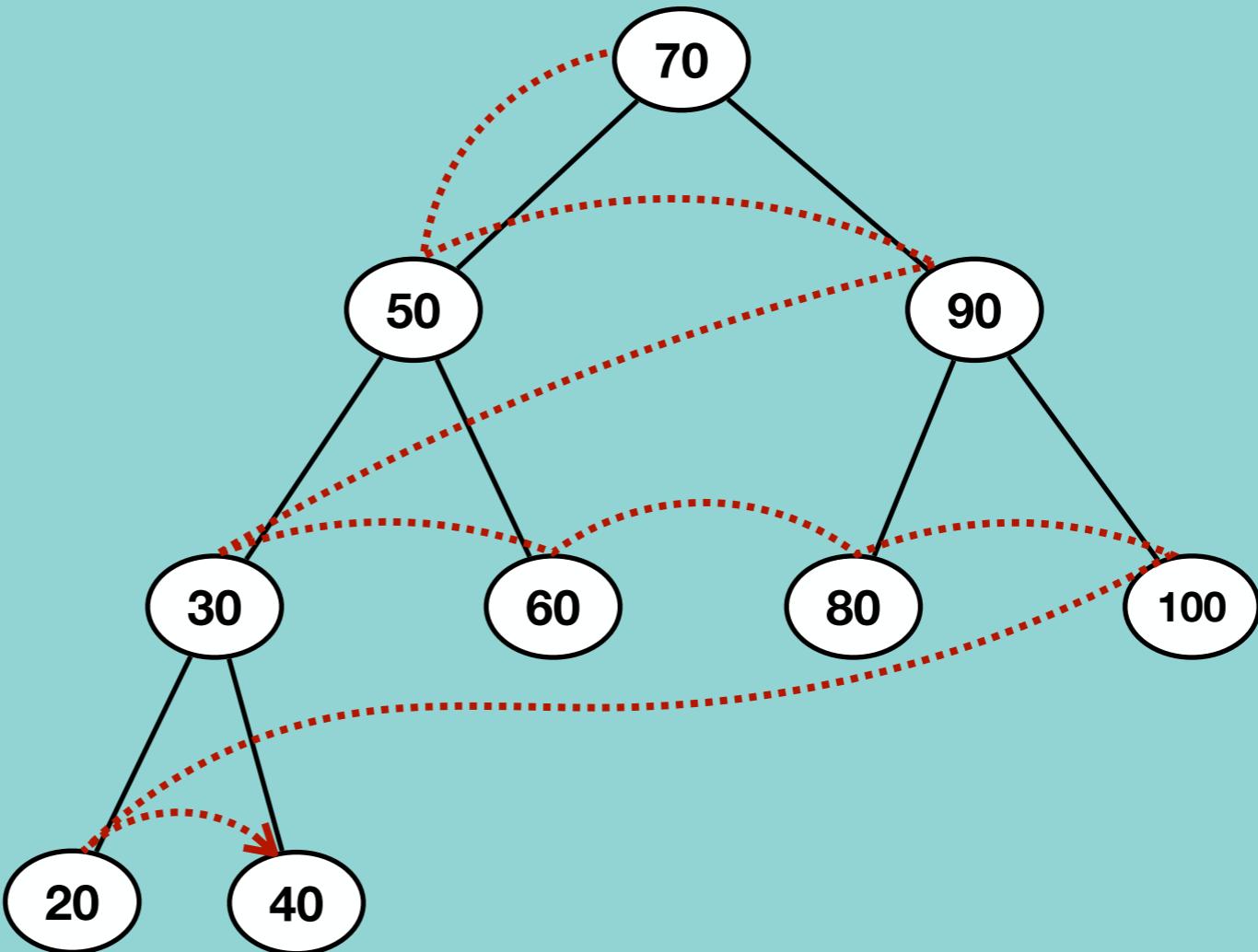


Root Node

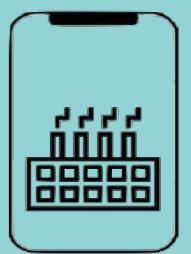
Time complexity : O(n)  
Space complexity : O(n)



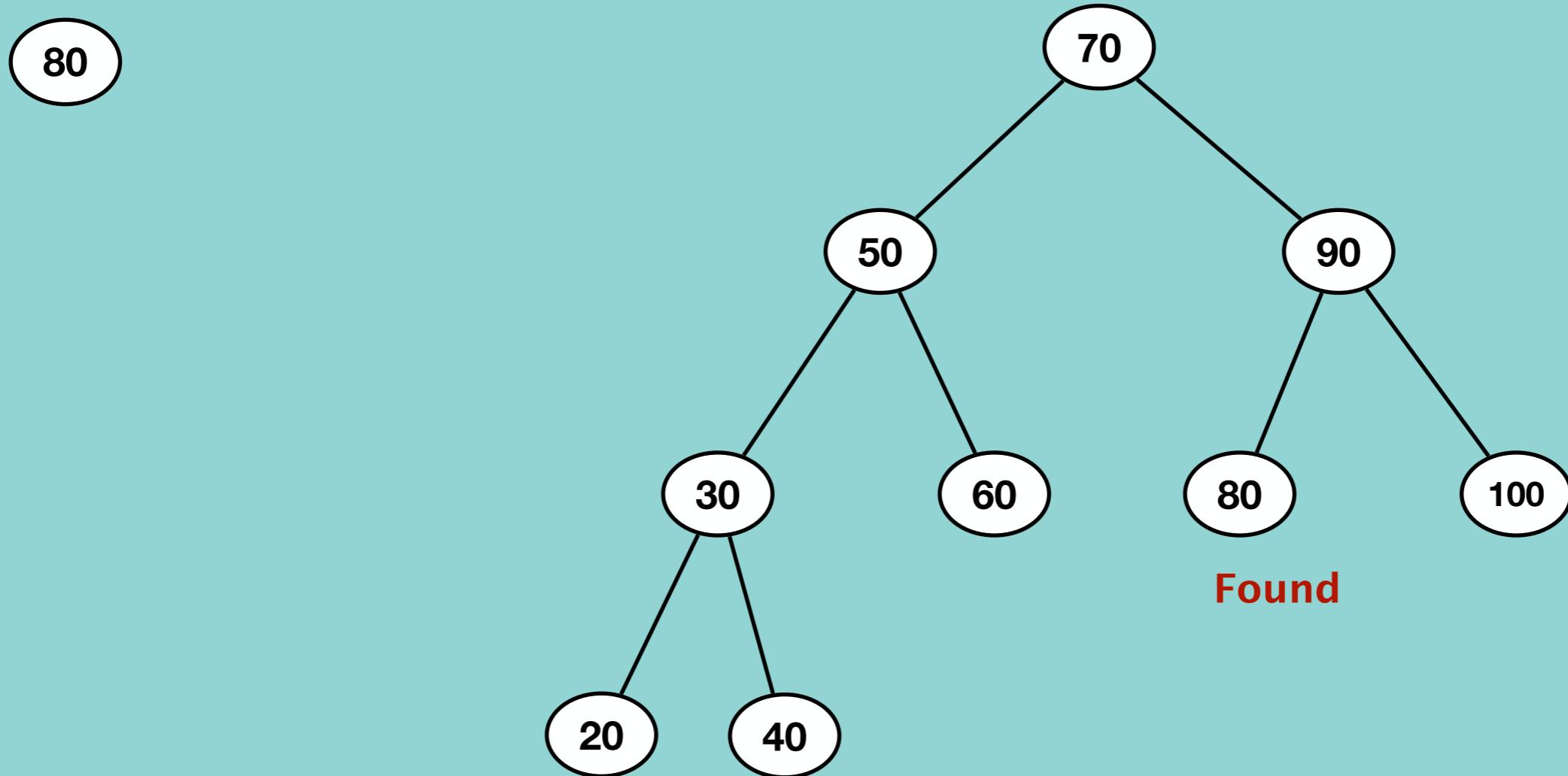
# LevelOrder Traversal of AVL Tree



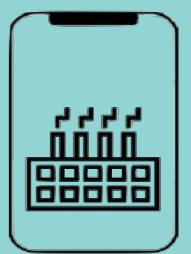
Time complexity :  $O(n)$   
Space complexity :  $O(n)$



# Search for a node in AVL Tree



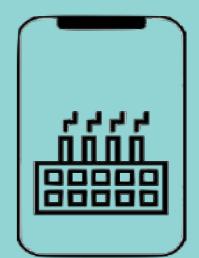
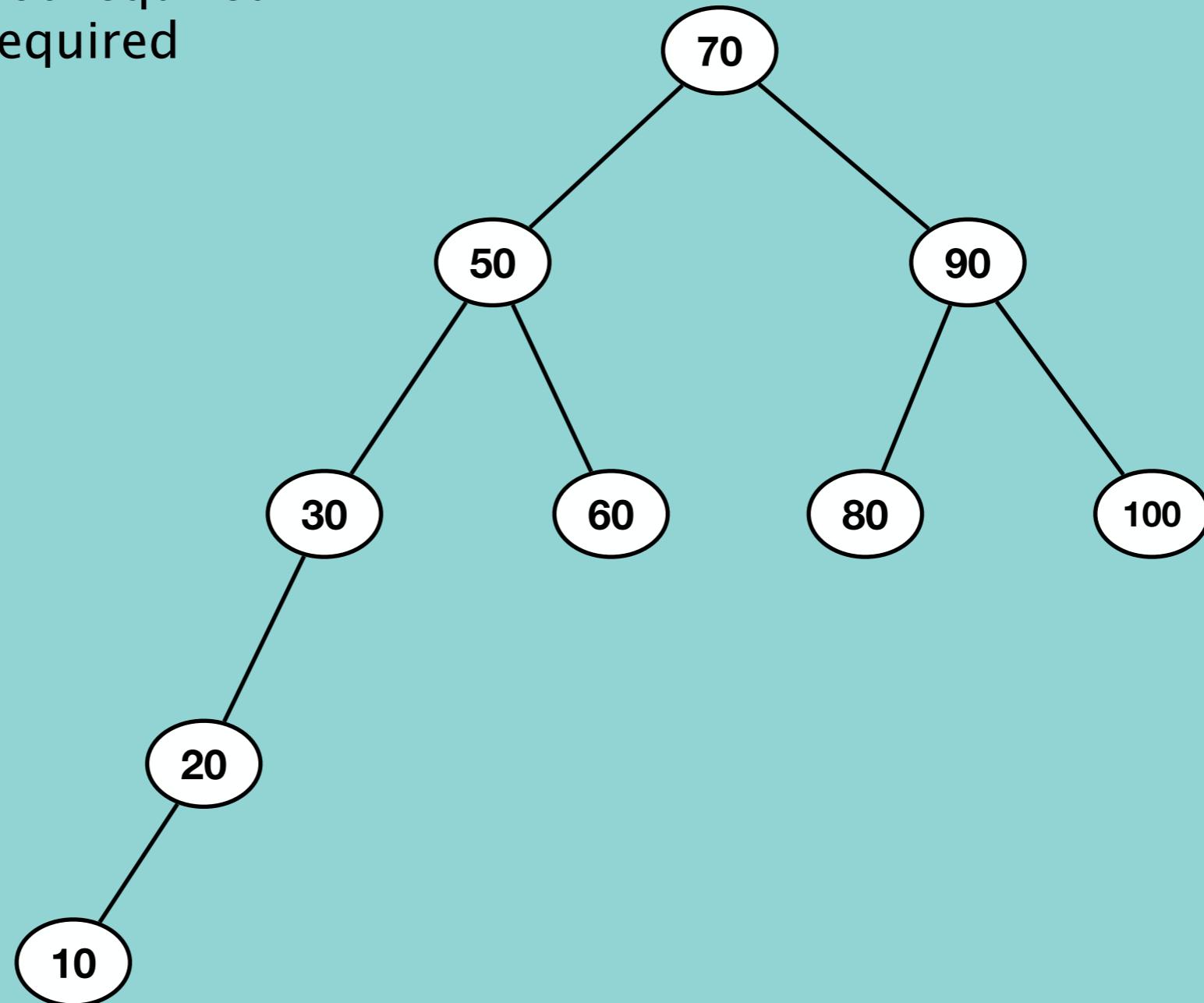
Time complexity : O(LogN)  
Space complexity : O(LogN)



# Insert a node in AVL Tree

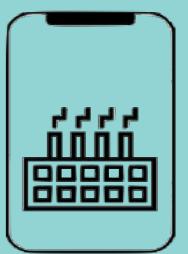
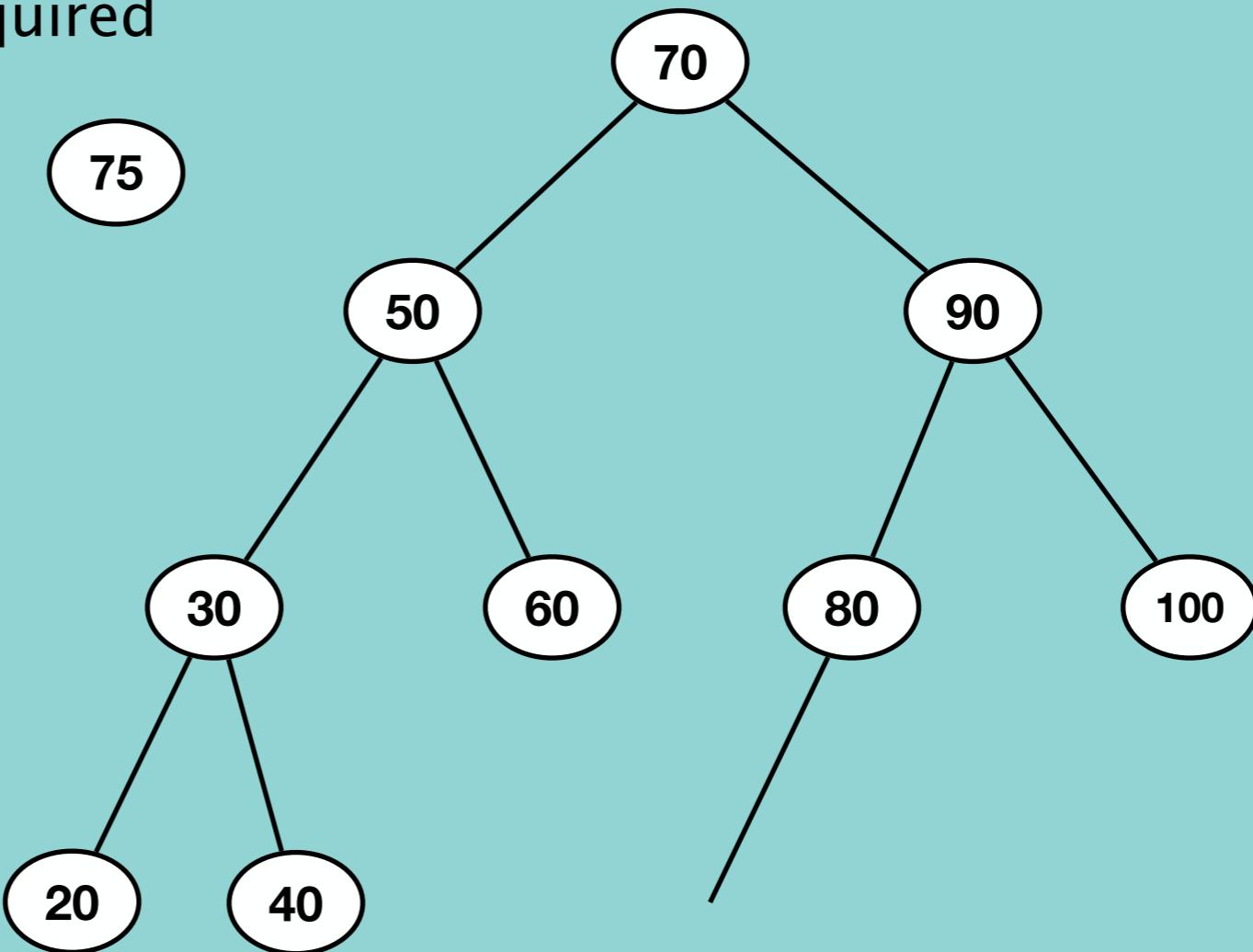
Case 1 : Rotation is not required

Case 2 : Rotation is required



# Insert a node in AVL Tree

Case 1 : Rotation is not required



# Insert a node in AVL Tree

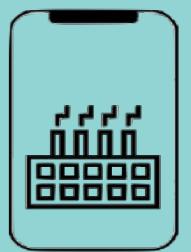
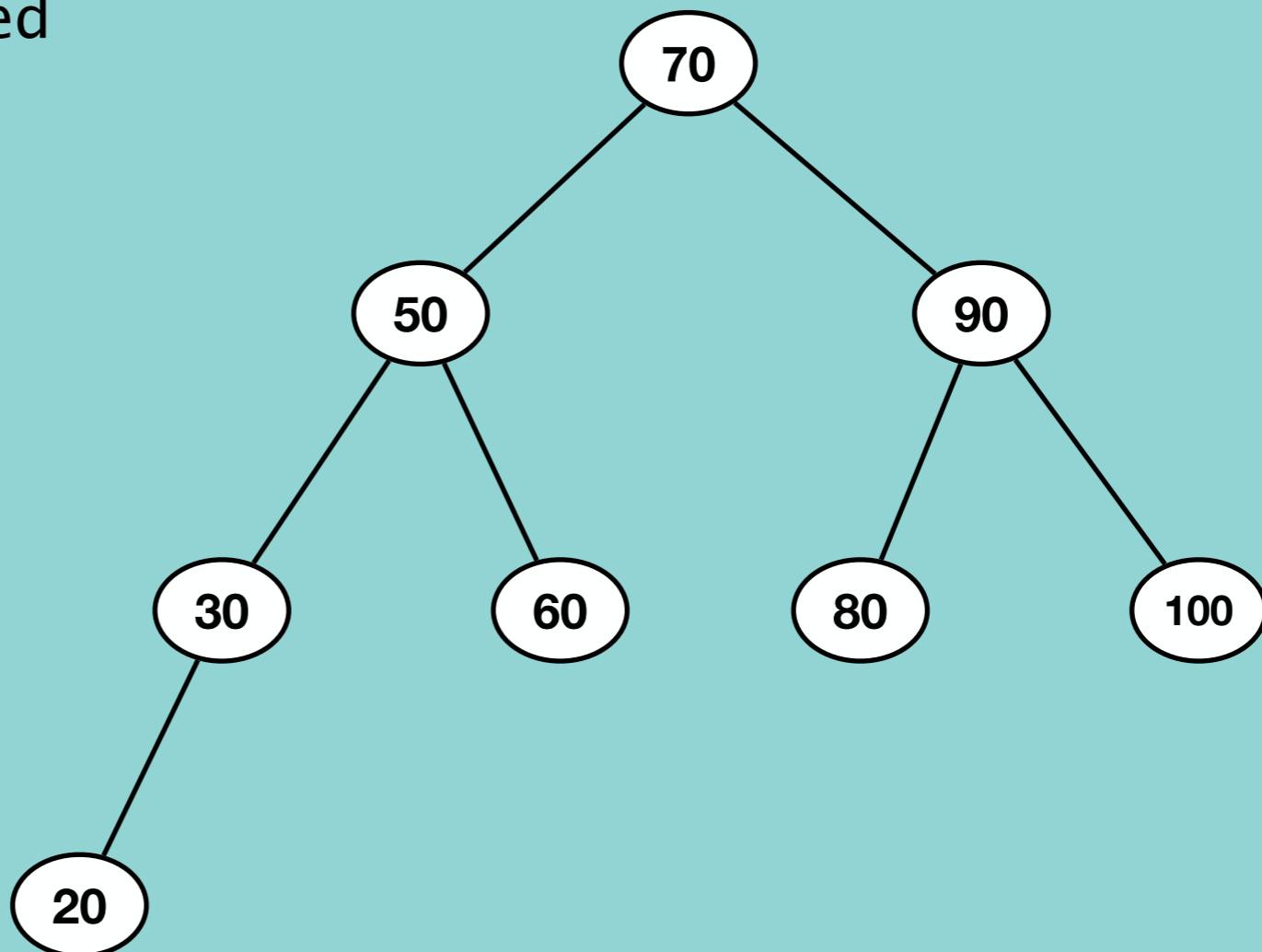
Case 2 : Rotation is required

LL - left left condition

LR - left right condition

RR - right right condition

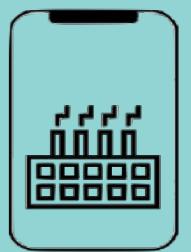
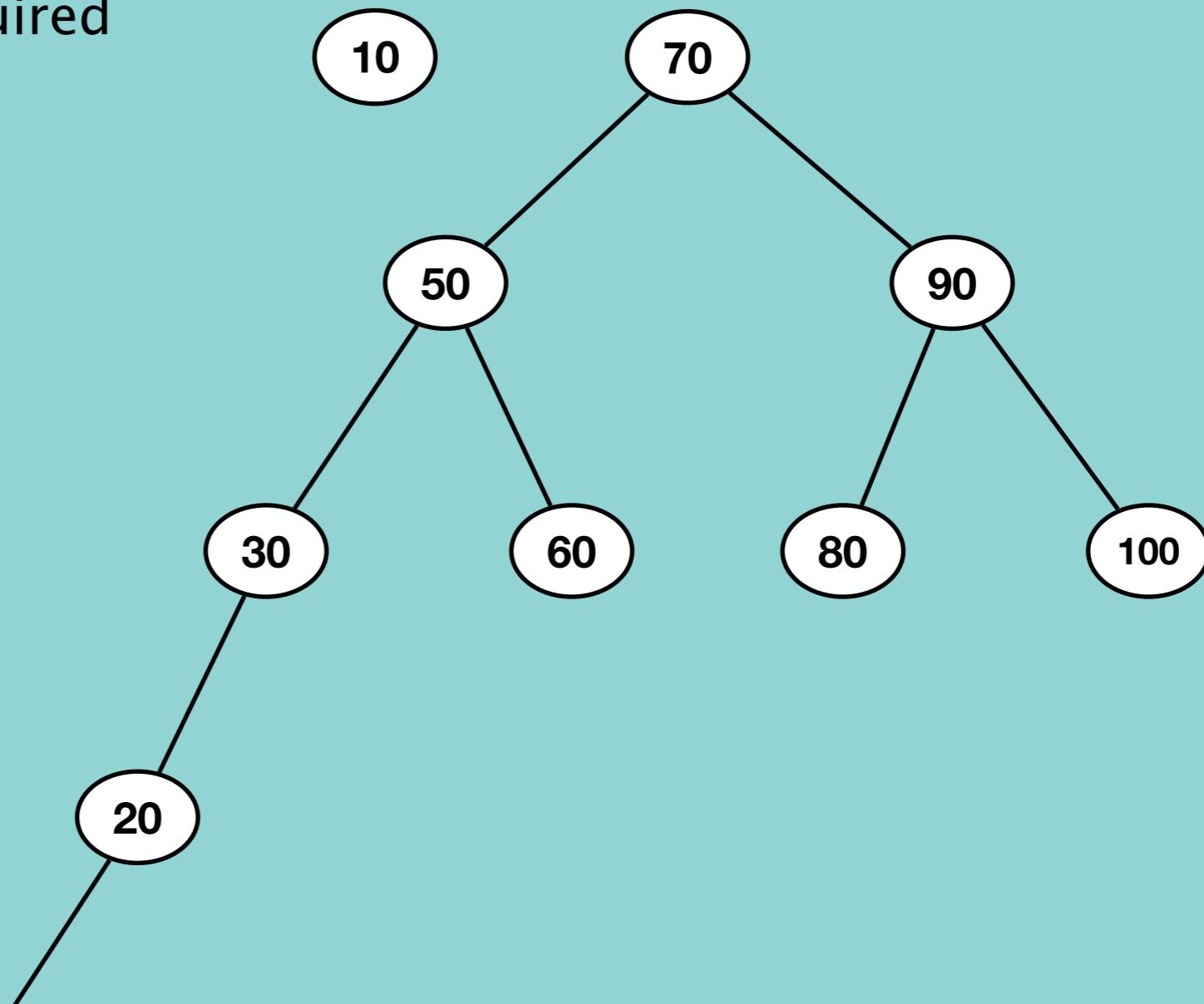
RL - right left condition



# Insert a node in AVL Tree

Case 2 : Rotation is required

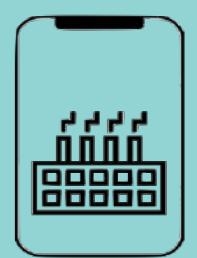
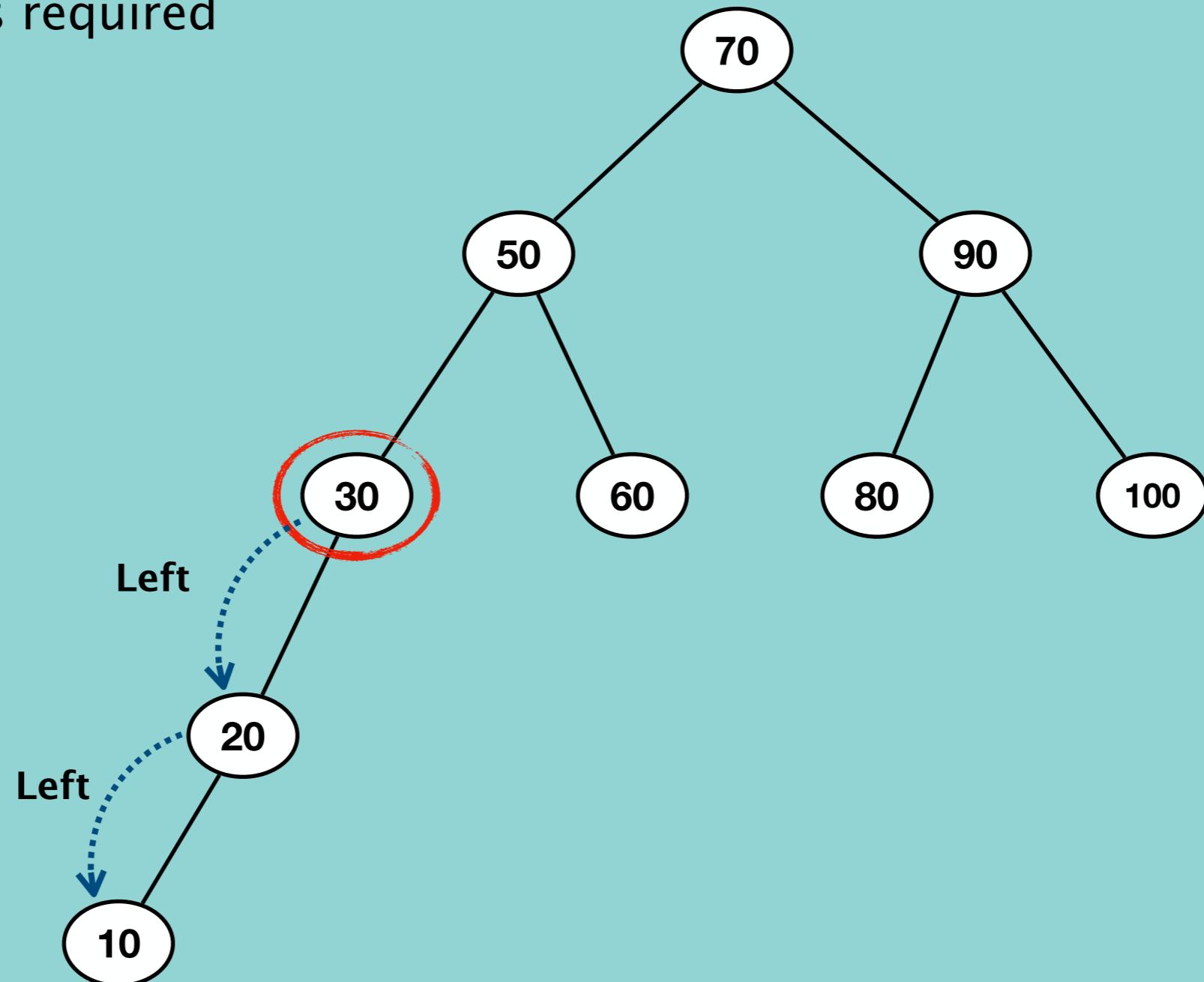
LL – left left condition



# Insert a node in AVL Tree

Case 2 : Rotation is required

LL – left left condition

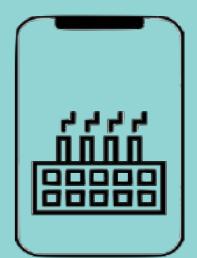
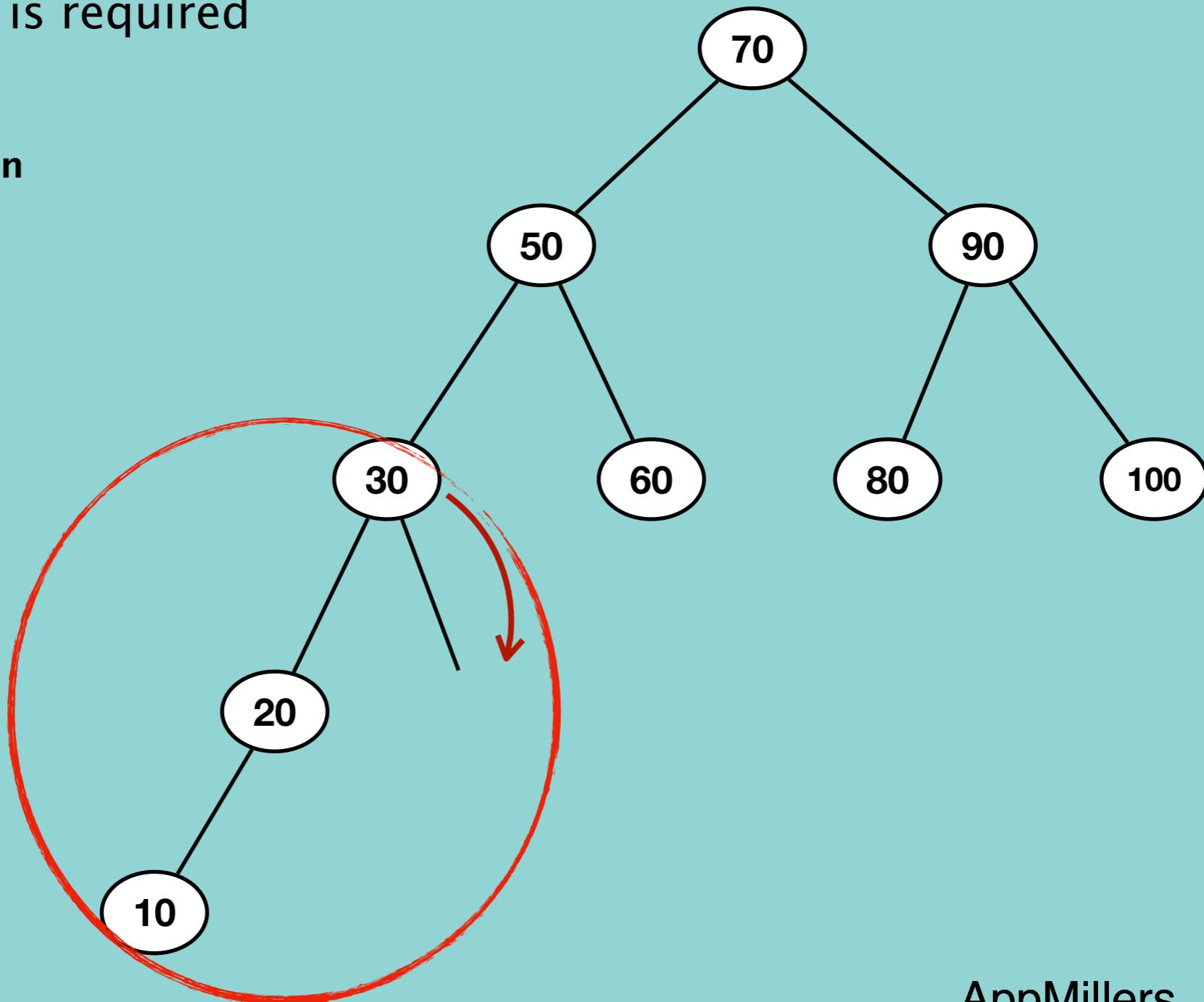


# Insert a node in AVL Tree

Case 2 : Rotation is required

LL – left left condition

Right rotation

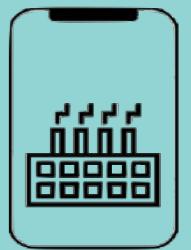
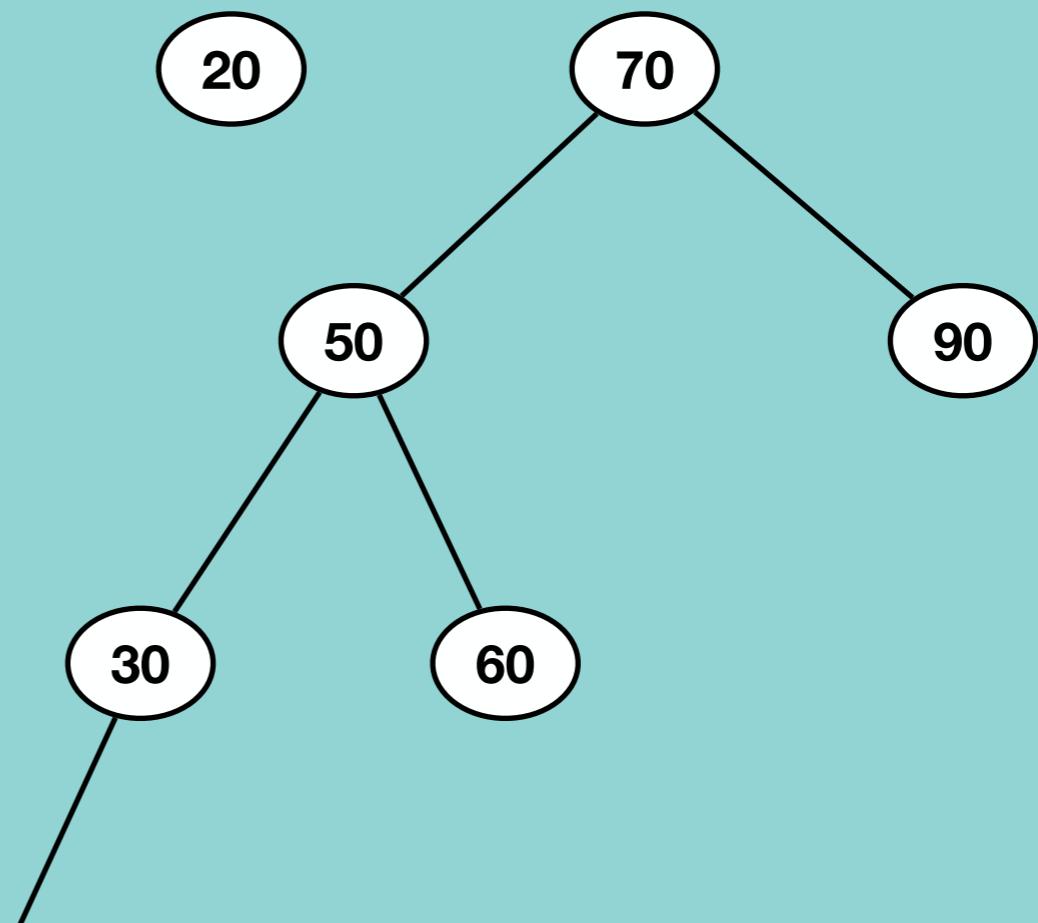


# Insert a node in AVL Tree

Case 2 : Rotation is required

**LL – left left condition**

**Right rotation – example 2**

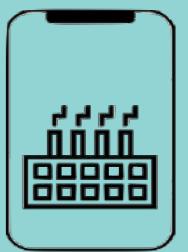
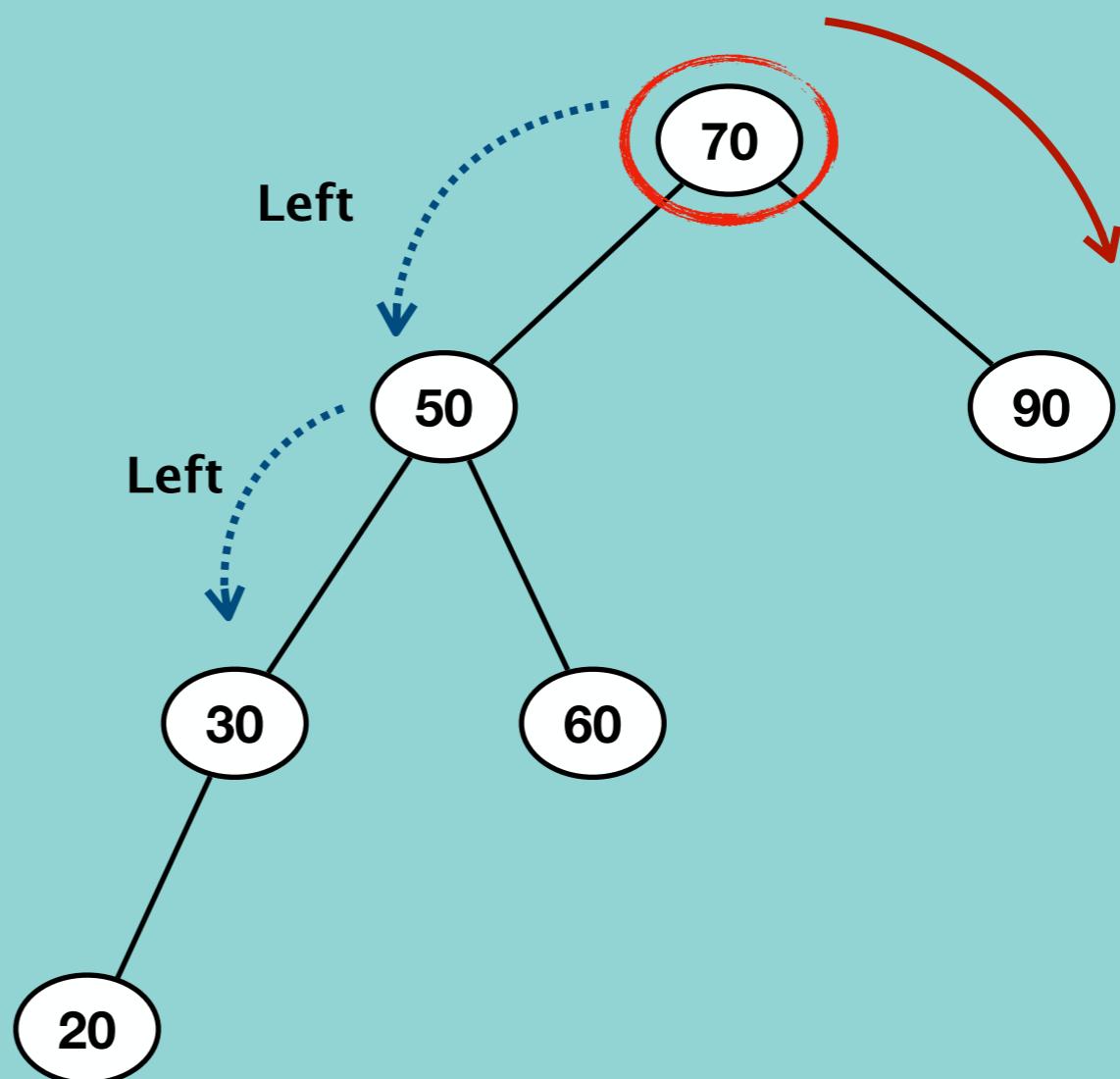


# Insert a node in AVL Tree

Case 2 : Rotation is required

LL – left left condition

Right rotation – example 2

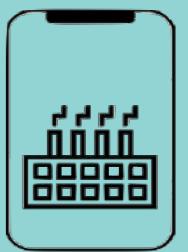
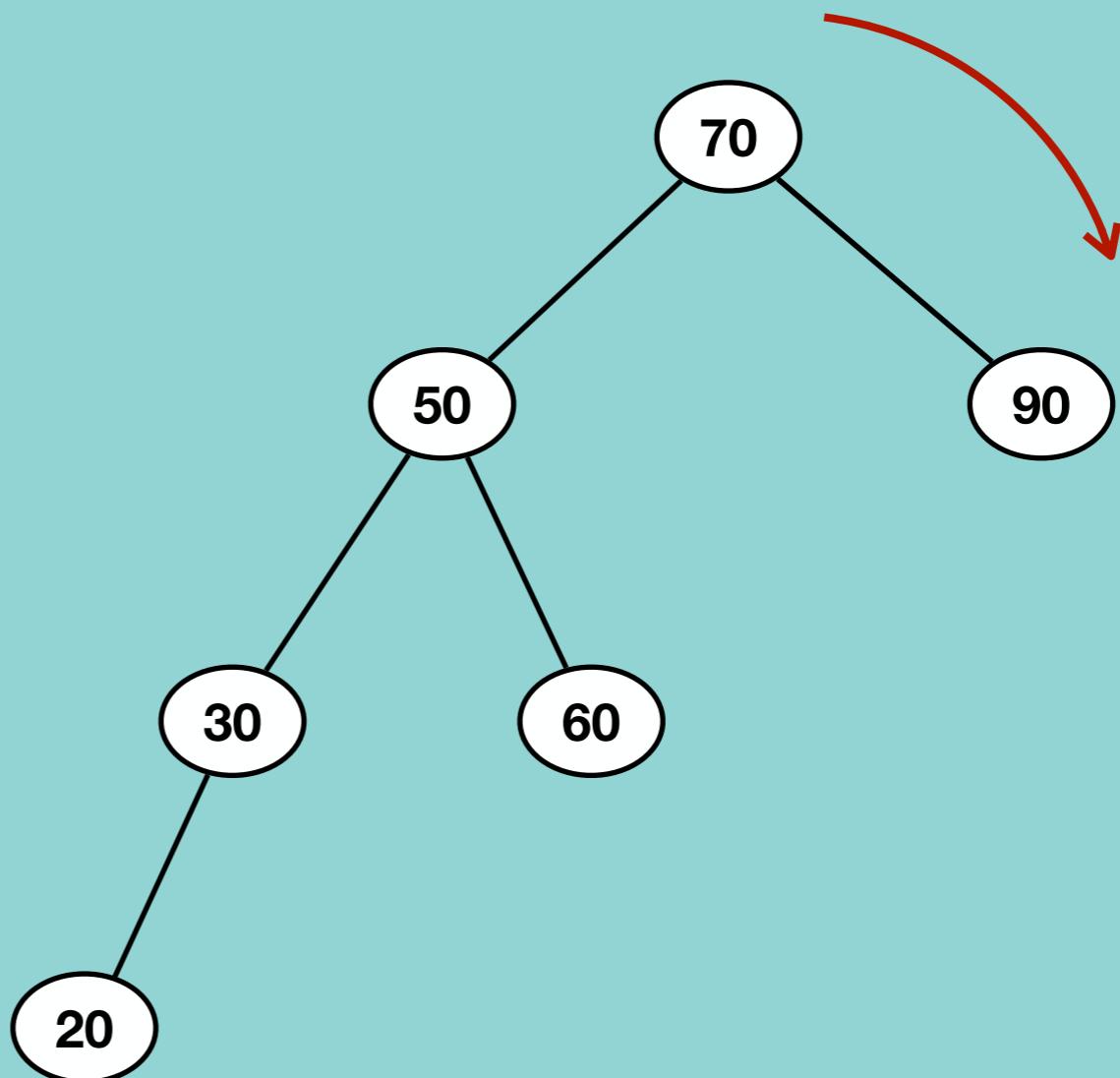


# Insert a node in AVL Tree

Case 2 : Rotation is required

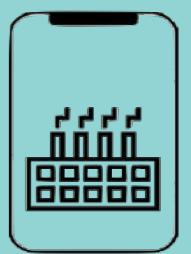
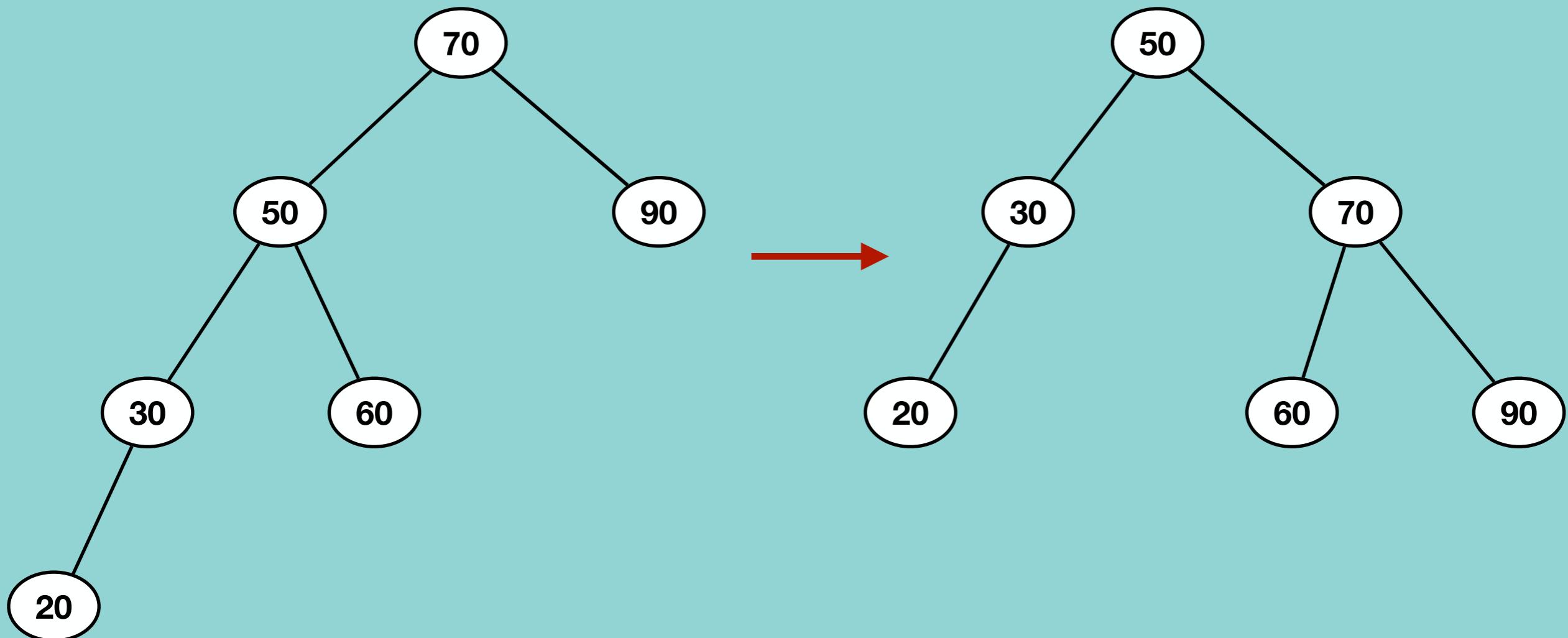
**LL – left left condition**

**Right rotation – example 2**



# Insert a node in AVL Tree

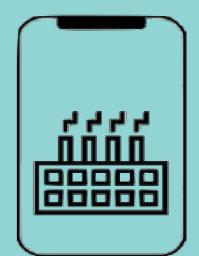
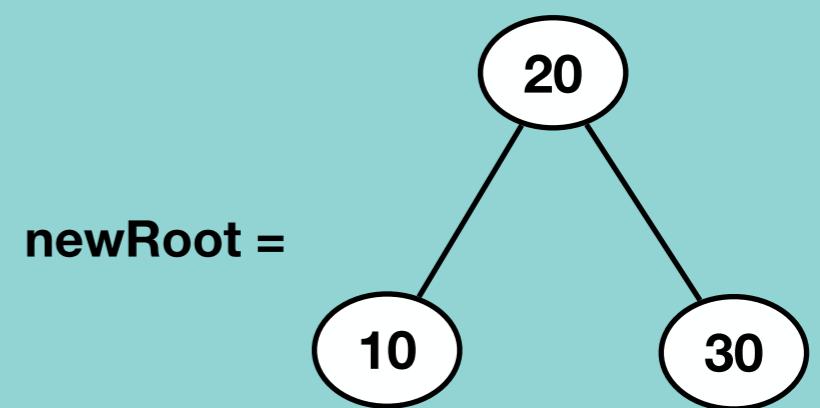
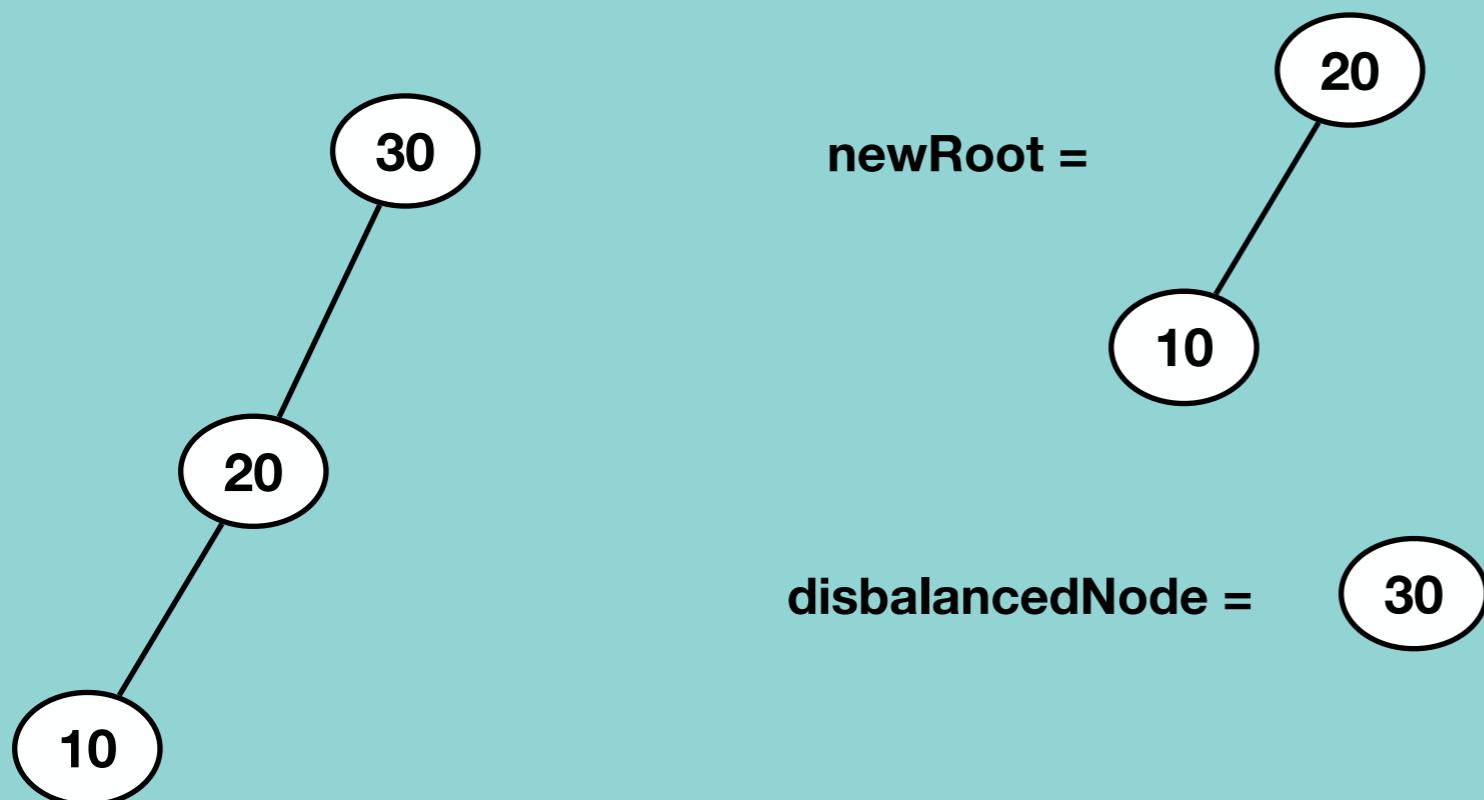
Right rotation – example 2



# Insert a node in AVL Tree

## Algorithm of Left Left (LL) Condition

```
rotateRight(disbalancedNode):
    newRoot = disbalancedNode.leftChild ← None
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
    newRoot.rightChild = disbalancedNode
    update height of disbalancedNode and newRoot
    return newRoot
```



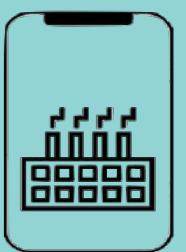
# Insert a node in AVL Tree

## Algorithm of Left Left (LL) Condition

```
rotateRight(disbalancedNode):
    newRoot = disbalancedNode.leftChild
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
    newRoot.rightChild = disbalancedNode
    update height of disbalancedNode and newRoot
    return newRoot
```

Time complexity : O(1)

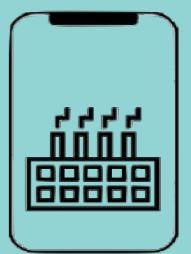
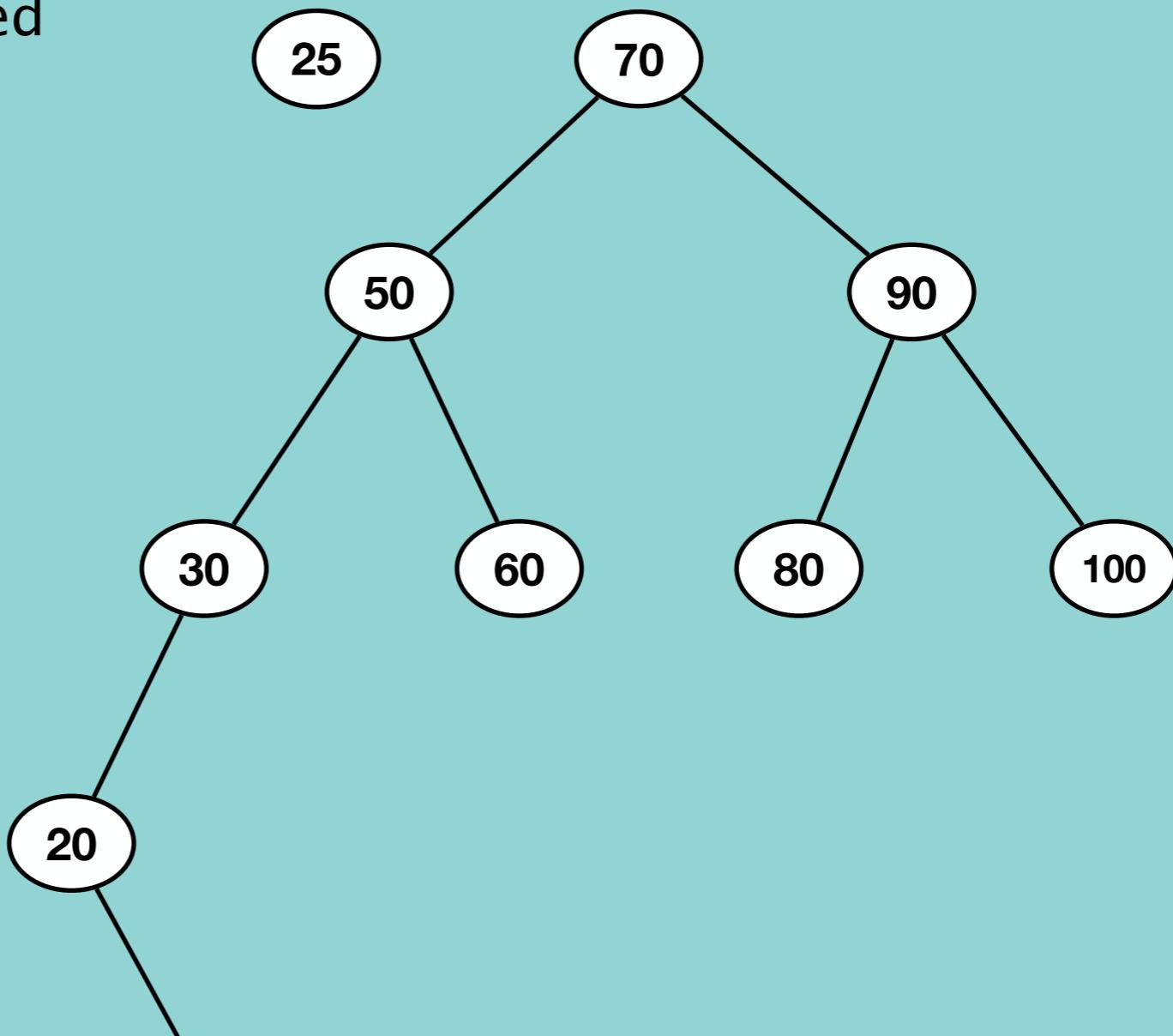
Space complexity : O(1)



# Insert a node in AVL Tree

Case 2 : Rotation is required

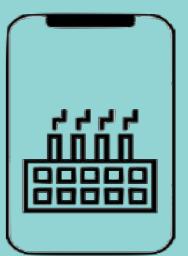
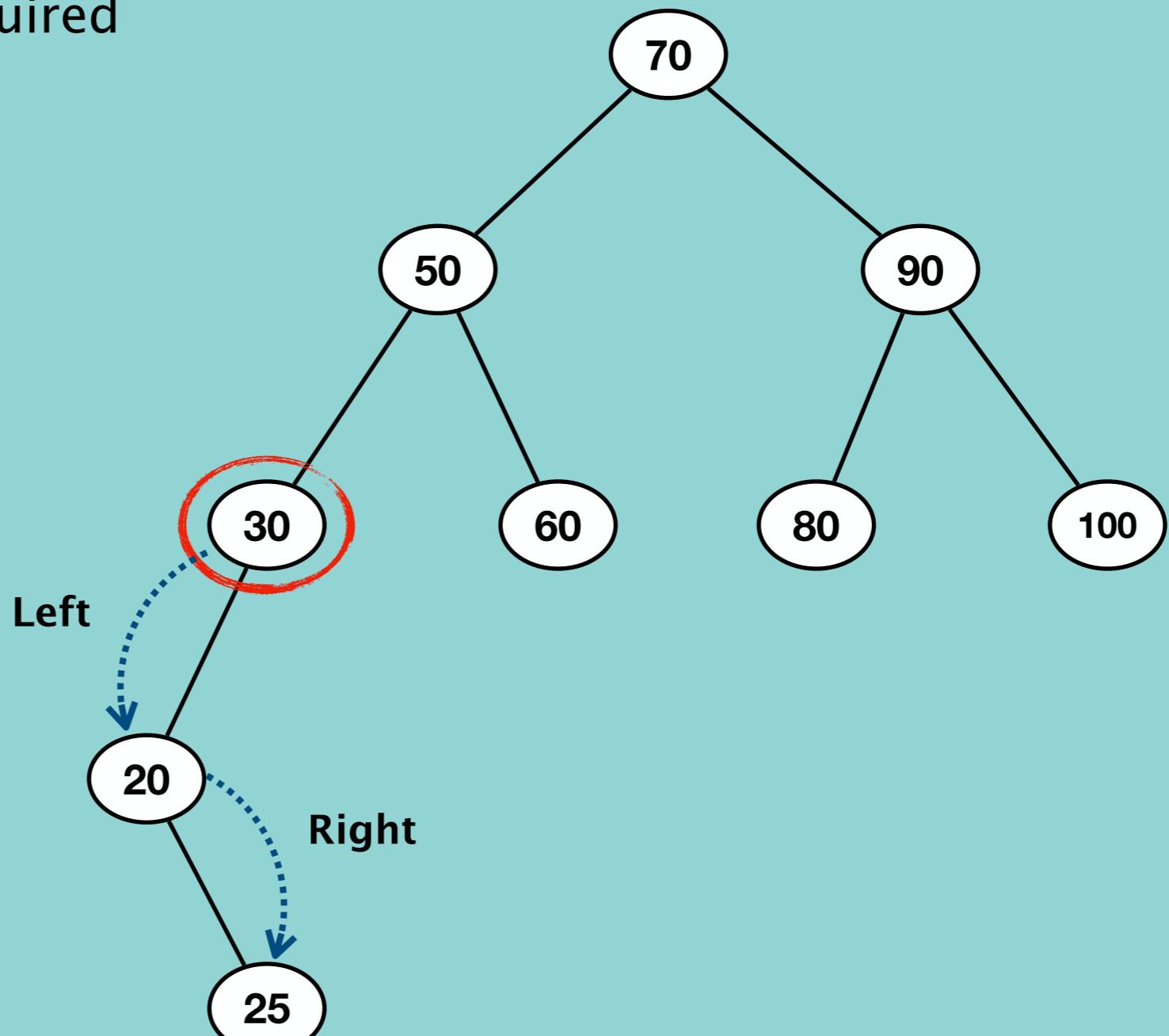
LR – left right condition



# Insert a node in AVL Tree

Case 2 : Rotation is required

LL – left right condition

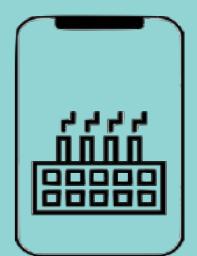
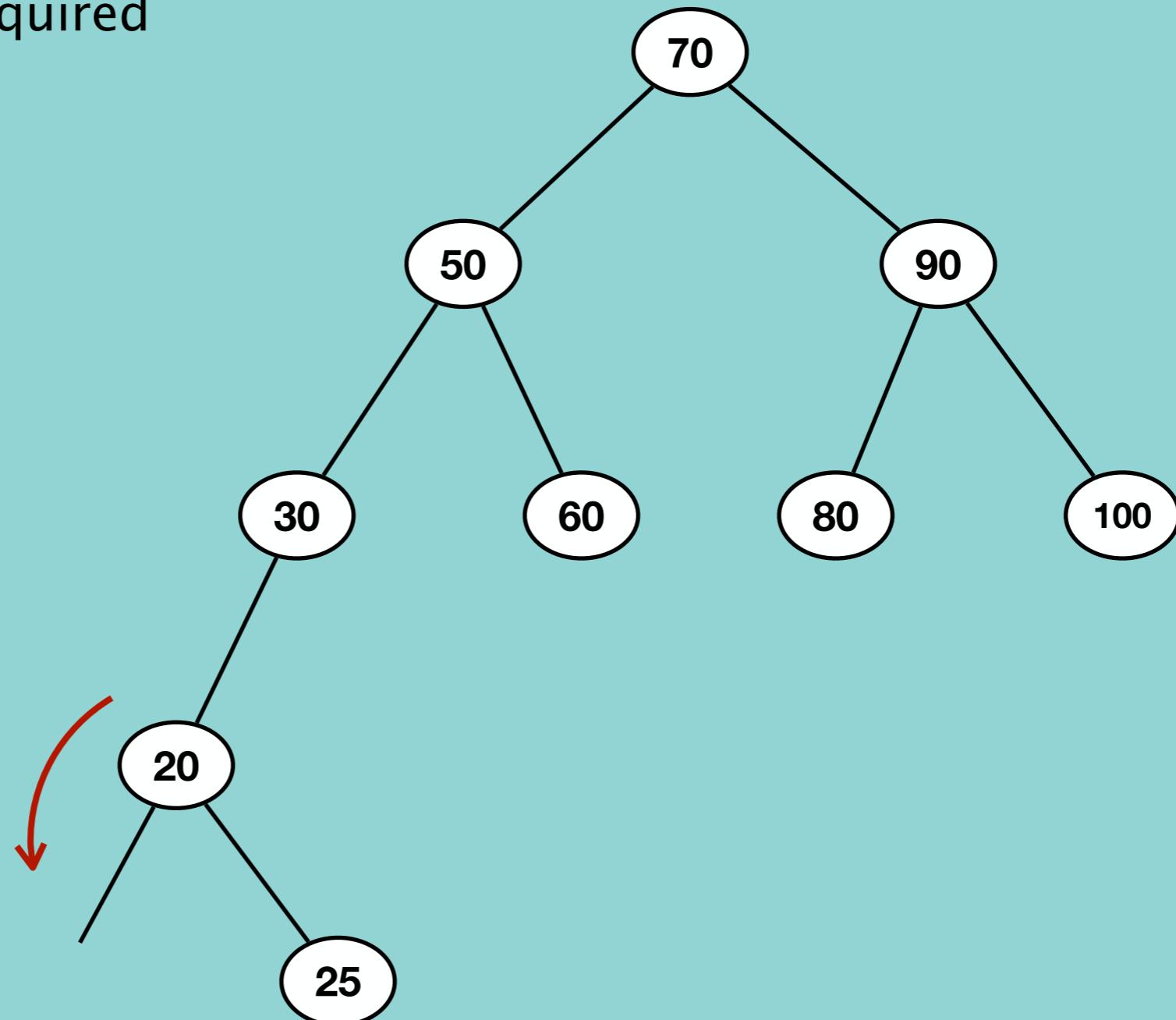


# Insert a node in AVL Tree

Case 2 : Rotation is required

**LR – left right condition**

1. Left rotation
2. Right rotation

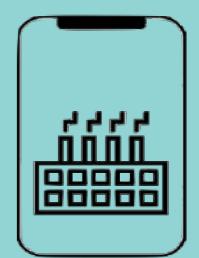
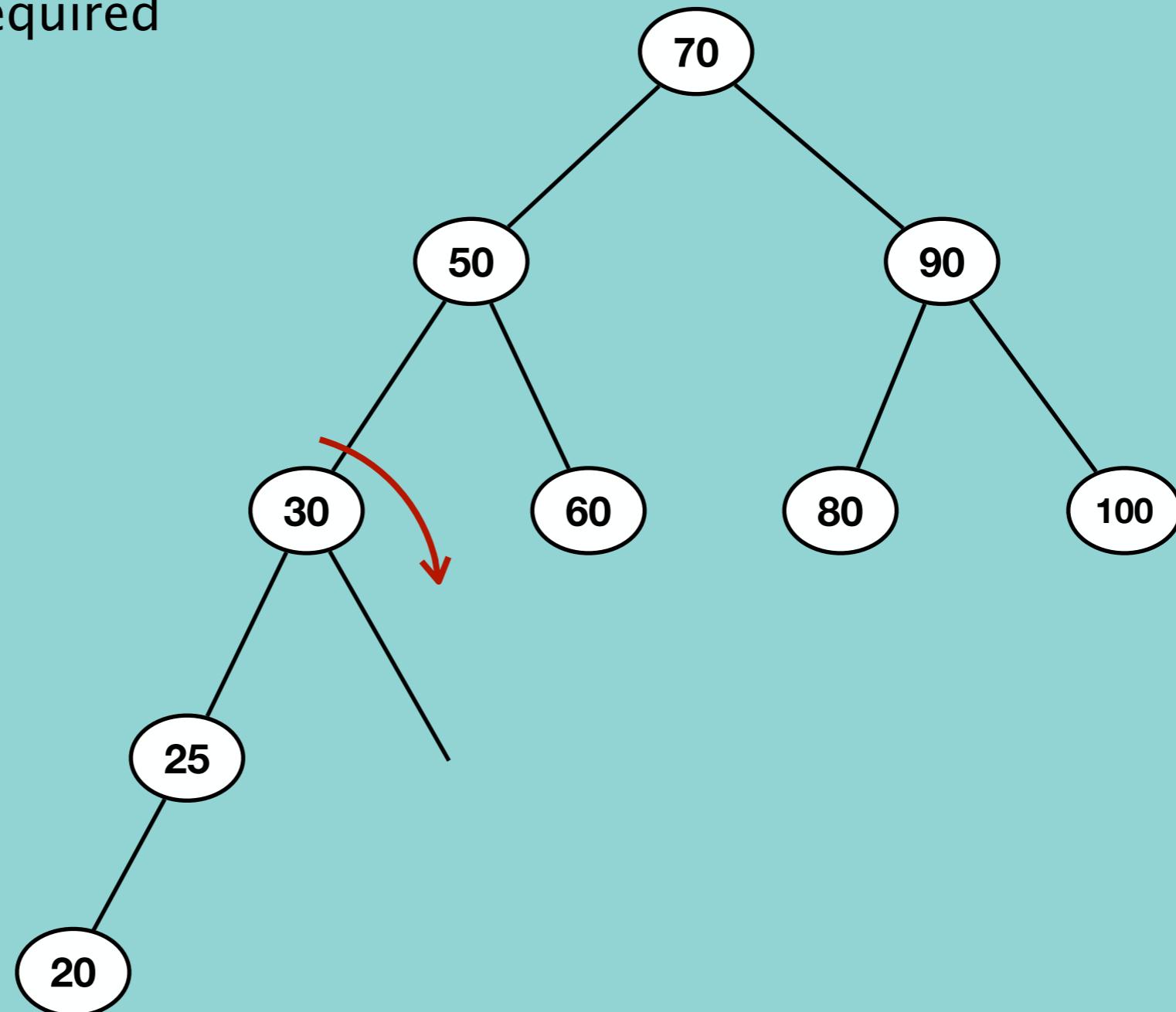


# Insert a node in AVL Tree

Case 2 : Rotation is required

**LR – left right condition**

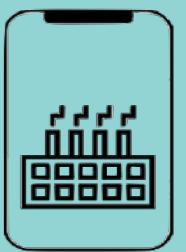
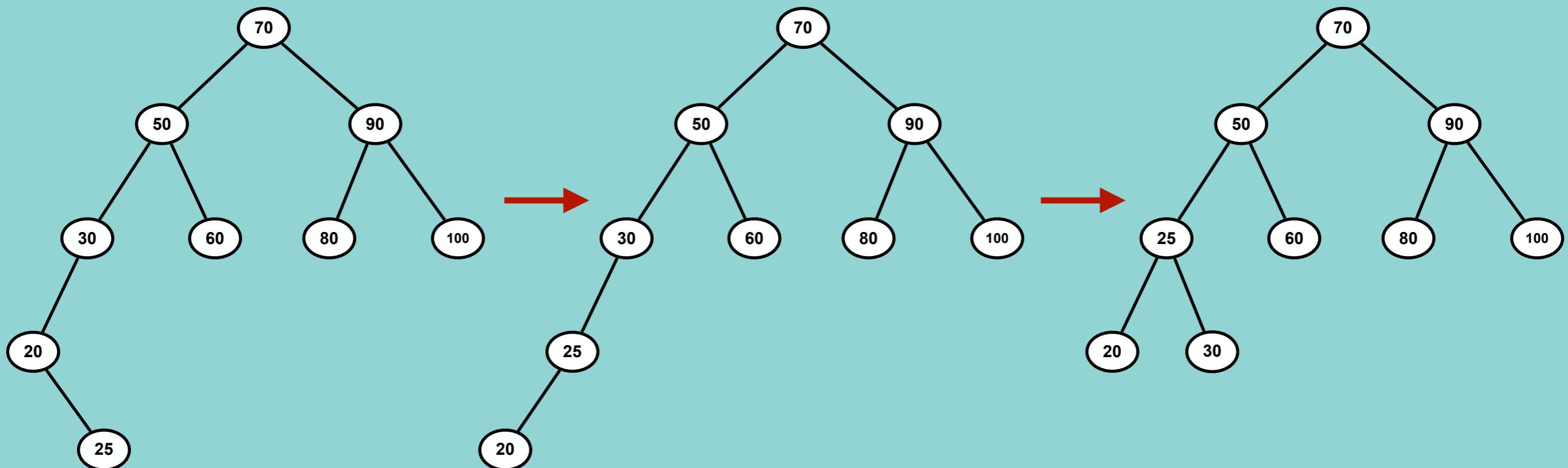
1. Left rotation
2. Right rotation



# Insert a node in AVL Tree

LR – left right condition

1. Left rotation
2. Right rotation



# Insert a node in AVL Tree

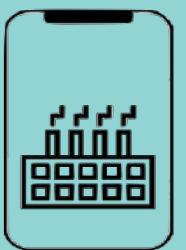
## Algorithm of Left Right (LR) Condition

Step 1 : rotate Left disbalancedNode.leftChild

Step 2 : rotate Right disbalancedNode

```
rotateLeft(disbalancedNode):
    newRoot = disbalancedNode.rightChild
    disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
    newRoot.leftChild = disbalancedNode
    update height of disbalancedNode and newRoot
    return newRoot
```

```
rotateRight(disbalancedNode):
    newRoot = disbalancedNode.leftChild
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
    newRoot.rightChild = disbalancedNode
    update height of disbalancedNode and newRoot
    return newRoot
```

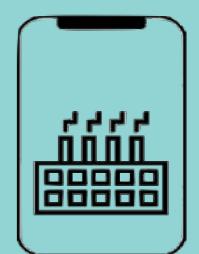
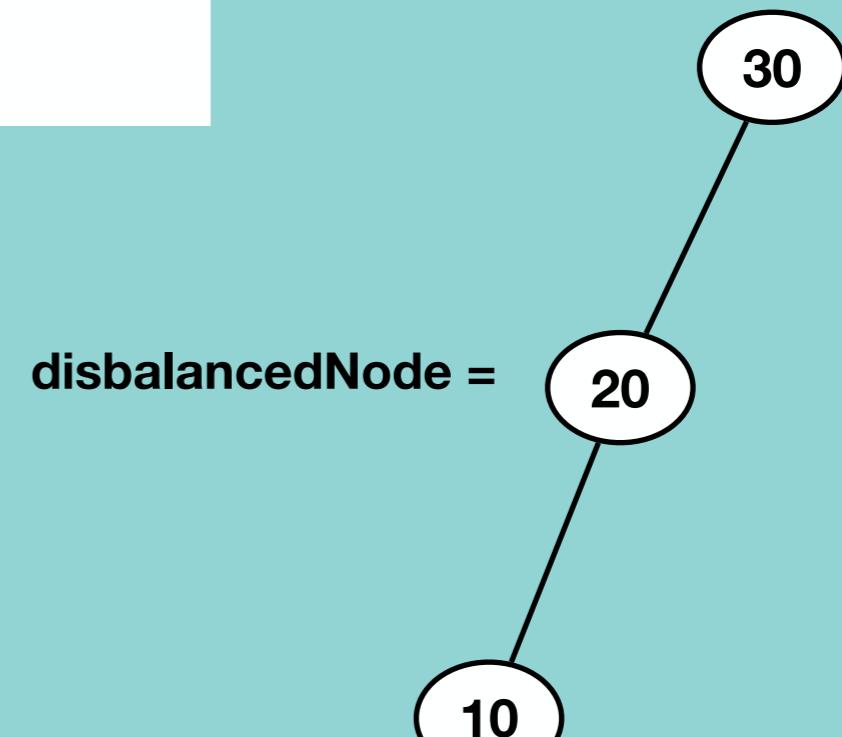
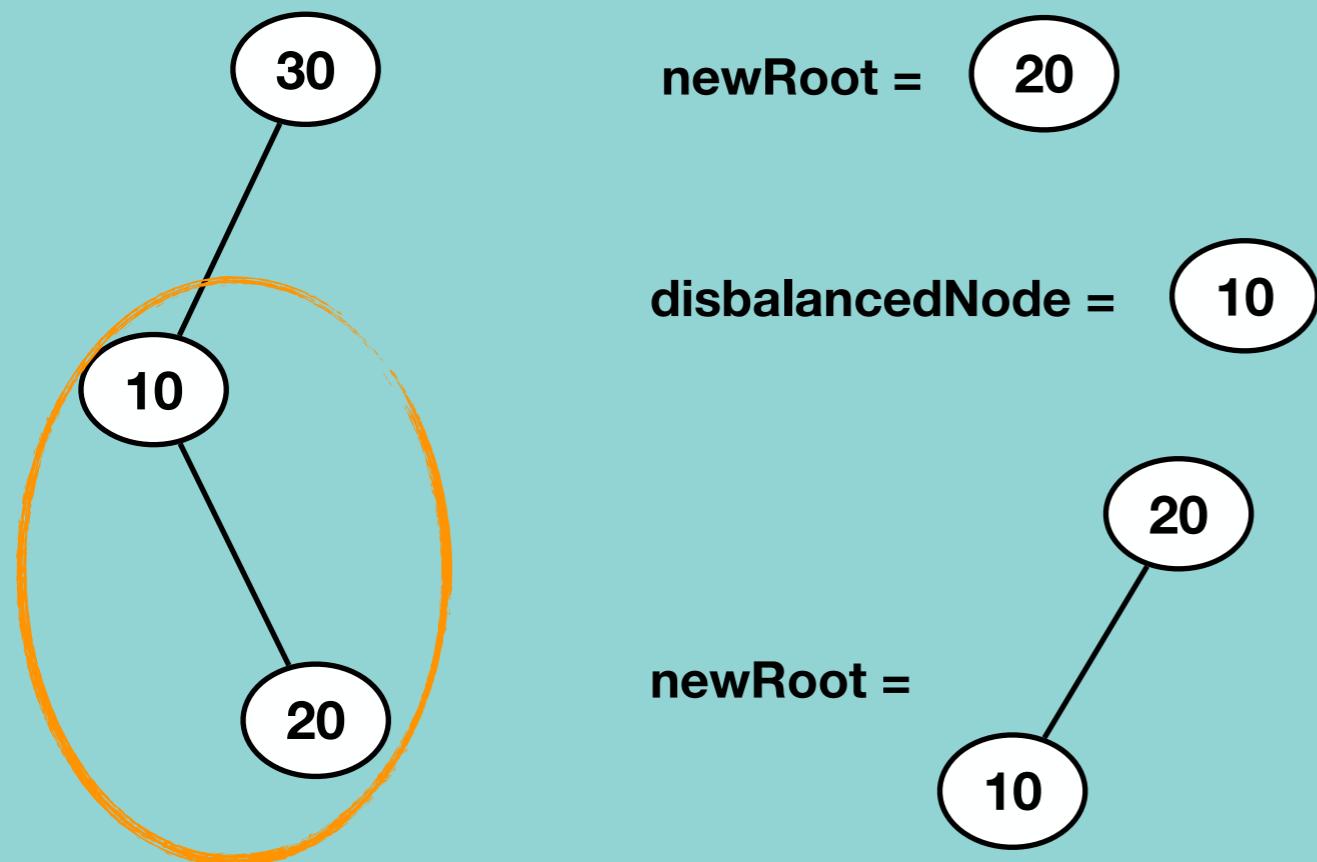


# Insert a node in AVL Tree

## Algorithm of Left Right (LR) Condition

Step 1 : rotate Left disbalancedNode.leftChild  
Step 2 : rotate Right disbalancedNode

```
rotateLeft(disbalancedNode):  
    newRoot = disbalancedNode.rightChild  
    disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild  
    newRoot.leftChild = disbalancedNode  
    update height of disbalancedNode and newRoot  
    return newRoot
```



# Insert a node in AVL Tree

## Algorithm of Left Right (LR) Condition

Step 1 : rotate Left disbalancedNode.leftChild

Step 2 : rotate Right disbalancedNode

```
rotateRight(disbalancedNode):
```

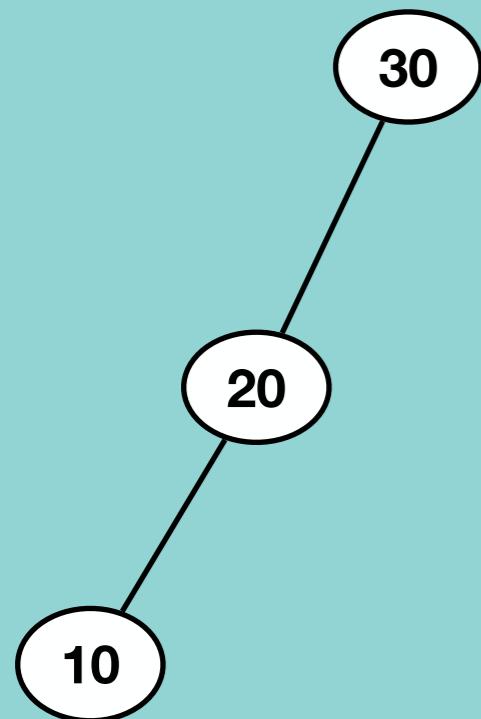
```
    newRoot = disbalancedNode.leftChild
```

```
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
```

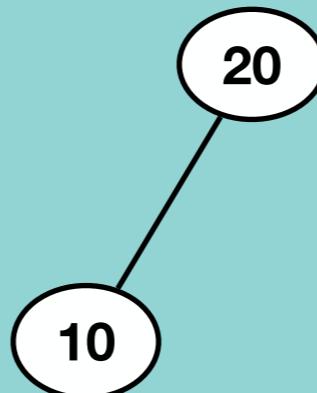
```
    newRoot.rightChild = disbalancedNode
```

```
    update height of disbalancedNode and newRoot
```

```
    return newRoot
```



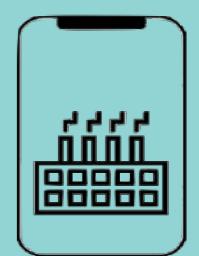
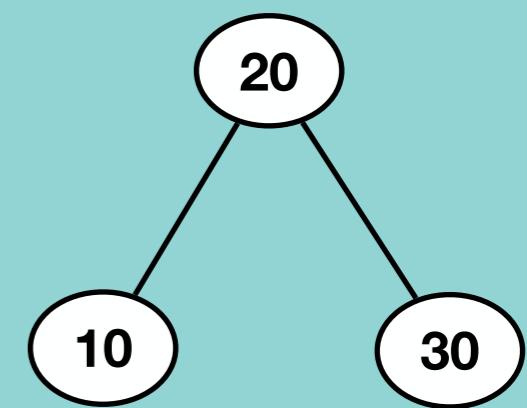
**newRoot =**



**disbalancedNode =**



**newRoot =**



# Insert a node in AVL Tree

## Algorithm of Left Right (LR) Condition

Step 1 : rotate Left disbalancedNode.leftChild

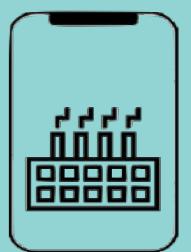
Step 2 : rotate Right disbalancedNode

```
rotateLeft(disbalancedNode):
    newRoot = disbalancedNode.rightChild
    disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
    newRoot.leftChild = disbalancedNode
    update height of disbalancedNode and newRoot
    return newRoot
```

```
rotateRight(disbalancedNode):
    newRoot = disbalancedNode.leftChild
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
    newRoot.rightChild = disbalancedNode
    update height of disbalancedNode and newRoot
    return newRoot
```

Time complexity : O(1)

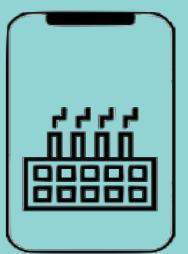
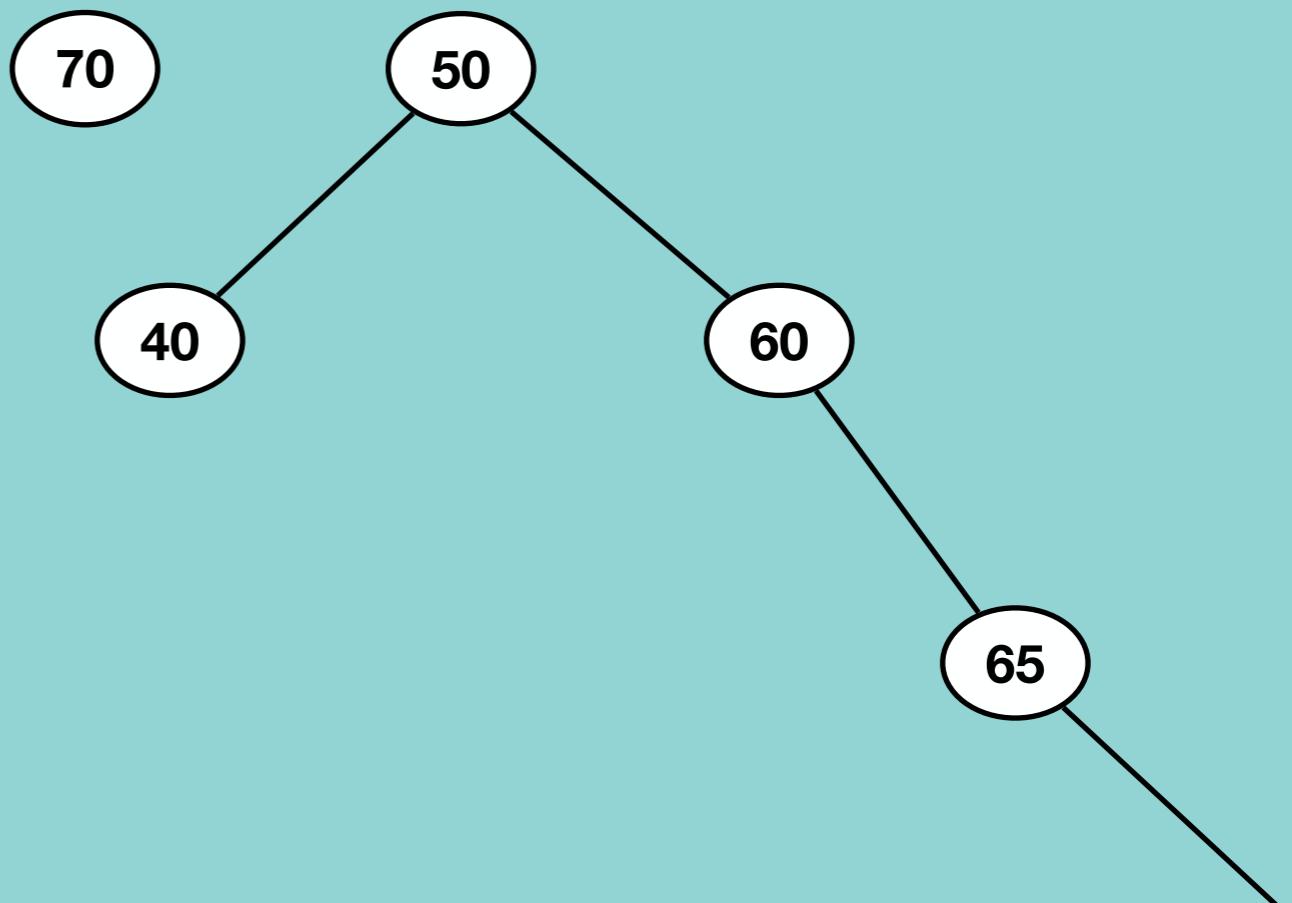
Space complexity : O(1)



# Insert a node in AVL Tree

Case 2 : Rotation is required

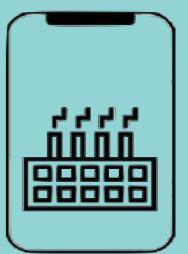
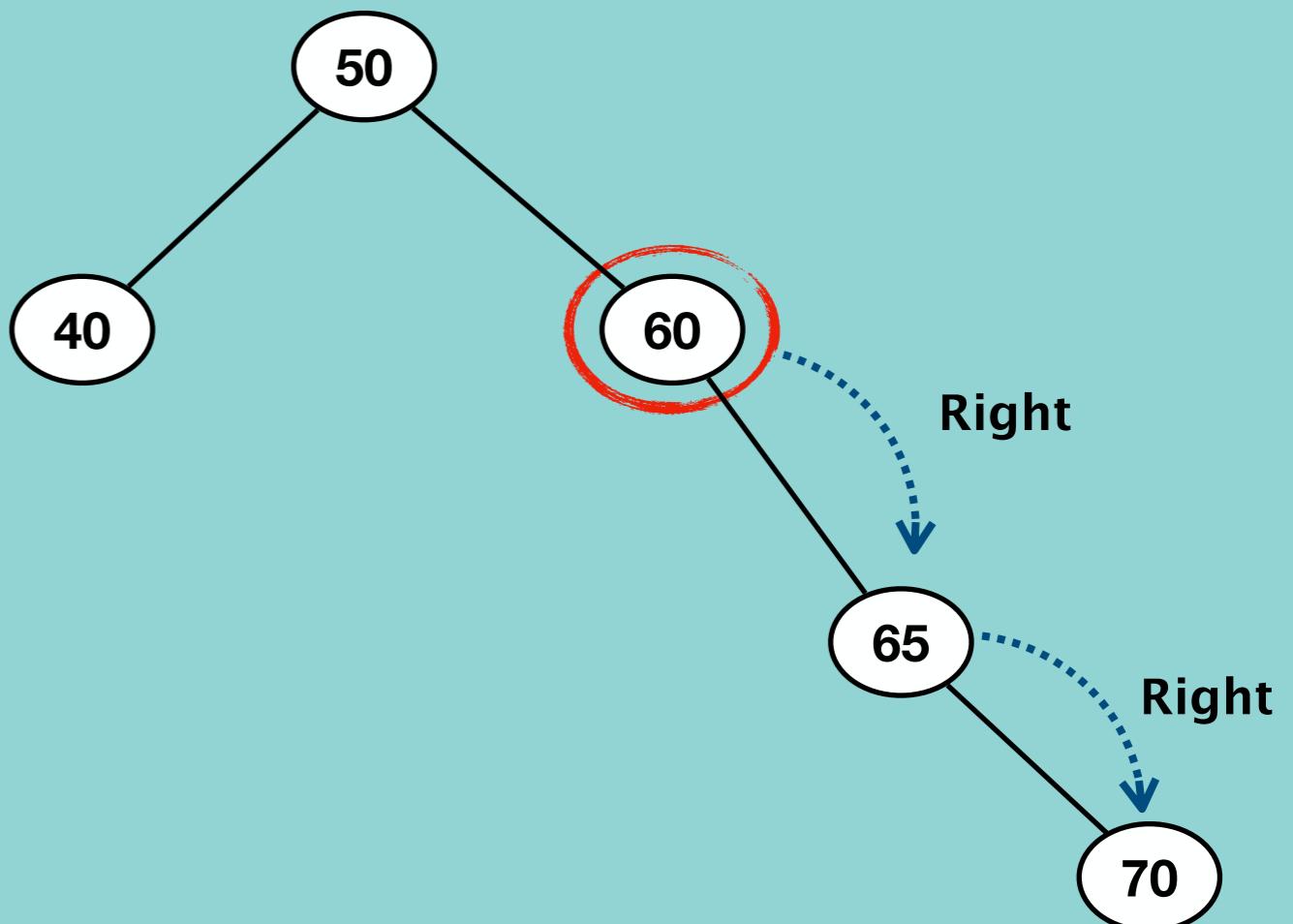
RR – right right condition



# Insert a node in AVL Tree

Case 2 : Rotation is required

RR – right right condition

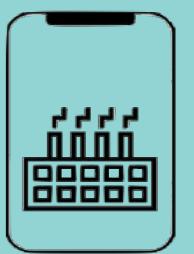
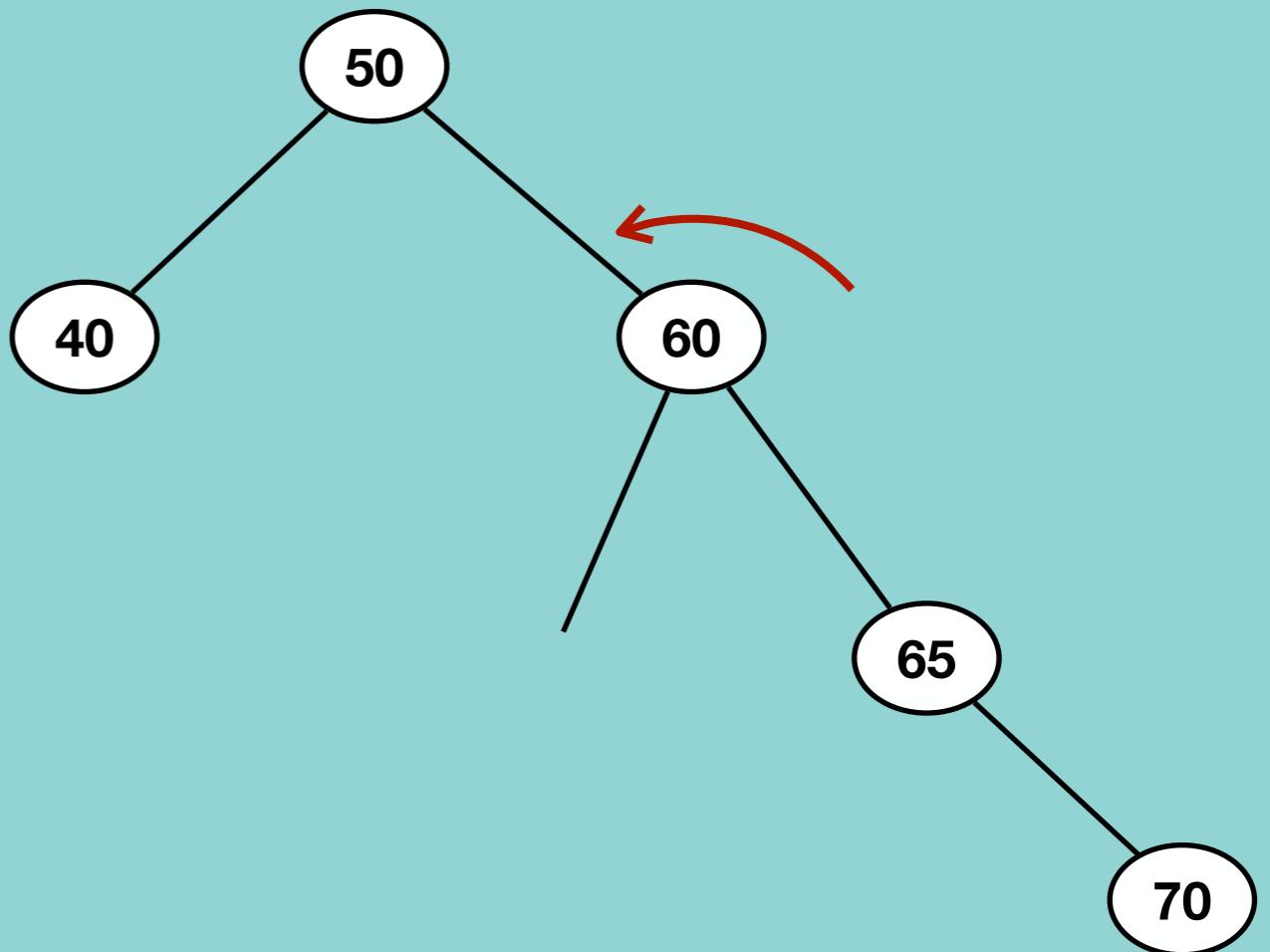


# Insert a node in AVL Tree

Case 2 : Rotation is required

**RR – right right condition**

**Left rotation**

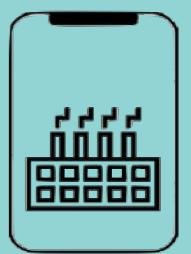
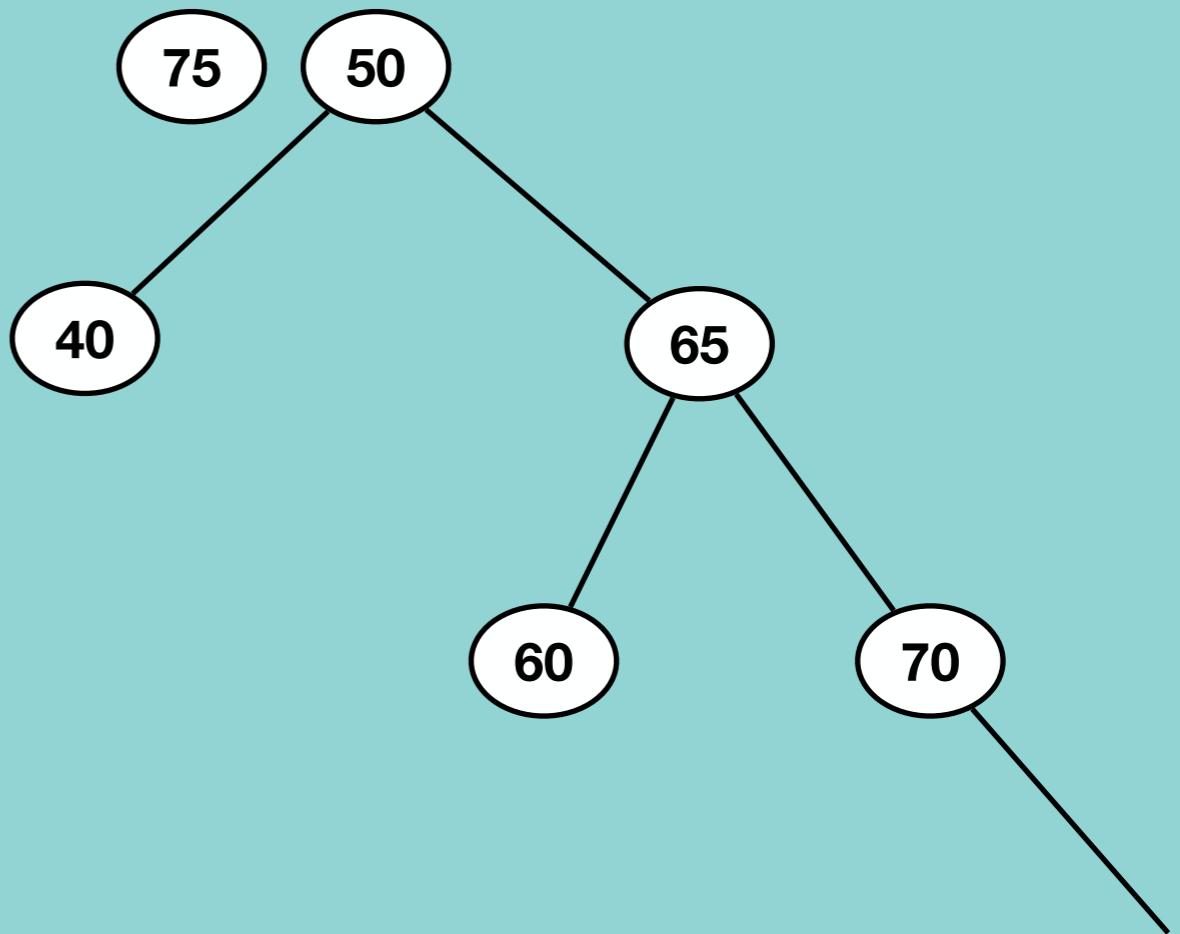


# Insert a node in AVL Tree

Case 2 : Rotation is required

RR – right right condition

Left rotation – example 2

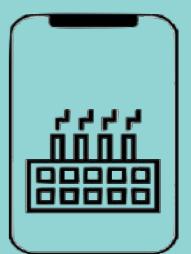
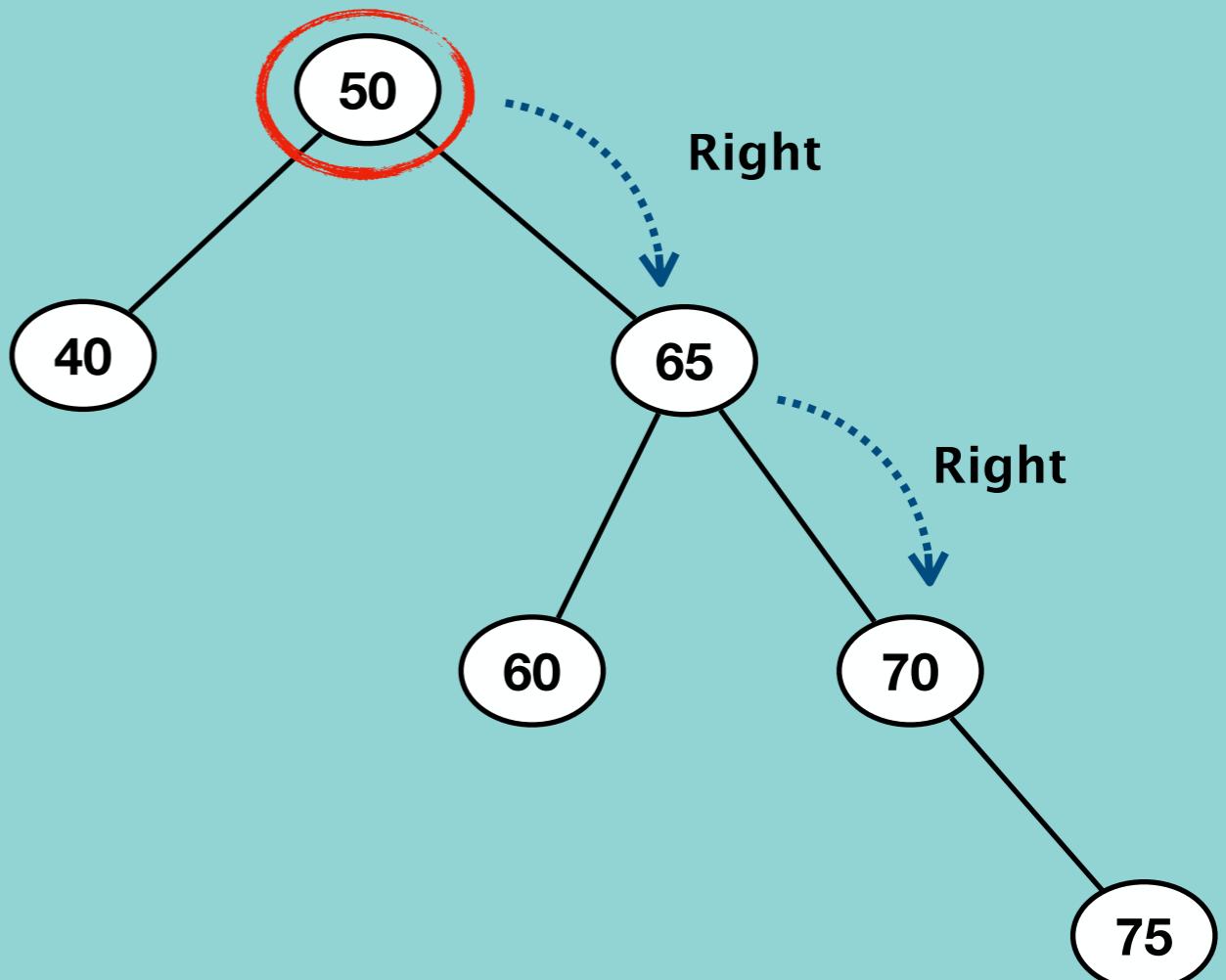


# Insert a node in AVL Tree

Case 2 : Rotation is required

RR – right right condition

Left rotation – example 2

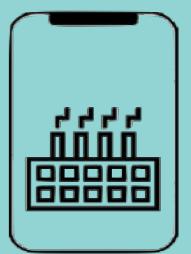
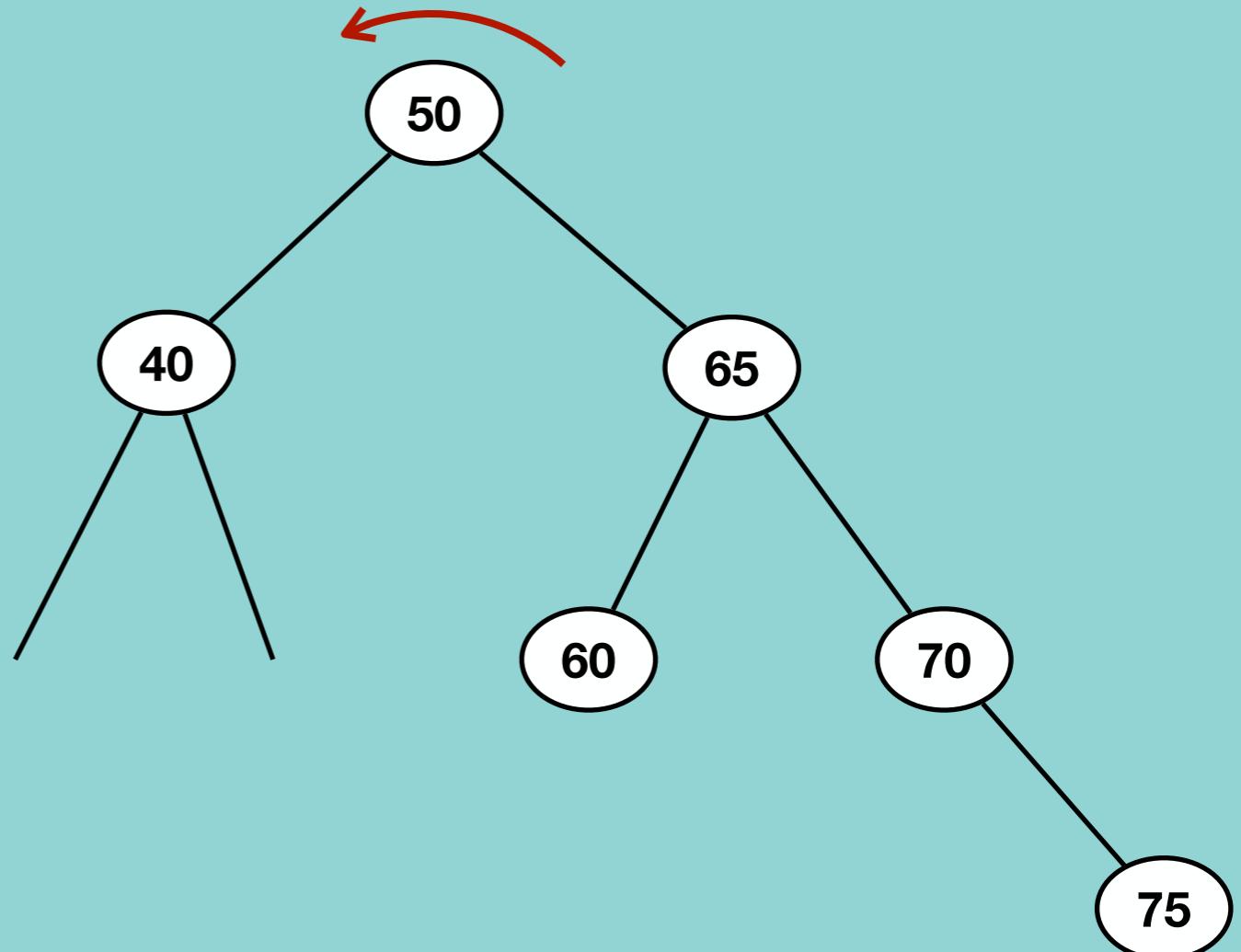


# Insert a node in AVL Tree

Case 2 : Rotation is required

RR – right right condition

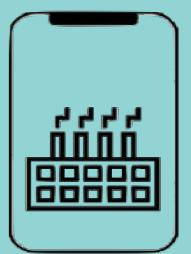
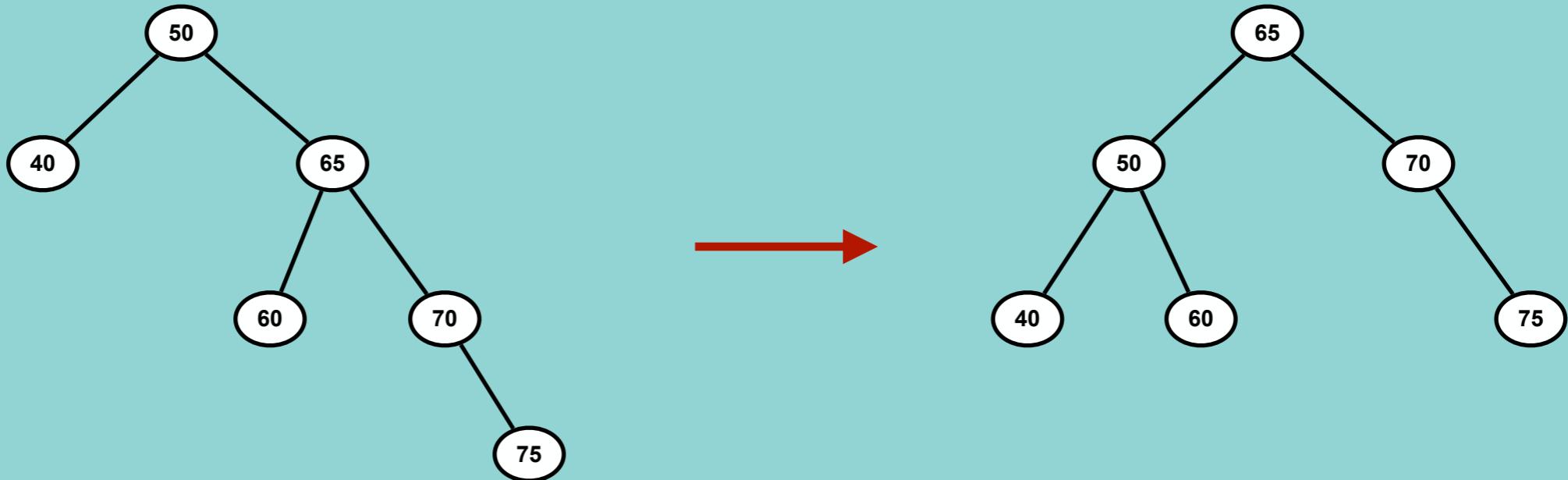
Left rotation – example 2



# Insert a node in AVL Tree

RR – right right condition

Left rotation

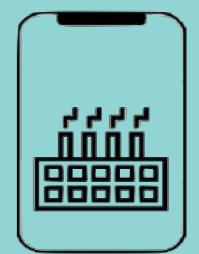
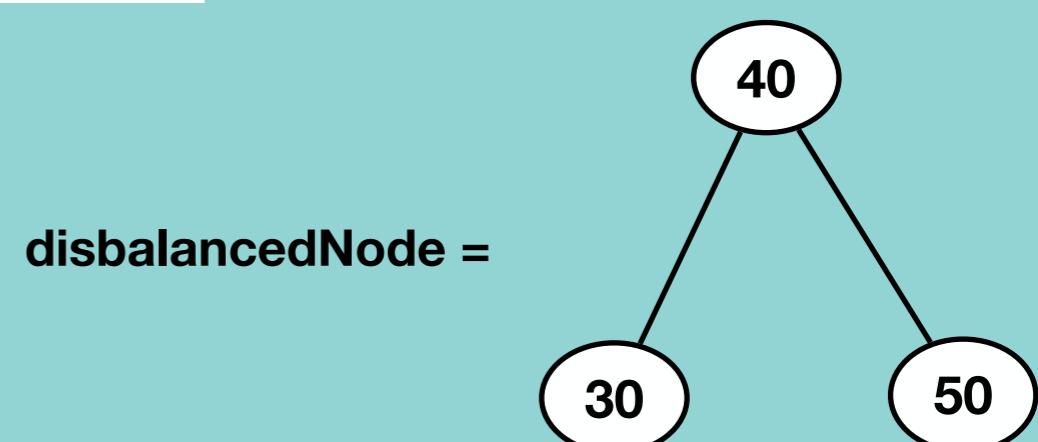
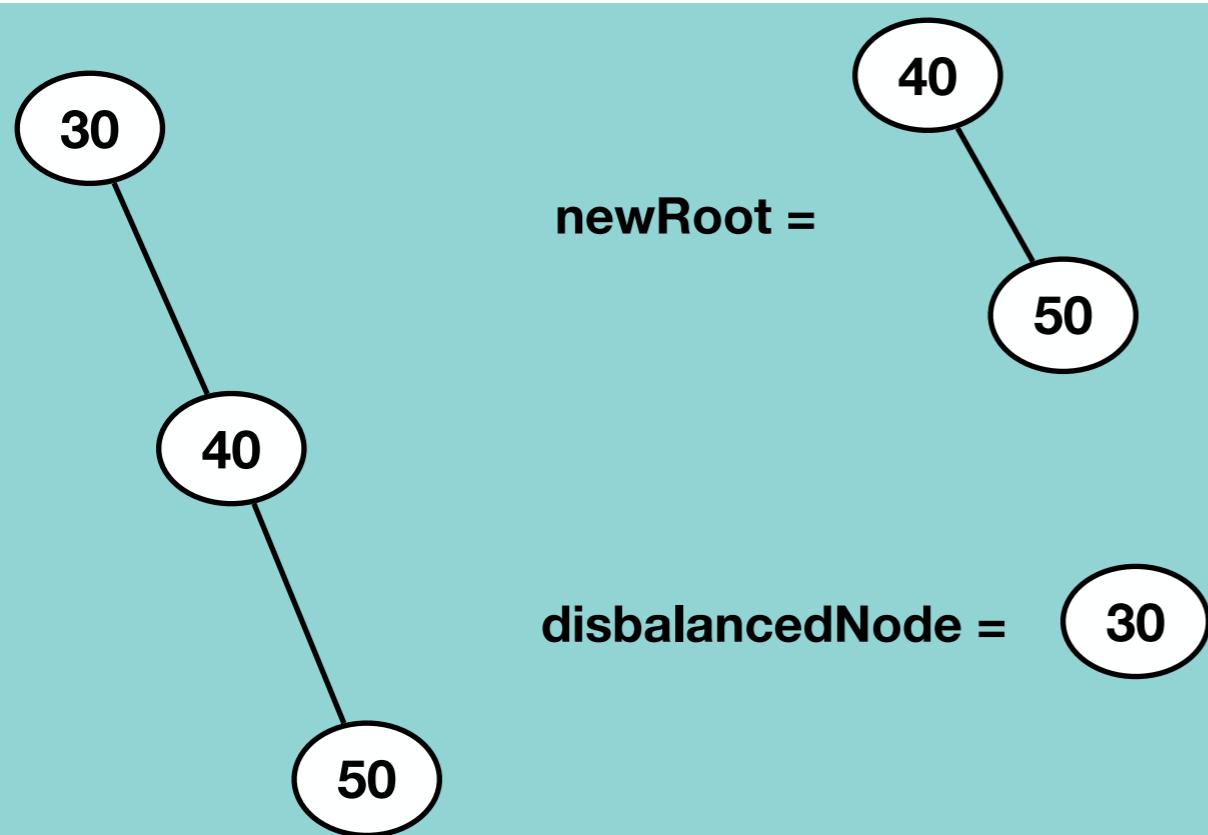


# Insert a node in AVL Tree

## Algorithm of Right Right (RR) Condition

Step 1 : rotate Left disbalancedNode

```
rotateLeft(disbalancedNode):  
    newRoot = disbalancedNode.rightChild  
    disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild  
    newRoot.leftChild = disbalancedNode  
    update height of disbalancedNode and newRoot  
    return newRoot
```



# Insert a node in AVL Tree

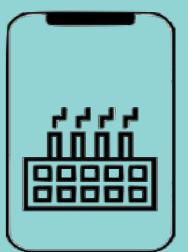
## Algorithm of Right Right (RR) Condition

Step 1 : rotate Left disbalancedNode

```
rotateLeft(disbalancedNode):
    newRoot = disbalancedNode.rightChild
    disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
    newRoot.leftChild = disbalancedNode
    update height of disbalancedNode and newRoot
    return newRoot
```

Time complexity : O(1)

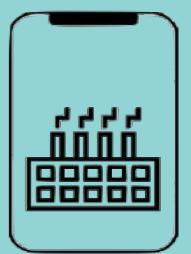
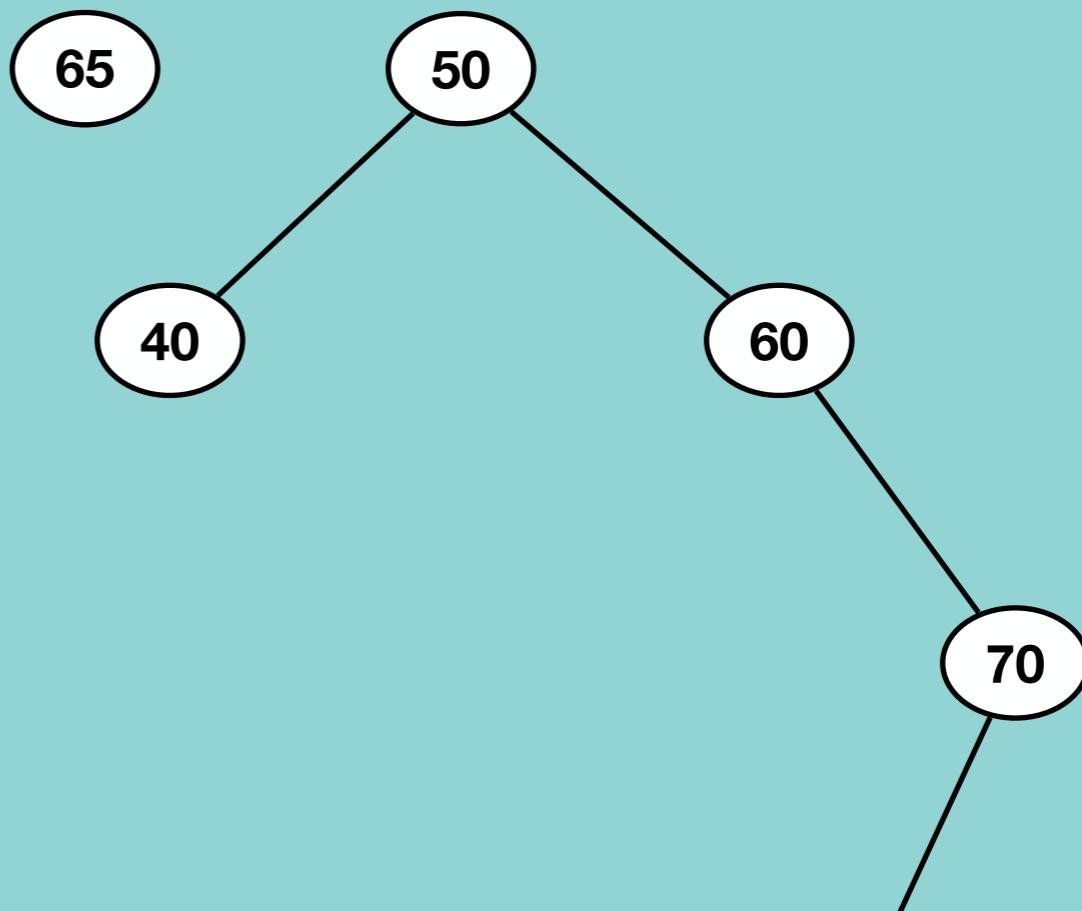
Space complexity : O(1)



# Insert a node in AVL Tree

Case 2 : Rotation is required

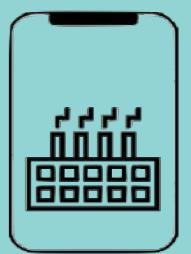
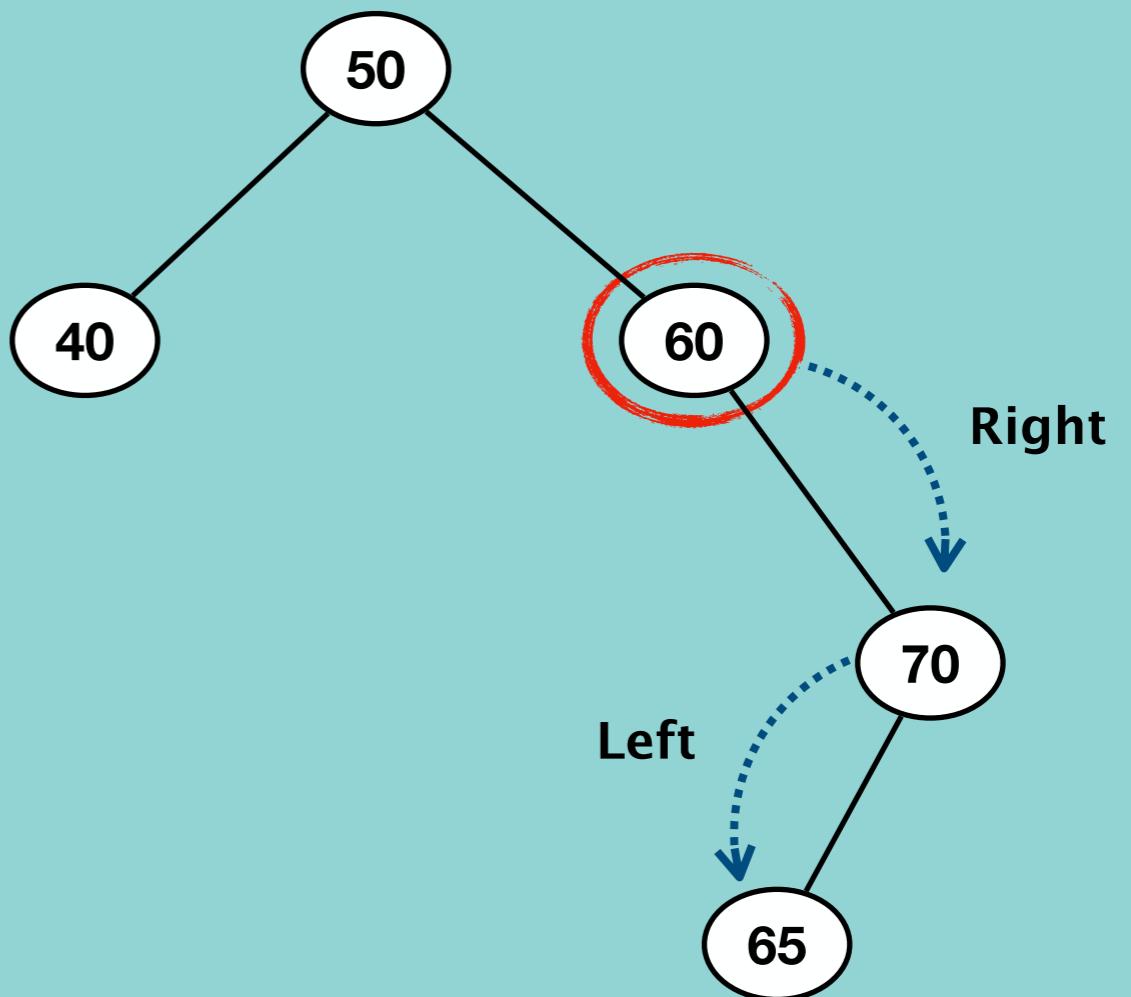
RL - right left condition



# Insert a node in AVL Tree

Case 2 : Rotation is required

RL – right left condition

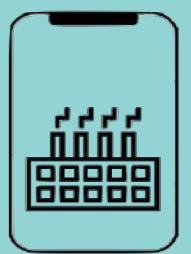
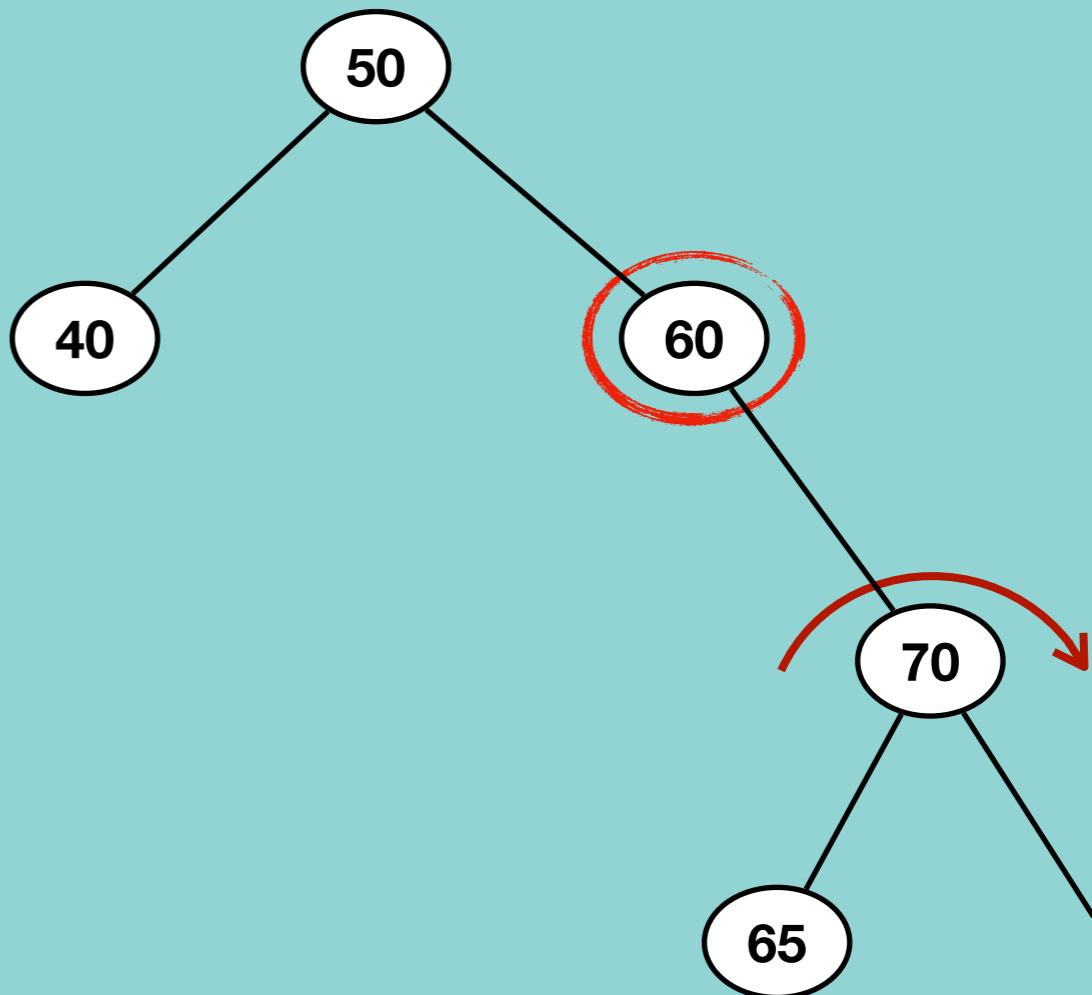


# Insert a node in AVL Tree

Case 2 : Rotation is required

RL – right left condition

1. Right rotation
2. Left rotation

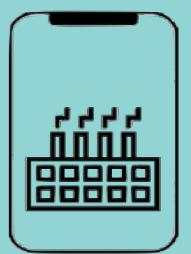
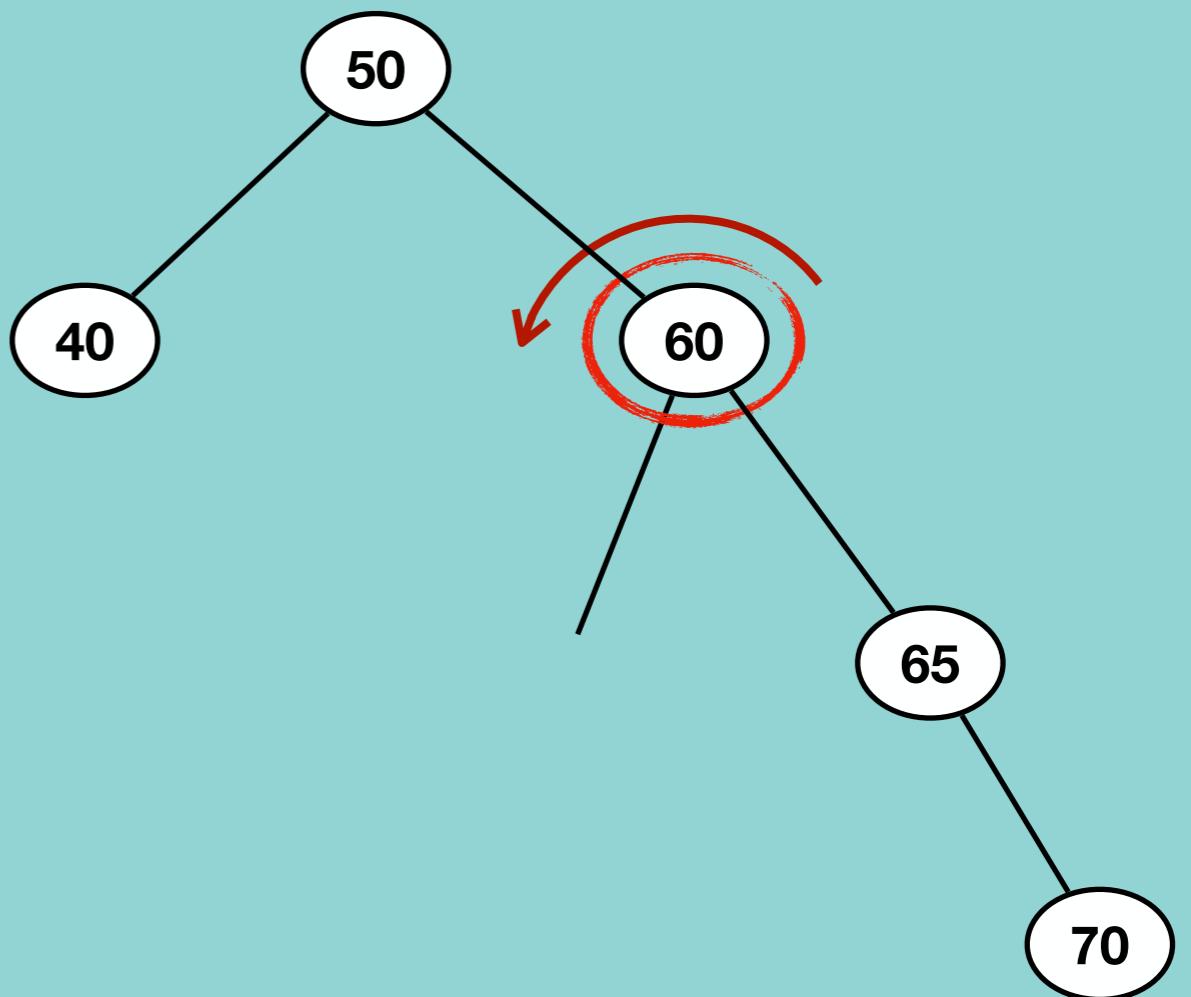


# Insert a node in AVL Tree

Case 2 : Rotation is required

RL – right left condition

1. Right rotation
2. Left rotation



# Insert a node in AVL Tree

## Algorithm of Right Left (RL) Condition

Step 1 : rotate Right `disbalancedNode.rightChild`

Step 2 : rotate Left `disbalancedNode`

```
rotateRight(disbalancedNode):
```

```
    newRoot = disbalancedNode.leftChild
```

```
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
```

```
    newRoot.rightChild = disbalancedNode
```

```
    update height of disbalancedNode and newRoot
```

```
    return newRoot
```

```
rotateLeft(disbalancedNode):
```

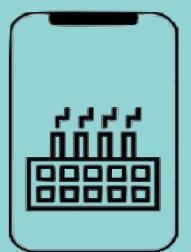
```
    newRoot = disbalancedNode.rightChild
```

```
    disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
```

```
    newRoot.leftChild = disbalancedNode
```

```
    update height of disbalancedNode and newRoot
```

```
    return newRoot
```



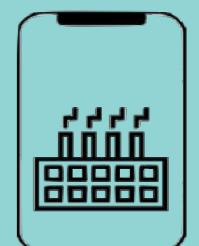
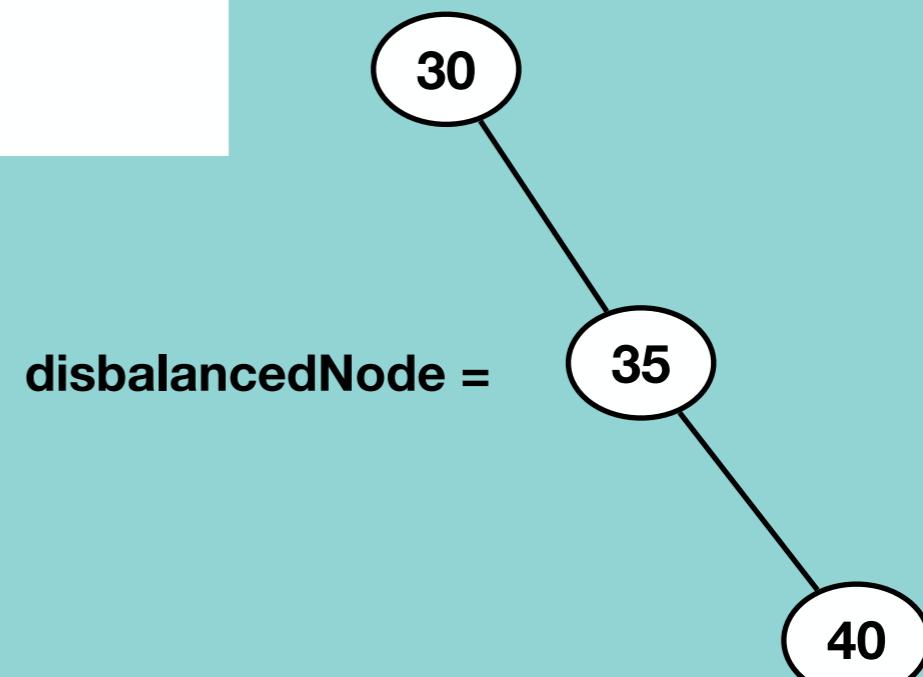
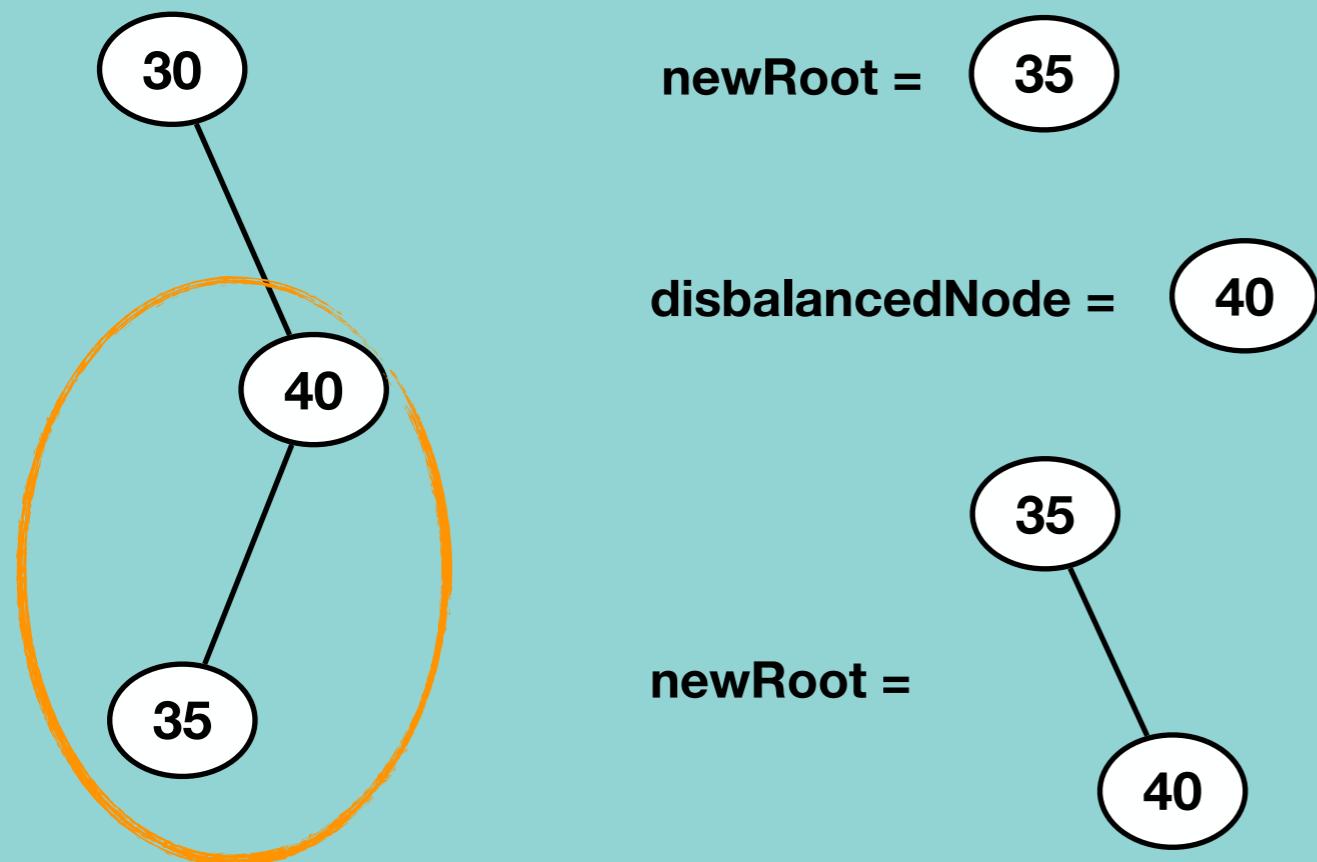
# Insert a node in AVL Tree

## Algorithm of Right Left (RL) Condition

Step 1 : rotate Right disbalancedNode.rightChild

Step 2 : rotate Left disbalancedNode

```
rotateRight(disbalancedNode):  
    newRoot = disbalancedNode.leftChild  
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild  
    newRoot.rightChild = disbalancedNode  
    update height of disbalancedNode and newRoot  
    return newRoot
```



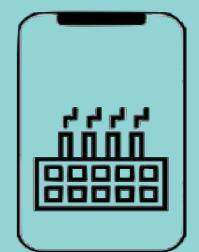
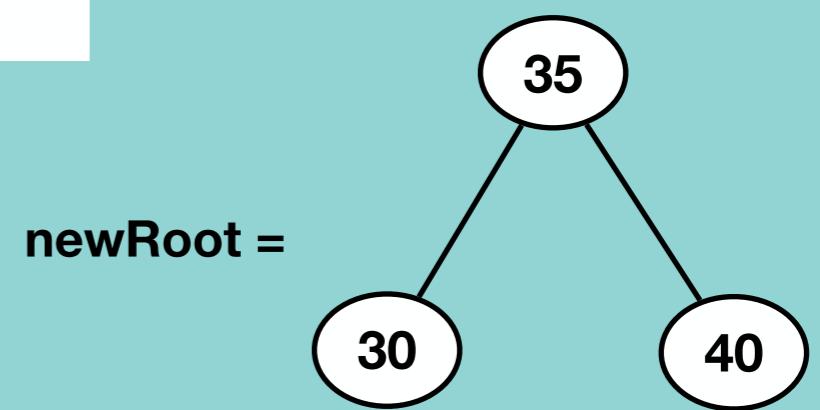
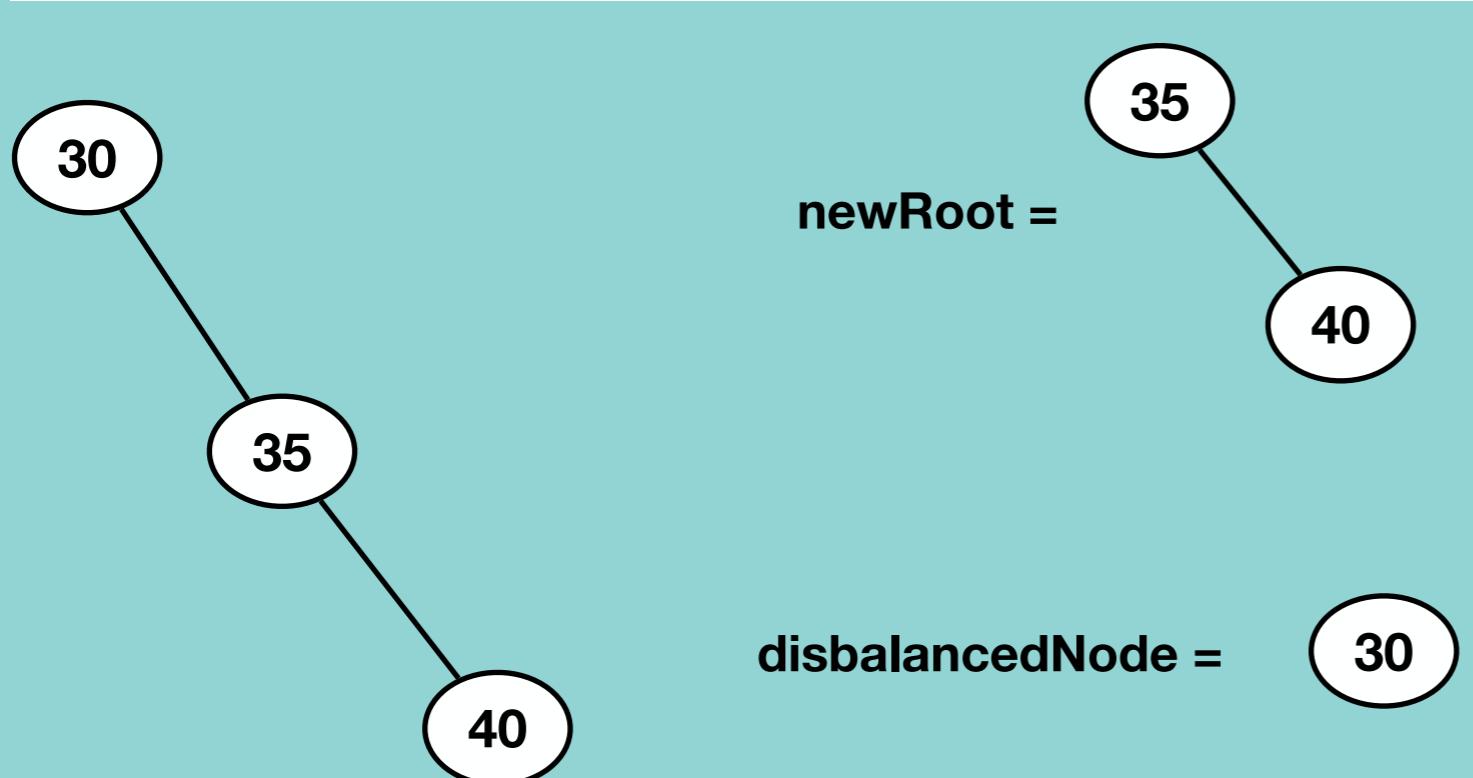
# Insert a node in AVL Tree

## Algorithm of Right Left (RL) Condition

Step 1 : rotate Right disbalancedNode.rightChild

Step 2 : rotate Left disbalancedNode

```
rotateLeft(disbalancedNode):  
    newRoot = disbalancedNode.rightChild  
    disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild  
    newRoot.leftChild = disbalancedNode  
    update height of disbalancedNode and newRoot  
    return newRoot
```



# Insert a node in AVL Tree

## Algorithm of Right Left (RL) Condition

Step 1 : rotate Right `disbalancedNode.rightChild`

Step 2 : rotate Left `disbalancedNode`

```
rotateRight(disbalancedNode):
```

```
    newRoot = disbalancedNode.leftChild
```

```
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
```

```
    newRoot.rightChild = disbalancedNode
```

```
    update height of disbalancedNode and newRoot
```

```
    return newRoot
```

```
rotateLeft(disbalancedNode):
```

```
    newRoot = disbalancedNode.rightChild
```

```
    disbalancedNode.rightChild = disbalancedNode.rightChild.leftChild
```

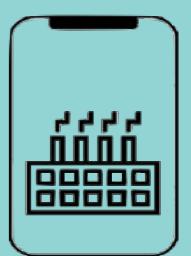
```
    newRoot.leftChild = disbalancedNode
```

```
    update height of disbalancedNode and newRoot
```

```
    return newRoot
```

Time complexity :  $O(1)$

Space complexity :  $O(1)$

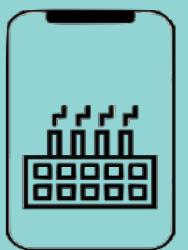


# Insert a node in AVL Tree (all together)

Case 1 : Rotation is not required

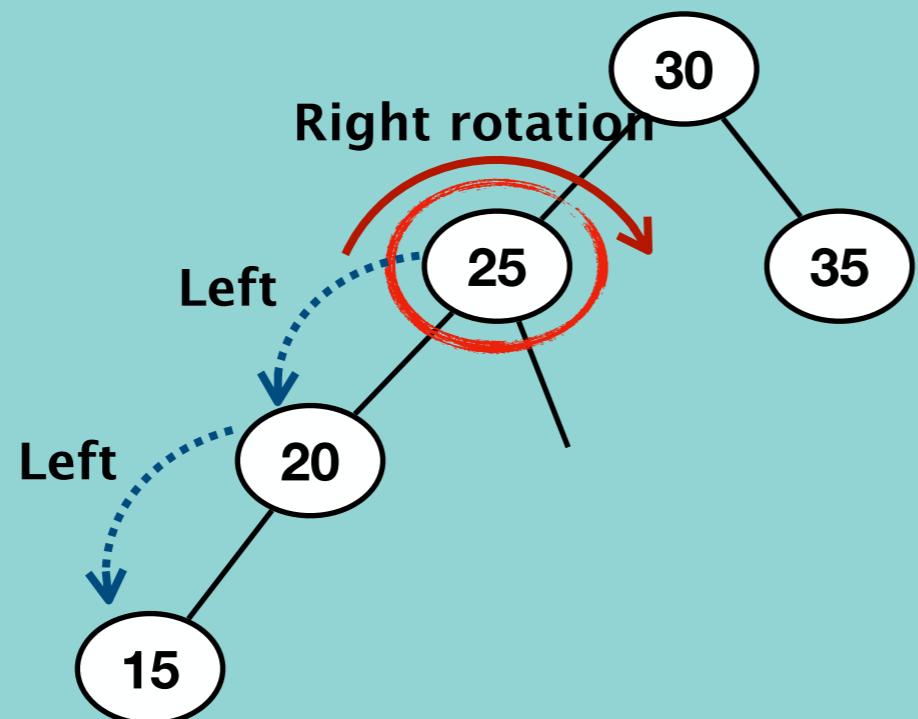
Case 2 : Rotation is required

- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)

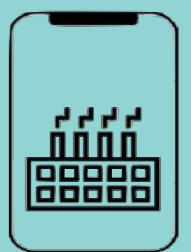


# Insert a node in AVL Tree (all together)

30,25,35,20,15,5,10,50,60,70,65

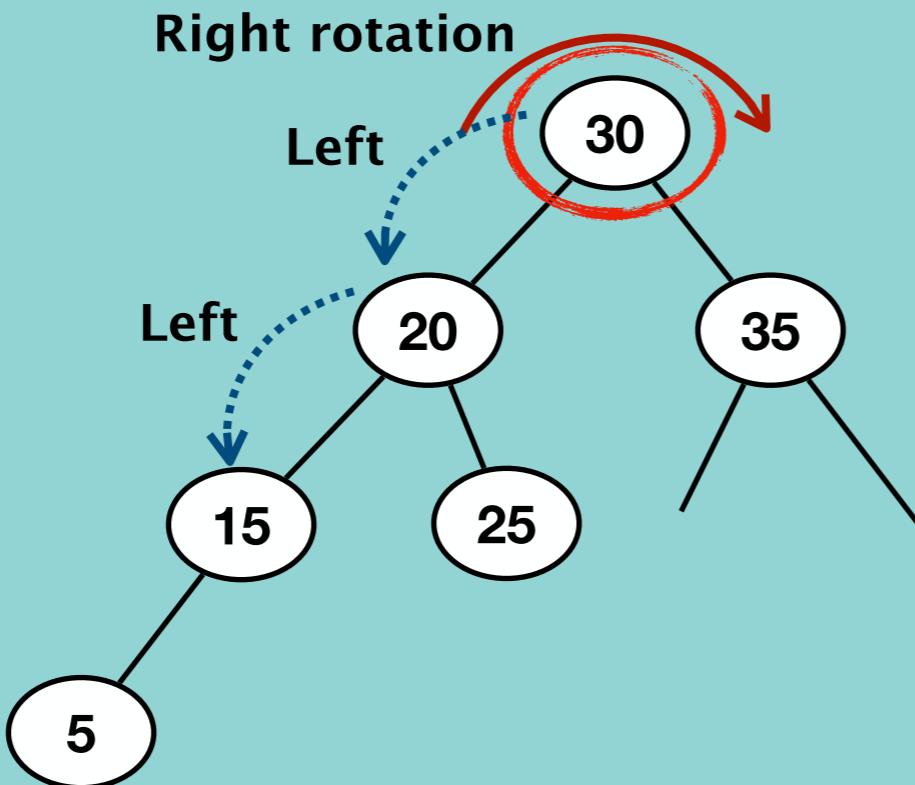


- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)

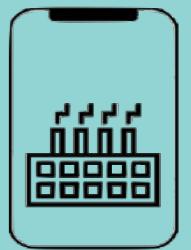


# Insert a node in AVL Tree (all together)

30,25,35,20,15,5,10,50,60,70,65



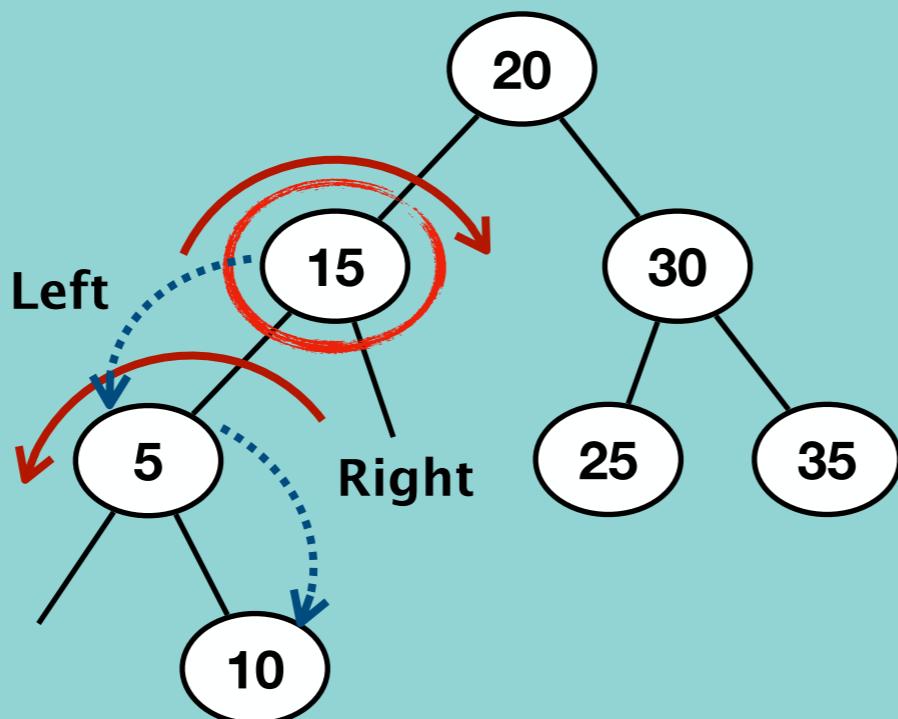
- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)



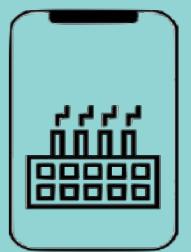
# Insert a node in AVL Tree (all together)

30,25,35,20,15,5,10,50,60,70,65

Left rotation  
Right rotation



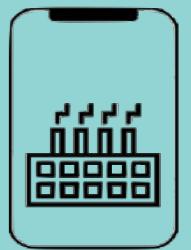
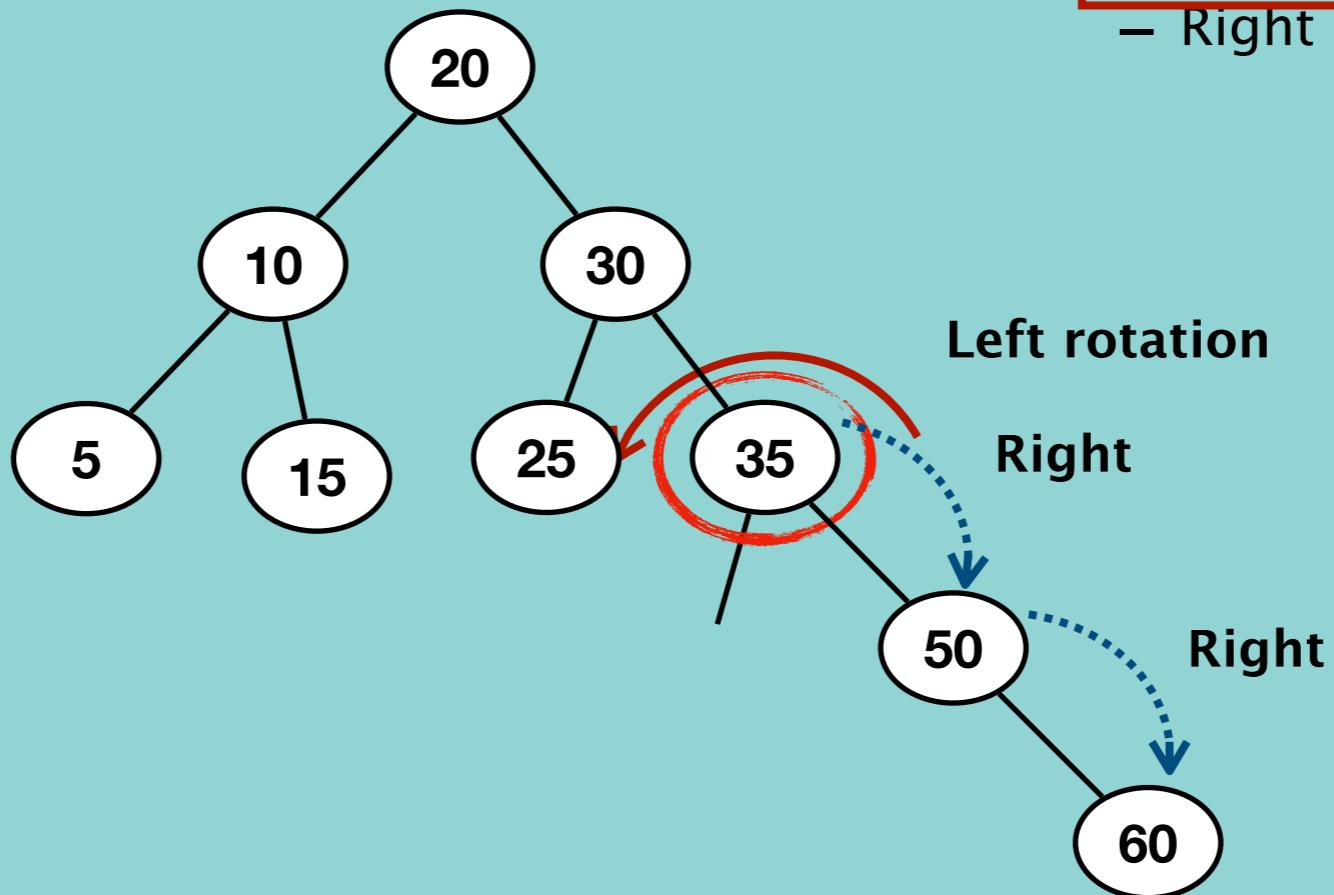
- Left Left condition (LL)
- Left Right condition (LR) **(highlighted with a red border)**
- Right Right condition (RR)
- Right Left condition (RL)



# Insert a node in AVL Tree (all together)

30,25,35,20,15,5,10,50,60,70,65

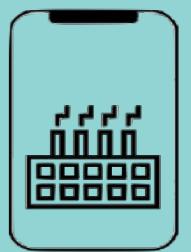
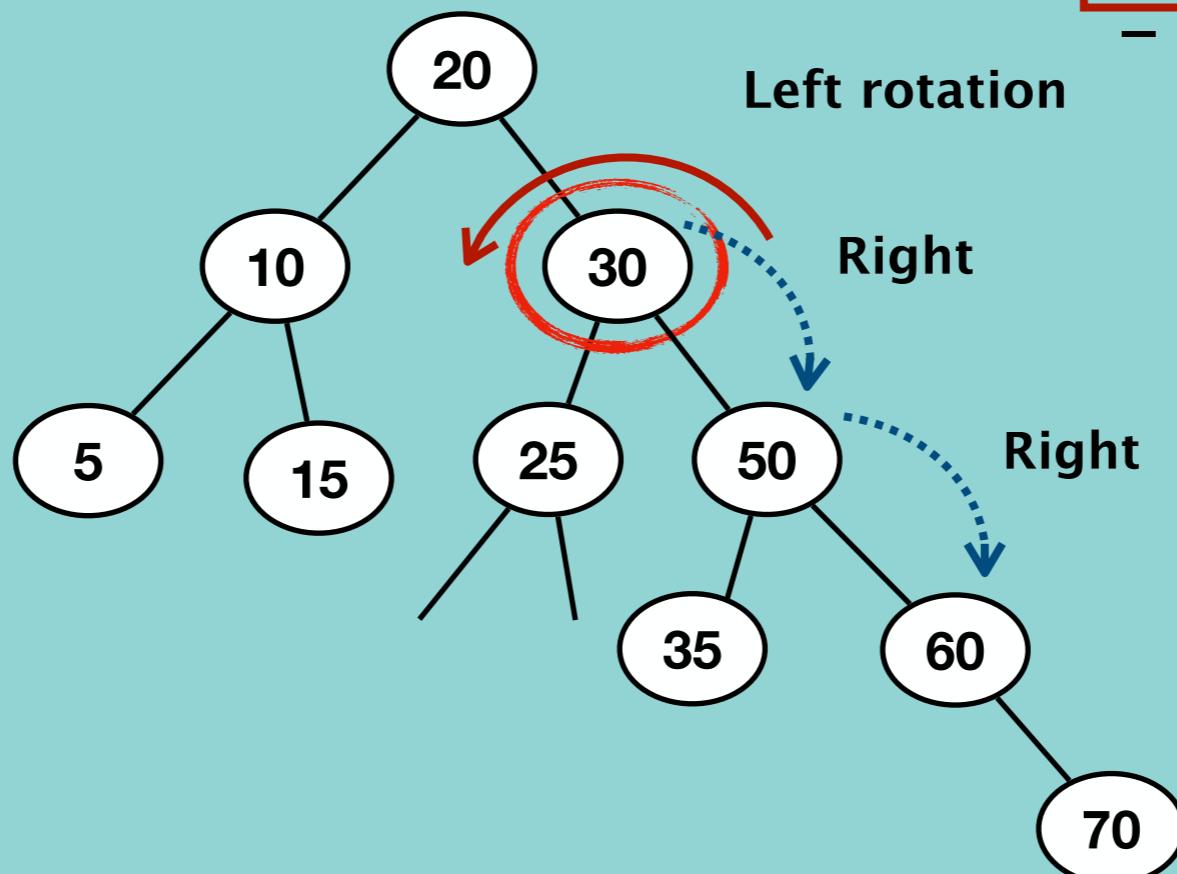
- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR) (RR)
- Right Left condition (RL)



# Insert a node in AVL Tree (all together)

30,25,35,20,15,5,10,50,60,70,65

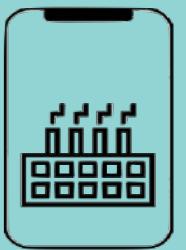
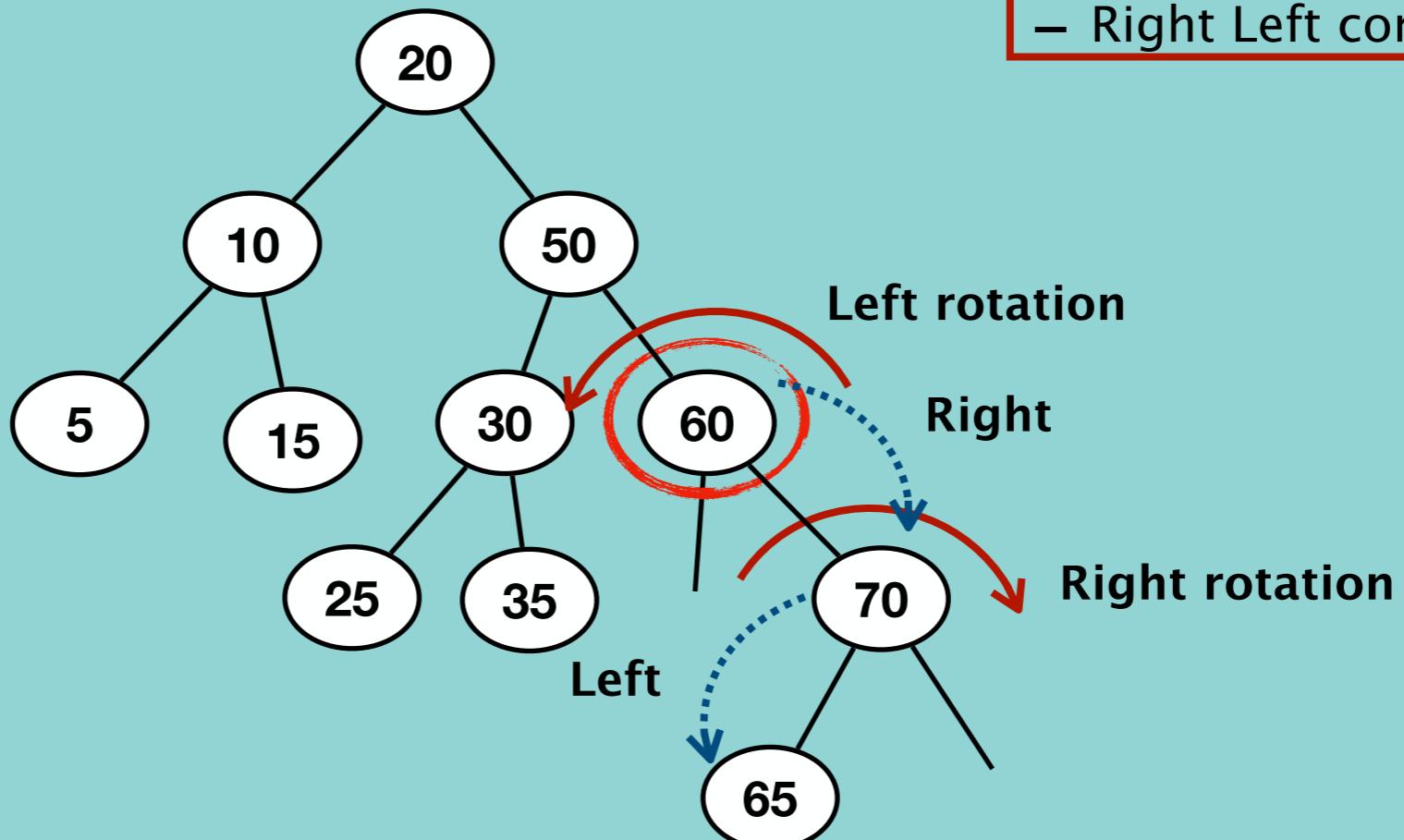
- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR) (RR)
- Right Left condition (RL)



# Insert a node in AVL Tree (all together)

30,25,35,20,15,5,10,50,60,70,65

- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)

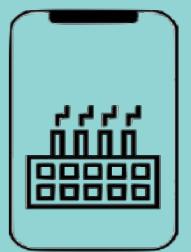
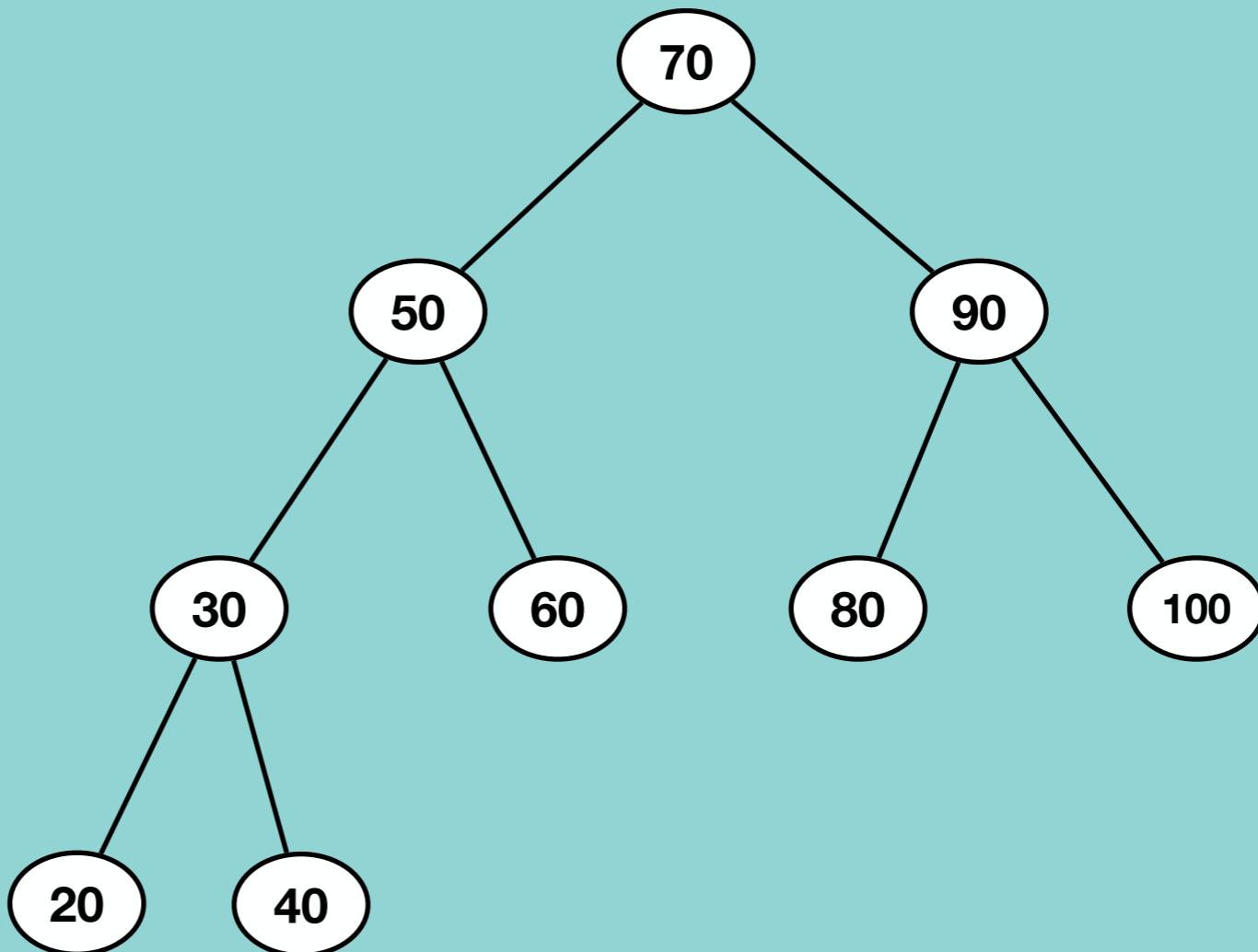


# Delete a node from AVL Tree

Case 1 – The tree does not exist

Case 2 – Rotation is not required

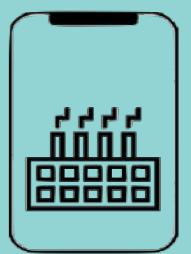
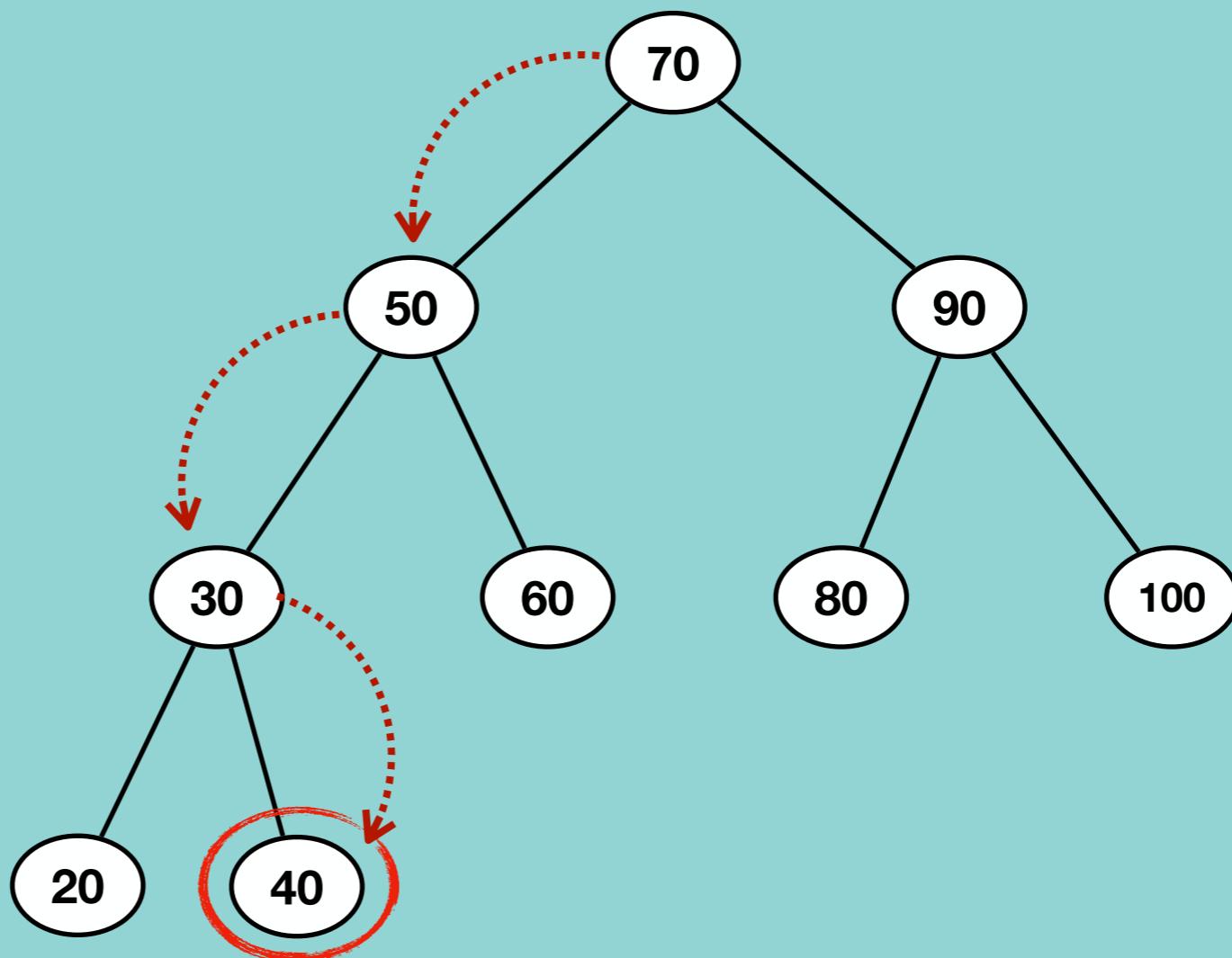
Case 3 – Rotation is required (LL, LR, RR, RL)



# Delete a node from AVL Tree

## Case 2 – Rotation is not required

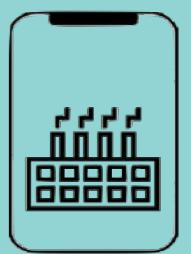
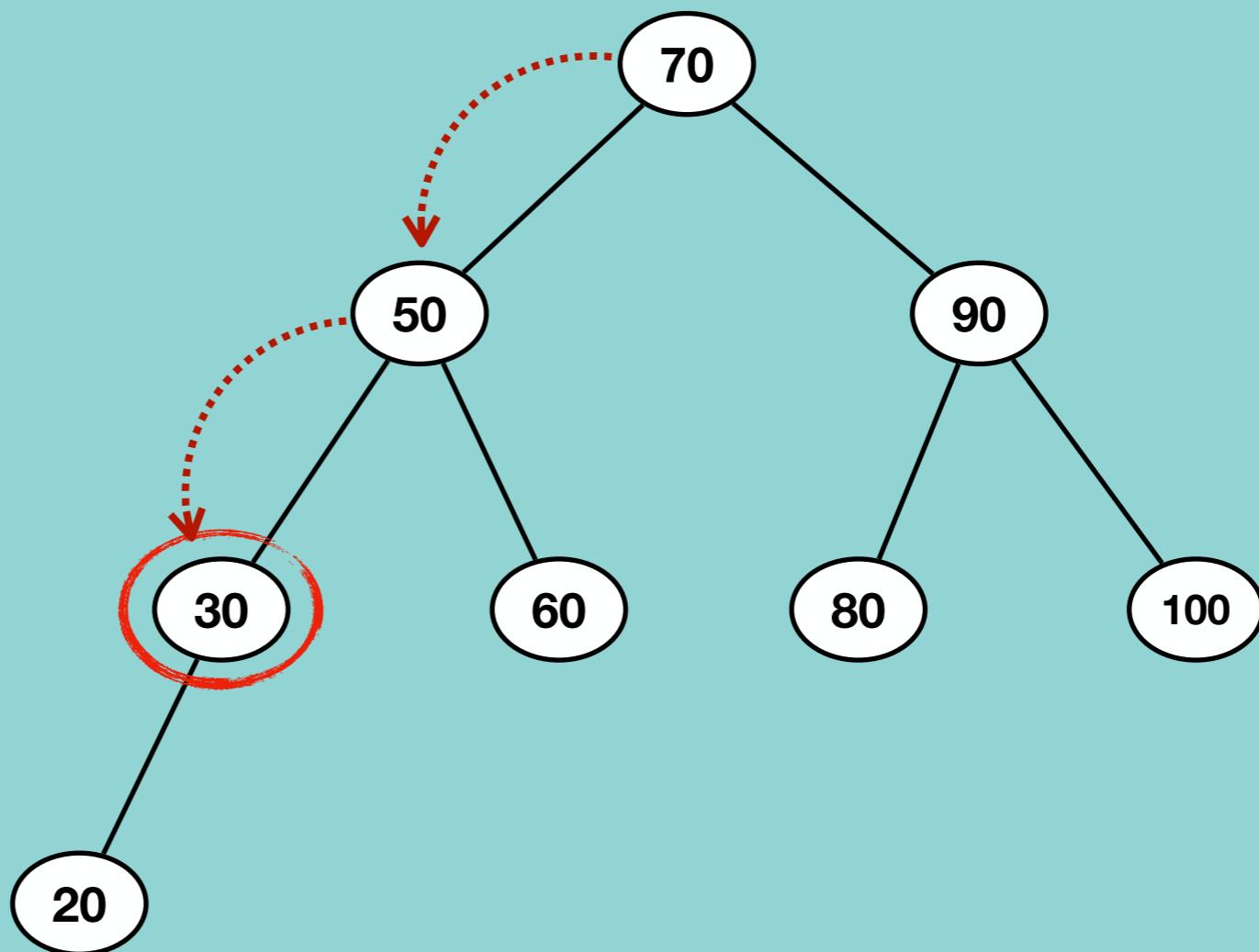
- The node to be deleted is a leaf node



# Delete a node from AVL Tree

## Case 2 – Rotation is not required

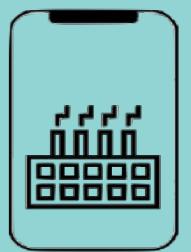
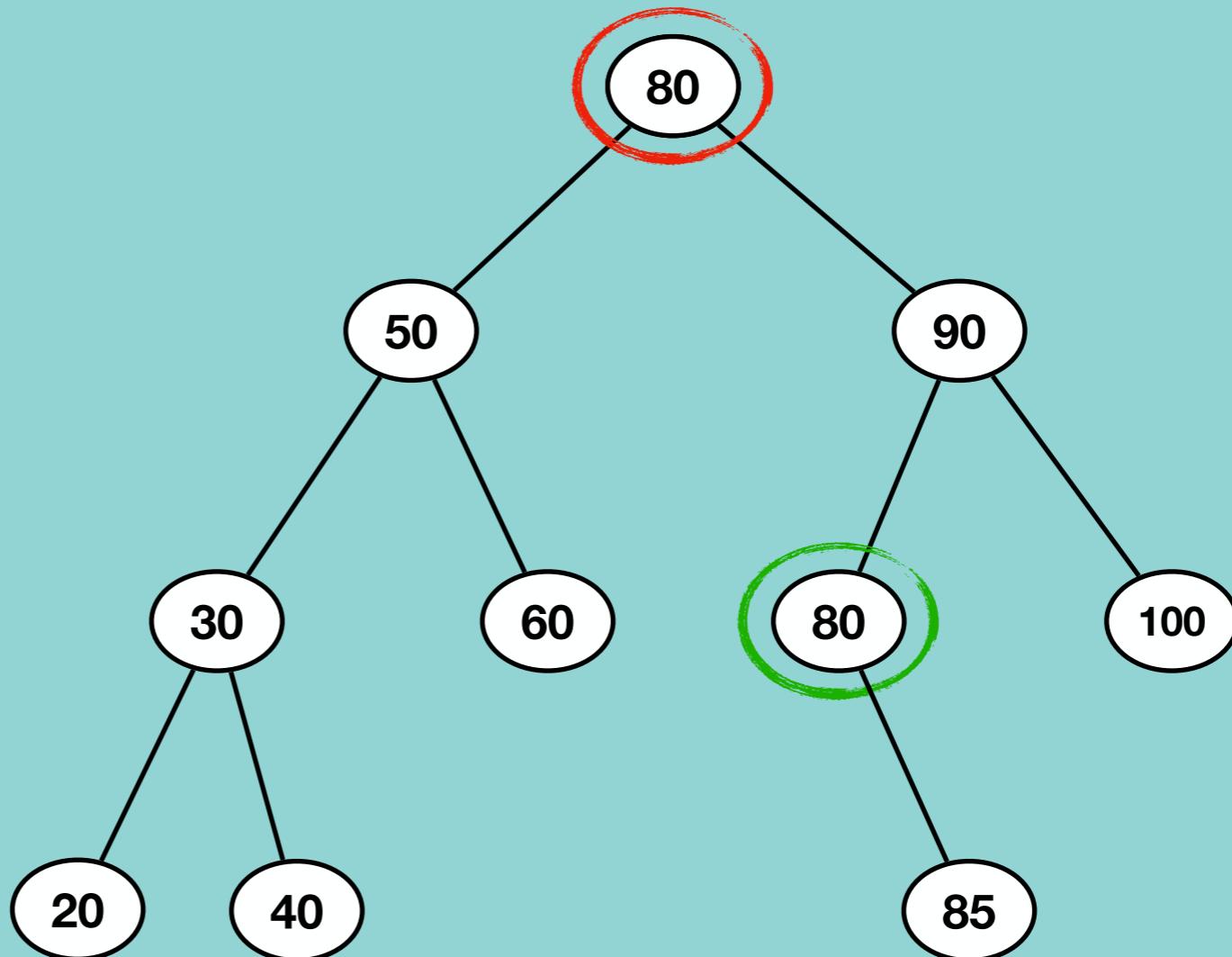
- The node to be deleted has a child node



# Delete a node from AVL Tree

## Case 2 – Rotation is not required

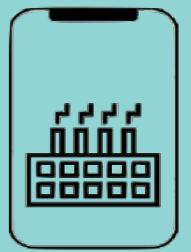
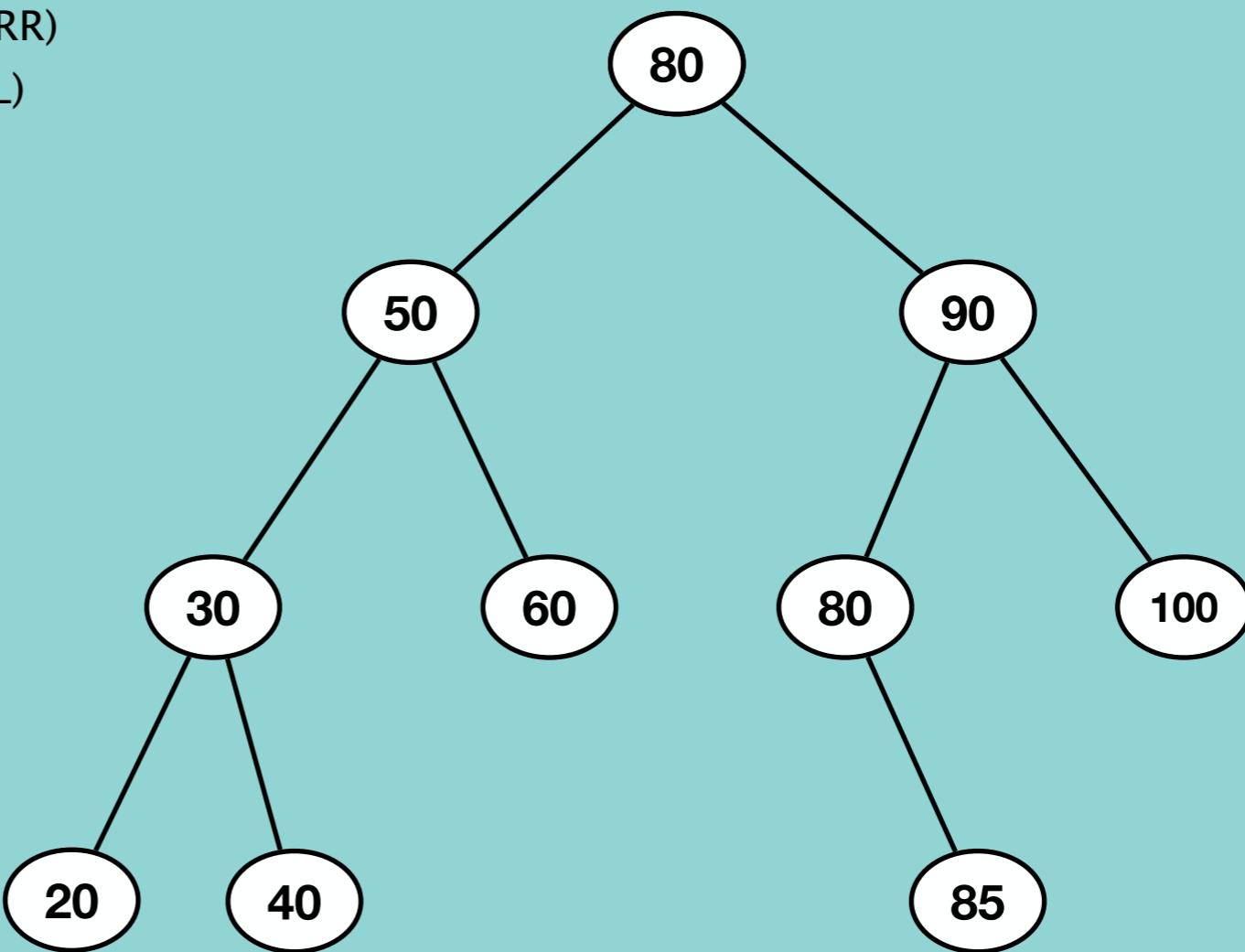
- The node to be deleted has two children



# Delete a node from AVL Tree

## Case 3 – Rotation is required

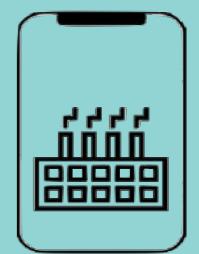
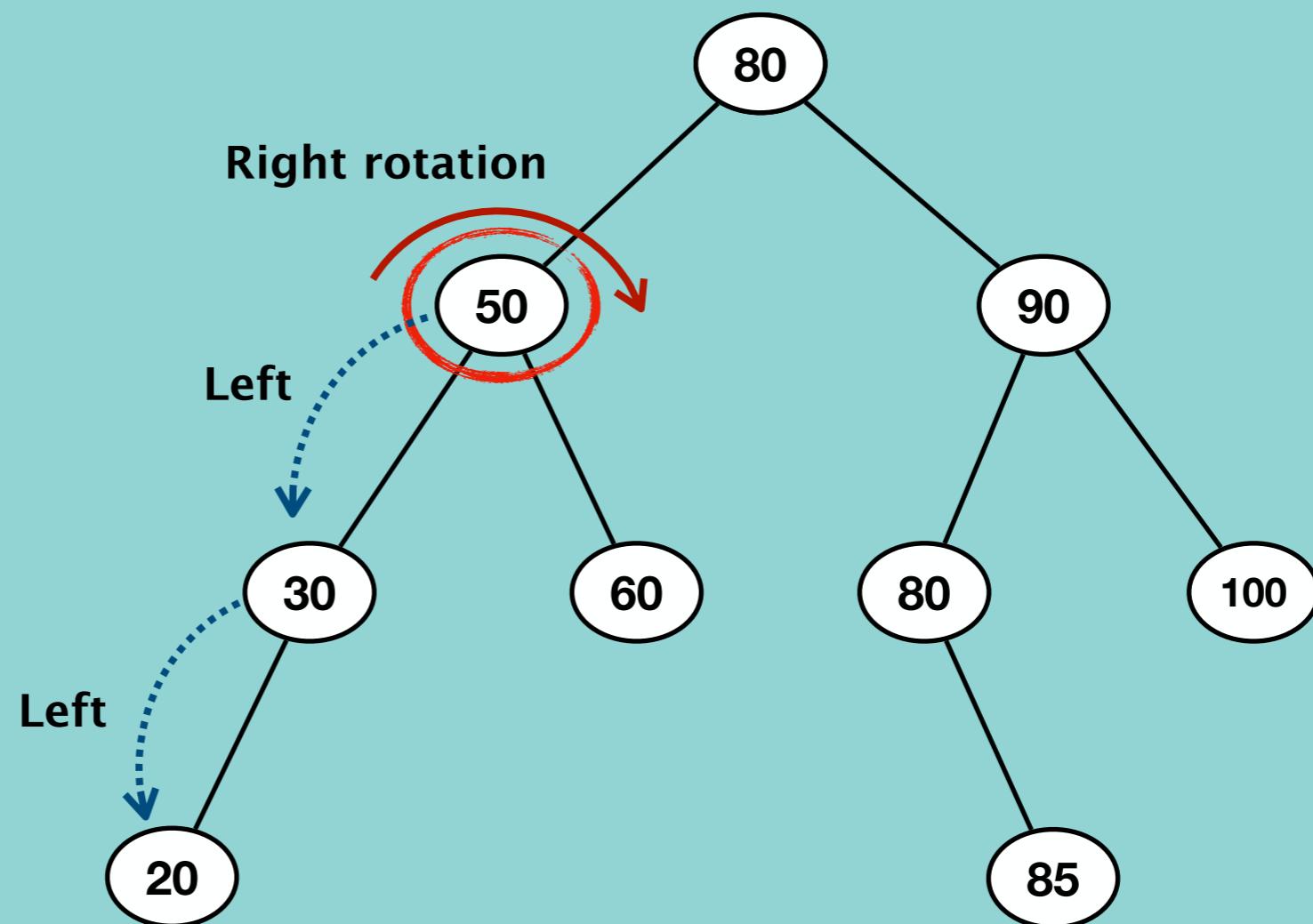
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



# Delete a node from AVL Tree

## Case 3 – Rotation is required

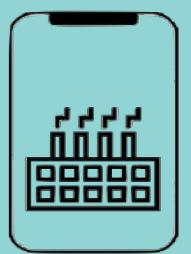
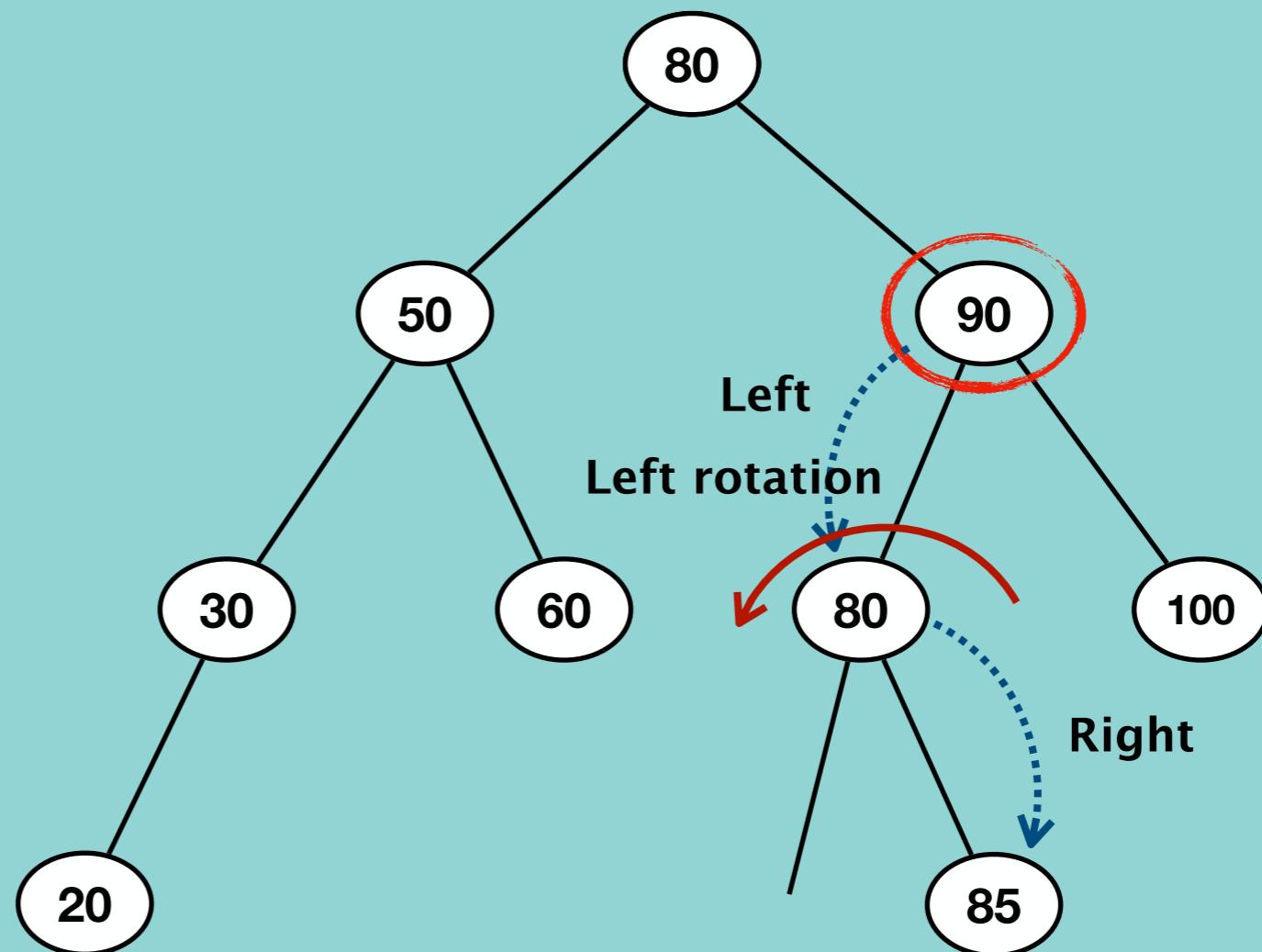
### Left Left Condition (LL)



# Delete a node from AVL Tree

## Case 3 – Rotation is required

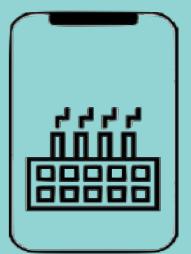
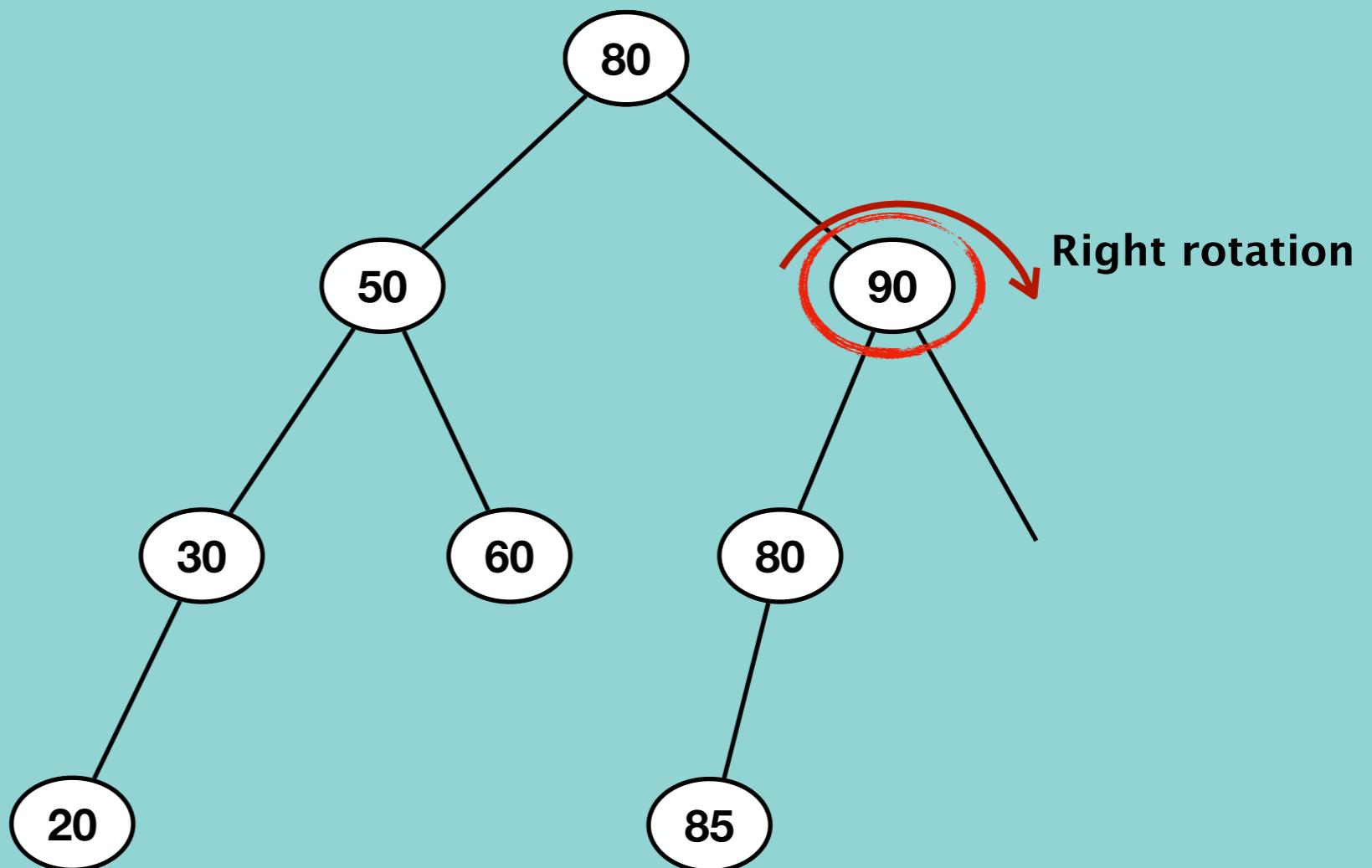
Left Right Condition (LR)



# Delete a node from AVL Tree

Case 3 – Rotation is required

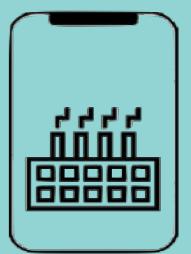
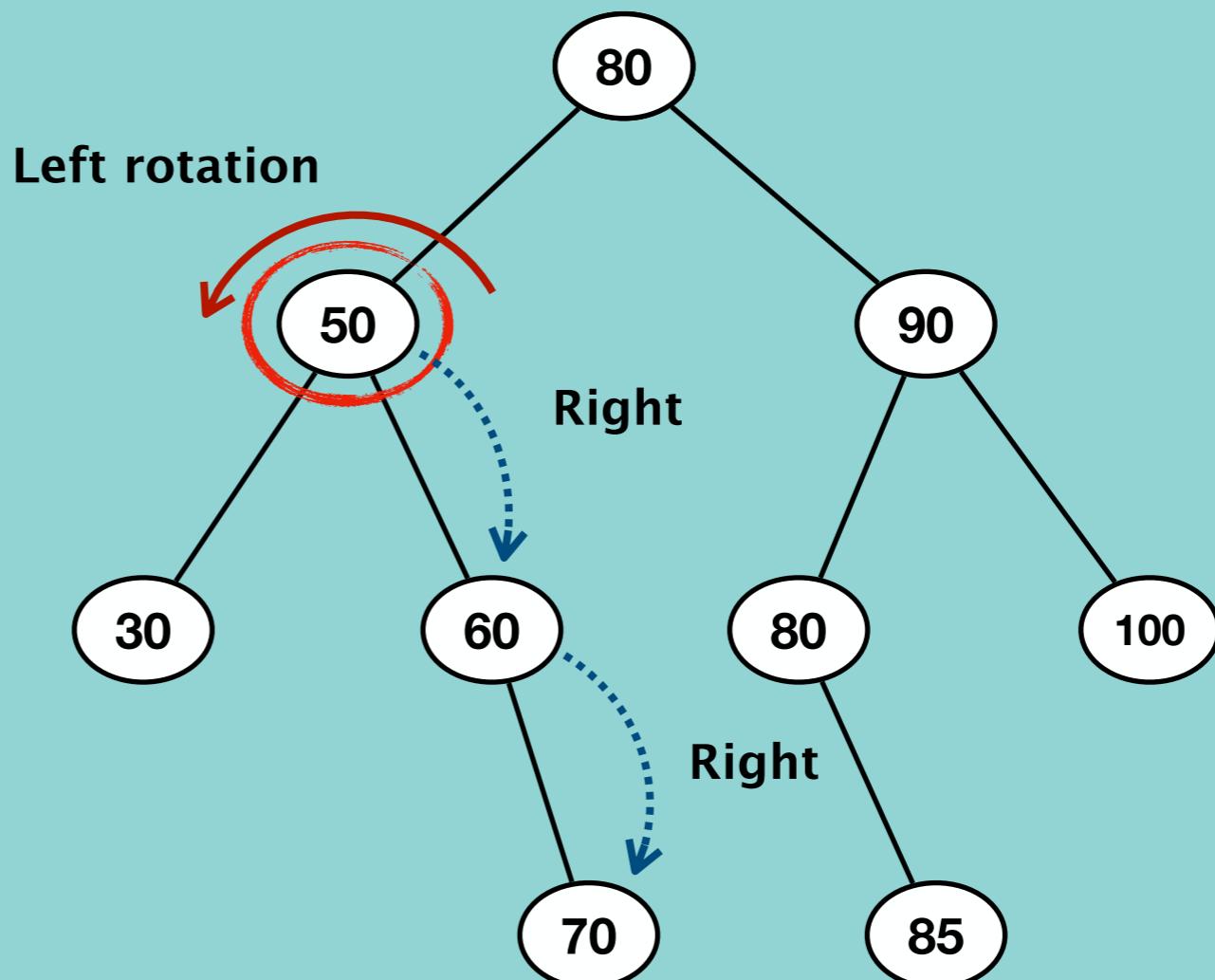
Left Right Condition (LR)



# Delete a node from AVL Tree

## Case 3 – Rotation is required

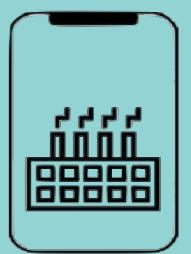
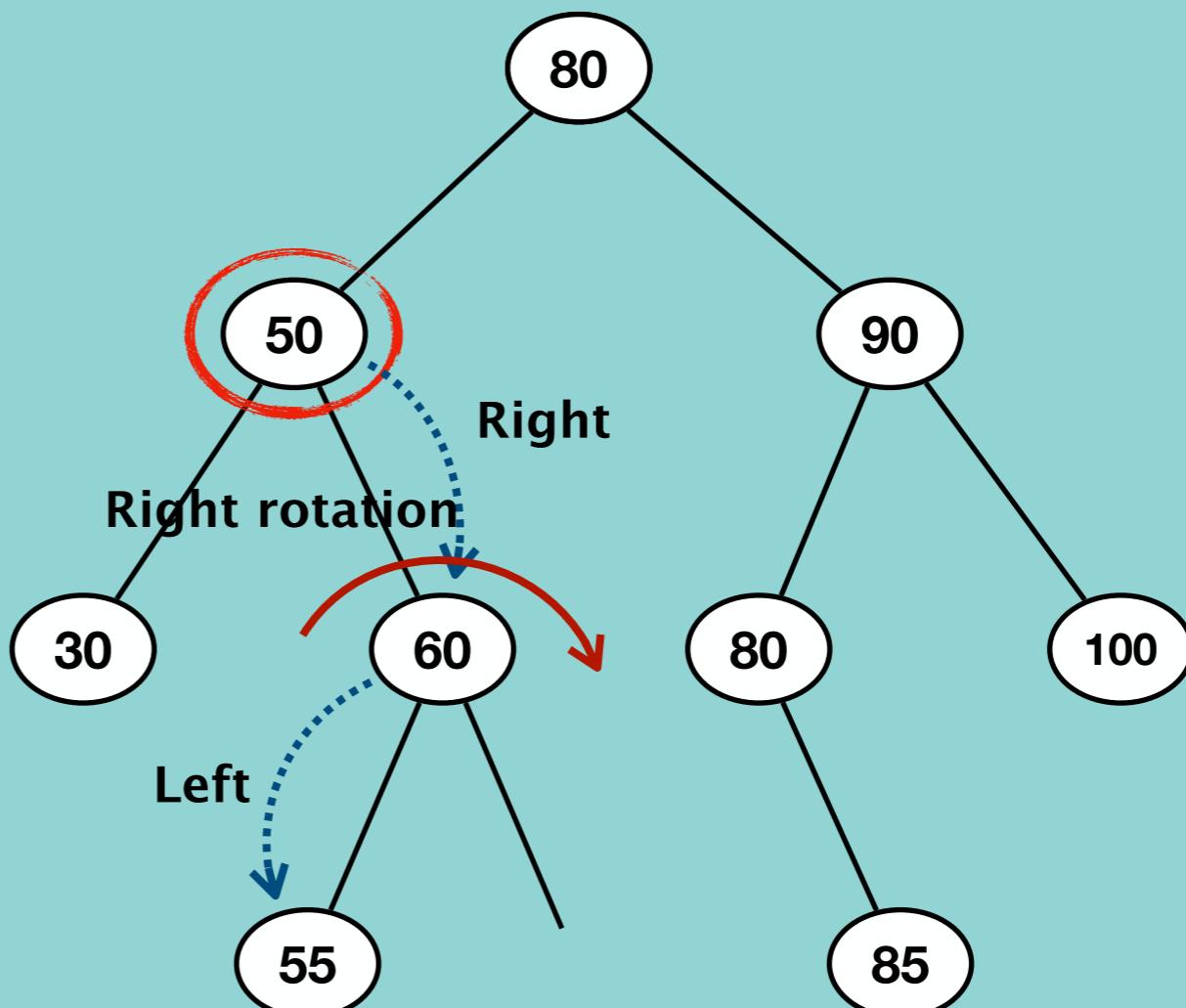
### Right Right Condition (RR)



# Delete a node from AVL Tree

## Case 3 – Rotation is required

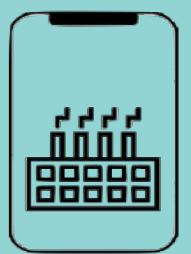
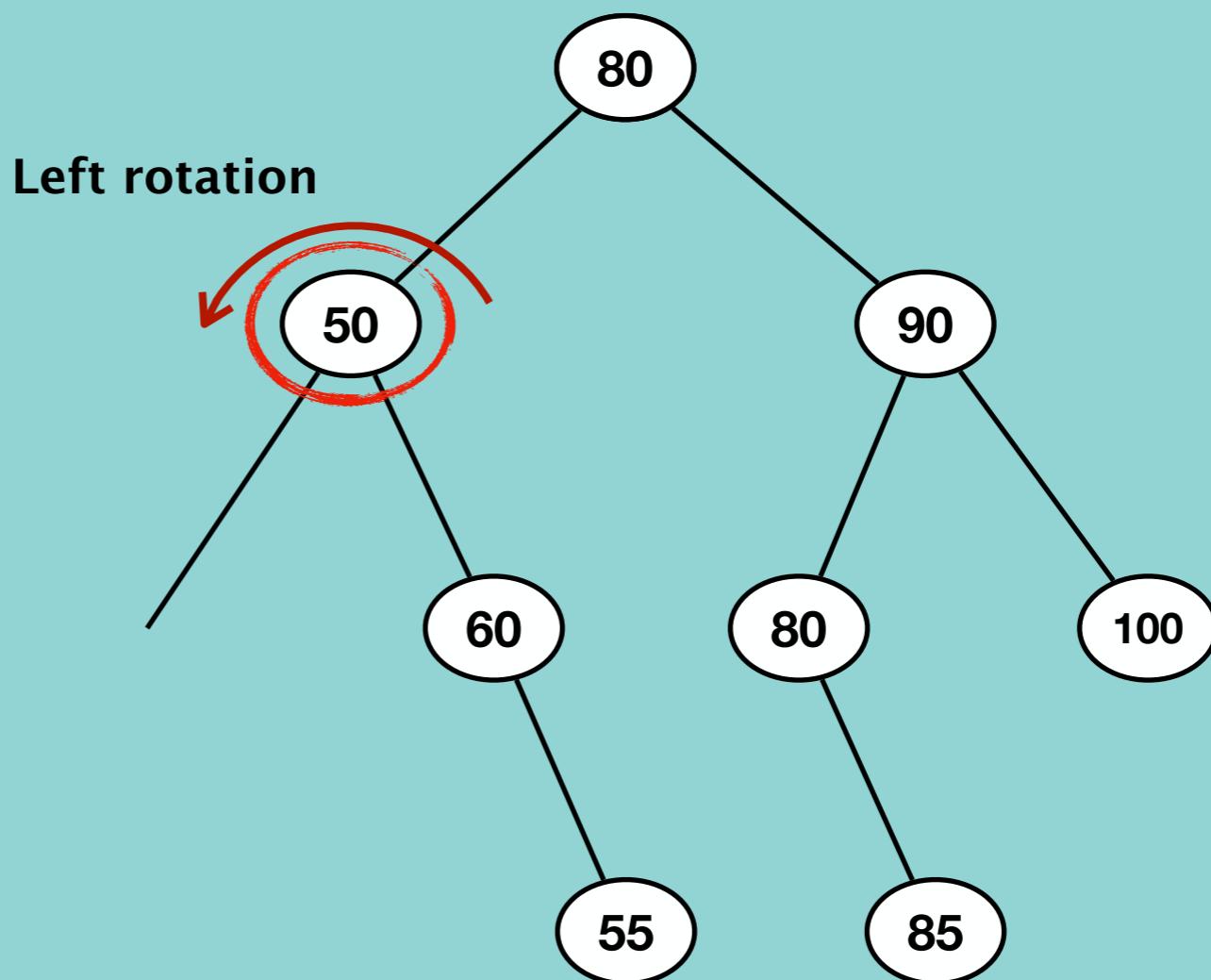
### Right Left Condition (RL)



# Delete a node from AVL Tree

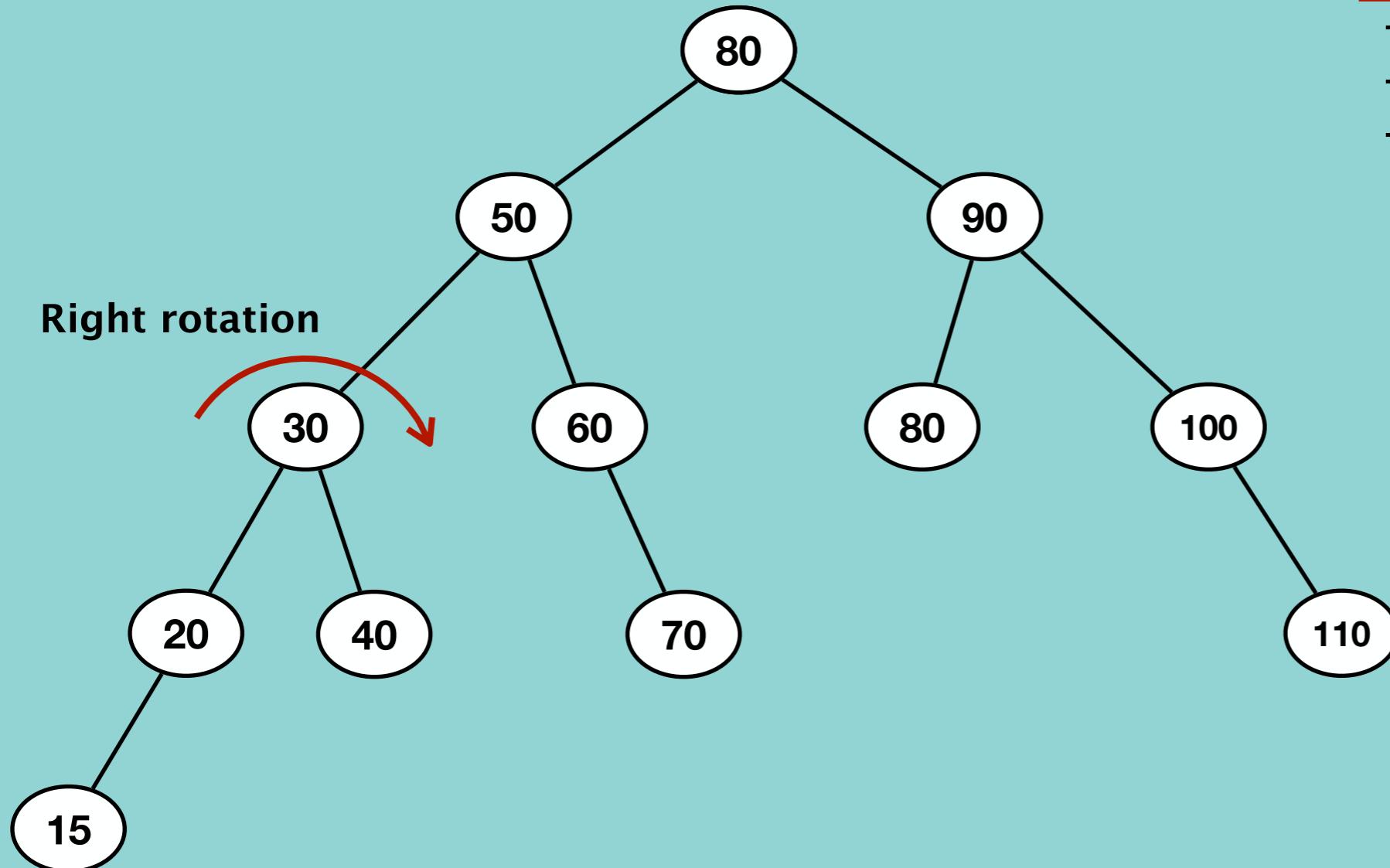
## Case 3 – Rotation is required

### Right Left Condition (RL)

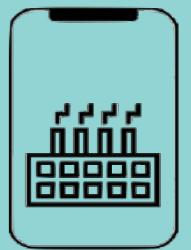


# Delete a node from AVL Tree (all together)

Delete 40



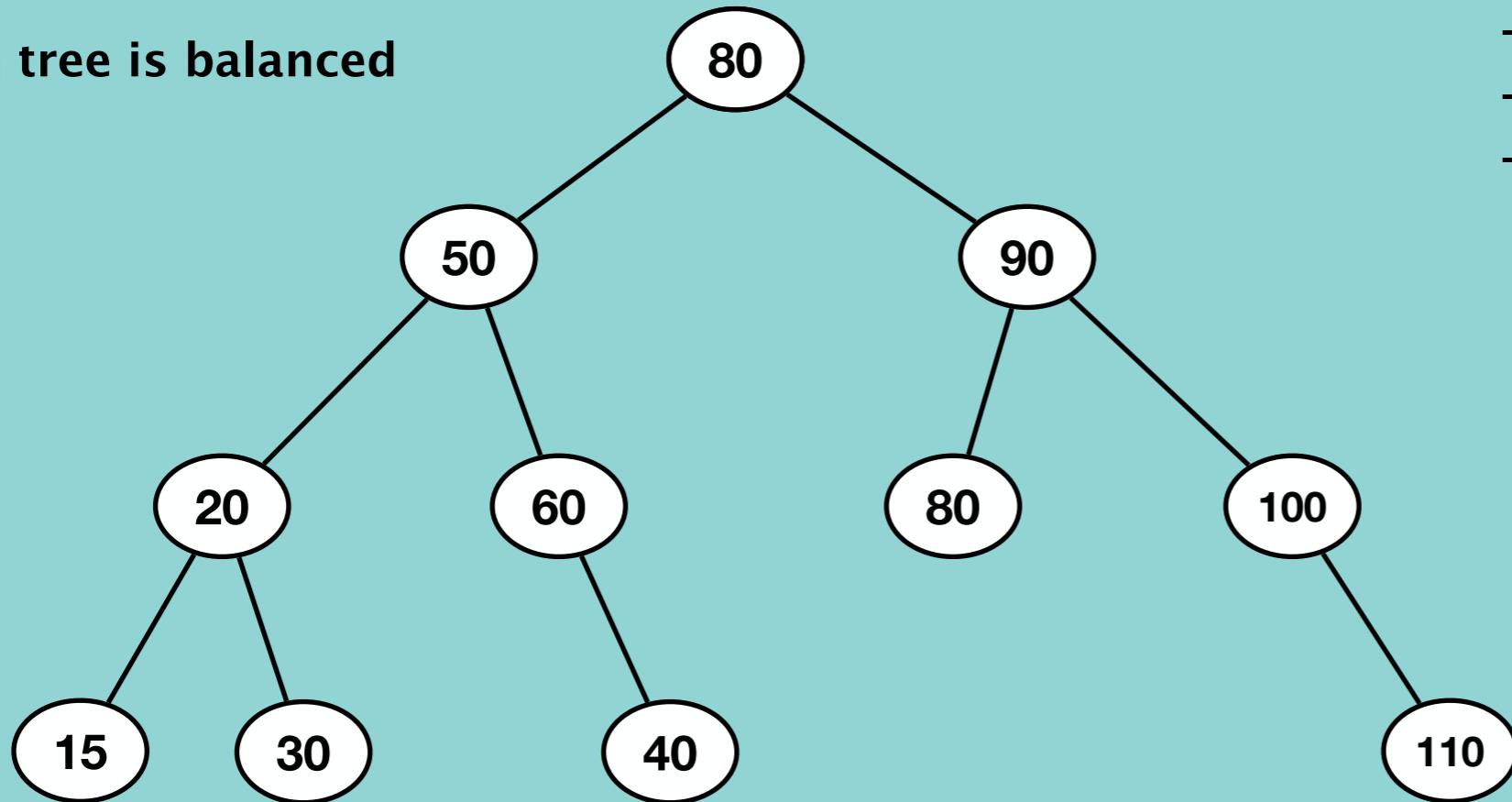
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



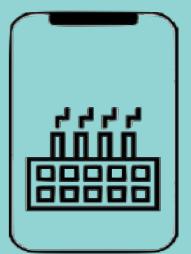
# Delete a node from AVL Tree (all together)

**Delete 15**

The tree is balanced



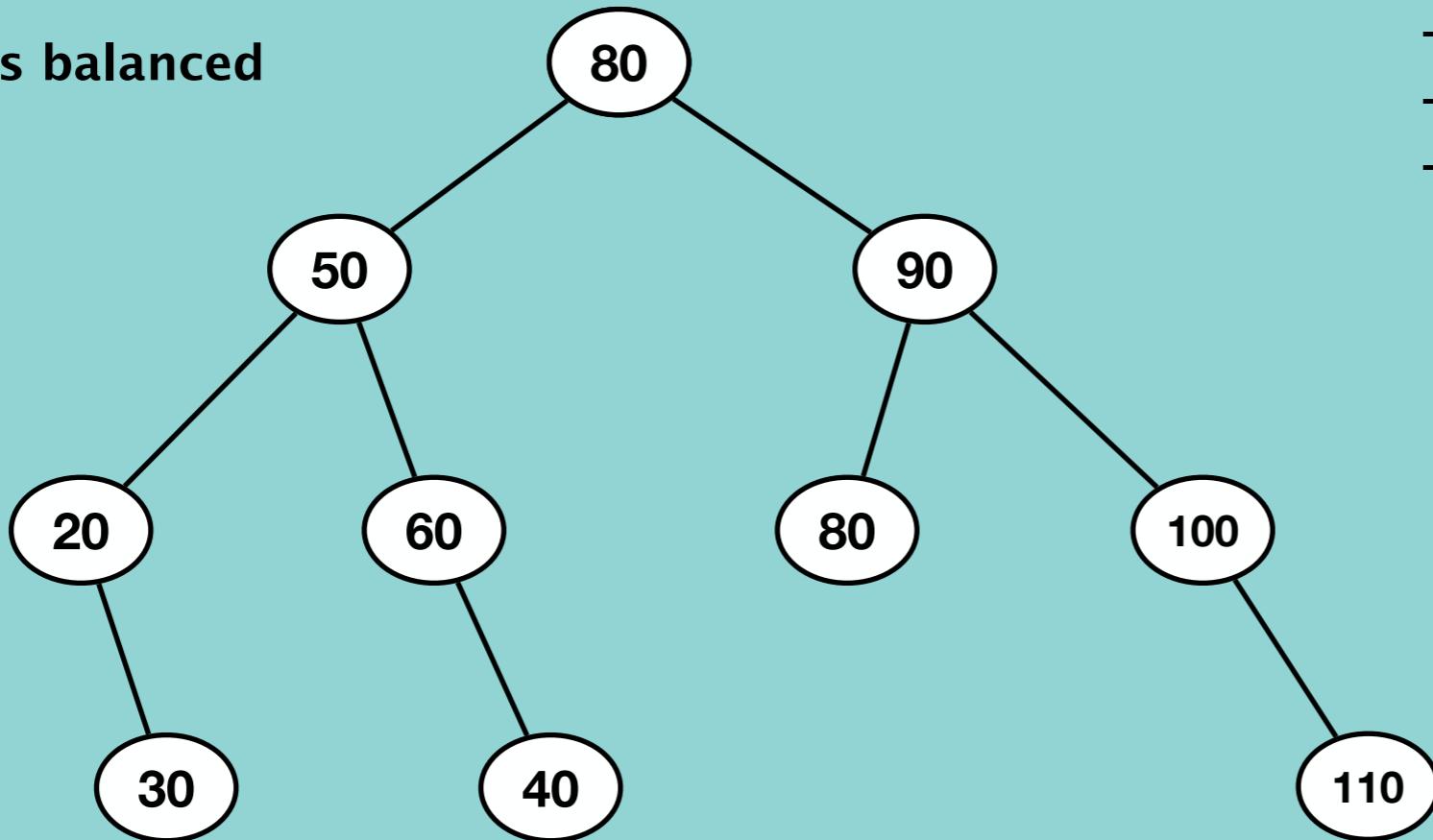
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



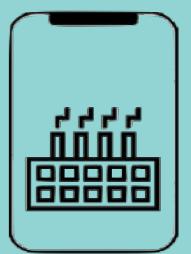
# Delete a node from AVL Tree (all together)

Delete 40

The tree is balanced



- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



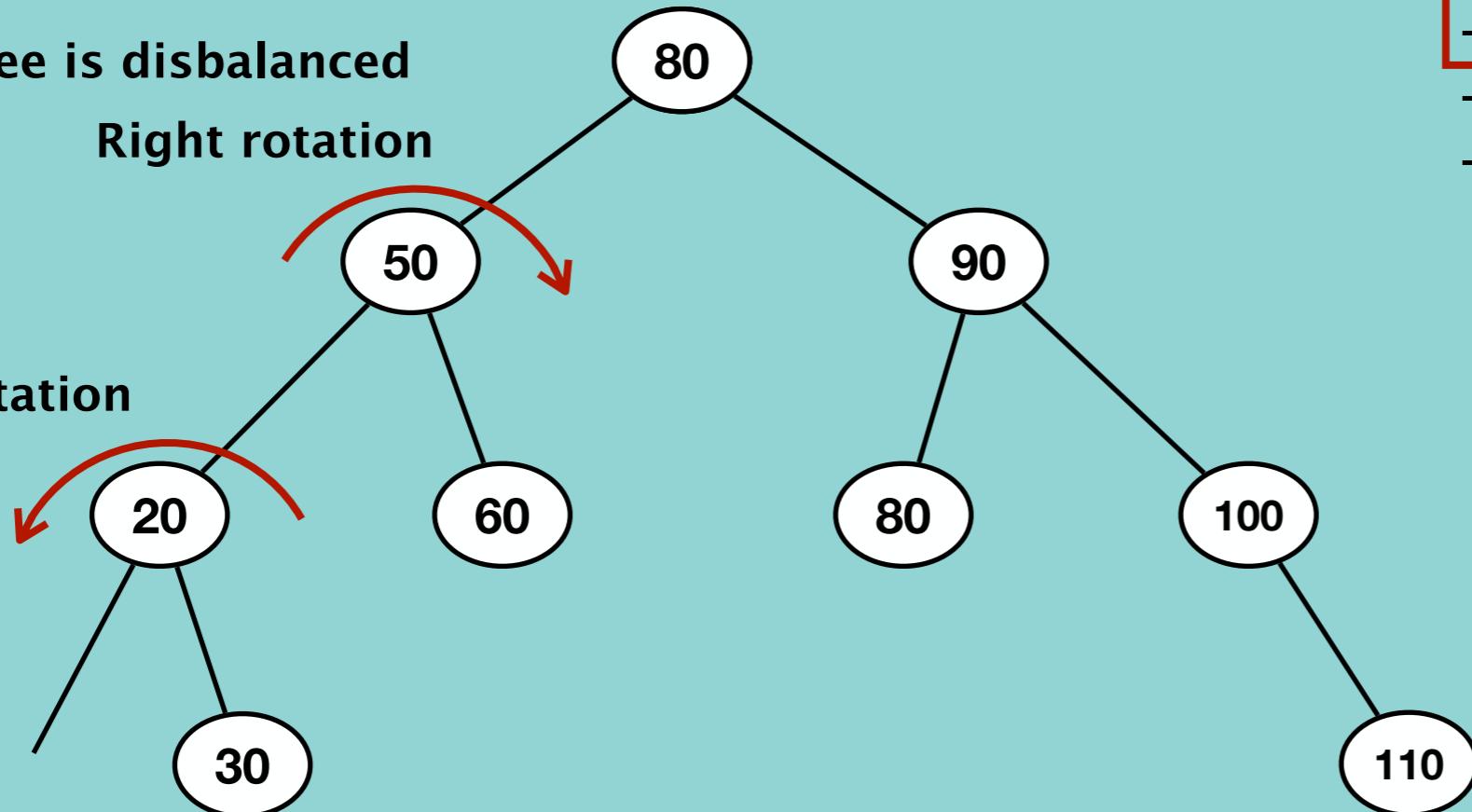
# Delete a node from AVL Tree (all together)

Delete 60

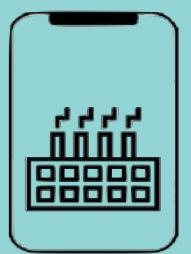
The tree is disbalanced

Right rotation

Left rotation



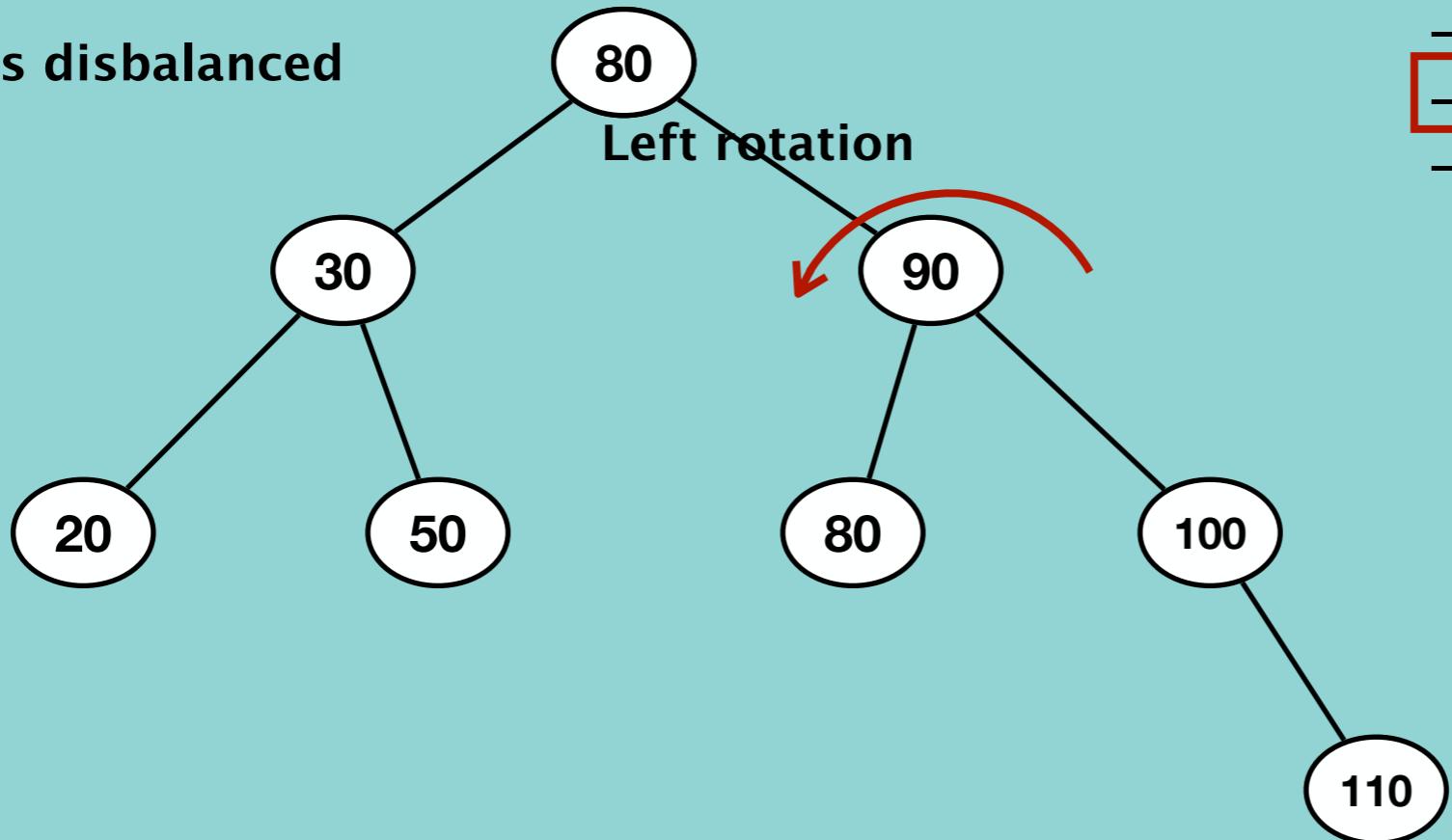
- Left Left Condition (LL)
- Left Right Condition (LR) (highlighted)
- Right Right Condition (RR)
- Right Left Condition (RL)



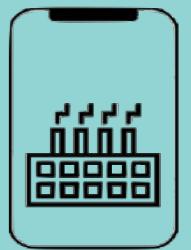
# Delete a node from AVL Tree (all together)

Delete 80

The tree is disbalanced

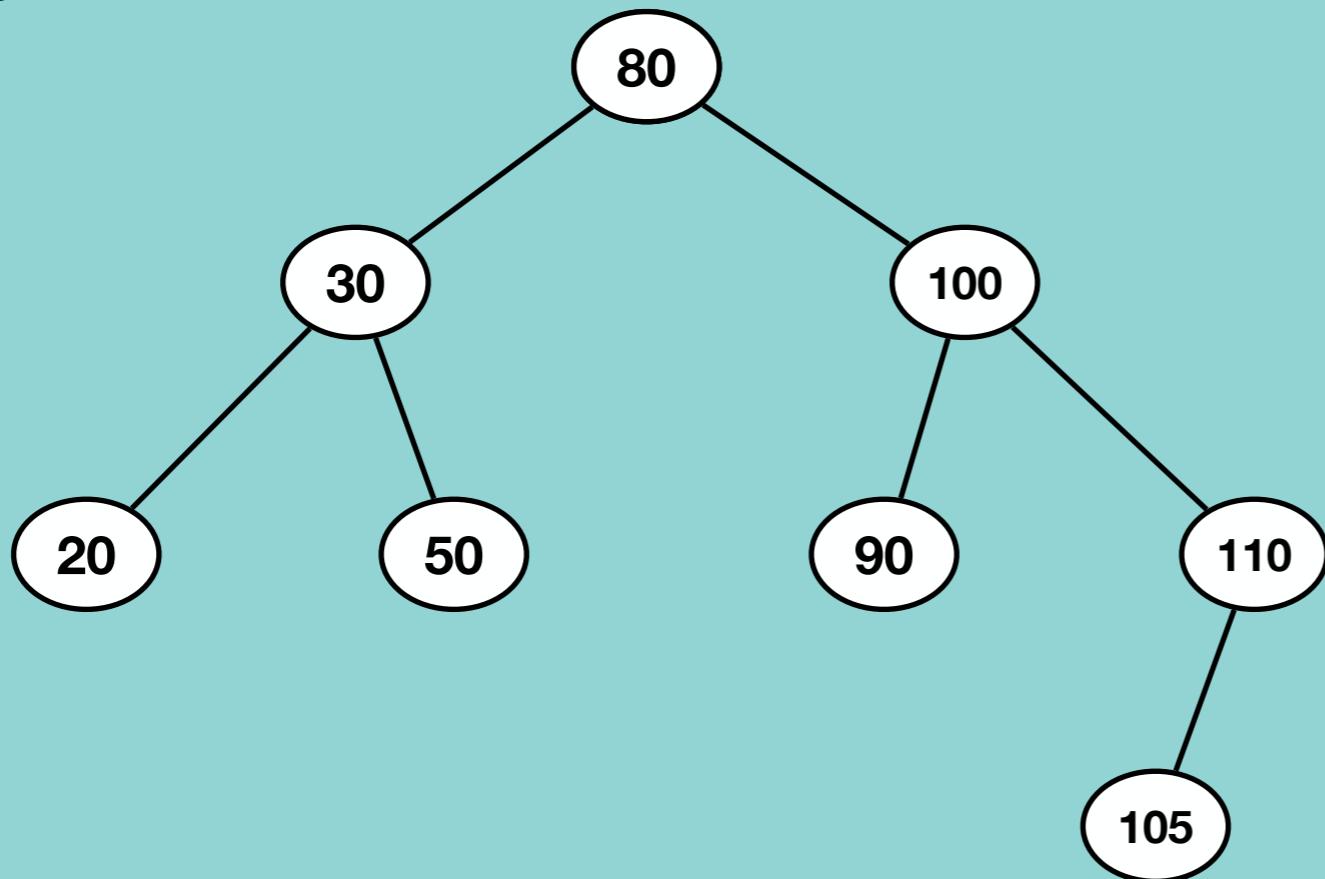


- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR) (highlighted)
- Right Left Condition (RL)

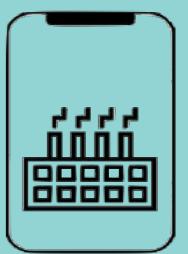


# Delete a node from AVL Tree (all together)

Insert 105

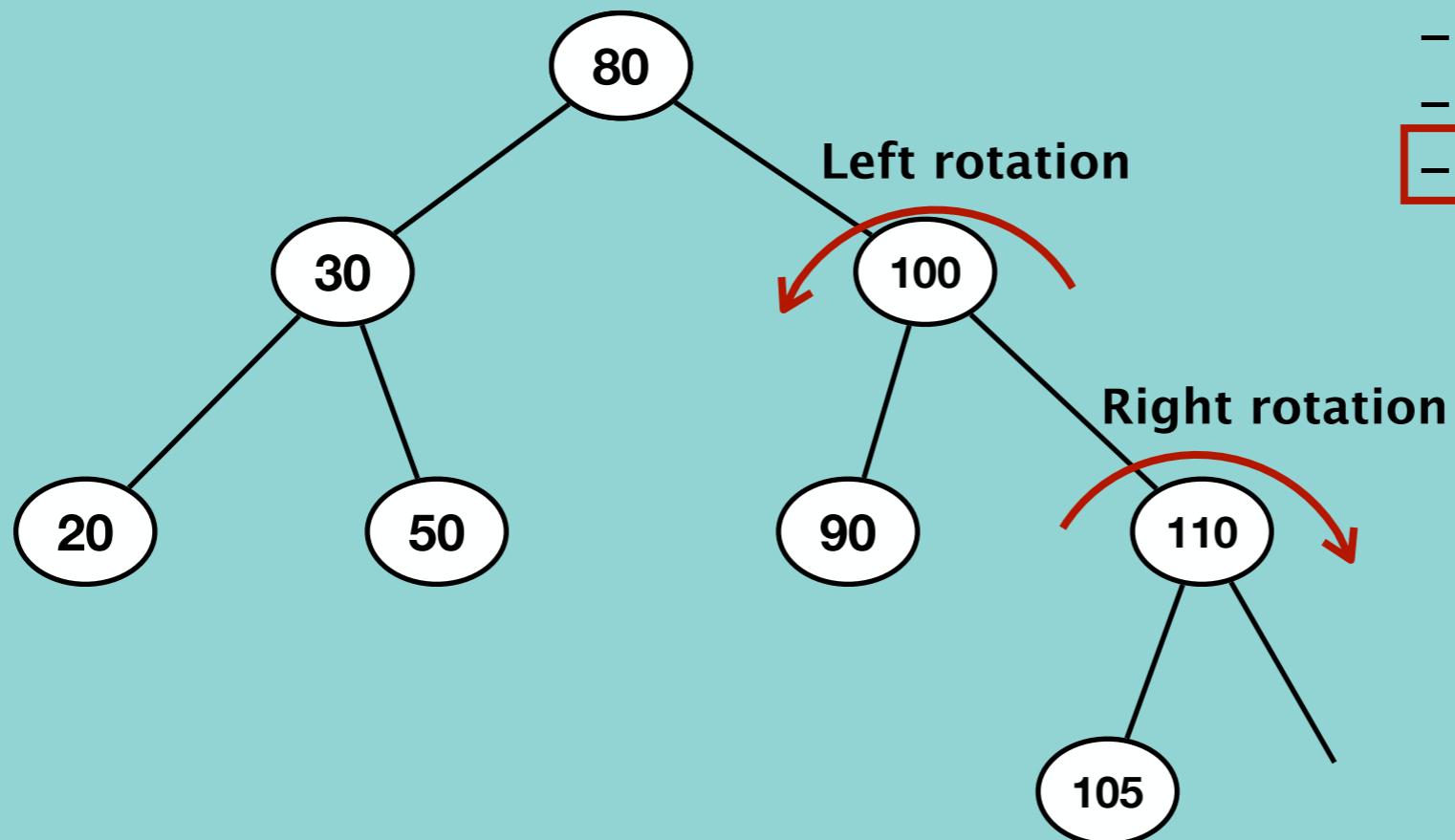


- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)

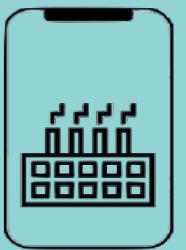


# Delete a node from AVL Tree (all together)

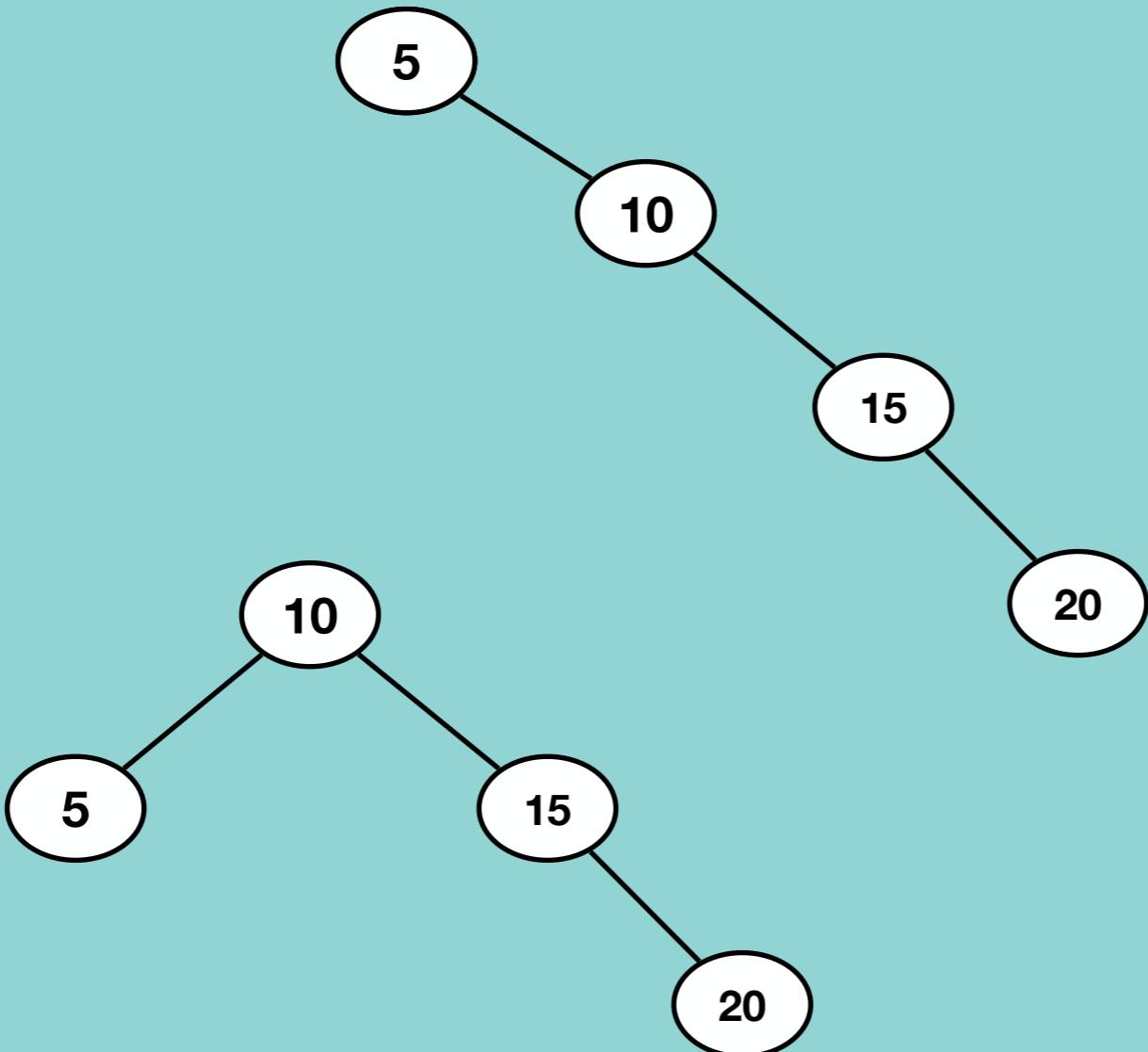
Delete 90



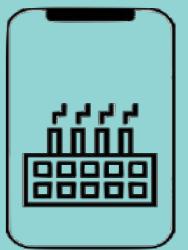
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



# Delete a node from AVL Tree (all together)



- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)

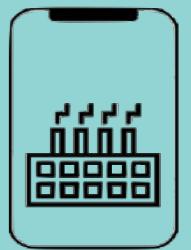
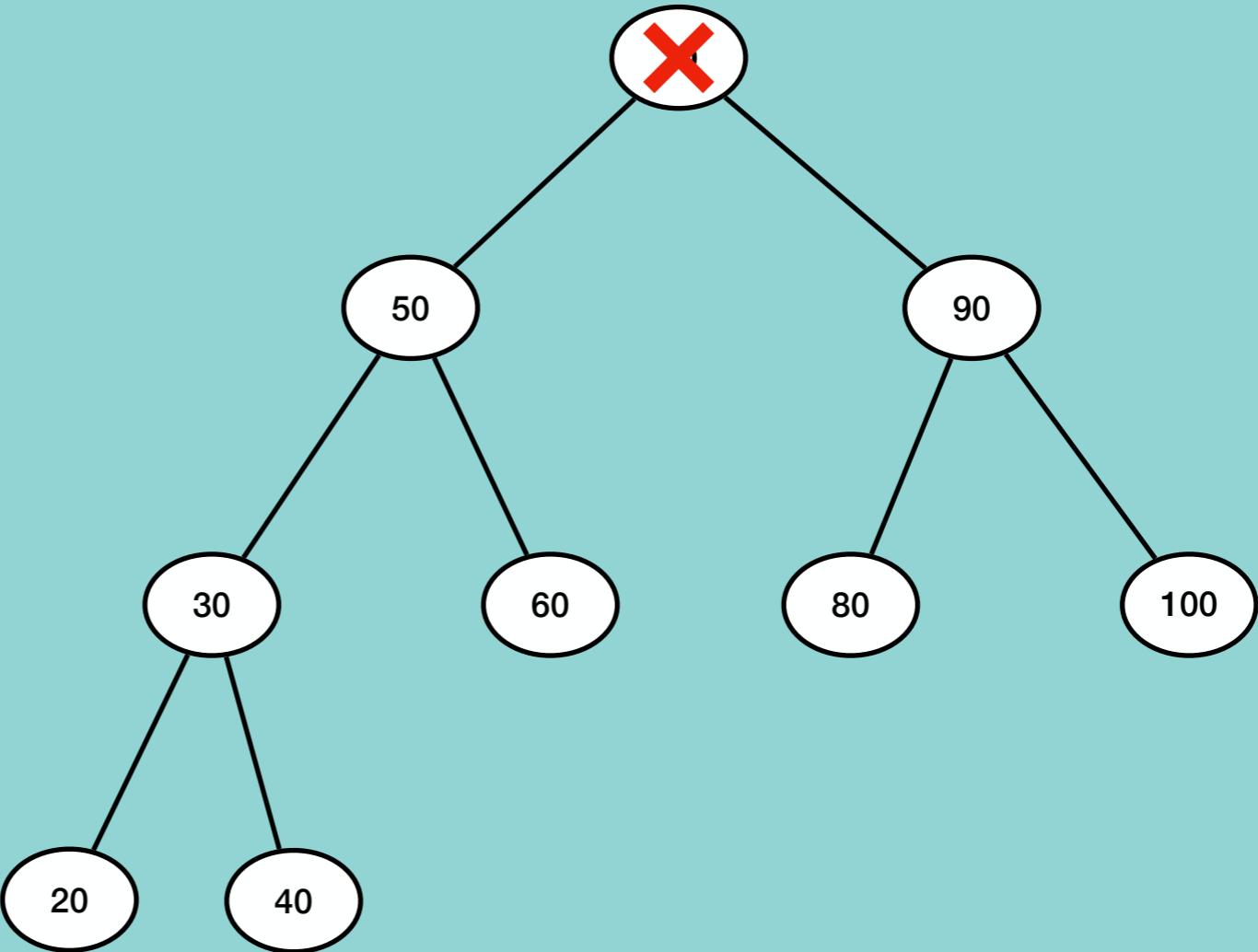


# Delete entire AVL Tree

```
rootNode = None
```

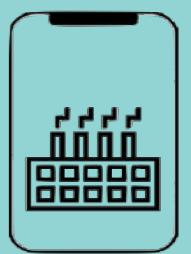
```
rootNode.leftChild = None
```

```
rootNode.rightChild = None
```



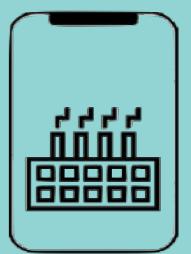
# Time and Space complexity of AVL Tree

	Time complexity	Space complexity
Create AVL	O(1)	O(1)
Insert a node AVL	O(logN)	O(logN)
Traverse AVL	O(N)	O(N)
Search for a node AVL	O(logN)	O(logN)
Delete node from AVL	O(logN)	O(logN)
Delete Entire AVL	O(1)	O(1)



# Binary Search Tree vs AVL

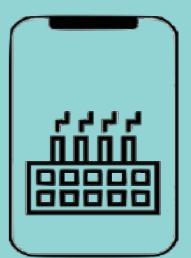
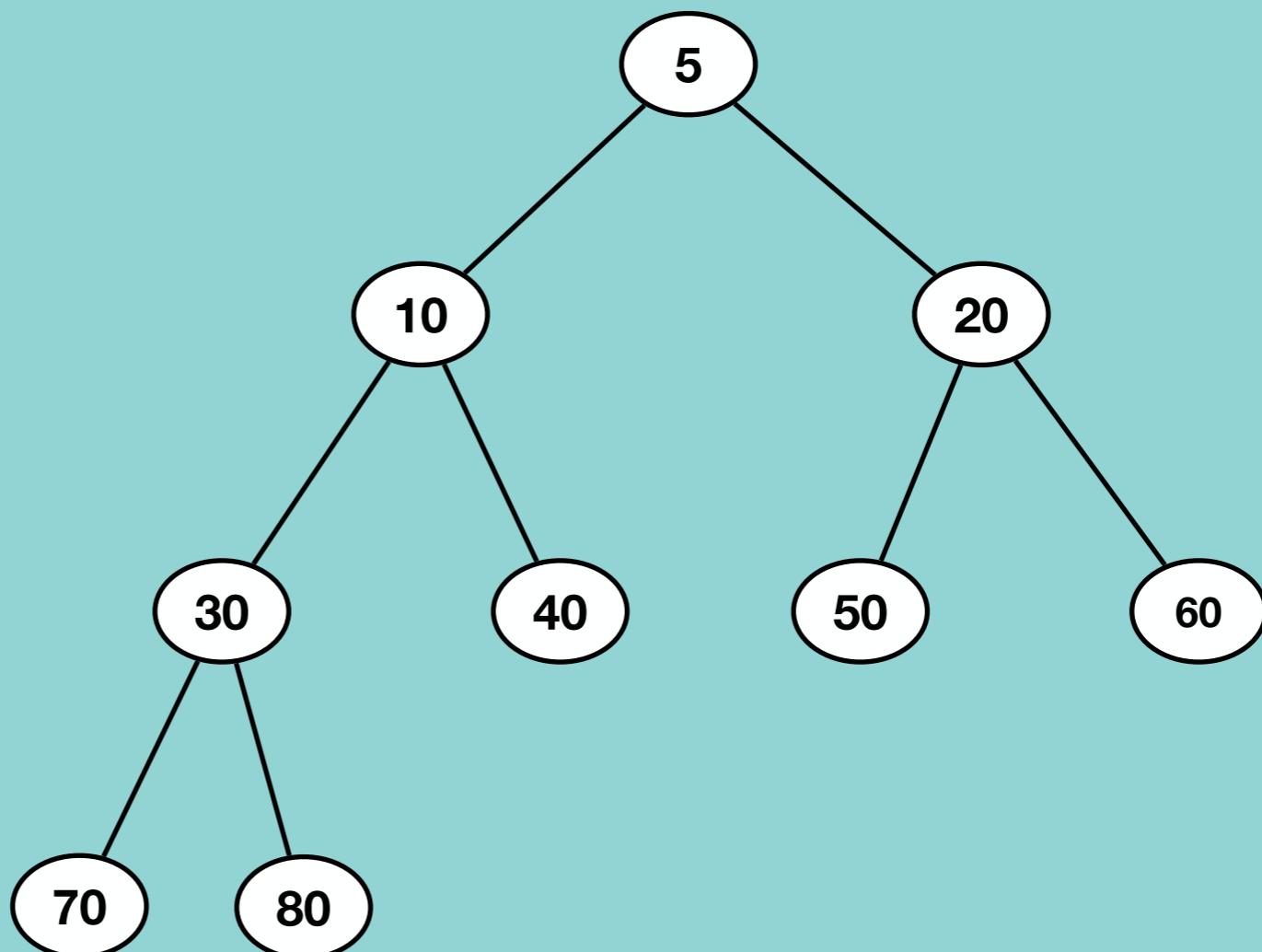
	BST	AVL
Create Tree	O(1)	O(1)
Insert a node Tree	O(N)	O(logN)
Traverse Tree	O(N)	O(N)
Search for a node Tree	O(N)	O(logN)
Delete node from Tree	O(N)	O(logN)
Delete Entire Tree	O(1)	O(1)



# What is a Binary Heap?

A Binary Heap is a Binary Tree with following properties.

- A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree.
- It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.



# Why we need a Binary Heap?

Find the minimum or maximum number among a set of numbers in  $\log N$  time. And also we want to make sure that inserting additional numbers does not take more than  $O(\log N)$  time

## Possible Solutions

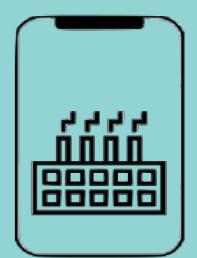
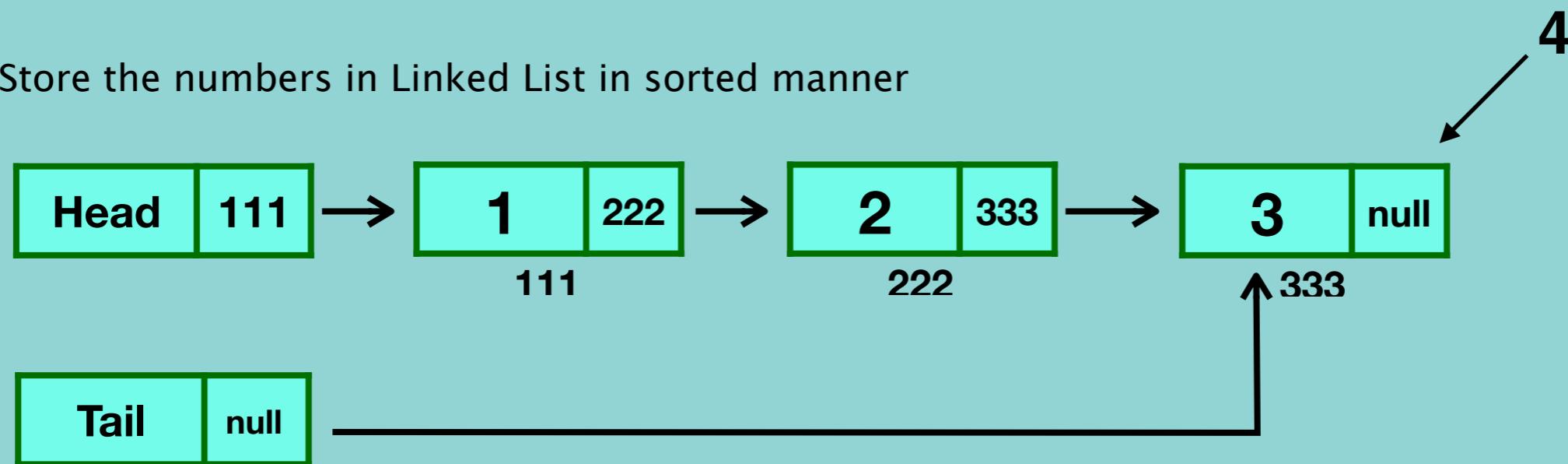
- Store the numbers in sorted Array



Find minimum:  $O(1)$

Insertion:  $O(n)$

- Store the numbers in Linked List in sorted manner

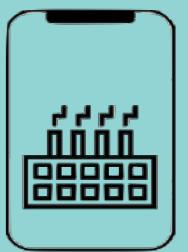


# Why we need a Binary Heap?

Find the minimum or maximum number among a set of numbers in  $\log N$  time. And also we want to make sure that inserting additional numbers does not take more than  $O(\log N)$  time

## Practical Use

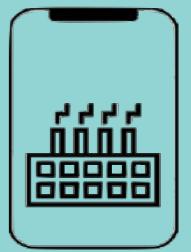
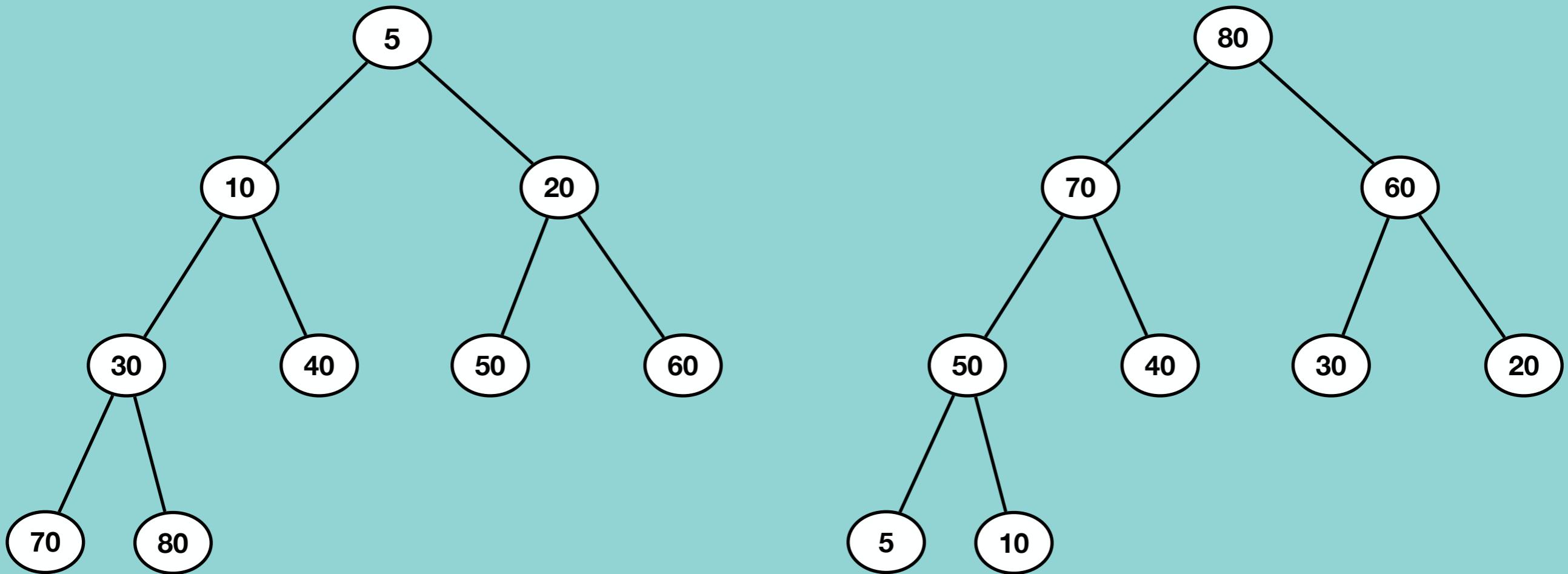
- Prim's Algorithm
- Heap Sort
- Priority Queue



# Types of Binary Heap

**Min heap** – the value of each node is less than or equal to the value of both its children.

**Max heap** – it is exactly the opposite of min heap that is the value of each node is more than or equal to the value of both its children.

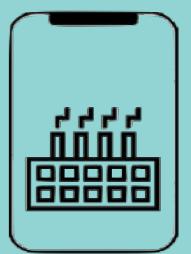
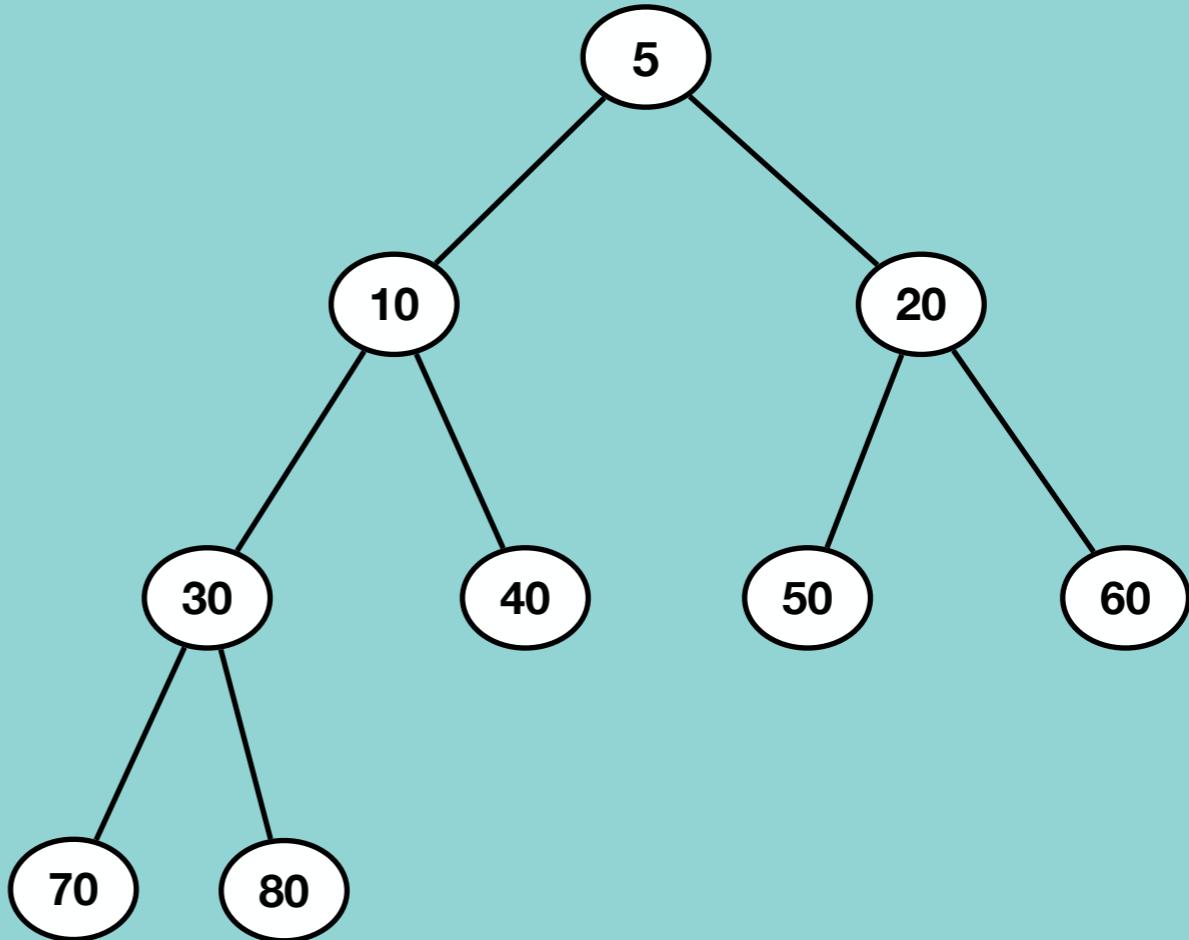


# Common operations on Binary Heap

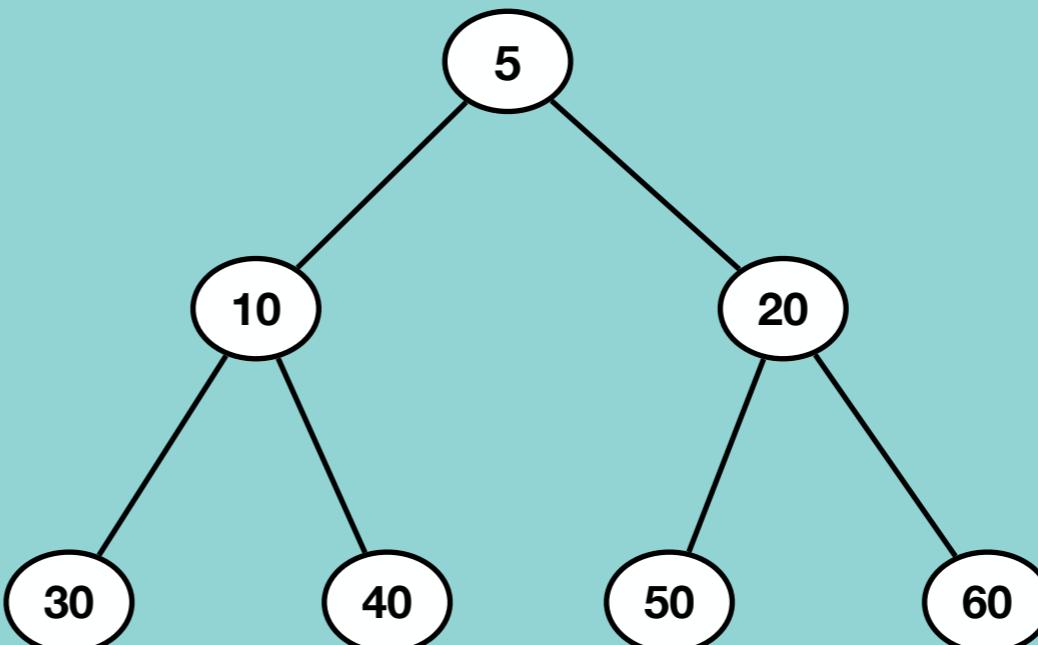
- Creation of Binary Heap,
- Peek top of Binary Heap
- Extract Min / Extract Max
- Traversal of Binary Heap
- Size of Binary Heap
- Insert value in Binary Heap
- Delete the entire Binary heap

## Implementation Options

- Array Implementation
- Reference /pointer Implementation



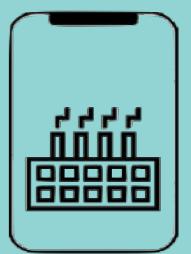
# Common operations on Binary Heap



0	1	2	3	4	5	6	7	8
✗	5	10	20	30	40	50	60	

Left child = `cell[2x]`

Right child = `cell[2x+1]`

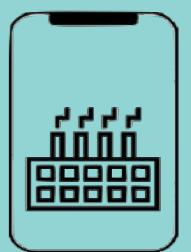


# Common operations on Binary Heap

- Creation of Binary Heap

Initialize fixed size List

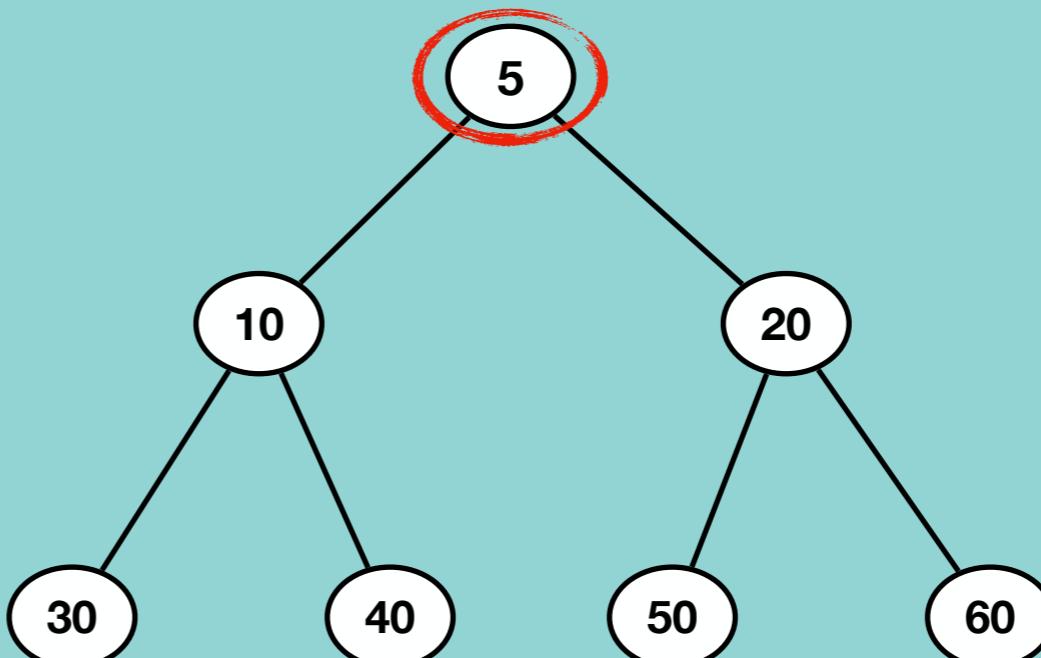
set size of Binary Heap to 0



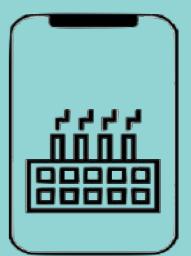
# Common operations on Binary Heap

- Peek of Binary Heap

Return List[1]



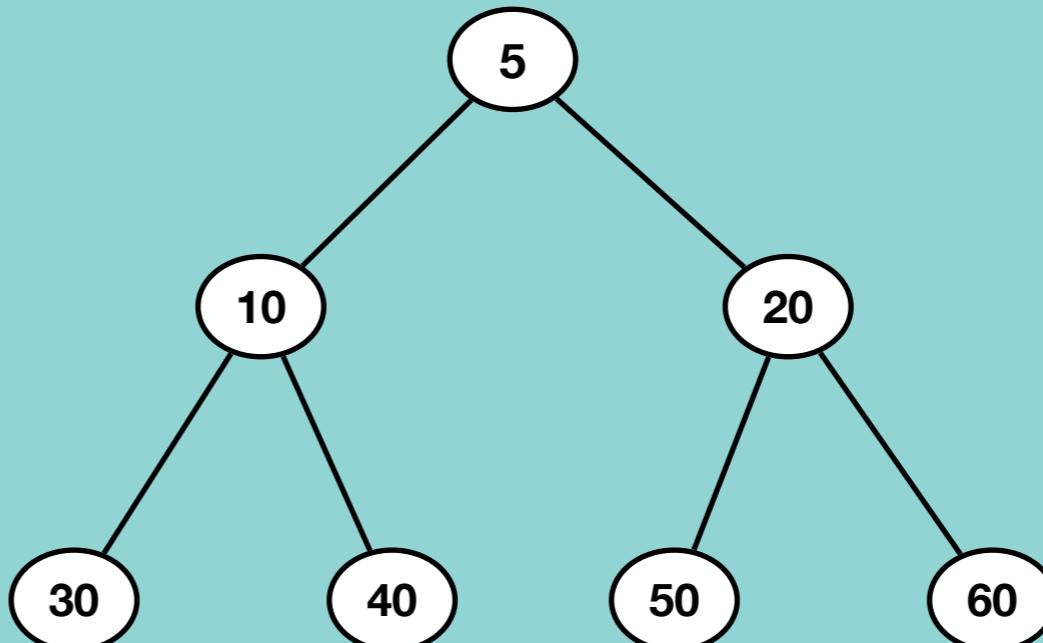
0	1	2	3	4	5	6	7	8
✗	5	10	20	30	40	50	60	



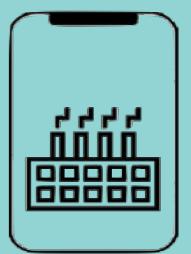
# Common operations on Binary Heap

- Size Binary Heap

Return number of filled cells

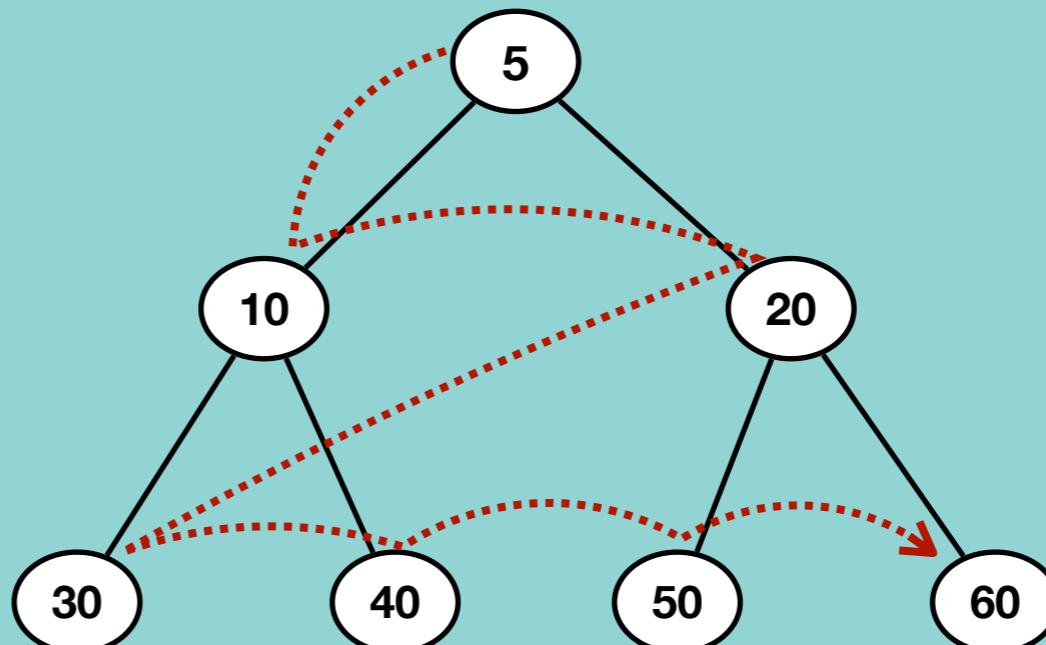


0	1	2	3	4	5	6	7	8
✗	5	10	20	30	40	50	60	

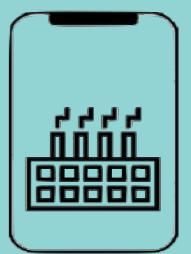


# Common operations on Binary Heap

- Level Order Traversal

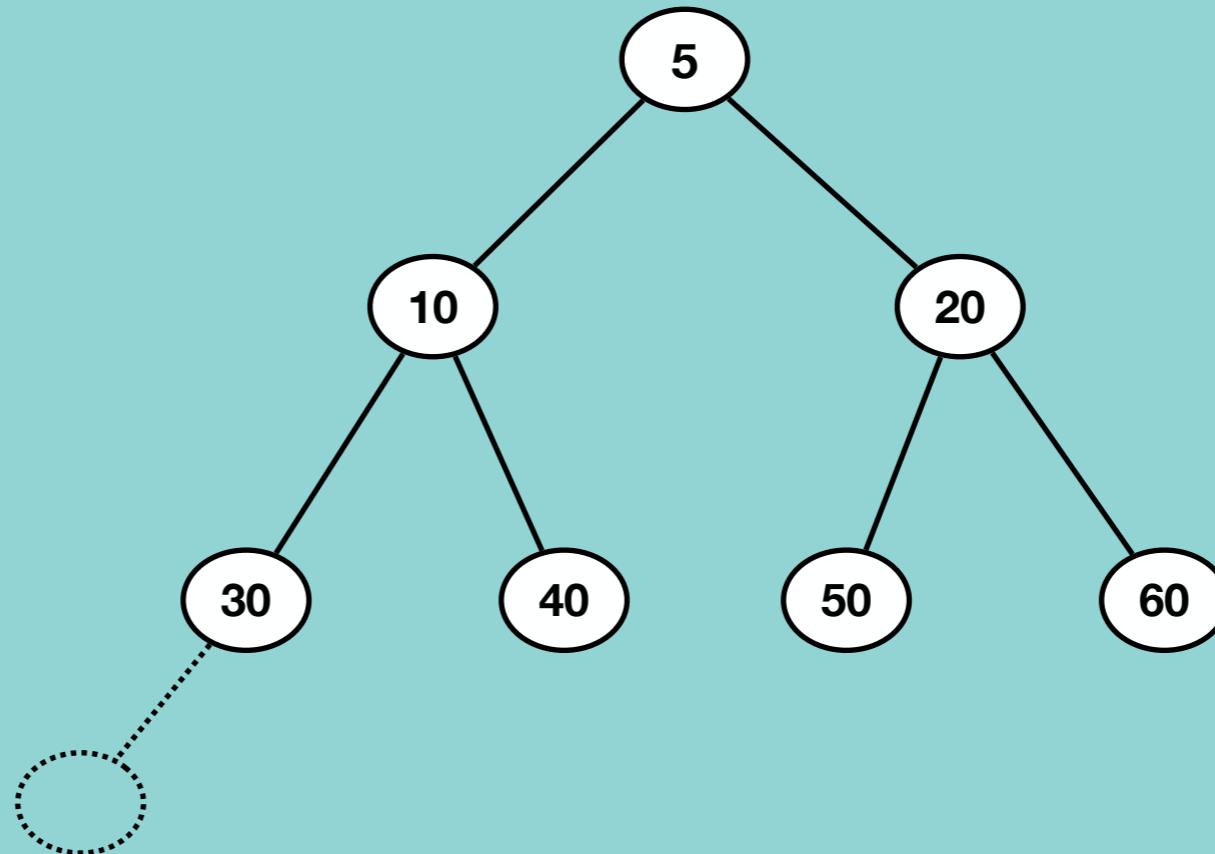


0	1	2	3	4	5	6	7	8
✗	5	10	20	30	40	50	60	

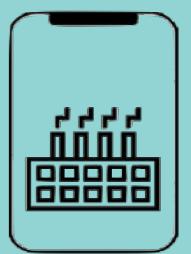


# Insert a node to Binary Heap

1

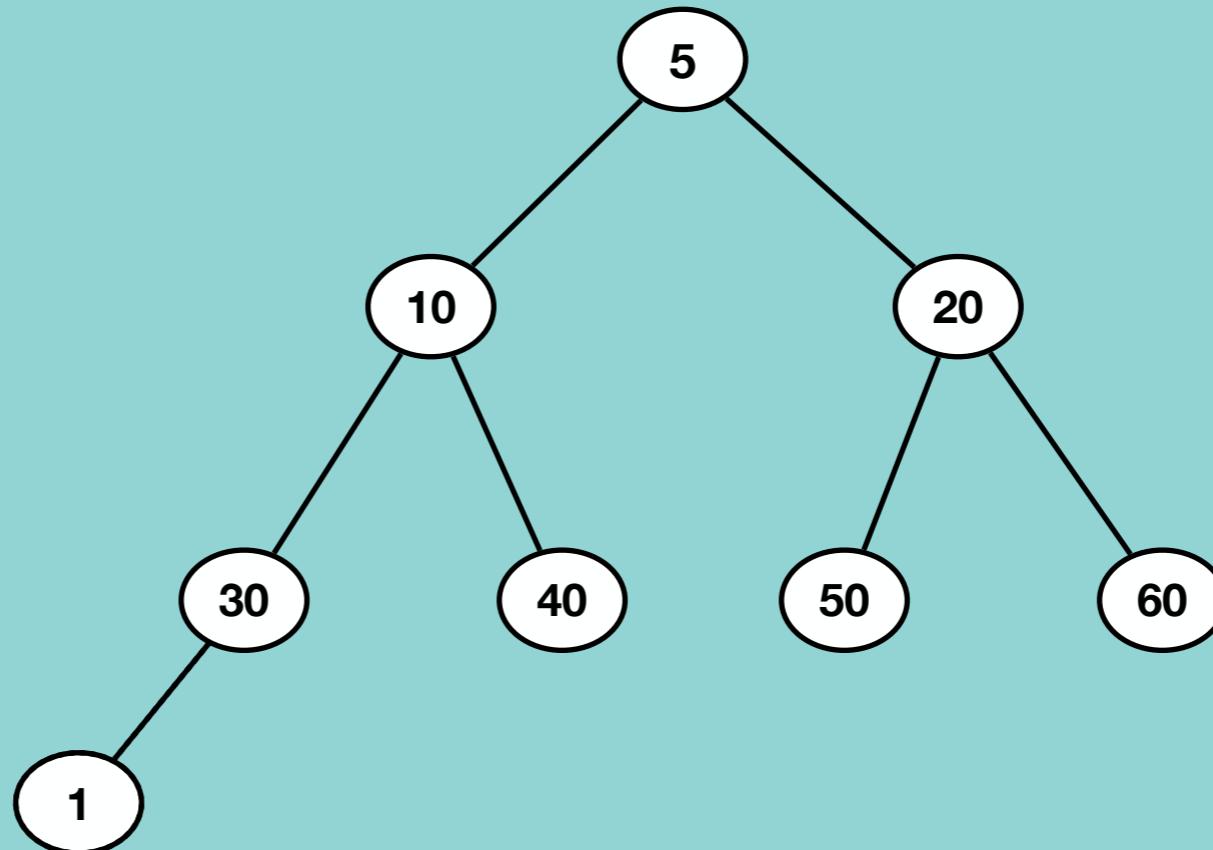


0	1	2	3	4	5	6	7	8
X	5	10	20	30	40	50	60	

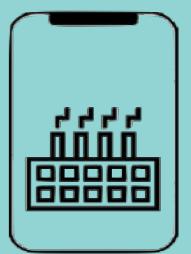


# Insert a node to Binary Heap

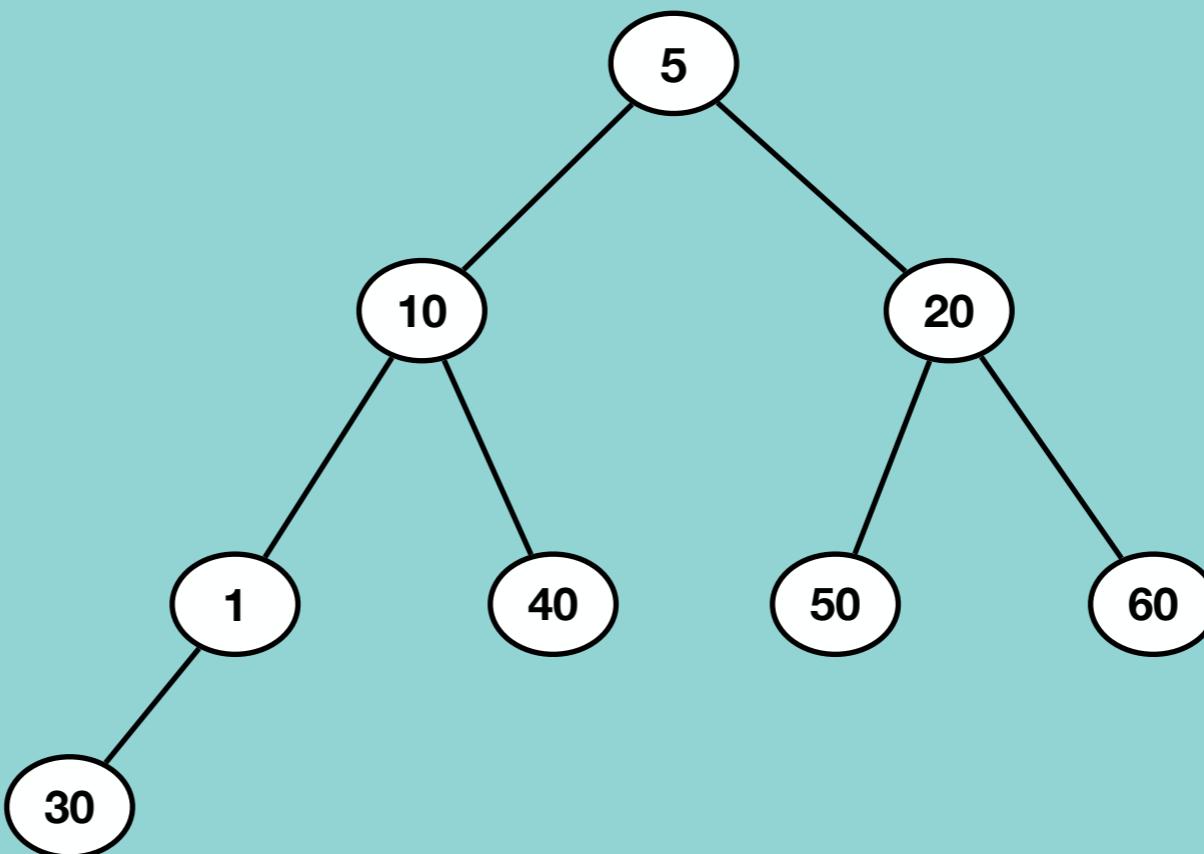
1



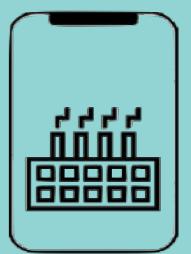
0	1	2	3	4	5	6	7	8
X	5	10	20	1	40	50	60	30



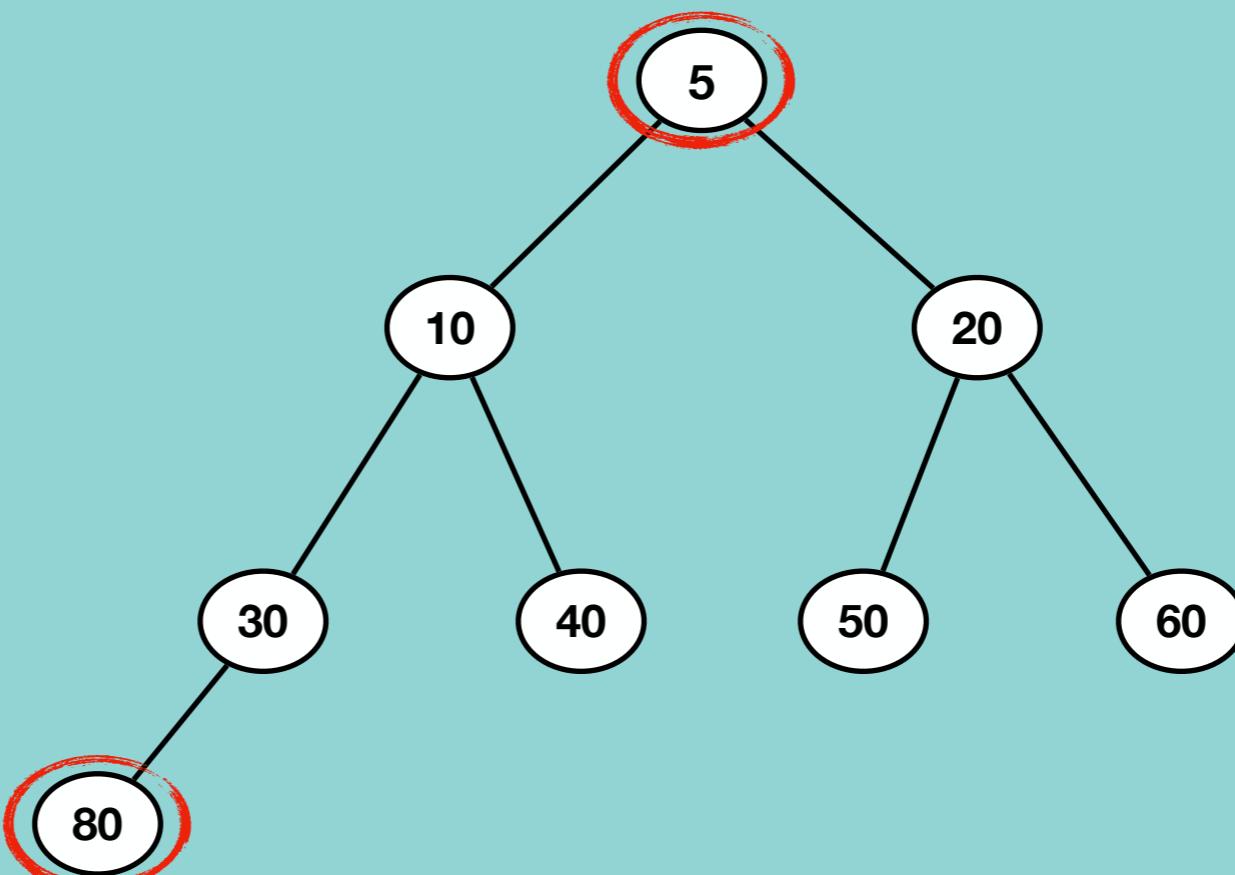
# Insert a node to Binary Heap



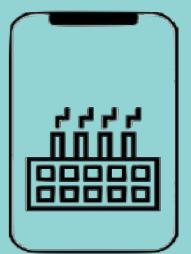
0	1	2	3	4	5	6	7	8
✗	5	5	20	10	40	50	60	30



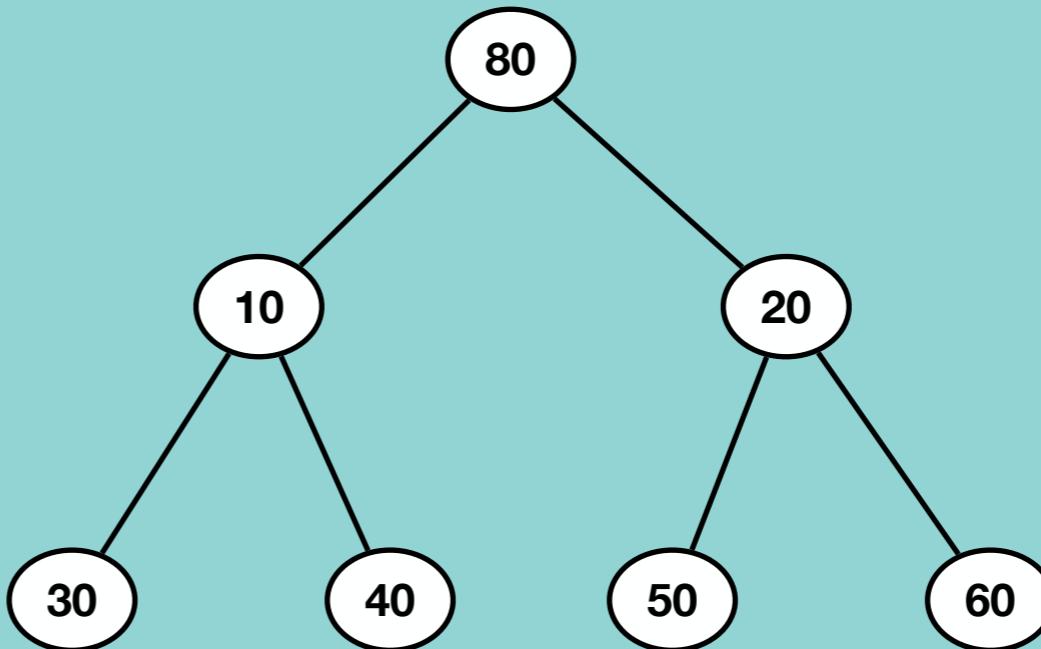
# Extract a node from Binary Heap



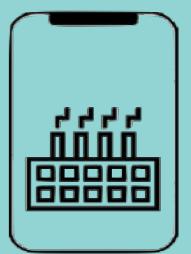
0	1	2	3	4	5	6	7	8
✗	5	10	20	30	40	50	60	80



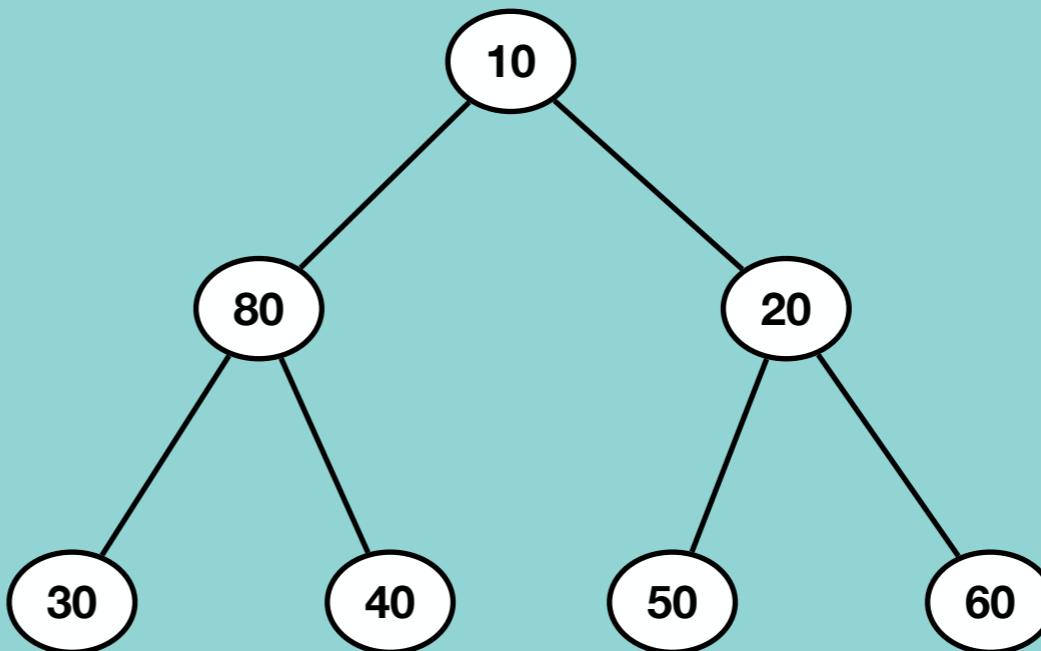
# Extract a node from Binary Heap



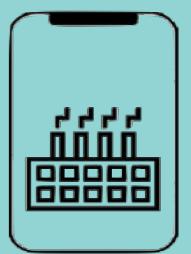
0	1	2	3	4	5	6	7	8
✗	80	10	20	30	40	50	60	



# Extract a node from Binary Heap

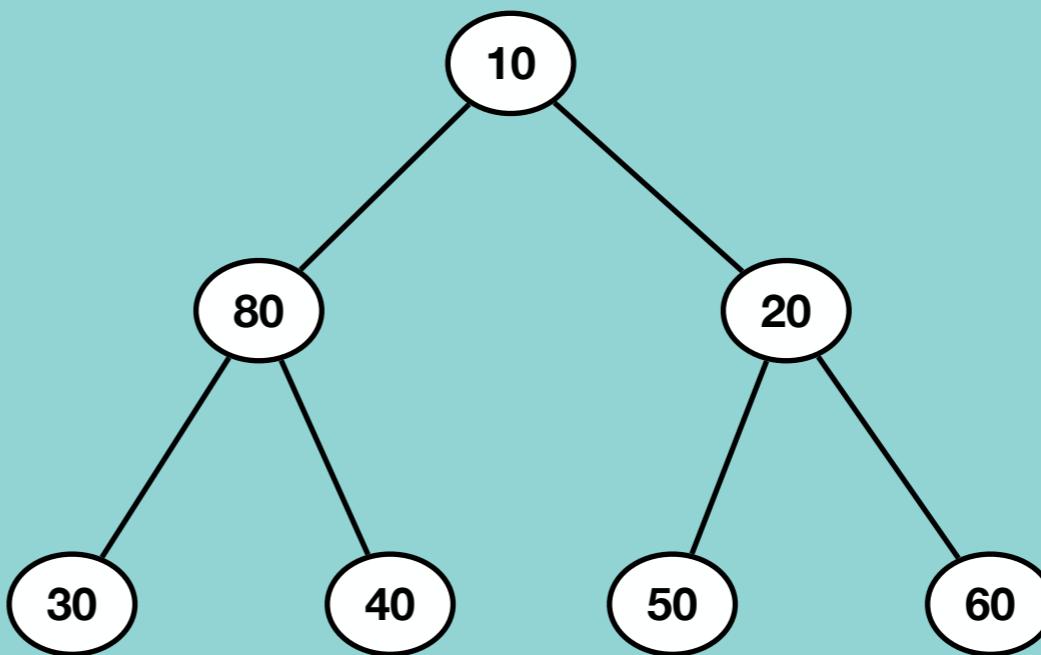


0	1	2	3	4	5	6	7	8
✗	10	80	20	30	40	50	60	

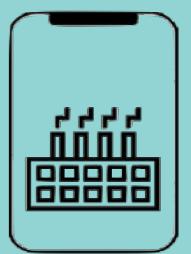


# Delete entire Binary Heap

customList = None

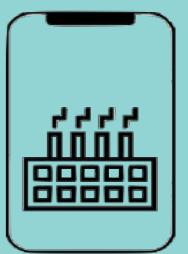


0	1	2	3	4	5	6	7	8
X	10	80	20	30	40	50	60	



# Time and Space complexity of Binary Heap

	Time complexity	Space complexity
Create Binary Heap	O(1)	O(N)
Peek of Heap	O(1)	O(1)
Size of Heap	O(1)	O(1)
Traversal of Heap	O(N)	O(1)
Insert a node to Binary Heap	O(logN)	O(logN)
Extract a node from Binary Heap	O(logN)	O(logN)
Delete Entire AVL	O(1)	O(1)



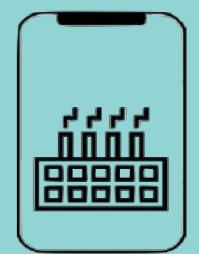
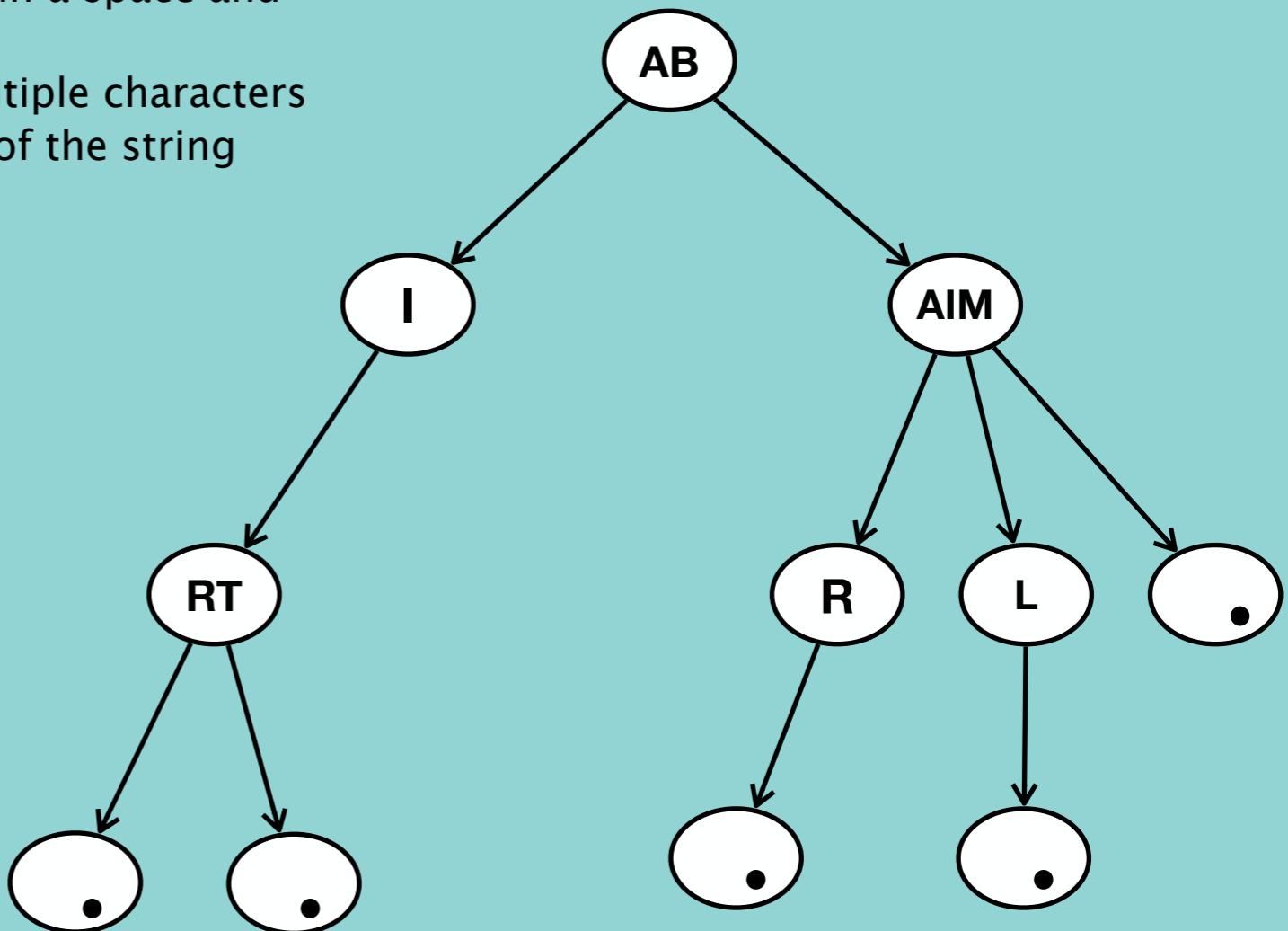
# What is a Trie?

A Trie is a tree-based data structure that organizes information in a hierarchy.

## Properties:

- It is typically used to store or search strings in a space and time efficient way.
- Any node in trie can store non repetitive multiple characters
- Every node stores link of the next character of the string
- Every node keeps track of ‘end of string’

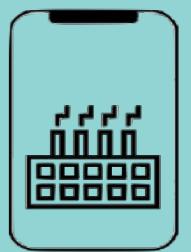
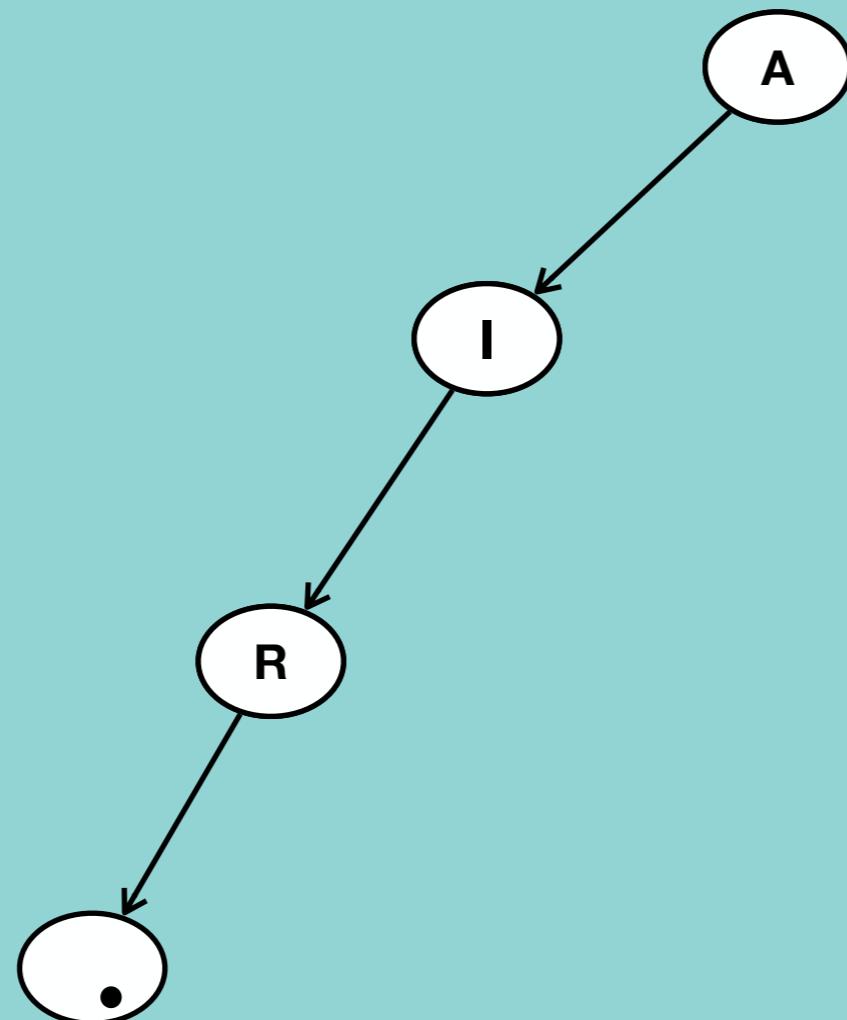
AIR, AIT, BAR, BIL, BM



# What is a Trie?

A Trie is a tree-based data structure that organizes information in a hierarchy.

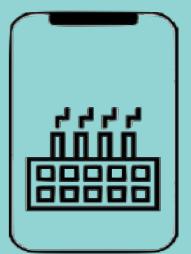
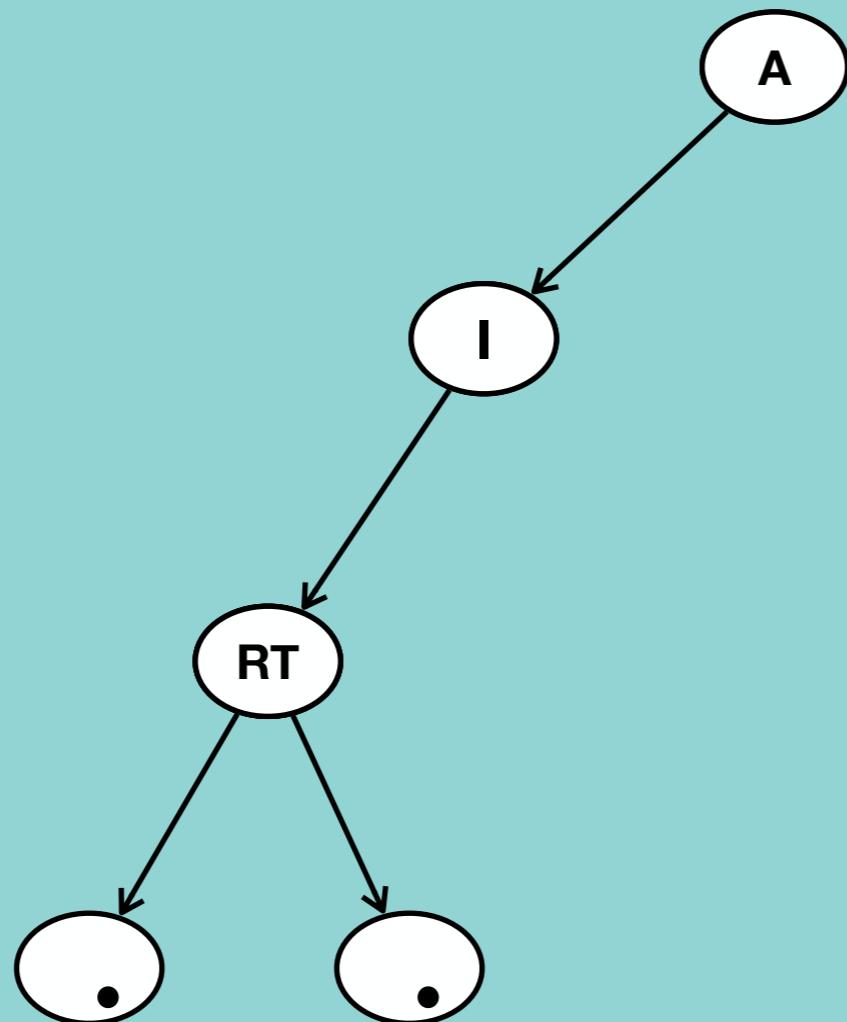
AIR



# What is a Trie?

A Trie is a tree-based data structure that organizes information in a hierarchy.

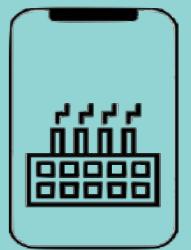
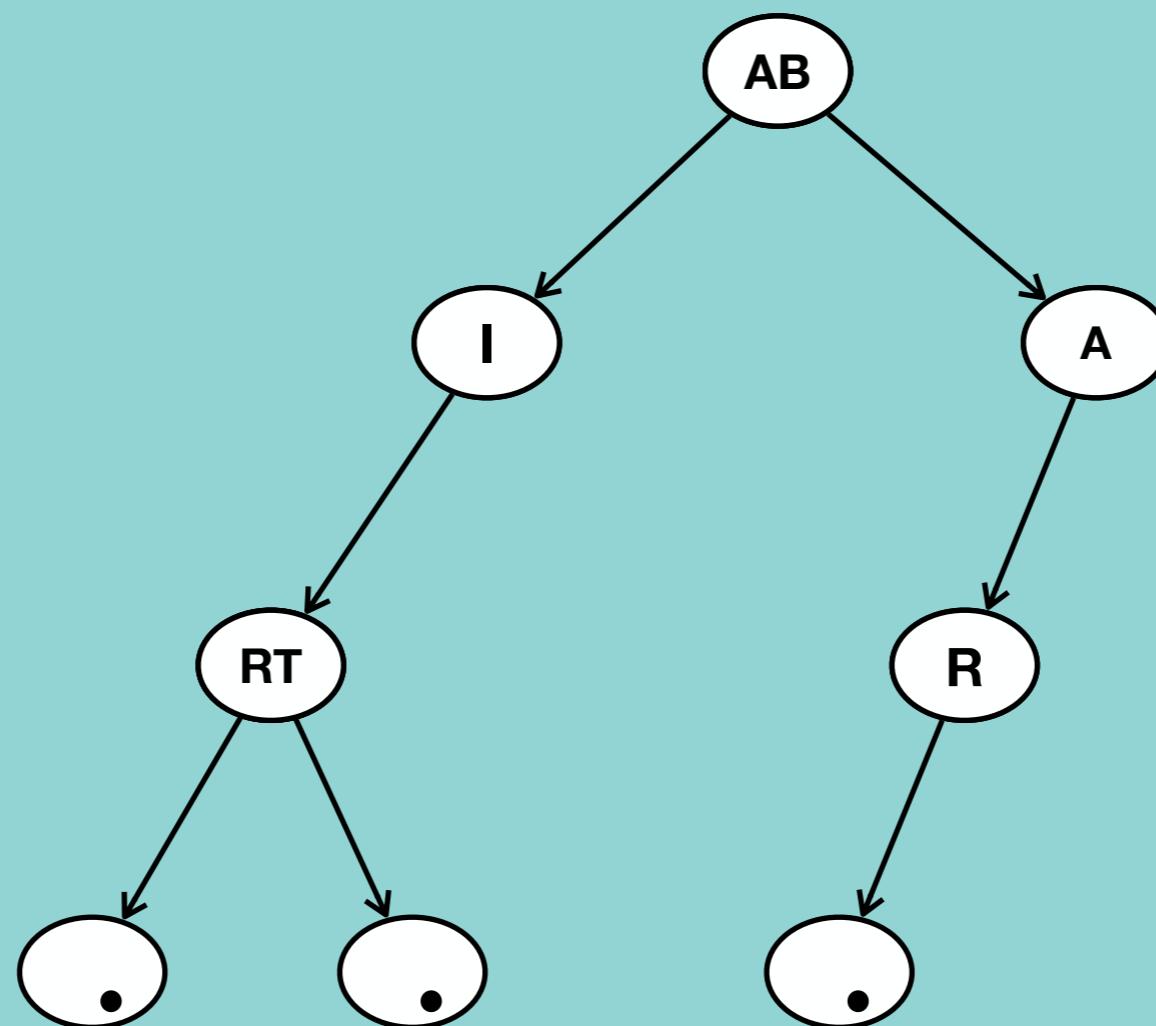
AIT



# What is a Trie?

A Trie is a tree-based data structure that organizes information in a hierarchy.

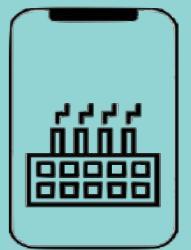
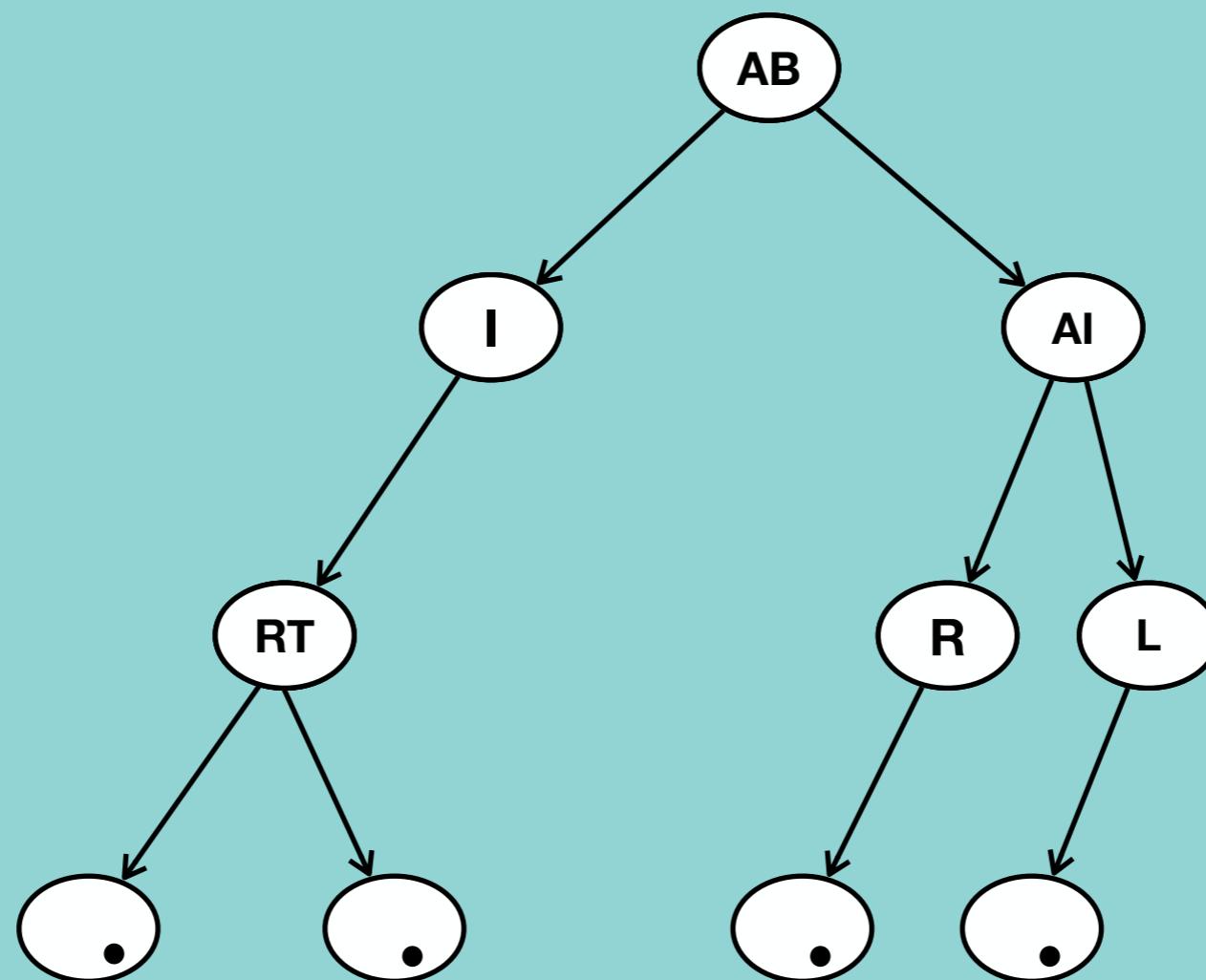
BAR



# What is a Trie?

A Trie is a tree-based data structure that organizes information in a hierarchy.

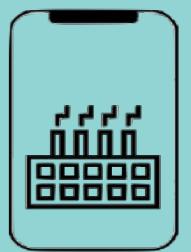
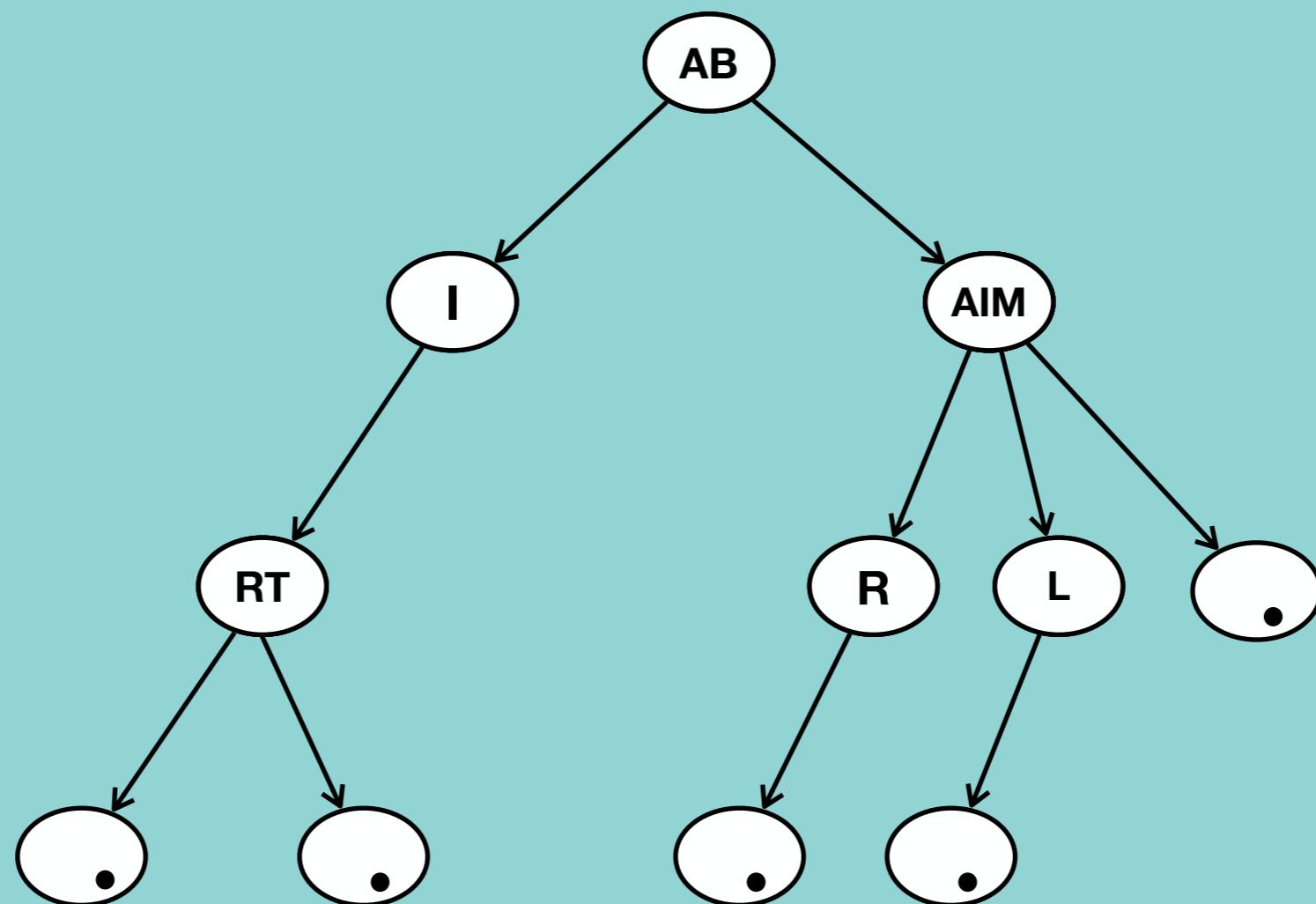
BIL



# What is a Trie?

A Trie is a tree-based data structure that organizes information in a hierarchy.

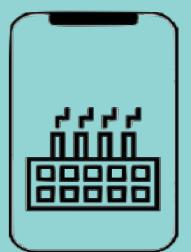
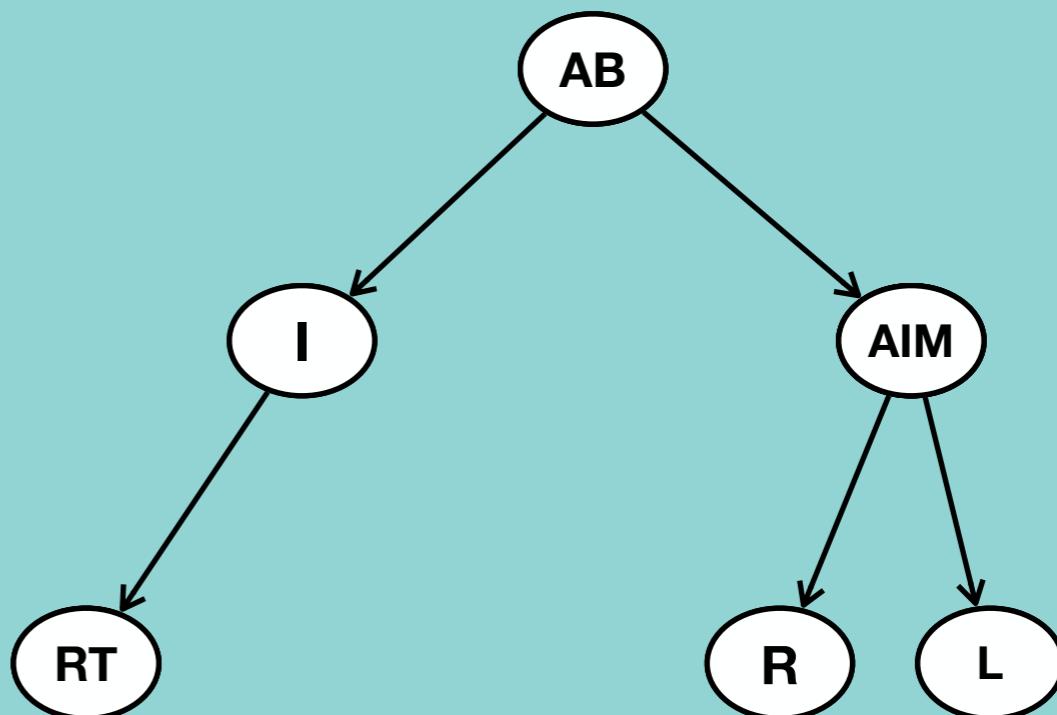
BM



# Why we need Trie?

To solve many standard problems in efficient way

- Spelling checker
- Auto completion

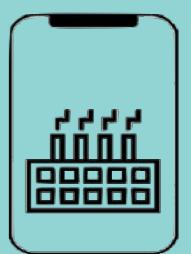
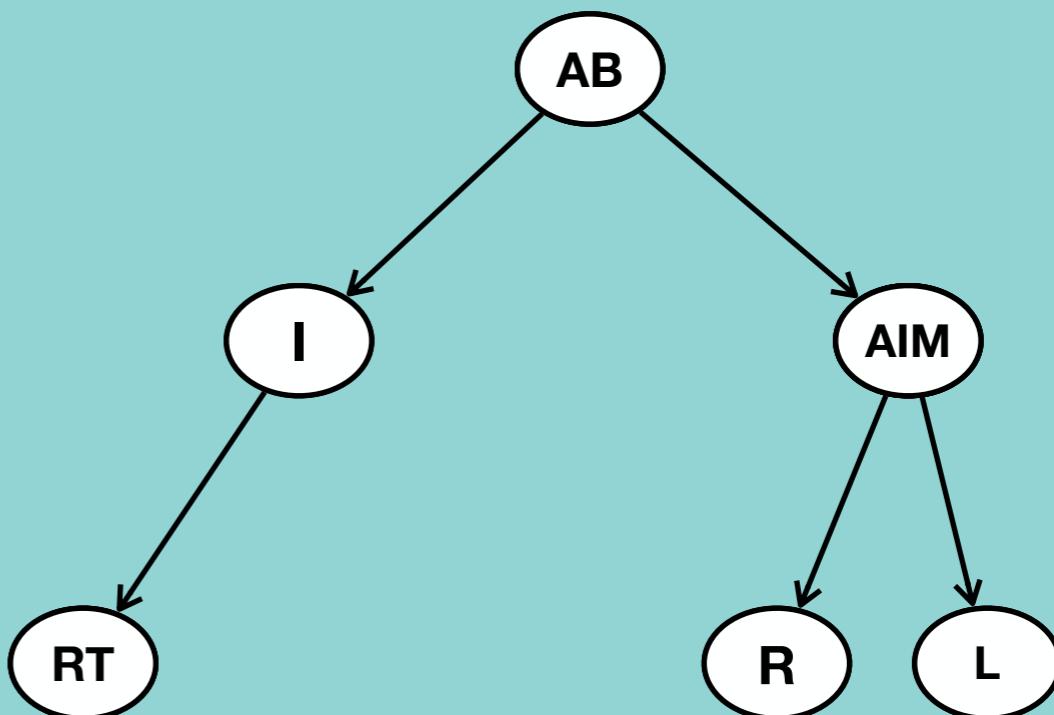


# Why we need Trie?

To solve many standard problems in efficient way

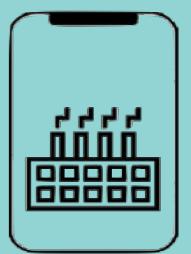
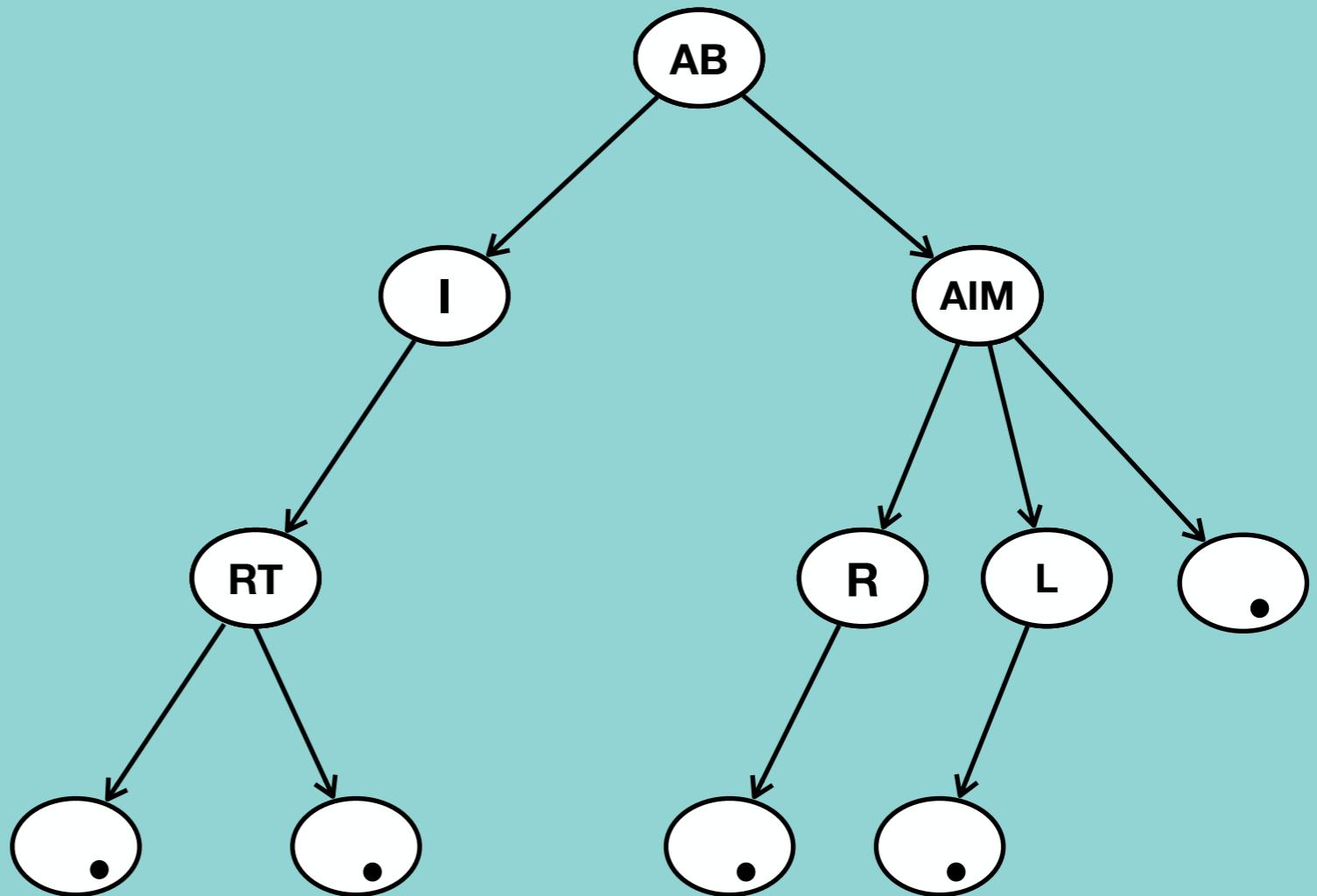
- Spelling checker
- Auto completion

Map	
Characters	Link to Trie Node
End of String	



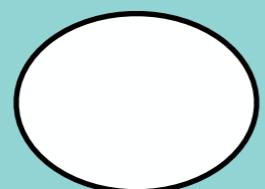
# Common operations on Trie

- Creation of Trie
- Insertion in Trie
- Search for a String in trie
- Deletion from Trie



# Common operations on Trie

## – Creation of Trie

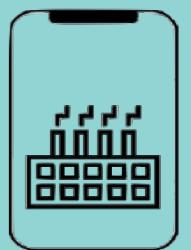


Logical

Initialize Trie() class

Physically

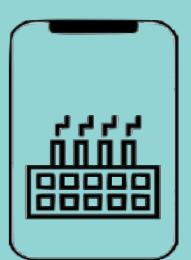
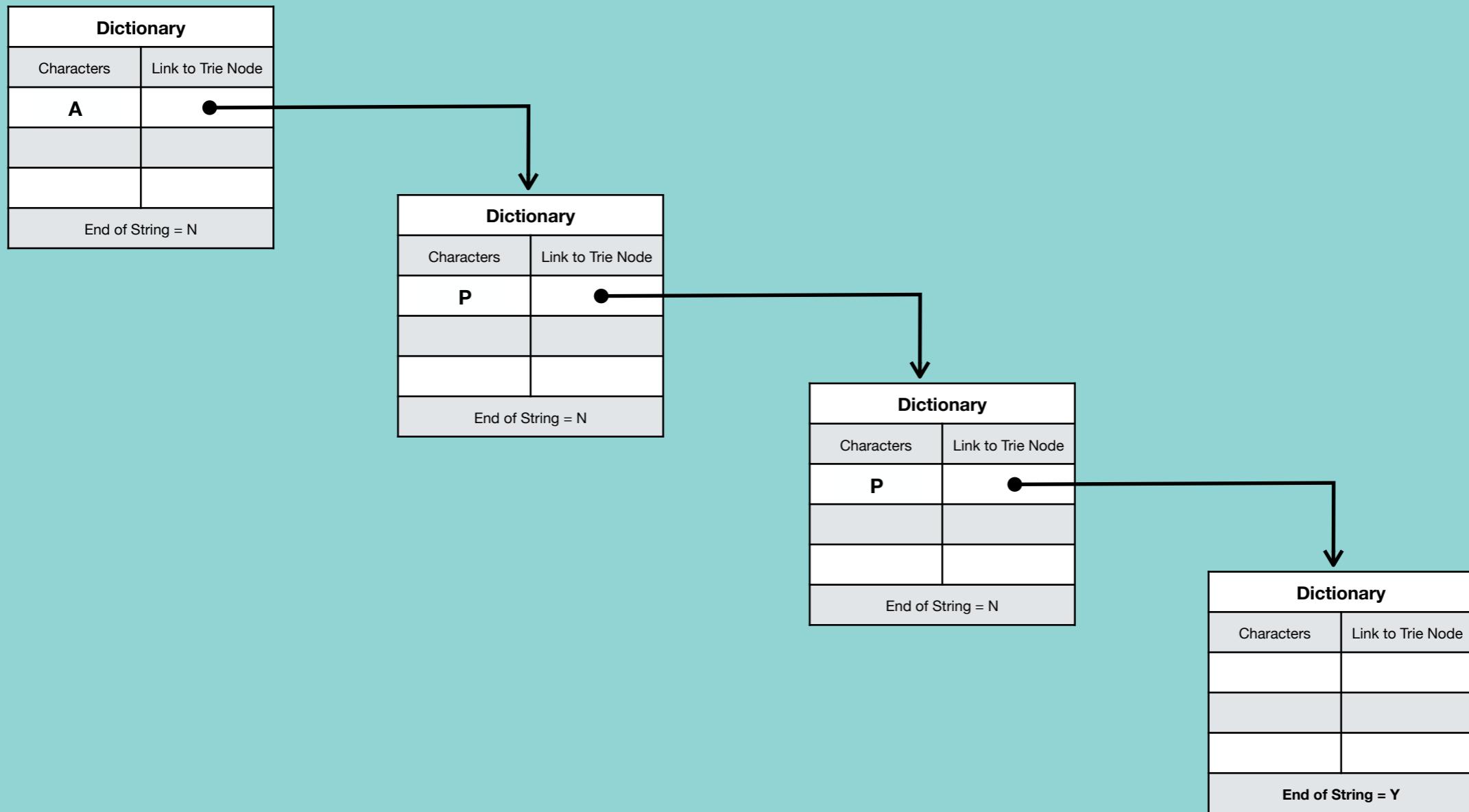
Map	
Characters	Link to Trie Node
End of String	



# Insert a string in a Trie

Case 1 : A Trie is Blank

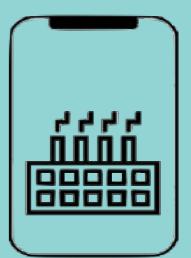
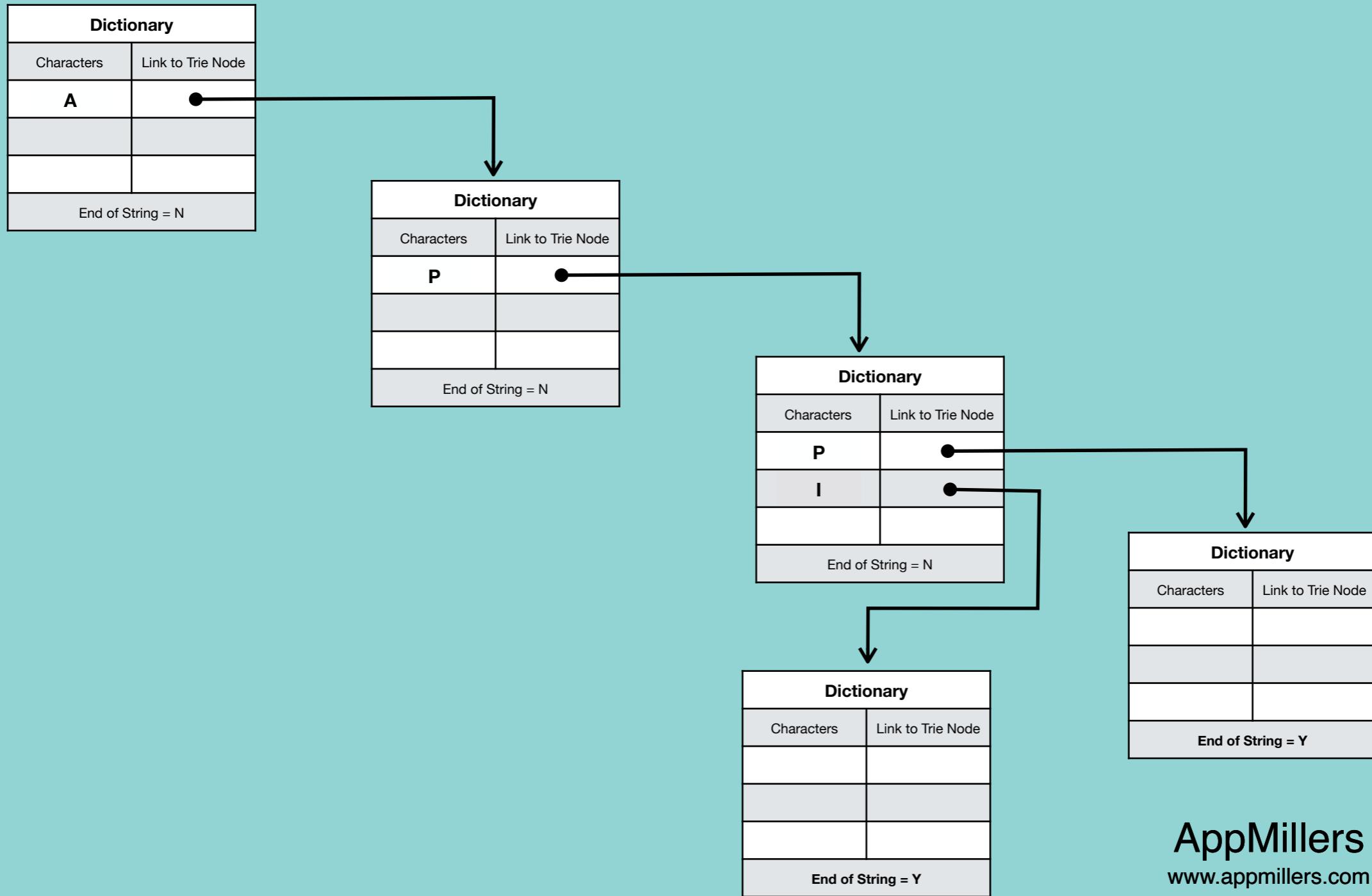
APP



# Insert a string in a Trie

Case 2: New string's prefix is common to another strings prefix

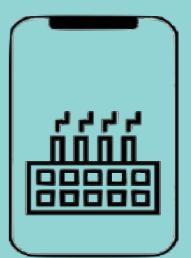
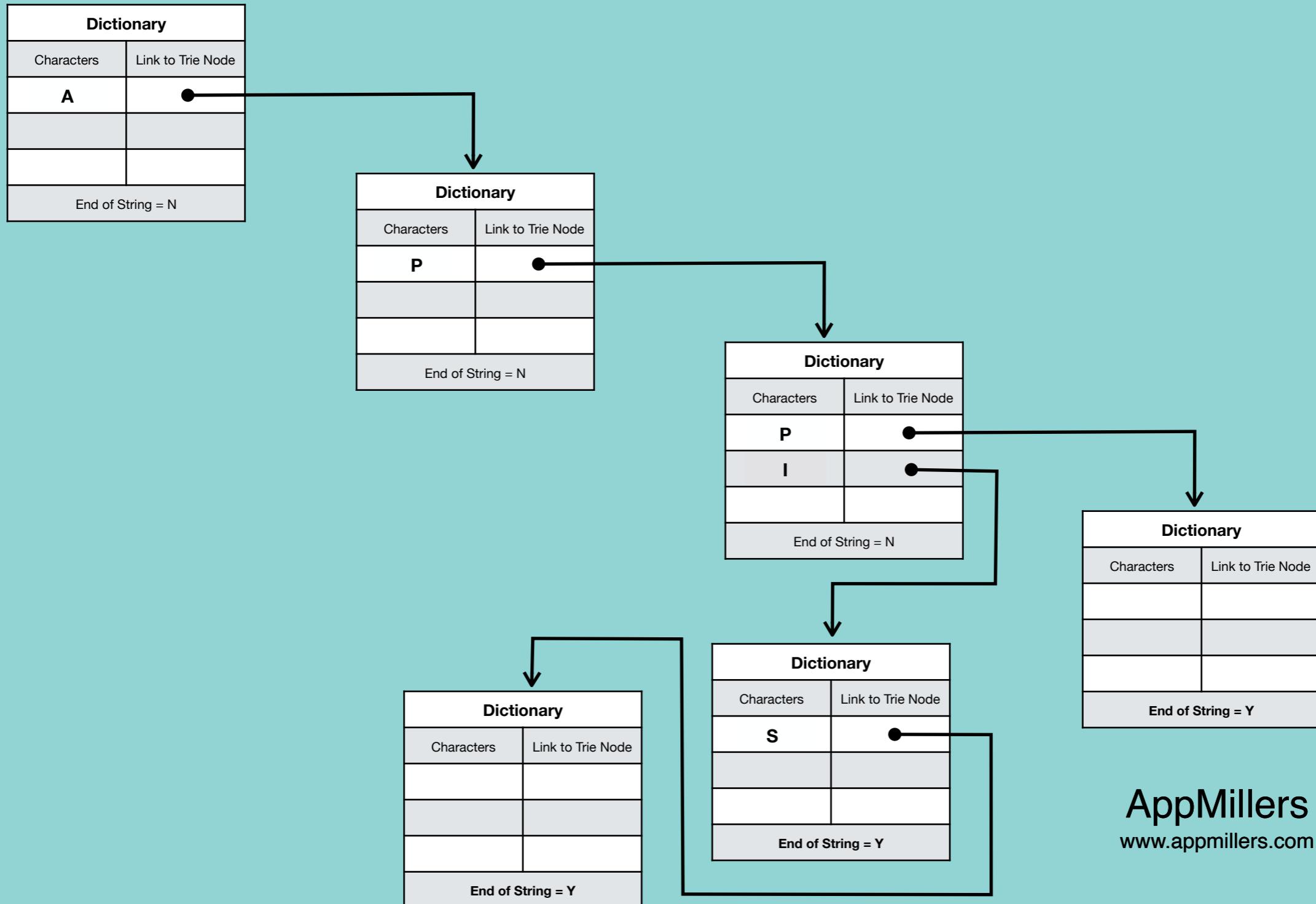
API



# Insert a string in a Trie

Case 3: New string's prefix is already present as complete string

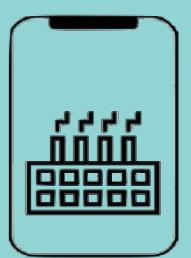
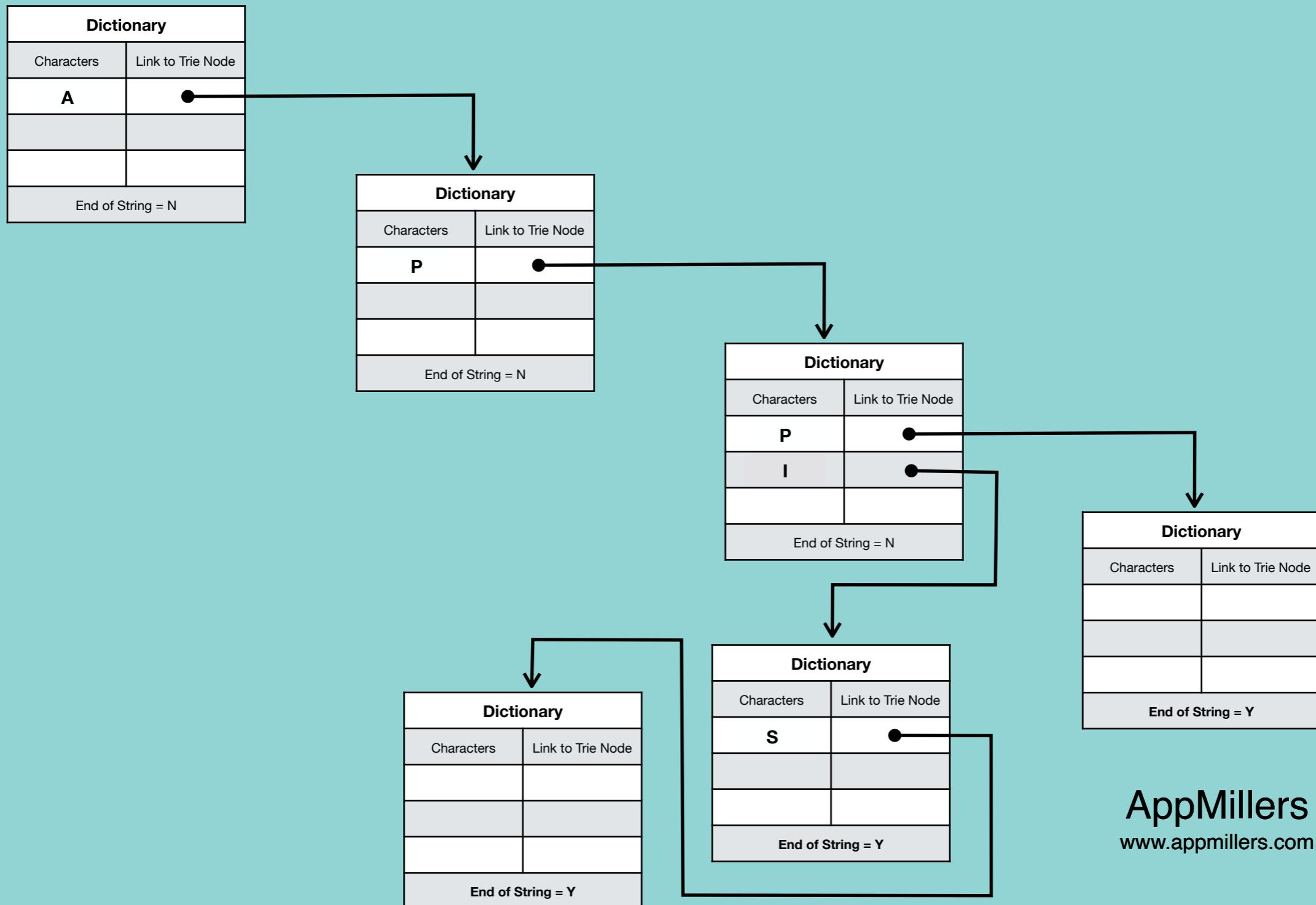
APIS



# Insert a string in a Trie

Case 4: String to be inserted is already presented in Trie

APIS

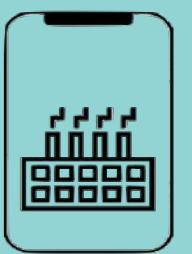
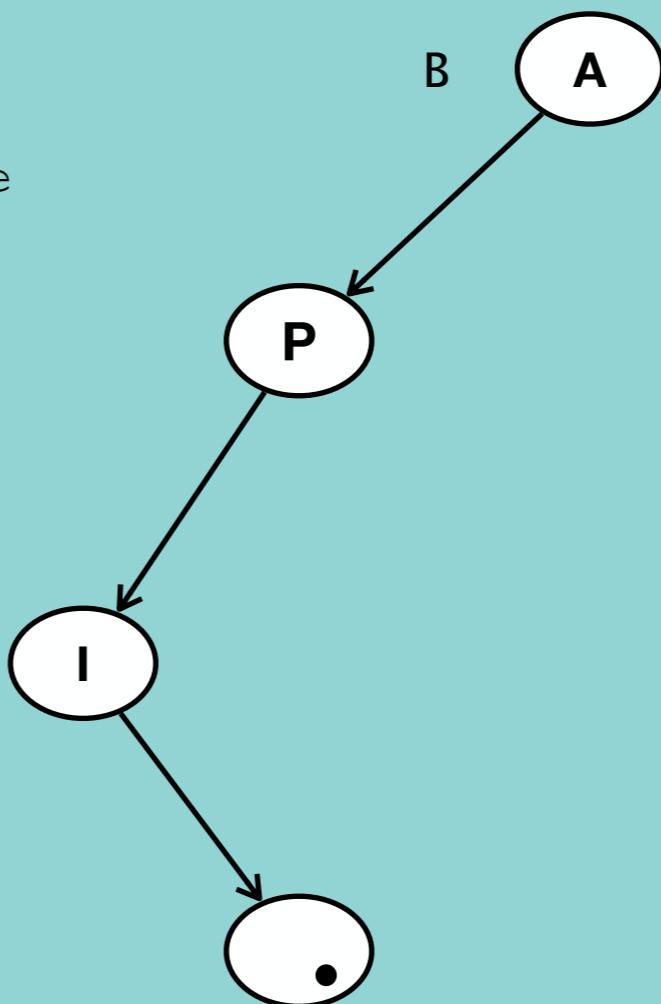


# Search for a string in a Trie

Case 1: String does not exist in Trie

BCD

**Return :** The string does not exist in Trie

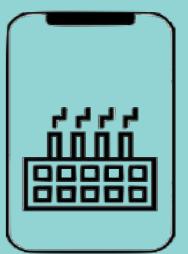
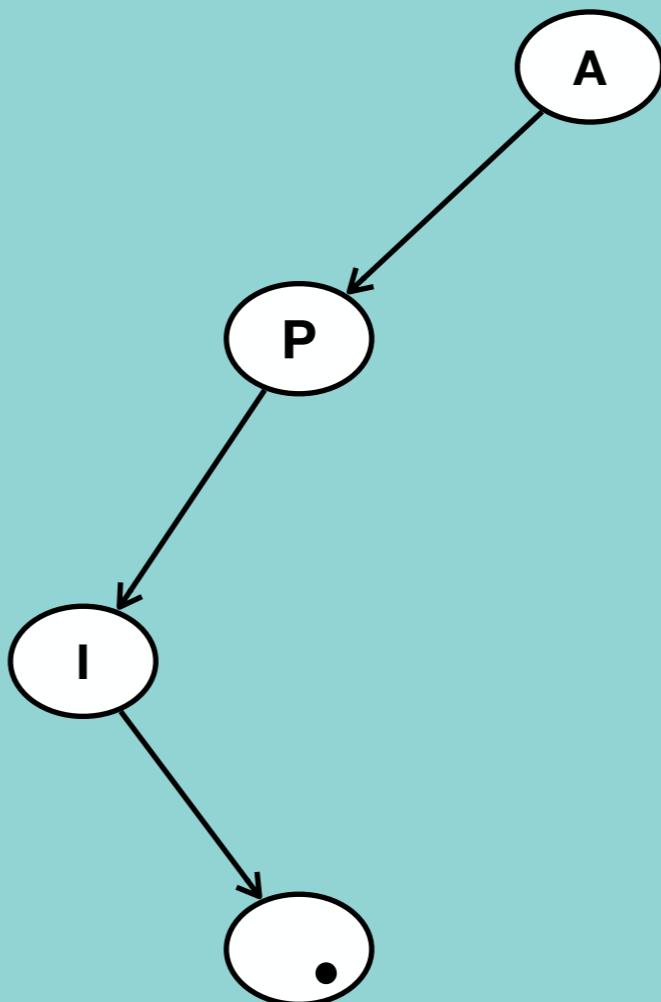


# Search for a string in a Trie

Case 2: String exists in Trie

API

Return : TRUE

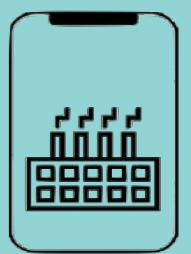
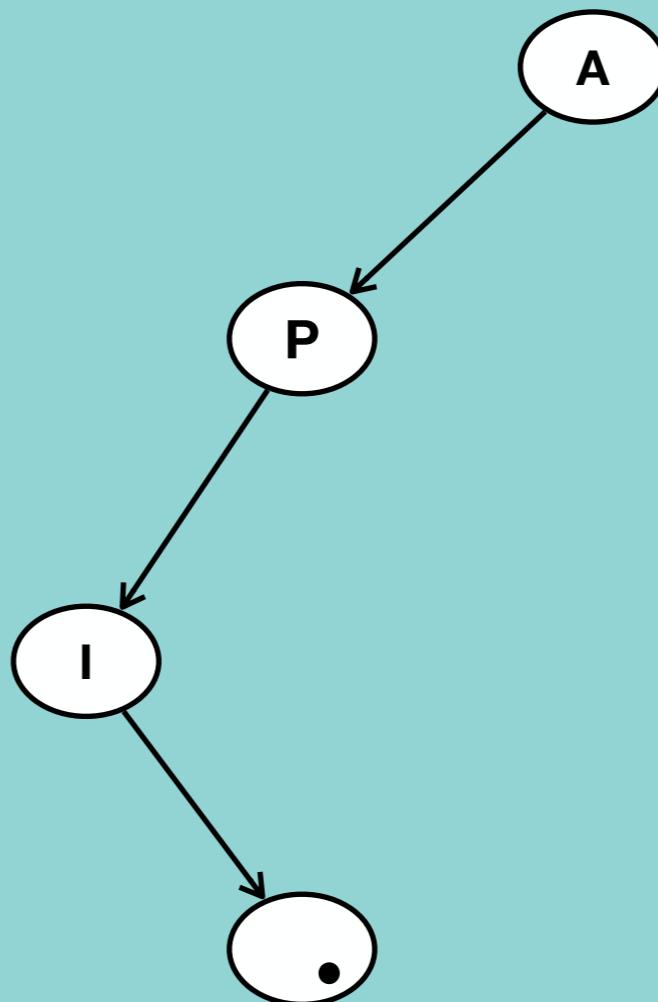


# Search for a string in a Trie

Case 3: String is a prefix of another string, but it does not exist in a Trie

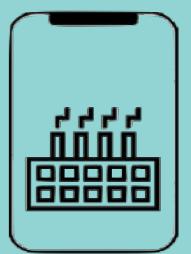
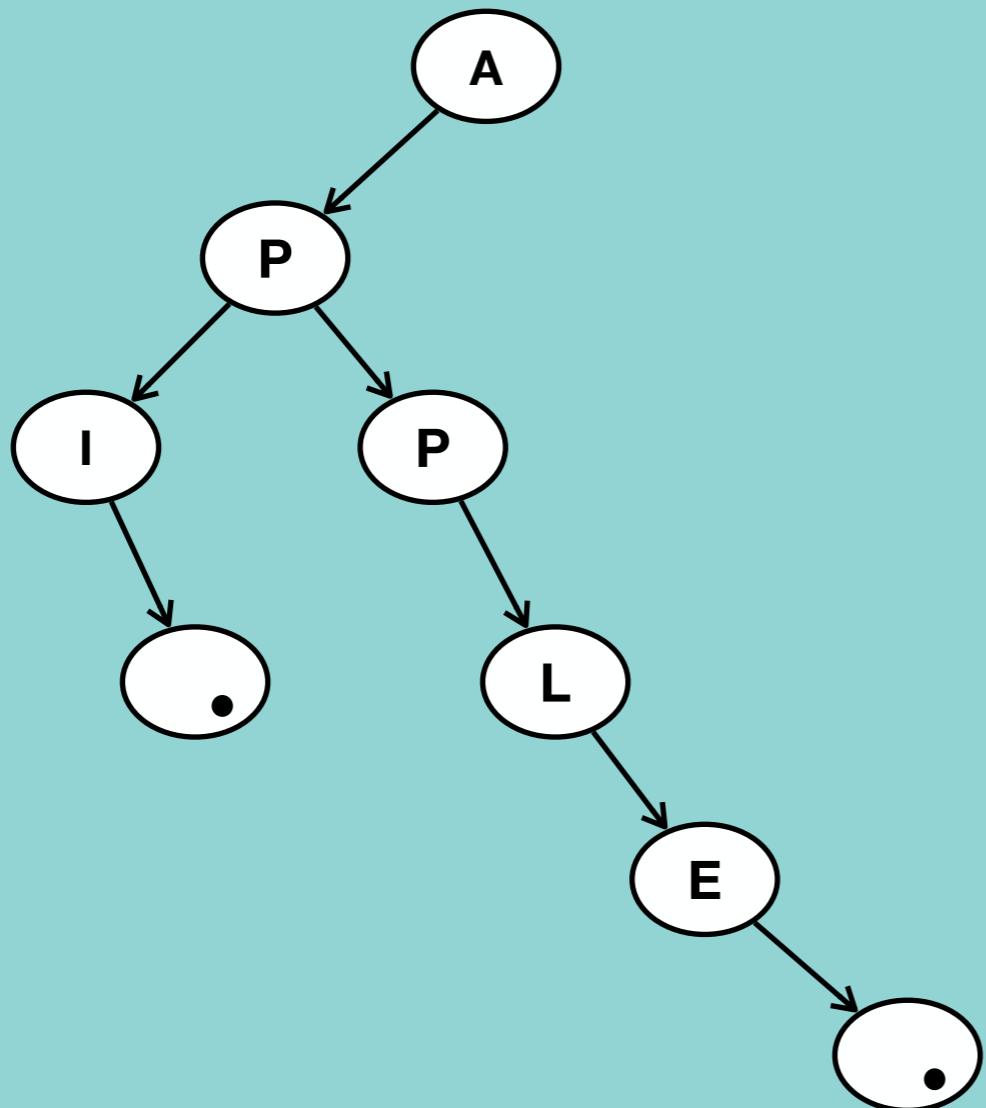
AP

Return : FALSE



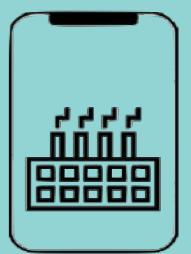
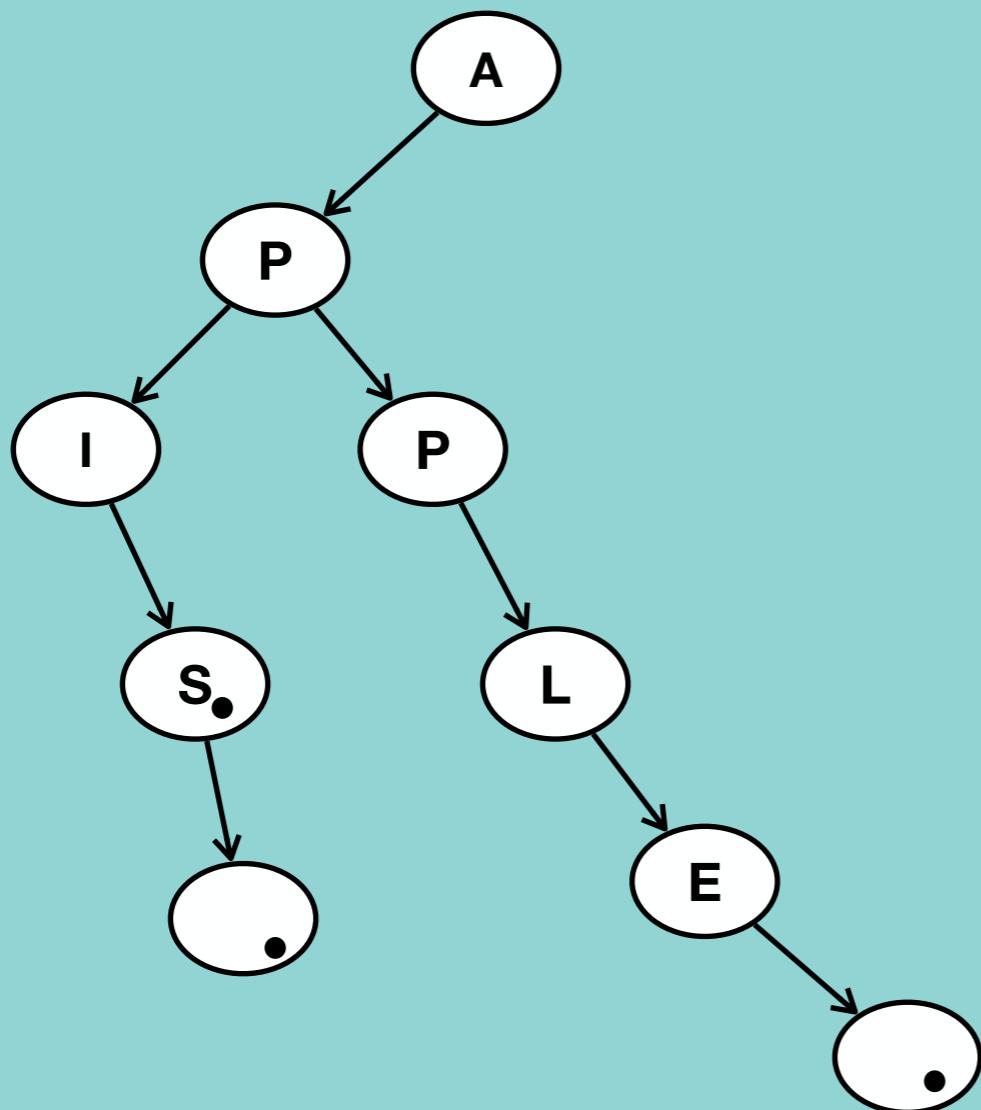
# Delete a string from Trie

Case 1: Some other prefix of string is same as the one that we want to delete. (API, APPLE)



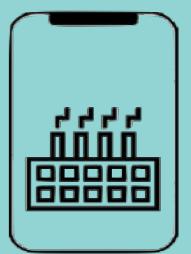
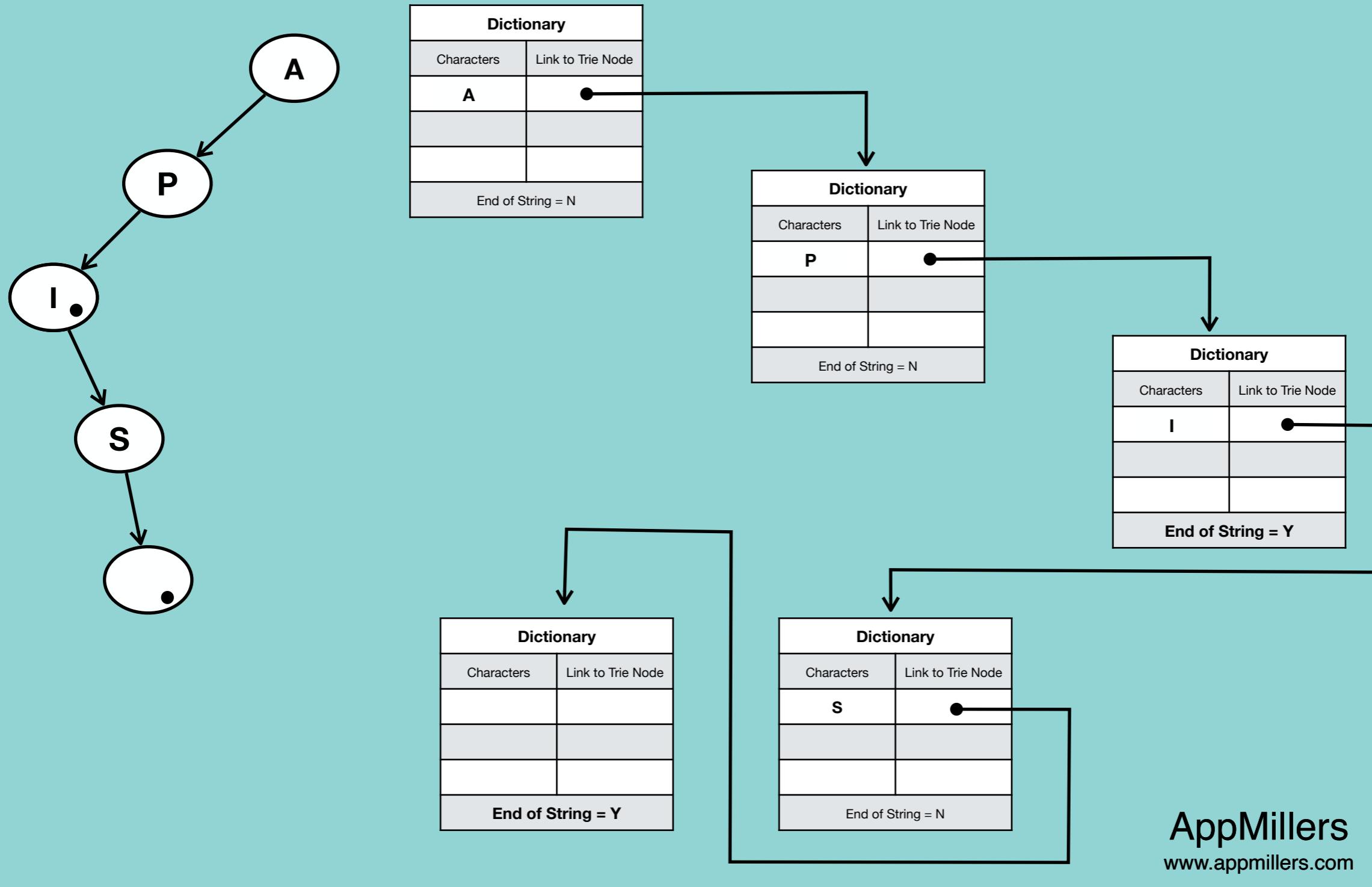
# Delete a string from Trie

Case 2: The string is a prefix of another string. (API, APIS)



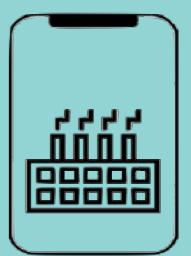
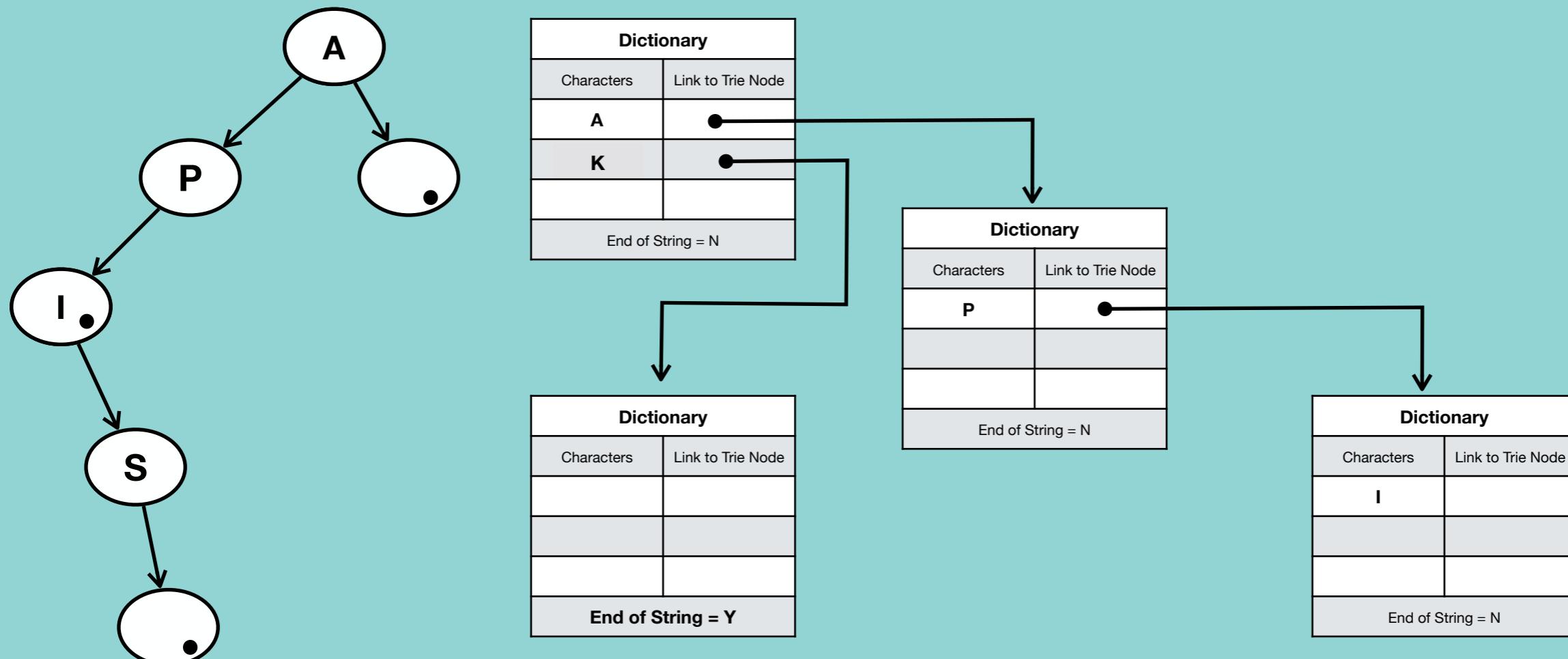
# Delete a string from Trie

Case 3: Other string is a prefix of this string. (APIS, AP)



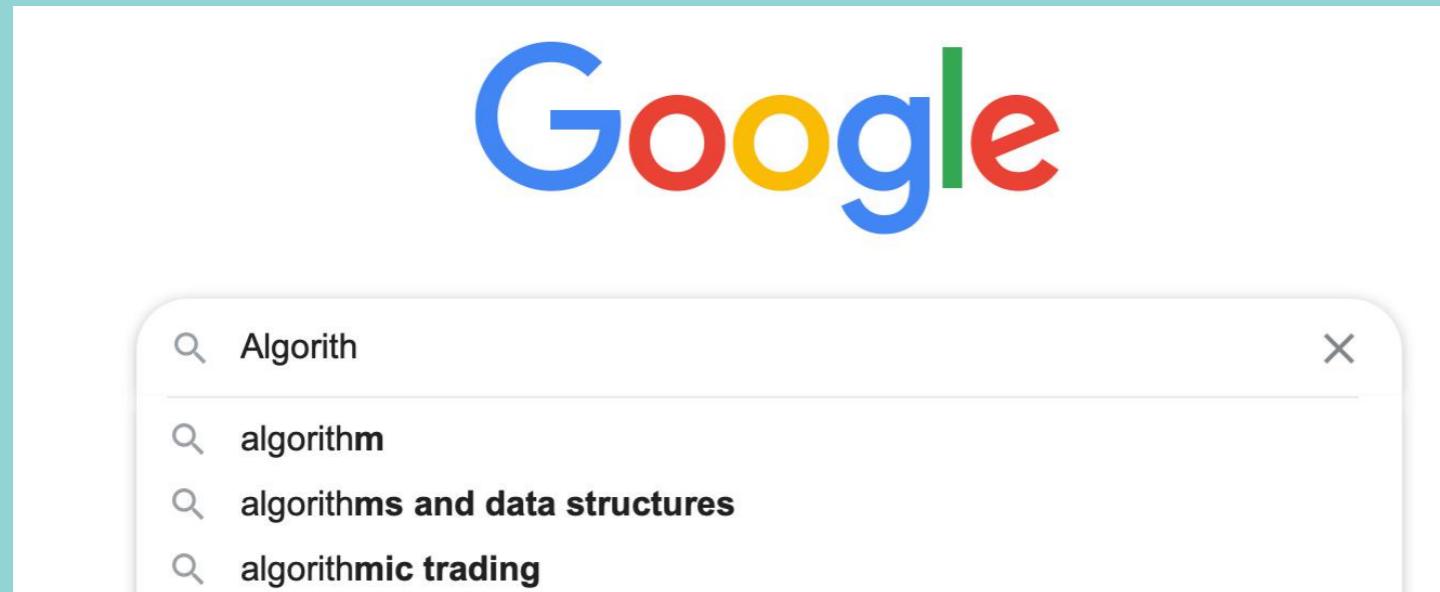
# Delete a string from Trie

Case 4: Not any node depends on this String (K)

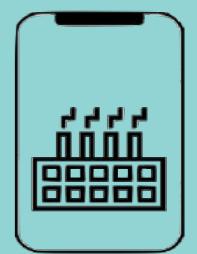


# Practical use of Trie

## – Auto completion

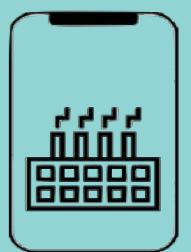
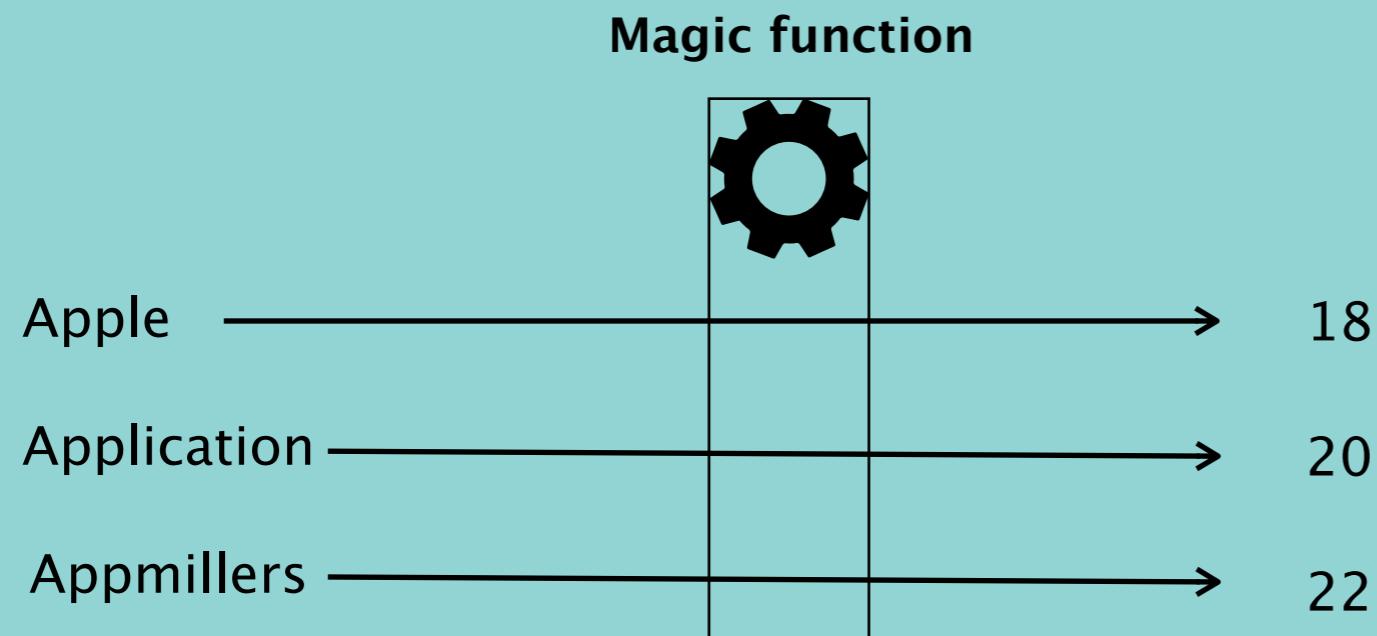


## – Spelling checker



# What is Hashing?

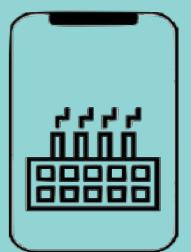
Hashing is a method of sorting and indexing data. The idea behind hashing is to allow large amounts of data to be indexed using keys commonly created by formulas



# Why Hashing?

It is time efficient in case of SEARCH Operation

Data Structure	Time complexity for SEARCH
Array/ Python List	$O(\log N)$
Linked List	$O(N)$
Tree	$O(\log N)$
Hashing	$O(1) / O(N)$



# Hashing Terminology

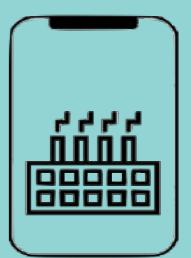
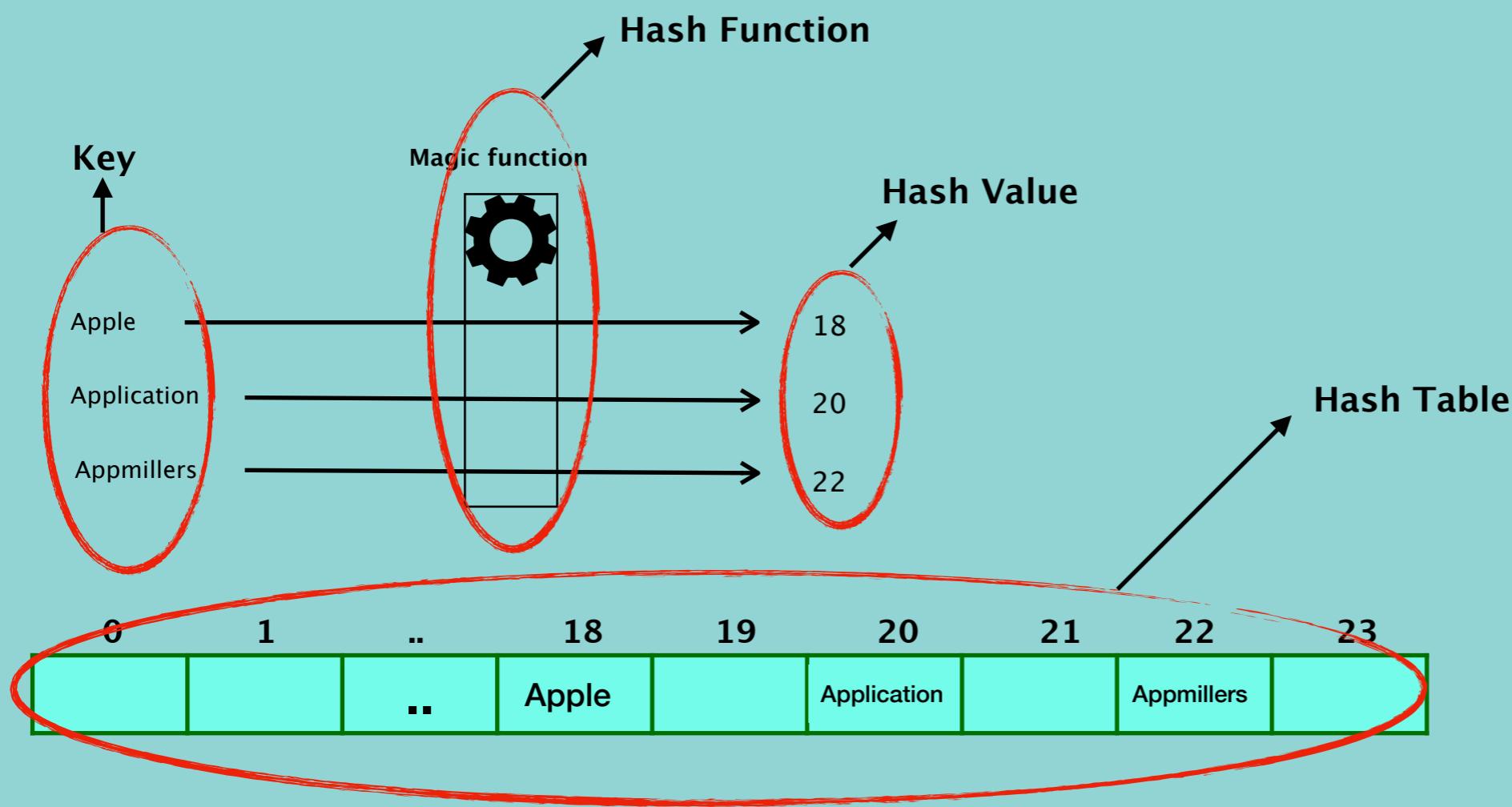
**Hash function** : It is a function that can be used to map of arbitrary size to data of fixed size.

**Key** : Input data by a user

**Hash value** : A value that is returned by Hash Function

**Hash Table** : It is a data structure which implements an associative array abstract data type, a structure that can map keys to values

**Collision** : A collision occurs when two different keys to a hash function produce the same output.



# Hashing Terminology

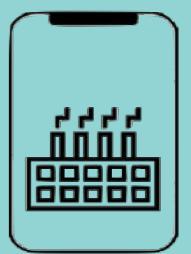
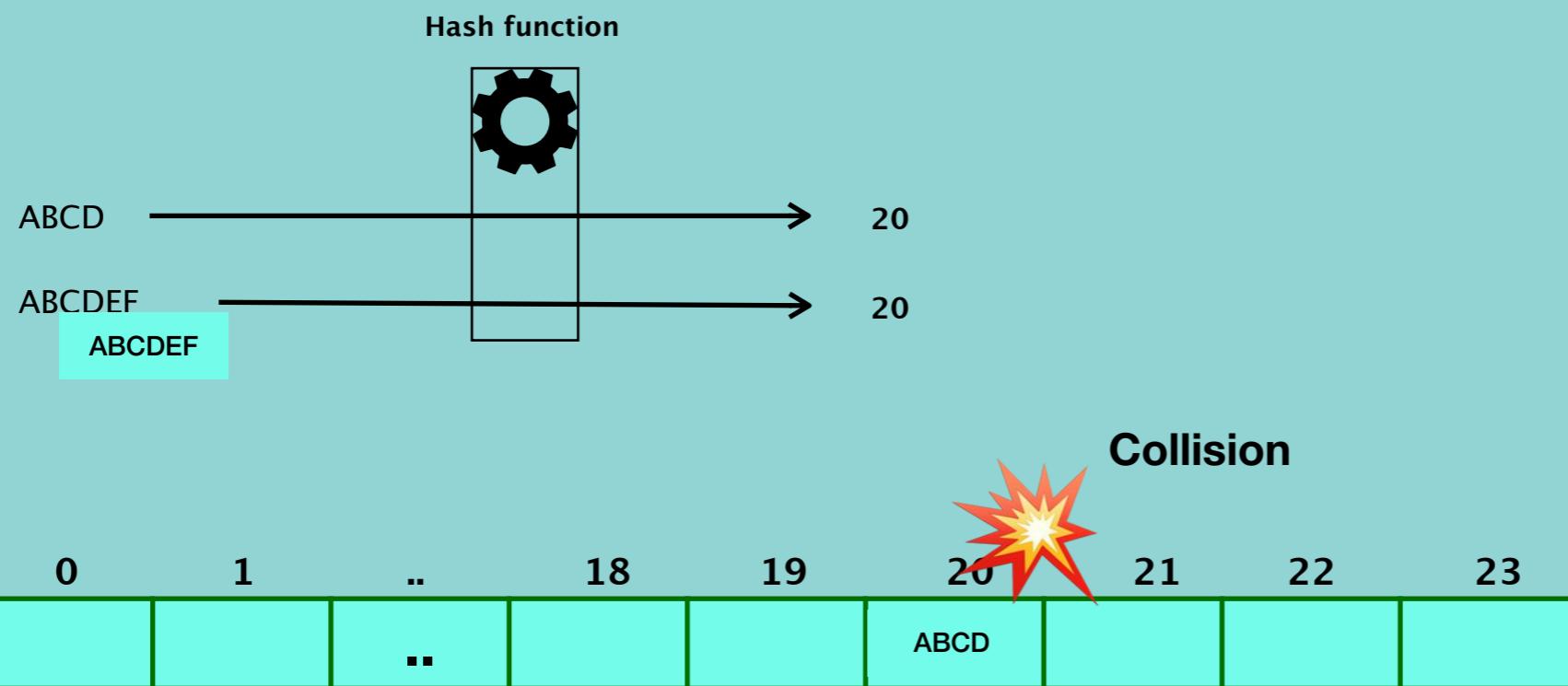
**Hash function** : It is a function that can be used to map of arbitrary size to data of fixed size.

**Key** : Input data by a user

**Hash value** : A value that is returned by Hash Function

**Hash Table** : It is a data structure which implements an associative array abstract data type, a structure that can map keys to values

**Collision** : A collision occurs when two different keys to a hash function produce the same output.



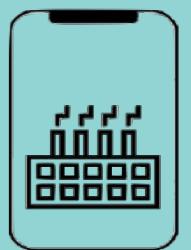
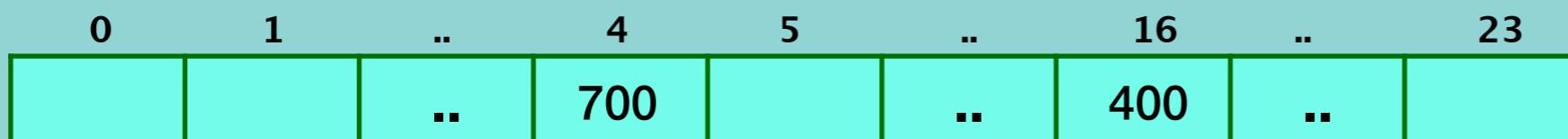
# Hash Functions

## Mod function

```
def mod(number, cellNumber):  
    return number % cellNumber
```

mod(400, 24) → 16

mod(700, 24) → 4



# Hash Functions

## ASCII function

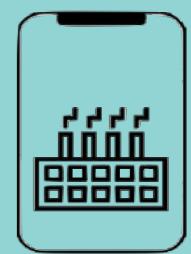
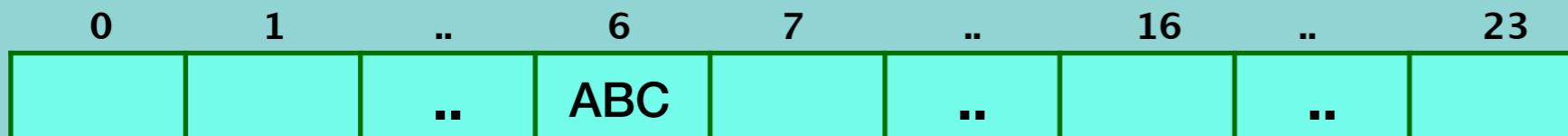
```
def modASCII(string, cellNumber):
    total = 0
    for i in string:
        total += ord(i)
    return total % cellNumber
```

modASCII("ABC", 24) → 6

A → 65       $65+66+67 = 198$        $\begin{array}{r} 24 \\ \hline 192 \\ \hline 6 \end{array}$   
B → 66  
C → 67

ASCII Table

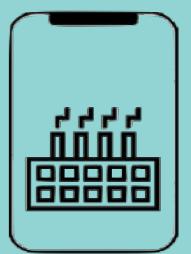
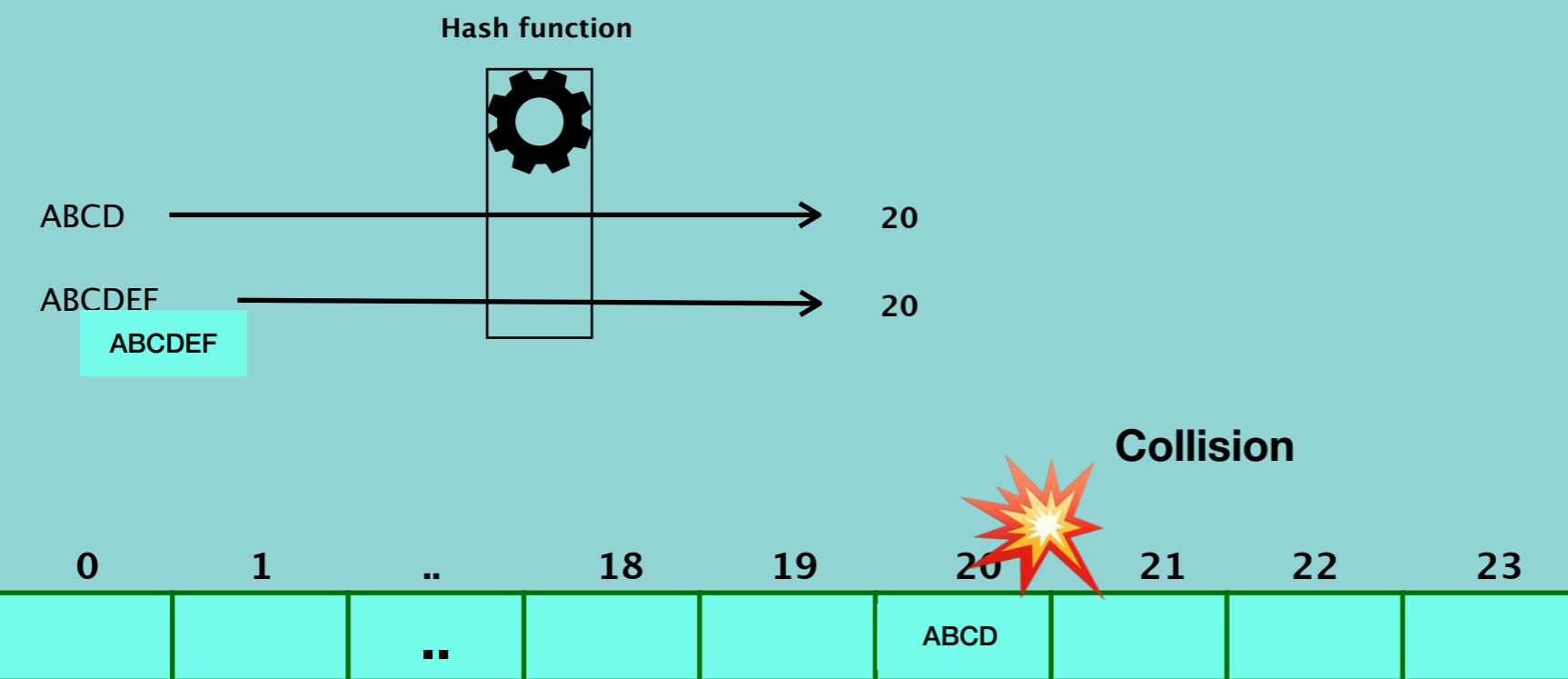
Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@
1	1	1	!	33	21	41	"	65	41	101	A
2	2	2	#	34	22	42	&	66	42	102	B
3	3	3	'	35	23	43	(	67	43	103	C
4	4	4	)	36	24	44	,	68	44	104	D
5	5	5	*	37	25	45	%	69	45	105	E
6	6	6	+	38	26	46	,	70	46	106	F
7	7	7	-	39	27	47	>	71	47	107	G
8	8	10	<	40	28	50	?	72	48	110	H
9	9	11		41	29	51		73	49	111	I
10	A	12		42	2A	52		74	4A	112	J
11	B	13		43	2B	53		75	4B	113	K
12	C	14		44	2C	54		76	4C	114	L
13	D	15		45	2D	55		77	4D	115	M
14	E	16		46	2E	56		78	4E	116	N
15	F	17		47	2F	57		79	4F	117	O
16		20		48	30	60	0	80	50	120	P
17		21		49	31	61	1	81	51	121	Q
18		22		50	32	62	2	82	52	122	R
19		23		51	33	63	3	83	53	123	S
20		24		52	34	64	4	84	54	124	T
21		25		53	35	65	5	85	55	125	U
22		26		54	36	66	6	86	56	126	V
23		27		55	37	67	7	87	57	127	W
24		30		56	38	70	8	88	58	130	X
25		31		57	39	71	9	89	59	131	Y
26		32		58	3A	72	:	90	5A	132	Z
27		33		59	3B	73	;	91	5B	133	[
28		34		60	3C	74	<	92	5C	134	\
29		35		61	3D	75	=	93	5D	135	]
30		36		62	3E	76	>	94	5E	136	^
31		37		63	3F	77	?	95	5F	137	_



# Hash Functions

## Properties of good Hash function

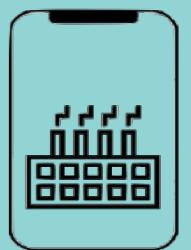
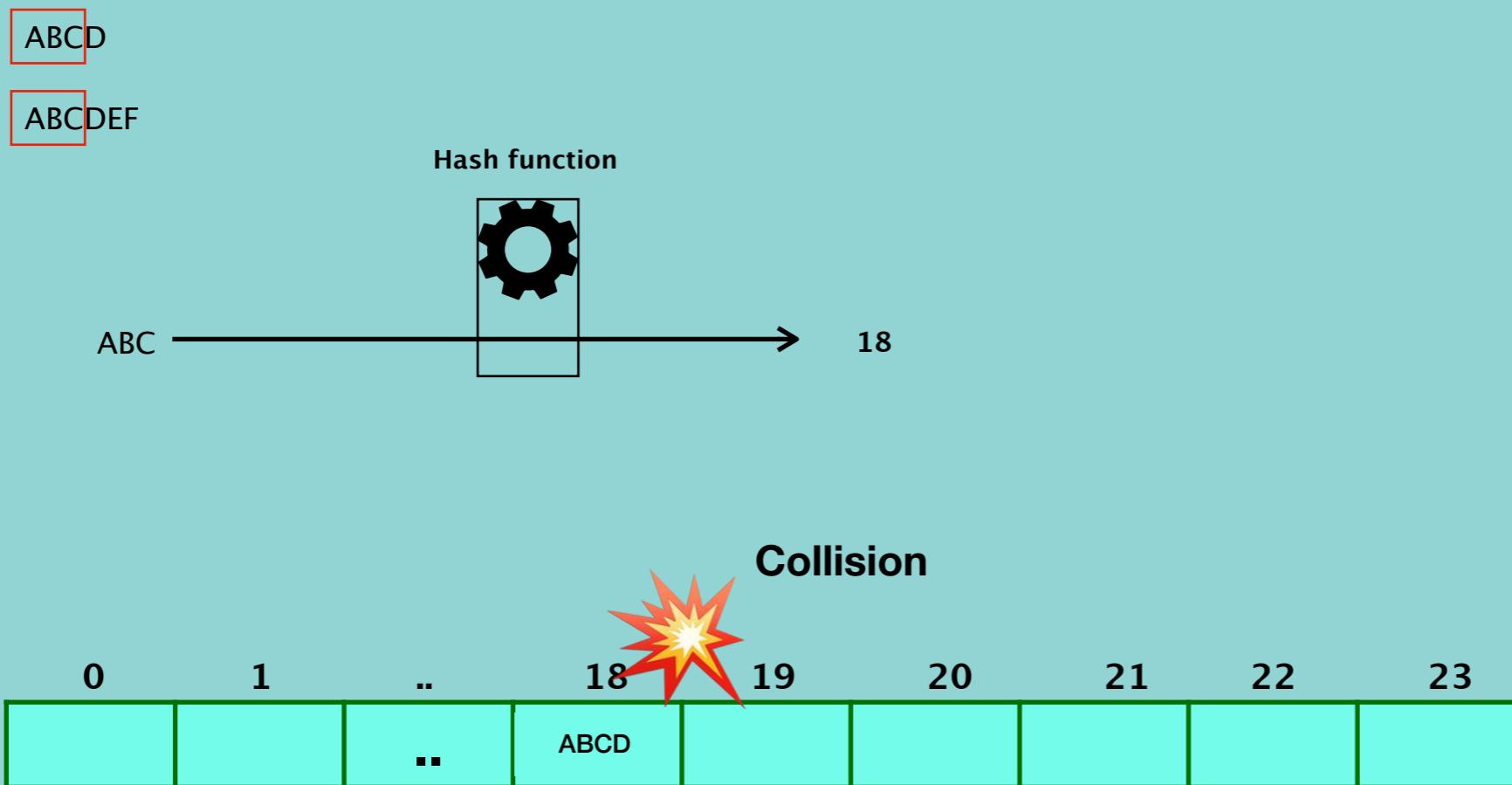
- It distributes hash values uniformly across hash tables



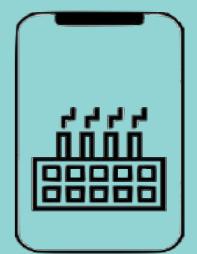
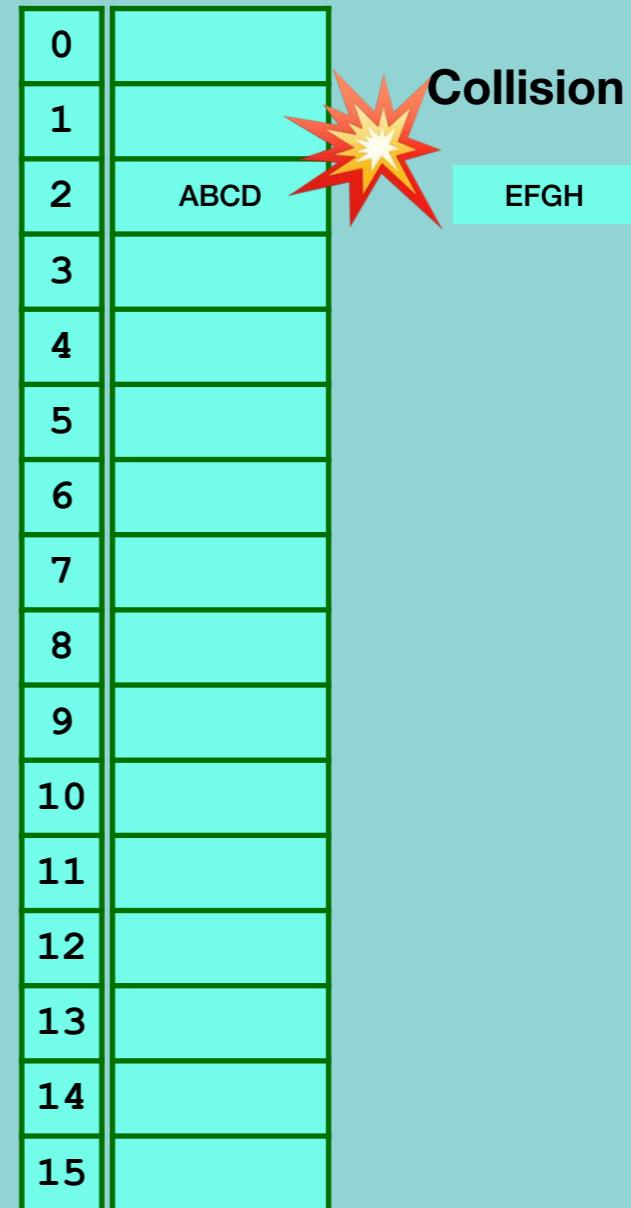
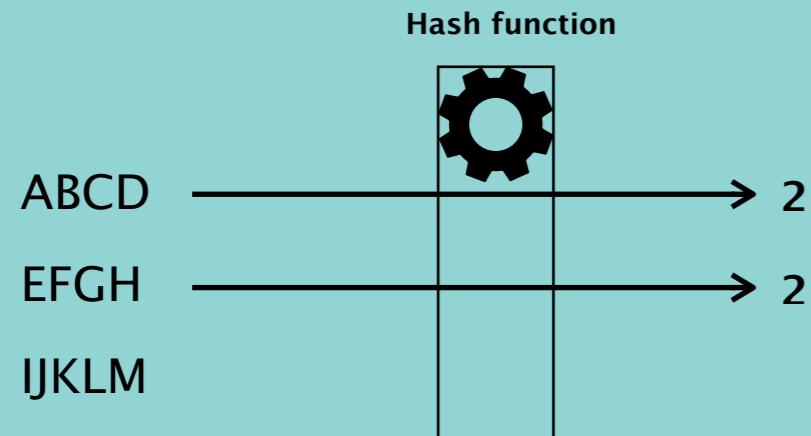
# Hash Functions

## Properties of good Hash function

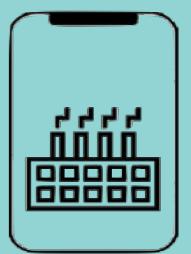
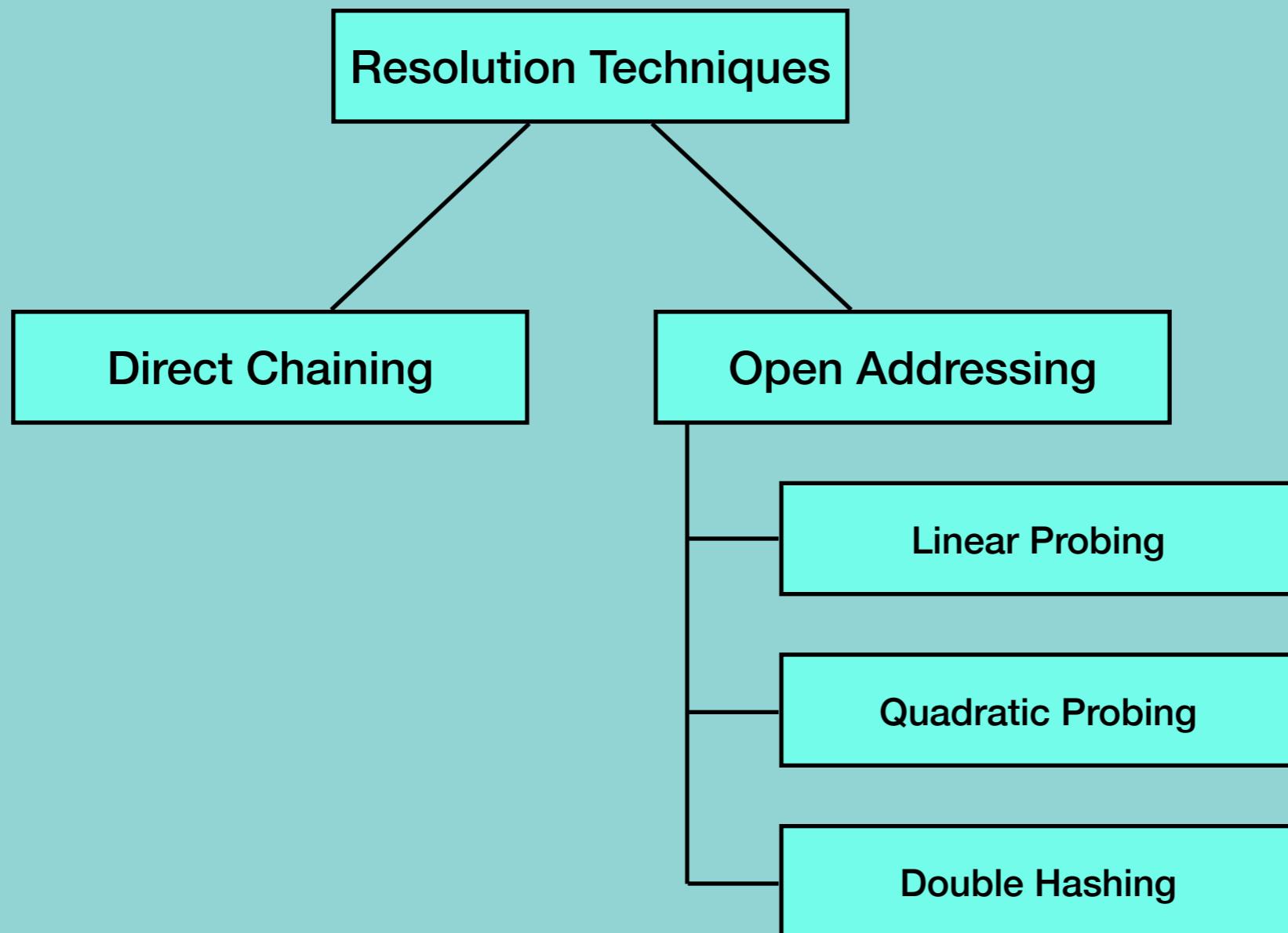
- It distributes hash values uniformly across hash tables
- It has to use all the input data



# Collision Resolution Techniques

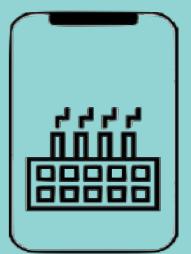
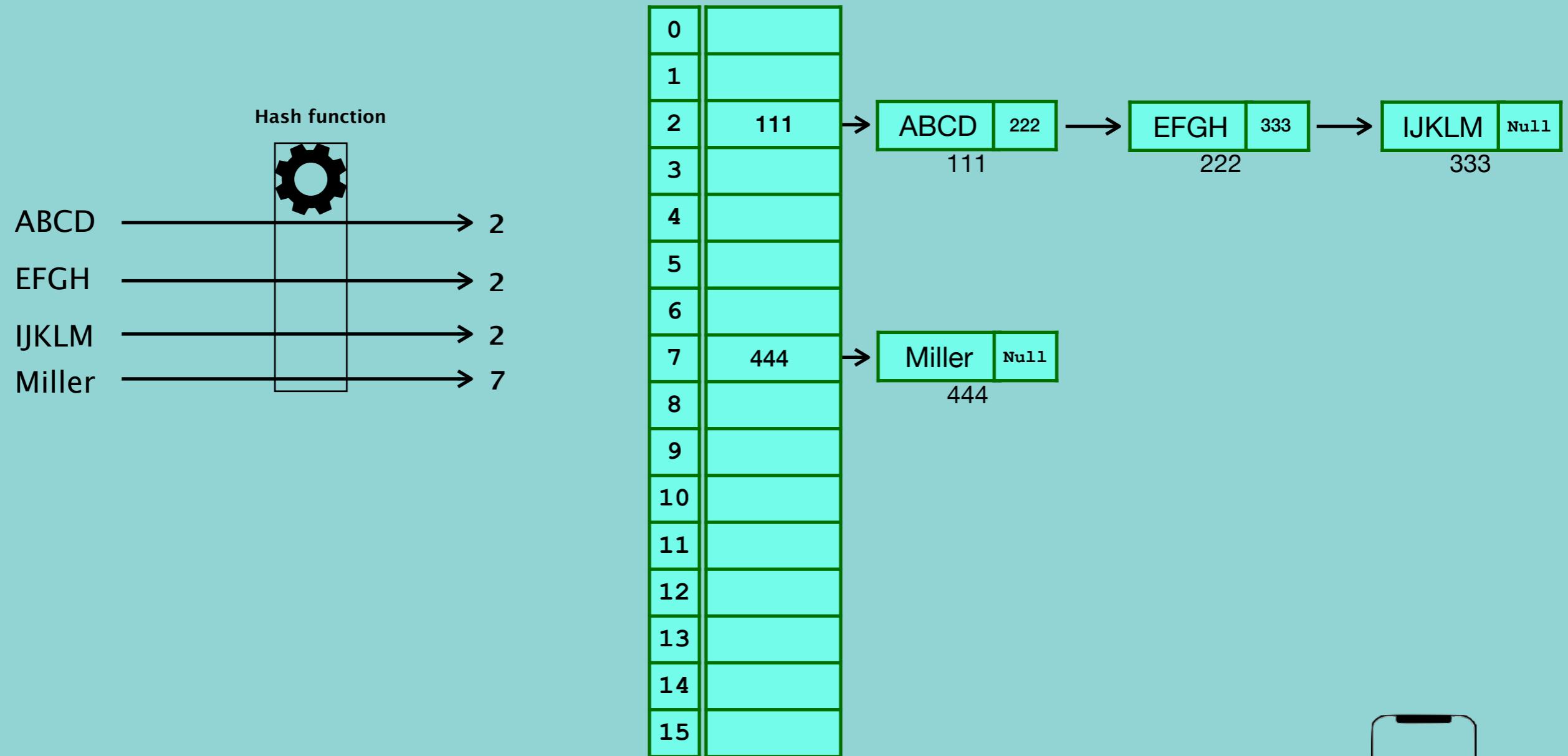


# Collision Resolution Techniques



# Collision Resolution Techniques

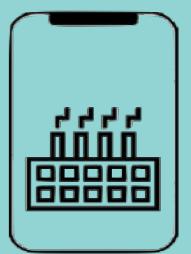
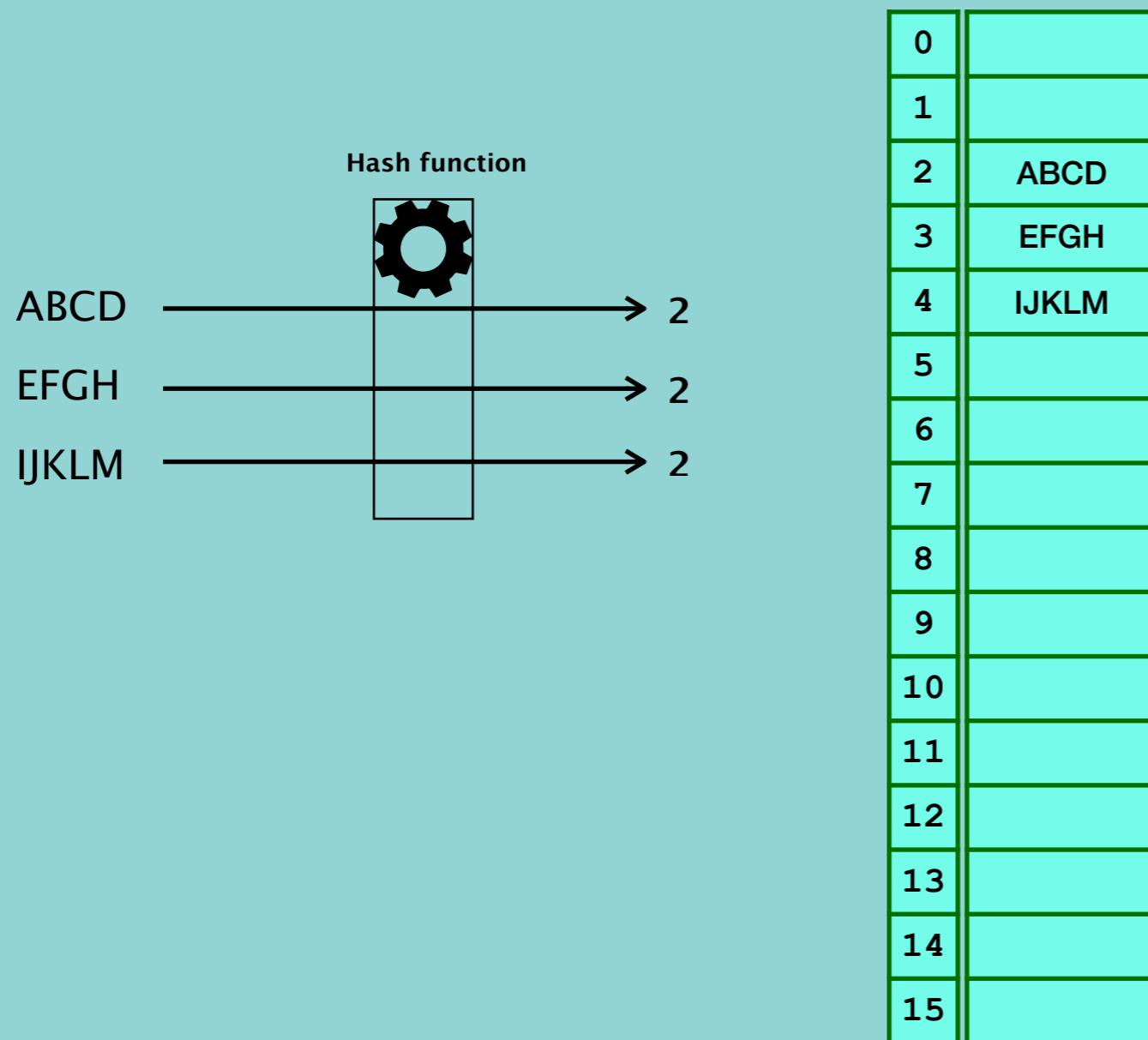
**Direct Chaining :** Implements the buckets as linked list. Colliding elements are stored in this lists



# Collision Resolution Techniques

**Open Addressing:** Colliding elements are stored in other vacant buckets. During storage and lookup these are found through so called probing.

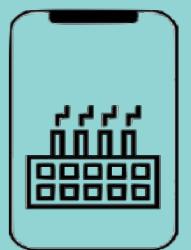
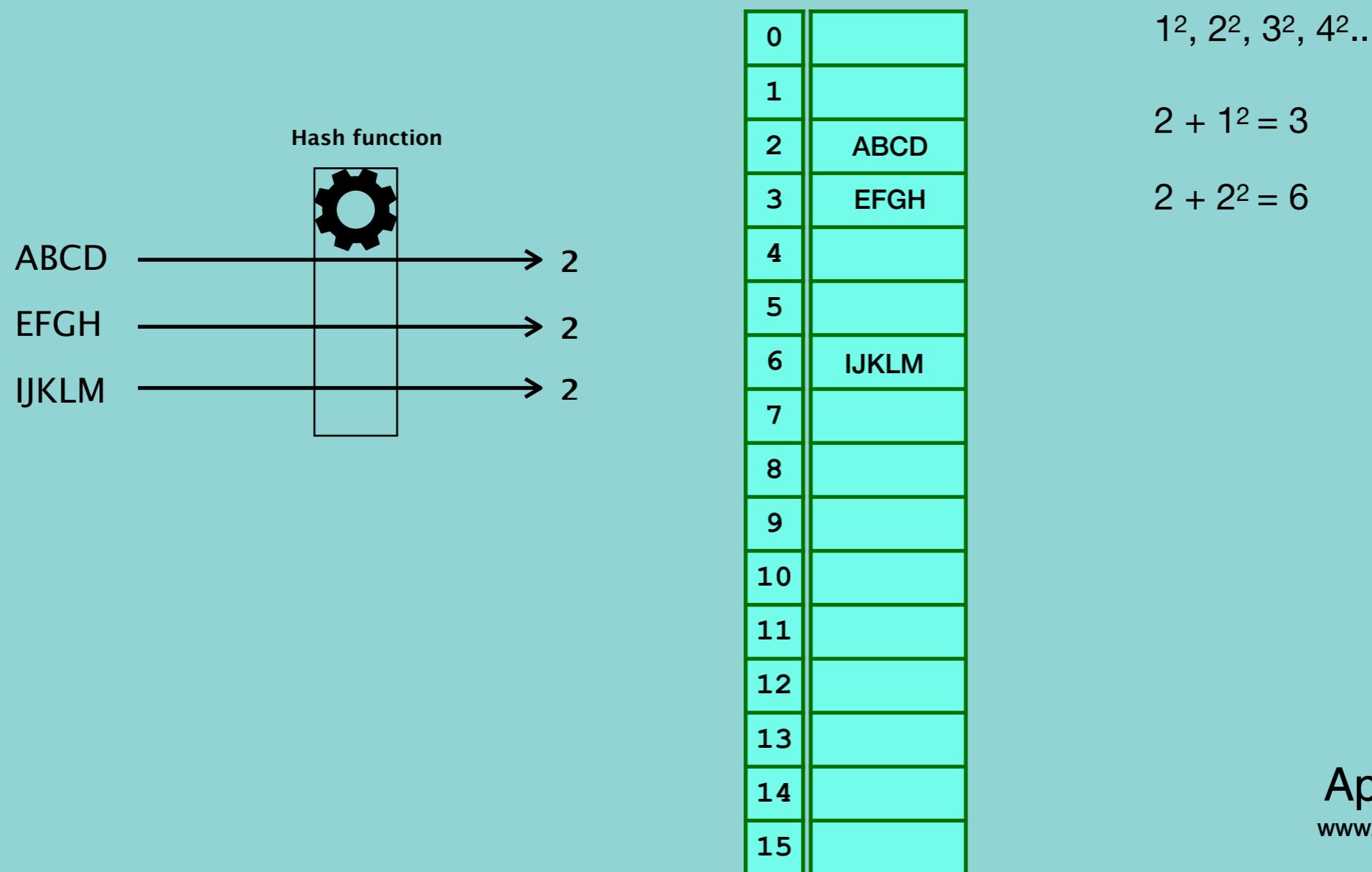
**Linear probing :** It places new key into closest following empty cell



# Collision Resolution Techniques

**Open Addressing:** Colliding elements are stored in other vacant buckets. During storage and lookup these are found through so called probing.

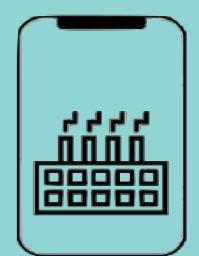
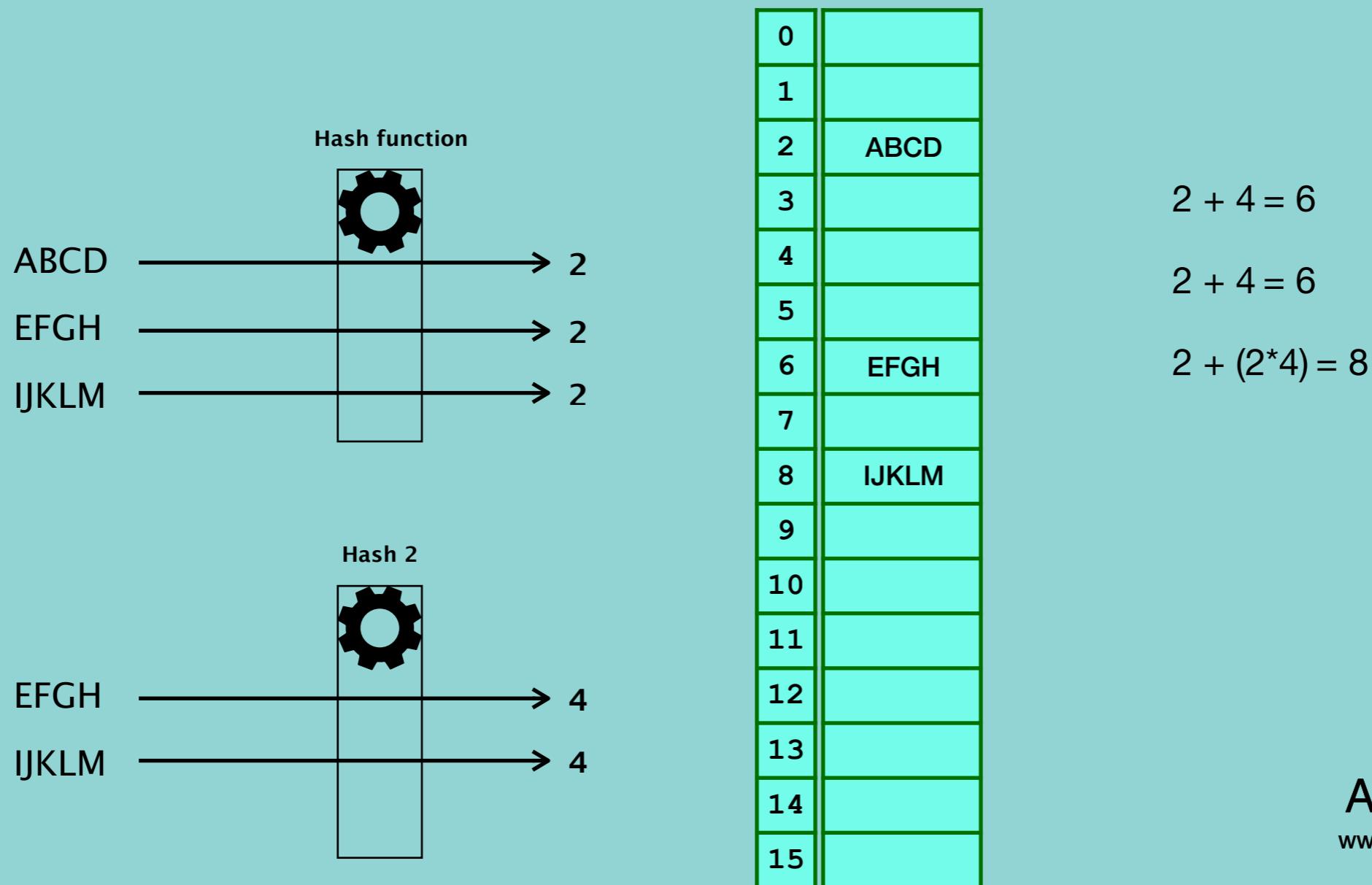
**Quadratic probing :** Adding arbitrary quadratic polynomial to the index until an empty cell is found



# Collision Resolution Techniques

**Open Addressing:** Colliding elements are stored in other vacant buckets. During storage and lookup these are found through so called probing.

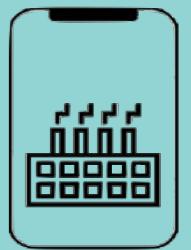
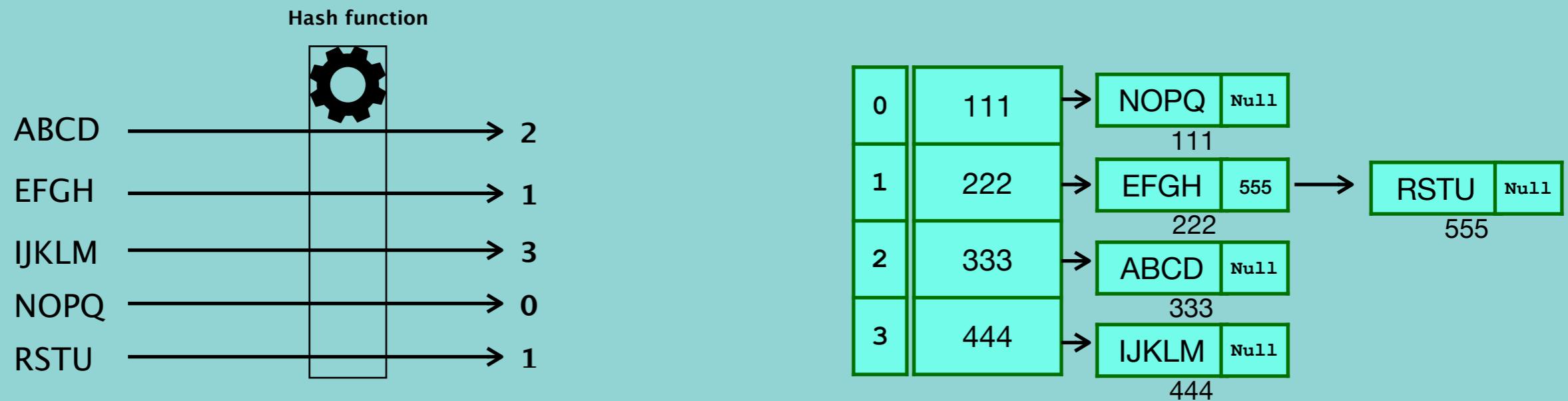
**Double Hashing :** Interval between probes is computed by another hash function



# Hash Table is Full

## Direct Chaining

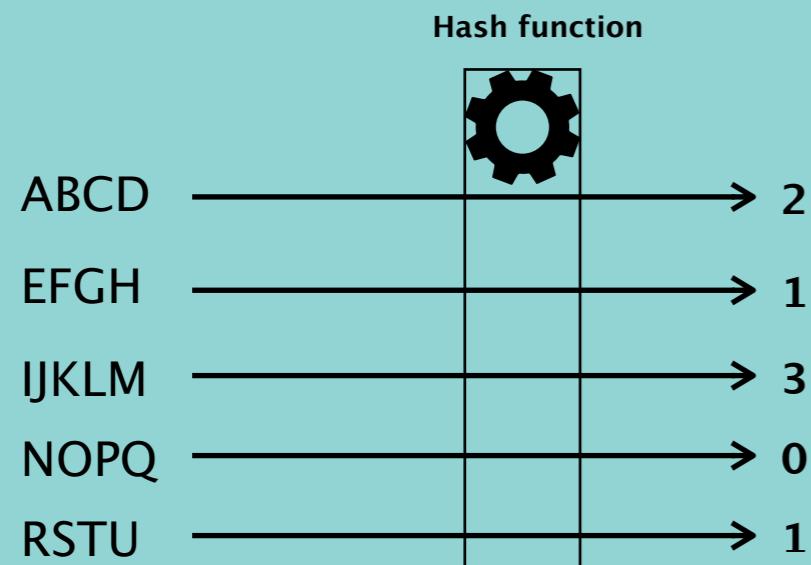
This situation will never arise.



# Hash Table is Full

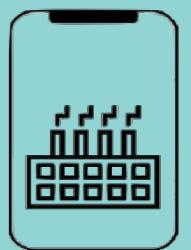
## Open addressing

Create 2X size of current Hash Table and recall hashing for current keys



0	NOPQ
1	EFGH
2	ABCD
3	IJKL

0	NOPQ
1	EFGH
2	ABCD
3	IJKL
4	RSTU
5	
6	
7	



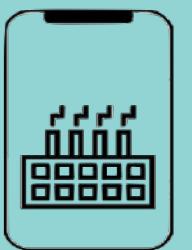
# Pros and Cons of Collision resolution techniques

## Direct chaining

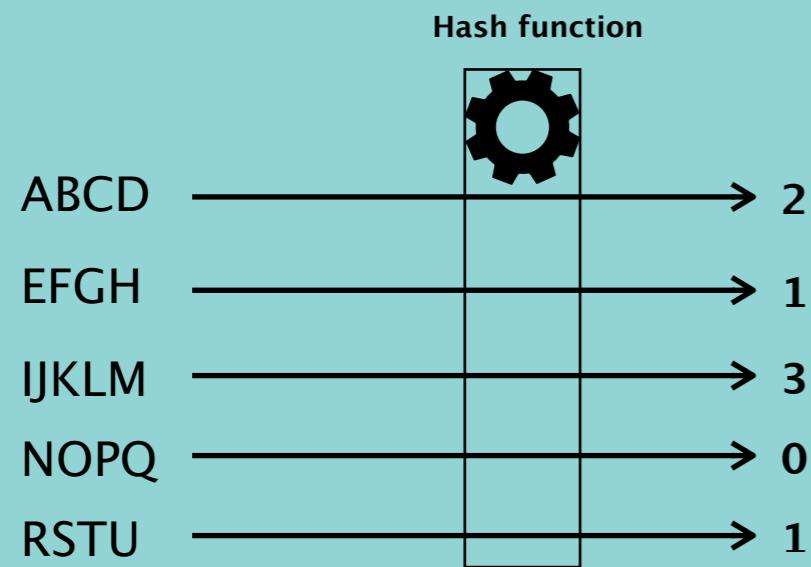
- Hash table never gets full
- Huge Linked List causes performance leaks (Time complexity for search operation becomes  $O(n)$ .)

## Open addressing

- Easy Implementation
  - When Hash Table is full, creation of new Hash table affects performance (Time complexity for search operation becomes  $O(n)$ .)
- 
- ▶ If the input size is known we always use “Open addressing”
  - ▶ If we perform deletion operation frequently we use “Direct Chaining”

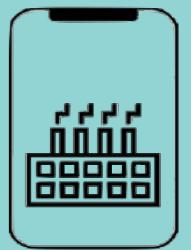


# Pros and Cons of Collision resolution techniques



0	NOPQ
1	EFGH
2	
3	IJKLM
4	RSTU

Linear Probing



# Practical Use of Hashing

## Password verification

Login Required

User:

Password:

Personal Computer

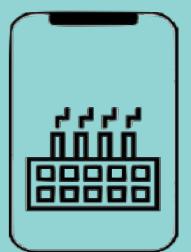


Login : [elshad@google.com](mailto:elshad@google.com)  
Password: 123456

Google Servers

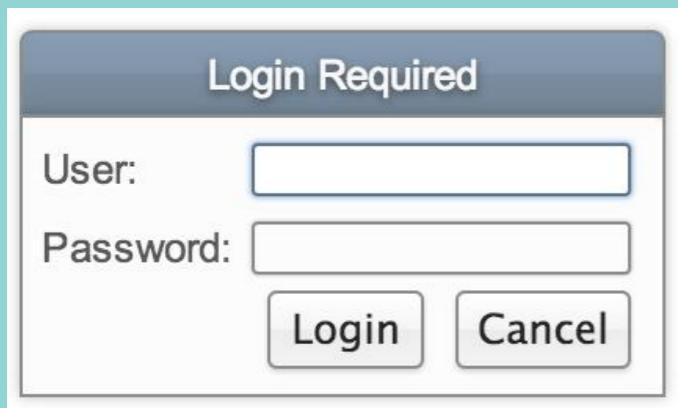


Hash value: \*&71283\*a12

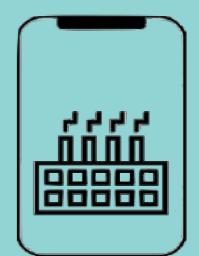
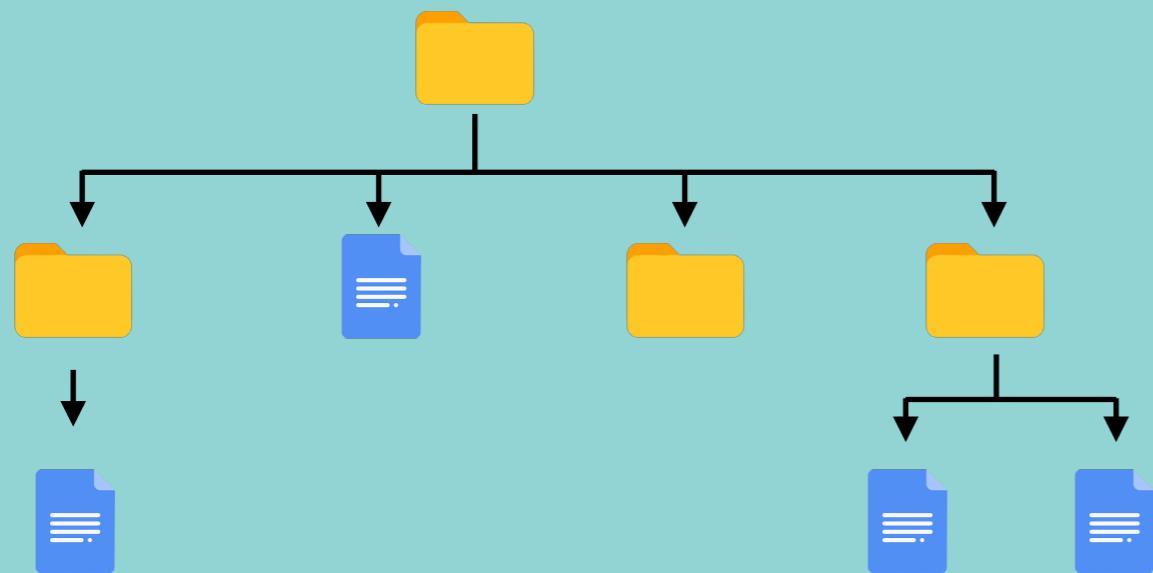


# Practical Use of Hashing

## Password verification

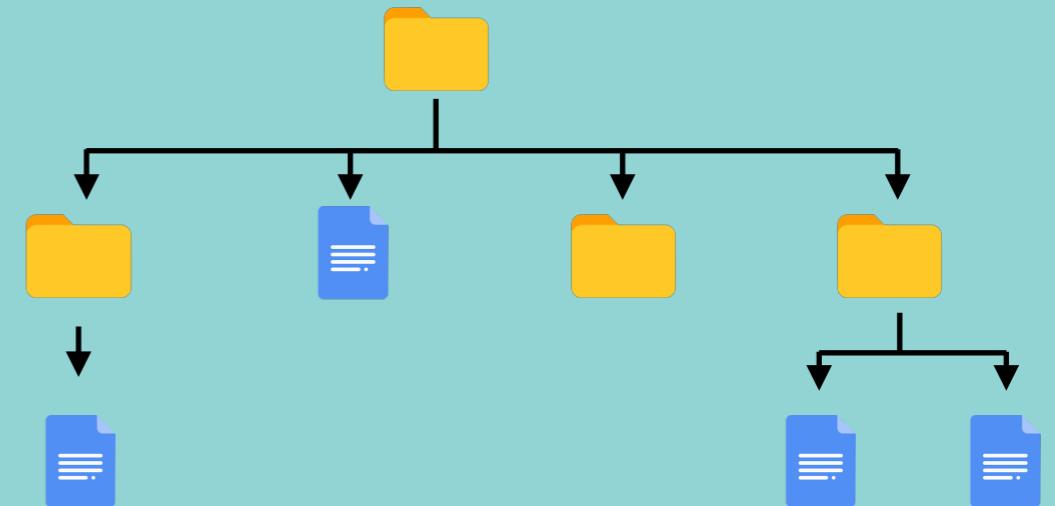


**File system :** File path is mapped to physical location on disk



# Practical Use of Hashing

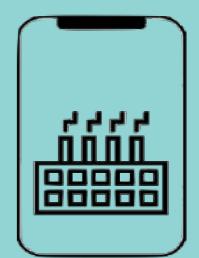
**File system :** File path is mapped to physical location on disk



Path: /Documents/Files/hashing.txt

0	
1	/Documents/Files/hashing.txt
2	
3	

Physical location: sector 4

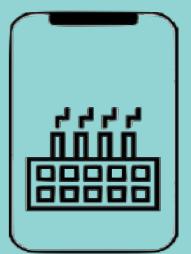


# Pros and Cons of Hashing

✓ On an average Insertion/Deletion/Search operations take  $O(1)$  time.

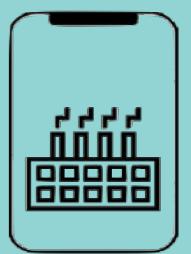
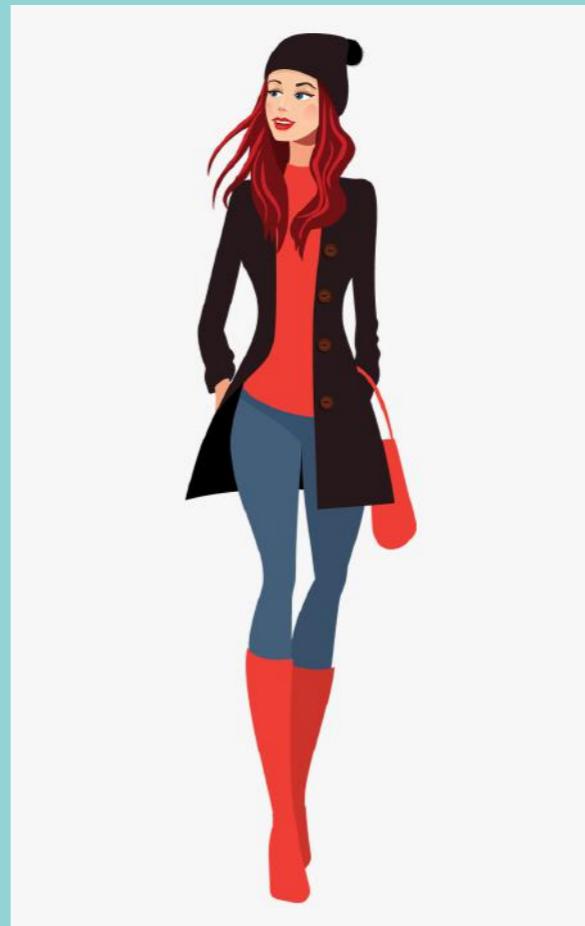
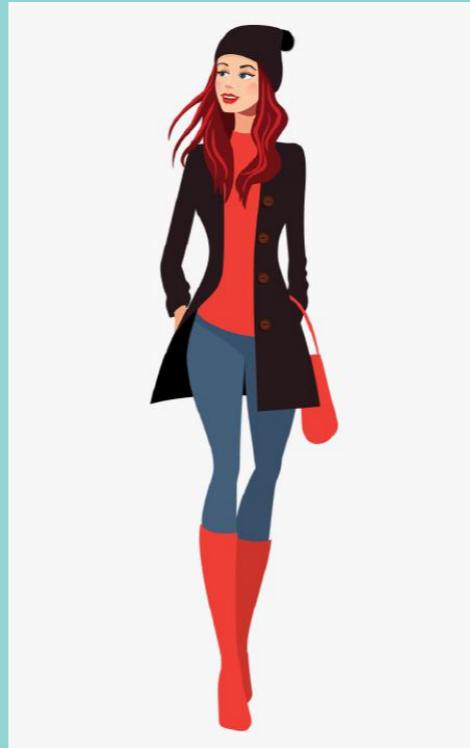
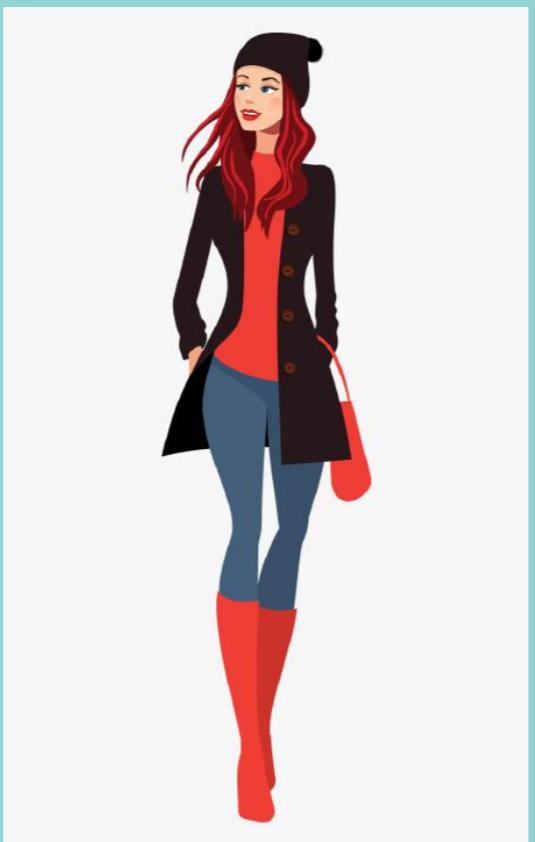
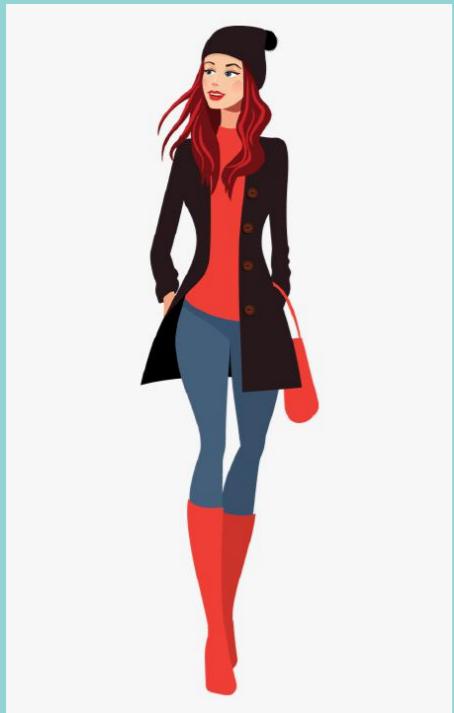
✗ When Hash function is not good enough Insertion/Deletion/Search operations take  $O(n)$  time

Operations	Array /Python List	Linked List	Tree	Hashing
Insertion	$O(N)$	$O(N)$	$O(\text{Log}N)$	$O(1)/O(N)$
Deletion	$O(N)$	$O(N)$	$O(\text{Log}N)$	$O(1)/O(N)$
Search	$O(N)$	$O(N)$	$O(\text{Log}N)$	$O(1)/O(N)$



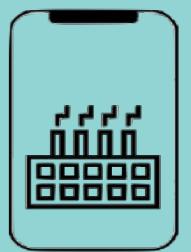
# What is Sorting?

By definition sorting refers to arranging data in a particular format : either ascending or descending.



# What is Sorting?

By definition sorting refers to arranging data in a particular format : either ascending or descending.



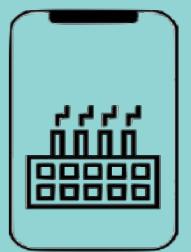
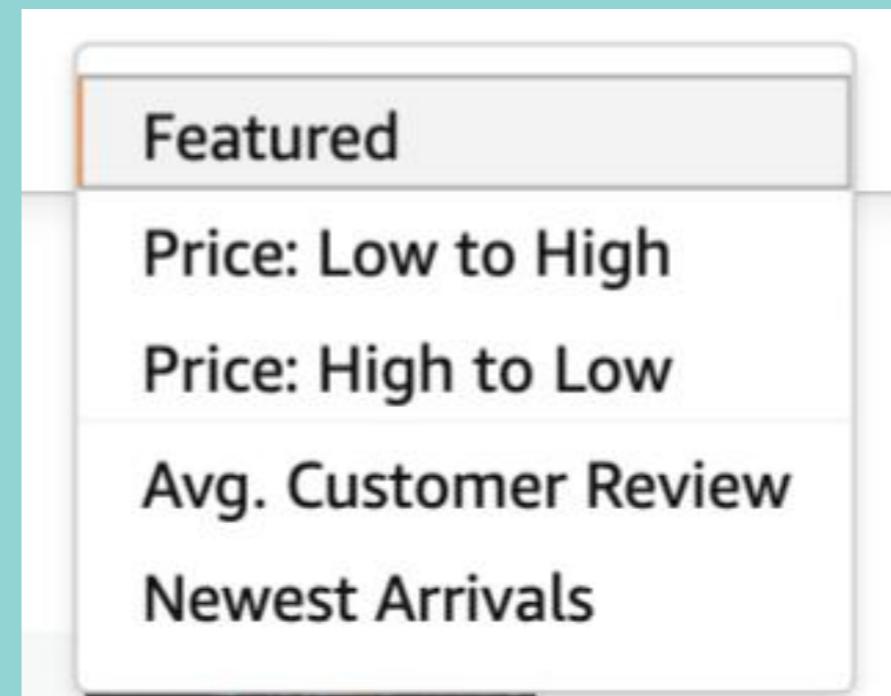
# What is Sorting?

## Practical Use of Sorting

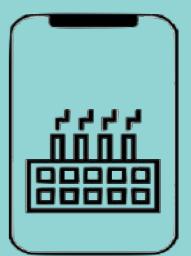
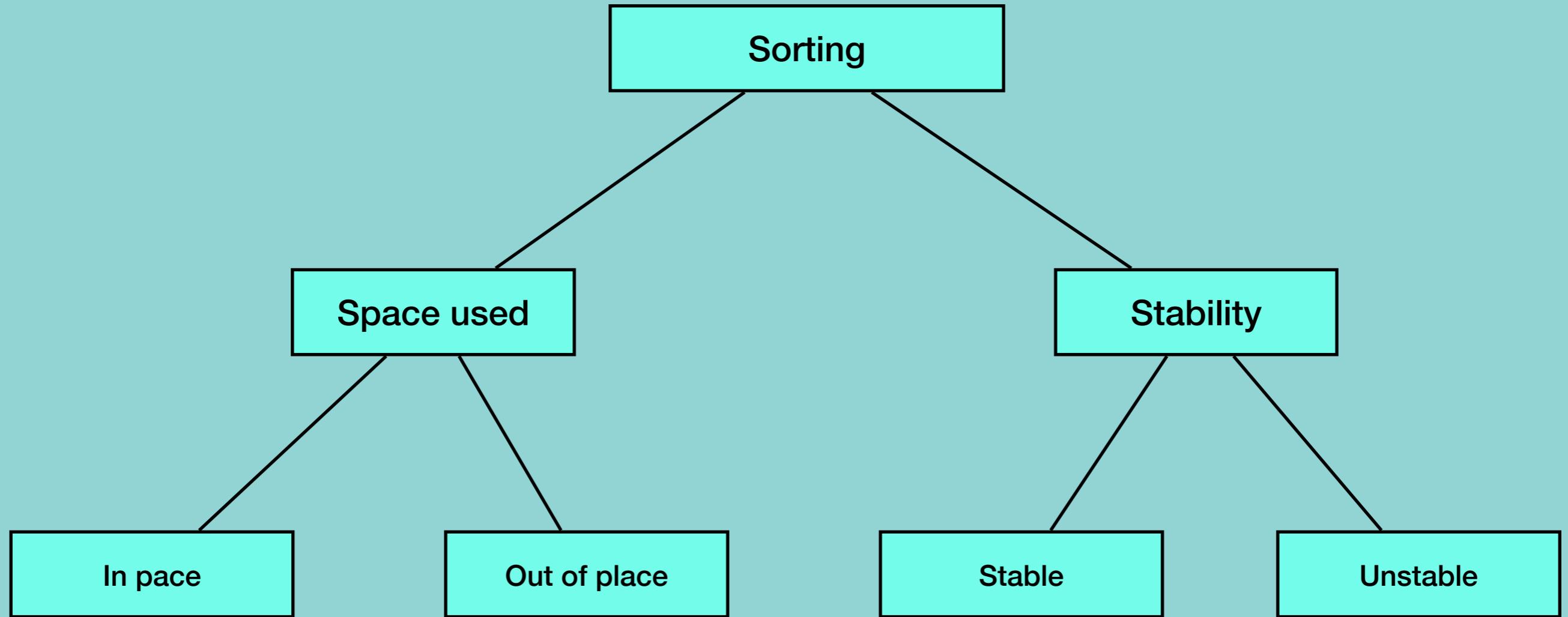
Microsoft Excel : Built in functionality to sort data

Online Stores: Online shopping websites generally have option for sorting by price, review, ratings..

The screenshot shows a Microsoft Excel spreadsheet with data in columns A through I. The 'DATA' tab is selected in the ribbon. A yellow callout box points to the 'Sort' button in the 'Sort & Filter' group. A red box highlights the 'Sort' button in the ribbon. A 'Sort' dialog box is open, showing three levels of sorting: 'Sort by Reporting Manager', 'Then by Employee Name', and 'Then by Order Total'. The 'Order' dropdown for the first level is set to 'A to Z'.



# Types of Sorting?



## Space used

**In place sorting :** Sorting algorithms which does not require any extra space for sorting

**Example :** Bubble Sort

70	10	80	30	20	40	60	50	90
----	----	----	----	----	----	----	----	----

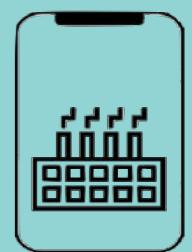
10	20	30	40	50	60	70	80	90
----	----	----	----	----	----	----	----	----

**Out place sorting :** Sorting algorithms which requires an extra space for sorting

**Example :** Merge Sort

70	10	80	30	20	40	60	50	90
----	----	----	----	----	----	----	----	----

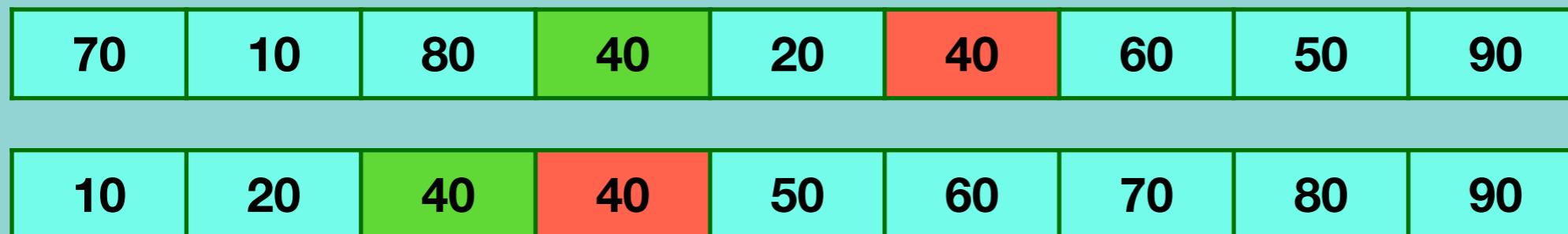
10	20	30	40	50	60	70	80	90
----	----	----	----	----	----	----	----	----



# Stability

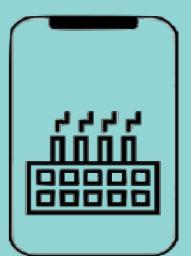
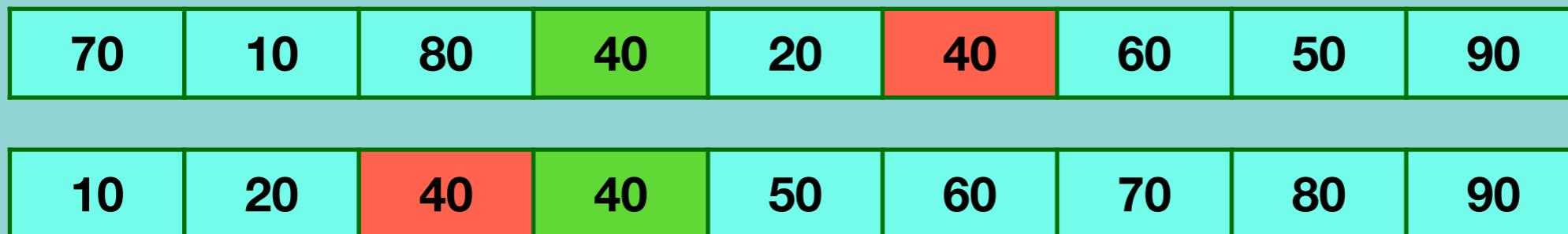
**Stable sorting :** If a sorting algorithm after sorting the contents does not change the sequence of similar content in which they appear, then this sorting is called stable sorting.

**Example :** Insertion Sort



**UnStable sorting :** If a sorting algorithm after sorting the content changes the sequence of similar content in which they appear, then it is called unstable sort.

**Example :** Quick Sort



# Stability

## UnStable sorting example

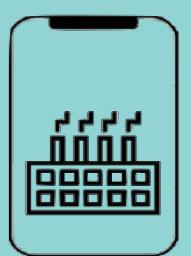
Unsorted data	
Name	Age
Renad	7
Nick	6
Richard	6
Parker	7
Sofia	7

Sorted by name	
Name	Age
Nick	6
Parker	7
Renad	7
Richard	6
Sofia	7

Sorted by age (stable)	
Name	Age
Nick	6
Richard	6
Parker	7
Renad	7
Sofia	7

{

Sorted by age (unstable)	
Name	Age
Nick	6
Richard	6
Renad	7
Parker	7
Sofia	7



# Sorting Terminology

## Increasing Order

- If successive element is greater than the previous one
- Example : 1, 3, 5, 7, 9 ,11

## Decreasing Order

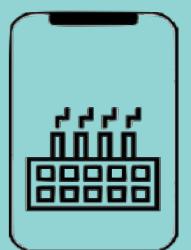
- If successive element is less than the previous one
- Example : 11, 9, 7, 5, 3, 1

## Non Increasing Order

- If successive element is less than or equal to its previous element in the sequence.
- Example : 11, 9, 7, 5, 5, 3, 1

## Non Decreasing Order

- If successive element is greater than or equal to its previous element in the sequence
- Example : 1, 3, 5, 7, 7, 9, 11



# Sorting Algorithms

**Bubble sort**

**Selection sort**

**Insertion sort**

**Bucket sort**

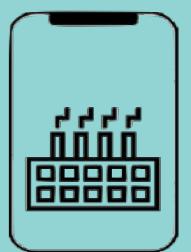
**Merge sort**

**Quick sort**

**Heap sort**

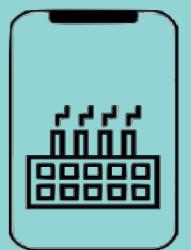
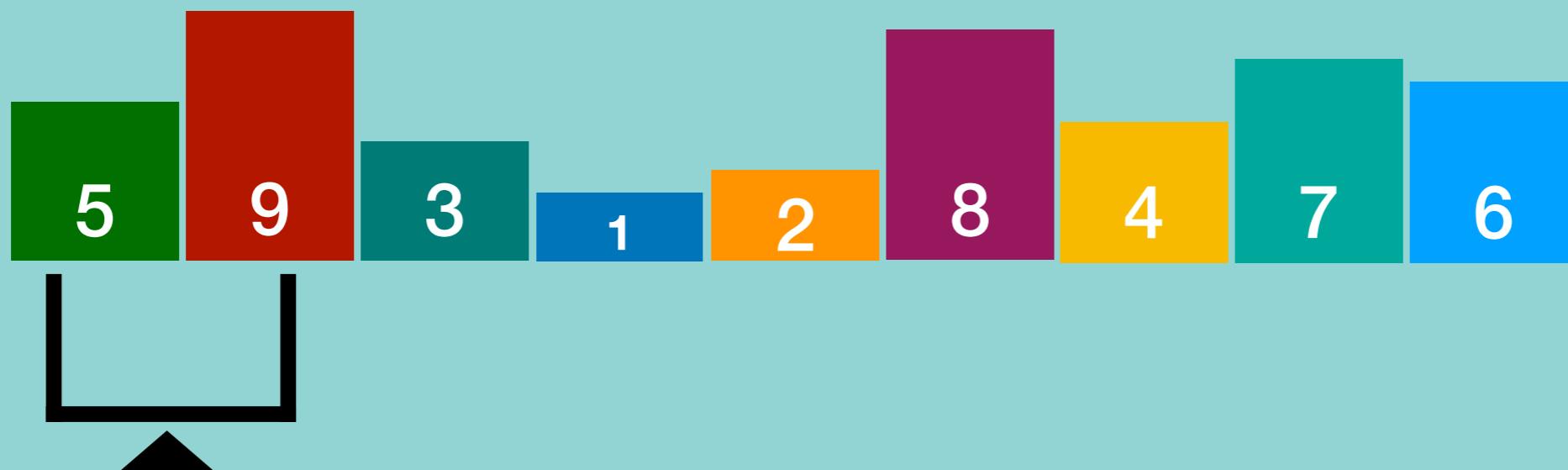
**Which one to select?**

- Stability
- Space efficient
- Time efficient



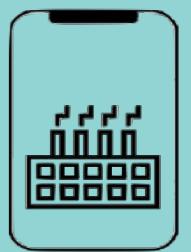
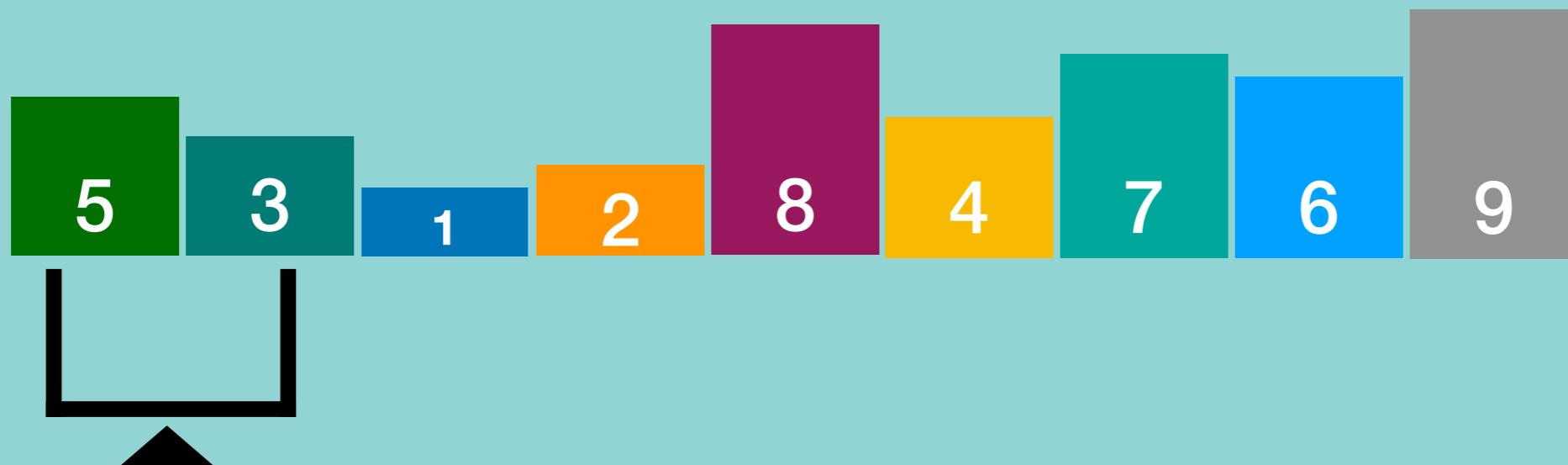
# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



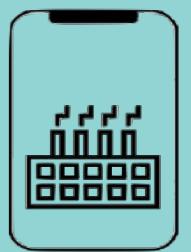
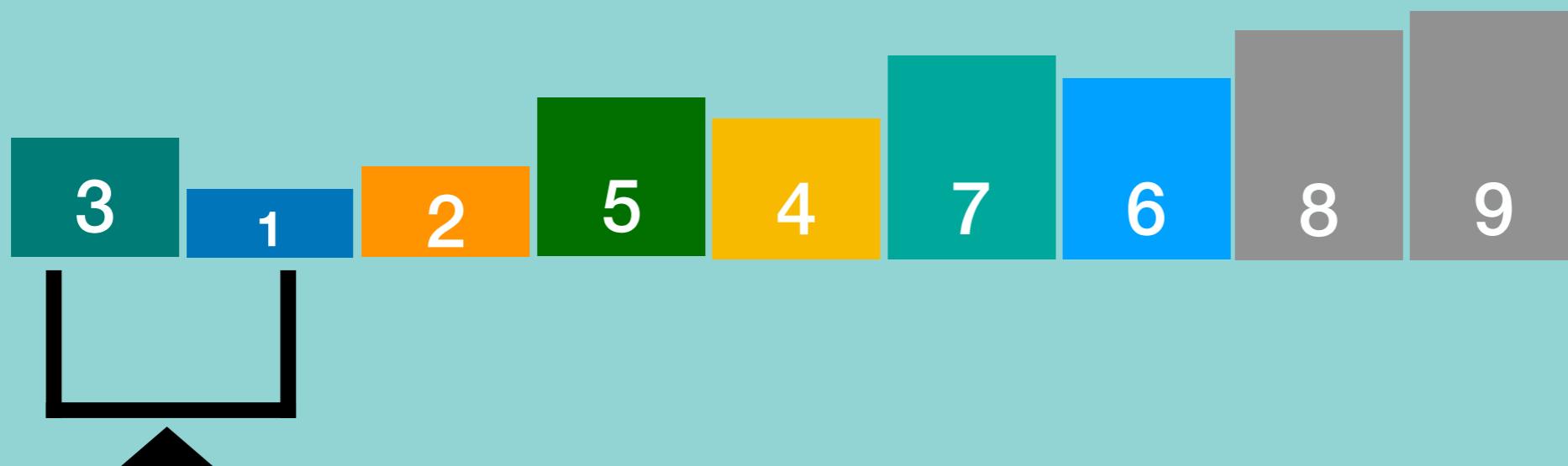
# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



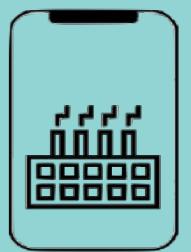
# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



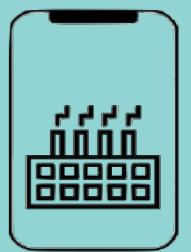
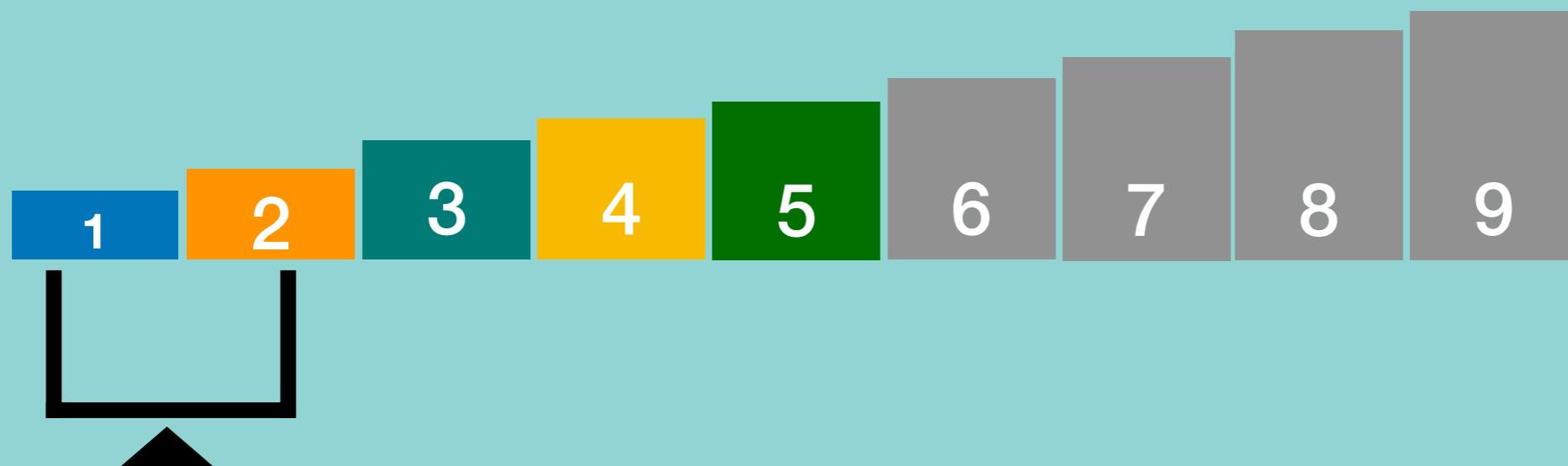
# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



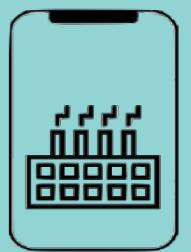
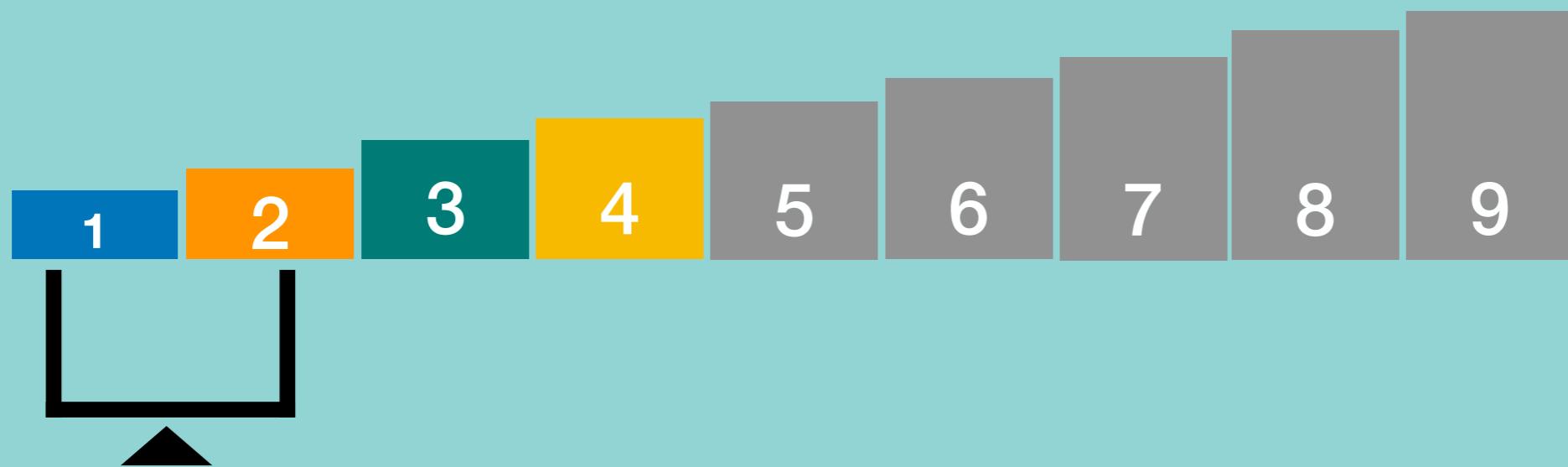
# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



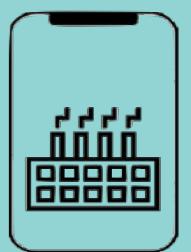
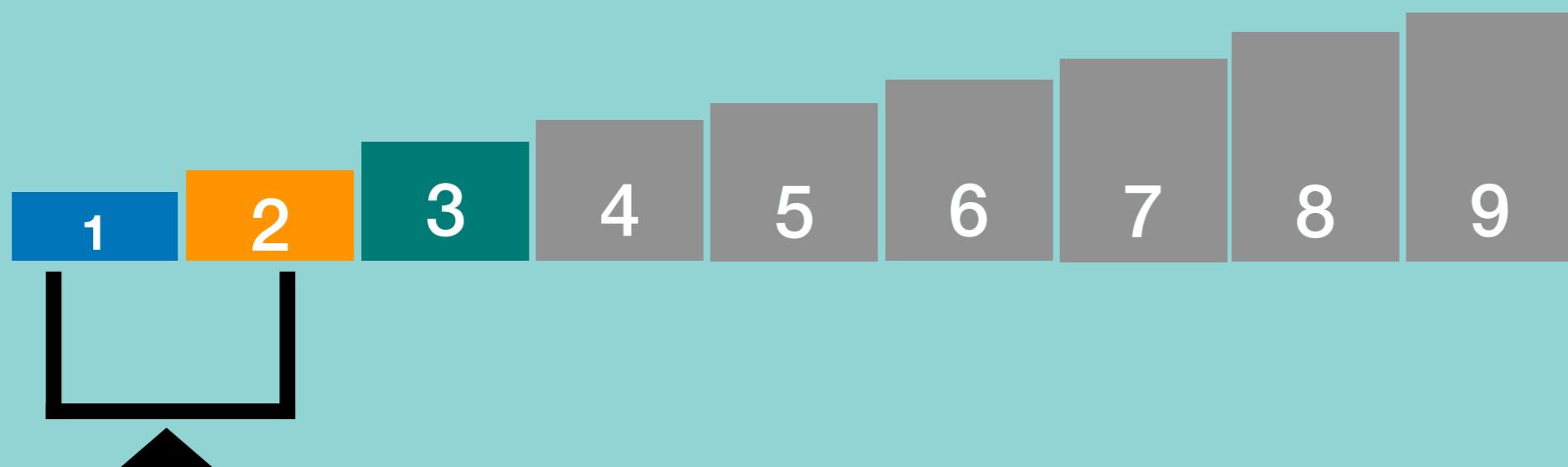
# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



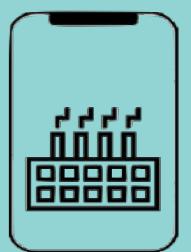
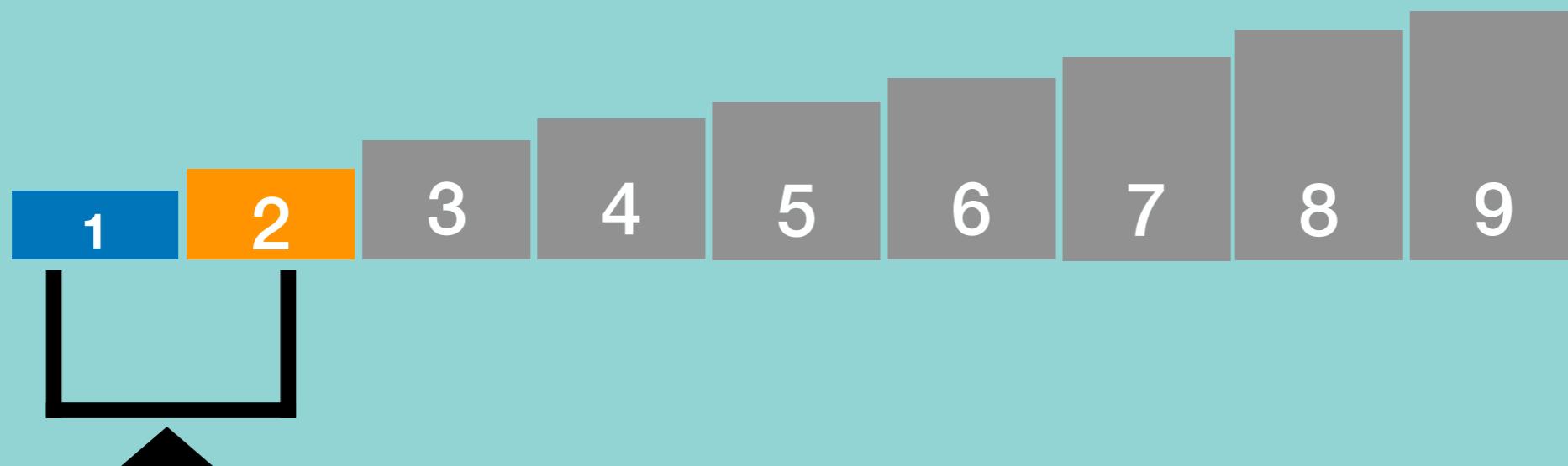
# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



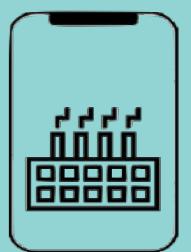
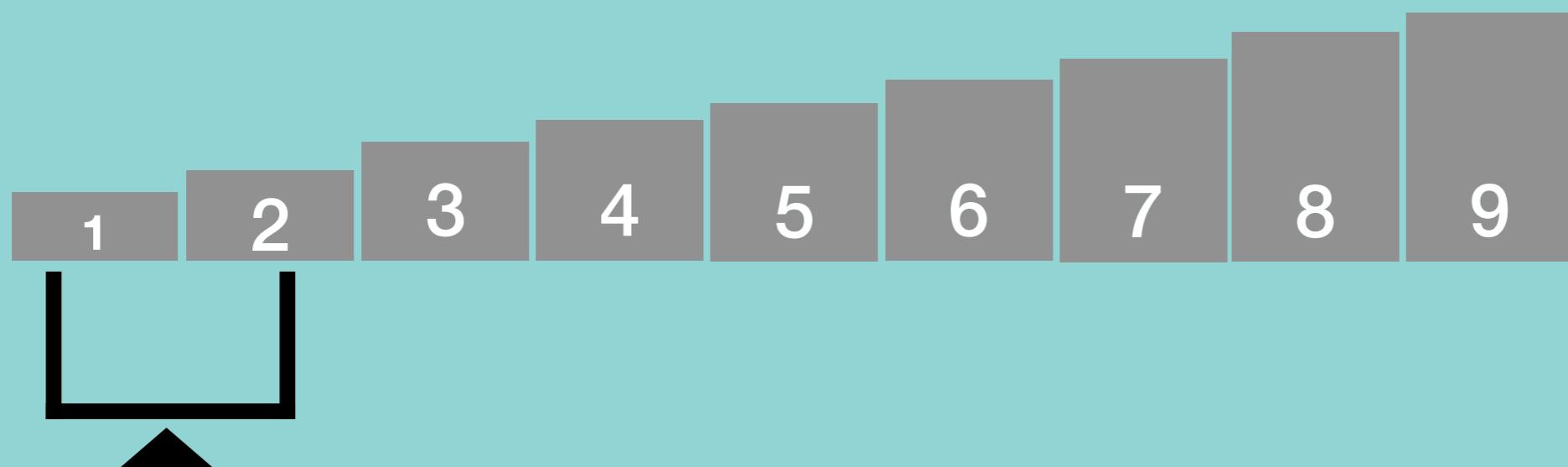
# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



# Bubble Sort

- Bubble sort is also referred as Sinking sort
- We repeatedly compare each pair of adjacent items and swap them if they are in the wrong order



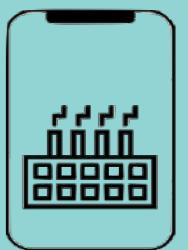
# Bubble Sort

## When to use Bubble Sort?

- When the input is already sorted
- Space is a concern
- Easy to implement

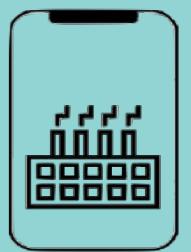
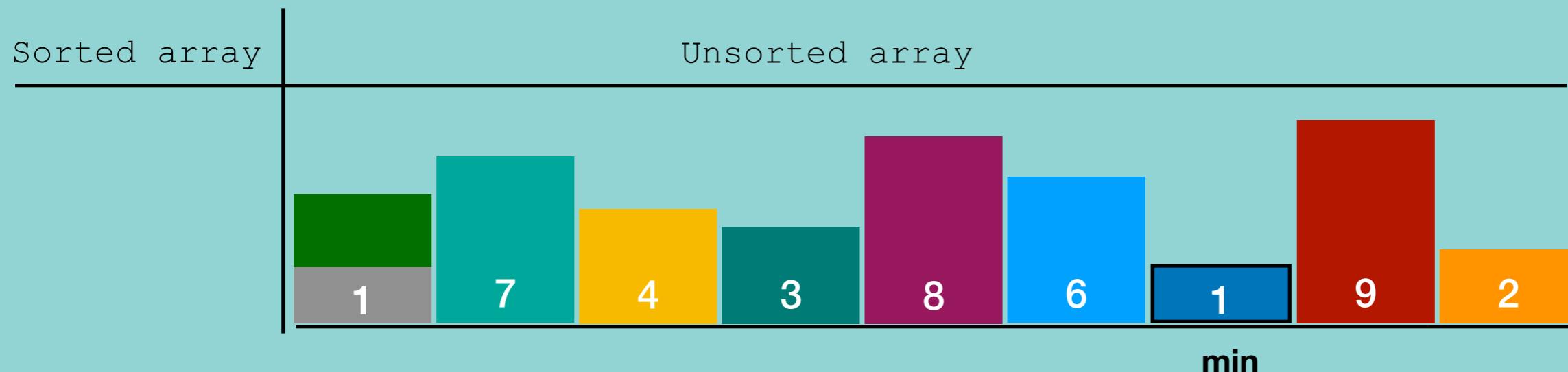
## When to avoid Bubble Sort?

- Average time complexity is poor



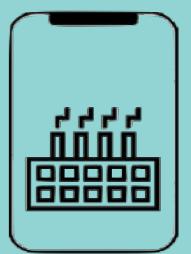
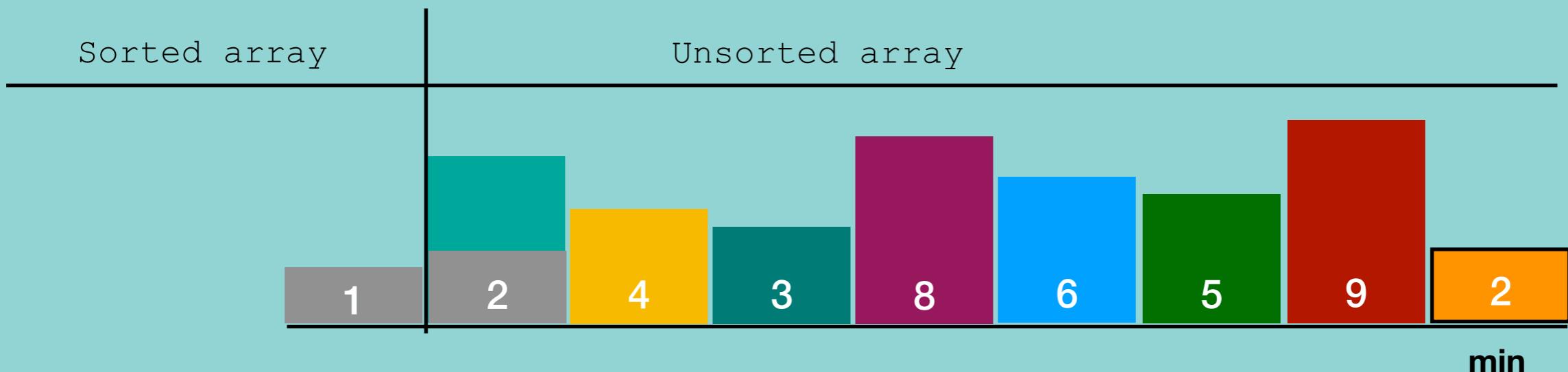
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



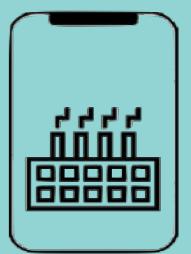
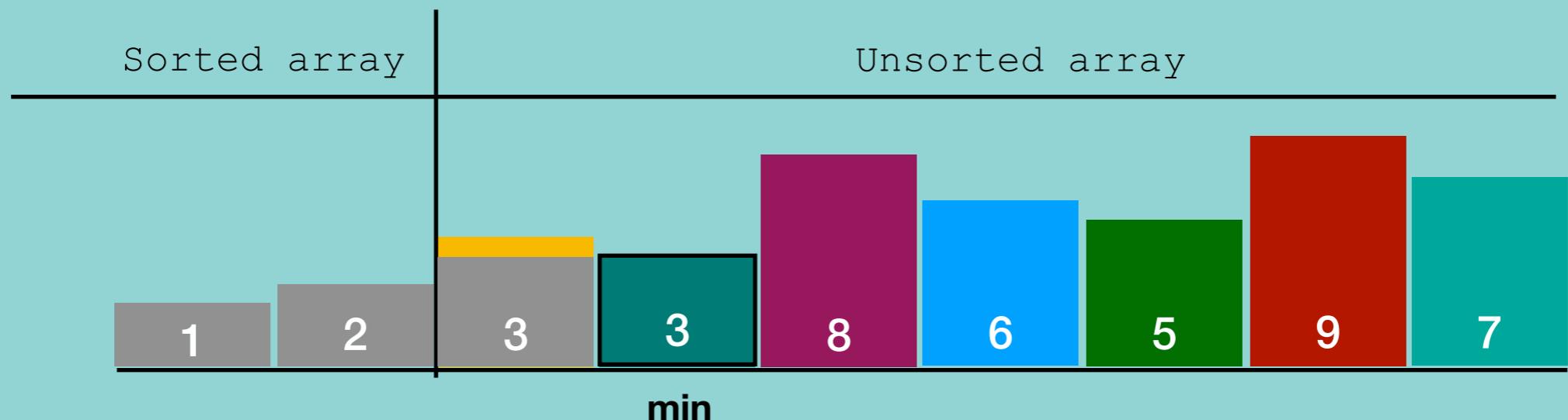
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



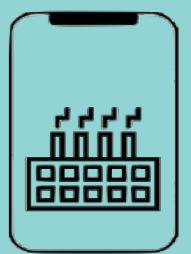
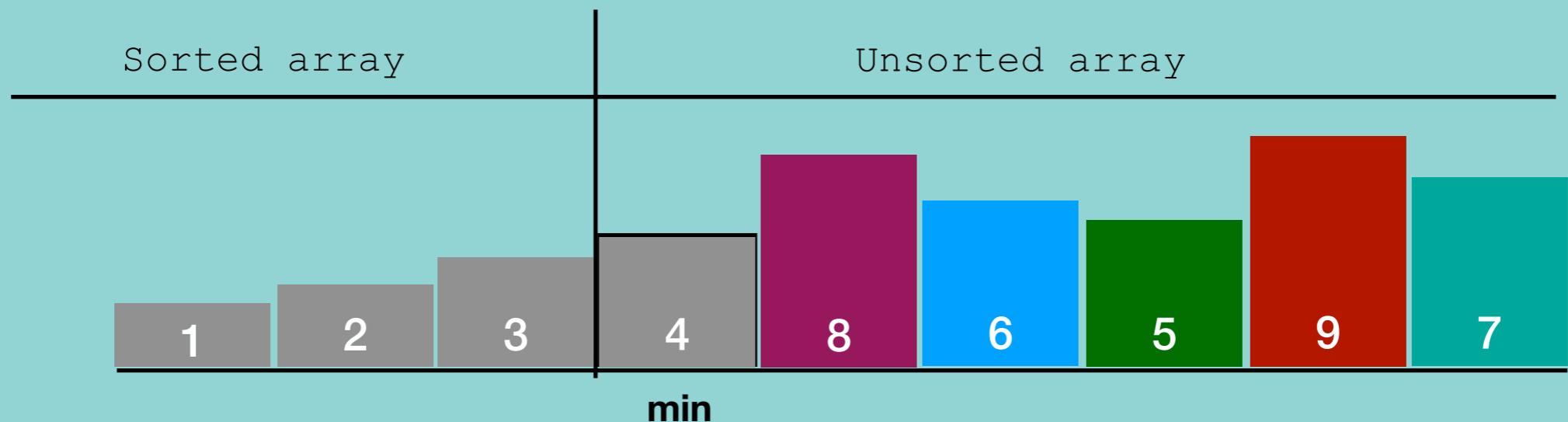
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



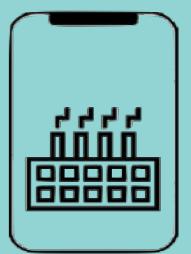
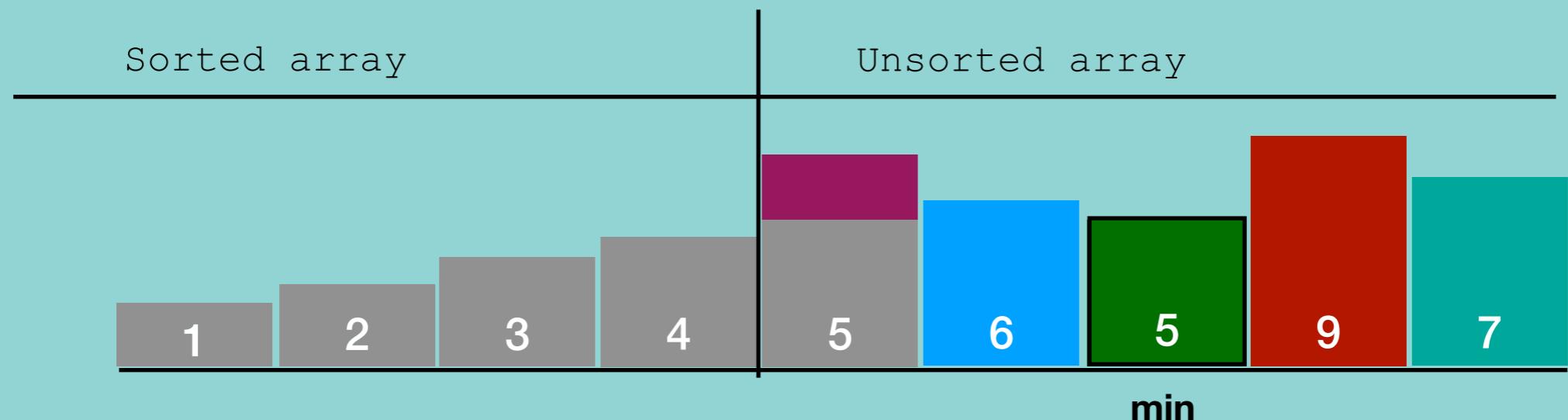
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



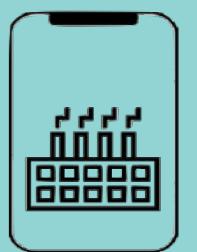
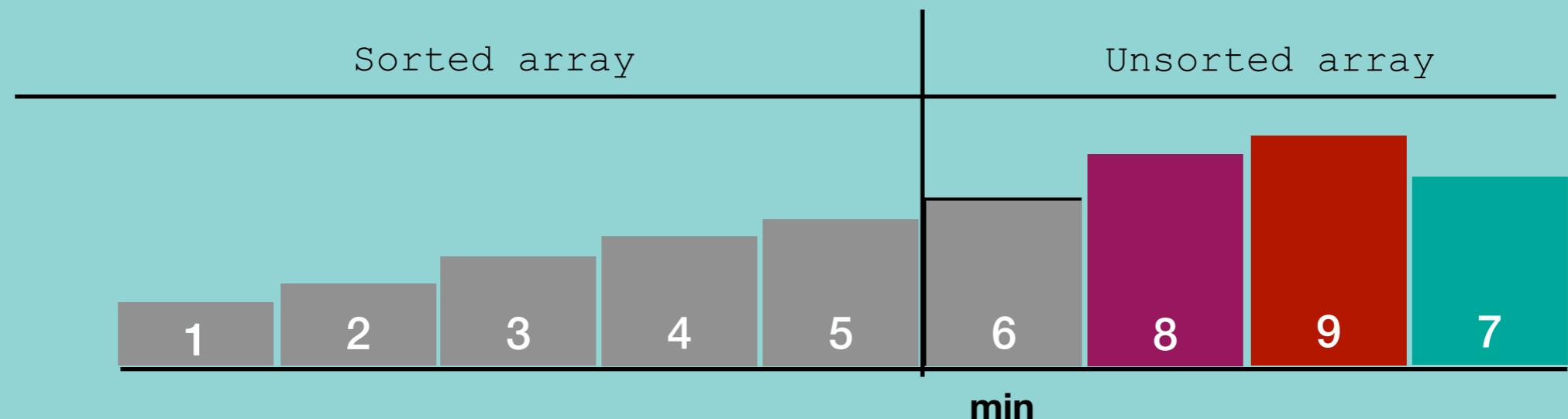
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



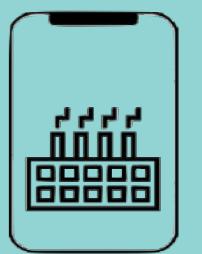
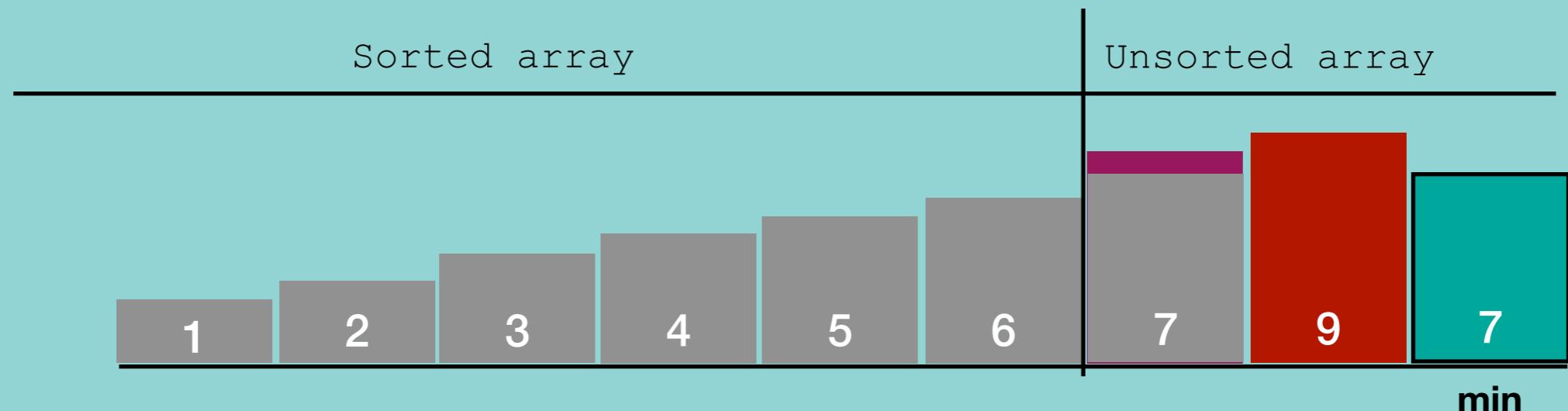
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



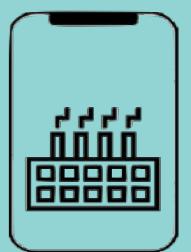
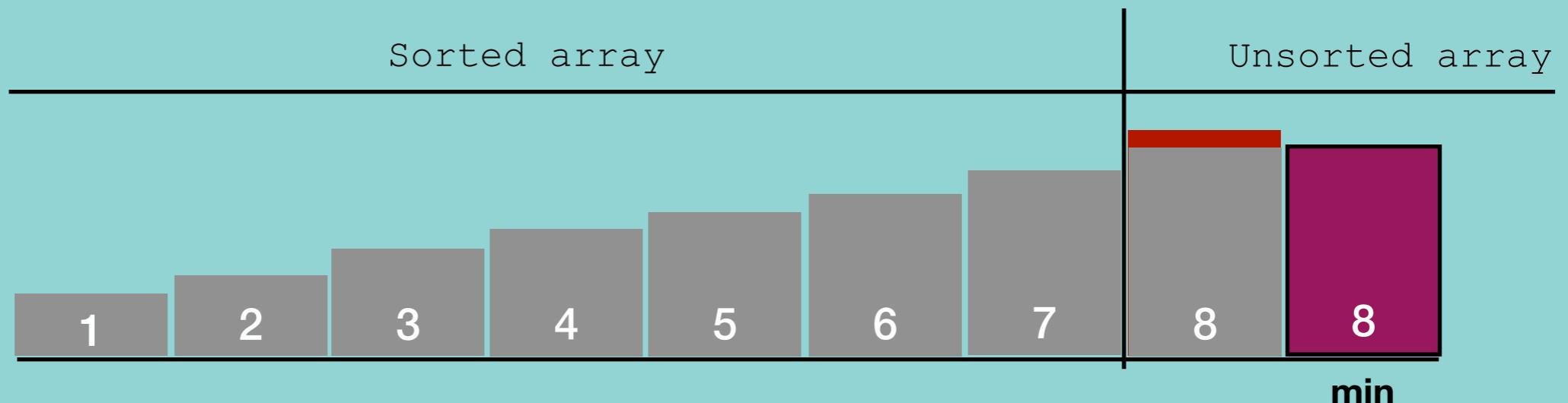
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



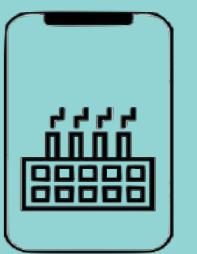
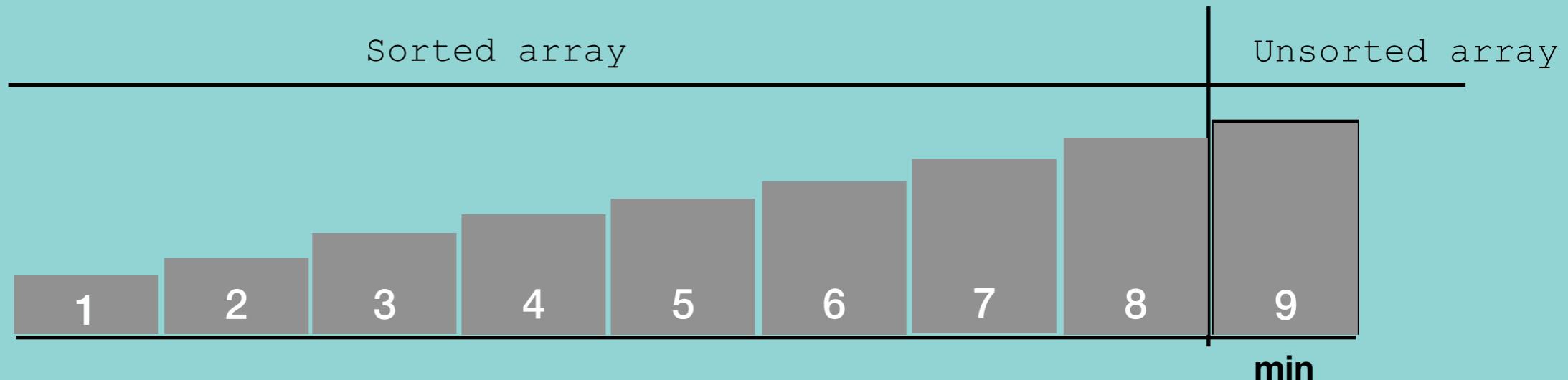
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



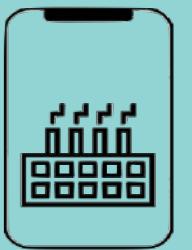
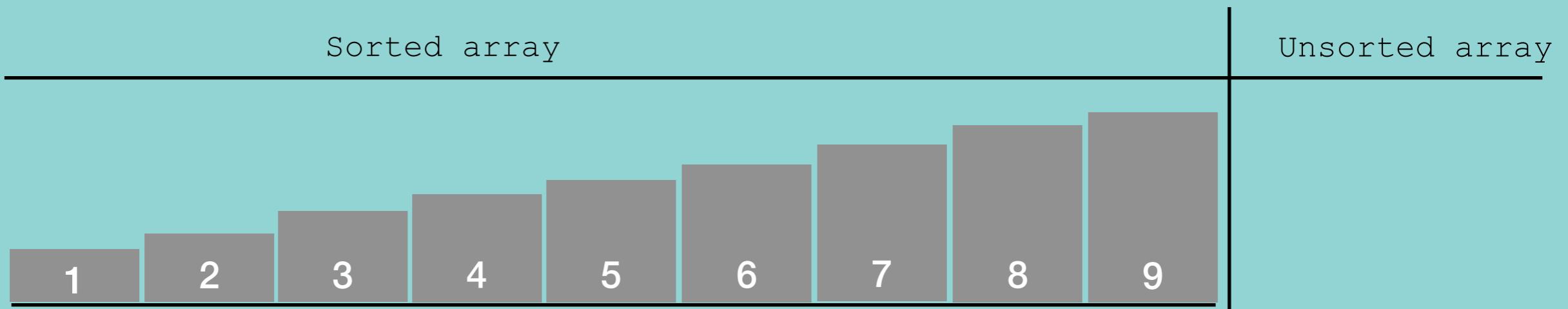
## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



## Selection Sort

- In case of selection sort we repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted.



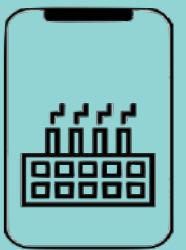
# Selection Sort

## When to use Selection Sort?

- When we have insufficient memory
- Easy to implement

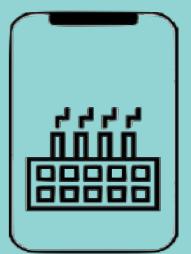
## When to avoid Selection Sort?

- When time is a concern



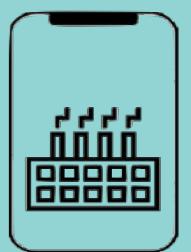
# Insertion Sort

- Divide the given array into two part
- Take first element from unsorted array and find its correct position in sorted array
- Repeat until unsorted array is empty



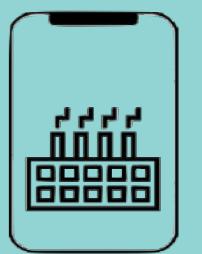
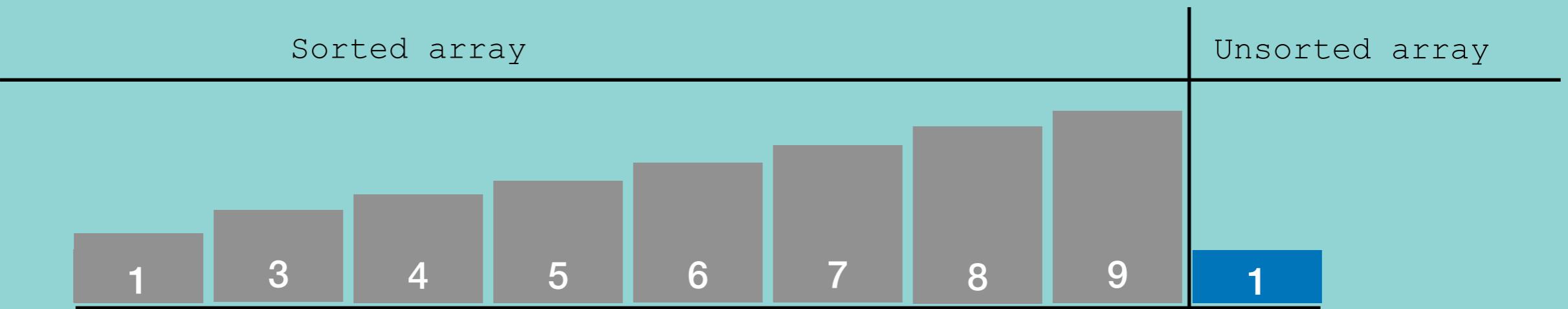
# Insertion Sort

- Divide the given array into two part
- Take first element from unsorted array and find its correct position in sorted array
- Repeat until unsorted array is empty



# Insertion Sort

- Divide the given array into two part
- Take first element from unsorted array and find its correct position in sorted array
- Repeat until unsorted array is empty



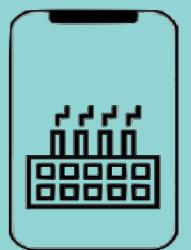
# Insertion Sort

## When to use Insertion Sort?

- When we have insufficient memory
- Easy to implement
- When we have continuous inflow of numbers and we want to keep them sorted

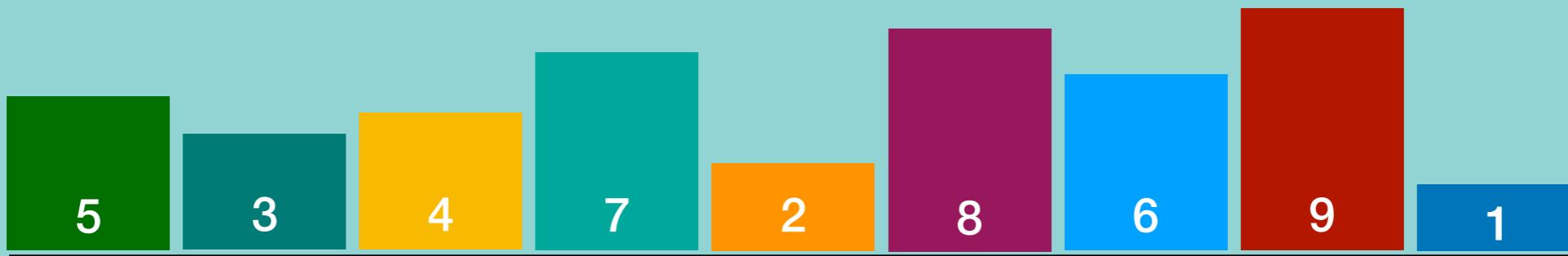
## When to avoid Insertion Sort?

- When time is a concern

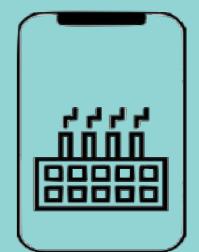
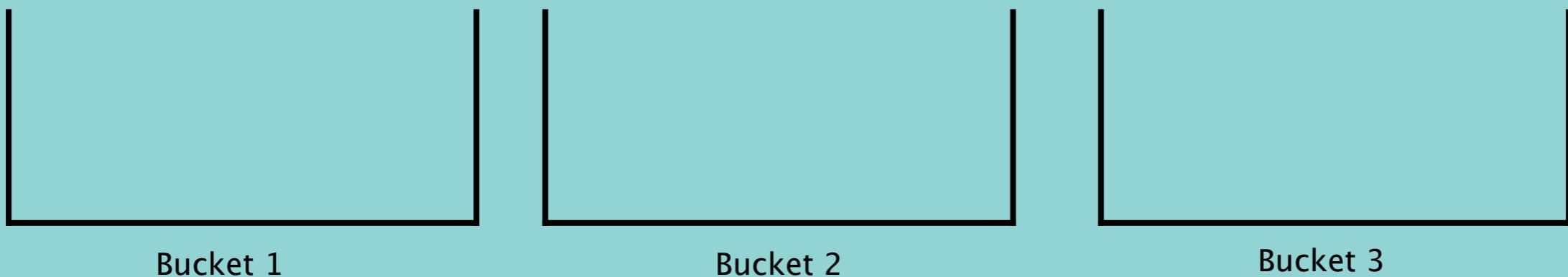


# Bucket Sort

- Create buckets and distribute elements of array into buckets
- Sort buckets individually
- Merge buckets after sorting

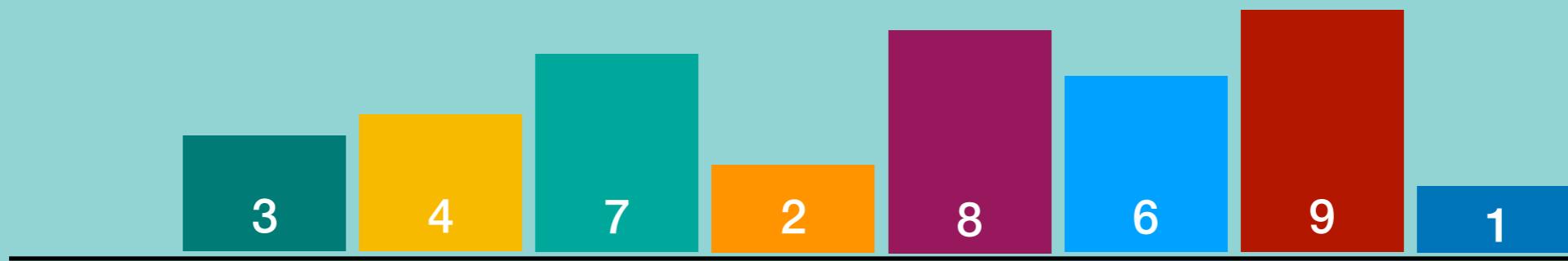


- Number of buckets =  $\text{round}(\text{Sqrt}(\text{number of elements}))$   
 $\text{round}(\sqrt{9}) = 3$
- Appropriate bucket =  $\text{ceil}(\text{Value} * \text{number of buckets} / \text{maxValue})$   
 $\text{ceil}(5 * 3 / 9) = \text{ceil}(1.6) = 2$

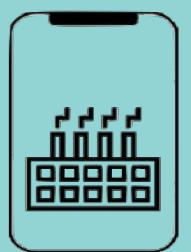
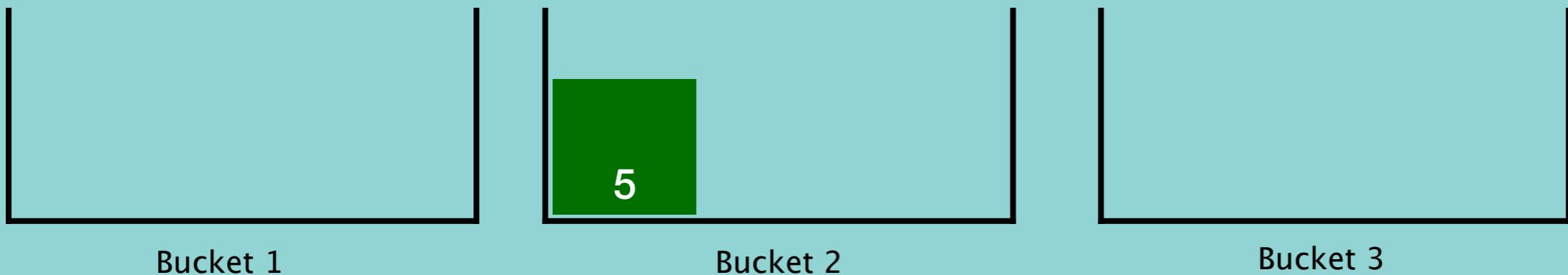


# Bucket Sort

- Create buckets and distribute elements of array into buckets
- Sort buckets individually
- Merge buckets after sorting

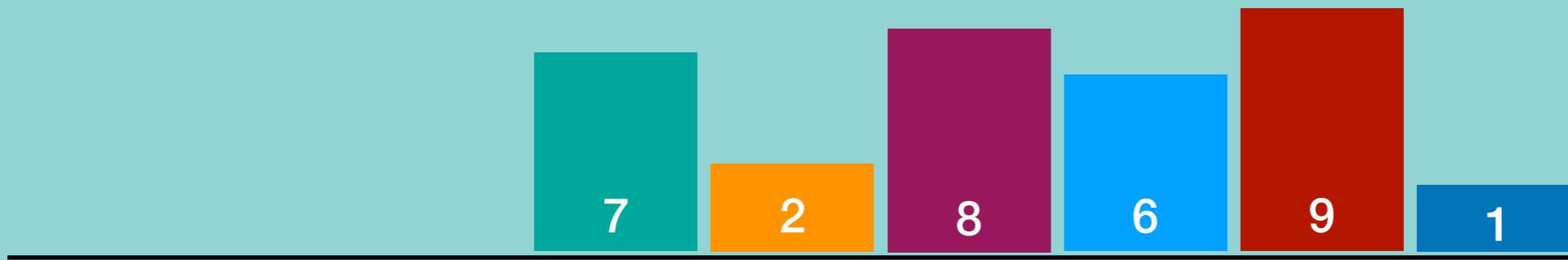


- Number of buckets = `round(Sqrt(number of elements))`  
 $\text{round}(\sqrt{9}) = 3$
- Appropriate bucket = `ceil(Value * number of buckets / maxValue)`  
 $\text{ceil}(4 * 3 / 9) = \text{ceil}(1.33) \geq 2$

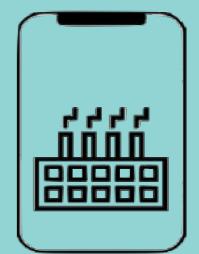
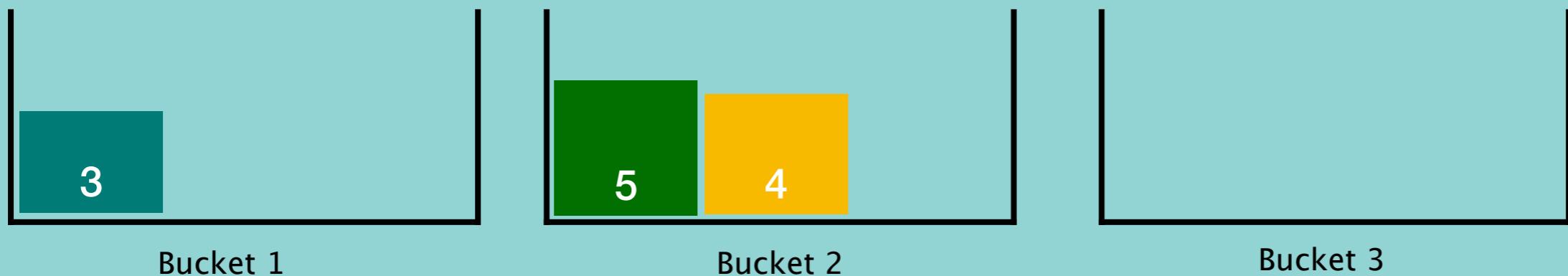


# Bucket Sort

- Create buckets and distribute elements of array into buckets
- Sort buckets individually
- Merge buckets after sorting

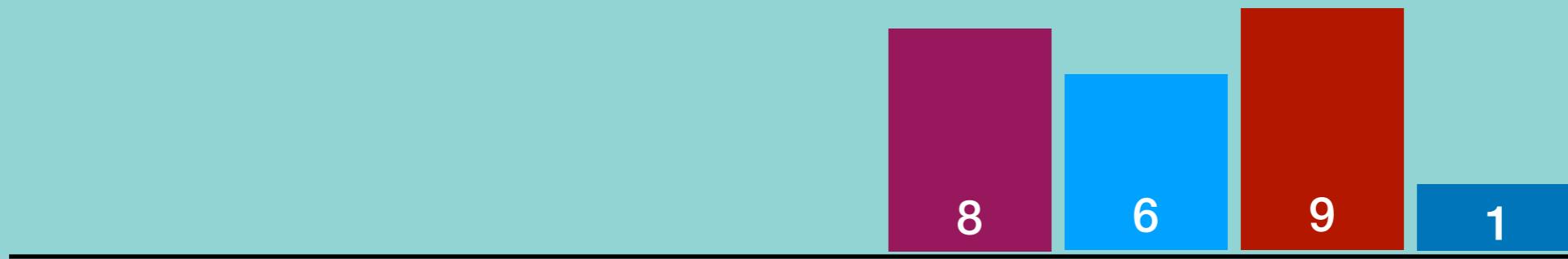


- Number of buckets = `round(Sqrt(number of elements))`  
 $\text{round}(\sqrt{9}) = 3$
- Appropriate bucket = `ceil(Value * number of buckets / maxValue)`  
 $\text{ceil}(2*3/9) = \text{ceil}(0.6) = 1$

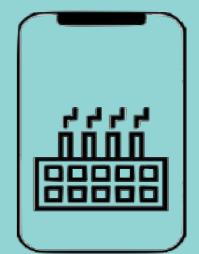
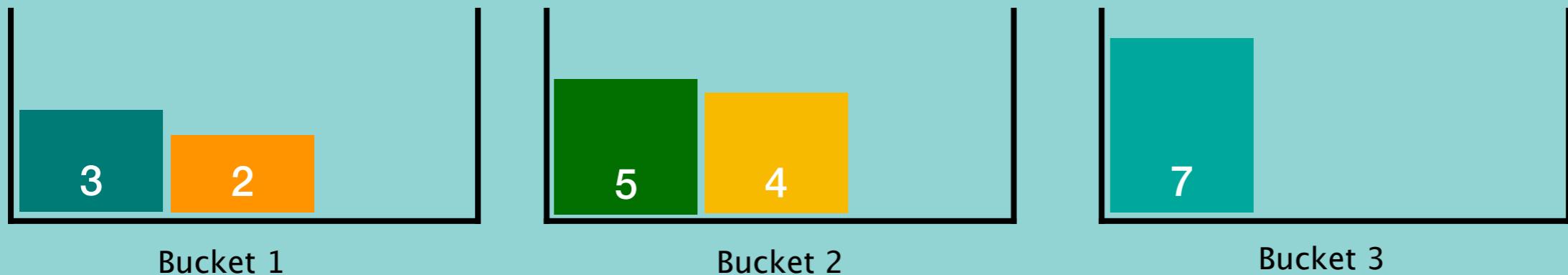


# Bucket Sort

- Create buckets and distribute elements of array into buckets
- Sort buckets individually
- Merge buckets after sorting

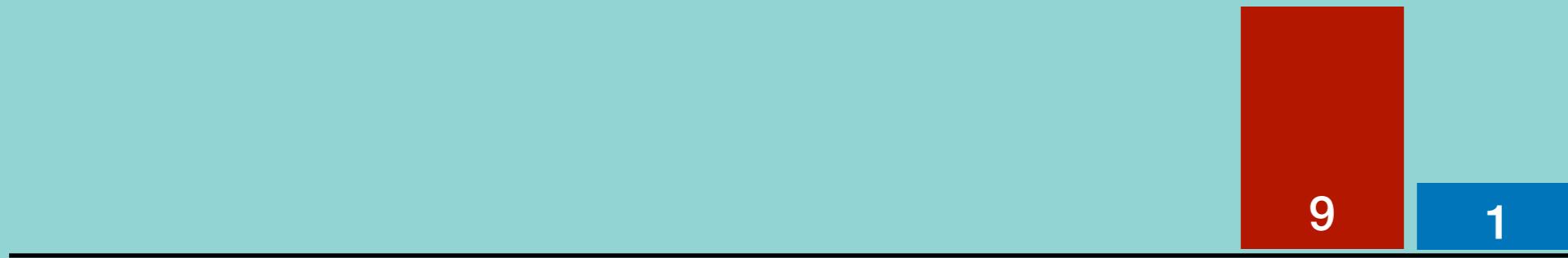


- Number of buckets =  $\text{round}(\text{Sqrt}(\text{number of elements}))$   
 $\text{round}(\sqrt{9}) = 3$
- Appropriate bucket =  $\text{ceil}(\text{Value} * \text{number of buckets} / \text{maxValue})$   
 $\text{ceil}(8 * 3 / 9) = \text{ceil}(2.67) \geq 3$

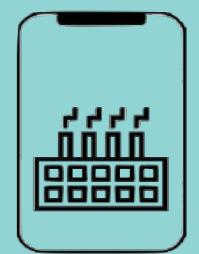
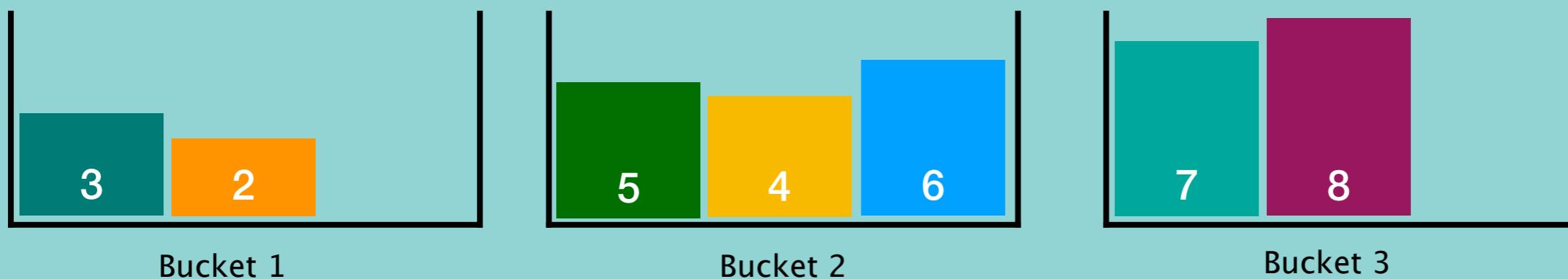


# Bucket Sort

- Create buckets and distribute elements of array into buckets
- Sort buckets individually
- Merge buckets after sorting



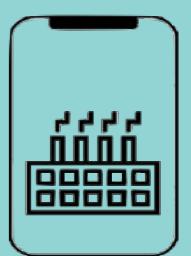
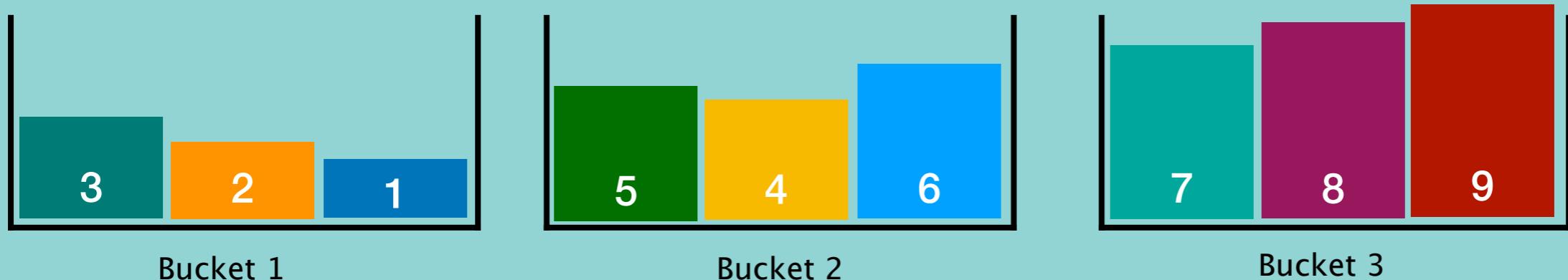
- Number of buckets = `round(Sqrt(number of elements))`  
 $\text{round}(\sqrt{9}) = 3$
- Appropriate bucket = `ceil(Value * number of buckets / maxValue)`
- $\text{ceil}(9 \cdot 3 / 9) = \text{ceil}(0) \geq 1$  (using any sorting algorithm)



# Bucket Sort

- Create buckets and distribute elements of array into buckets
- Sort buckets individually
- Merge buckets after sorting

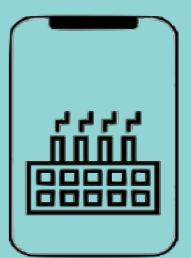
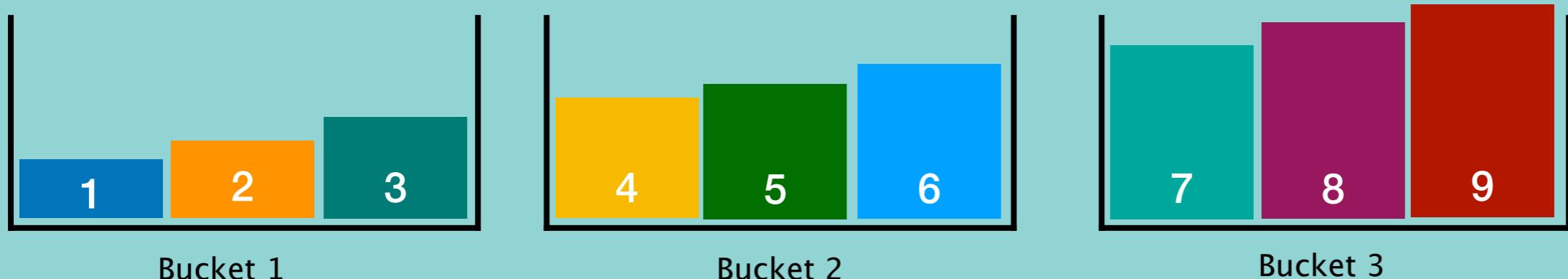
- 
- Number of buckets = `round(Sqrt(number of elements))`  
`round(sqrt(9)) = 3`
  - Appropriate bucket = `ceil(Value * number of buckets / maxValue)`
  - Sort all buckets (using any sorting algorithm)



# Bucket Sort

- Create buckets and distribute elements of array into buckets
- Sort buckets individually
- Merge buckets after sorting

- 
- Number of buckets = `round(Sqrt(number of elements))`  
`round(sqrt(9)) = 3`
  - Appropriate bucket = `ceil(Value * number of buckets / maxValue)`
  - Sort all buckets (using any sorting algorithm)



# Bucket Sort

## When to use Bucket Sort?

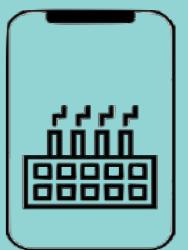
- When input uniformly distributed over range

1,2,4,5,3,8,7,9

1,2,4,~~9~~,93,95

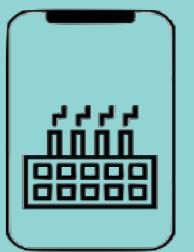
## When to avoid Bucket Sort?

- When space is a concern

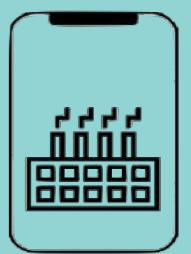
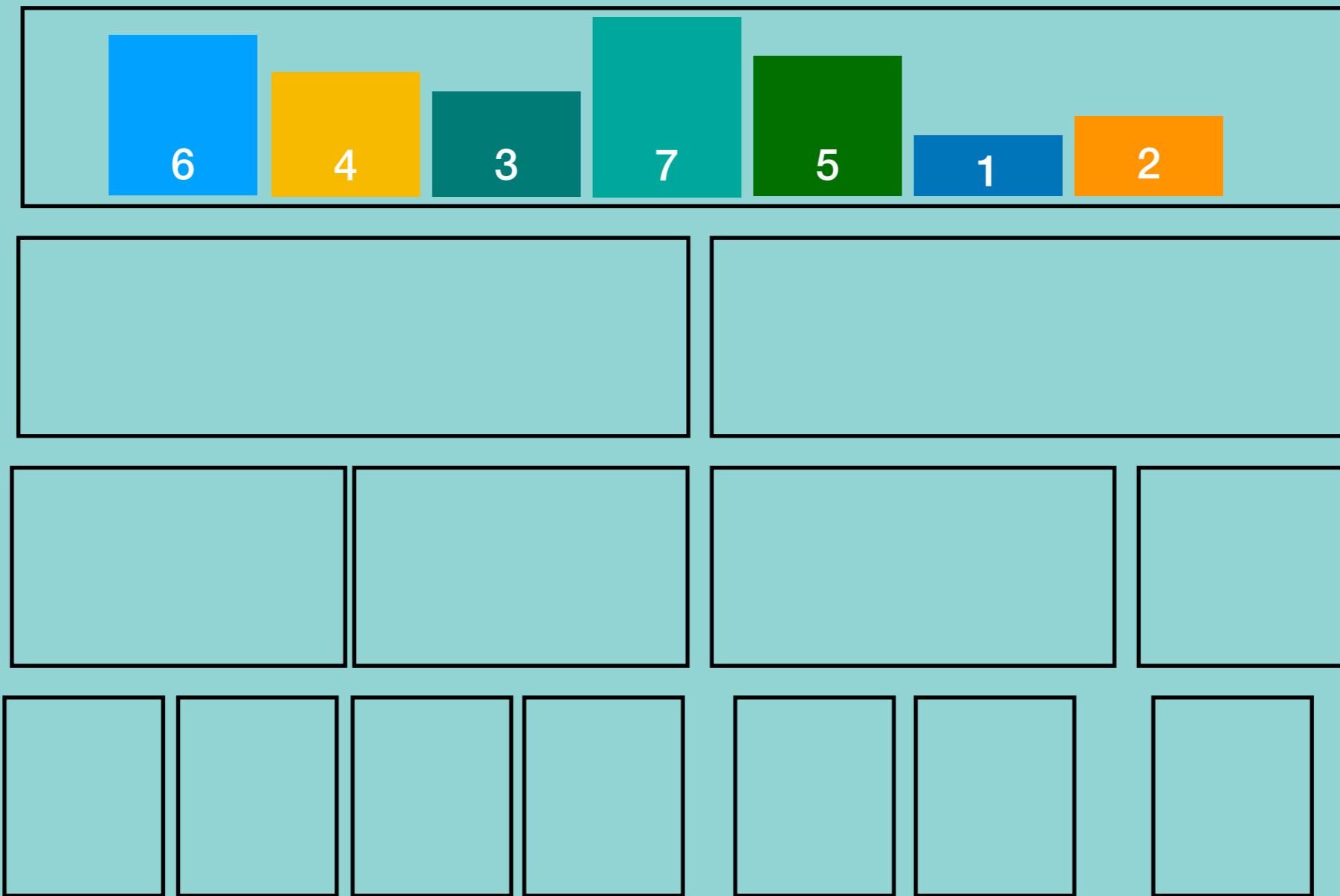


# Merger Sort

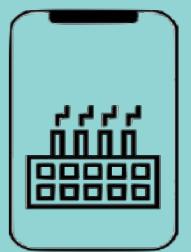
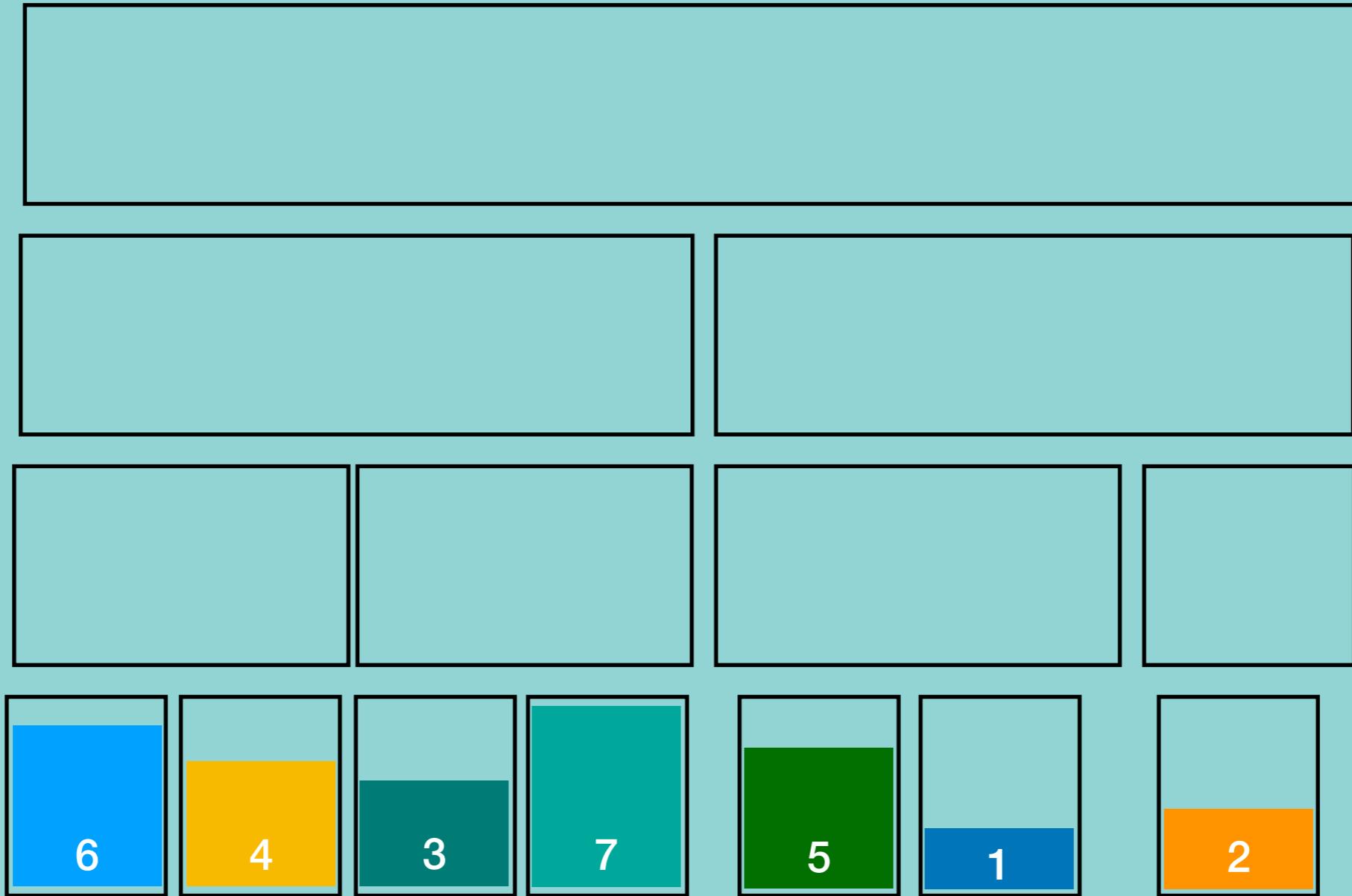
- Merge sort is a divide and conquer algorithm
- Divide the input array in two halves and we keep halving recursively until they become too small that cannot be broken further
- Merge halves by sorting them



# Merge Sort



# Merge Sort



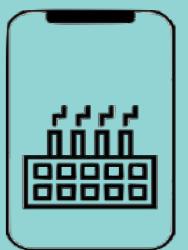
# Merge Sort

## When to use Merge Sort?

- When you need stable sort
- When average expected time is  $O(N \log N)$

## When to avoid Merge Sort?

- When space is a concern



# Quick Sort

- Quick sort is a divide and conquer algorithm
- Find pivot number and make sure smaller numbers located at the left of pivot and bigger numbers are located at the right of the pivot.
- Unlike merge sort extra space is not required

70	10	80	30	90	40	60	20	50
----	----	----	----	----	----	----	----	----

10	30	40	20	50	80	60	70	90
----	----	----	----	----	----	----	----	----

10	20	40	30	50	80	60	70	90
----	----	----	----	----	----	----	----	----

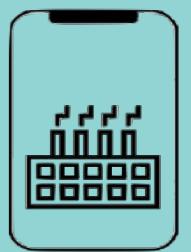
10	20	30	40	50	80	60	70	90
----	----	----	----	----	----	----	----	----

10	20	30	40	50	80	60	70	90
----	----	----	----	----	----	----	----	----

10	20	30	40	50	60	70	80	90
----	----	----	----	----	----	----	----	----

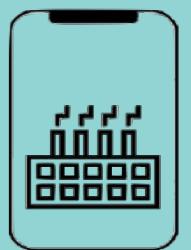
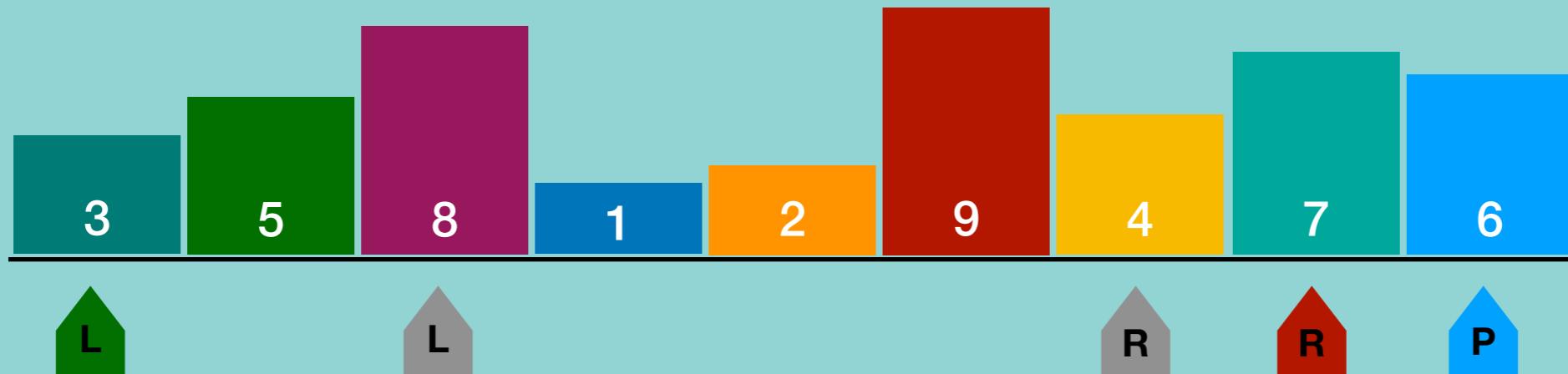
10	20	30	40	50	60	70	80	90
----	----	----	----	----	----	----	----	----

10	20	30	40	50	60	70	80	90
----	----	----	----	----	----	----	----	----



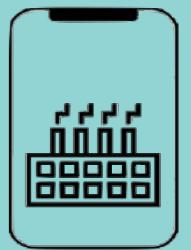
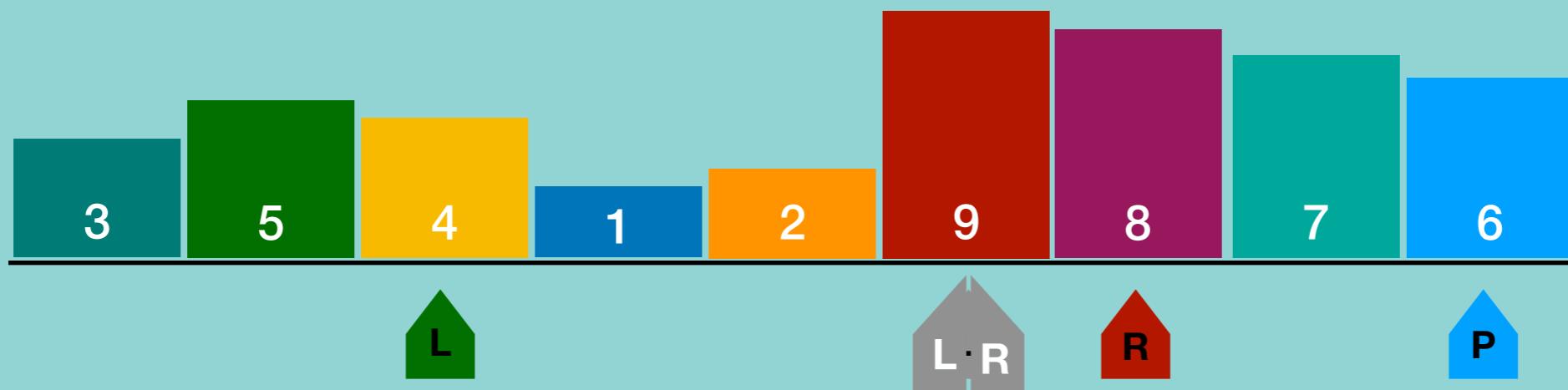
## Quick Sort

- Quick sort is a divide and conquer algorithm
- Find pivot number and make sure smaller numbers located at the left of pivot and bigger numbers are located at the right of the pivot.
- Unlike merge sort extra space is not required



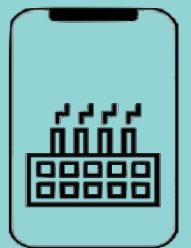
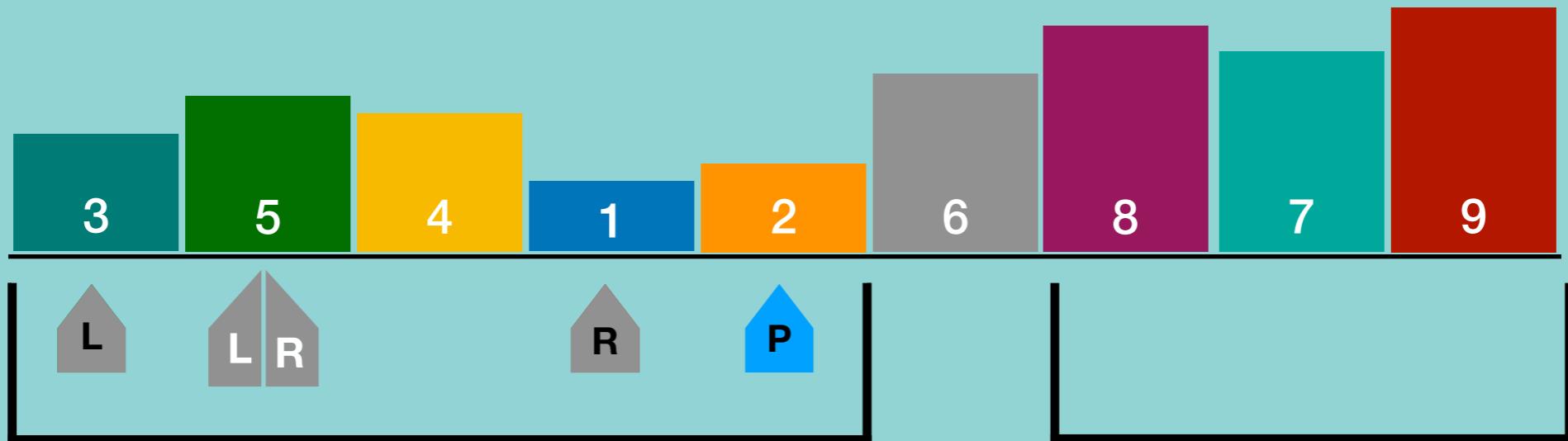
## Quick Sort

- Quick sort is a divide and conquer algorithm
- Find pivot number and make sure smaller numbers located at the left of pivot and bigger numbers are located at the right of the pivot.
- Unlike merge sort extra space is not required



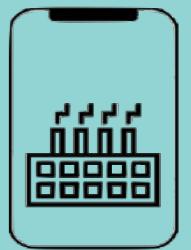
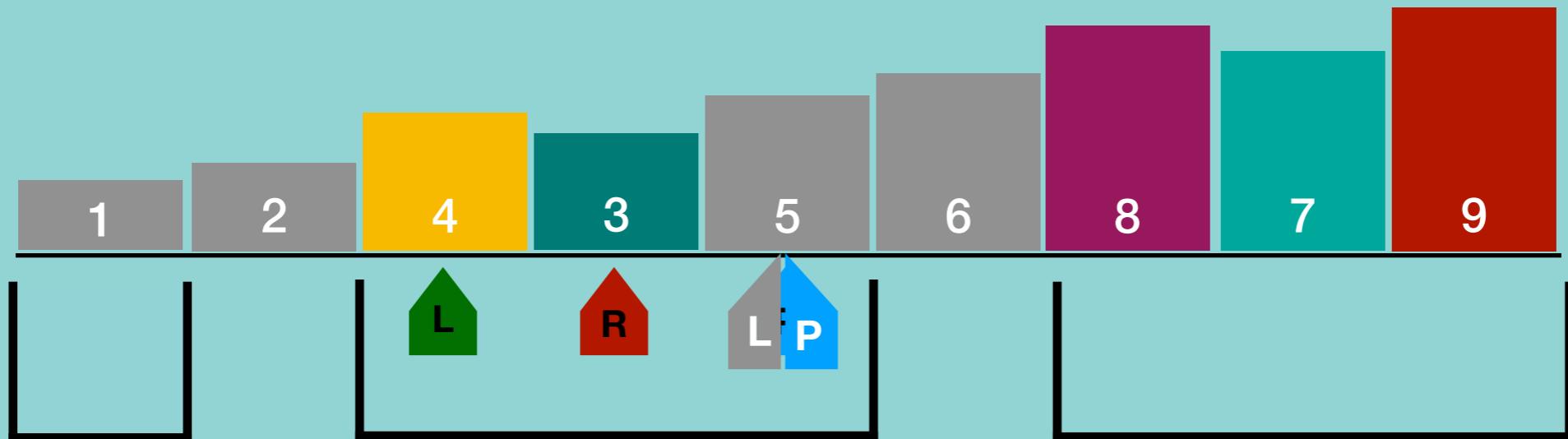
# Quick Sort

- Quick sort is a divide and conquer algorithm
- Find pivot number and make sure smaller numbers located at the left of pivot and bigger numbers are located at the right of the pivot.
- Unlike merge sort extra space is not required



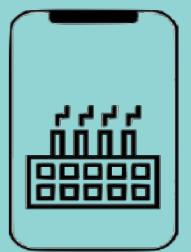
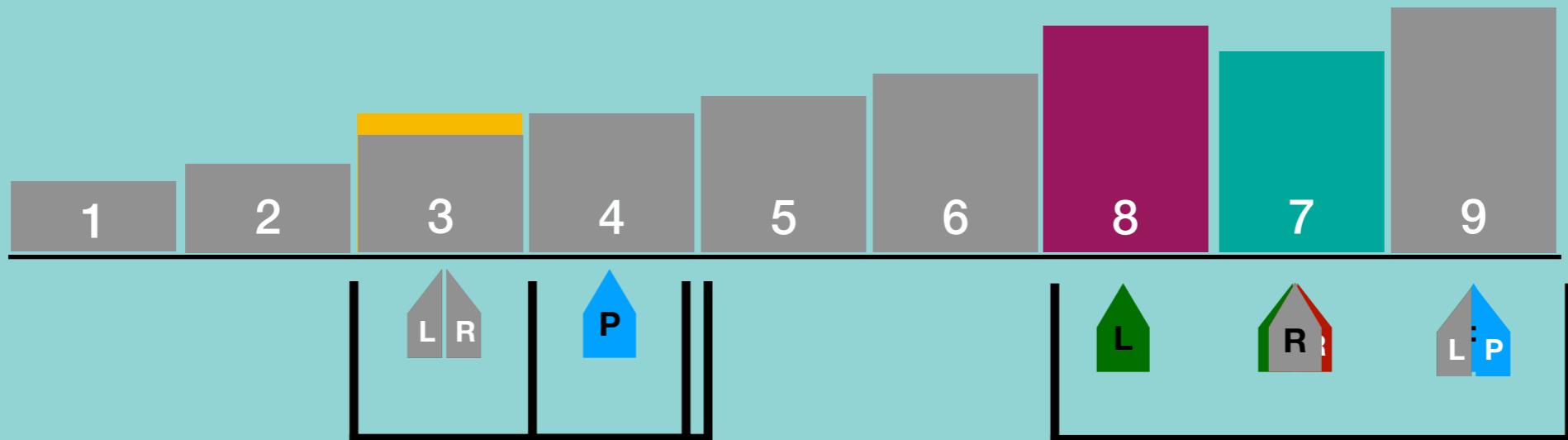
# Quick Sort

- Quick sort is a divide and conquer algorithm
- Find pivot number and make sure smaller numbers located at the left of pivot and bigger numbers are located at the right of the pivot.
- Unlike merge sort extra space is not required



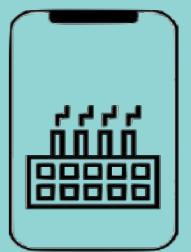
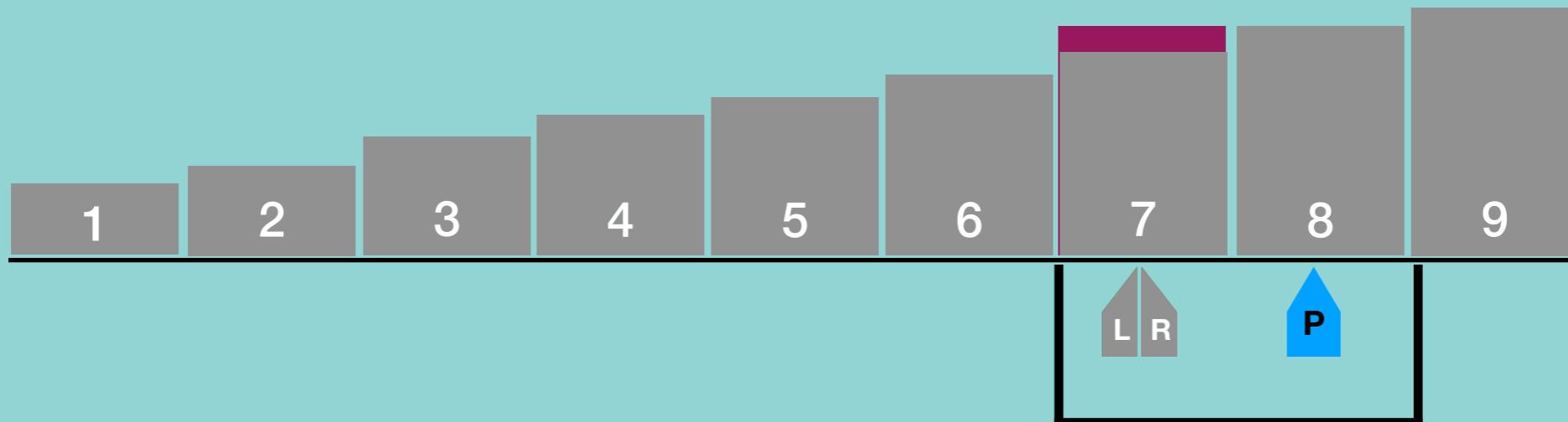
# Quick Sort

- Quick sort is a divide and conquer algorithm
- Find pivot number and make sure smaller numbers located at the left of pivot and bigger numbers are located at the right of the pivot.
- Unlike merge sort extra space is not required



# Quick Sort

- Quick sort is a divide and conquer algorithm
- Find pivot number and make sure smaller numbers located at the left of pivot and bigger numbers are located at the right of the pivot.
- Unlike merge sort extra space is not required



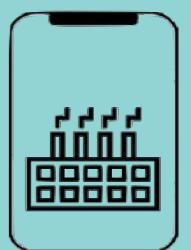
# Quick Sort

## When to use Quick Sort?

- When average expected time is  $O(N \log N)$

## When to avoid Quick Sort?

- When space is a concern
- When you need stable sort

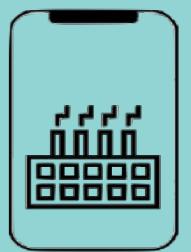
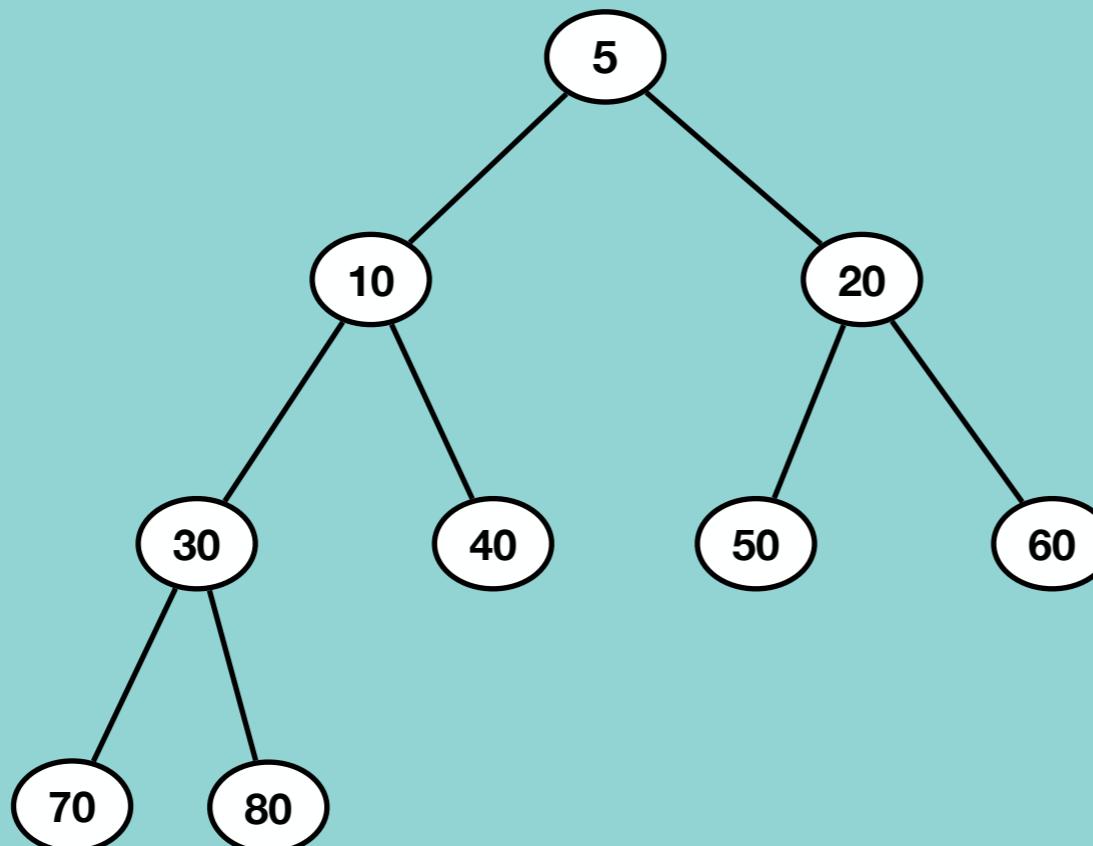


# Heap Sort

- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap
- It is best suited with array, it does not work with Linked List

**Binary Heap** is a binary tree with special properties

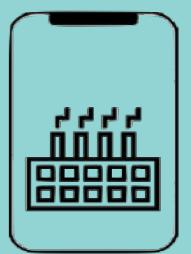
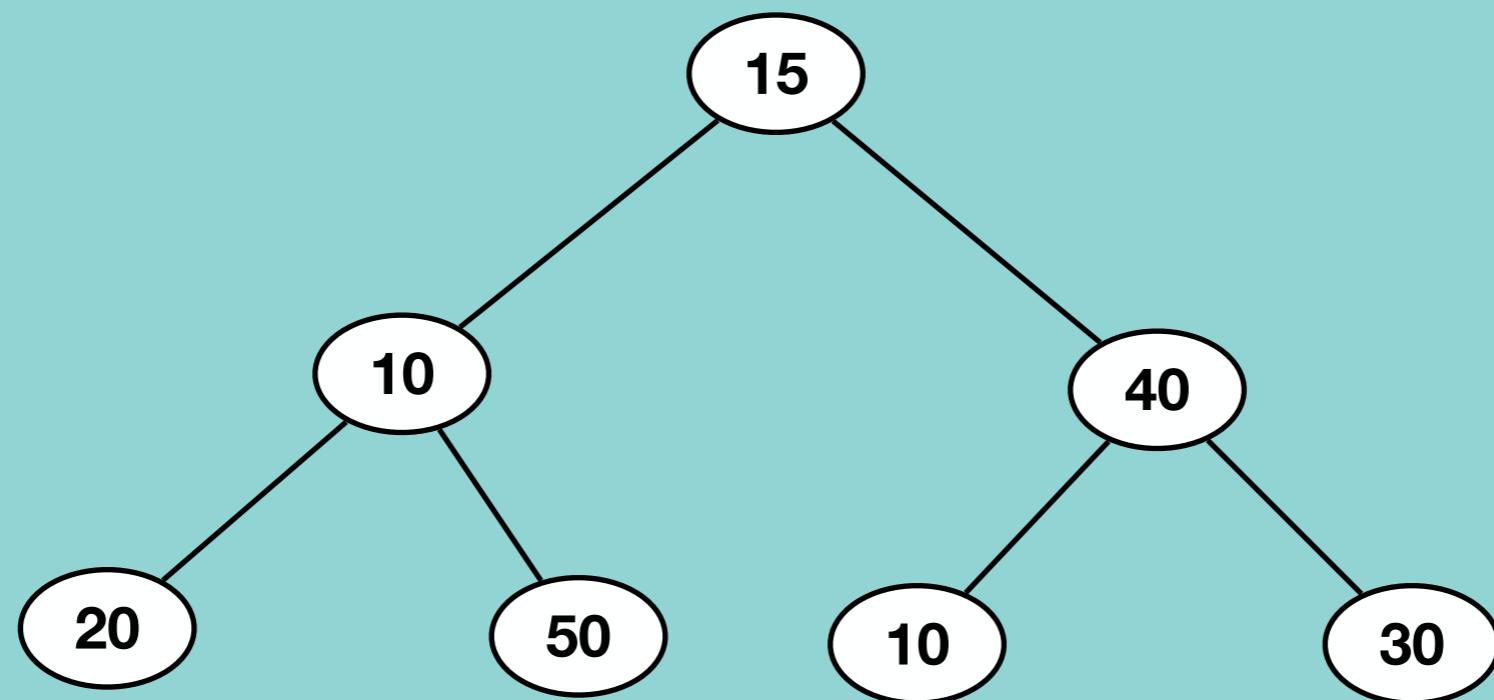
- The value of any given node must be less or equal of its children (min heap)
- The value of any given node must be greater or equal of its children (max heap)



# Heap Sort

- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap

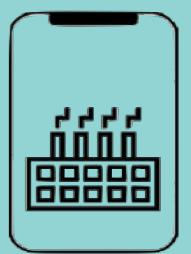
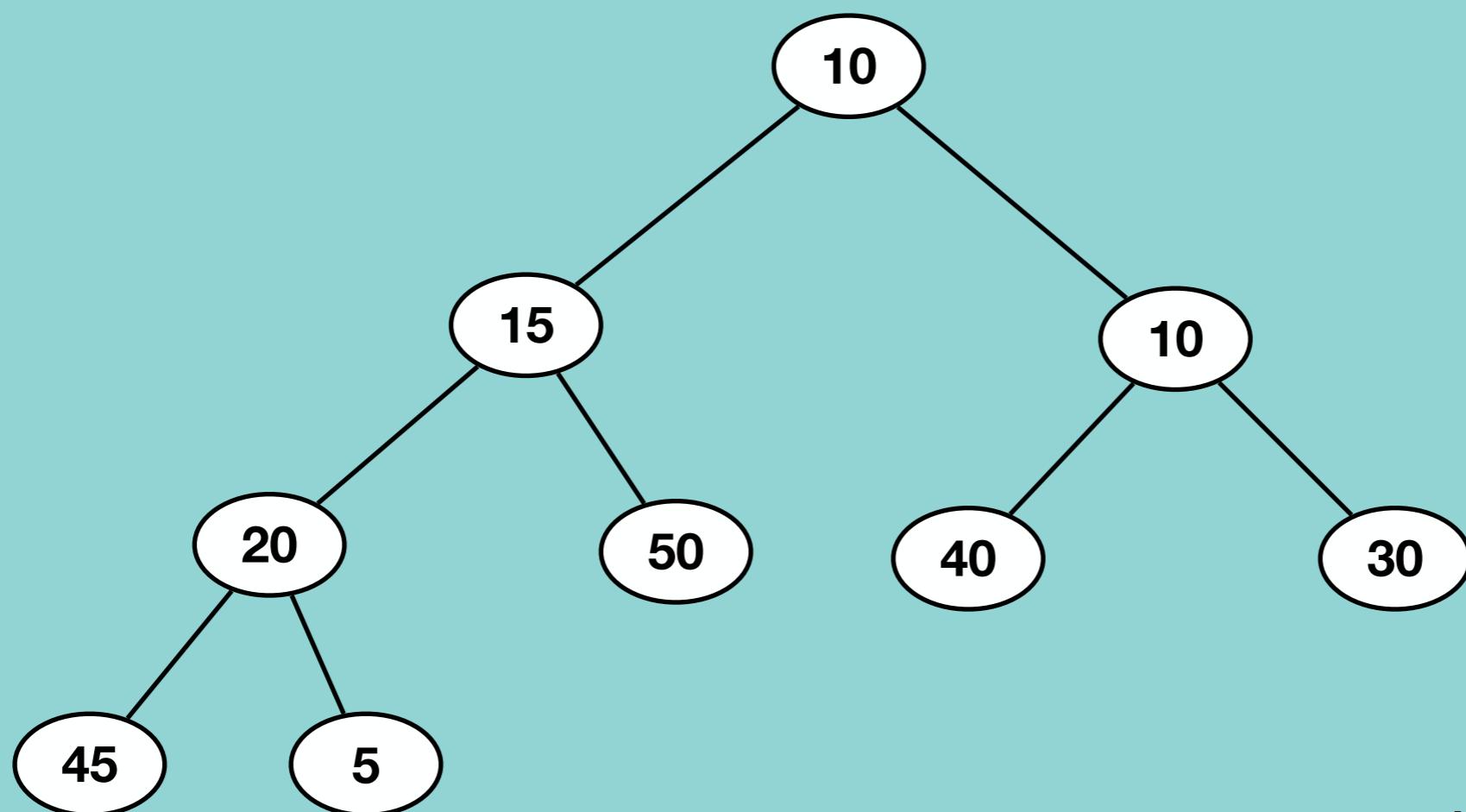
15	10	40	20	50	10	30	45	5
----	----	----	----	----	----	----	----	---



# Heap Sort

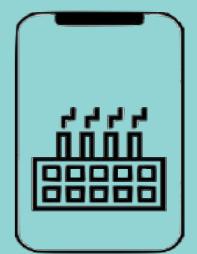
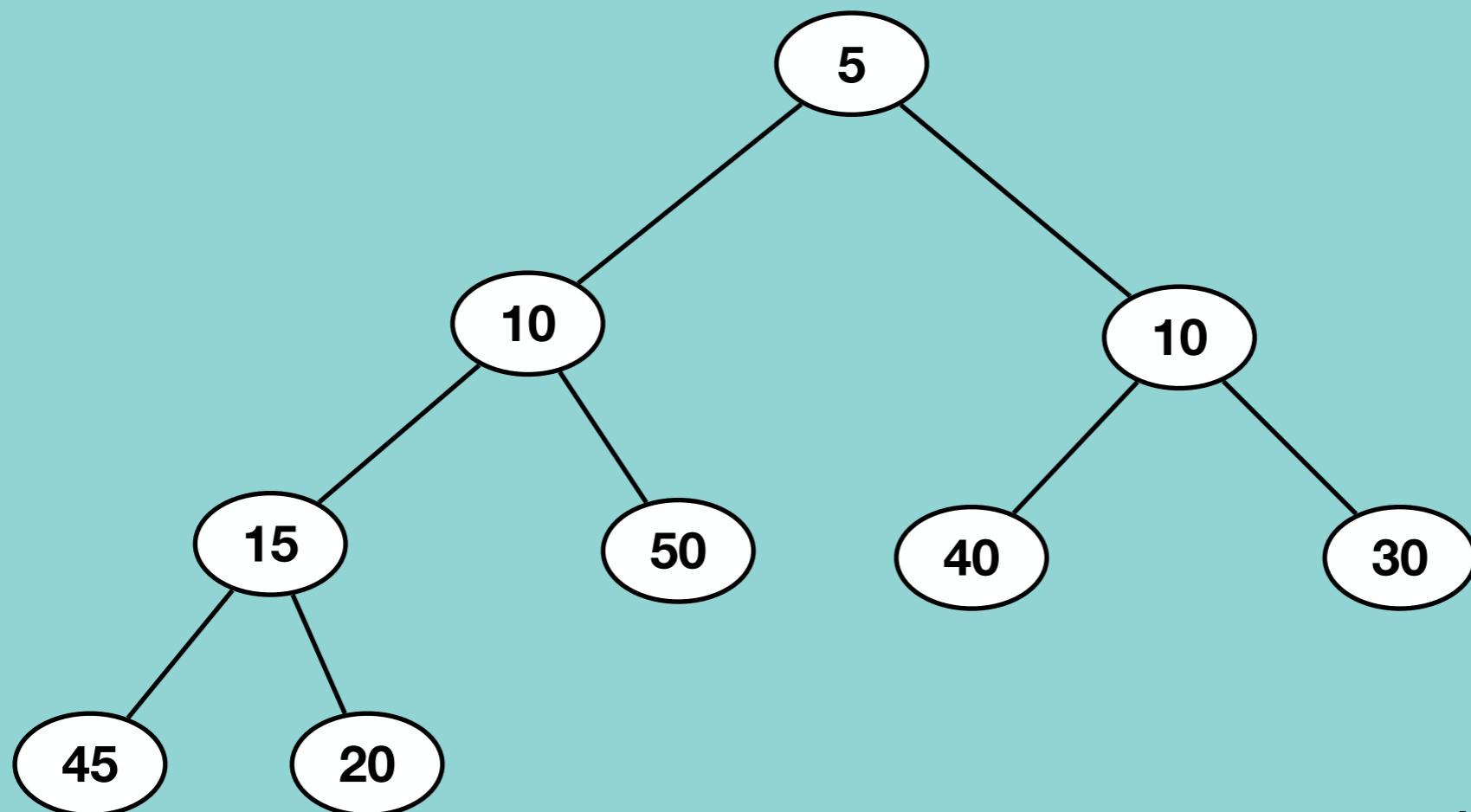
- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap

15	10	40	20	50	10	30	45	5
----	----	----	----	----	----	----	----	---



# Heap Sort

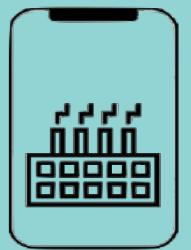
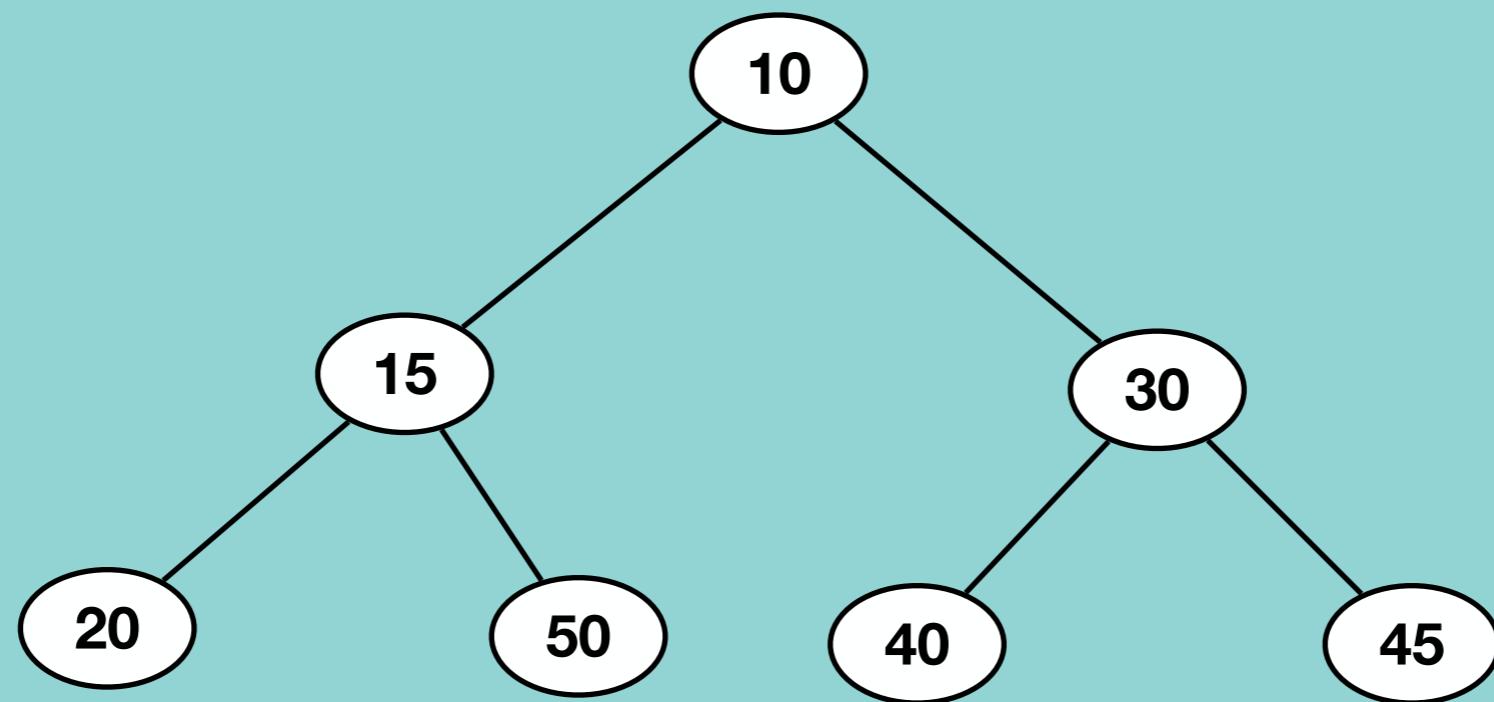
- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap



# Heap Sort

- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap

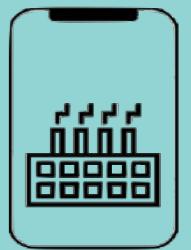
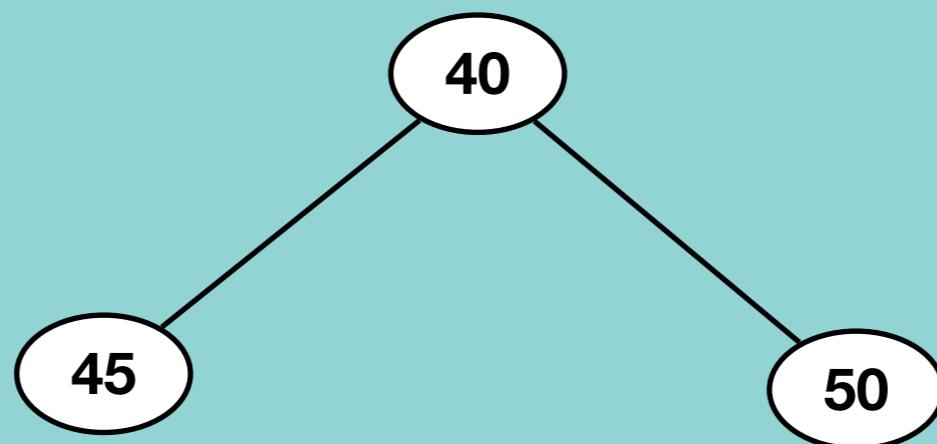
5	10	10	15	20	30			
---	----	----	----	----	----	--	--	--



# Heap Sort

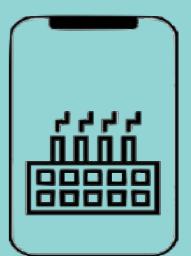
- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap

5	10	10	15	20	30	40	45	50
---	----	----	----	----	----	----	----	----



# Sorting Algorithms

Name	Time Complexity	Space Complexity	Stable
Bubble Sort	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n^2)$	$O(1)$	Yes
Bucket Sort	$O(n \log n)$	$O(n)$	Yes
Merge Sort	$O(n \log n)$	$O(n)$	Yes
Quick Sort	$O(n \log n)$	$O(n)$	No
Heap Sort	$O(n \log n)$	$O(1)$	No



**What is a graph? Why do we need it?**

**Graph Terminologies**

**Types of graphs. Graph Representation**

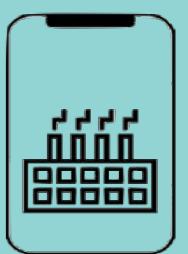
**Traversal of graphs. (BFS and DFS)**

**Topological Sorting**

**Single source shortest path (BFS, Dijkstra and Bellman Ford )**

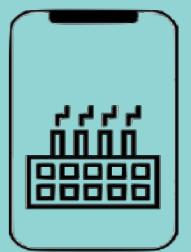
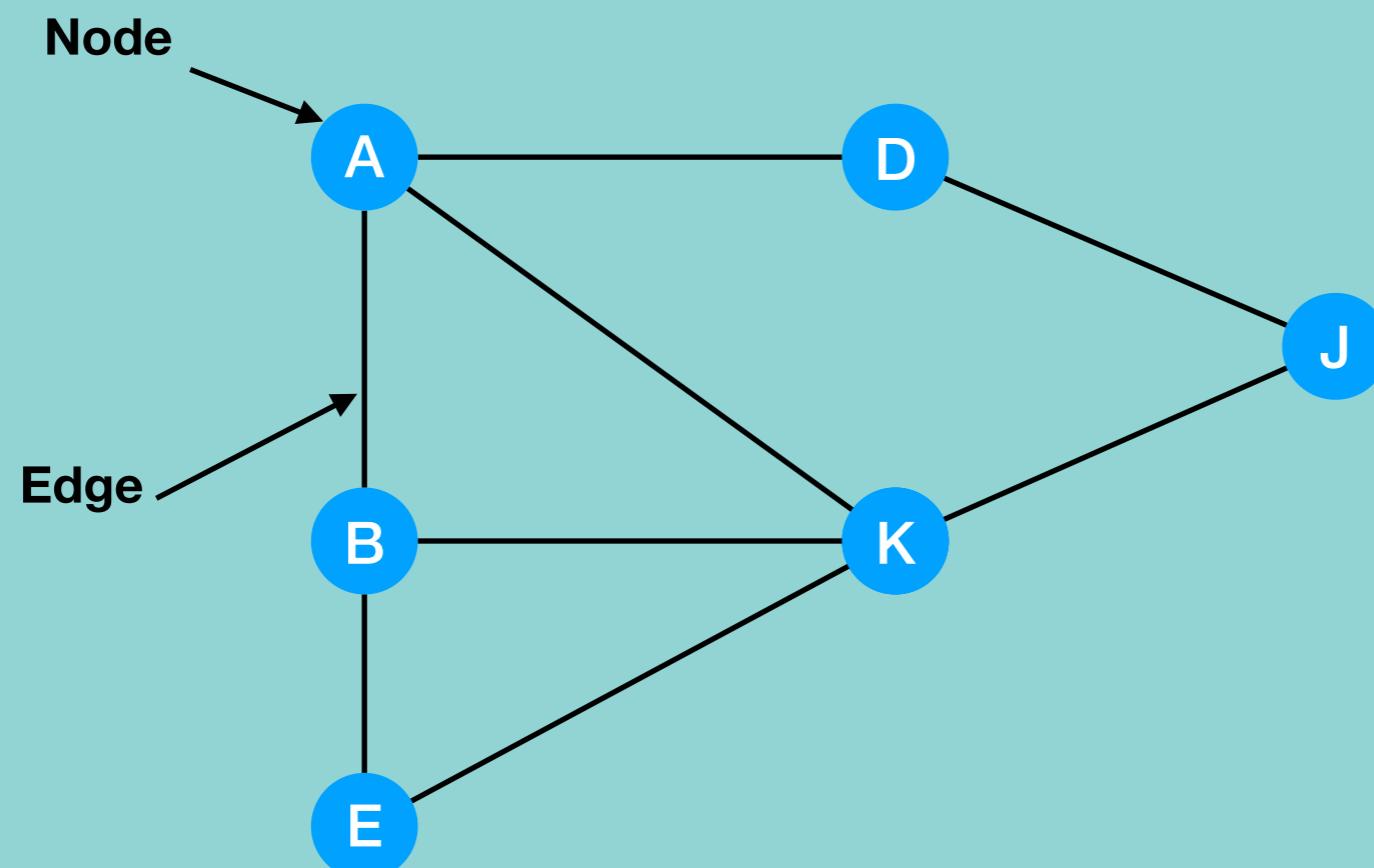
**All pairs shortest path (BFS, Dijkstra, Bellman Ford and Floyd Warshall algorithms)**

**Minimum Spanning Tree (Kruskal and Prim algorithms)**

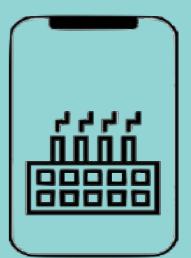
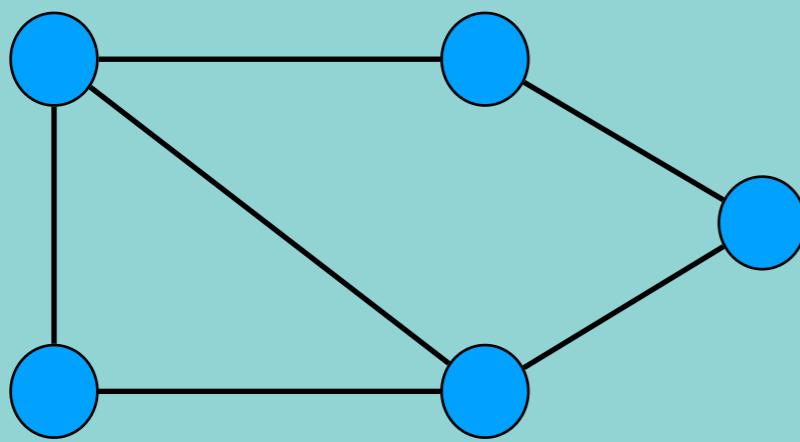
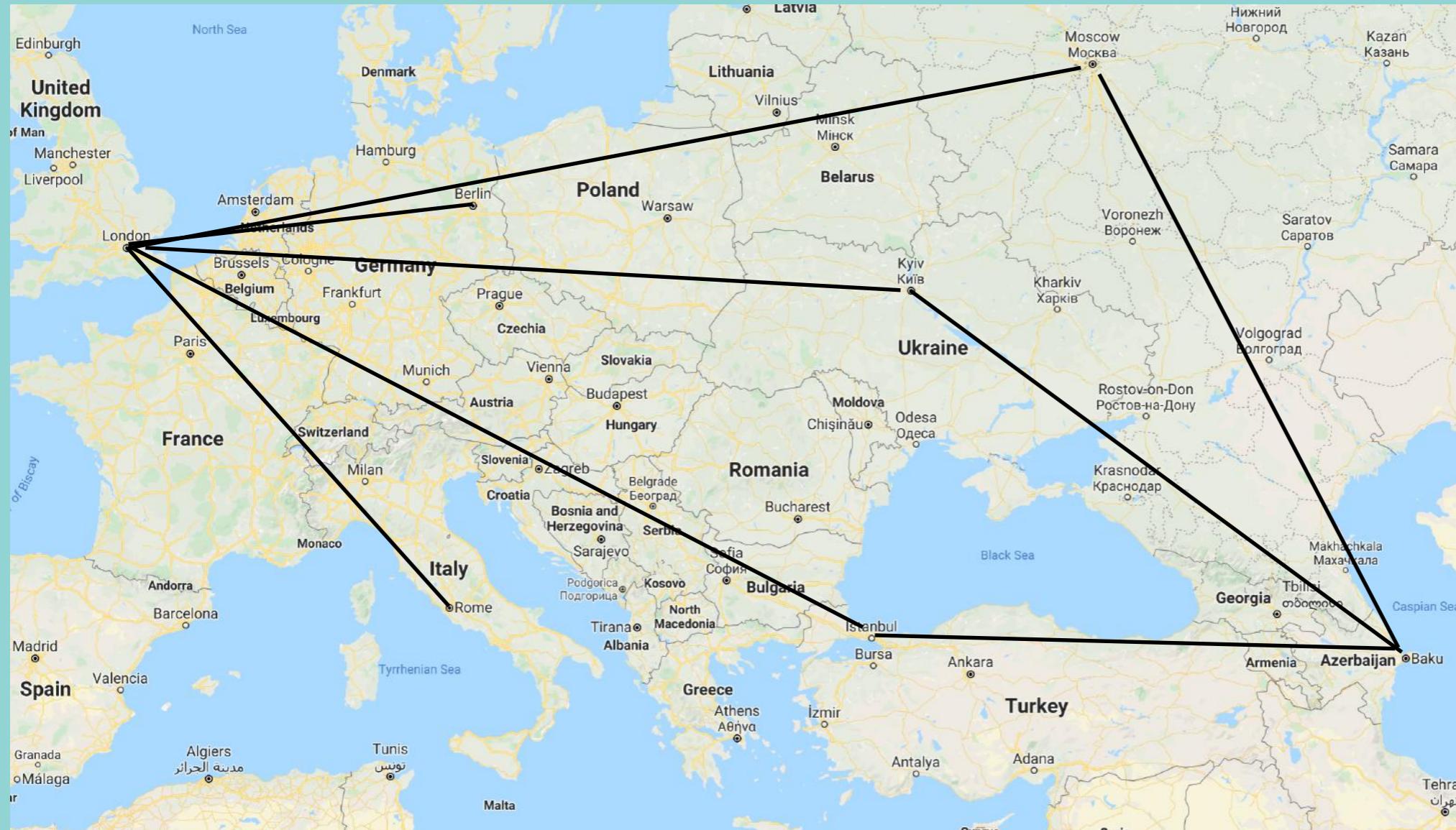


# What is Graph?

Graph consists of a finite set of Vertices(or nodes) and a set of Edges which connect a pair of nodes.

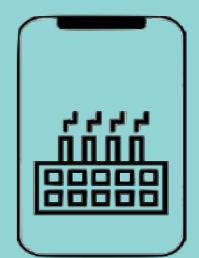
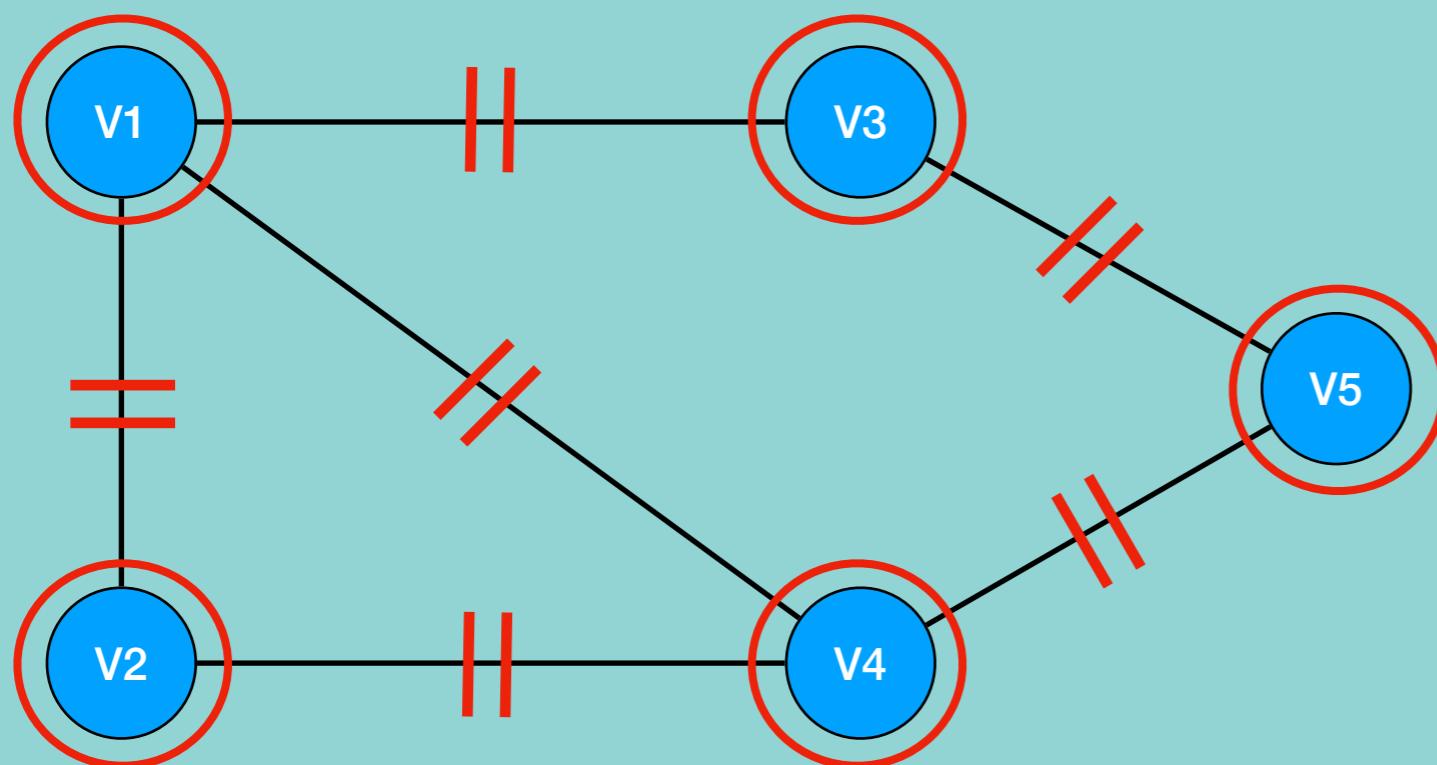


# Why Graph?



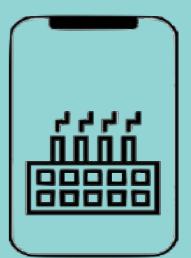
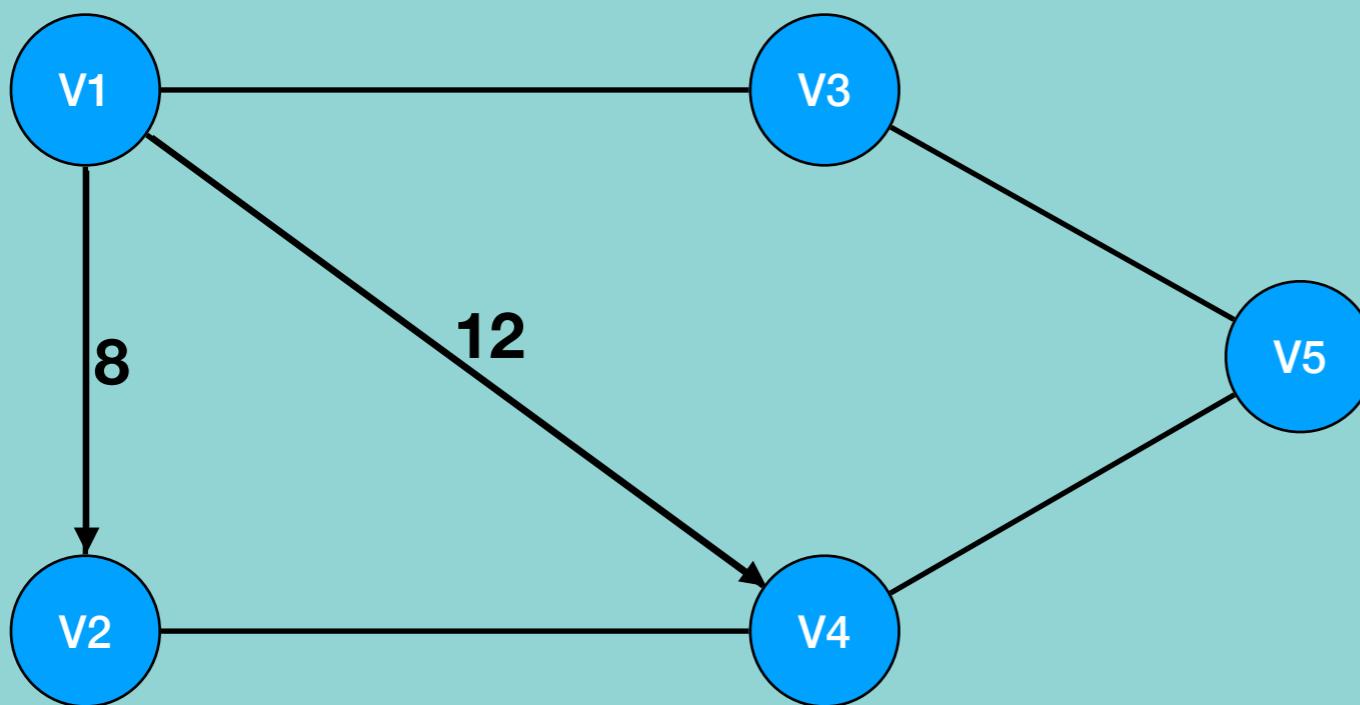
# Graph Terminology

- **Vertices (vertex)** : Vertices are the nodes of the graph
- **Edge** : The edge is the line that connects pairs of vertices



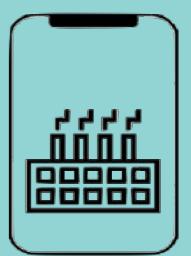
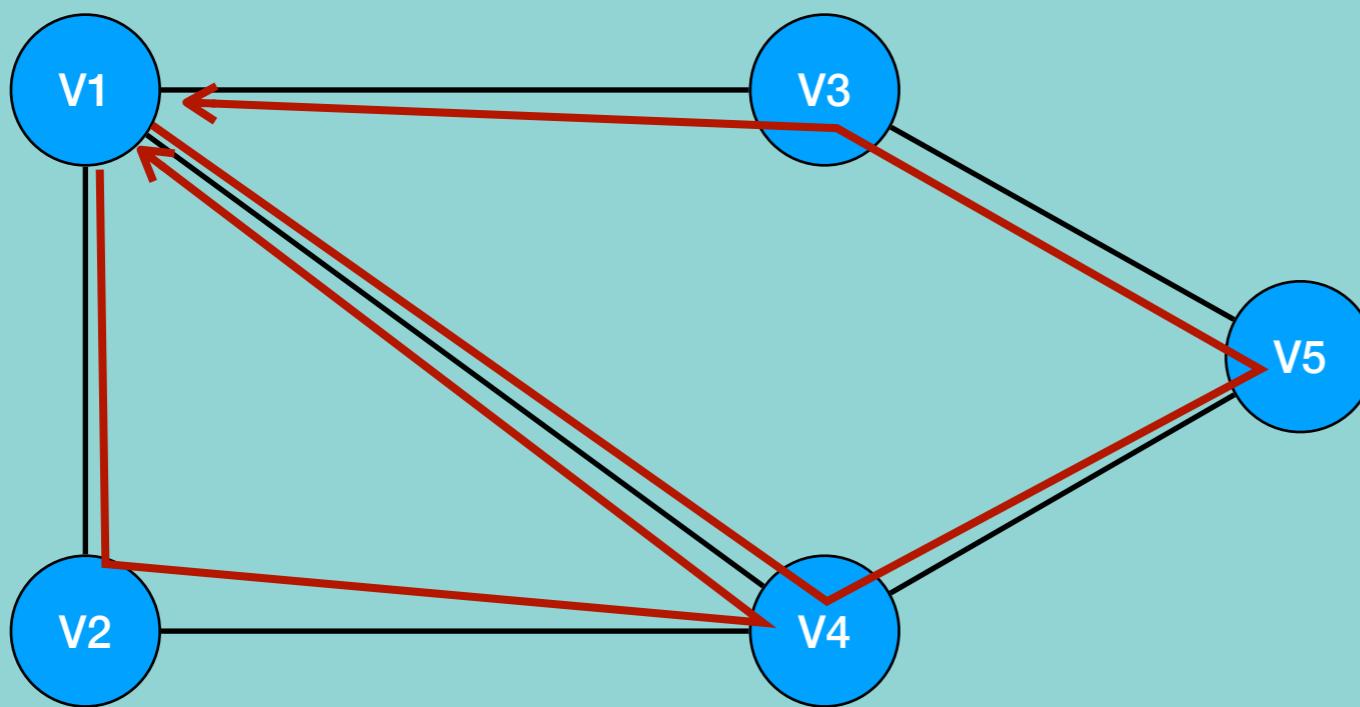
# Graph Terminology

- **Vertices** : Vertices are the nodes of the graph
- **Edge** : The edge is the line that connects pairs of vertices
- **Unweighted graph** : A graph which does not have a weight associated with any edge
- **Weighted graph** : A graph which has a weight associated with any edge
- **Undirected graph** : In case the edges of the graph do not have a direction associated with them
- **Directed graph** : If the edges in a graph have a direction associated with them



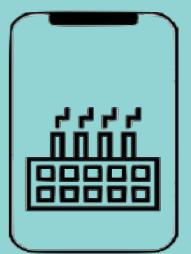
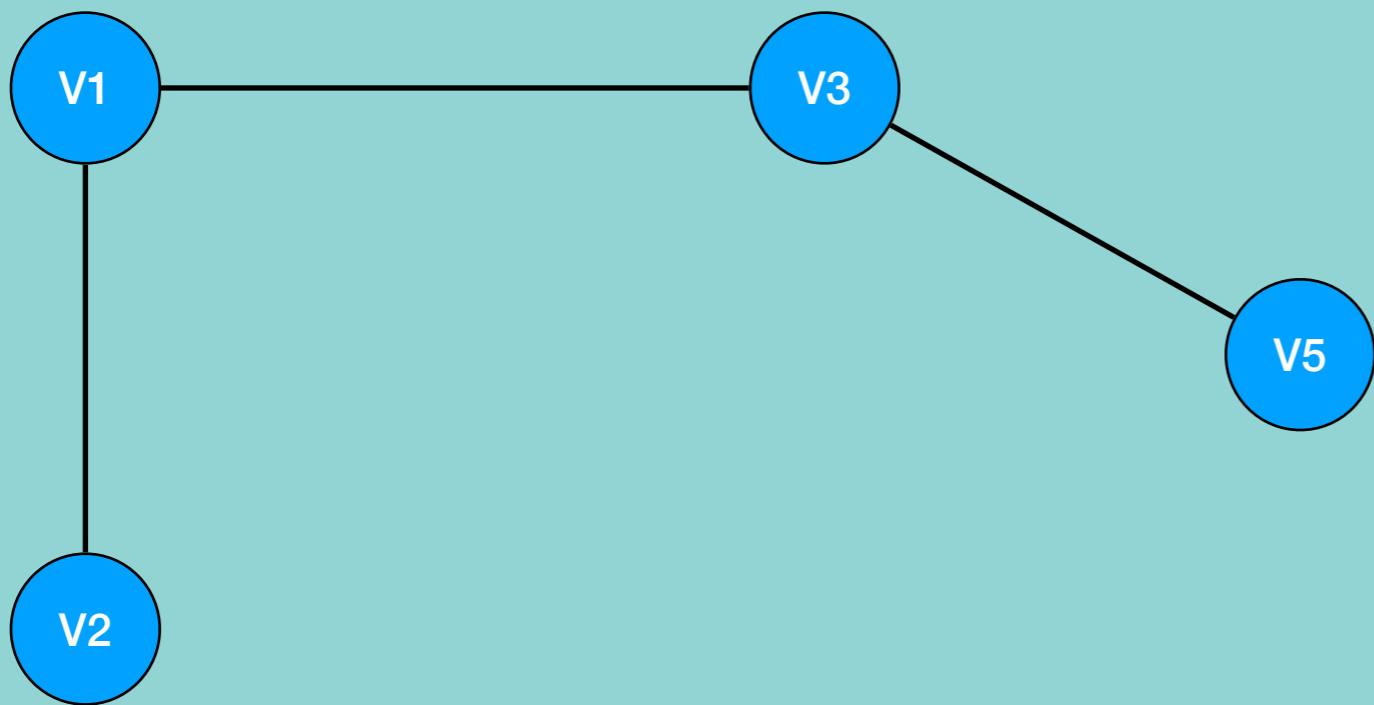
# Graph Terminology

- **Vertices** : Vertices are the nodes of the graph
- **Edge** : The edge is the line that connects pairs of vertices
- **Unweighted graph** : A graph which does not have a weight associated with any edge
- **Weighted graph** : A graph which has a weight associated with any edge
- **Undirected graph** : In case the edges of the graph do not have a direction associated with them
- **Directed graph** : If the edges in a graph have a direction associated with them
- **Cyclic graph** : A graph which has at least one loop



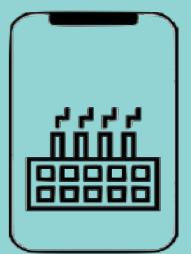
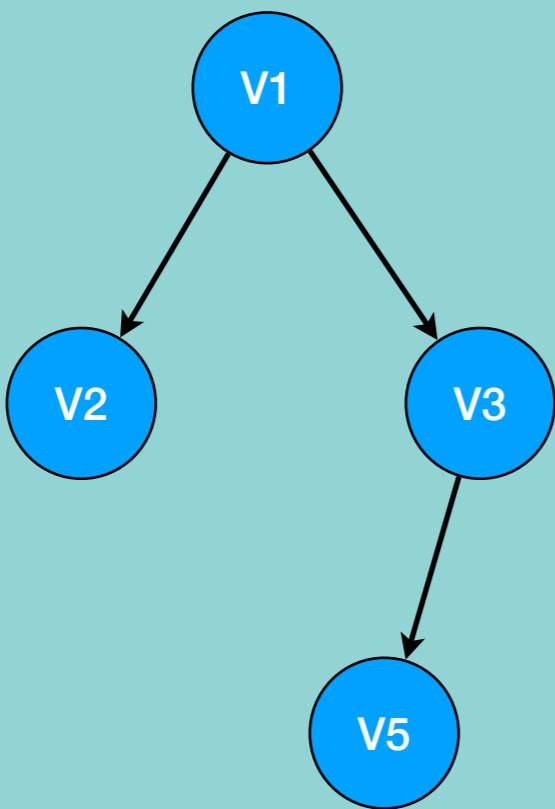
# Graph Terminology

- **Vertices** : Vertices are the nodes of the graph
- **Edge** : The edge is the line that connects pairs of vertices
- **Unweighted graph** : A graph which does not have a weight associated with any edge
- **Weighted graph** : A graph which has a weight associated with any edge
- **Undirected graph** : In case the edges of the graph do not have a direction associated with them
- **Directed graph** : If the edges in a graph have a direction associated with them
- **Cyclic graph** : A graph which has at least one loop
- **Acyclic graph** : A graph with no loop

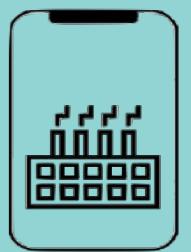
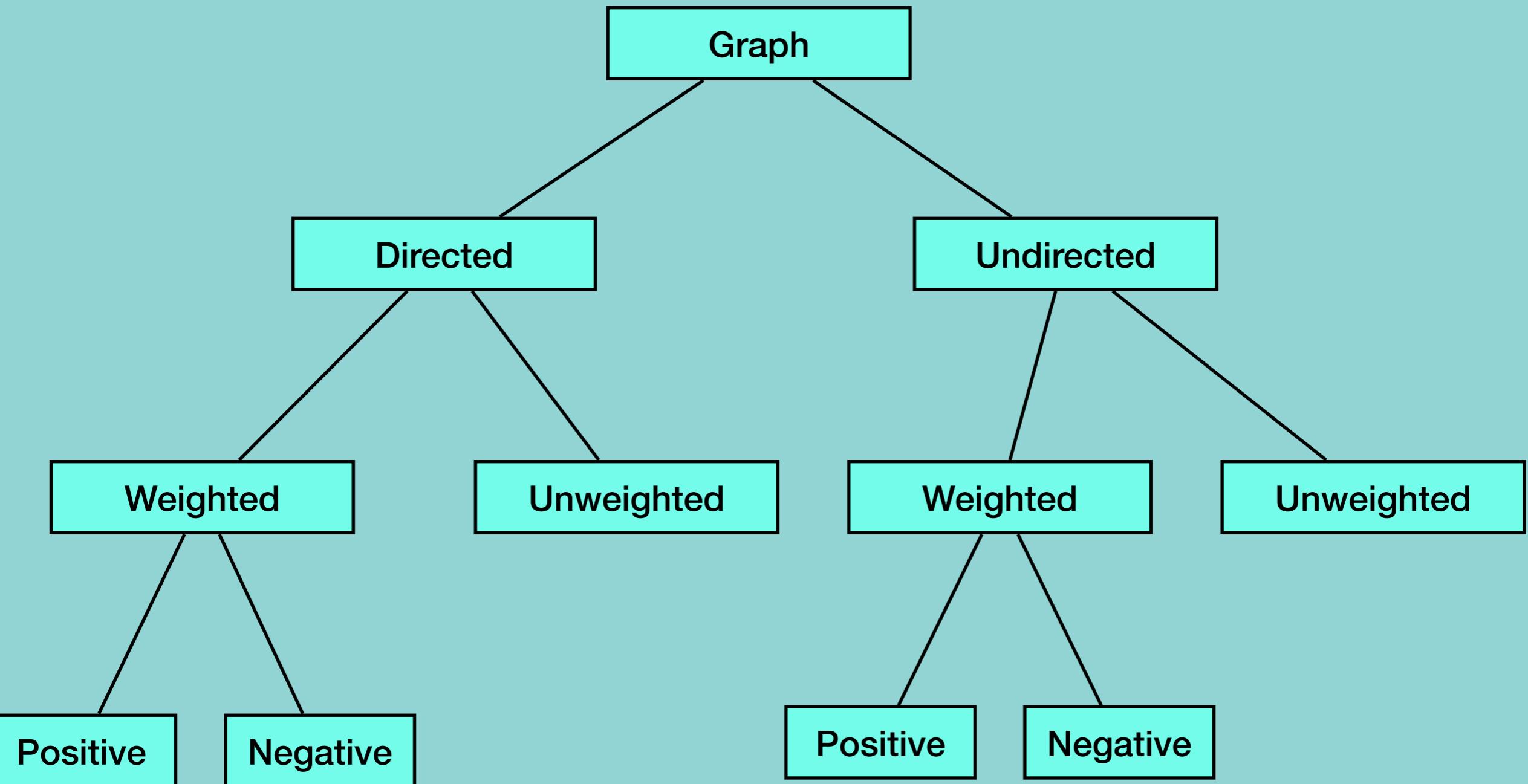


# Graph Terminology

- **Vertices** : Vertices are the nodes of the graph
- **Edge** : The edge is the line that connects pairs of vertices
- **Unweighted graph** : A graph which does not have a weight associated with any edge
- **Weighted graph** : A graph which has a weight associated with any edge
- **Undirected graph** : In case the edges of the graph do not have a direction associated with them
- **Directed graph** : If the edges in a graph have a direction associated with them
- **Cyclic graph** : A graph which has at least one loop
- **Acyclic graph** : A graph with no loop
- **Tree**: It is a special case of directed acyclic graphs

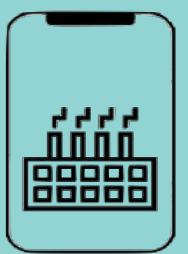
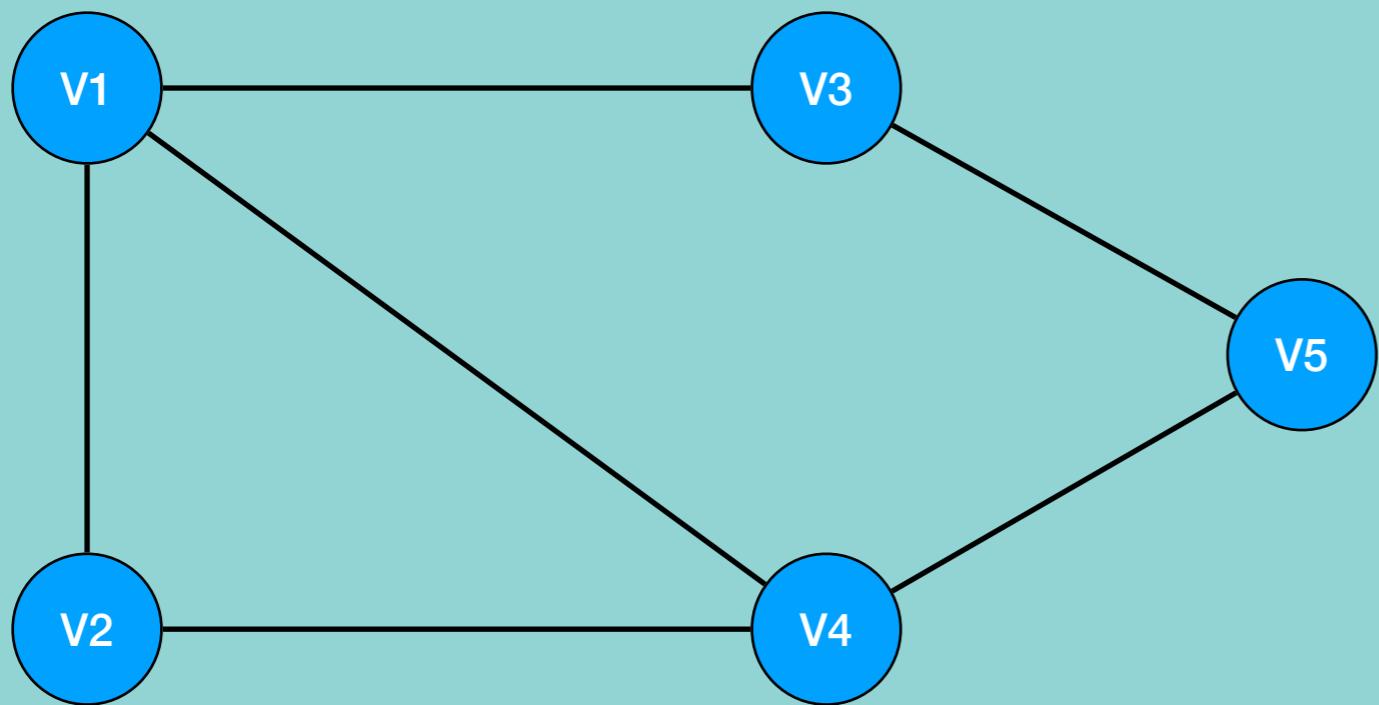


# Graph Types



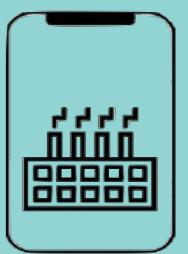
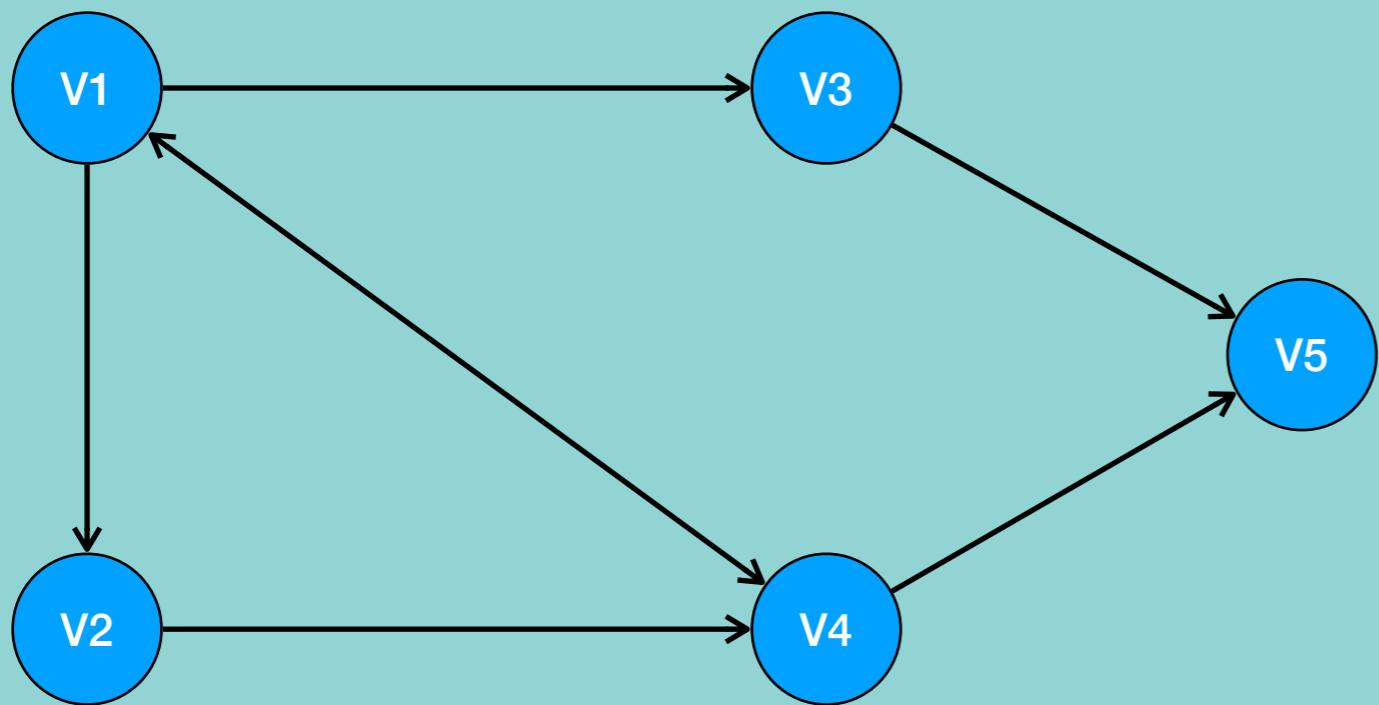
# Graph Types

## 1. Unweighted – undirected



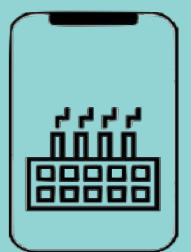
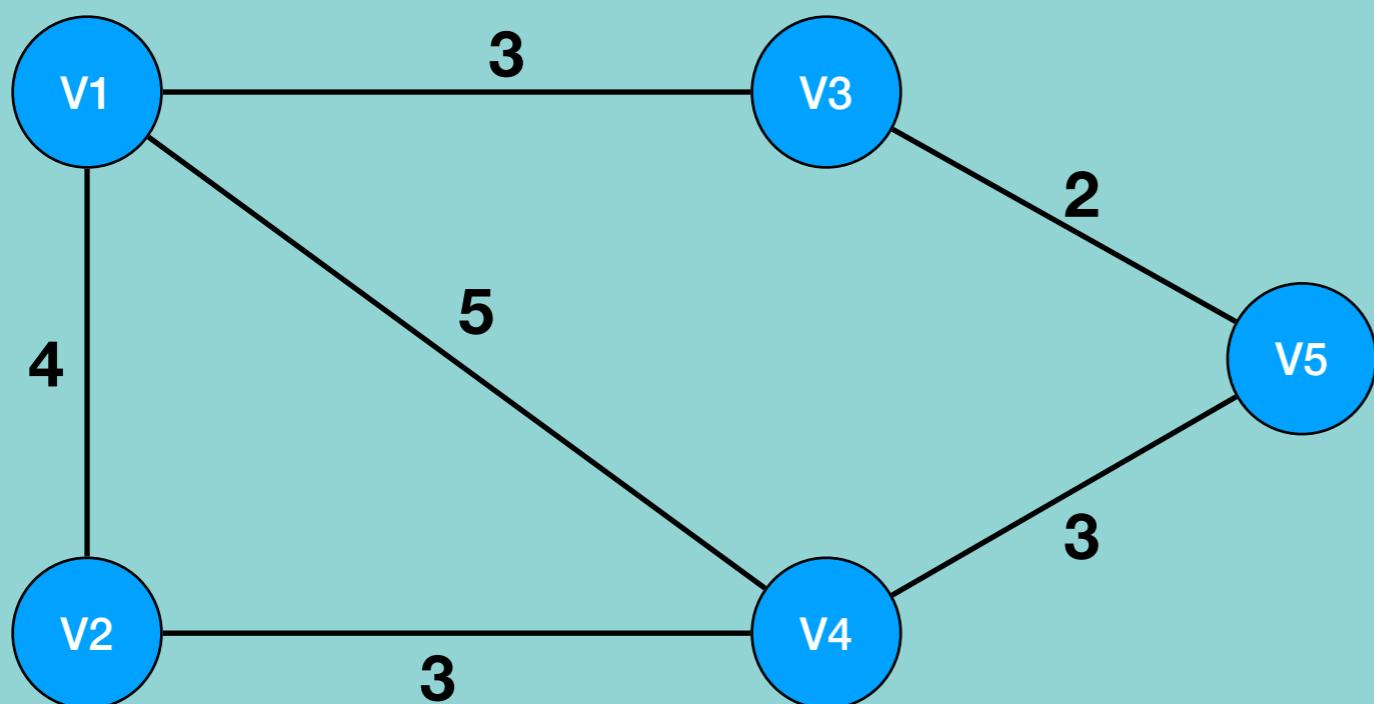
# Graph Types

1. Unweighted – undirected
2. Unweighted – directed



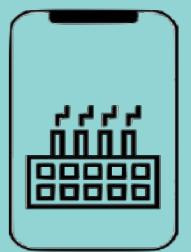
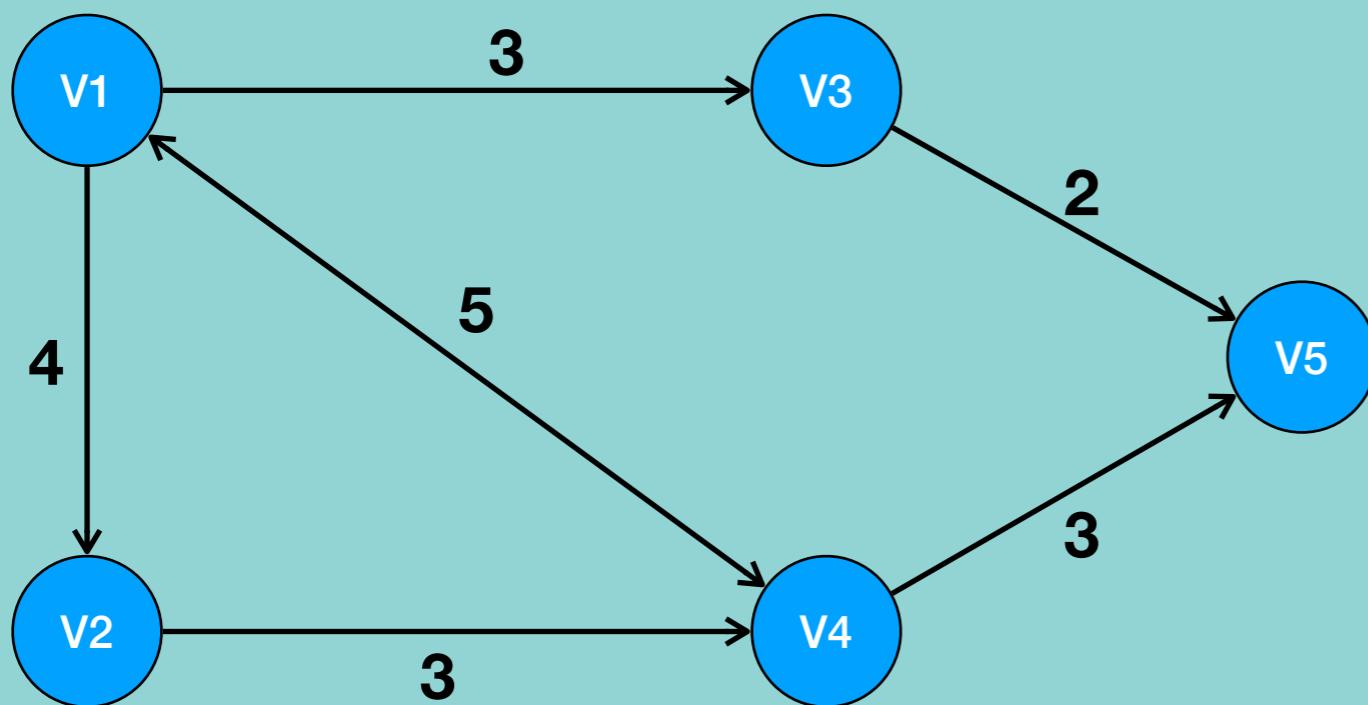
# Graph Types

1. Unweighted – undirected
2. Unweighted – directed
3. Positive – weighted – undirected



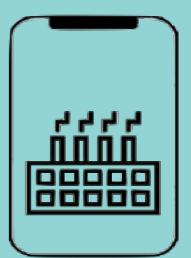
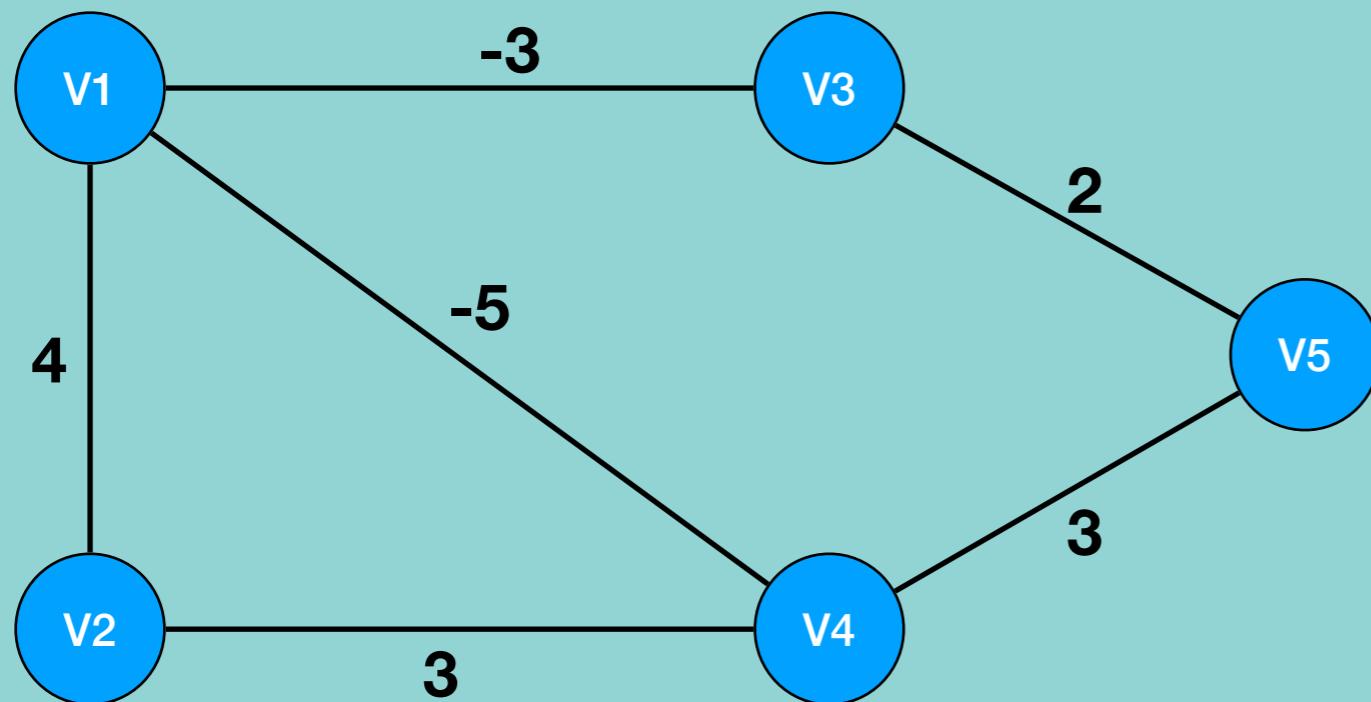
# Graph Types

1. Unweighted – undirected
2. Unweighted – directed
3. Positive – weighted – undirected
4. Positive – weighted – directed



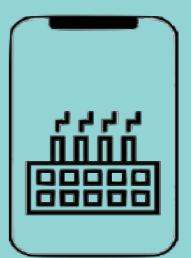
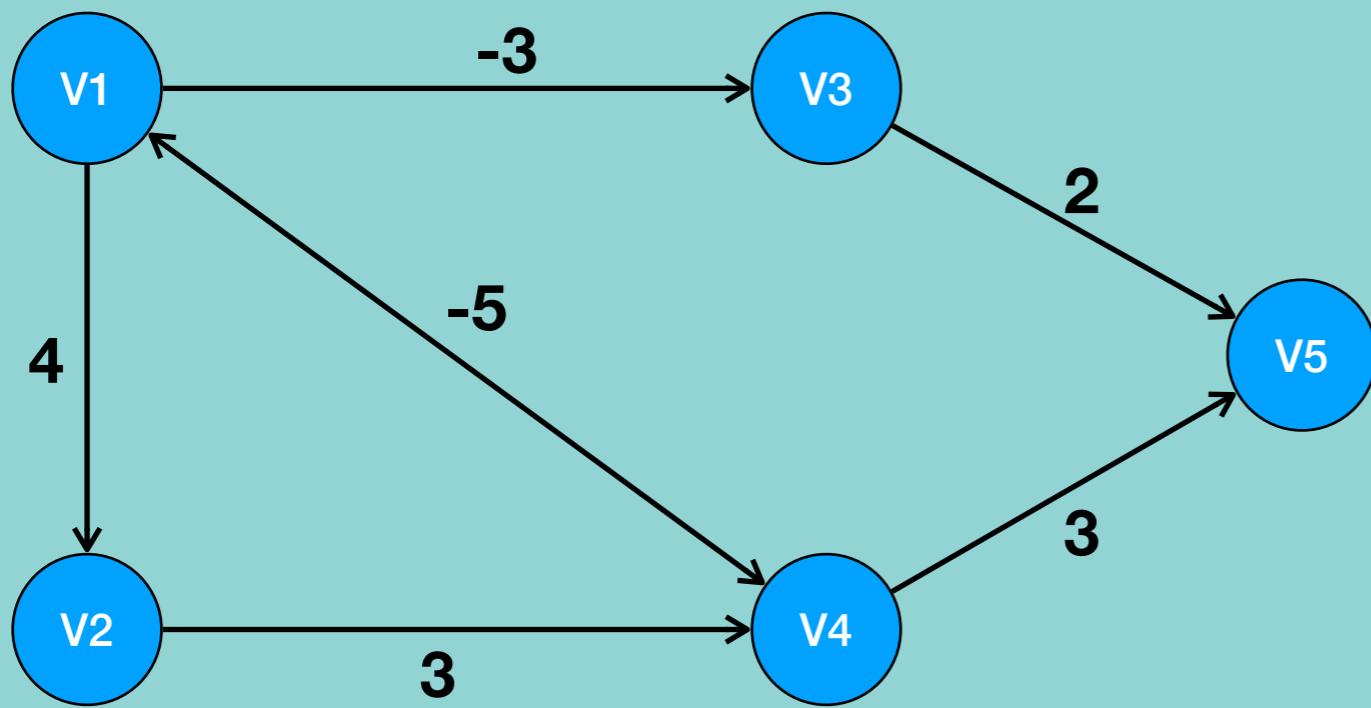
# Graph Types

1. Unweighted – undirected
2. Unweighted – directed
3. Positive – weighted – undirected
4. Positive – weighted – directed
5. Negative – weighted – undirected



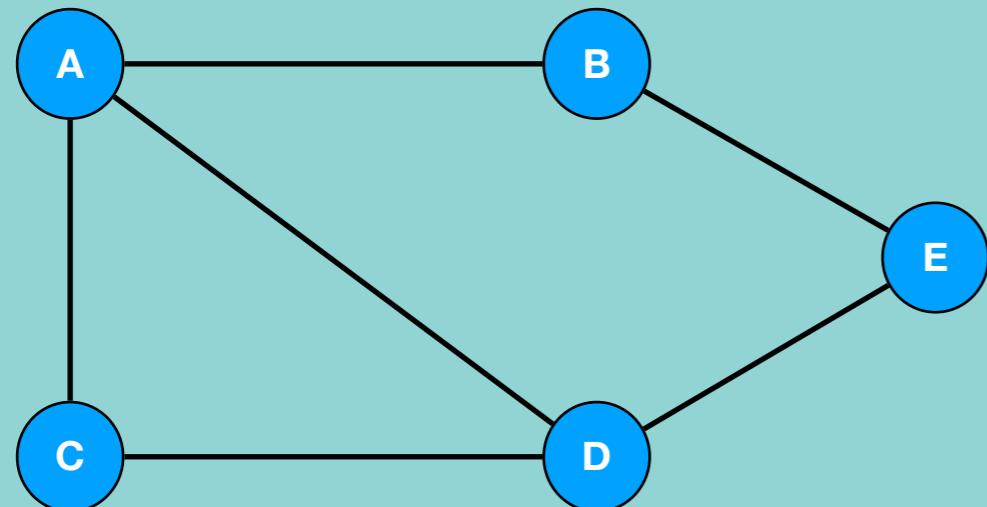
# Graph Types

1. Unweighted – undirected
2. Unweighted – directed
3. Positive – weighted – undirected
4. Positive – weighted – directed
5. Negative – weighted – undirected
6. Negative – weighted – directed

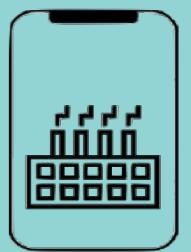


# Graph Representation

**Adjacency Matrix** : an adjacency matrix is a square matrix or you can say it is a 2D array. And the elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph

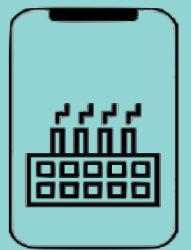
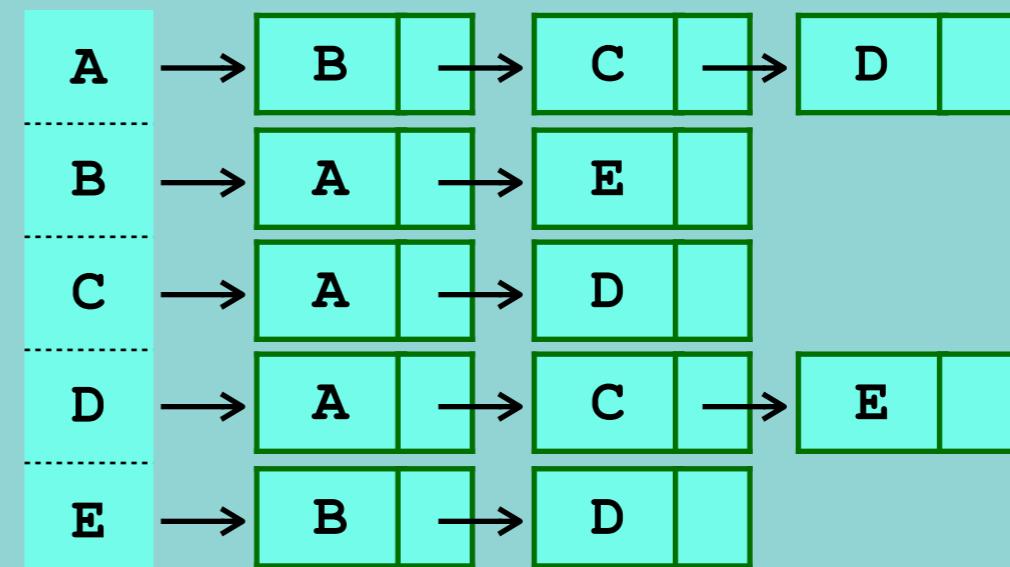
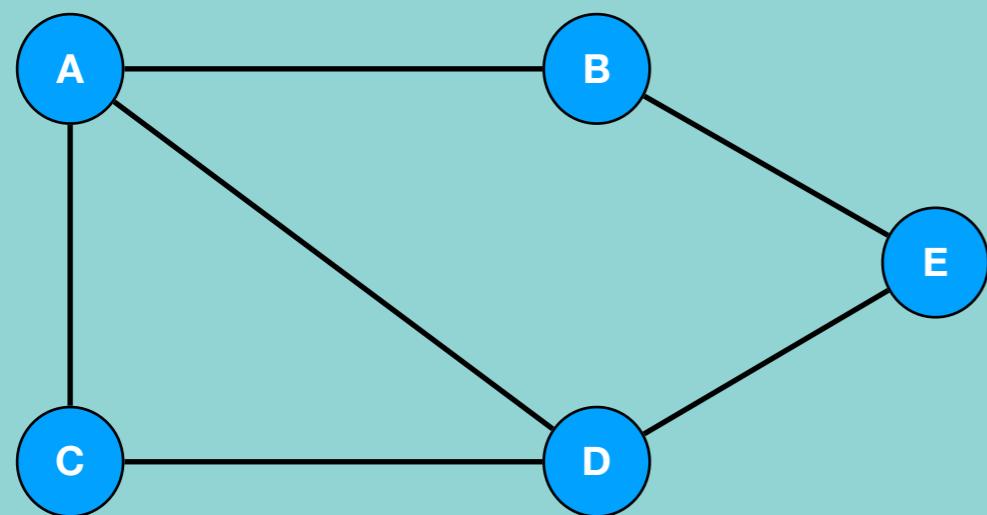


	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	0	1
C	1	0	0	1	0
D	1	0	1	0	1
E	0	1	0	1	0



# Graph Representation

**Adjacency List** : an adjacency list is a collection of unordered list used to represent a graph. Each list describes the set of neighbors of a vertex in the graph.

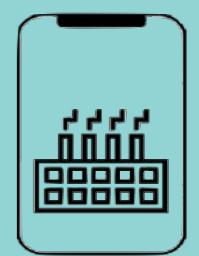
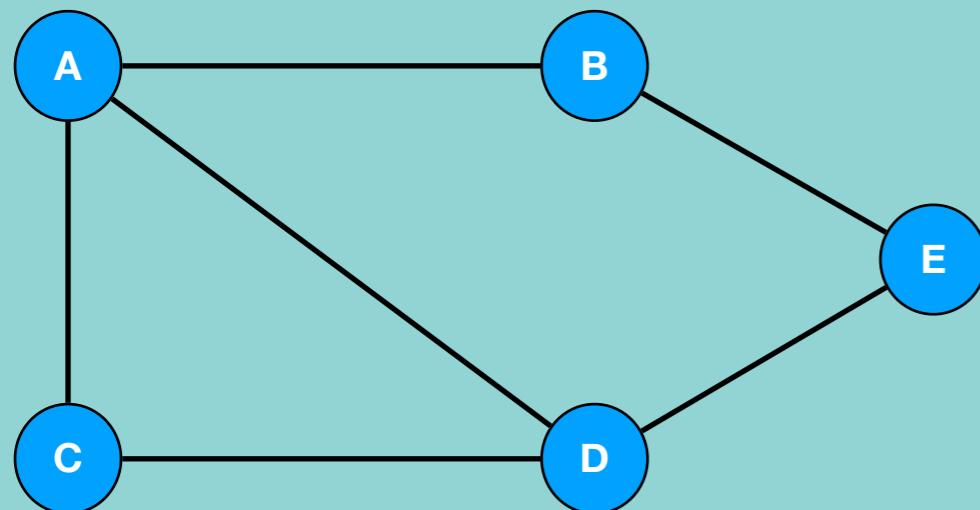
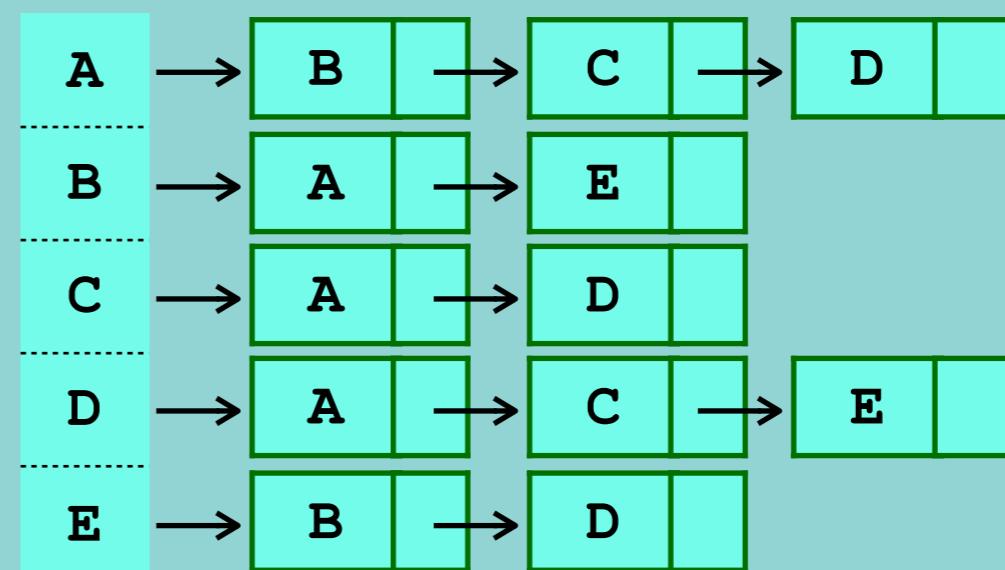


# Graph Representation

If a graph is complete or almost complete we should use **Adjacency Matrix**

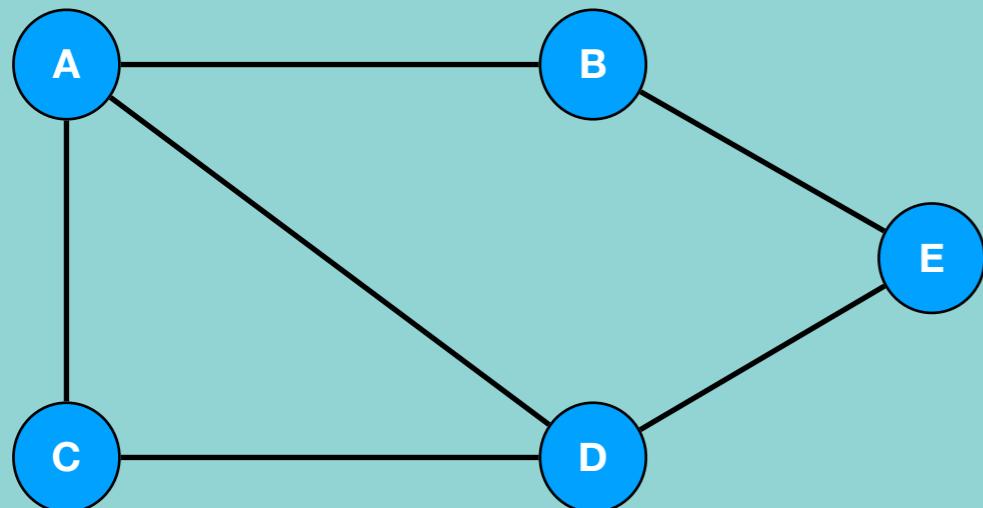
If the number of edges are few then we should use **Adjacency List**

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	0	1
C	1	0	0	1	0
D	1	0	1	0	1
E	0	1	0	1	0

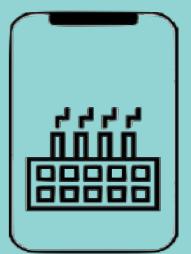


# Graph Representation

Python Dictionary implementation

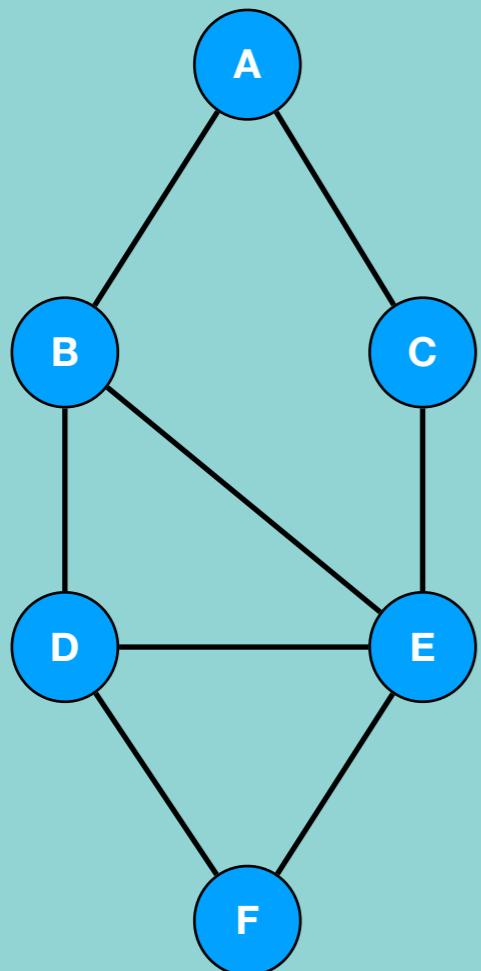


```
{ A : [B, C, D],  
  B: [A, E],  
  C: [A, D],  
  D: [A, C, E],  
  E: [B, D] }
```

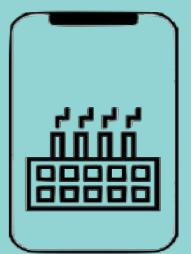


# Graph in Python

Dictionary implementation

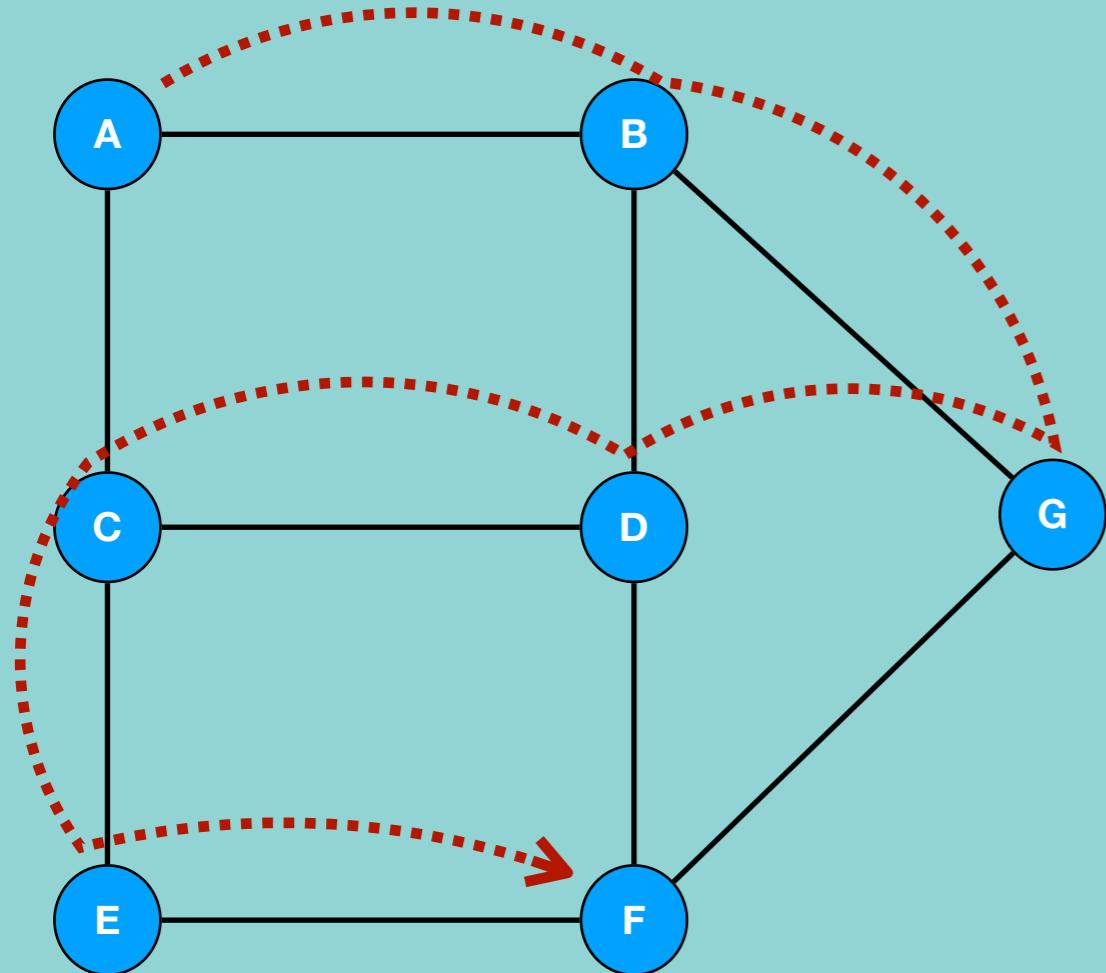


```
{ A : [B, C] ,  
      B: [A, D, E] ,  
      C: [A, E] ,  
      D: [B, E, F] ,  
      E: [C, D, F] ,  
      F: [D, E] }
```

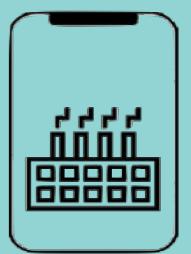


# Graph Traversal

It is a process of visiting all vertices in a given Graph

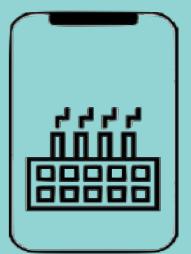
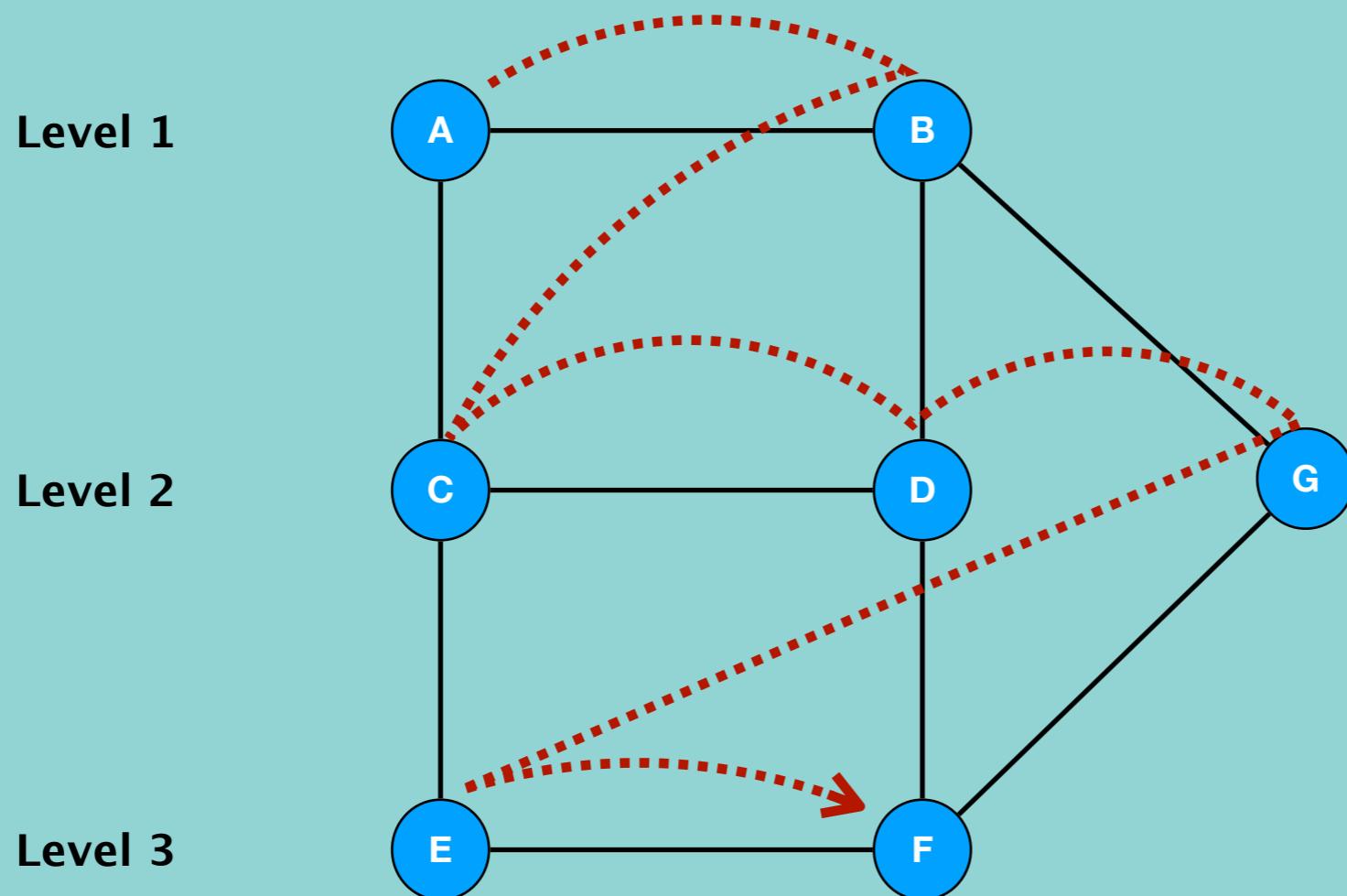


- Breadth First Search
- Depth First Search



# Breadth First Search

BFS is an algorithm for traversing Graph data structure. It starts at some arbitrary node of a graph and explores the neighbor nodes (which are at current level) first, before moving to the next level neighbors.



# Breadth First Search Algorithm

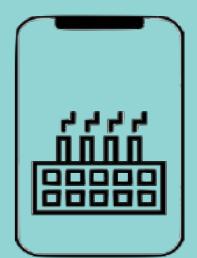
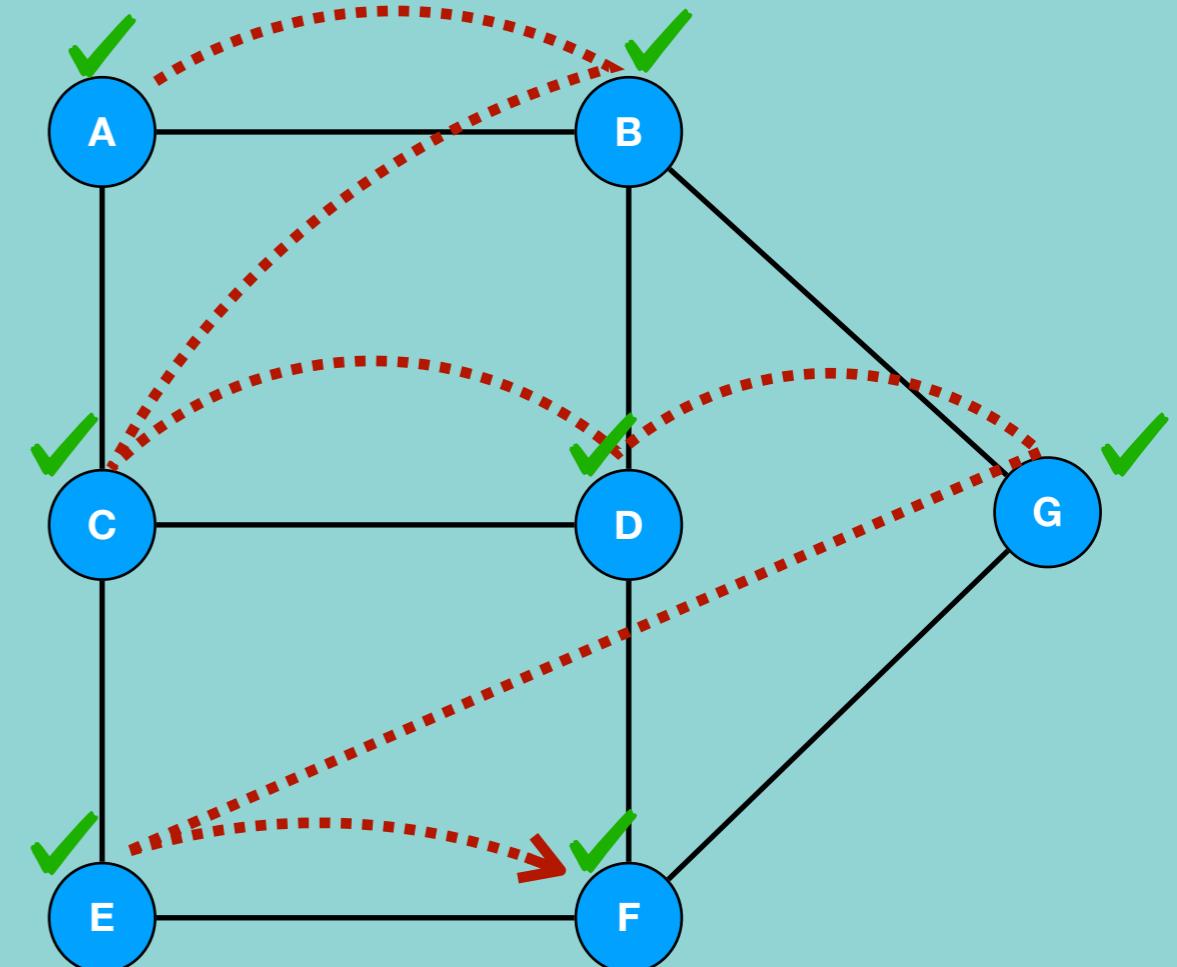
BFS

```
enqueue any starting vertex
while queue is not empty
    p = dequeue()
    if p is unvisited
        mark it visited
        enqueue all adjacent
        unvisited vertices of p
```

A B C D G E F

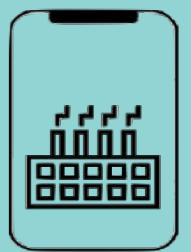
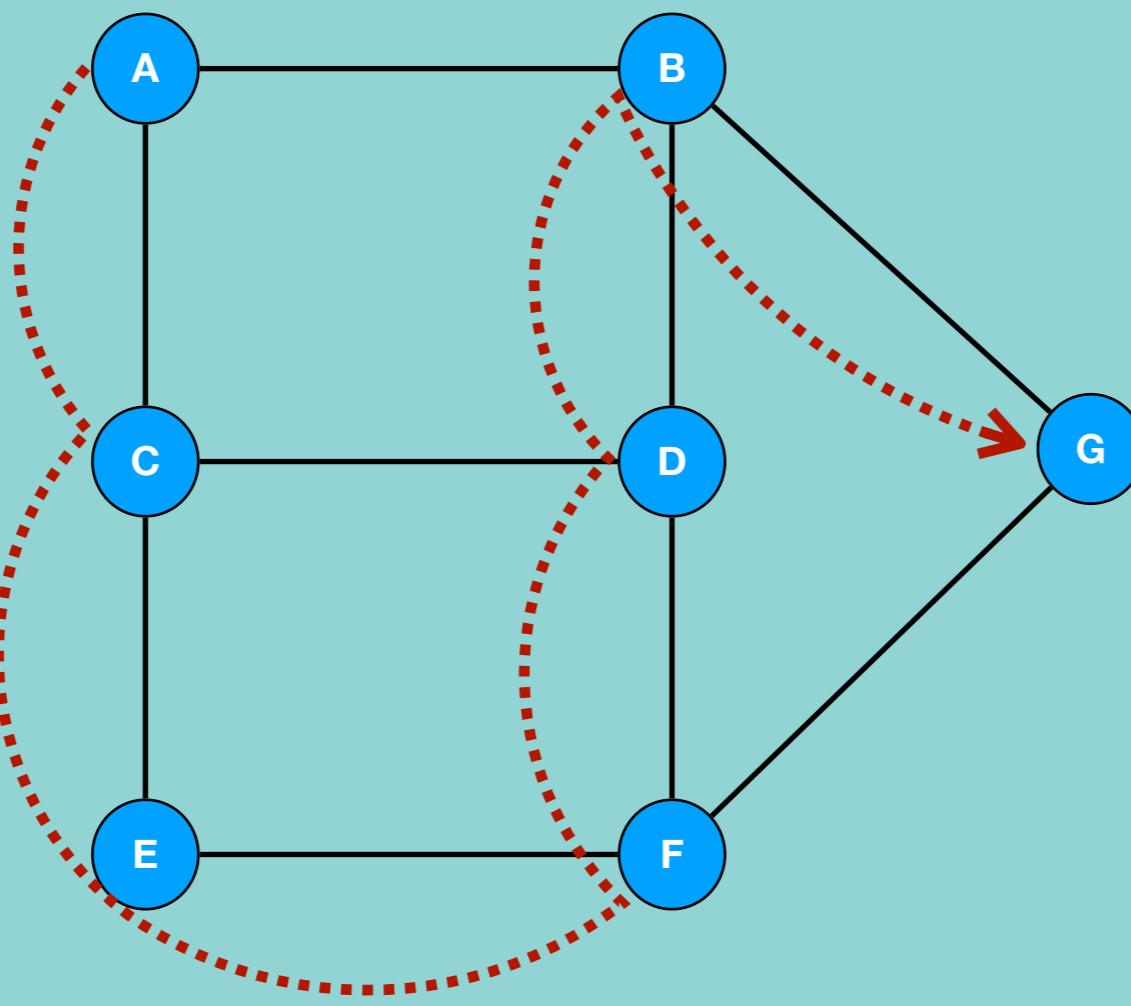
Queue

<del>A</del>	<del>B</del>	<del>C</del>	<del>D</del>	<del>G</del>	<del>E</del>	<del>F</del>	<del>F</del>	<del>F</del>
--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------



# Depth First Search

DFS is an algorithm for traversing a graph data structure which starts selecting some arbitrary node and explores as far as possible along each edge before backtracking



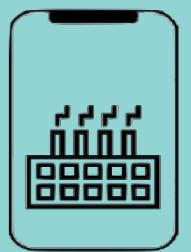
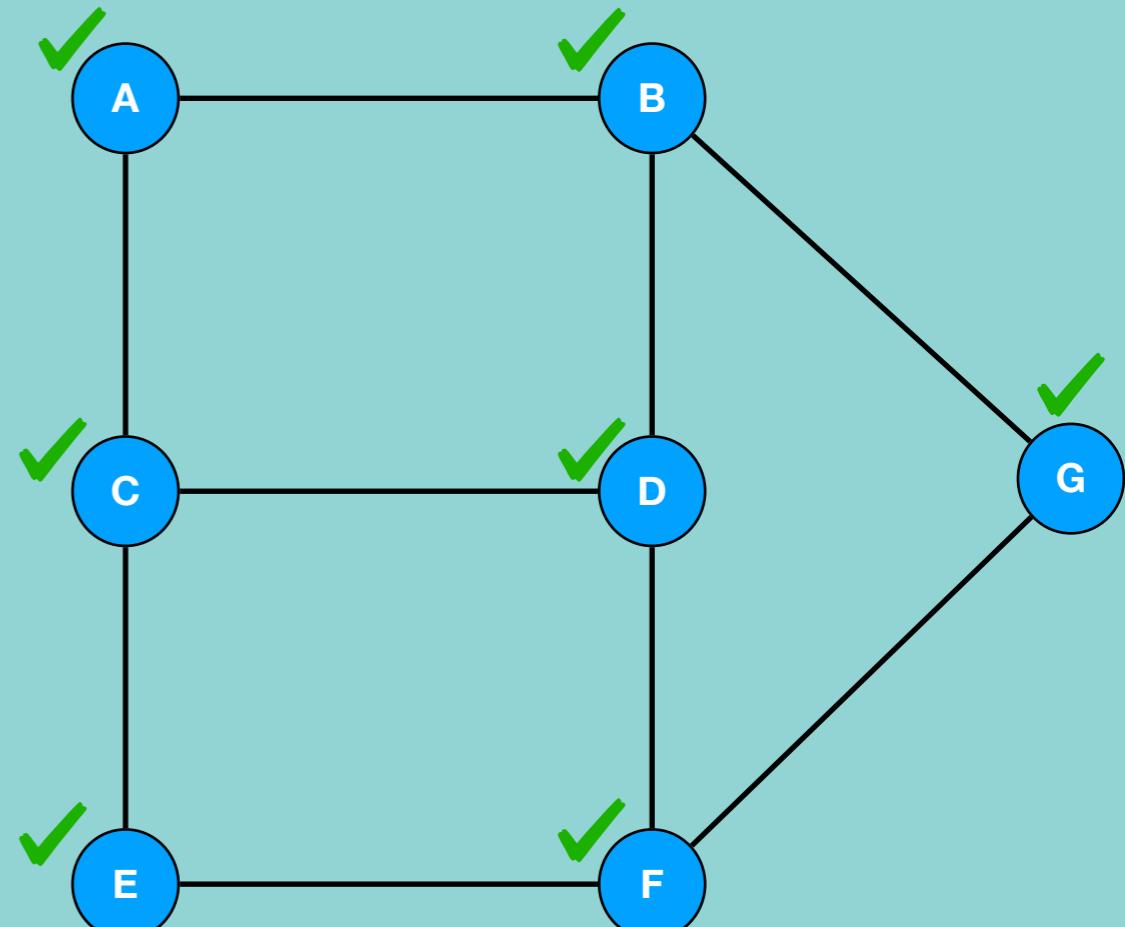
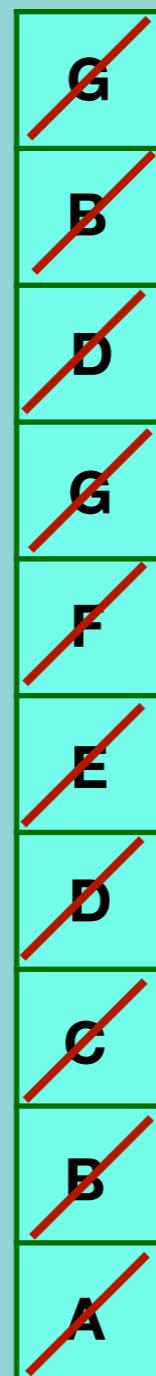
# Depth First Search Algorithm

DFS

```
push any starting vertex
while stack is not empty
    p = pop()
    if p is unvisited
        mark it visited
        Push all adjacent
        unvisited vertices of p
```

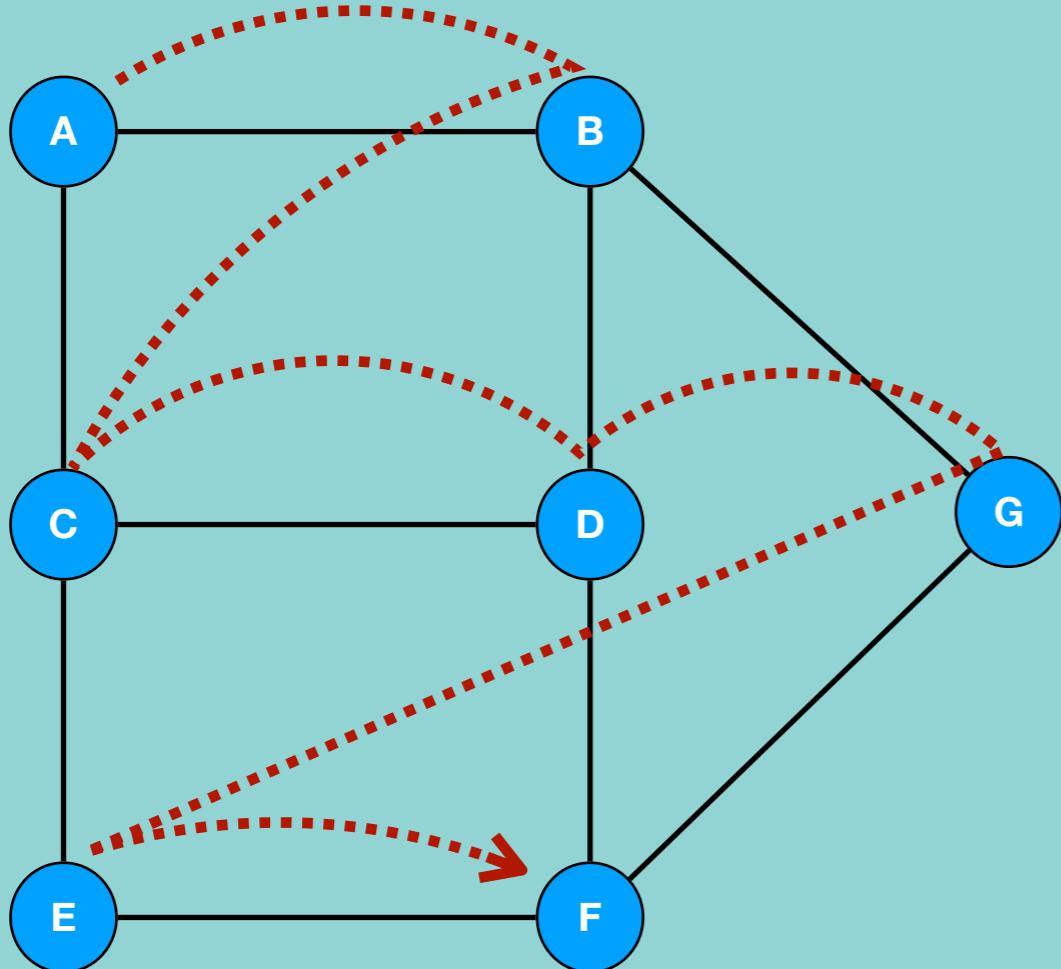
A C E F D B G

Stack

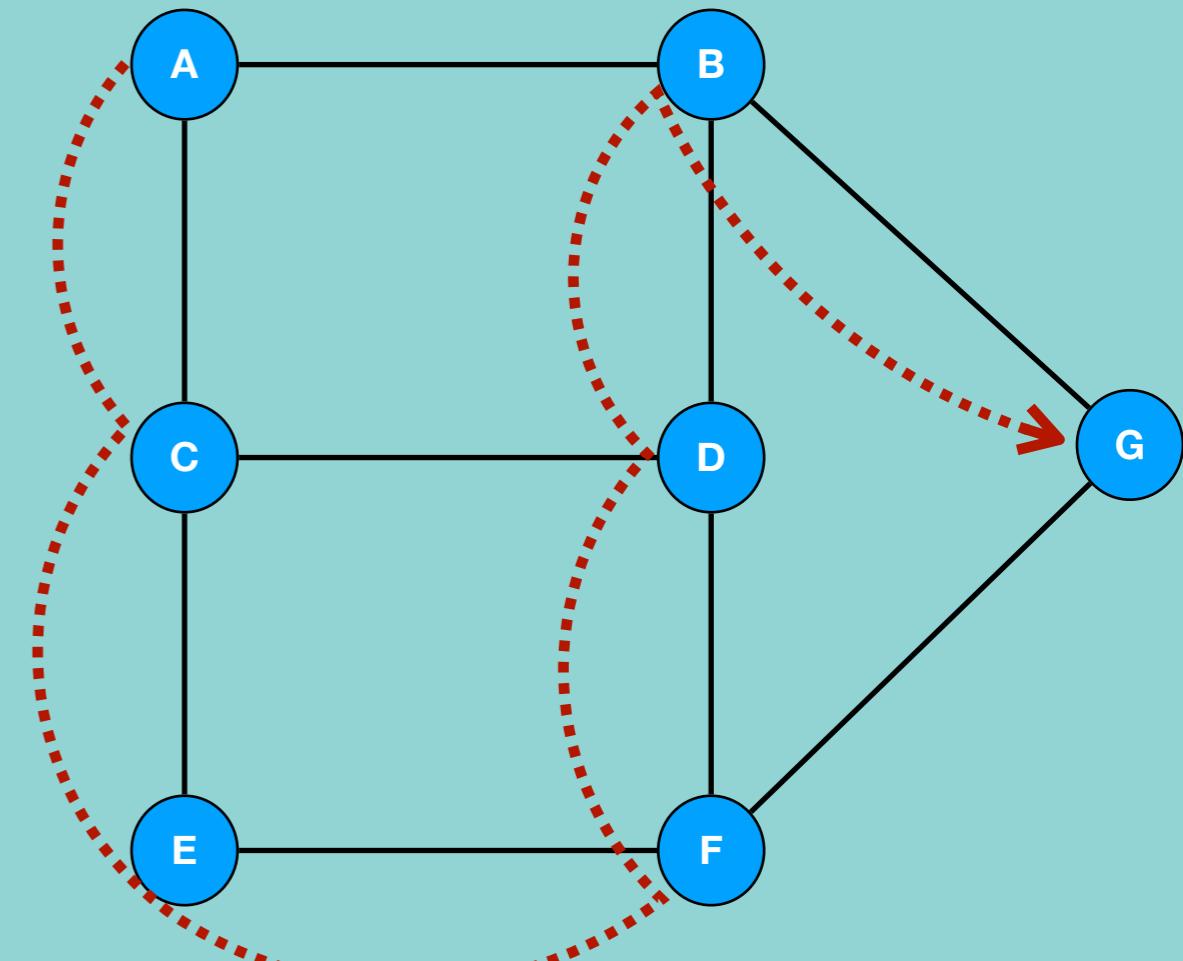


# BFS vs DFS

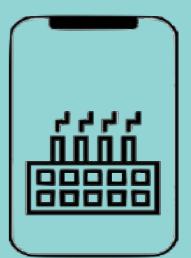
Level 1



**BFS**

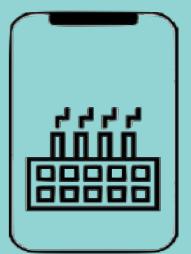


**DFS**



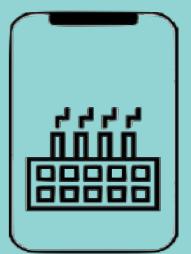
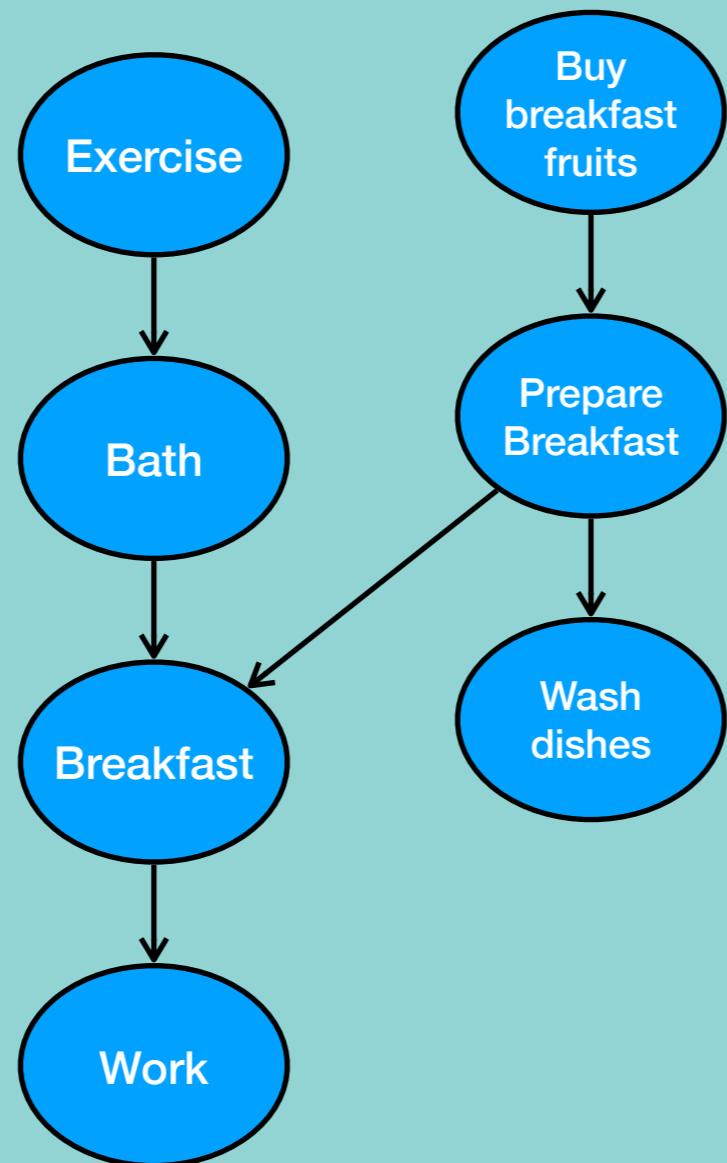
# BFS vs DFS

	BFS	DFS
How does it work internally?	It goes in breath first	It goes in depth first
Which DS does it use internally?	Queue	Stack
Time Complexity	$O(V+E)$	$O(V+E)$
Space Complexity	$O(V+E)$	$O(V+E)$
When to use?	If we know that the target is close to the starting point	If we already know that the target vertex is buried very deep



# Topological Sort

**Topological Sort:** Sorts given actions in such a way that if there is a dependency of one action on another, then the dependent action always comes later than its parent action.



# Topological Sort Algorithm

```
if a vertex depends on currentVertex:  
    Go to that vertex and  
    then come back to currentVertex  
else  
    Push currentVertex to Stack
```

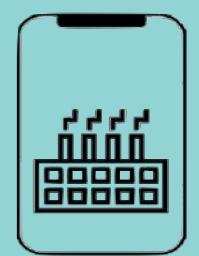
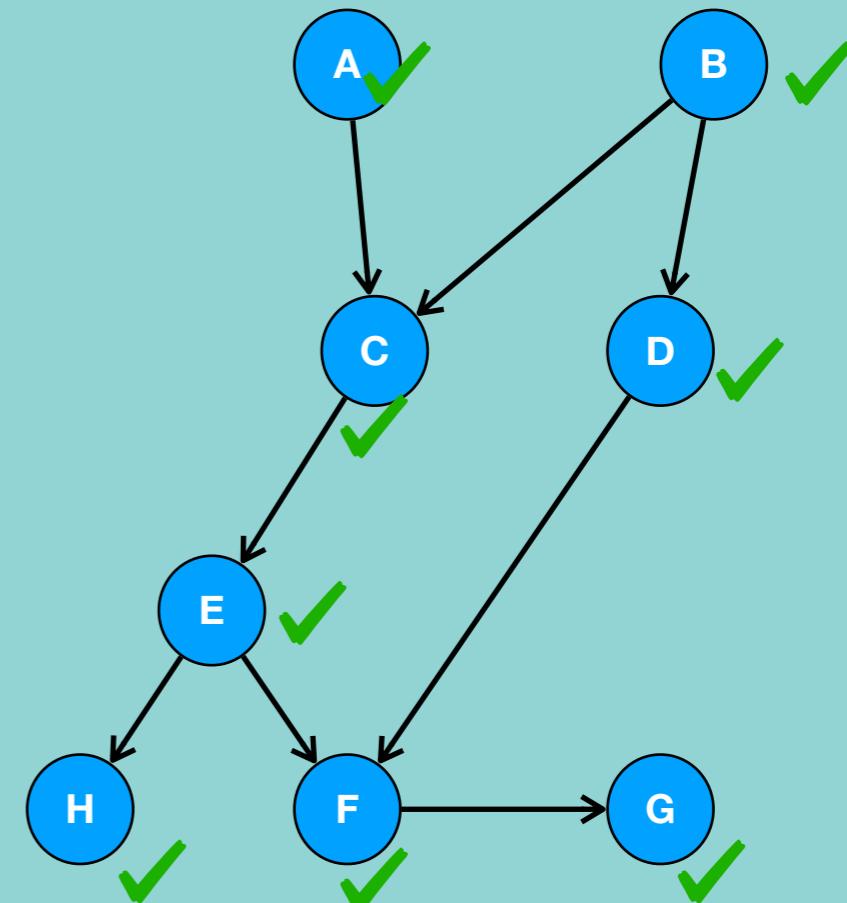
A B C D E H F G

B A C D E H F G

B D A C E F G H

Stack

B
D
A
C
E
F
G
H

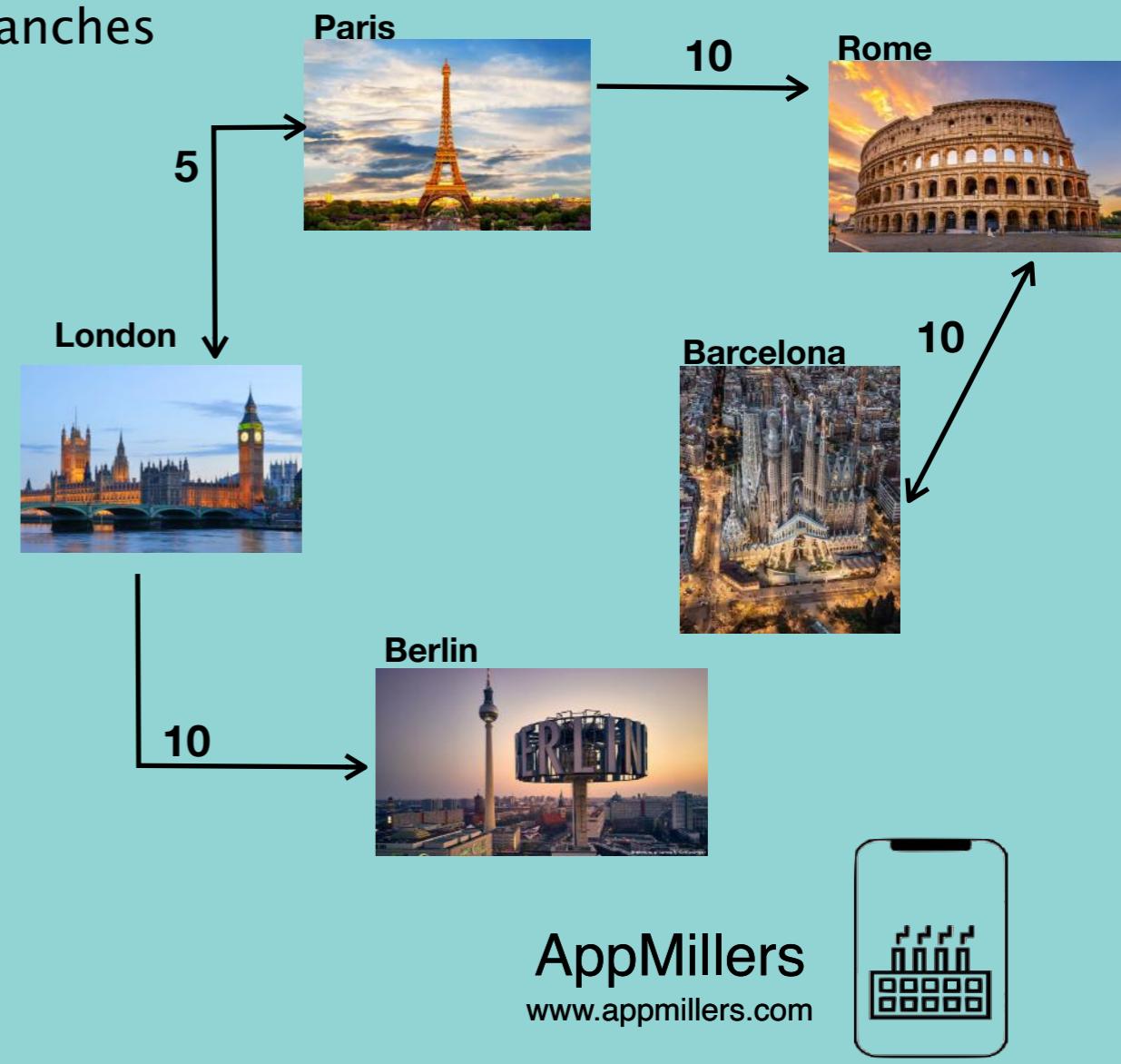
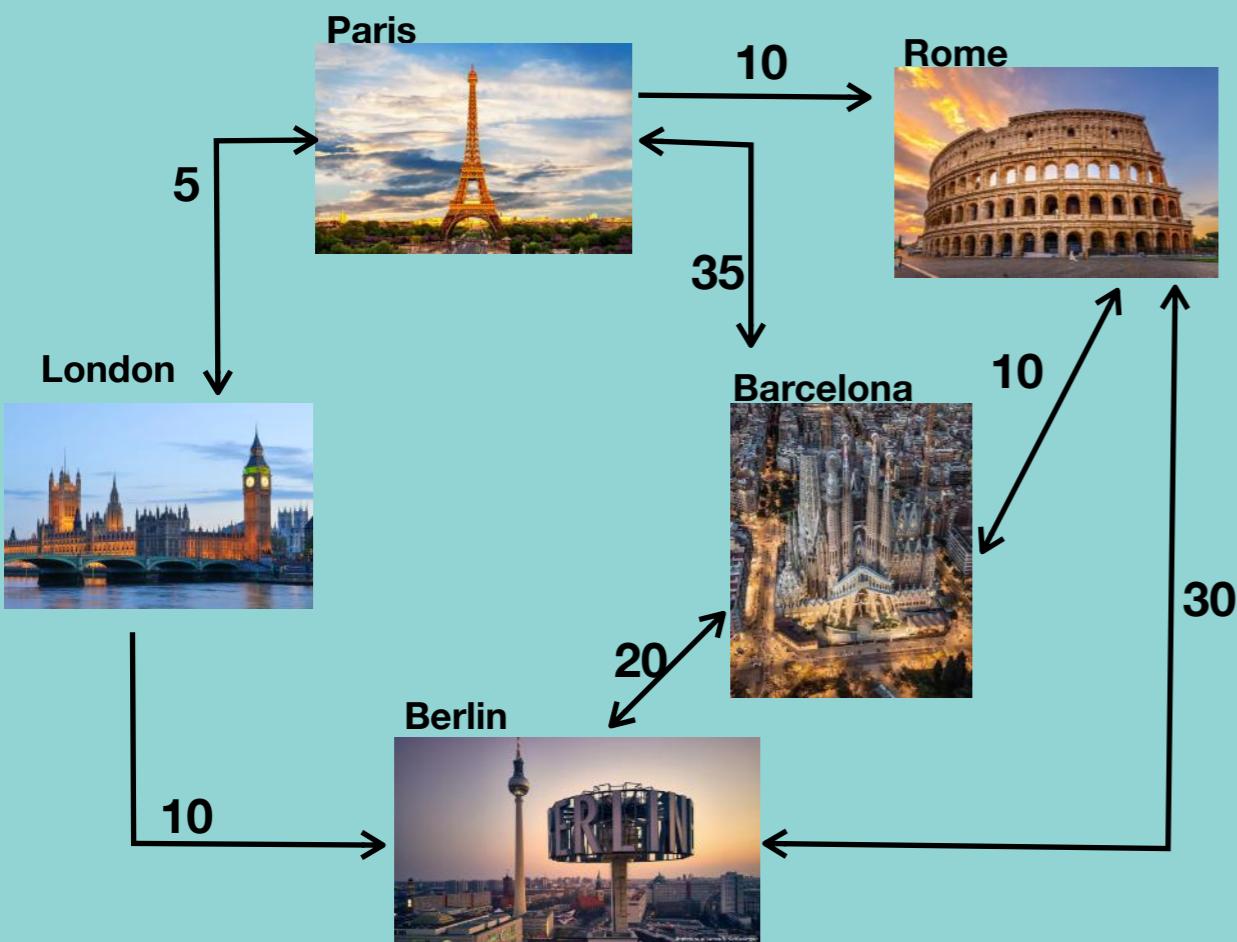


# Single Source Shortest Path Problem

A single source problem is about finding a path between a given vertex (called source) to all other vertices in a graph such that the total distance between them (source and destination) is minimum.

The problem:

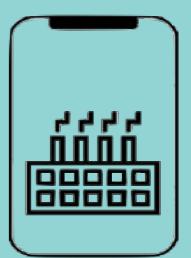
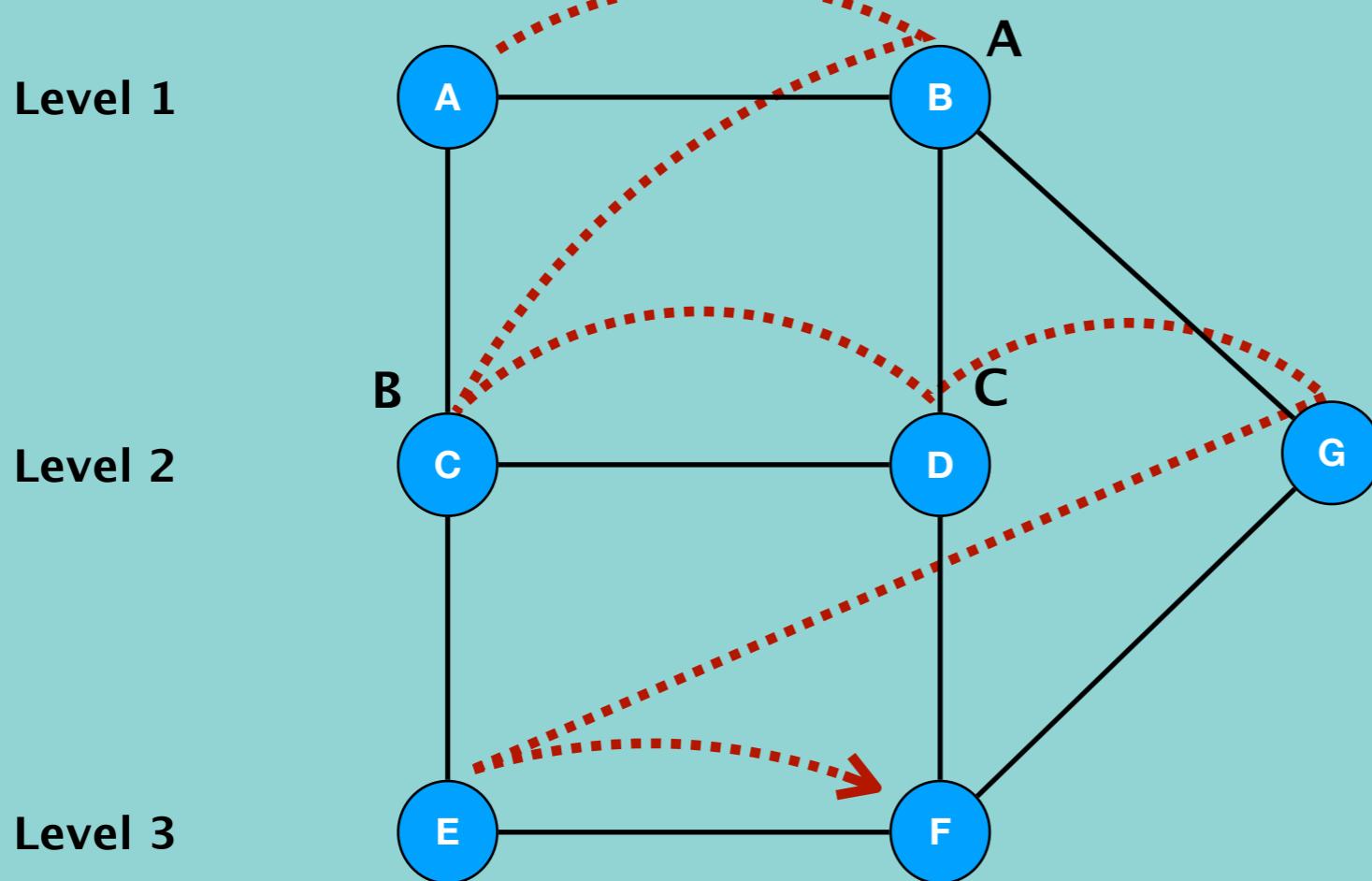
- Five offices in different cities.
- Travel costs between these cities are known.
- Find the cheapest way from head office to branches in different cities



# Single Source Shortest Path Problem

- BFS
- Dijkstra's Algorithm
- Bellman Ford

BFS – Breadth First Search



# BFS for SSSP

BFS

enqueue any starting vertex

while queue is not empty

  p = dequeue()

  if p is unvisited

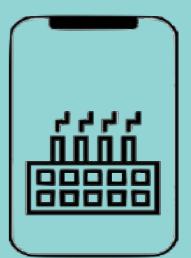
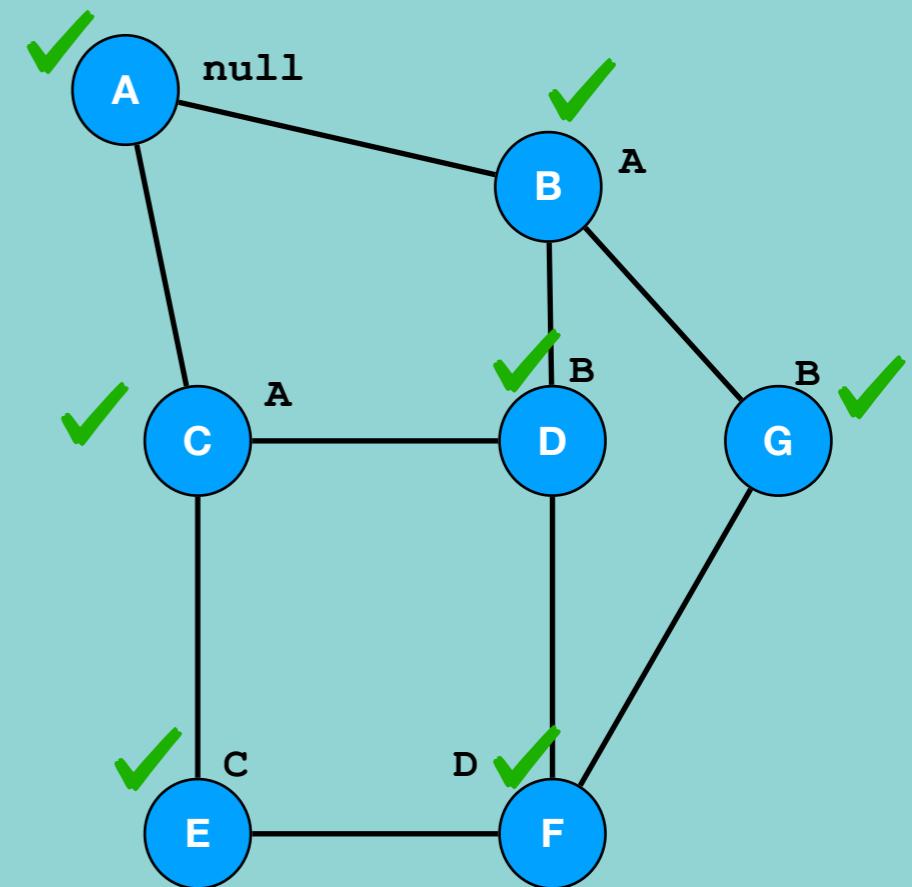
    mark it visited

    enqueue all adjacent unvisited vertices of p

    update parent of adjacent vertices to curVertex

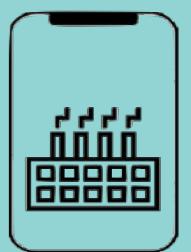
A B C D G E F

Queue

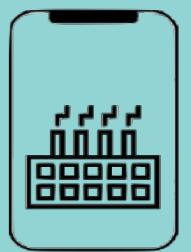
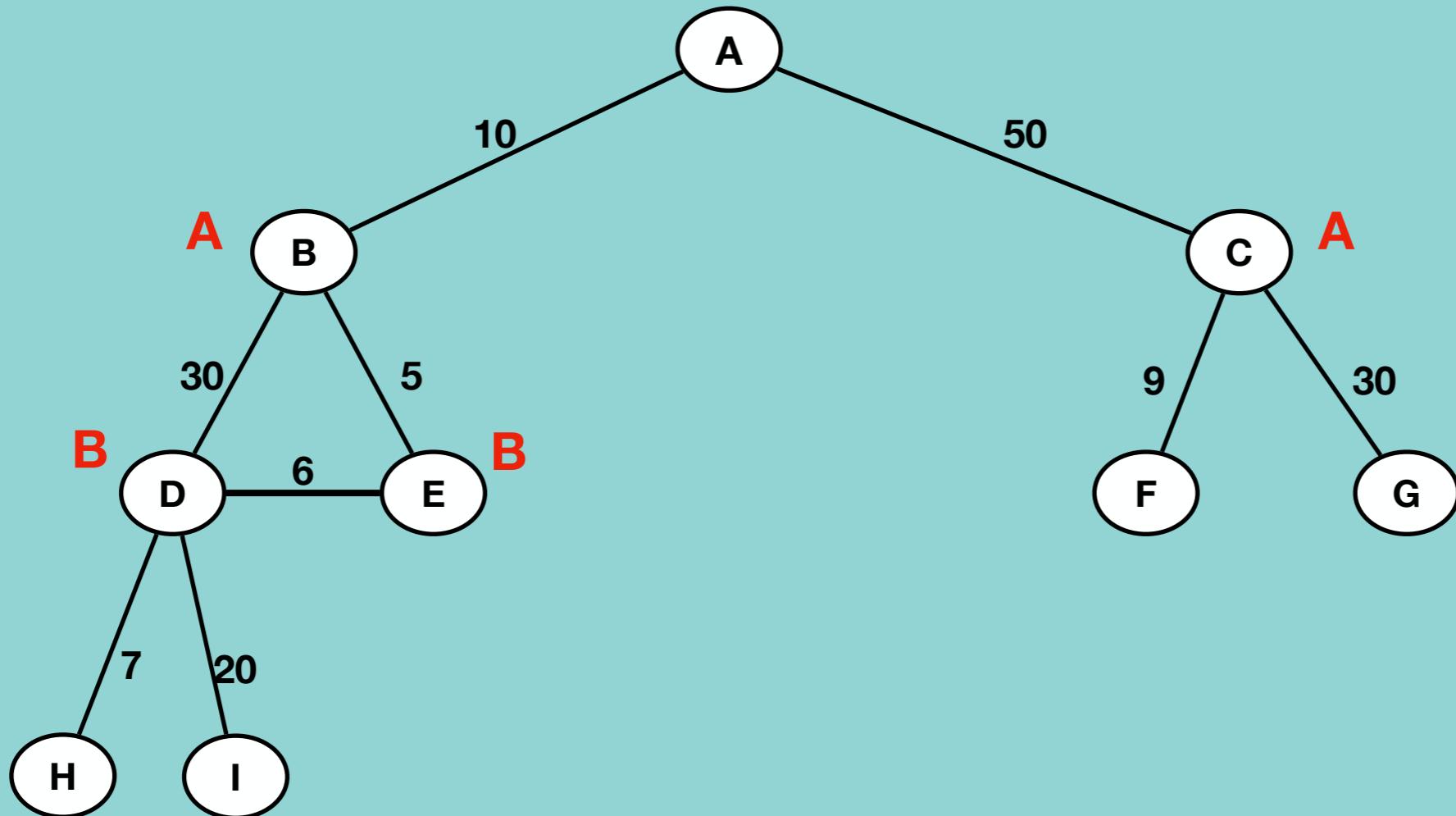


# Why BFS not work with weighted graph

Graph Type	BFS
Unweighted – undirected	OK
Unweighted – directed	OK
Positive – weighted – undirected	X
Positive – weighted – directed	X
Negative – weighted – undirected	X
Negative – weighted – directed	X

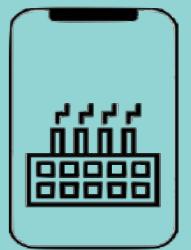
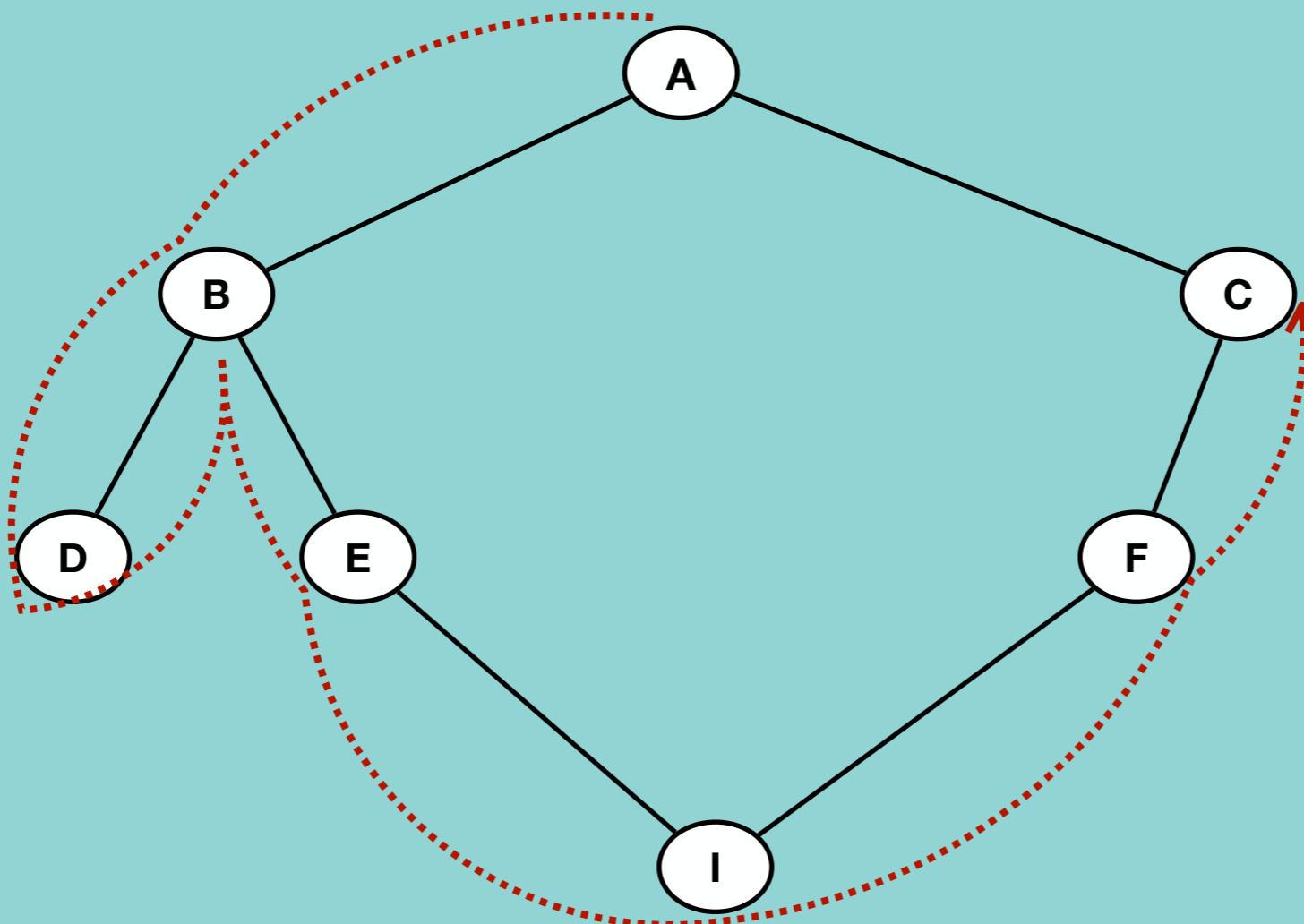


# Why BFS not work with weighted graph

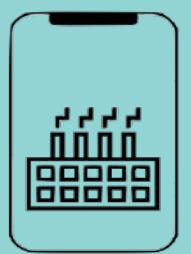
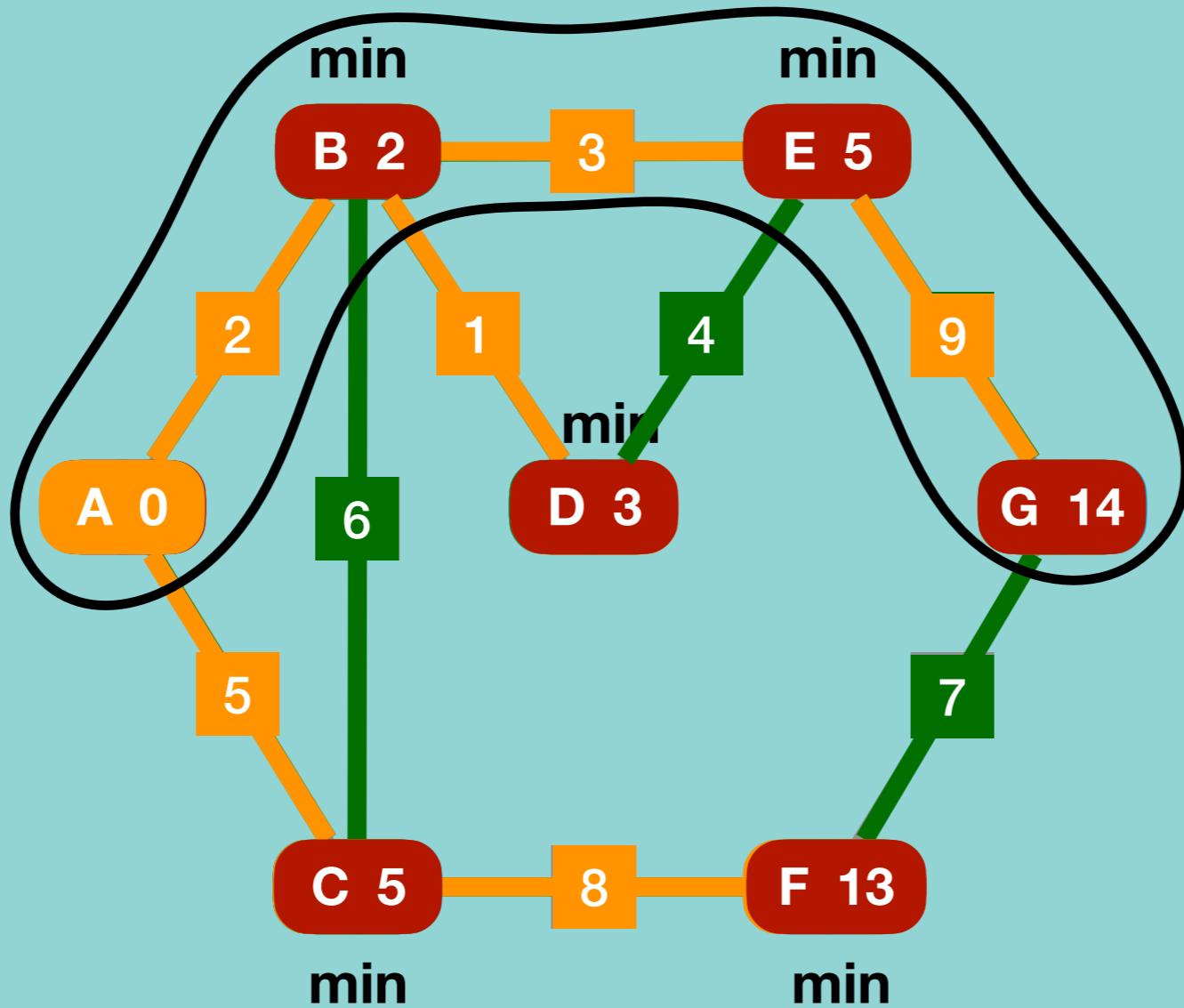


# Why does DFS not work with SSSP?

DFS has the tendency to go “as far as possible” from source, hence it can never find “Shortest Path”



# Dijkstra's Algorithm for SSSP

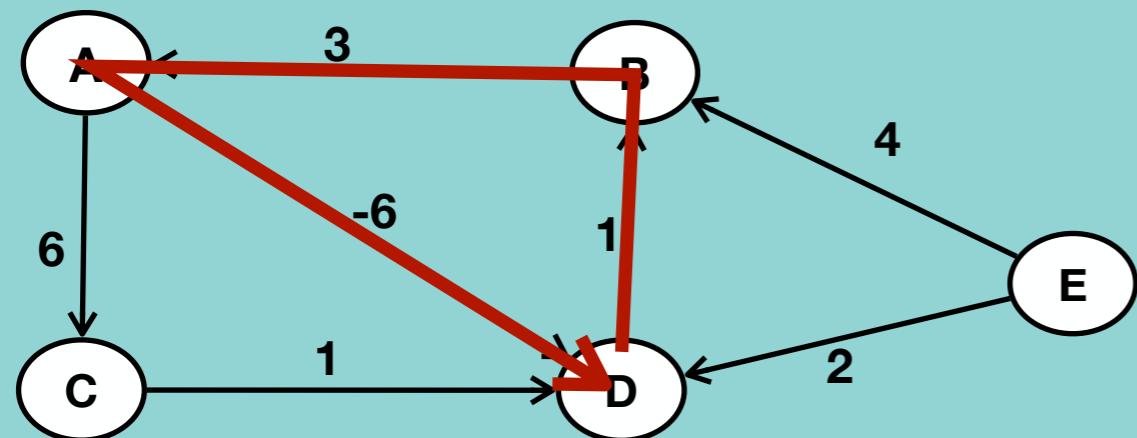


# Dijkstra's Algorithm with negative cycle

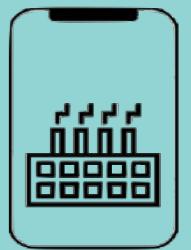
A path is called a negative cycle if:

There is a cycle (a cycle is a path of edges or vertices wherein a vertex is reachable from itself)

Total weight of cycle is a negative number

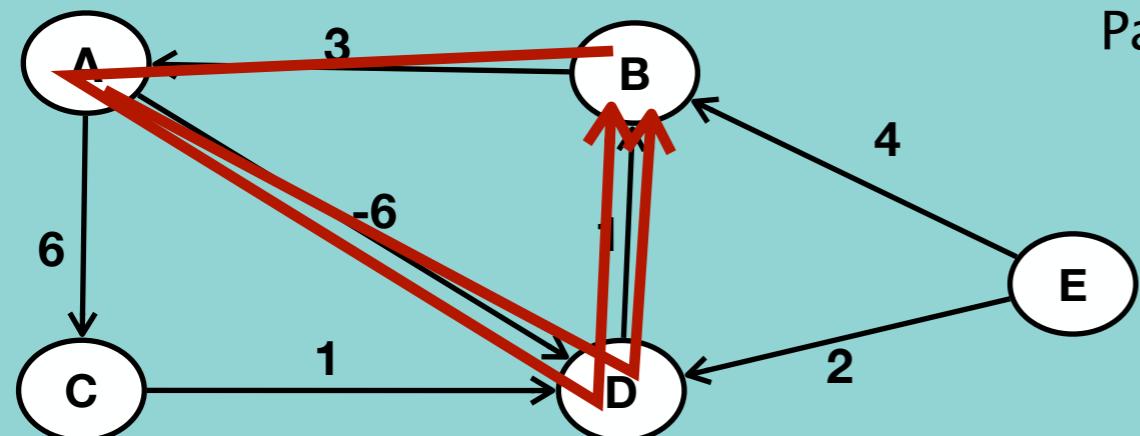


$$1 + 3 + (-6) = -2$$

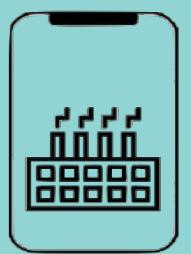


# Dijkstra's Algorithm with negative cycle

We cannot never find a negative cycle in a graph



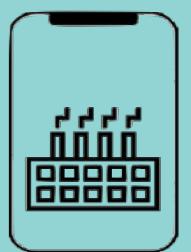
Path from A to B =  $6 + 1 = -5$   
 $= -5 + 3 + (-6) + 1 = -7$   
 $= -7 + 3 + (-6) + 1 = -9$   
 $= -9 + 3 + (-6) + 1 = -11$



# Bellman Ford Algorithm

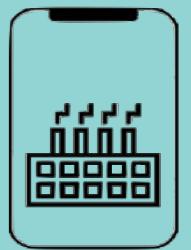
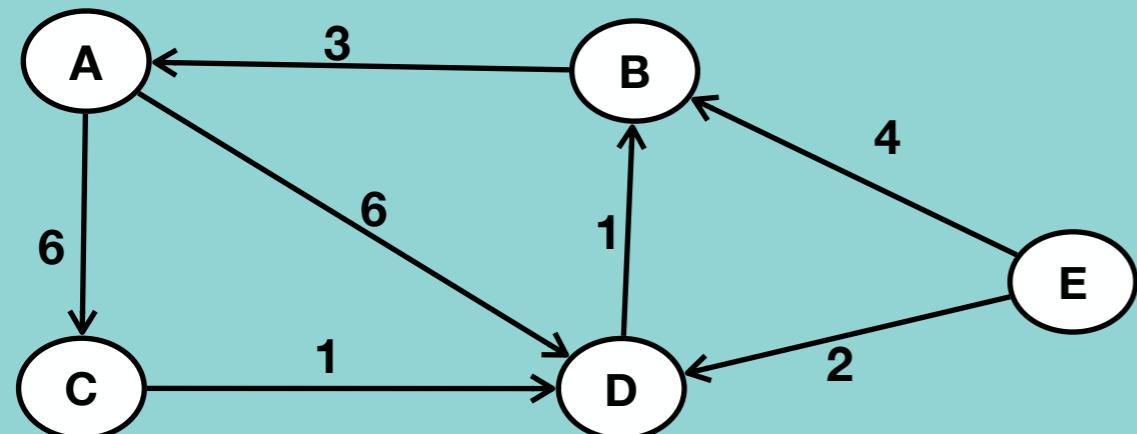
## Single Source Shortest Path Algorithm

Graph Type	BFS	Dijkstra	Bellman Ford
Unweighted – undirected	OK	OK	OK
Unweighted – directed	OK	OK	OK
Positive – weighted – undirected	X	OK	OK
Positive – weighted – directed	X	OK	OK
Negative – weighted – undirected	X	OK	OK
Negative – weighted – directed	X	OK	OK
Negative Cycle	X	X	OK

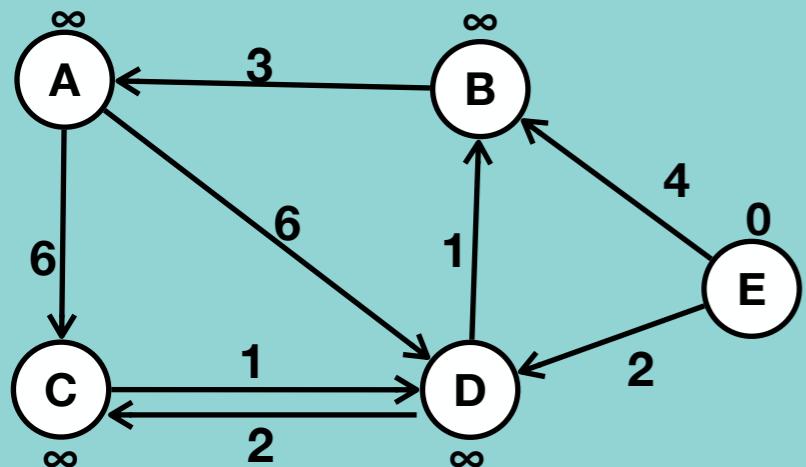


# Bellman Ford Algorithm

Bellman Ford algorithm is used to find single source shortest path problem. If there is a negative cycle it catches it and report its existence.



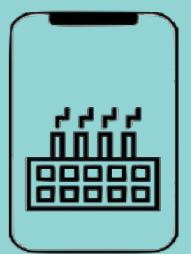
# Bellman Ford Algorithm



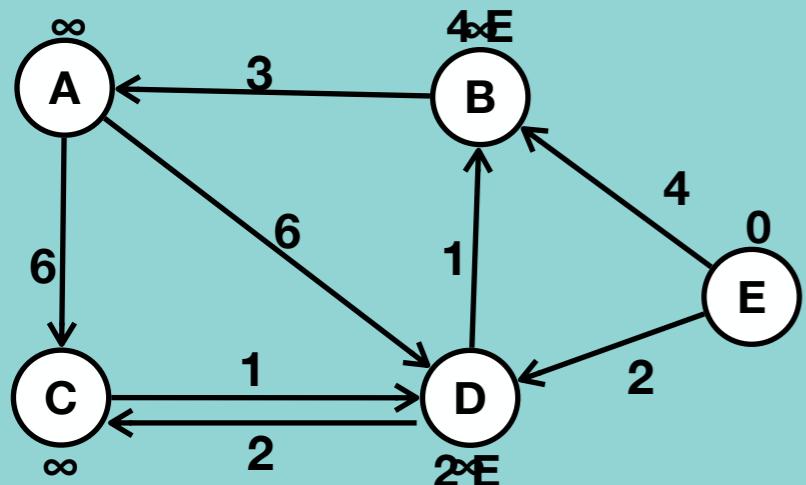
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Distance Matrix	
Vertex	Distance
A	$\infty$
B	$\infty$
C	$\infty$
D	$\infty$
E	0



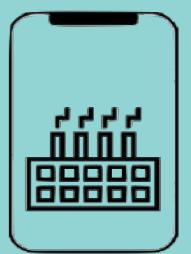
# Bellman Ford Algorithm



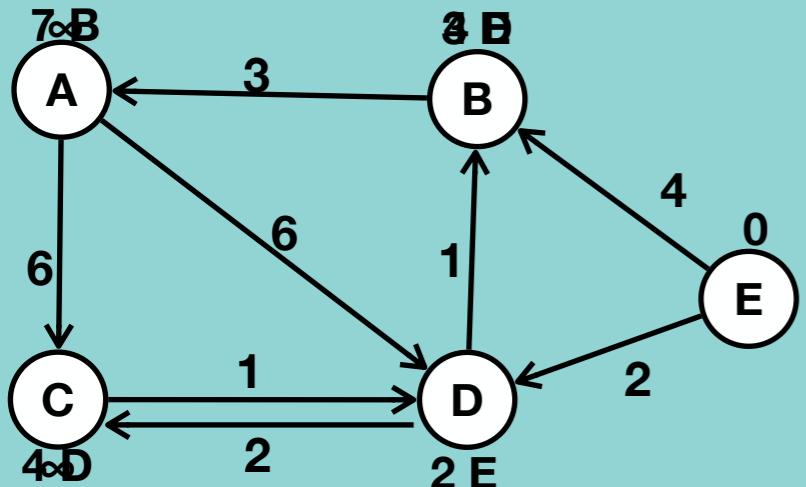
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Distance Matrix		Matrix	
Vertex	Distance	Iteration 1	
		Inc	Parent
A	$\infty$		-
B	$\infty$		E
C	$\infty$		-
D	$\infty$		E
E	0		-



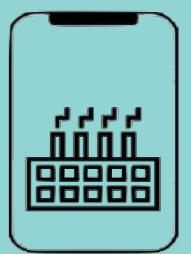
# Bellman Ford Algorithm



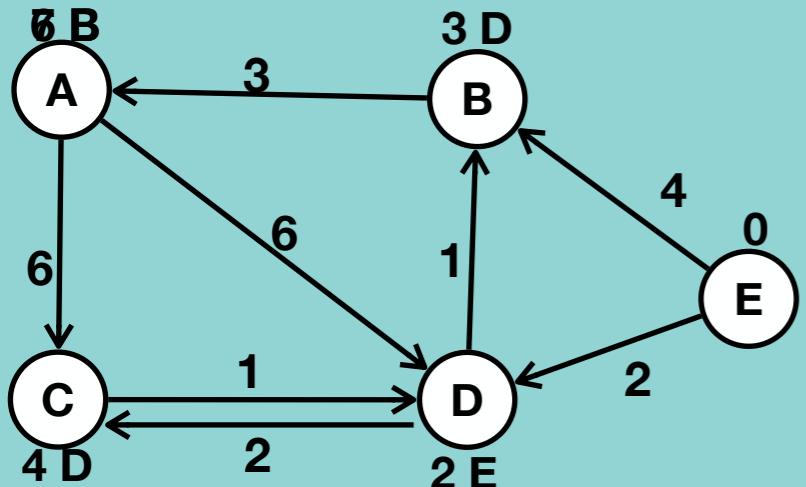
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

		Distance Matrix			
Vertex	Distance	Iteration 1		Iteration 2	
		Distance	Parent	Distance	Parent
A	$\infty$	$\infty$	-	$4+3=7$	B
B	$\infty$	4	E	$2+1=3$	D
C	$\infty$	$\infty$	-	$2+1=4$	D
D	$\infty$	2	E	2	E
E	0	0	-	0	-



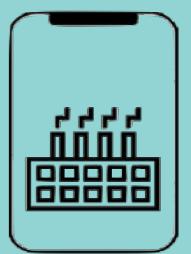
# Bellman Ford Algorithm



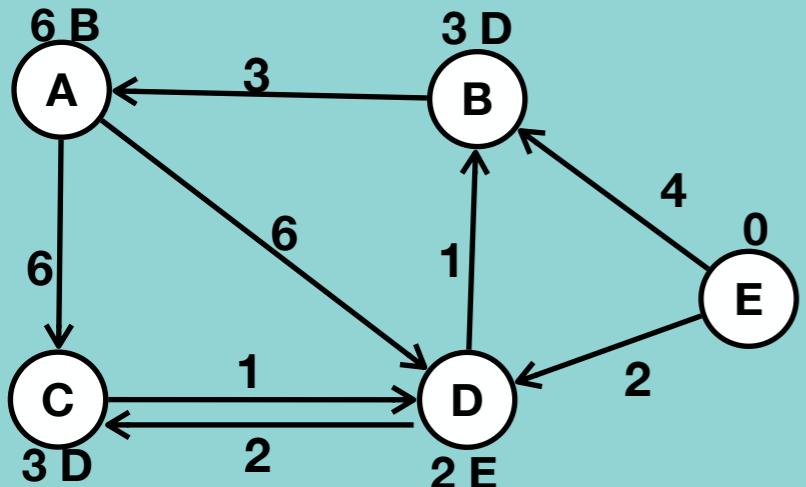
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

		Distance Matrix					
Vertex	Distance	Iteration 1		Iteration 2		Iteration 3	
		Distance	Parent	Distance	Parent	Distance	Parent
A	$\infty$	$\infty$	-	$4+3=7$	B	$3+3=6$	B
B	$\infty$	4	E	$2+1=3$	D	3	D
C	$\infty$	$\infty$	-	$2+1=4$	D	4	D
D	$\infty$	2	E	2	E	2	E
E	0	0	-	0	-	0	



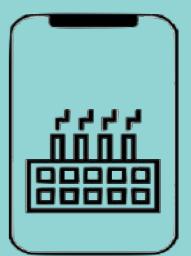
# Bellman Ford Algorithm



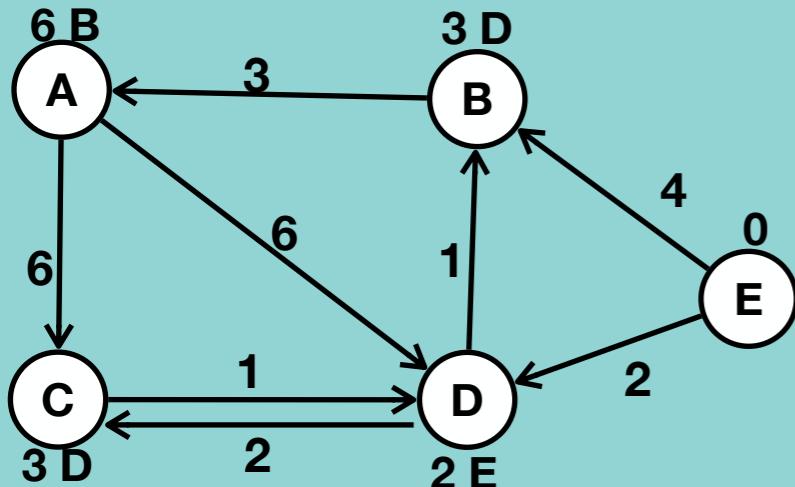
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Distance Matrix									
Vertex	Distance	Iteration 1		Iteration 2		Iteration 3		Iteration 4	
		Distance	Parent	Distance	Parent	Distance	Parent	Distance	Parent
A	$\infty$	$\infty$	-	$4+3=7$	B	$3+3=6$	B	6	B
B	$\infty$	4	E	$2+1=3$	D	3	D	3	D
C	$\infty$	$\infty$	-	$2+2=4$	D	4	D	4	D
D	$\infty$	2	E	2	E	2	E	2	E
E	0	0	-	0	-	0	-	0	-



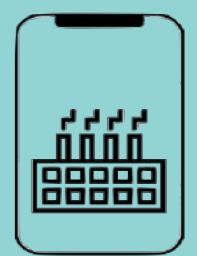
# Bellman Ford Algorithm



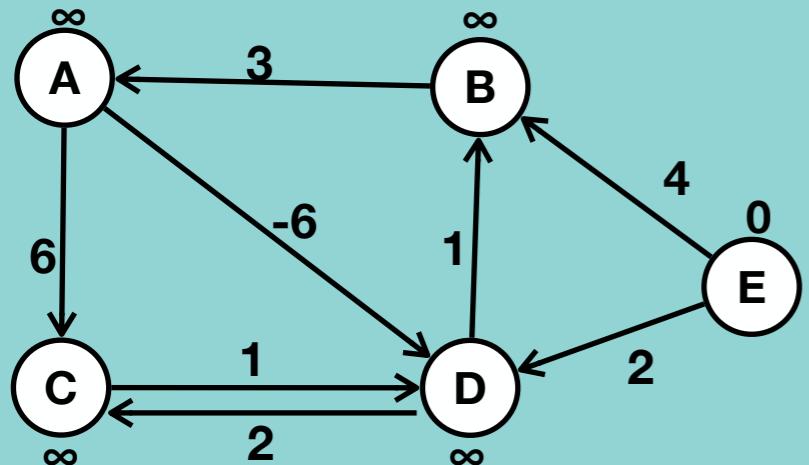
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Final Solution		
Vertex	Distance from E	Path from E
A	6	E -> D -> B -> A
B	3	E -> D -> B
C	4	E -> D -> C
D	2	E -> D
E	0	0



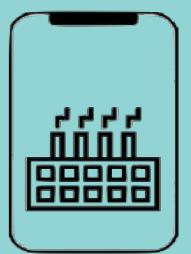
# Bellman Ford Algorithm with negative cycle



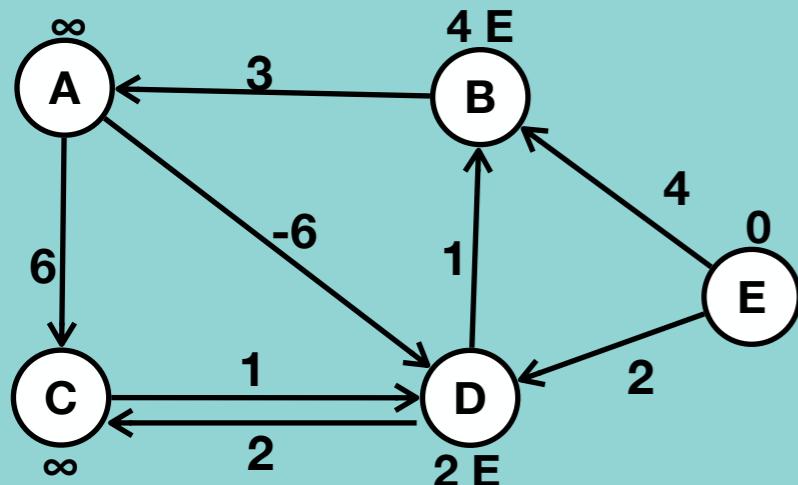
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	-6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Distance Matrix	
Vertex	Distance
A	$\infty$
B	$\infty$
C	$\infty$
D	$\infty$
E	0



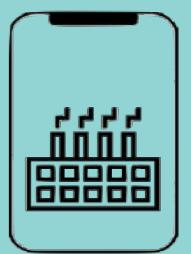
# Bellman Ford Algorithm with negative cycle



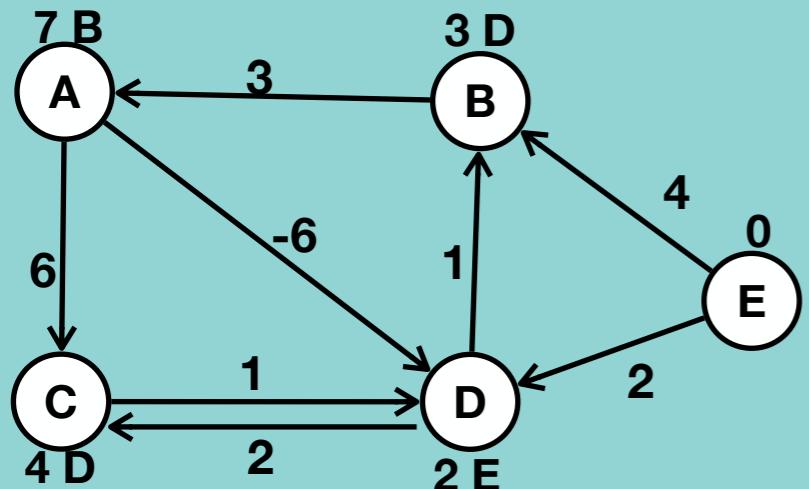
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	-6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Distance Matrix				
Vertex	Distance	Iteration 1		Parent
		Distance	Parent	
A	$\infty$	$\infty$	-	-
B	$\infty$	4	E	-
C	$\infty$	$\infty$	-	-
D	$\infty$	2	E	-
E	0	0	-	-



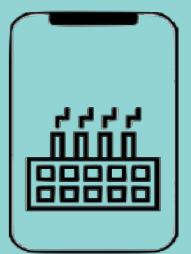
# Bellman Ford Algorithm with negative cycle



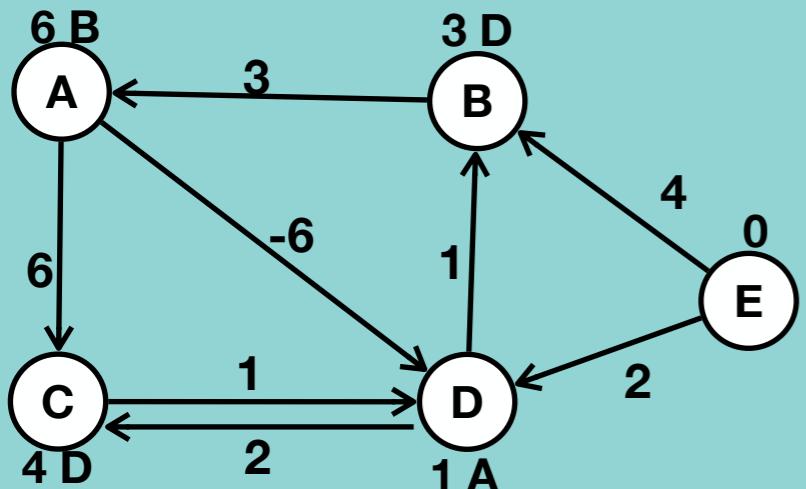
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	-6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Distance Matrix						
Vertex	Distance	Iteration 1		Iteration 2		Parent
		Distance	Parent	Distance	Parent	
A	$\infty$	$\infty$	-	$4+3=7$	B	
B	$\infty$	4	E	$2+1=3$	D	
C	$\infty$	$\infty$	-	$2+2=4$	D	
D	$\infty$	2	E	2	E	
E	0	0	-	0	-	



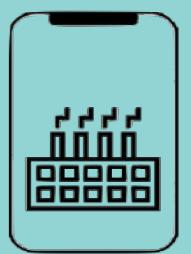
# Bellman Ford Algorithm with negative cycle



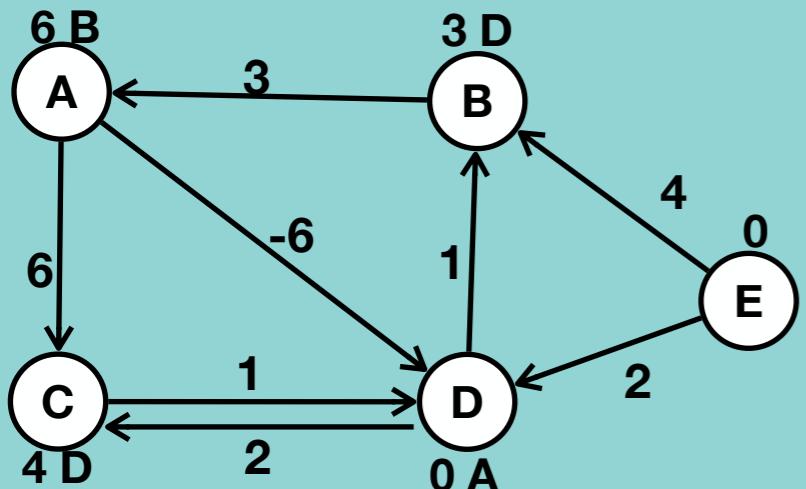
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Edge	Weight
A->C	6
A->D	-6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Distance Matrix							
Vertex	Distance	Iteration 1		Iteration 2		Iteration 3	
		Distance	Parent	Distance	Parent	Distance	Parent
A	$\infty$	$\infty$	-	$4+3=7$	B	$3+3=6$	B
B	$\infty$	4	E	$2+1=3$	D	3	D
C	$\infty$	$\infty$	-	$2+2=4$	D	4	D
D	$\infty$	2	E	2	E	$7+$	A
E	0	0	-	0	-	0	



# Bellman Ford Algorithm with negative cycle

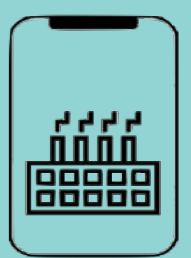


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :

Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

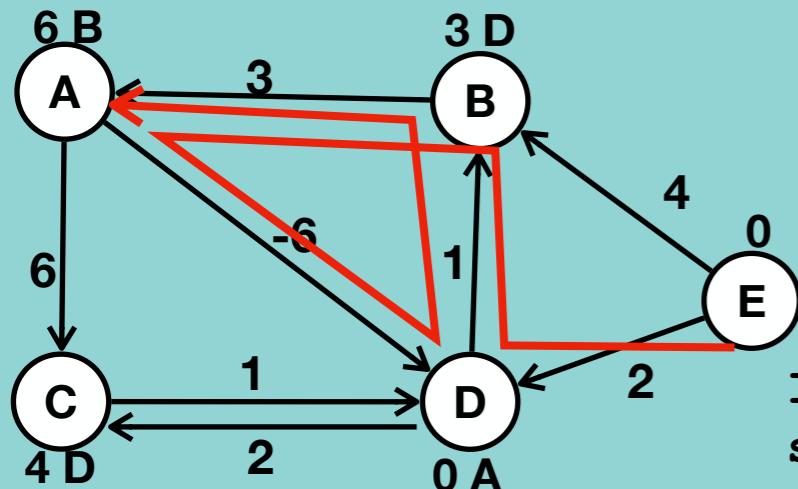
Edge	Weight
A->C	6
A->D	-6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Distance Matrix										
Edge	Vertex	Distance	Iteration 1		Iteration 2		Iteration 3		Iteration 4	
			Distance	Parent	Distance	Parent	Distance	Parent	Distance	Parent
A->C	A	$\infty$	$\infty$	-	$4+3=7$	B	$3+3=6$	B	6	B
A->D	B	$\infty$	4	E	$2+1=3$	D	3	D	3	D
B->A	C	$\infty$	$\infty$	-	$2+2=4$	D	4	D	4	D
C->D	D	$\infty$	2	E	2	E	$7+$	A	$6+$	A
D->C	E	0	0	-	0	-	0	-	0	-



Edge	Weight
A->C	6
A->D	-6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

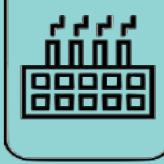
## Bellman Ford Algorithm with negative cycle



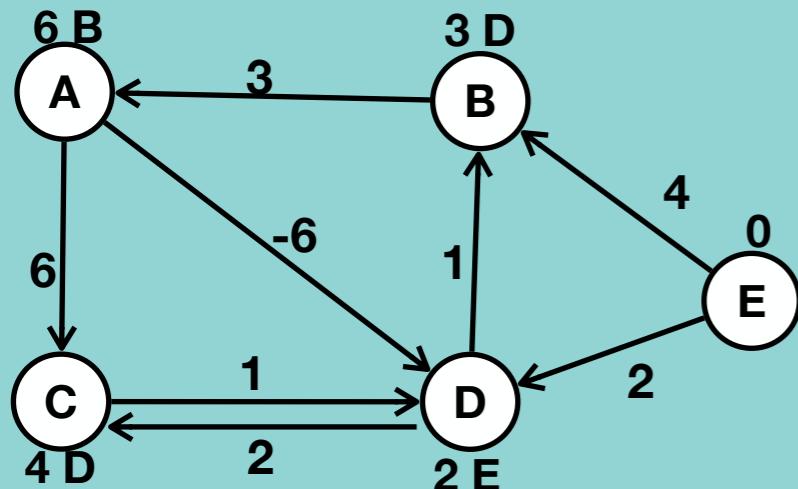
If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

Distance Matrix											
Vertex	Distance	Iteration 1		Iteration 2		Iteration 3		Iteration 4		Iteration 5	
		Distance	Parent								
A	$\infty$	$\infty$	-	$4+3=7$	B	$3+3=6$	B	6	B	4	B
B	$\infty$	4	E	$2+1=3$	D	3	D	3	D	0	D
C	$\infty$	$\infty$	-	$2+2=4$	D	4	D	4	D	0	D
D	$\infty$	2	E	2	E	$7+(-6)=1$	A	$6+(-6)=0$	A	1	A
E	0	0	-	0	-	0	-	0	-	-	-



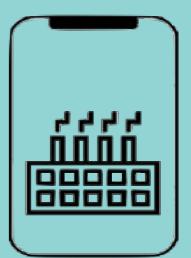
# Bellman Ford Algorithm with negative cycle



If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :  
 Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

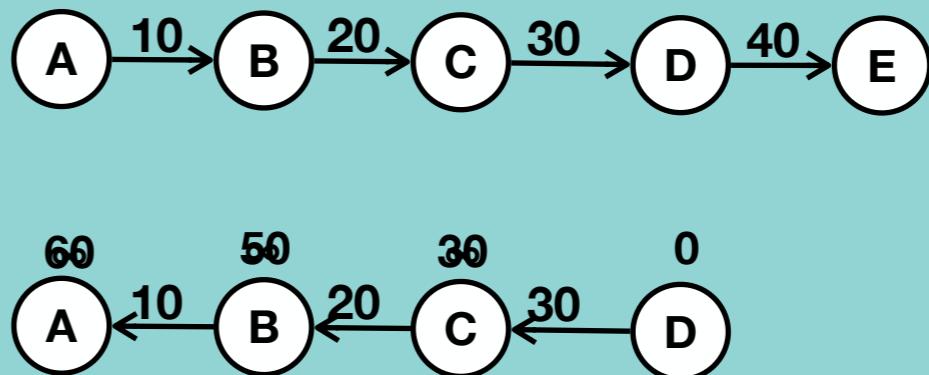
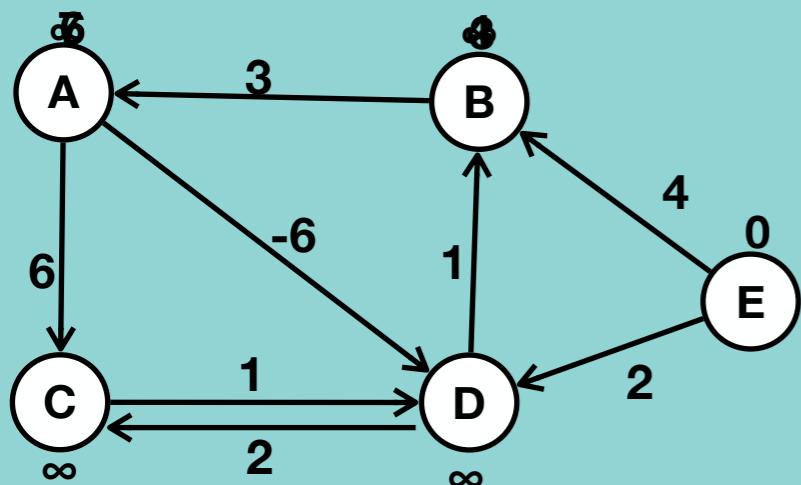
Edge	Weight
A->C	6
A->D	6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

Final Solution		
Vertex	Distance from E	Path from E
A	6	E -> D -> B -> A
B	3	E -> D -> B
C	4	E -> D -> C
D	2	E -> D
E	0	0

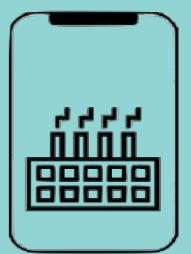


# Why does Bellman Ford run V-1 times?

- If any node is achieved better distance in previous iteration, then that better distance is used to improve distance of other vertices
- Identify worst case graph that can be given to us

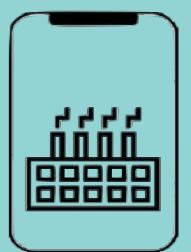


A->B  
B->C  
C->D



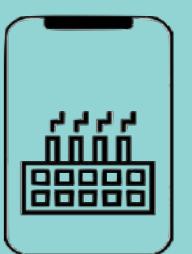
# BFS vs Dijkstra vs Bellman Ford

Graph Type	BFS	Dijkstra	Bellman Ford
Unweighted – undirected	OK	OK	OK
Unweighted – directed	OK	OK	OK
Positive – weighted – undirected	X	OK	OK
Positive – weighted – directed	X	OK	OK
Negative – weighted – undirected	X	OK	OK
Negative – weighted – directed	X	OK	OK
Negative Cycle	X	X	OK



# BFS vs Dijkstra vs Bellman Ford

	<b>BFS</b>	<b>Dijkstra</b>	<b>Bellman Ford</b>
Time complexity	$O(V^2)$	$O(V^2)$	$O(VE)$
Space complexity	$O(E)$	$O(V)$	$O(V)$
Implementation	Easy	Moderate	Moderate
Limitation	Not work for weighted graph	Not work for negative cycle	N/A
Unweighted graph	OK	OK	OK
	Use this as time complexity is good and easy to implement	Not use as implementation not easy	Not use as time complexity is bad
Weighted graph	X	OK	OK
	Not supported	Use as time complexity is better than Bellman	Not use as time complexity is bad
Negative Cycle	X	X	OK
	Not supported	Not supported	Use this as others not support

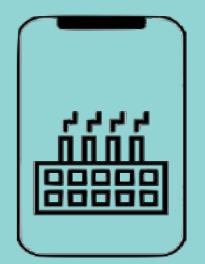
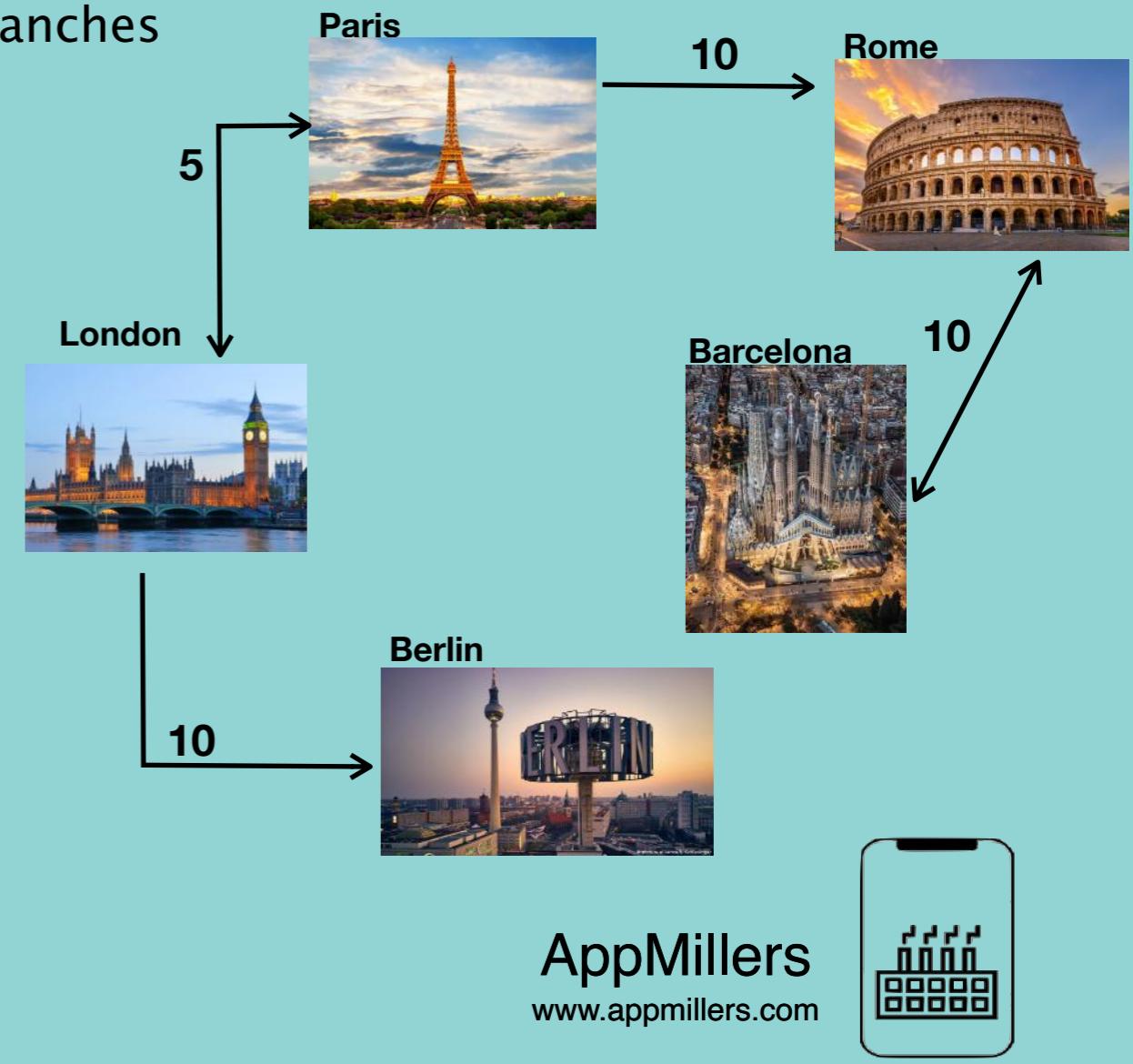
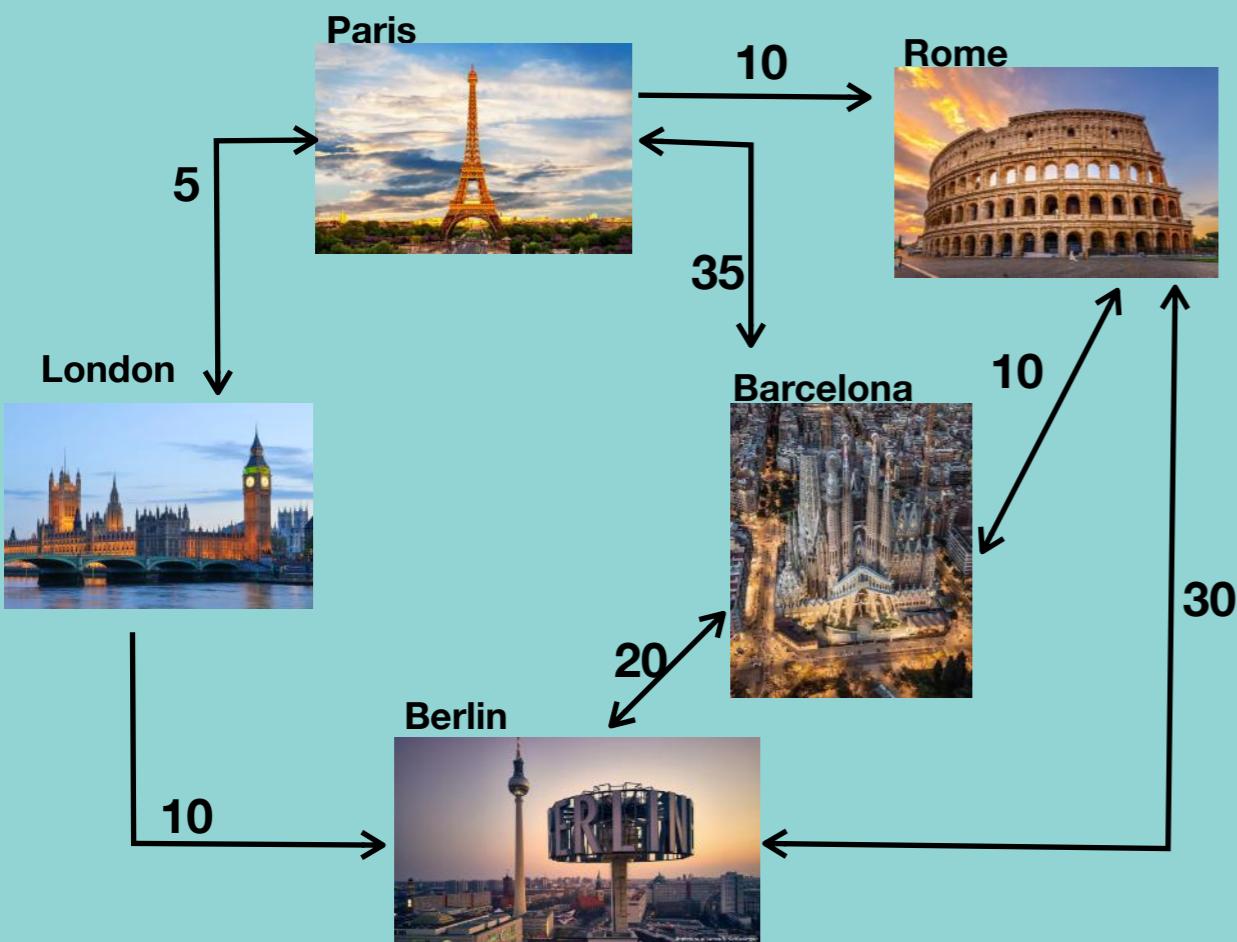


# All pair shortest path problem

## What is single source shortest path?

A single source problem is about finding a path between a given vertex (called source) to all other vertices in a graph such that the total distance between them (source and destination) is minimum.

- Five offices in different cities.
- Travel costs between these cities are known.
- Find the cheapest way from head office to branches in different cities

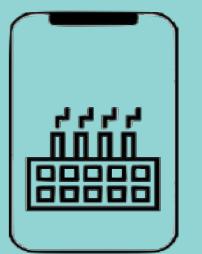
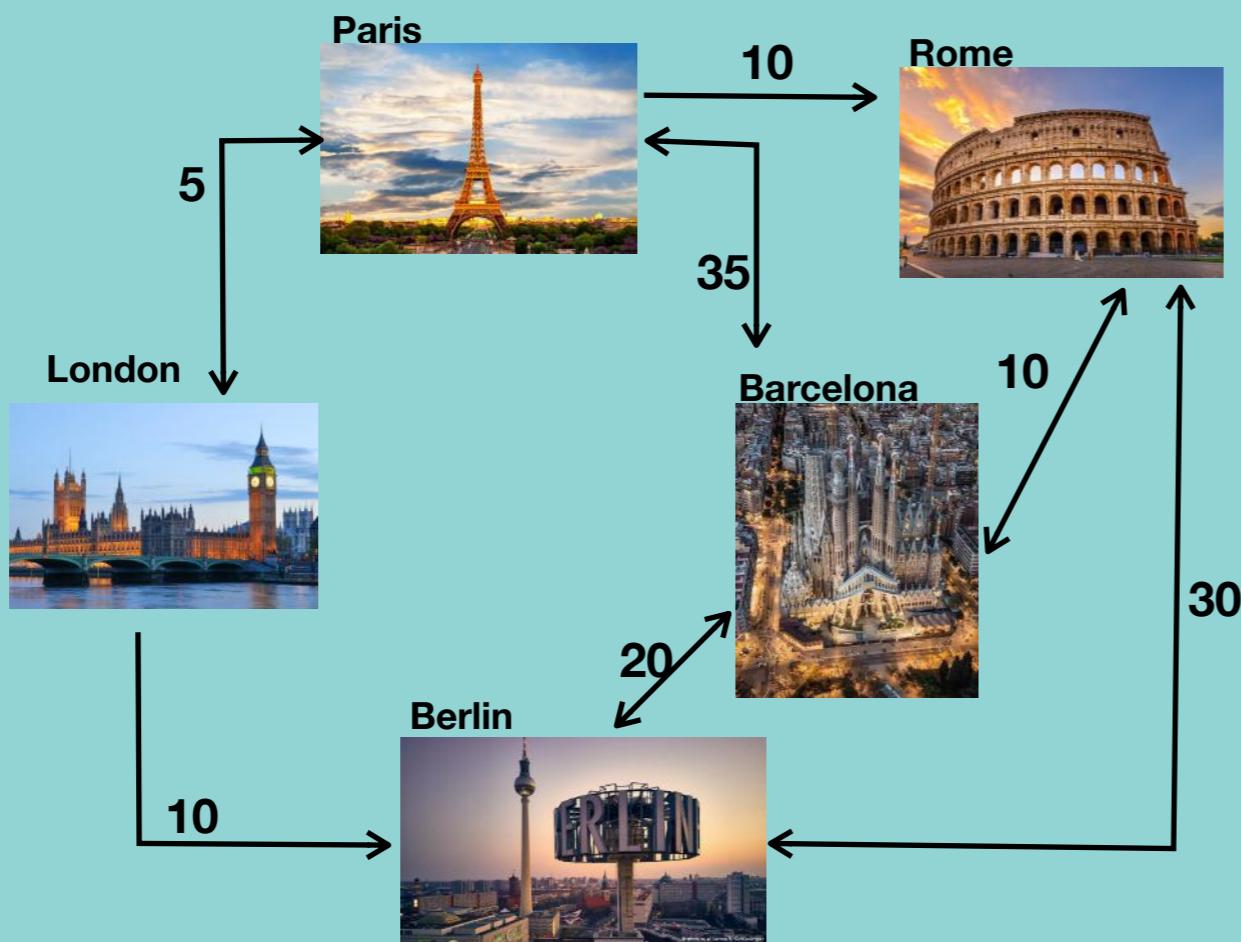


# All pair shortest path problem

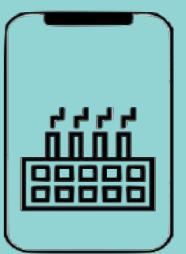
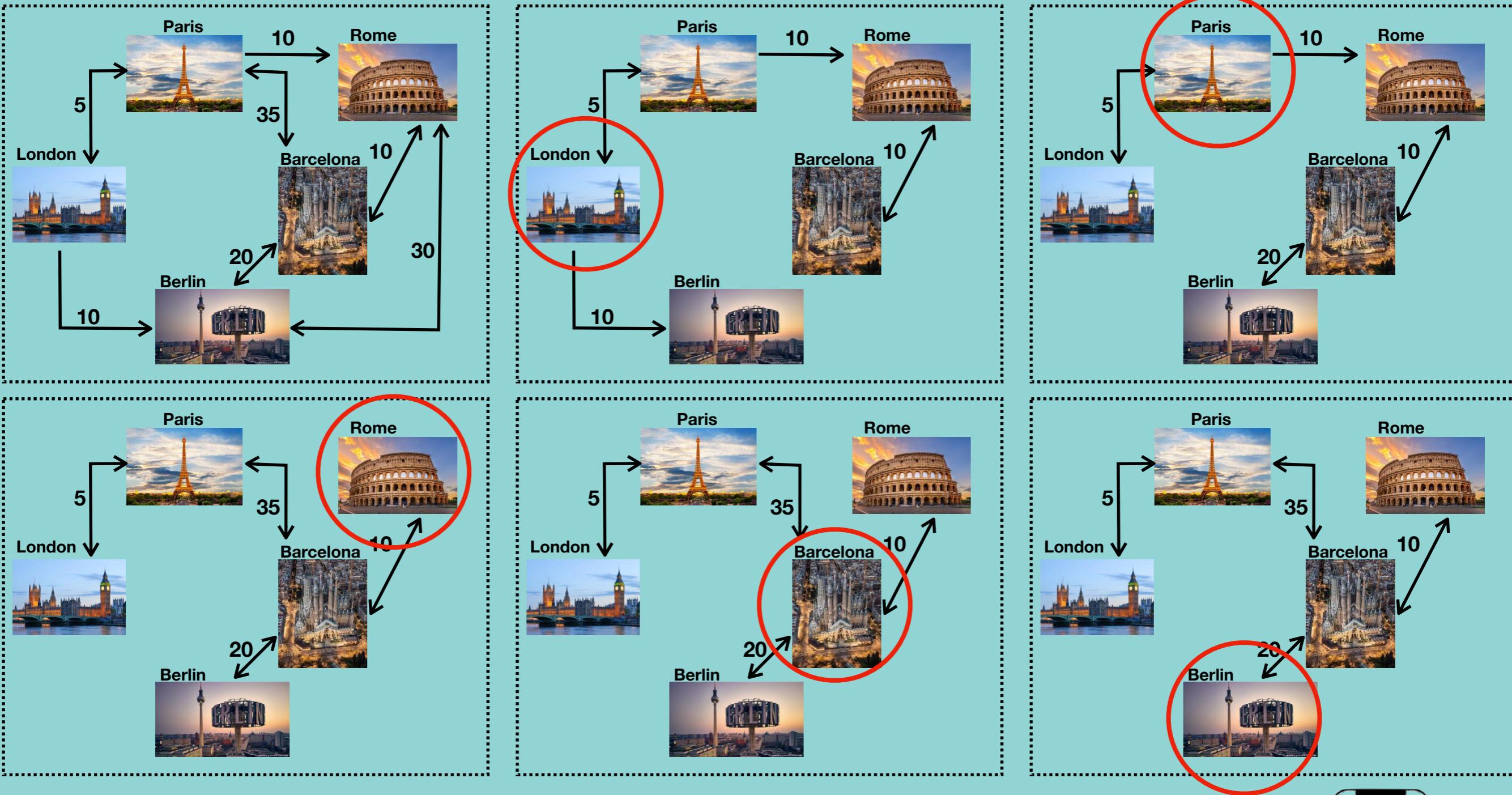
All pair shortest path problem is about finding a path between **every vertex** to all other vertices in a graph such that the total distance between them (source and destination) is minimum.

The problem:

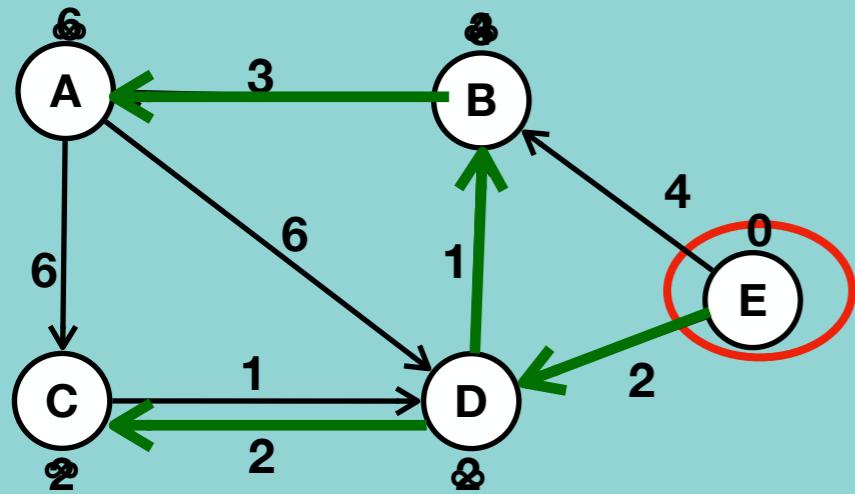
- Five offices in different cities.
- Travel costs between these cities are known.
- Find the cheapest way to reach each office from **every other office**



# All pair shortest path problem

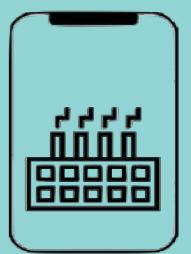


# Dry run for All pair shortest path problem

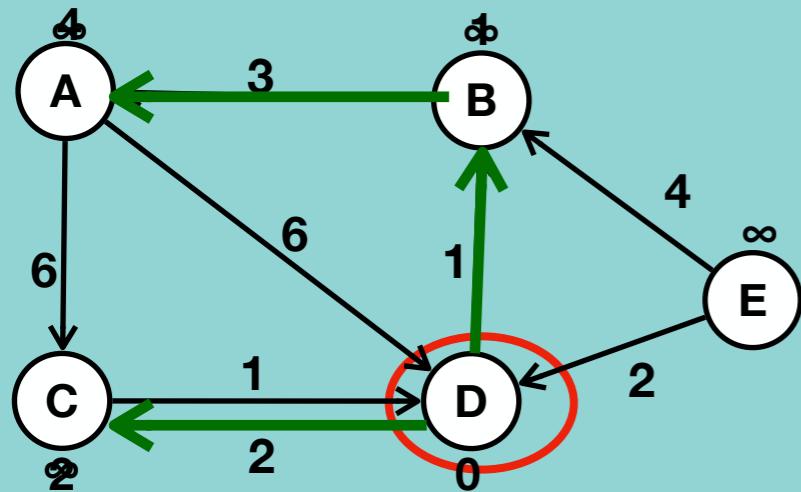


Source Vertex	Path
A	E->D->B->A
B	E->D->B
C	E->D->C
D	E->D
E	-

Dijkstra



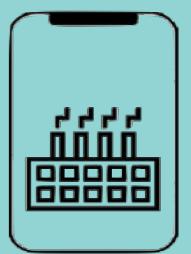
# Dry run for All pair shortest path problem



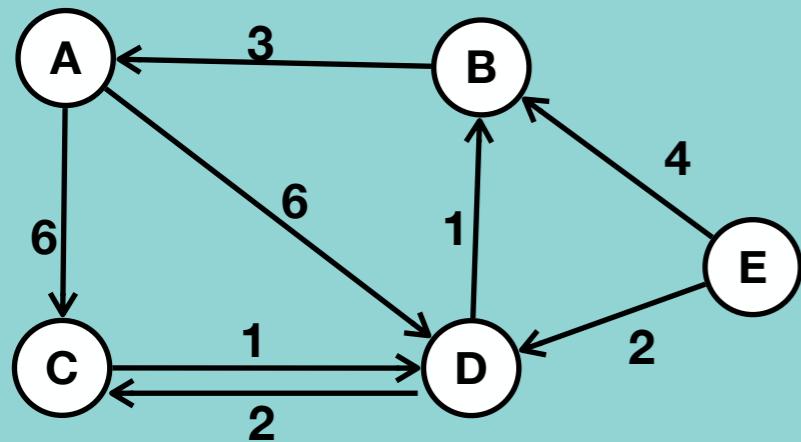
Source Vertex	Path
A	E->D->B->A
B	E->D->B
C	E->D->C
D	E->D
E	-

Source Vertex	Path
D	D->B->A
A	D->B
B	D->C
C	-
E	N/A

Dijkstra



# Dry run for All pair shortest path problem



Source Vertex	Path
E	E->D->B->A
A	E->D->B
B	E->D->C
C	E->D
D	-
E	-

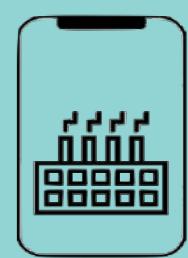
Source Vertex	Path
D	D->B->A
A	D->B
B	D->C
C	-
D	N/A
E	N/A

Dijkstra , BFS and Bellmand Ford

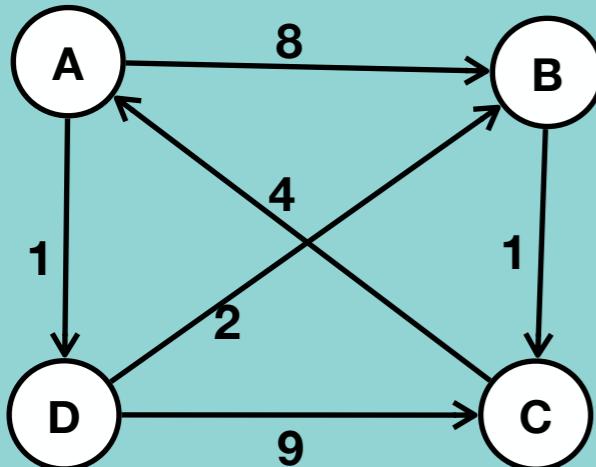
Source Vertex	Path
C	C->D->B->A
A	C->D->B
B	-
C	C->D
D	N/A

Source Vertex	Path
D	B->A
A	-
B	-
C	B->A->C
D	A->D
E	N/A

Source Vertex	Path
A	-
B	A->D->B
C	A->D->C
D	A->D
E	N/A



# Floyd Warshall



		Given			
		A	B	C	D
A		0	8	$\infty$	1
B		$\infty$	0	1	$\infty$
C		4	$\infty$	0	$\infty$
D		$\infty$	2	9	0

		Iteration 1				
		via A	A	B	C	D
A		0	8	$\infty$	1	
B		$\infty$	0	1	$\infty$	
C		4	$4+8=12$	0	$4+1=5$	
D		$\infty$	2	9	0	

If  $D[u][v] > D[u][\text{via } X] + D[\text{via } X][v]$ :  
 $D[u][v] = D[u][\text{via } X] + D[\text{via } X][v]$

C  $\rightarrow$  B

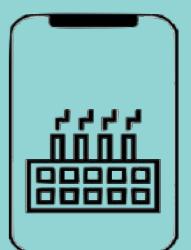
If  $D[C][B] > D[C][A] + D[A][B]$ :  
 $D[C][B] = D[C][A] + D[A][B]$

$$\infty > 4 + 8 = 12$$

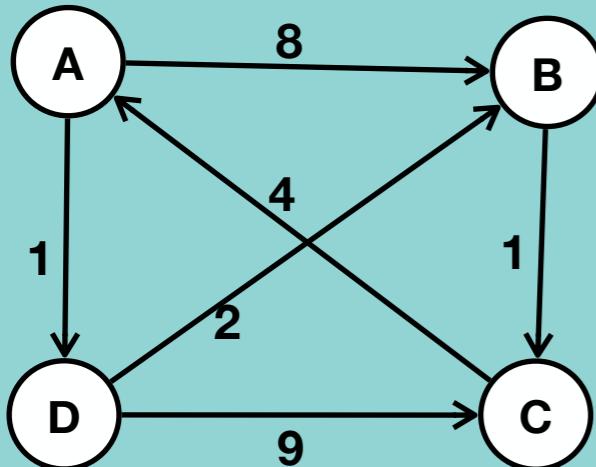
C  $\rightarrow$  D

If  $D[C][D] > D[C][A] + D[A][D]$ :  
 $D[C][D] = D[C][A] + D[A][D]$

$$\infty > 4 + 1 = 5$$



# Floyd Warshall



		Given			
		A	B	C	D
A		0	8	$\infty$	1
B		$\infty$	0	1	$\infty$
C		4	$\infty$	0	$\infty$
D		$\infty$	2	9	0

		Iteration 1				
		via A	A	B	C	D
A		0	8	$\infty$	1	
B		$\infty$	0	1	$\infty$	
C		4	$4+8=12$	0	$4+1=5$	
D		$\infty$	2	9	0	

If  $D[u][v] > D[u][\text{via } X] + D[\text{via } X][v]$ :  
 $D[u][v] = D[u][\text{via } X] + D[\text{via } X][v]$

A  $\rightarrow$  C

If  $D[A][C] > D[A][B] + D[B][C]$ :  
 $D[A][C] = D[A][B] + D[B][C]$

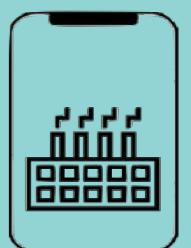
$$\infty > 8 + 1 = 9$$

D  $\rightarrow$  C

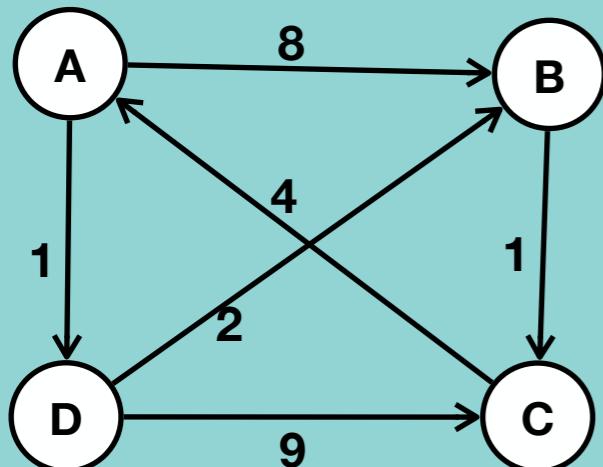
If  $D[D][C] > D[D][B] + D[B][C]$ :  
 $D[D][C] = D[D][B] + D[B][C]$

$$9 > 2 + 1 = 3$$

		Iteration 2				
		via B	A	B	C	D
A		0	8	$8+1=9$	1	
B		$\infty$	0	1	$\infty$	
C		4	12	0	5	
D		$\infty$	2	$2+1=3$	0	



# Floyd Warshall



Given				
	A	B	C	D
A	0	8	$\infty$	1
B	$\infty$	0	1	$\infty$
C	4	$\infty$	0	$\infty$
D	$\infty$	2	9	0

Iteration 1				
via A	A	B	C	D
A	0	8	$\infty$	1
B	$\infty$	0	1	$\infty$
C	4	$4+8=12$	0	$4+1=5$
D	$\infty$	2	9	0

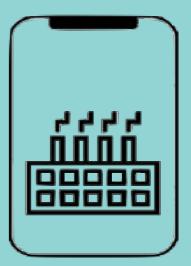
If  $D[u][v] > D[u][\text{via } X] + D[\text{via } X][v]$ :  
 $D[u][v] = D[u][\text{via } X] + D[\text{via } X][v]$

Iteration 2				
via B	A	B	C	D
A	0	8	$8+1=9$	1
B	$\infty$	0	1	$\infty$
C	4	12	0	5
D	$\infty$	2	$2+1=3$	0

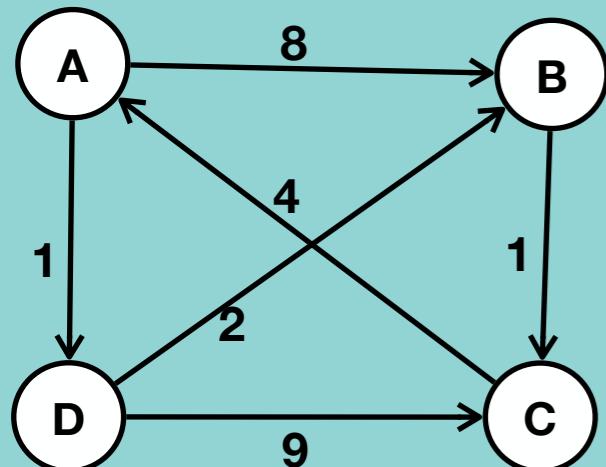
Iteration 3				
via C	A	B	C	D
A	0	8	9	1
B	$1+4=5$	0	1	$1+4+1=6$
C	4	12	0	5
D	$3+4=7$	2	3	0

Iteration 4				
via D	A	B	C	D
A	0	$1+2=3$	$3+1=4$	1
B	5	0	1	6
C	4	$5+2=7$	0	5
D	7	2	3	0

Final answer				
	A	B	C	D
A	0	3	4	1
B	5	0	1	6
C	4	7	0	5
D	7	2	3	0



# Why Floyd Warshall algorithm?



		Given			
	A	B	C	D	
A	0	8	$\infty$	1	
B	$\infty$	0	1	$\infty$	
C	4	$\infty$	0	$\infty$	
D	$\infty$	2	9	0	

		Iteration 1			
	via A	A	B	C	D
A	0	8	$\infty$	1	
B	$\infty$	0	1	$\infty$	
C	4	$4+8=12$	0	$4+1=5$	
D	$\infty$	2	9	0	

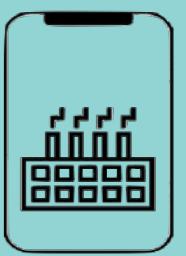
		Iteration 2			
	via B	A	B	C	D
A	0	8	$8+1=9$	1	
B	$\infty$	0	1	$\infty$	
C	4	12	0	5	
D	$\infty$	2	$2+1=3$	0	

		Iteration 3			
	via C	A	B	C	D
A	0	8	9	1	
B	$1+4=5$	0	1	$1+4+1=6$	
C	4	12	0	5	
D	$3+4=7$	2	3	0	

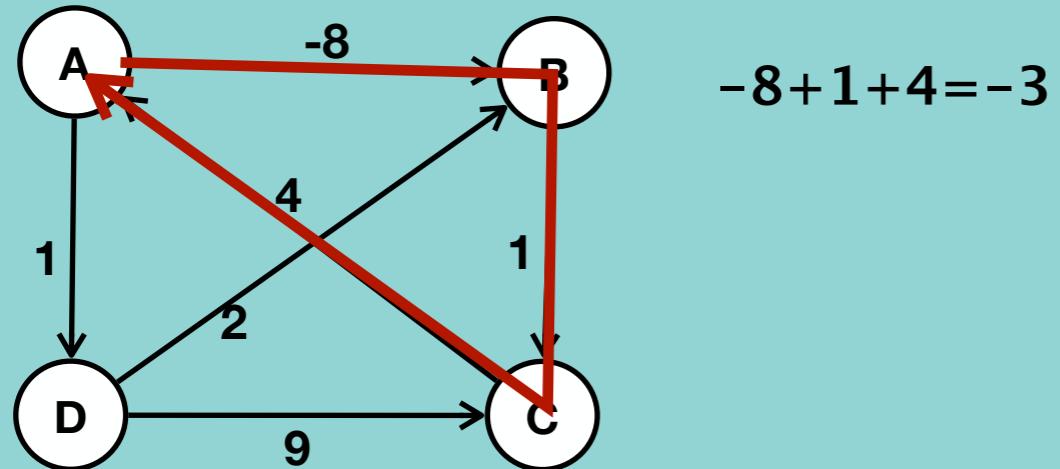
		Iteration 4			
	via D	A	B	C	D
A	0	$1+2=3$	$3+1=4$	1	
B	5	0	1	6	
C	4	$5+2=7$	0	5	
D	7	2	3	0	

		Final answer			
	A	B	C	D	
A	0	3	4	1	
B	5	0	1	6	
C	4	7	0	5	
D	7	2	3	0	

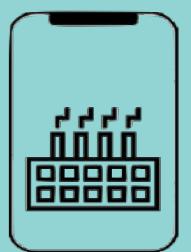
- The vertex is not reachable
- Two vertices are directly connected
  - This is the best solution
  - It can be improved via other vertex
- Two vertices are connected via other vertex



# Floyd Warshall negative cycle

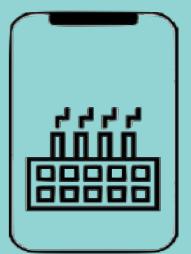


- To go through cycle we need to go via negative cycle participating vertex at least twice
- FW never runs loop twice via same vertex
- Hence, FW can never detect a negative cycle



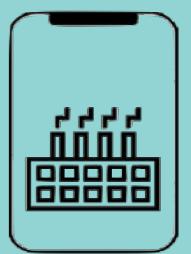
# Which algorithm to use for APSP?

Graph Type	BFS	Dijkstra	Bellman Ford	Floyd Warshall
Unweighted – undirected	OK	OK	OK	OK
Unweighted – directed	OK	OK	OK	OK
Positive – weighted – undirected	X	OK	OK	OK
Positive – weighted – directed	X	OK	OK	OK
Negative – weighted – undirected	X	OK	OK	OK
Negative – weighted – directed	X	OK	OK	OK
Negative Cycle	X	X	OK	X



# Which algorithm to use for APSP?

	<b>BFS</b>	<b>Dijkstra</b>	<b>Bellman Ford</b>	<b>Floyd Warshall</b>
Time complexity	$O(V^3)$	$O(V^3)$	$O(EV^2)$	$O(V^3)$
Space complexity	$O(EV)$	$O(EV)$	$O(V^2)$	$O(V^2)$
Implementation	Easy	Moderate	Moderate	Moderate
Limitation	Not work for weighted graph	Not work for negative cycle	N/A	Not work for negative cycle
Unweighted graph	OK Use this as time complexity is good and easy to implement	OK Not use as priority queue slows it	OK Not use as time complexity is bad	OK Can be used
Weighted graph	X Not supported	OK Can be used	OK Not use as time complexity is bad	OK Can be preferred as implementation easy
Negative Cycle	X Not supported	X Not supported	OK Use this as others not support	X No supported



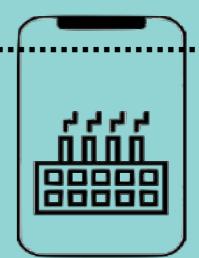
# Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subset of the edges of connected, weighted and undirected graph which :

- Connects all vertices together
- No cycle
- Minimum total edge

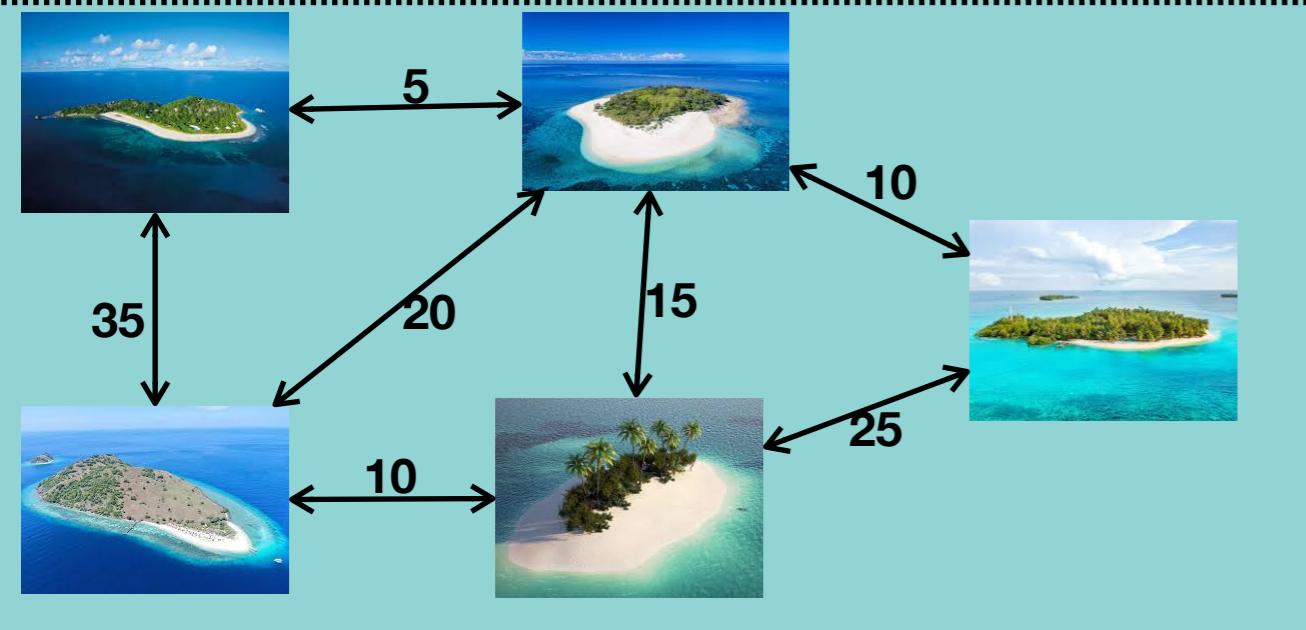
Real life problem

- Connect five island with bridges
- The cost of bridges between island varies based on different factors
- Which bridge should be constructed so that all islands are accessible and the cost is minimum

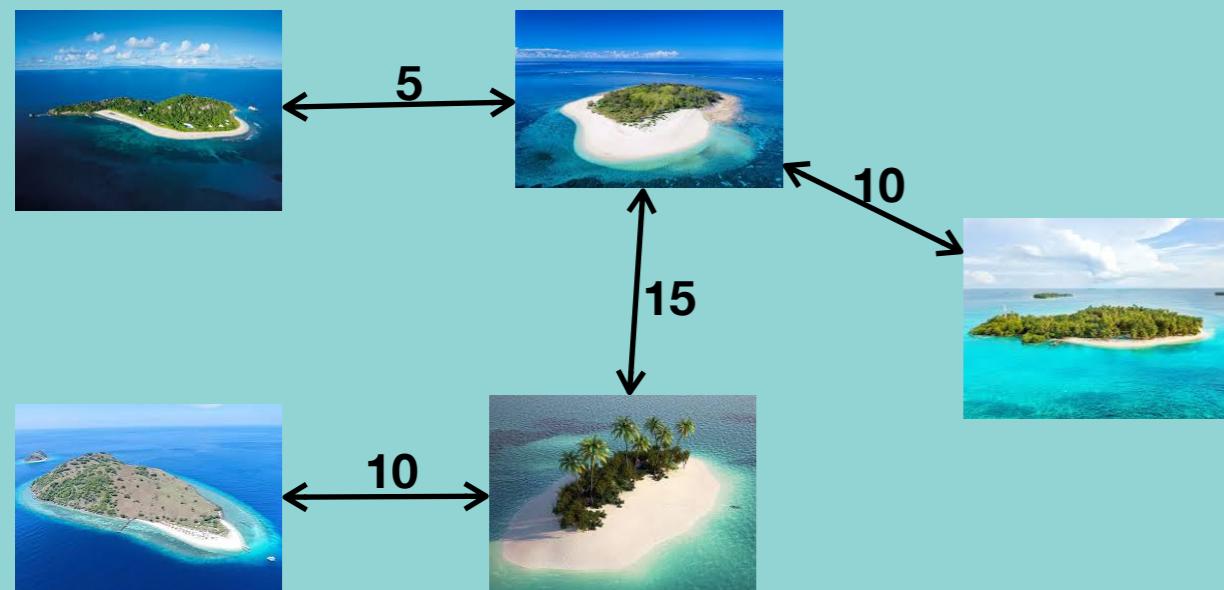


# Minimum Spanning Tree

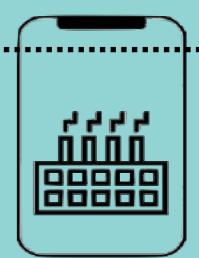
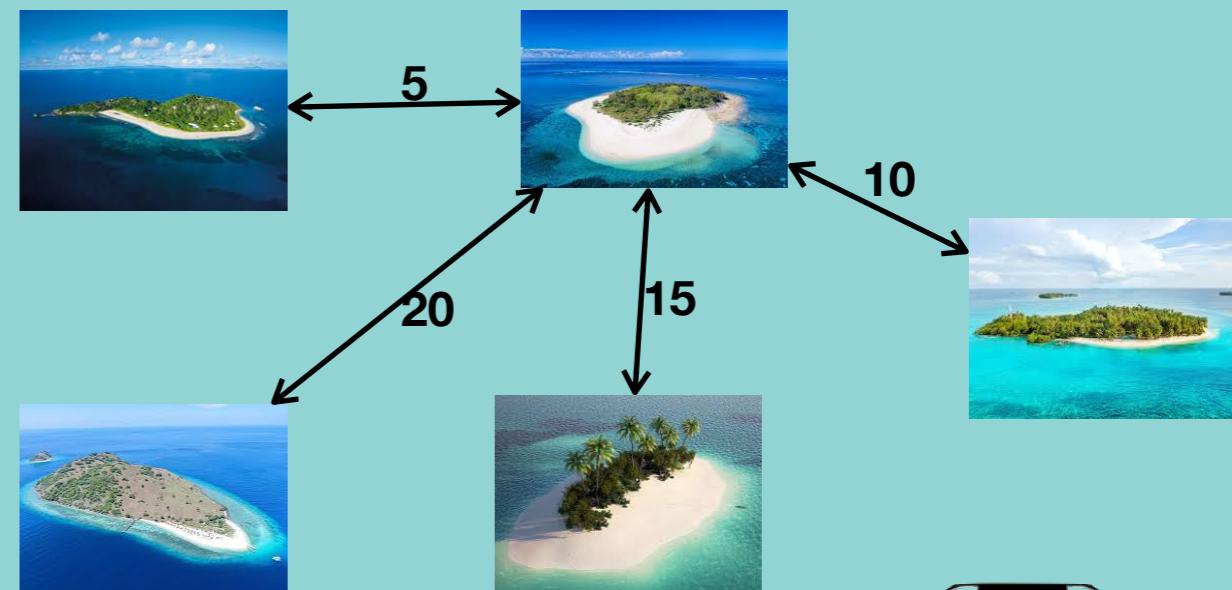
Real life problem



Minimum spanning Tree



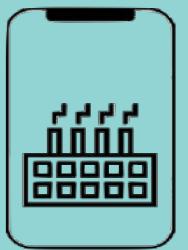
Single source shortest path



# Disjoint Set

It is a data structure that keeps track of set of elements which are partitioned into a number of disjoint and non overlapping sets and each sets have representative which helps in identifying that sets.

- Make Set
- Union
- Find Set

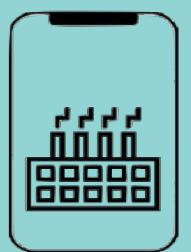
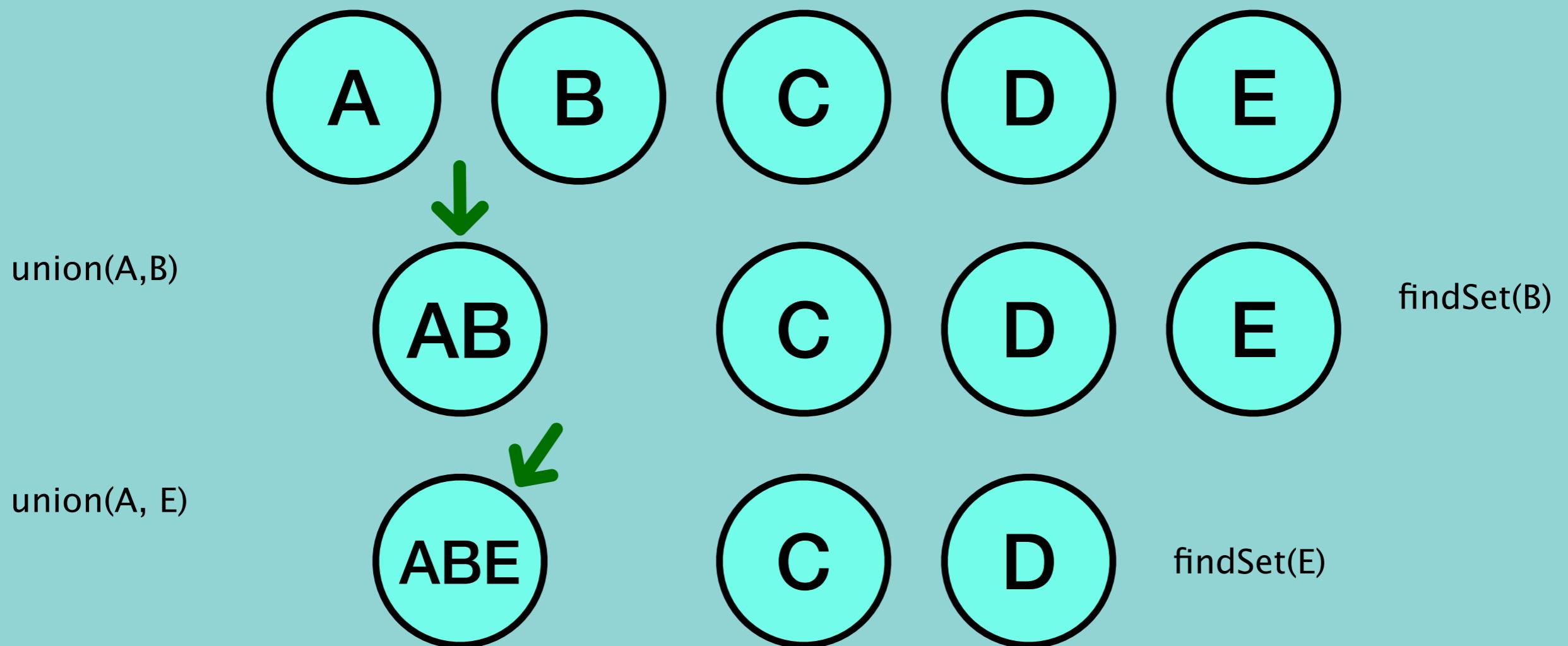


# Disjoint Set

`makeSet(N)` : used to create initial set

`union(x,y)`: merge two given sets

`findSet(x)`: returns the set name in which this element is there

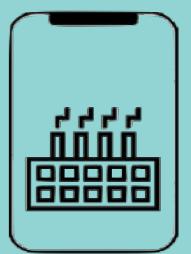
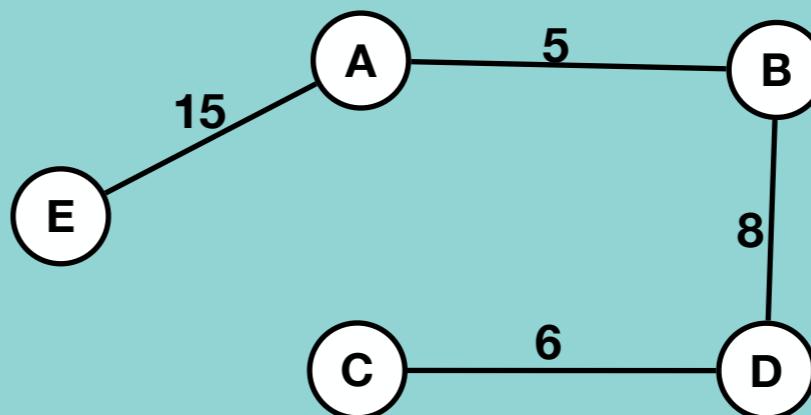
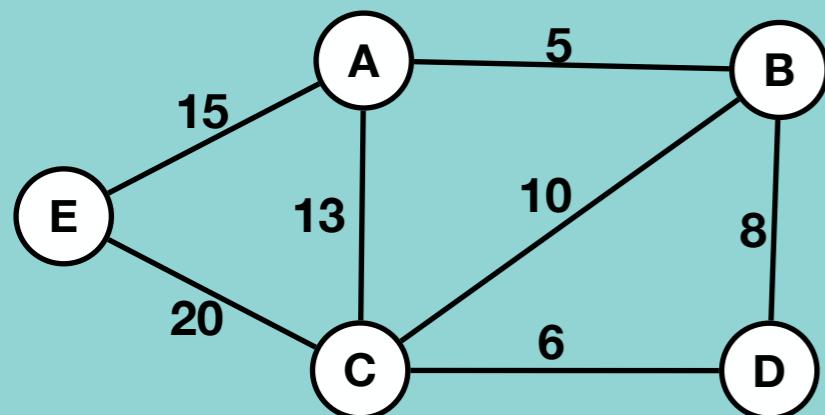


# Kruskal's Algorithm

It is a greedy algorithm

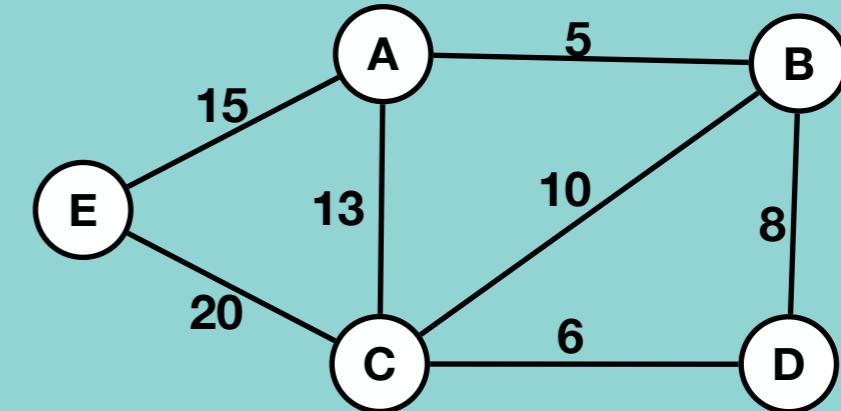
It finds a minimum spanning tree for weighted undirected graphs in two ways

- Add increasing cost edges at each step
- Avoid any cycle at each step

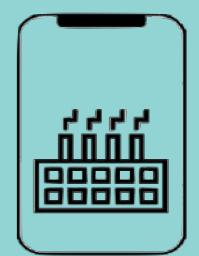
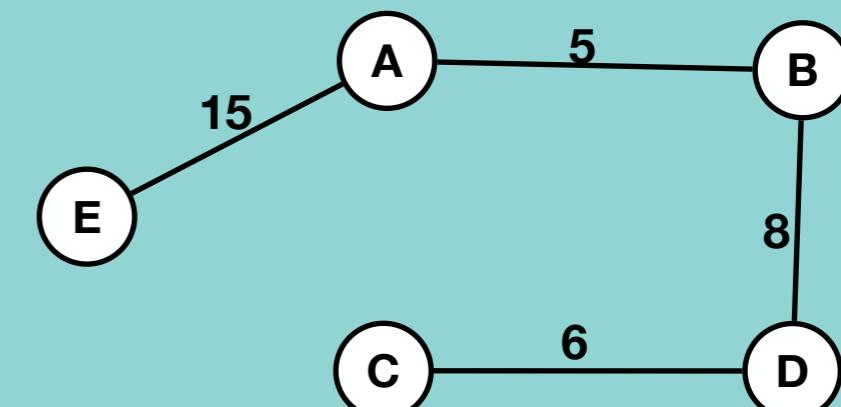
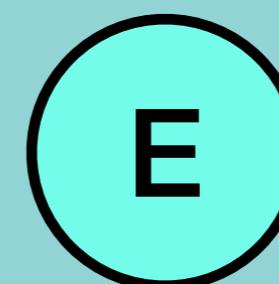
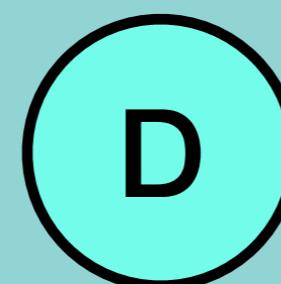
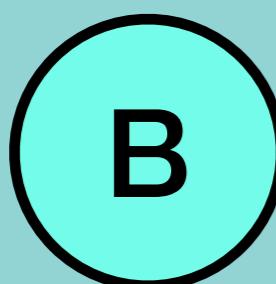
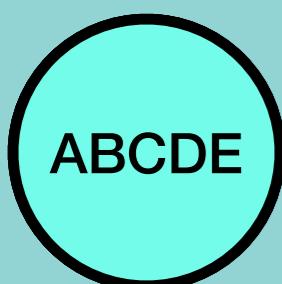


# Kruskal's Algorithm Pseudocode

```
Kruskal(G):
for each vertex:
    makeSet(v)
sort each edge in non decreasing order by weight
for each edge (u, v):
    if findSet(u) ≠ findSet(v):
        union(u, v)
        cost = cost + edge(u,v)
```



$$\text{Cost} = 0 + 5 + 6 + 8 + 15 = 34$$

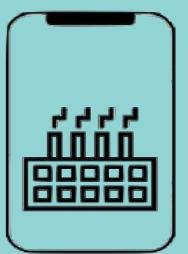


# Time and Space complexity of Kruskal's Algorithm

```
Kruskal(G):
for each vertex: ..... → O(v)
    makeSet(v)
sort each edge in non decreasing order by weight ..... → O(eloge)
for each edge (u, v): ..... → O(e)
    if findSet(u) ≠ findSet(v): ..... → O(1)
    union(u, v) ..... → O(v)
    cost = cost + edge(u, v) ..... → O(1) }
```

**Time complexity :  $O(V+E\log E+EV) = O(E\log(E))$**

**Space complexity :  $O(V+E)$**

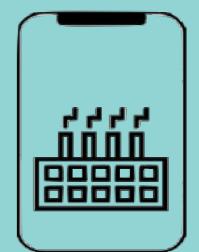
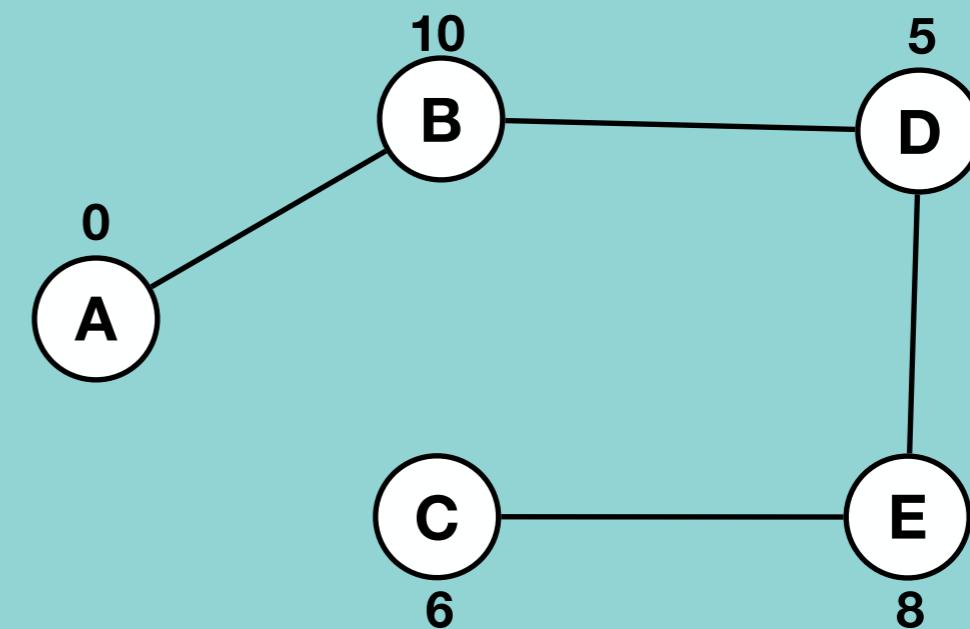
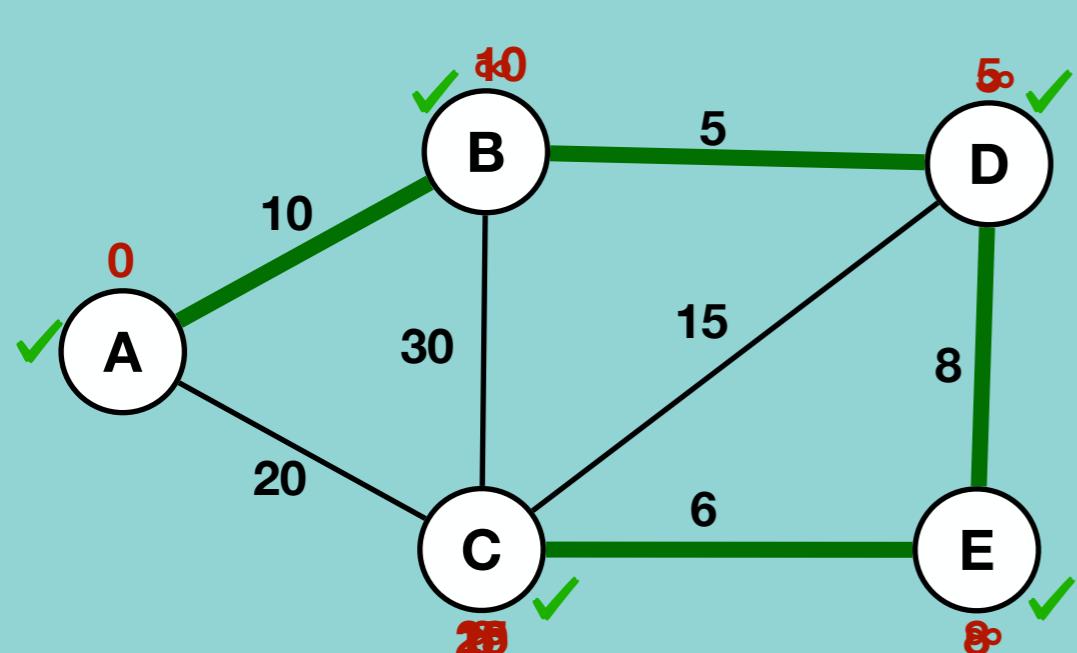


# Prims Algorithm

It is a greedy algorithm

It finds a minimum spanning tree for weighted undirected graphs in following ways

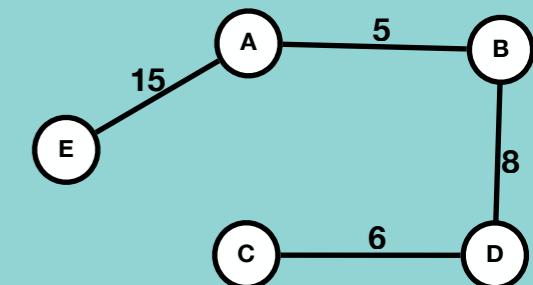
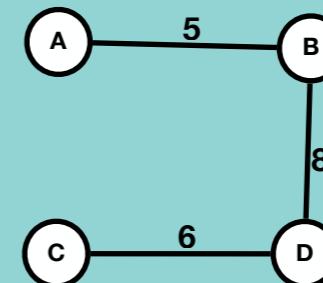
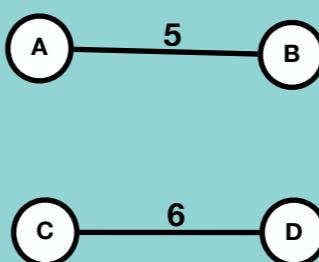
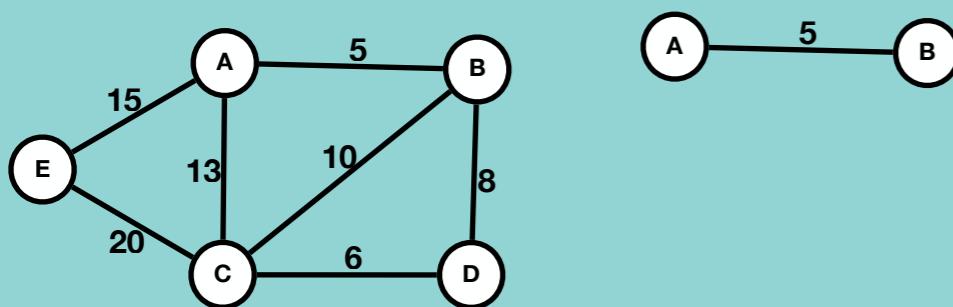
1. Take any vertex as a source set its weight to 0 and all other vertices' weight to infinity
2. For every adjacent vertices if the current weight is more than current edge then we set it to current edge
3. Then we mark current vertex as visited
4. Repeat these steps for all vertices in increasing order of weight



# Kruskal vs Prim's

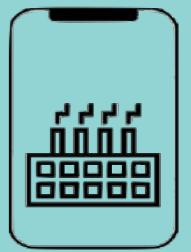
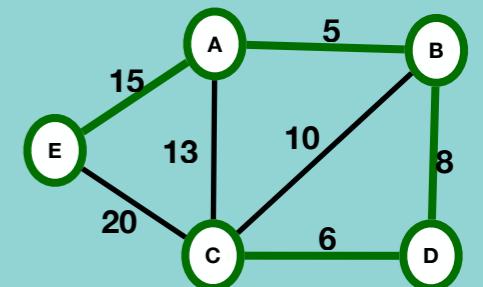
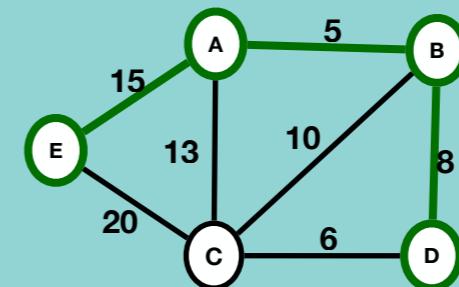
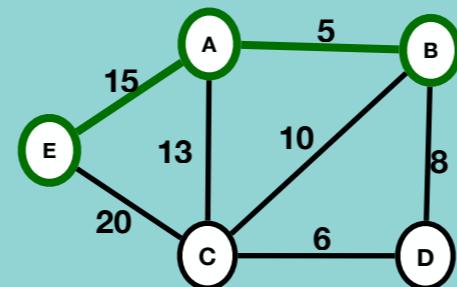
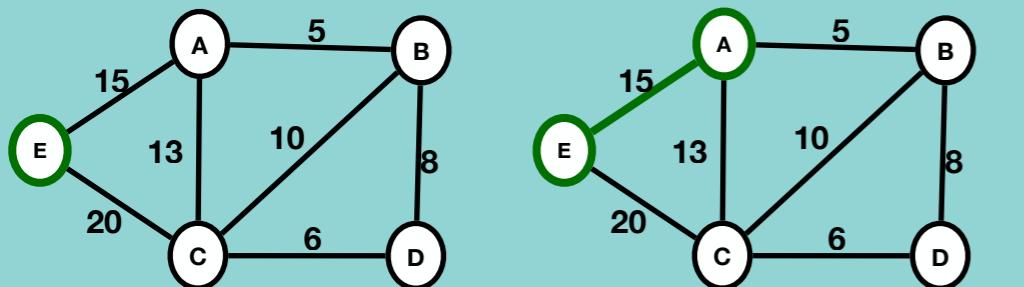
## Kruskal

- Concentrates on Edges
- Finalize edge in each iteration



## Prim's

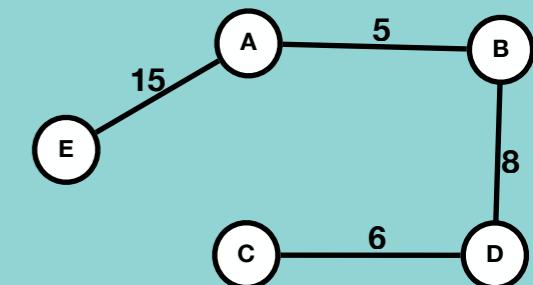
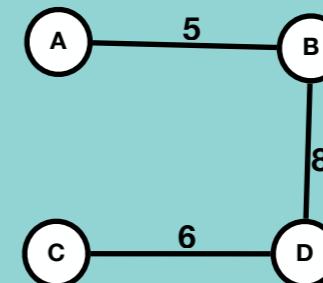
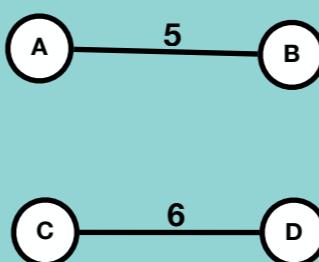
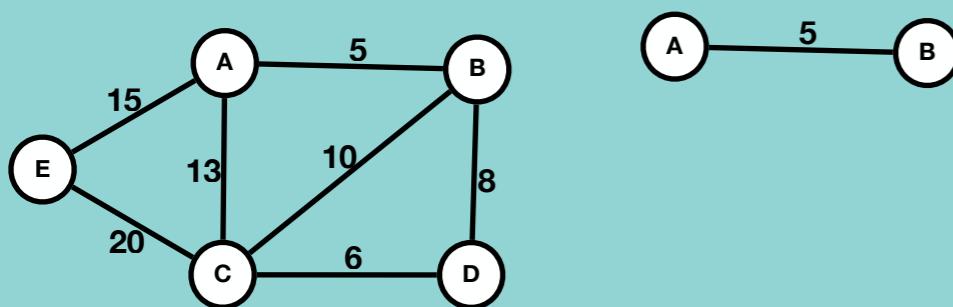
- Concentrates on Vertices
- Finalize Vertex in each iteration



# Kruskal vs Prim's

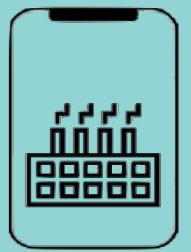
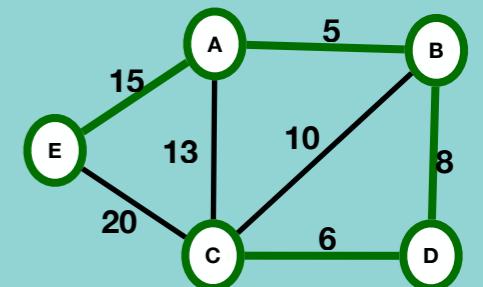
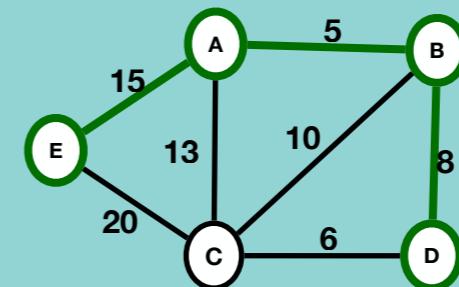
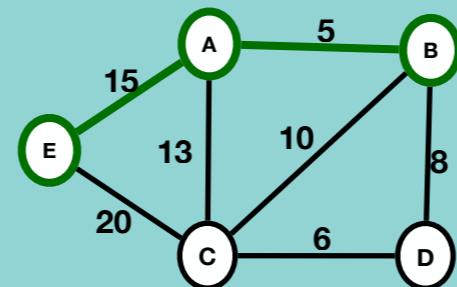
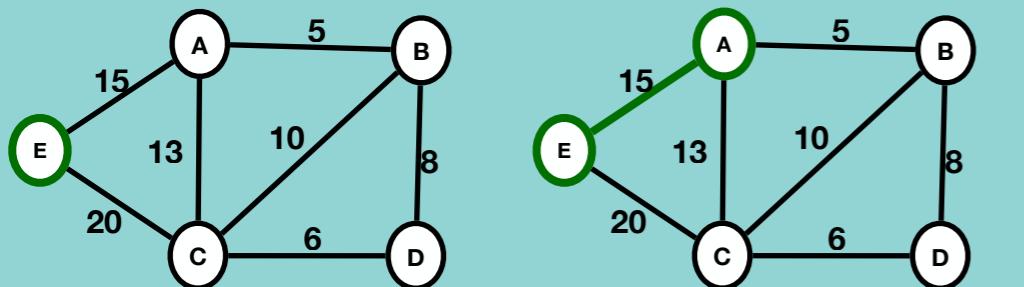
## Kruskal

- Concentrates on Edges
- Finalize edge in each iteration



## Prim's

- Concentrates on Vertices
- Finalize Vertex in each iteration



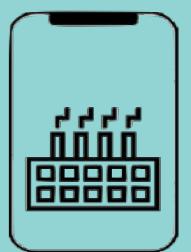
# Kruskal vs Prim's

## Kruskal Applications

- Landing cables
- TV Network
- Tour Operations
- LAN Networks
- A network of pipes for drinking water or natural gas.
- An electric grid
- Single-link Cluster

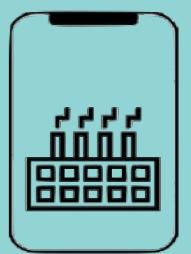
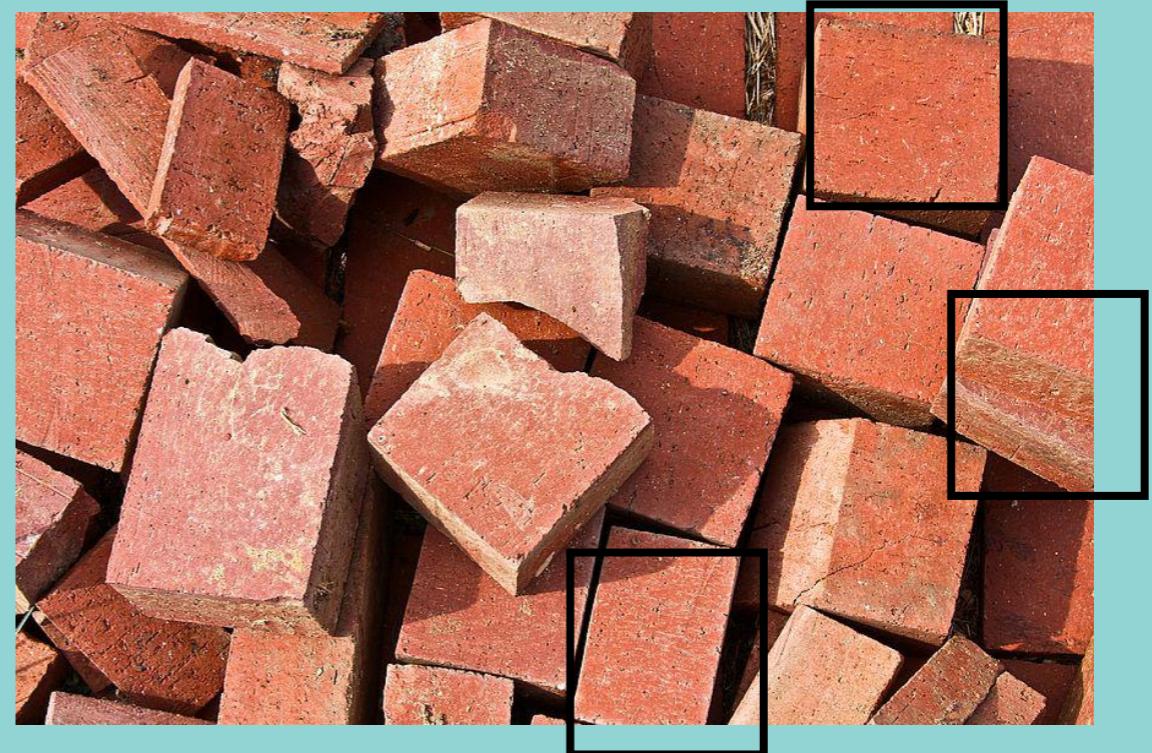
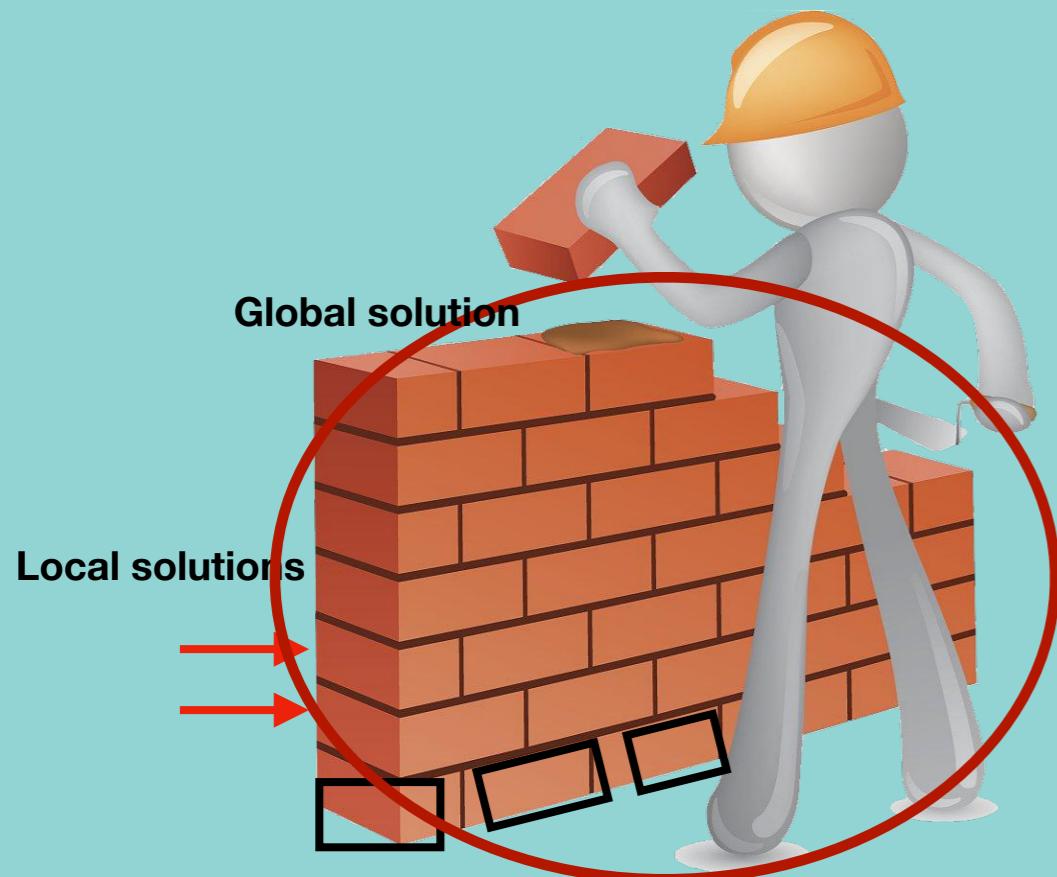
## Prim's Applications

- Network for roads and Rail tracks connecting all the cities.
- Irrigation channels and placing microwave towers
- Designing a fiber-optic grid or ICs.
- Traveling Salesman Problem.
- Cluster analysis.
- Pathfinding algorithms used in AI(Artificial Intelligence).



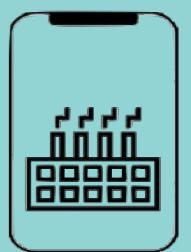
# What is Greedy Algorithm?

- It is an algorithmic paradigm that builds the solution piece by piece
- In each step it chooses the piece that offers most obvious and immediate benefit
- It fits perfectly for those solutions in which local optimal solutions lead to global solution



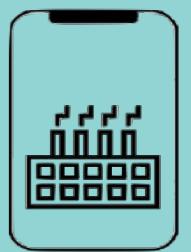
# What is Greedy Algorithm?

- Insertion Sort
  - Selection Sort
  - Topological Sort
  - Prim's Algorithm
  - Kruskal Algorithm
- 
- Activity Selection Problem
  - Coin change Problem
  - Fractional Knapsack Problem



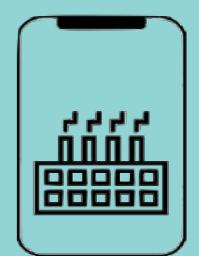
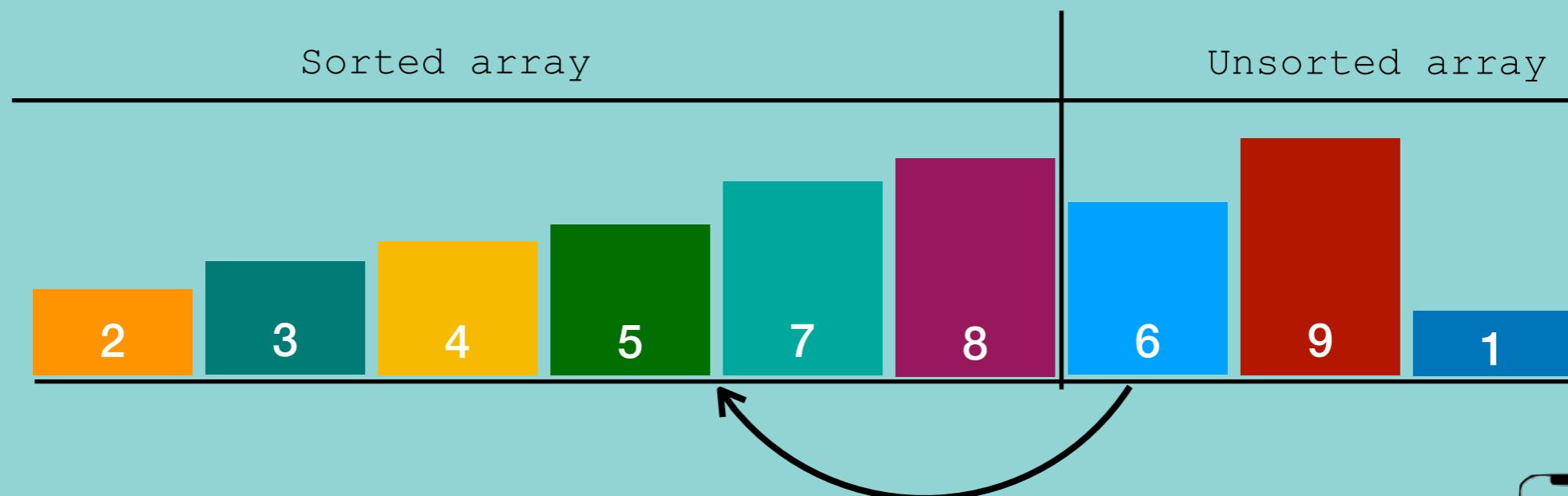
# Greedy Algorithms

- Insertion Sort
- Selection Sort
- Topological Sort
- Prim's Algorithm
- Kruskal Algorithm



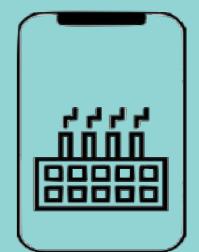
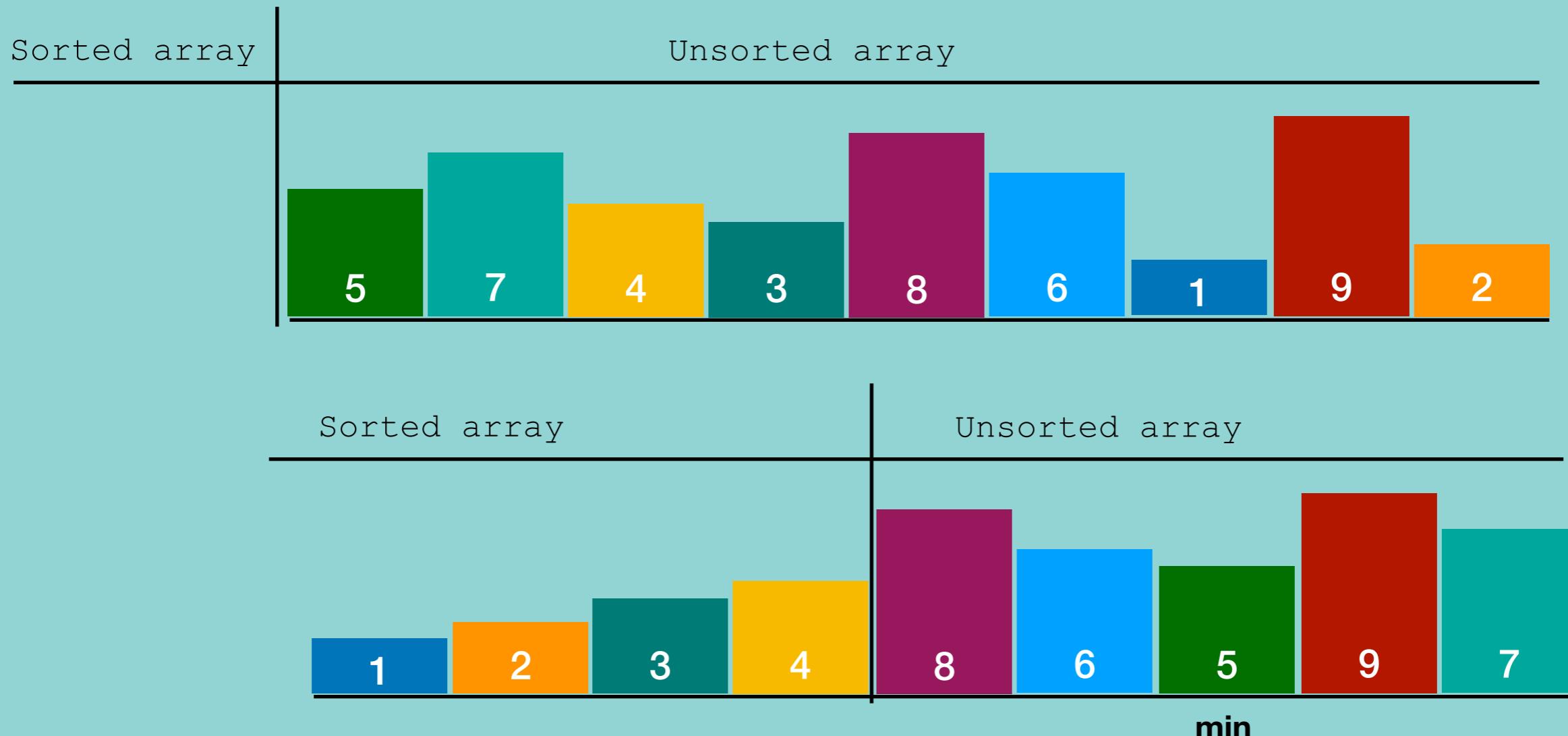
# Greedy Algorithms

## Insertion Sort



# Greedy Algorithms

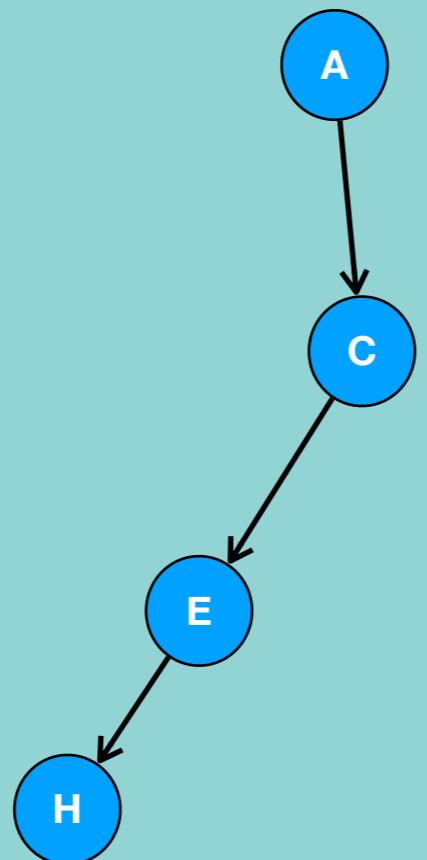
## Selection Sort



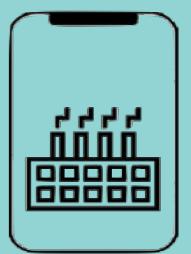
# Greedy Algorithms

## Topological Sort

A C E H



Stack



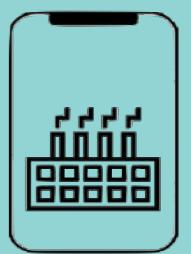
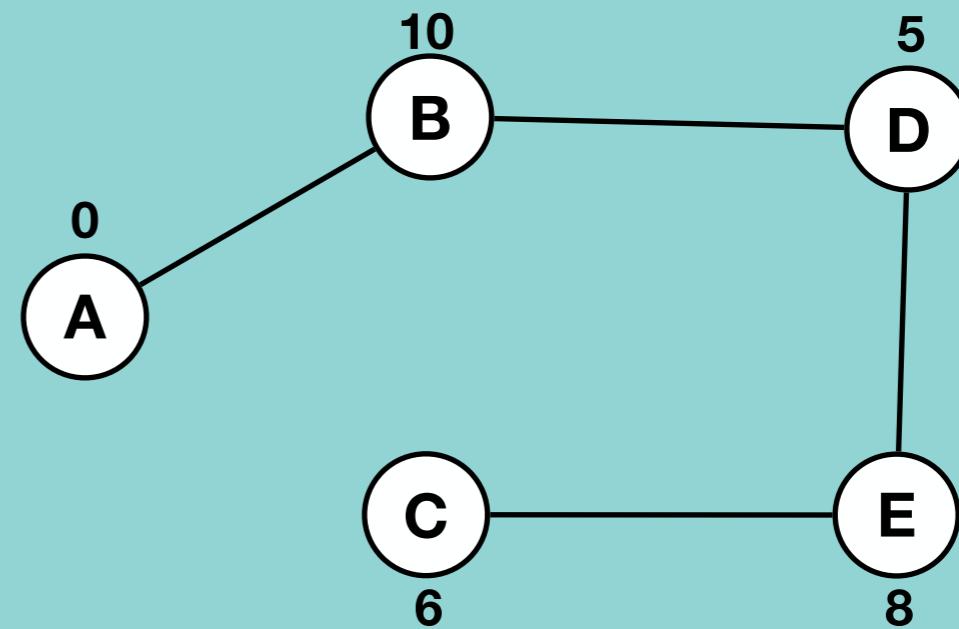
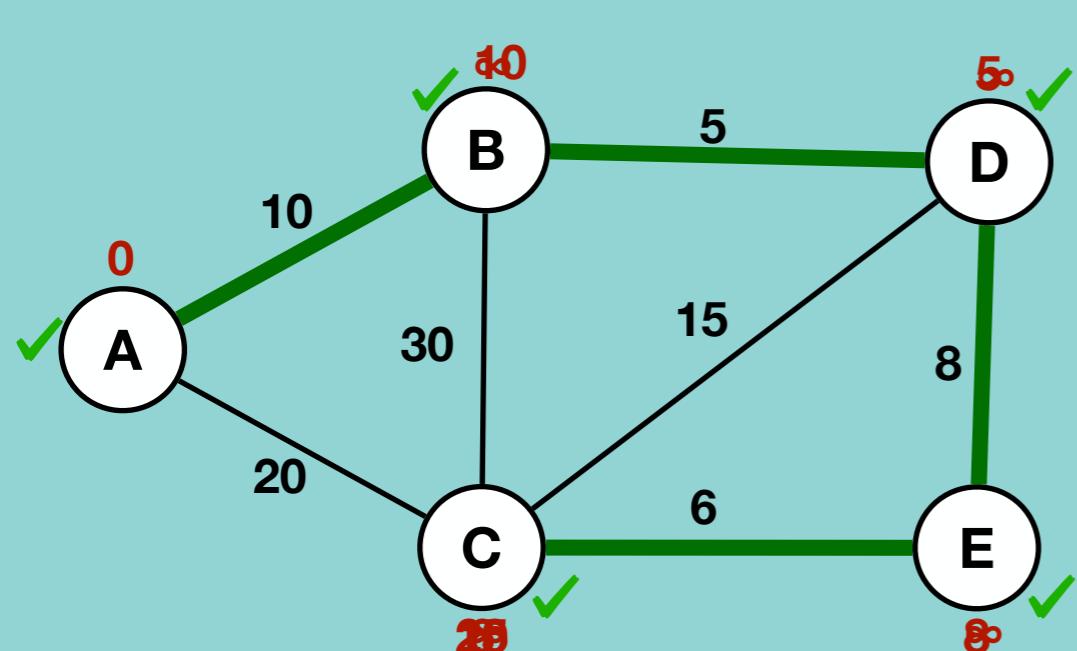
# Greedy Algorithms

## Prims Algorithm

It is a greedy algorithm

It finds a minimum spanning tree for weighted undirected graphs in following ways

1. Take any vertex as a source set its weight to 0 and all other vertices' weight to infinity
2. For every adjacent vertices if the current weight is more than current edge then we set it to current edge
3. Then we mark current vertex as visited
4. Repeat these steps for all vertices in increasing order of weight



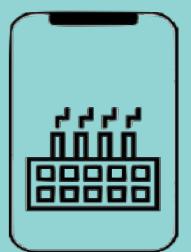
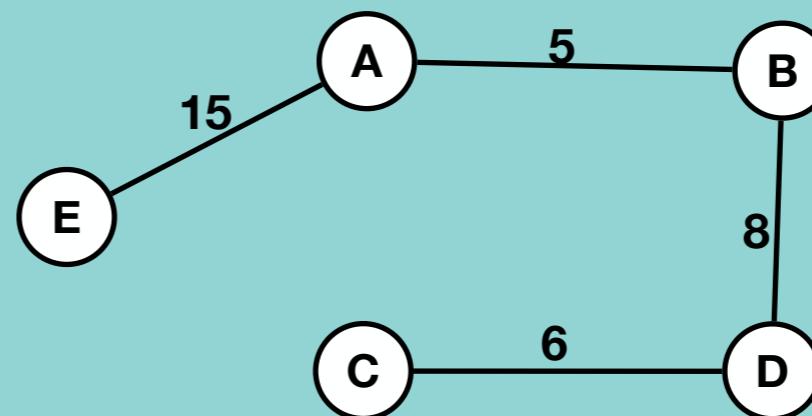
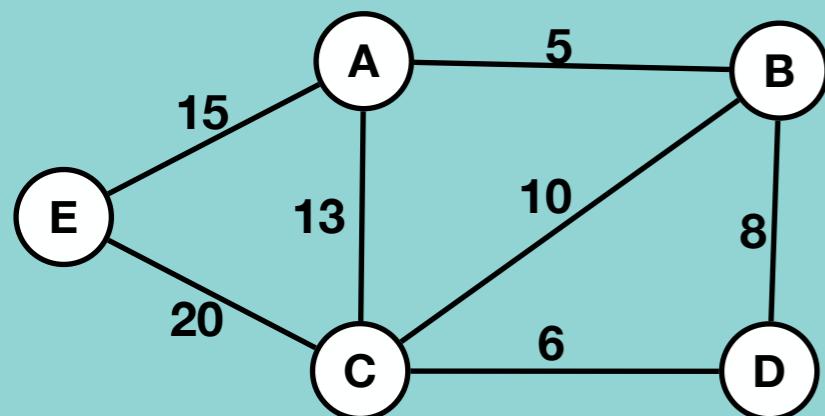
# Greedy Algorithms

## Kruskal's Algorithm

It is a greedy algorithm

It finds a minimum spanning tree for weighted undirected graphs in two ways

- Add increasing cost edges at each step
- Avoid any cycle at each step



# Activity selection problem

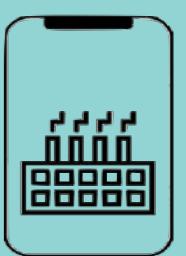
Given N number of activities with their start and end times. We need to select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Activity	A1	A2	A3	A4	A5	A6
Start	0	3	1	5	5	8
Finish	6	4	2	8	7	9

2 Activities

Activity	A3	A2	A1	A5	A4	A6
Start	1	3	0	5	5	8
Finish	2	4	6	7	8	9

4 Activities



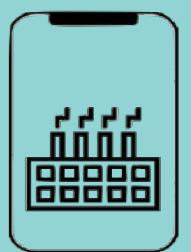
# Activity selection problem

Sort activities based on finish time

Select first activity from sorted array and print it

For all remainin activities:

If the start time of this activity is greater or equal to the finish time of previously selected activity then select this activity and print it



# Coin Change Problem

You are given coins of different denominations and total amount of money. Find the minimum number of coins that you need to make up the given amount.

Infinite supply of denominations : {1,2,5,10,20,50,100,1000}

## Example 1

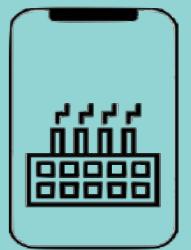
Total amount : 70

Answer: 2 —>  $50 + 20 = 70$

## Example 2

Total amount : 122

Answer: 3 —>  $100 + 20 + 2 = 121$



# Coin Change Problem

You are given coins of different denominations and total amount of money. Find the minimum number of coins that you need to make up the given amount.

Infinite supply of denominations : {1,2,5,10,20,50,100,1000}

Total amount : 2035

$$2035 - 1000 = 1035$$

Result : 1000, 1000, 20, 10, 5

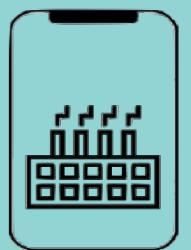
$$1035 - 1000 = 35$$

$$35 - 20 = 15$$

Answer: 5

$$15 - 10 = 5$$

$$5 - 5 = 0$$



# Coin Change Problem

Find the biggest coin that is less than given total number

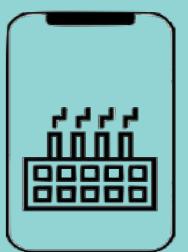
Add coin to the result and subtract coin from total number

If V is equal to zero:

    Then print result

else:

    Repeat Step 2 and 3

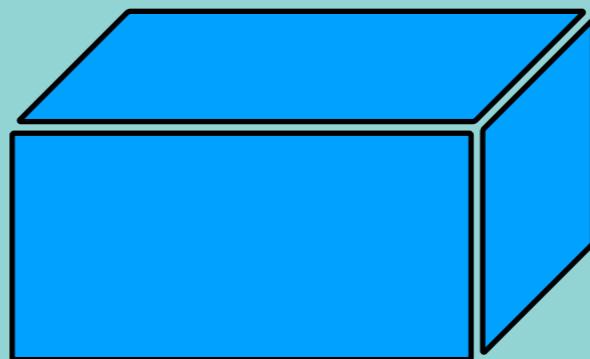


# Fractional Knapsack Problem

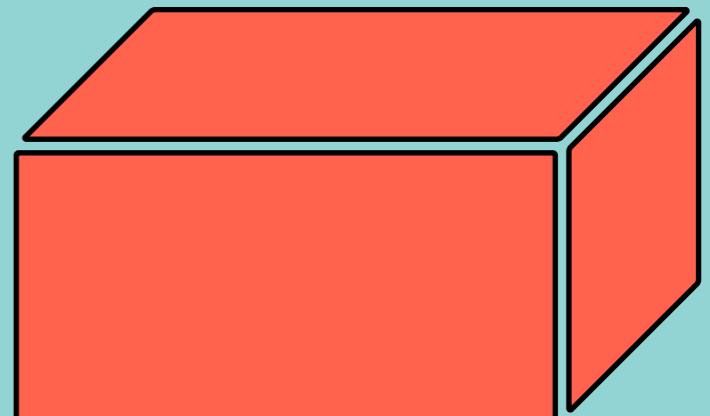
Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



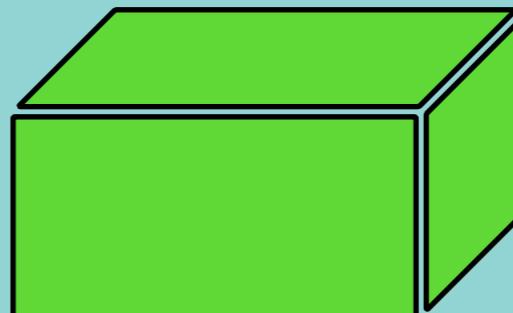
**30 kg**



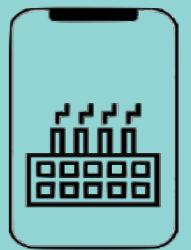
**10 kg, Value : 20**



**20 kg, Value : 10**

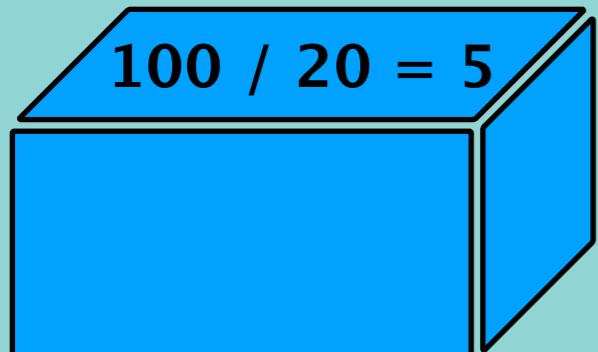


**10 kg, Value : 30**

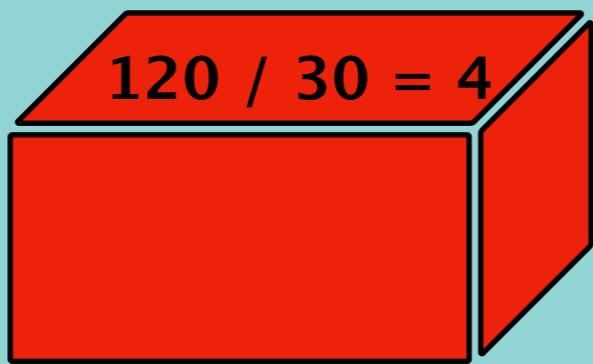


# Fractional Knapsack Problem

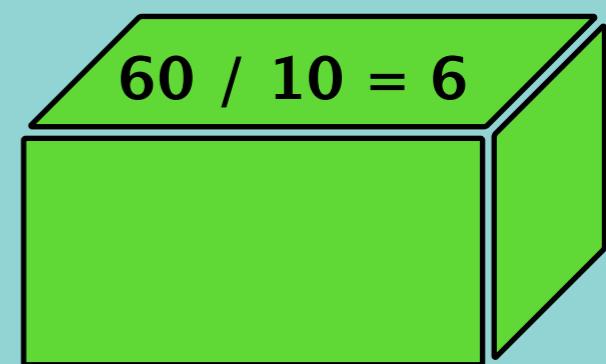
Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



20 kg, Value : 100



30 kg, Value : 120



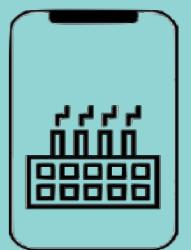
10 kg, Value : 60



50 kg

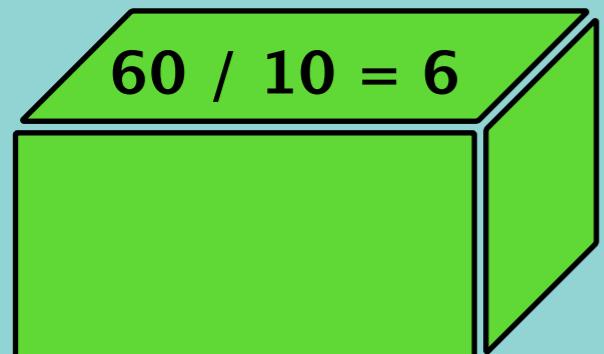
$$100 + 120 = 220$$

$$60 + 100 + 120 \times \frac{2}{3} = 240$$

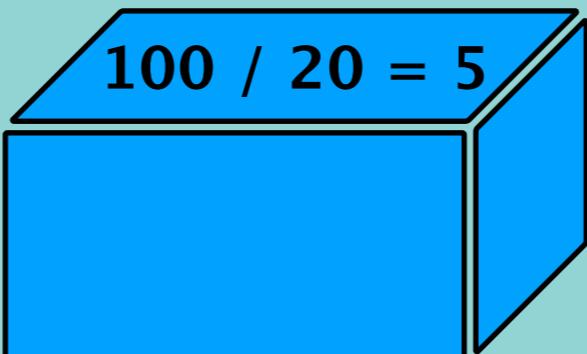


# Fractional Knapsack Problem

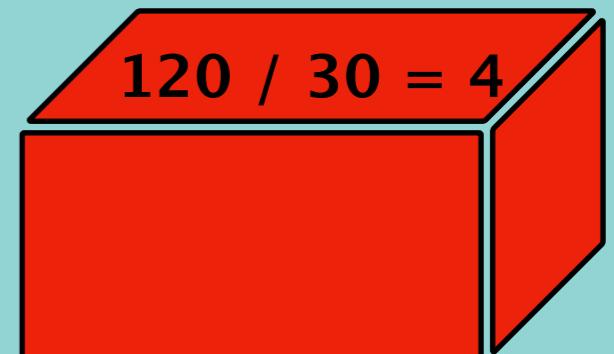
Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



10 kg, Value : 60



20 kg, Value : 100



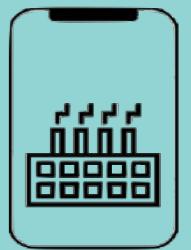
30 kg, Value : 120



50 kg

$$100 + 120 = 220$$

$$60 + 100 + 120 \times \frac{2}{3} = 240$$



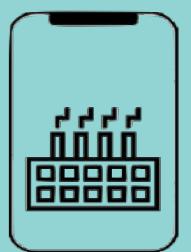
# Fractional Knapsack Problem

Calculate the density or ratio for each item

Sort items based on this ratio

Take items with the highest ratio sequentially until weight allows

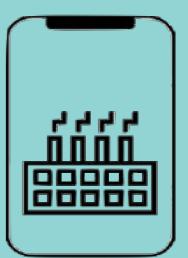
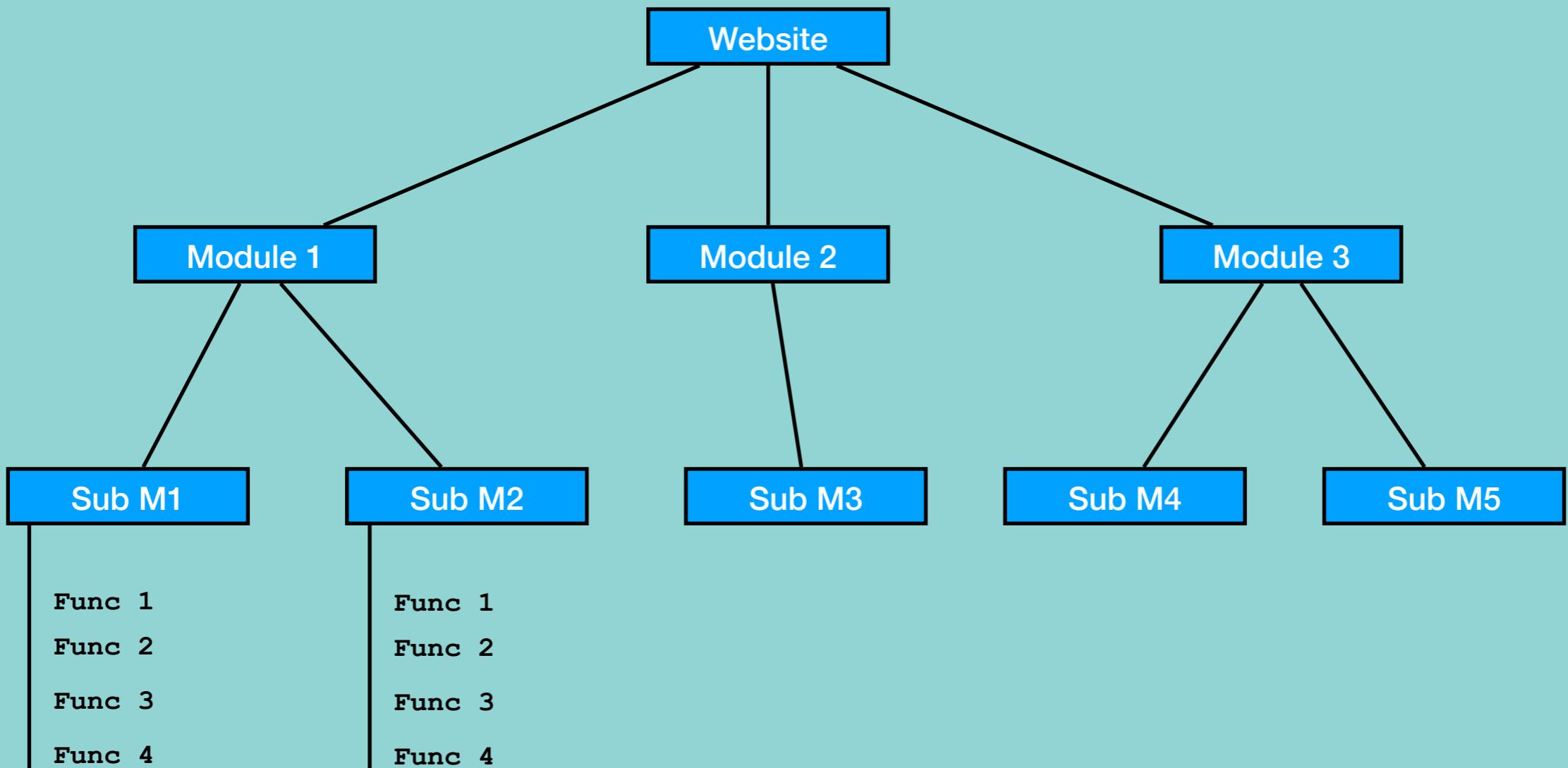
Add the next item as much (fractional) as we can



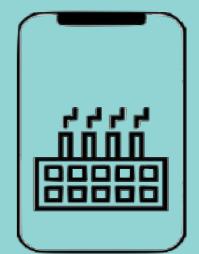
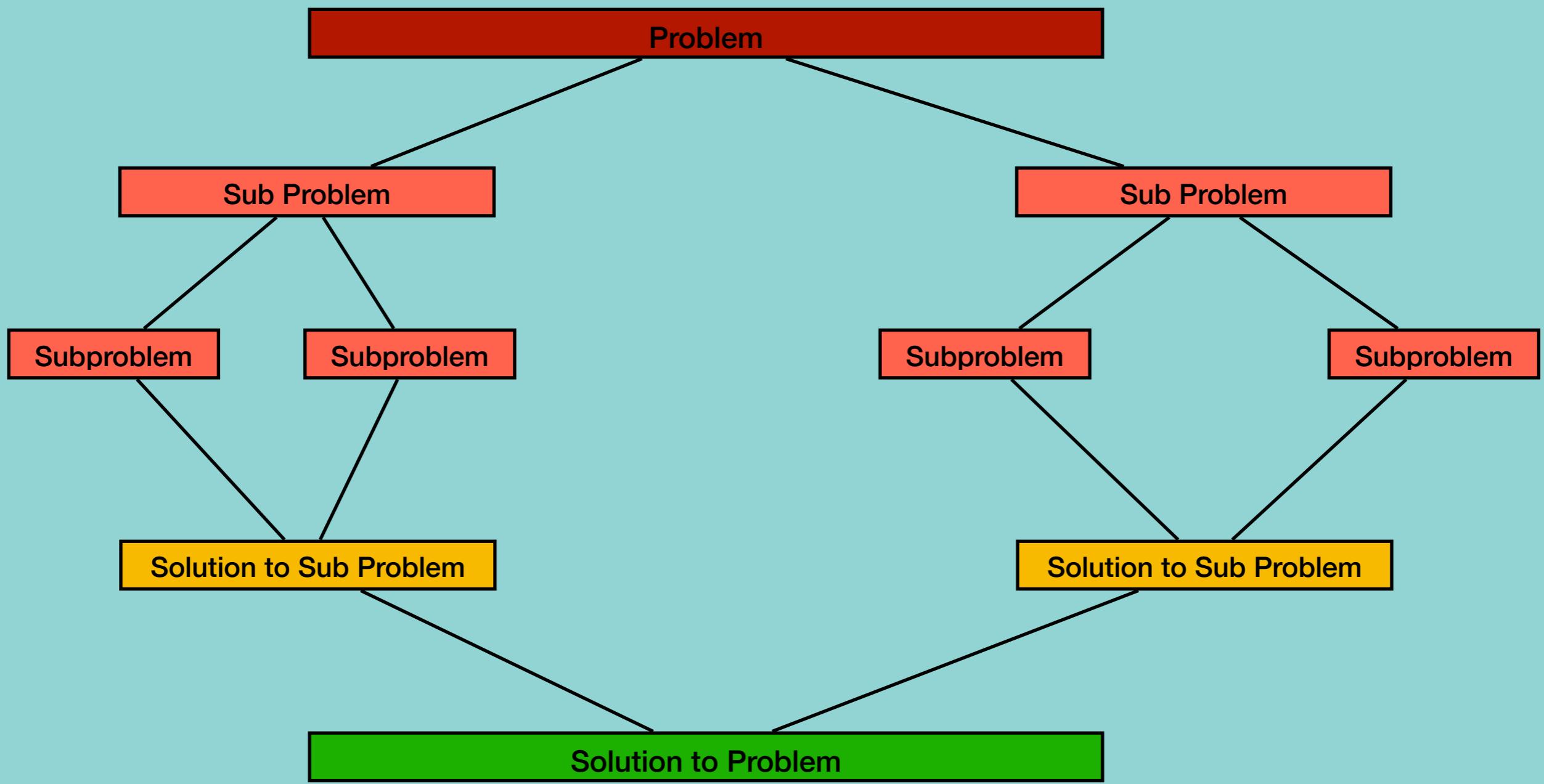
# What is Divide and Conquer Algorithm?

Divide and conquer is an algorithm design paradigm which works by recursively breaking down a problem into subproblems of similar type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.

Example : Developing a website



# What is Divide and Conquer Algorithm?



# Property of Divide and Conquer Algorithm

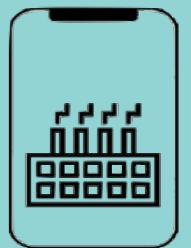
## Optimal Substructure:

If any problem's overall optimal solution can be constructed from the optimal solutions of its subproblem then this problem has optimal substructure

Example:  $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

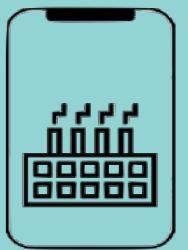
## Why we need it?

It is very effective when the problem has optimal substructure property.



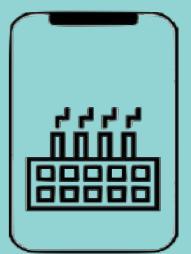
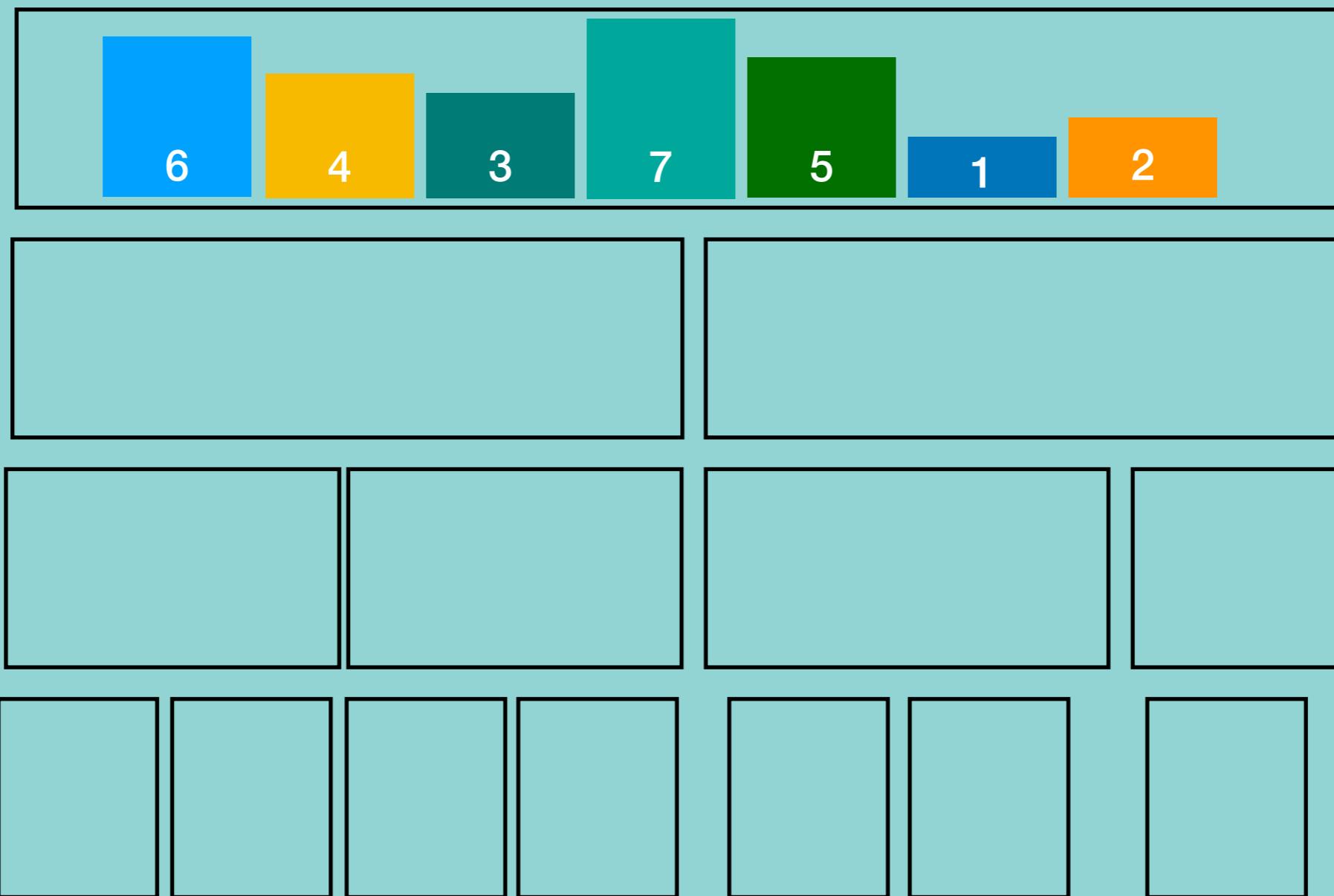
# Common Divide and Conquer Algorithms

## Merge Sort



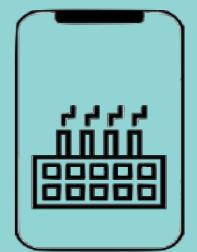
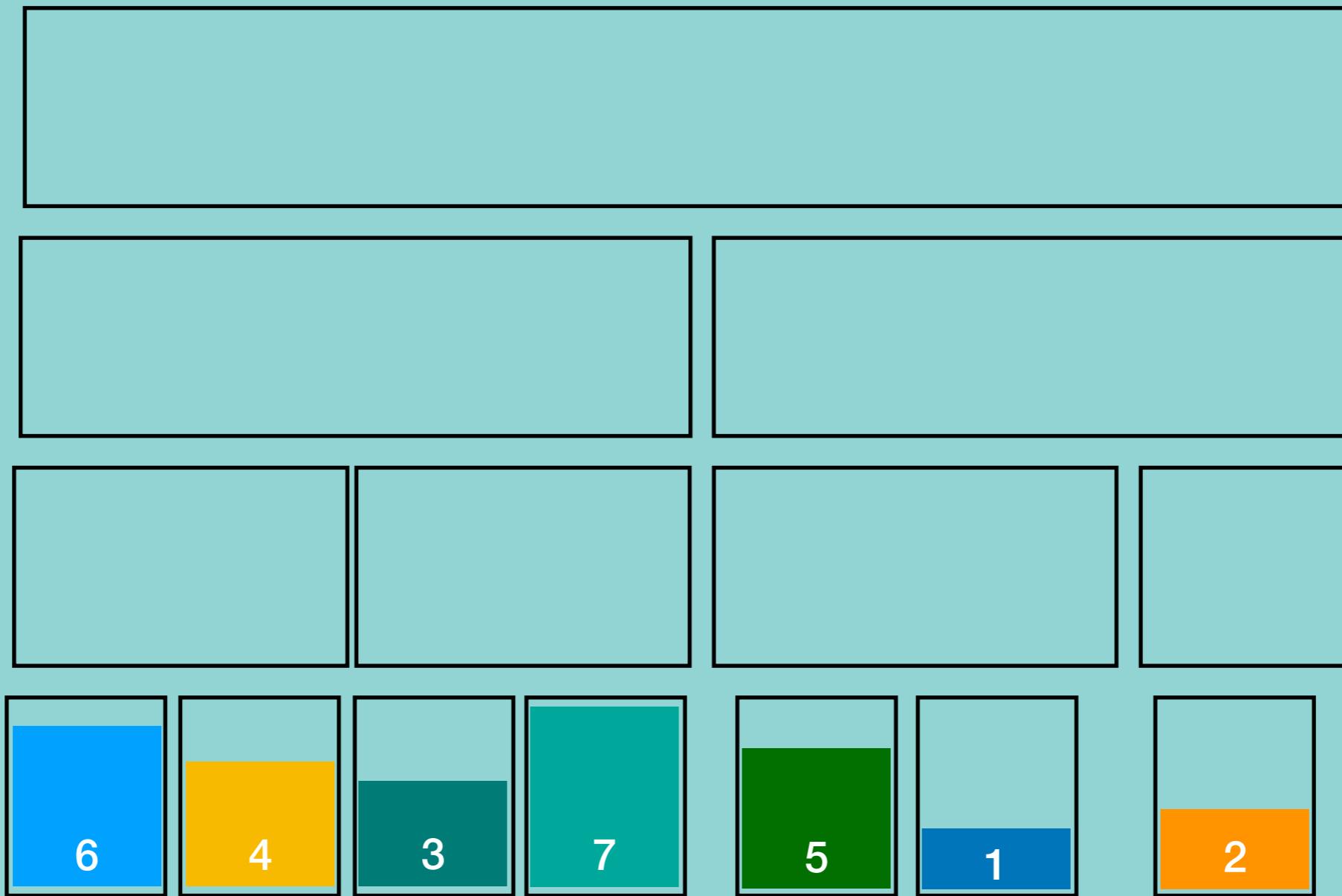
# Common Divide and Conquer Algorithms

Merge Sort



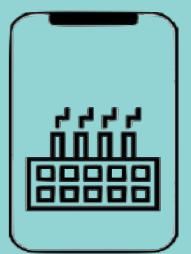
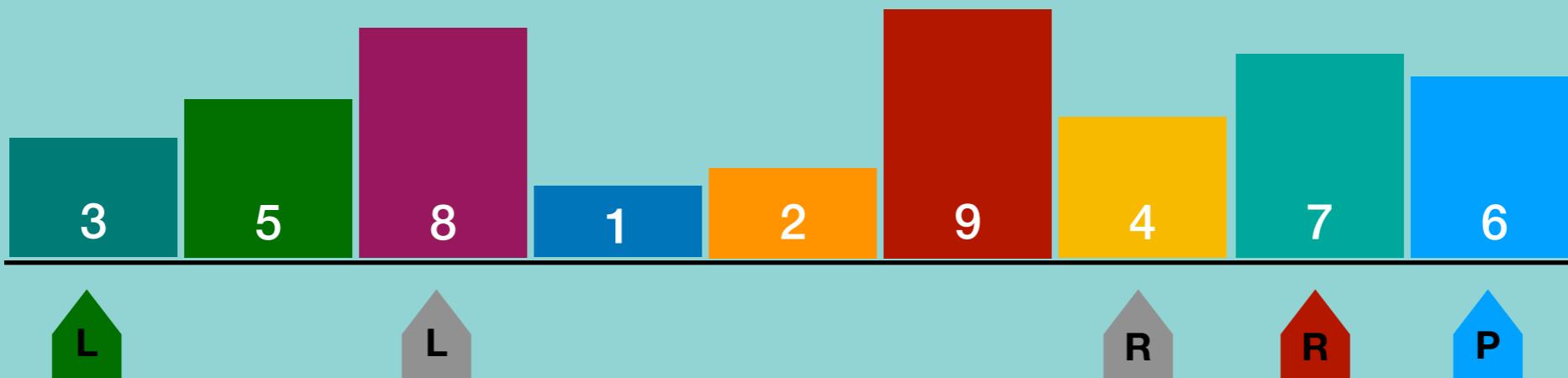
# Common Divide and Conquer Algorithms

## Merge Sort



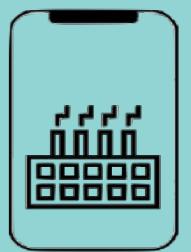
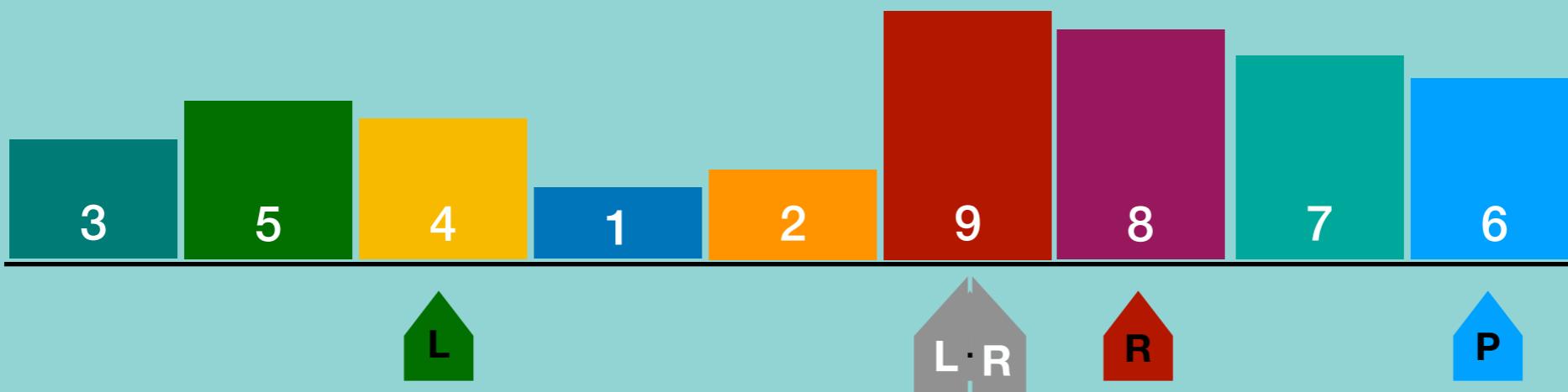
# Common Divide and Conquer Algorithms

## Quick Sort



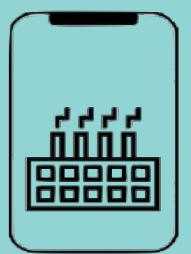
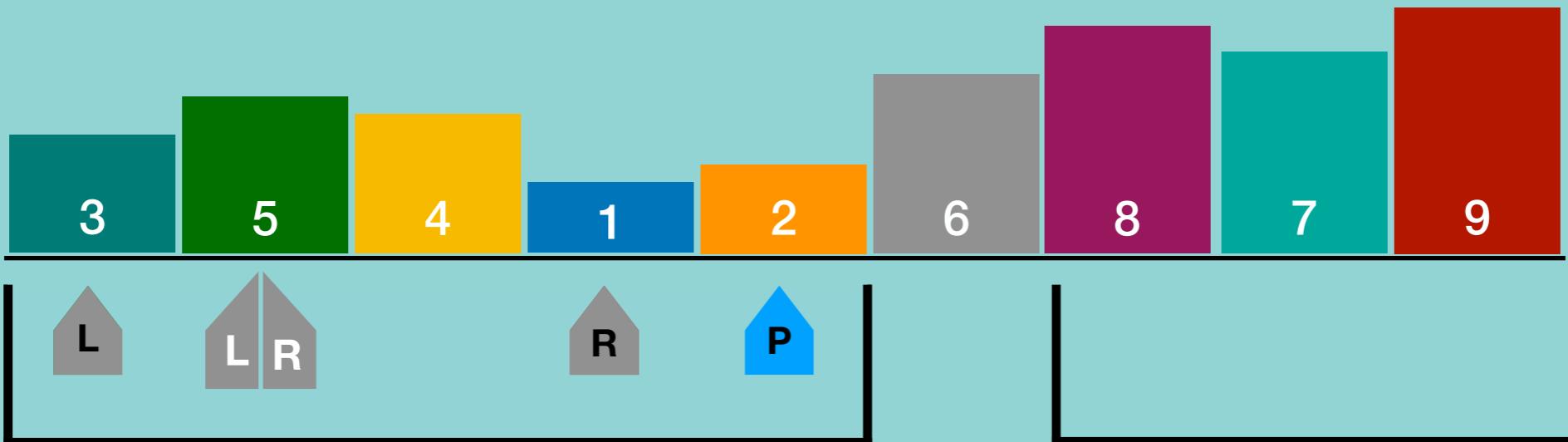
# Common Divide and Conquer Algorithms

## Quick Sort



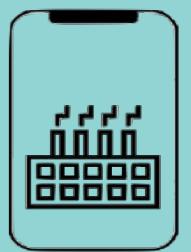
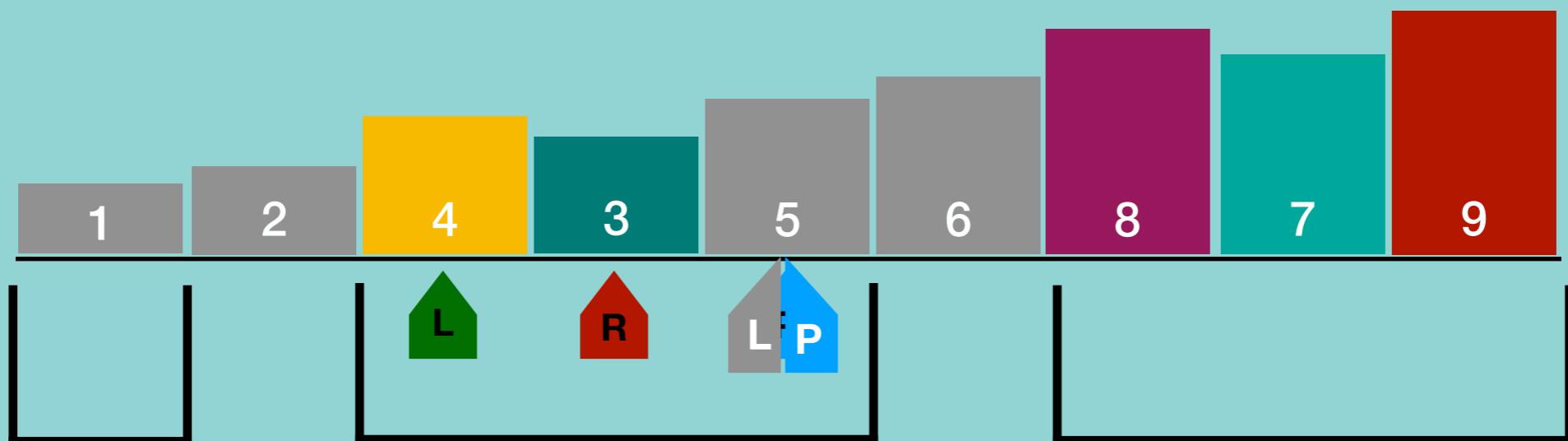
# Common Divide and Conquer Algorithms

## Quick Sort



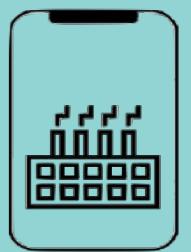
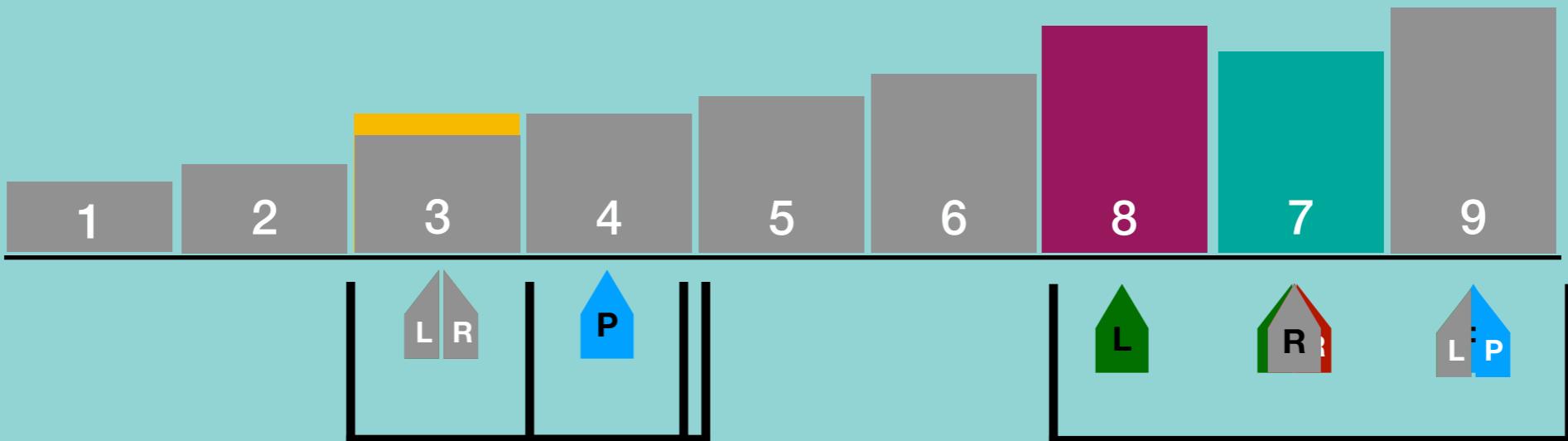
# Common Divide and Conquer Algorithms

## Quick Sort



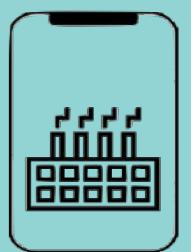
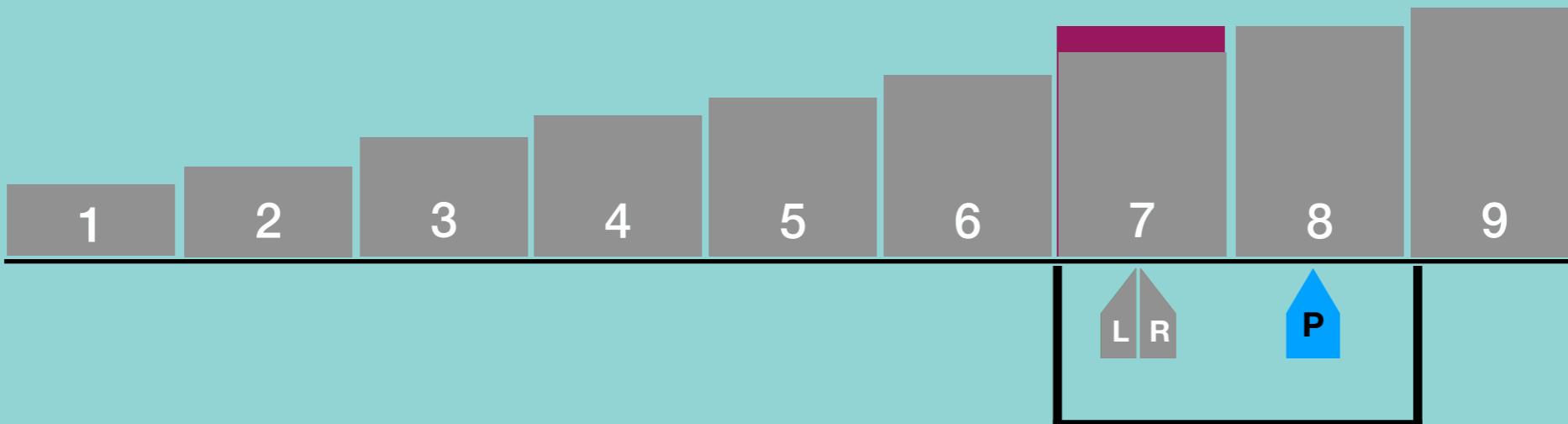
# Common Divide and Conquer Algorithms

## Quick Sort



# Common Divide and Conquer Algorithms

## Quick Sort

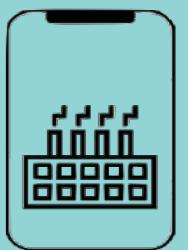


# Fibonacci Series

**Definition :** a series of numbers in which each number is the sum of the two preceding numbers. First two numbers by definition are 0 and 1.

**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

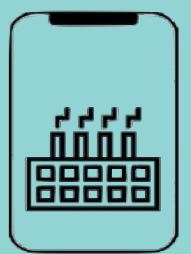
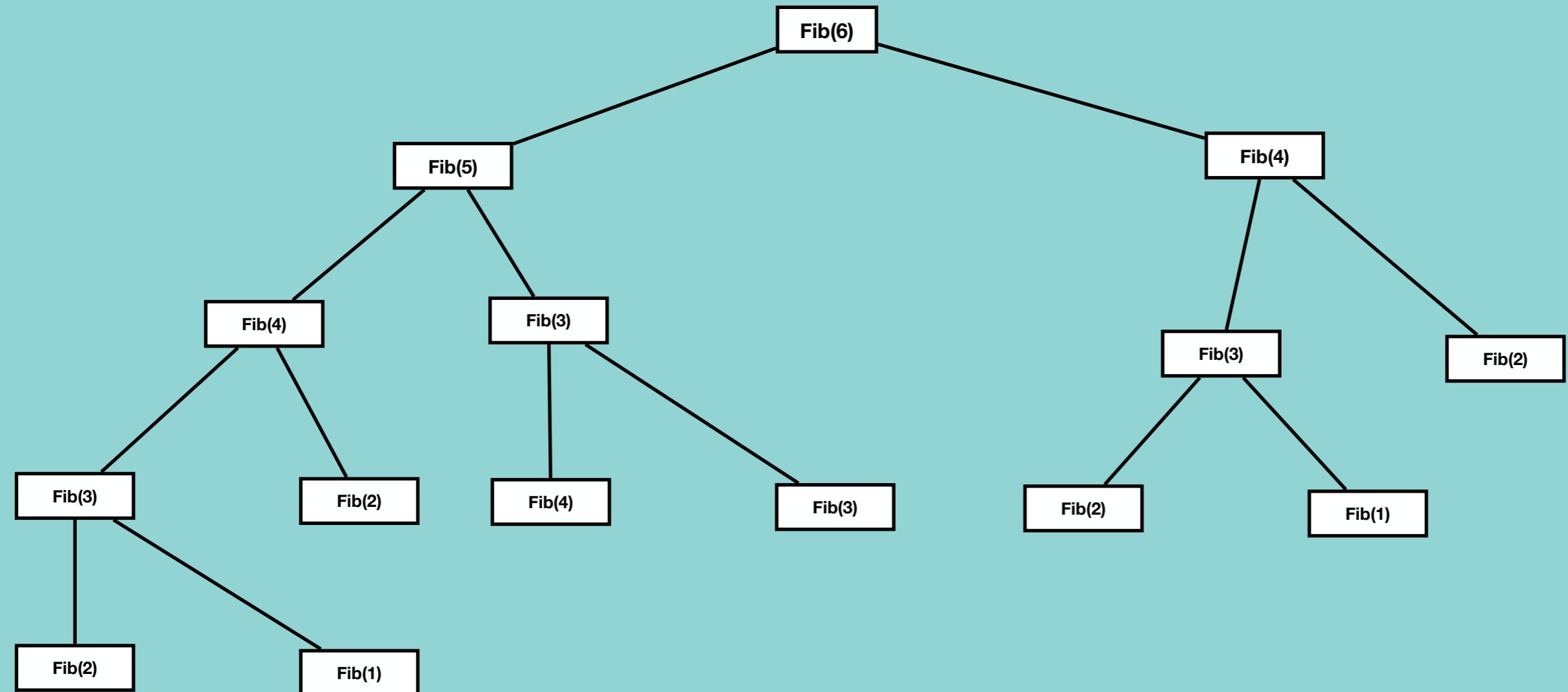
$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$



# Fibonacci Series

**Definition :** a series of numbers in which each number is the sum of the two preceding numbers. First two numbers by definition are 0 and 1.

**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....



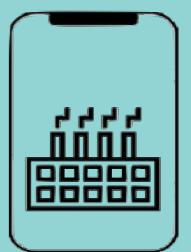
# Fibonacci Series

**Definition :** a series of numbers in which each number is the sum of the two preceding numbers. First two numbers by definition are 0 and 1.

**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$

```
Fibonacci(N):  
    If n < 1 return error message  
    If n = 1 return 0  
    If n = 2 return 1  
    Else  
        return Fibonacci(N-1) + Fibonacci(N-2)
```



# Number Factor

## Problem Statement:

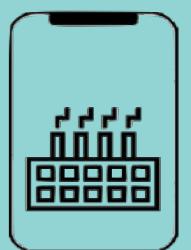
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

### Example 1

- N = 4
- Number of ways = 4
- Explanation : There are 4 ways we can express N. {4},{1,3},{3,1},{1,1,1,1}

### Example 2

- N = 5
- Number of ways = 6
- Explanation : There are 6 ways we can express N. {4,1},{1,4},{1,3,1},{3,1,1},{1,1,3},{1,1,1,1,1}



# Number Factor

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

## **Example:**

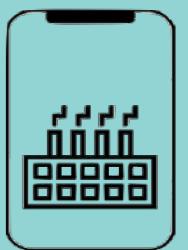
- $N = 5$
- Number of ways = 6
- Explanation : There are 6 ways we can express N.  $\{4,1\}, \{1,4\}, \{1,3,1\}, \{3,1,1\}, \{1,1,3\}, \{1,1,1,1,1\}$

$$f(4) \quad \{4\}, \{1,3\}, \{3,1\}, \{1,1,1,1\}$$

$$f(4) + 1 \quad \{4, 1\}, \{1,3,1\}, \{3,1,1\}, \{1,1,1,1,1\}$$

$$f(2) + 3 \quad \{1,1\} \quad \{1,1,3\}$$

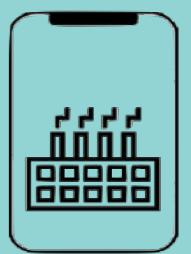
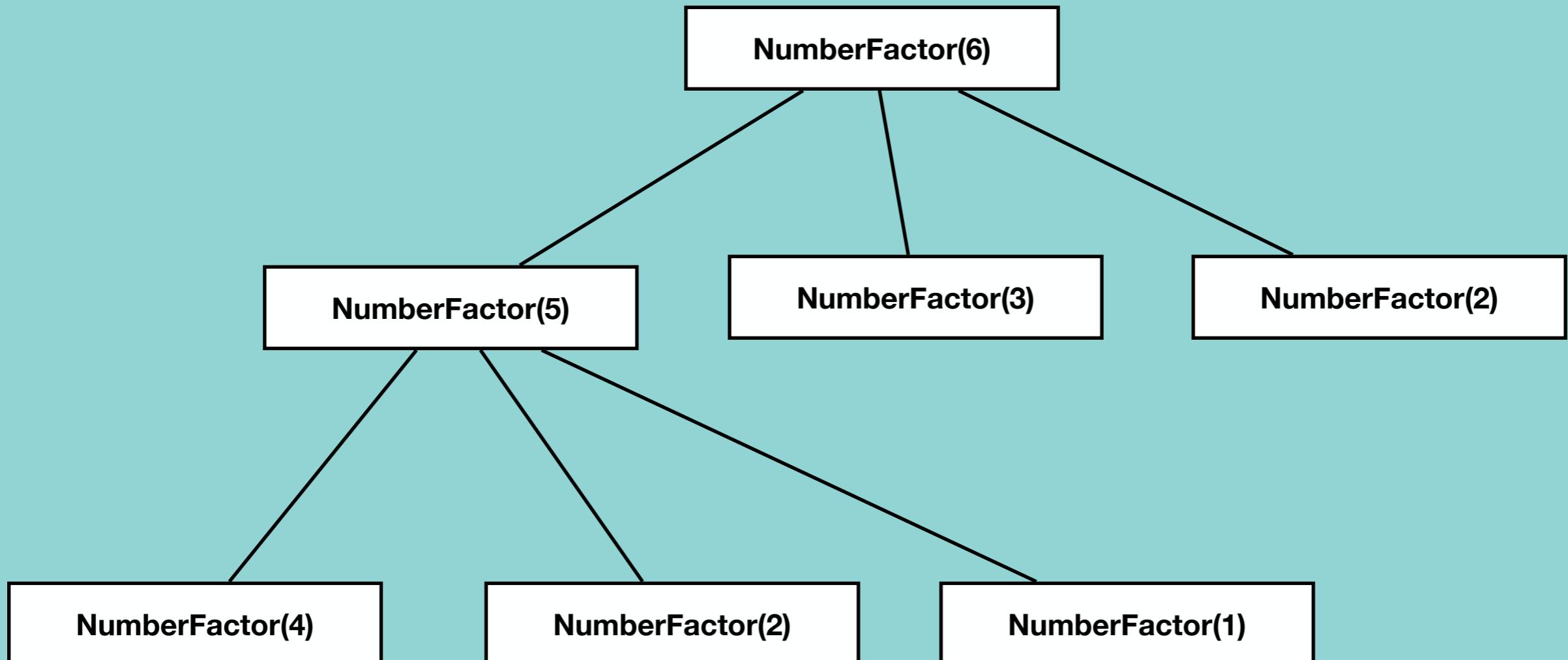
$$f(1) + 4 \quad \{1\} \quad \{1,4\}$$



# Number Factor

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

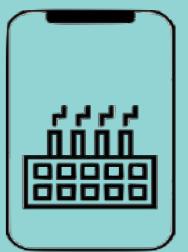


# Number Factor

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N):  
    If N in (0,1,2) return 1  
    If N = 3 return 2  
    Else  
        return NumberFactor(N-1) + NumberFactor(N-3) + NumberFactor(N-4)
```



# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

## Example 1



## Answer

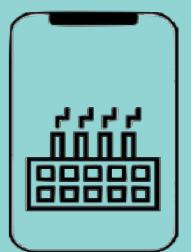
- Maximum amount = 41
- Houses that are stolen : 7, 30, 4

$$\text{Option1} = 6 + f(5)$$



$$\text{Max}(\text{Option1}, \text{Option2})$$

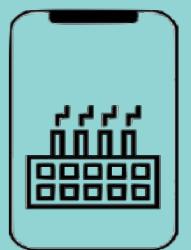
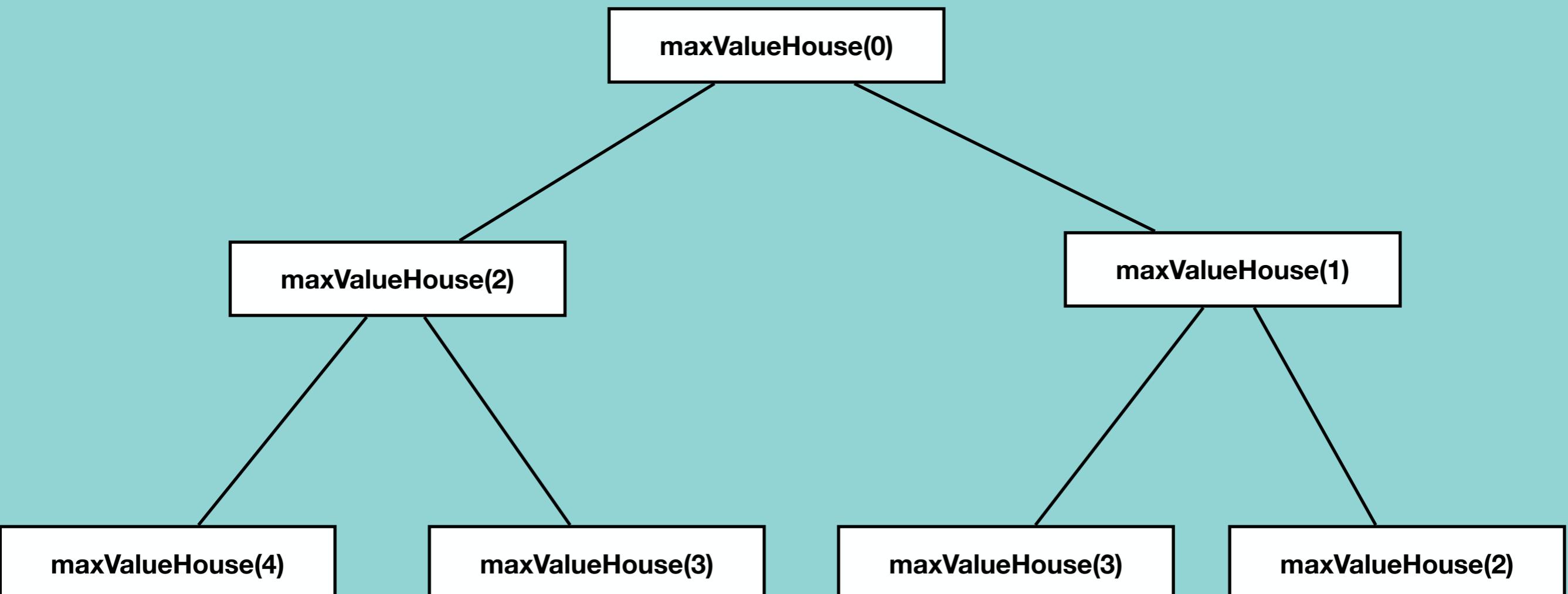
$$\text{Option2} = 0 + f(6)$$



# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

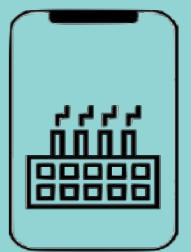


# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

```
maxValueHouse(houses, currentHouse):  
    If currentHouse > length of houses  
        return 0  
    Else  
        stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)  
        skipFirstHouse = maxValueHouse(houses, currentHouse+1)  
        return max(stealFirstHouse, skipFirstHouse)
```



# Convert String

## Problem Statement:

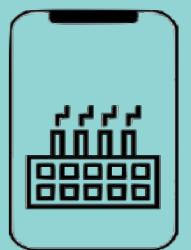
- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

## Example 1

- S1 = “catch”
- S2 = “carch”
- Output = 1
- Explanation : Replace “r” with “t”

## Example 2

- S1 = “table”
- S2 = “tbres”
- Output = 3
- Explanation : Insert “a” to second position, replace “r” with “l” and delete “s”



# Convert String

## Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

## Example

- S1 = “table”
- S2 = “tgable”

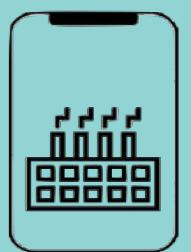
Delete      →       $f(2, 3)$

- S1 = “table”
- S2 = “tble”

Insert      →       $f(3, 2)$

- S1 = “table”
- S2 = “tcble”

Replace      →       $f(3, 3)$



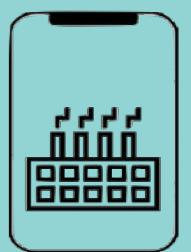
# Convert String

## Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

```
findMinOperation(s1, s2, index1, index2):
    If index1 == len(s1)
        return len(s2)-index2
    If index2 == len(s2)
        return len(s1)-index1
    If s1[index1] == s2[index2]
        return findMinOperation(s1, s2, index1+1, index2+1)

    Else
        deleteOp = 1 + findMinOperation(s1, s2, index1, index2+1)
        insertOp = 1 + findMinOperation(s1, s2, index1+1, index2)
        replaceOp = 1 + findMinOperation(s1, s2, index1+1, index2+1)
        return min(deleteOp, insertOp, replaceOp)
```



# Zero One Knapsack Problem

## Problem Statement:

- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken

## Example 1



**Mango**  
Weight : 3  
Profit : 31



**Apple**  
Weight : 1  
Profit : 26



**Orange**  
Weight : 2  
Profit : 17

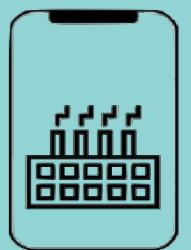


**Banana**  
Weight : 5  
Profit : 72

**Knapsack Capacity : 7**

## Answer Combinations

- Mango (W:3, P:31) + Apple (W:1,P:26) + Orange (W:2, P:17) = W:6, Profit:74
- Orange (W:2, P:17) + Banana (W:5,P:72) = W:7, Profit:89
- Apple (W:1,P:26) + Banana (W:5,P:72) = W:6, Profit:98



# Zero One Knapsack Problem

## Problem Statement:

- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken

## Example 1



**Mango**  
Weight : 3  
Profit : 31



**Apple**  
Weight : 1  
Profit : 26



**Orange**  
Weight : 2  
Profit : 17



**Banana**  
Weight : 5  
Profit : 72

**Knapsack Capacity : 7**

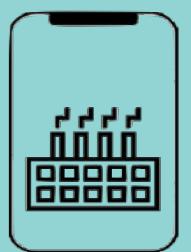
## Subproblems:

$$\text{Option1} = 31 + f(2,3,4)$$



**Max(Option1, Option2)**

$$\text{Option2} = 0 + f(2,3,4)$$

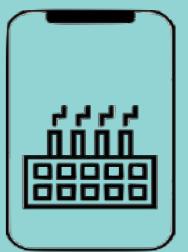


# Zero One Knapsack Problem

## Problem Statement:

- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken

```
zoKnapsack(items, capacity, currentIndex):
    If capacity<=0 or currentIndex<0 or currentIndex>len(profits)
        return 0
    Elif currentItemWeight <= capacity
        Profit1 = curentItemProfit + zoKnapsack(items, capacity- currentItemsWeight, nextItem)
        Profit2 = zoKnapsack(items, capacity, nextItem)
        return max(Profit1, Profit2)
    Else
        return 0
```



# The Longest Common Subsequence (LCS)

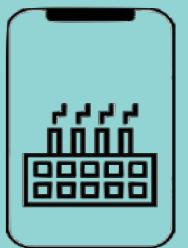
## Problem Statement:

- S1 and S2 are given strings
- Find the length of the longest subsequence which is common to both strings
- Subsequence: a sequence that can be derived from another sequence by deleting some elements without changing the order of them

ABCDE	ACE	ABCE
	ADE	ABDE
		ACB

## Example

- S1 = “elephant”
- S2 = “erepat”
- Output = 5
- Longest String : “eepat”



# The Longest Common Subsequence (LCS)

## Problem Statement:

- S1 and S2 are given strings
- Find the length of the longest subsequence which is common to both strings
- Subsequence: a sequence that can be derived from another sequence by deleting some elements without changing the order of them

## Example

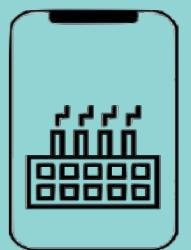
- S1 = “elephant”
- S2 = “erepat”
- Output = 5
- Longest String : “eepat”

## Subproblems:

**Option1** = 1 + f(2,8 : 2,7)

**Option2** = 0 + f(3,8 : 2,7) —————→ Max(**Option1**, **Option2**, **Option3**)

**Option3** = 0 + f(2,8 : 3,7)



# The Longest Common Subsequence (LCS)

## Problem Statement:

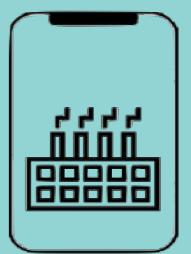
- S1 and S2 are given strings
- Find the length of the longest subsequence which is common to both strings
- Subsequence: a sequence that can be derived from another sequence by deleting some elements without changing the order of them

## Example

- S1 = “elephant”
- S2 = “erepat”
- Output = 5
- Longest String : “eepat”

```
findCLS(s1, s2, index1, index2):
    If index1 == len(s1) or index2 == len(s2):
        return 0
    If s1[index1] == s2[index2]
        return 1 + findCLS(s1, s2, index1+1, index2+1)

    Else
        op1 = findCLS(s1, s2, index1, index2+1)
        op2 = findCLS(s1, s2, index1+1, index2)
    return max(op1, op2)
```



# The Longest Palindromic Subsequence (LPS)

## **Problem Statement:**

- S is a given string
- Find the longest palindromic subsequence (LPS)
- Subsequence: a sequence that can be derived from another sequence by deleting some elements without changing the order of them
- Palindrome is a string that reads the same backward as well as forward

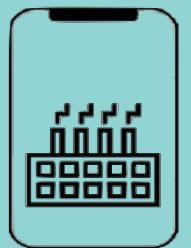
MADAM

## **Example 1**

- S = “ELRMENMET”
- Output = 5
- LPS: “EMEME”

## **Example 2**

- S = “AMEEWMEA”
- Output = 6
- LPS: “AMEEMA”



# The Longest Palindromic Subsequence (LPS)

## **Problem Statement:**

- S is a given string
- Find the longest palindromic subsequence (LPS)
- Subsequence: a sequence that can be derived from another sequence by deleting some elements without changing the order of them
- Palindrome is a string that reads the same backward as well as forward

## **Example 1**

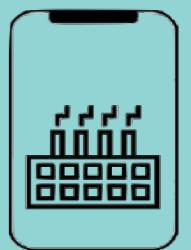
- S = “ELRMENMET”
- Output = 5
- LPS: “EMEME”

## **Subproblems:**

$$\text{Option1} = 2 + f(2,8)$$

$$\text{Option2} = 0 + f(1,8) \quad \longrightarrow \text{Max}(\text{Option1}, \text{Option2}, \text{Option3})$$

$$\text{Option3} = 0 + f(2,9)$$



# The Longest Palindromic Subsequence (LPS)

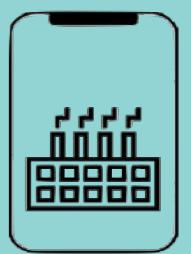
## **Problem Statement:**

- S is a given string
- Find the longest palindromic subsequence (LPS)
- Subsequence: a sequence that can be derived from another sequence by deleting some elements without changing the order of them
- Palindrome is a string that reads the same backward as well as forward

## **Example 1**

- S = “ELRMENMET”
- Output = 5
- LPS: “EMEME”

```
findLPS(S, startIndex, endIndex):  
    If startIndex > endIndex:  
        return 0  
    If s1[startIndex] == s2[endIndex]  
        return 2 + findLPS(s, startIndex+1, endIndex+1)  
  
    Else  
        op1 = findCLS(s, startIndex, endIndex-1)  
        op2 = findCLS(s, startIndex+1, endIndex)  
    return max(op1, op2)
```



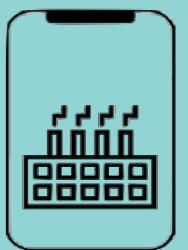
# The Longest Palindromic Substring (LPS)

## **Problem Statement:**

- S is a given string
- Find the longest palindromic substring (LPS)
- Substring: contiguous sequence of characters within a string
- Palindrome is a string that reads the same backward as well as forward

## **Example 1**

- S = “ABCCBUA”
- Output = 4
- LPS: “BCCB”



# The Longest Palindromic Substring (LPS)

## Problem Statement:

- S is a given string
- Find the longest palindromic substring (LPS)
- Substring: contiguous sequence of characters within a string
- Palindrome is a string that reads the same backward as well as forward

## Example

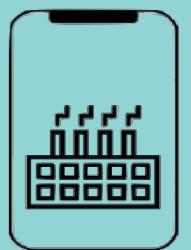
- S = “MAMDRDM”
- Output = 5
- LPS: “MDRDM”

**Subproblems:** If remaining string is palindromic also

$$\text{Option1} = 2 + f(2,6)$$

$$\text{Option2} = 0 + f(2,7) \quad \longrightarrow \text{Max}(\text{Option1}, \text{Option2}, \text{Option3})$$

$$\text{Option3} = 0 + f(1,6)$$



# The Longest Palindromic Subsequence (LPS)

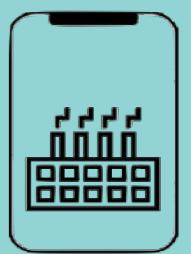
## **Problem Statement:**

- S is a given string
- Find the longest palindromic subsequence (LPS)
- Subsequence: a sequence that can be derived from another sequence by deleting some elements without changing the order of them
- Palindrome is a string that reads the same backward as well as forward

## **Example 1**

- S = “ELRMENMET”
- Output = 5
- LPS: “EMEME”

```
findLPS(S, startIndex, endIndex):  
    If startIndex > endIndex:  
        return 0  
    If s1[startIndex] == s2[endIndex]  
        return 2 + findLPS(s, startIndex+1, endIndex+1)  
  
    Else  
        op1 = findCLS(s, startIndex, endIndex-1)  
        op2 = findCLS(s, startIndex+1, endIndex)  
    return max(op1, op2)
```



# Minimum cost to reach the last cell

## Problem Statement:

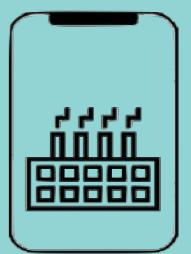
- 2D Matrix is given
- Each cell has a cost associated with it for accessing
- We need to start from (0,0) cell and go till (n-1,n-1) cell
- We can go only to right or down cell from current cell
- Find the way in which the cost is minimum

## Example

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

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

Min Cost : 36



# Minimum cost to reach the last cell

## Problem Statement:

- 2D Matrix is given
- Each cell has a cost associated with it for accessing
- We need to start from (0,0) cell and go till (n-1,n-1) cell
- We can go only to right or down cell from current cell
- Find the way in which the cost is minimum

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

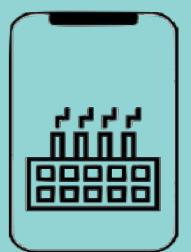
## Subproblems:

$$\text{Option1} = y + 9 + 3 \quad f(4,3)$$

$$\text{Option2} = z + 7 + 3 \quad f(3,4)$$



$$\text{Min}(\text{Option1}, \text{Option2})$$

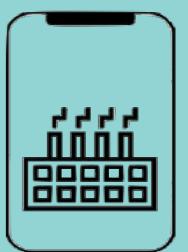


# Minimum cost to reach the last cell

## Problem Statement:

- 2D Matrix is given
- Each cell has a cost associated with it for accessing
- We need to start from (0,0) cell and go till (n-1,n-1) cell
- We can go only to right or down cell from current cell
- Find the way in which the cost is minimum

```
findMinCost(twoDArray, row, col):  
    If row == -1 or col == -1:  
        return inf  
    If row == 0 and col == 0:  
        return twoDArray[row][col]  
  
    Else  
        op1 = findMinCost(twoDArray, row-1, col)  
        op2 = findMinCost(twoDArray, row, col-1)  
    return cost[row][col] + min(op1, op2)
```



# Number of paths to reach the last cell with given cost

## Problem Statement:

- 2D Matrix is given
- Each cell has a cost associated with it for accessing
- We need to start from (0,0) cell and go till (n-1,n-1) cell
- We can go only to right or down cell from current cell
- We are given total cost to reach the last cell
- Find the number of ways to reach end of matrix with given “total cost”

Example      Total cost : 25

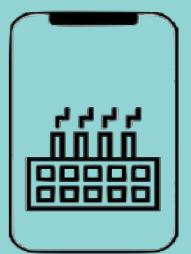
4	7	1	6
5	7	3	9
3	2	1	2
7	1	6	3

Answer 1

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

Answer 2

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



# Number of paths to reach the last cell with given cost

## Problem Statement:

- 2D Matrix is given
- Each cell has a cost associated with it for accessing
- We need to start from (0,0) cell and go till (n-1,n-1) cell
- We can go only to right or down cell from current cell
- We are given total cost to reach the last cell
- Find the number of ways to reach end of matrix with given “total cost”

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

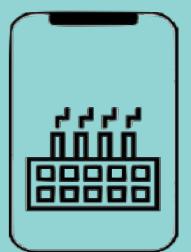
## Subproblems:

$$\text{Option1} = y + 2 = 22 \quad f(3,4,22)$$

$$\text{Option2} = z + 6 = 22 \quad f(4,3,22)$$



Sum(Option1, Option2)

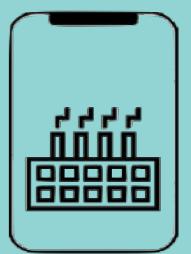


# Number of paths to reach the last cell with given cost

## Problem Statement:

- 2D Matrix is given
- Each cell has a cost associated with it for accessing
- We need to start from (0,0) cell and go till (n-1,n-1) cell
- We can go only to right or down cell from current cell
- We are given total cost to reach the last cell
- Find the number of ways to reach end of matrix with given “total cost”

```
number0fPaths(twoDArray, row, col, cost):
    if cost < 0:
        return 0
    elif row == 0 and col == 0:
        if twoDArray[0][0] - cost == 0:
            return 1
        else:
            return 0
    elif row == 0:
        return number0fPaths(twoDArray, 0, col - 1, cost - twoDArray[row][col])
    elif col == 0:
        return number0fPaths(twoDArray, row-1, 0, cost - twoDArray[row][col])
    else:
        op1 = number0fPaths(twoDArray, row-1, col, cost - twoDArray[row][col])
        op2 = number0fPaths(twoDArray, row, col-1, cost - twoDArray[row][col])
        return op1+op2
```



# What is Dynamic Programming?

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems

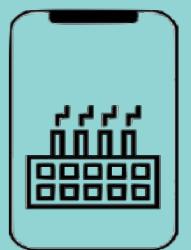
Example : 1

$$1 + 1 + 1 + 1 + 1 + 1 + 1 = 7$$

$$1 + 1 + 1 + 1 + 1 + 1 + 1 + 2 = 9$$



7



# What is Dynamic Programming?

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems

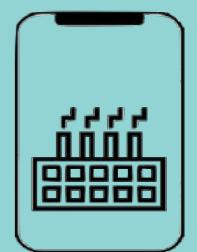
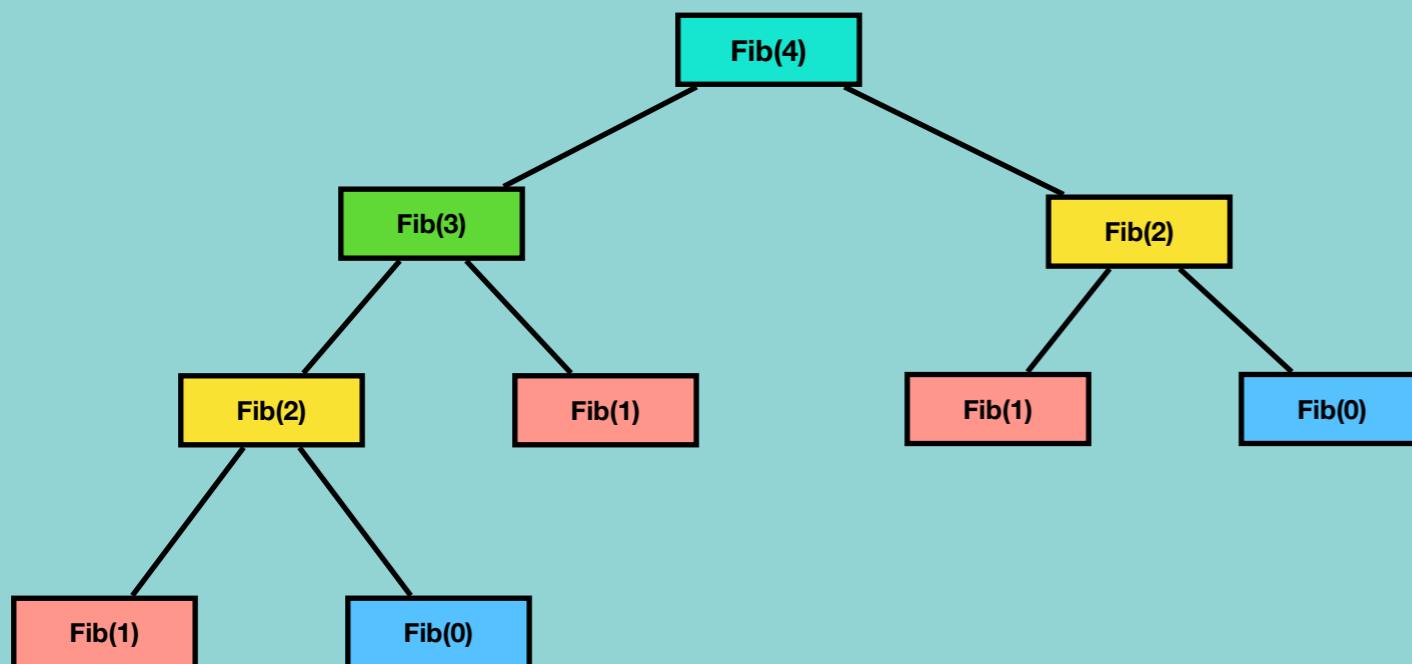
## Optimal Substructure:

If any problem's overall optimal solution can be constructed from the optimal solutions of its subproblem then this problem has optimal substructure

Example:  $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

## Overlapping Subproblem:

Subproblems are smaller versions of the original problem. Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times



# Top Down with Memoization

Solve the bigger problem by recursively finding the solution to smaller subproblems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of already solved subproblems is called **Memoization**.

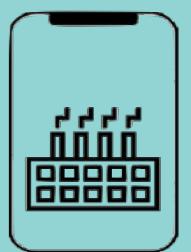
**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$

```
Fibonacci(N):
    If n < 1 return error message
    If n = 1 return 0
    If n = 2 return 1
    Else
        return Fibonacci(N-1) + Fibonacci(N-2)
```

Time complexity :  $O(c^n)$

Space complexity :  $O(n)$

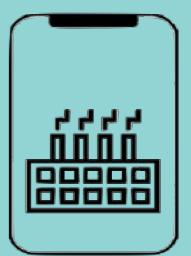
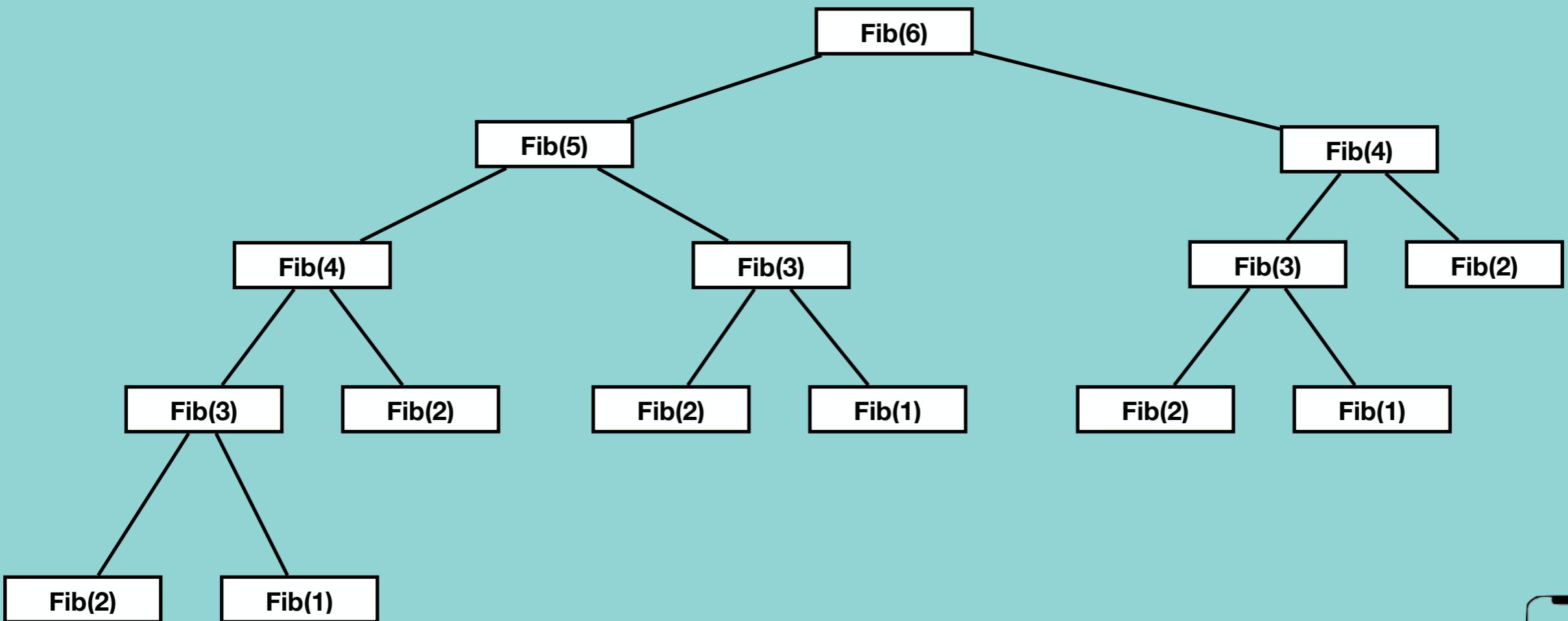


# Top Down with Memoization

Solve the bigger problem by recursively finding the solution to smaller subproblems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of already solved subproblems is called **Memoization**.

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

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$

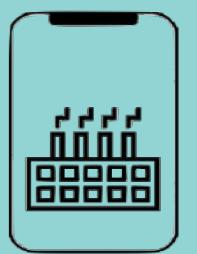
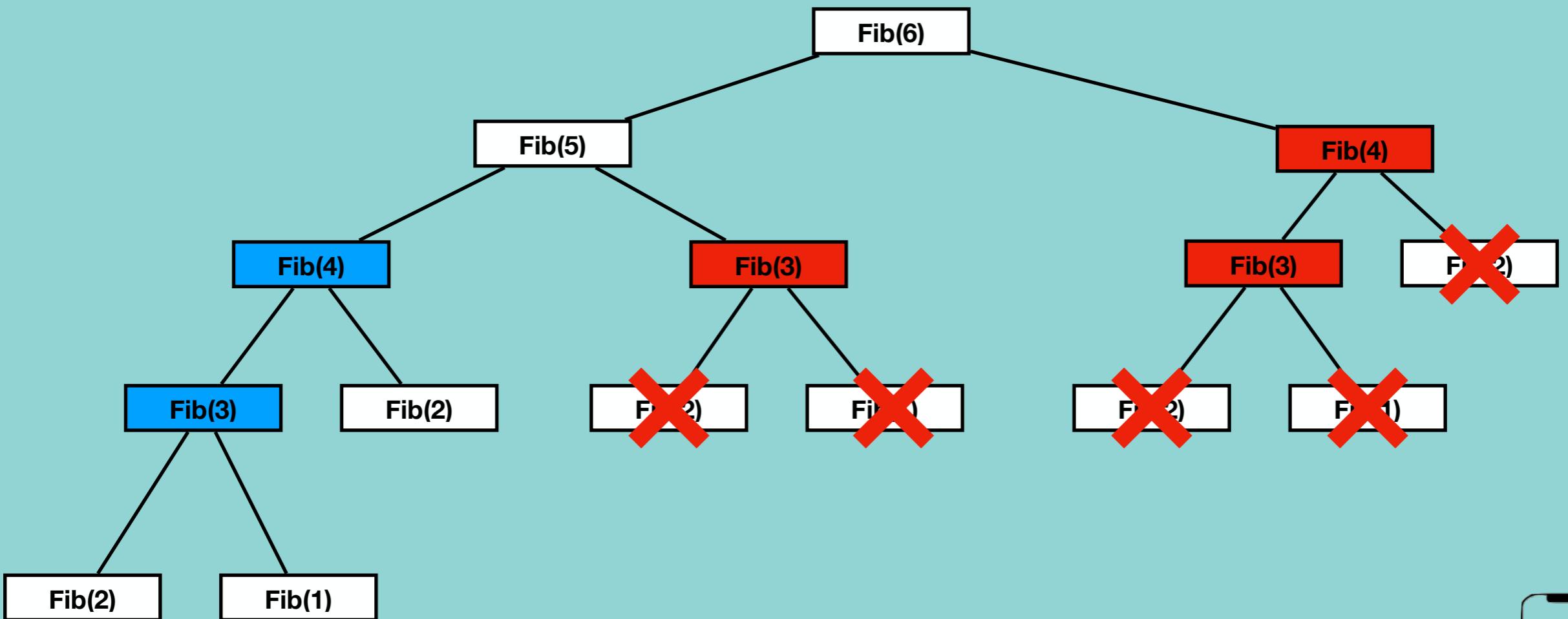


# Top Down with Memoization

Solve the bigger problem by recursively finding the solution to smaller subproblems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of already solved subproblems is called **Memoization**.

**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$



# Top Down with Memoization

Solve the bigger problem by recursively finding the solution to smaller subproblems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of already solved subproblems is called **Memoization**.

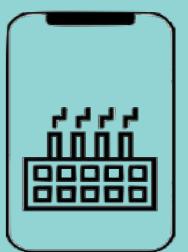
**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$

```
Fibonacci(n):
    if n < 1 return error message
    if n = 1 return 0
    if n = 2 return 1
    if not n in memo:
        memo[n] = Fibonacci(n-1, memo) + Fibonacci(n-2, memo)
    return memo[n]
```

Time complexity : O(n)

Space complexity : O(n)

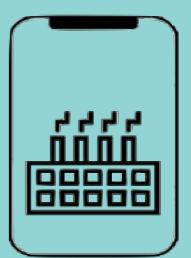
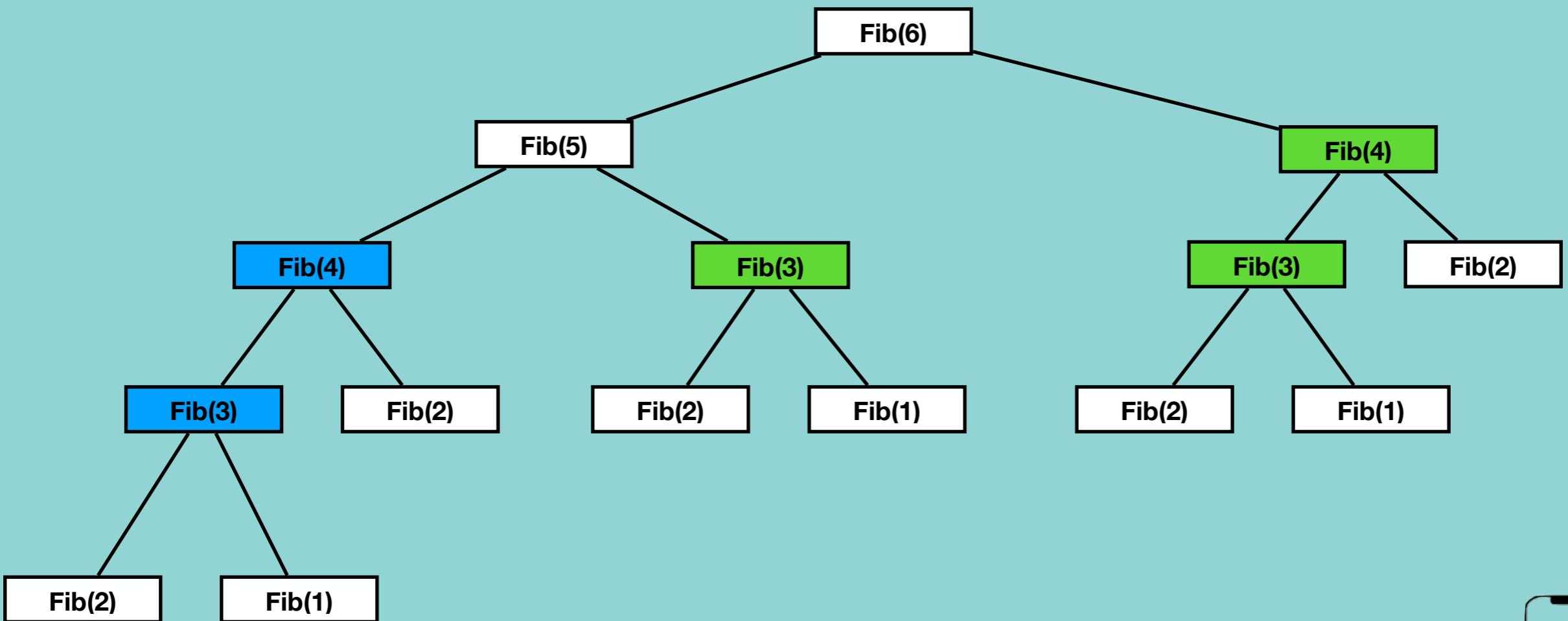


# Top Down with Memoization

Solve the bigger problem by recursively finding the solution to smaller subproblems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of already solved subproblems is called **Memoization**.

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

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$



# Top Down with Memoization

Solve the bigger problem by recursively finding the solution to smaller subproblems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of already solved subproblems is called **Memoization**.

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

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$

F1	F2	F3	F4	F5	F6
0	1				$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1	1	$F2 + F3$	$F3 + F4$	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1		$F3 + F4$	$F4 + F5$	

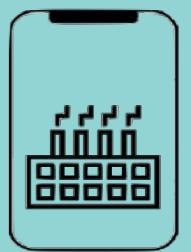
F1	F2	F3	F4	F5	F6
0	1	1	2	$F3 + F4$	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1		$F2 + F3$	$F3 + F4$	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1	1	2	3	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1	$F1 + F2$	$F2 + F3$	$F3 + F4$	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1	1	2	3	5



# Bottom Up with Tabulation

Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem “bottom-up” (i.e. by solving all the related subproblems first). This is done by filling up a table. Based on the results in the table, the solution to the top/original problem is then computed.

**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$

F1	F2	F3	F4	F5	F6
0	1				$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1	1	$F2 + F3$	$F3 + F4$	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1			$F3 + F4$	$F4 + F5$

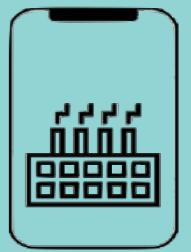
F1	F2	F3	F4	F5	F6
0	1	1	2	$F3 + F4$	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1		$F2 + F3$	$F3 + F4$	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1	1	2	3	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1	$F1 + F2$	$F2 + F3$	$F3 + F4$	$F4 + F5$

F1	F2	F3	F4	F5	F6
0	1	1	2	3	5



# Bottom Up with Tabulation

Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem “bottom-up” (i.e. by solving all the related subproblems first). This is done by filling up a table. Based on the results in the table, the solution to the top/original problem is then computed.

**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$

F1	F2	F3	F4	F5	F6
0	1	$F_1 + F_2$			

F1	F2	F3	F4	F5	F6
0	1	1			

F1	F2	F3	F4	F5	F6
0	1	1	$F_2 + F_3$		

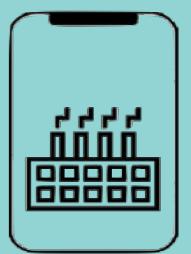
F1	F2	F3	F4	F5	F6
0	1	1	2		

F1	F2	F3	F4	F5	F6
0	1	1	2	$F_3 + F_4$	

F1	F2	F3	F4	F5	F6
0	1	1	2	3	

F1	F2	F3	F4	F5	F6
0	1	1	2	3	$F_4 + F_5$

F1	F2	F3	F4	F5	F6
0	1	1	2	3	5



# Bottom Up with Tabulation

Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem “bottom-up” (i.e. by solving all the related subproblems first). This is done by filling up a table. Based on the results in the table, the solution to the top/original problem is then computed.

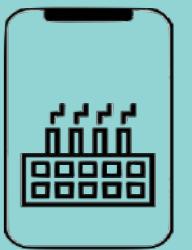
**Example :** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

$$\text{Fibonacci}(N) = \text{Fibonacci}(N-1) + \text{Fibonacci}(N-2)$$

```
def fibonacciTab(n):
    tb = [0, 1]
    for i in range(2, n + 1):
        tb.append(tb[i - 1] + tb[i - 2])
    return tb[n-1]
```

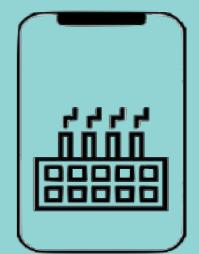
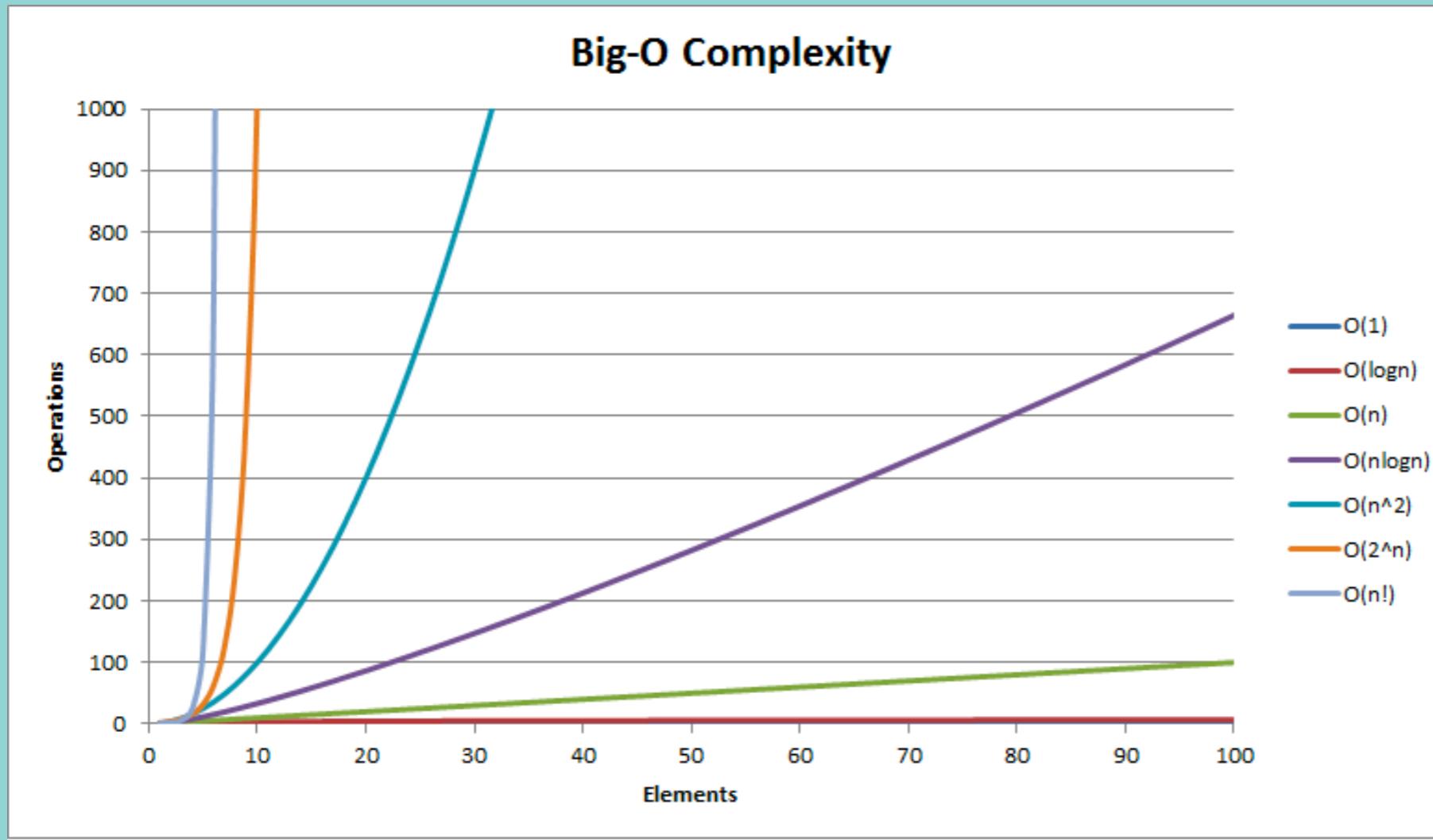
Time complexity : O(n)

Space complexity : O(n)



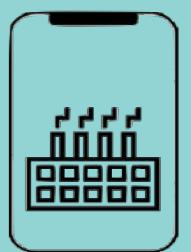
# Top Down vs Bottom Up

Problem	Divide and Conquer	Top Down	Bottom Up
Fibonacci numbers	$O(c^n)$	$O(n)$	$O(n)$



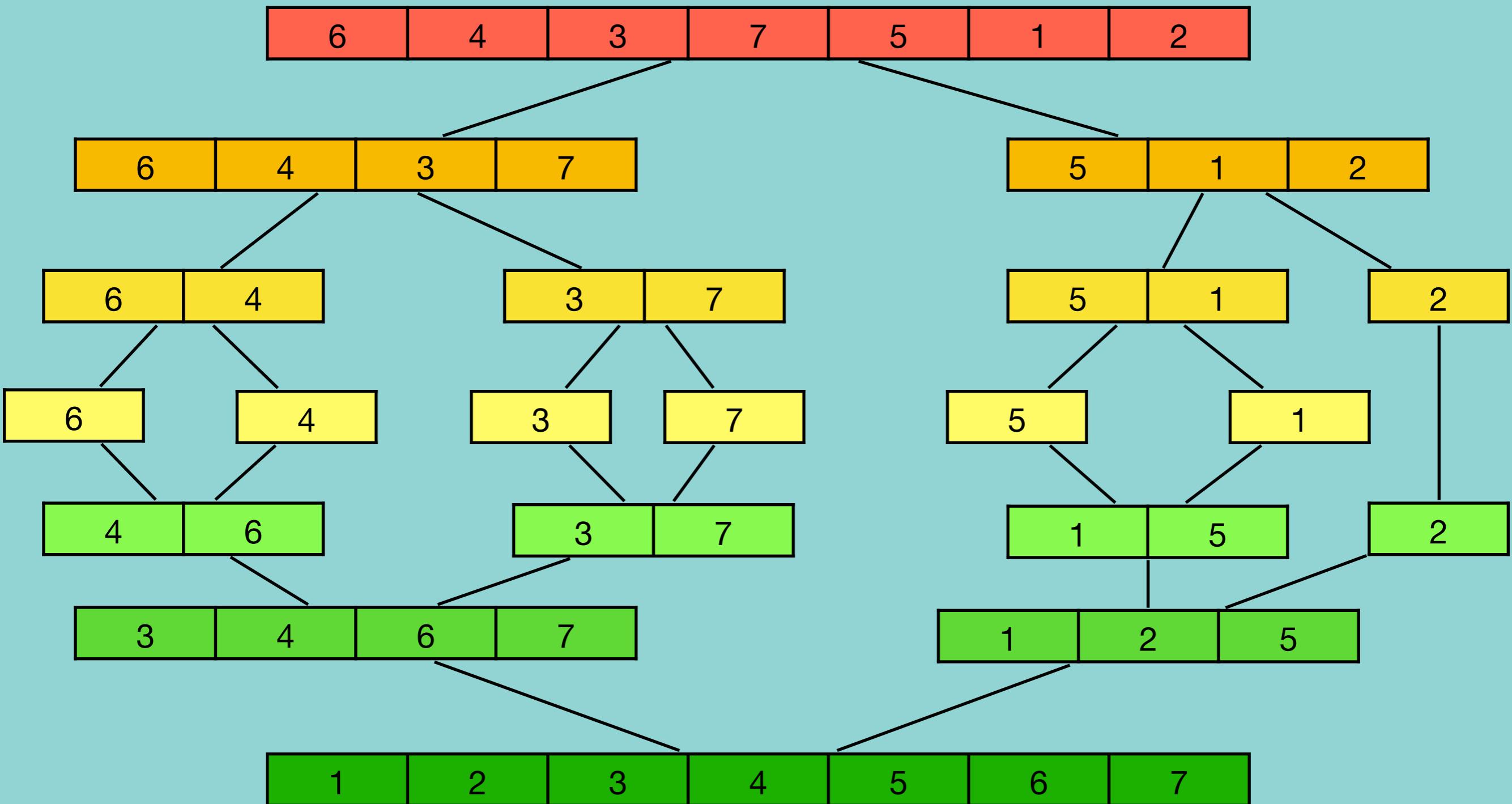
# Top Down vs Bottom Up

	<b>Top Down</b>	<b>Bottom Up</b>
Easyness	Easy to come up with solution as it is extension of divide and conquer	Difficult to come up with solution
Runtime	Slow	Fast
Space efficiency	Unnecessary use of stack space	Stack is not used
When to use	Need a quick solution	Need an efficient solution

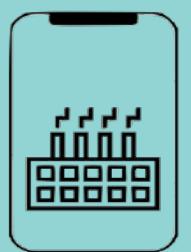
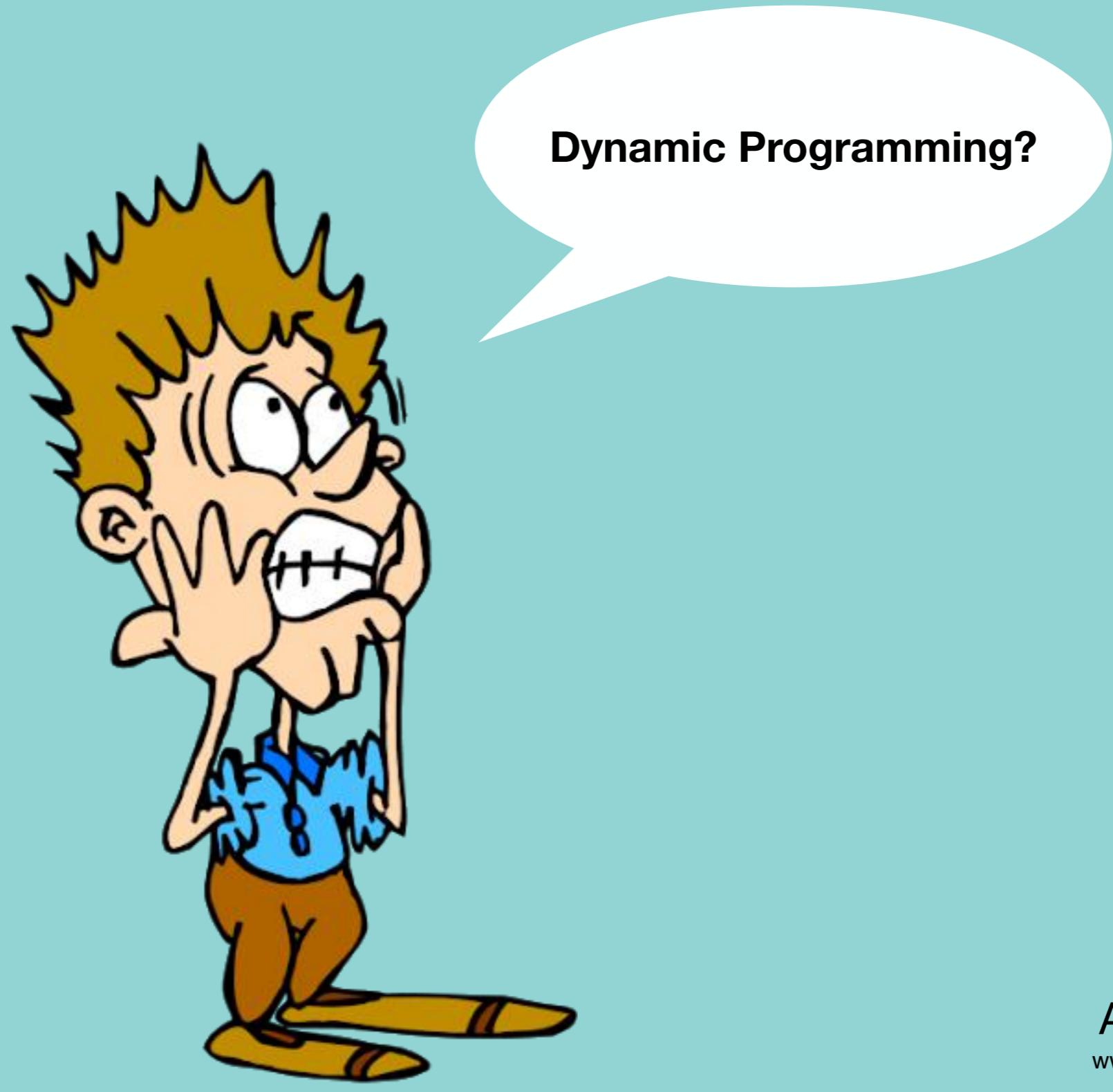


# Is Merge Sort Dynamic Programming?

1. Does it have Optimal Substructure property?
2. Does it have Overlapping Subproblems property?



# Where does the name of DP come from?



# Dynamic Programming - Number Factor problem

## Problem Statement:

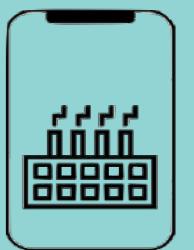
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

### Example 1

- N = 4
- Number of ways = 4
- Explanation : There are 4 ways we can express N. {4},{1,3},{3,1},{1,1,1,1}

### Example 2

- N = 5
- Number of ways = 6
- Explanation : There are 6 ways we can express N. {4,1},{1,4},{1,3,1},{3,1,1},{1,1,3},{1,1,1,1,1}

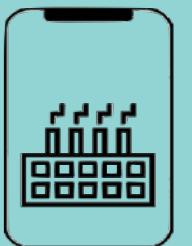


# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

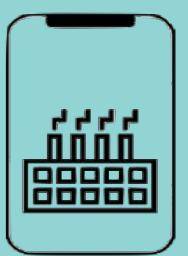
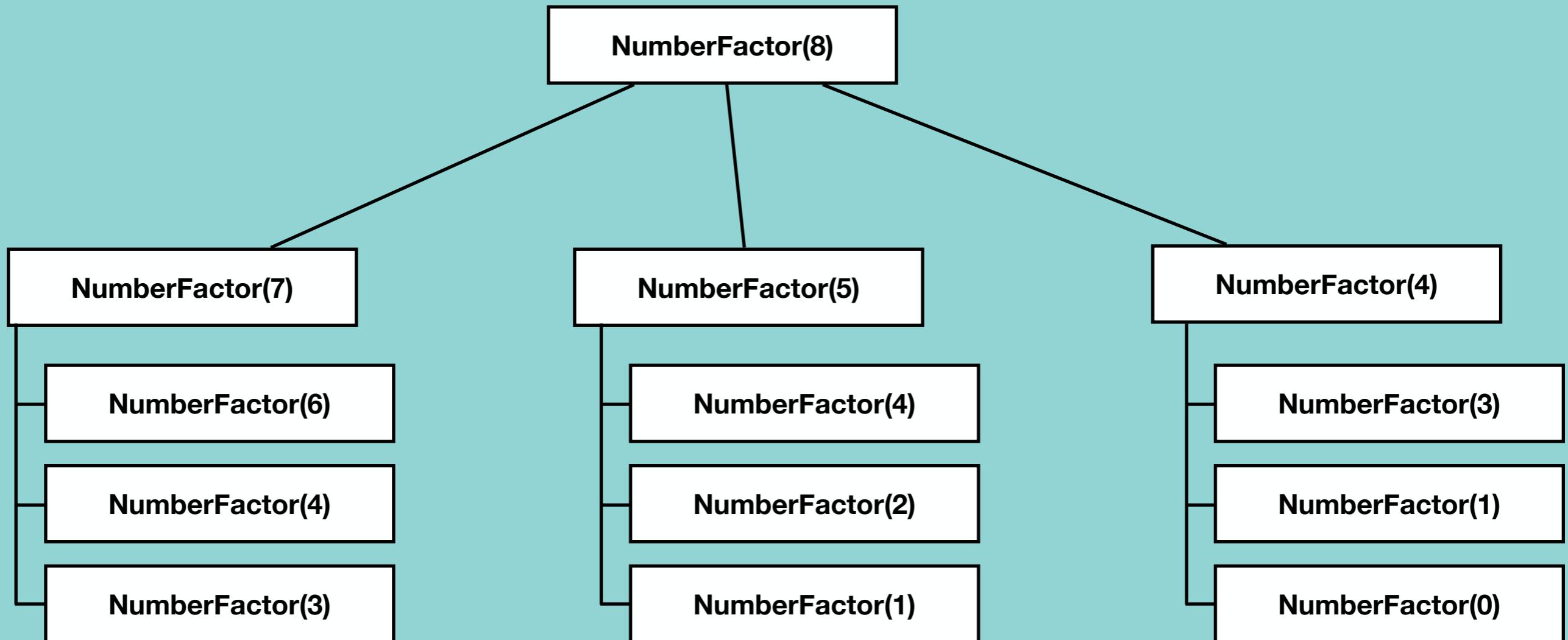
```
NumberFactor(N):  
    If N in (0,1,2) return 1  
    If N = 3 return 2  
    Else  
        return NumberFactor(N-1) + NumberFactor(N-3) + NumberFactor(N-4)
```



# Dynamic Programming - Number Factor problem

## Problem Statement:

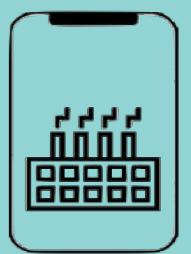
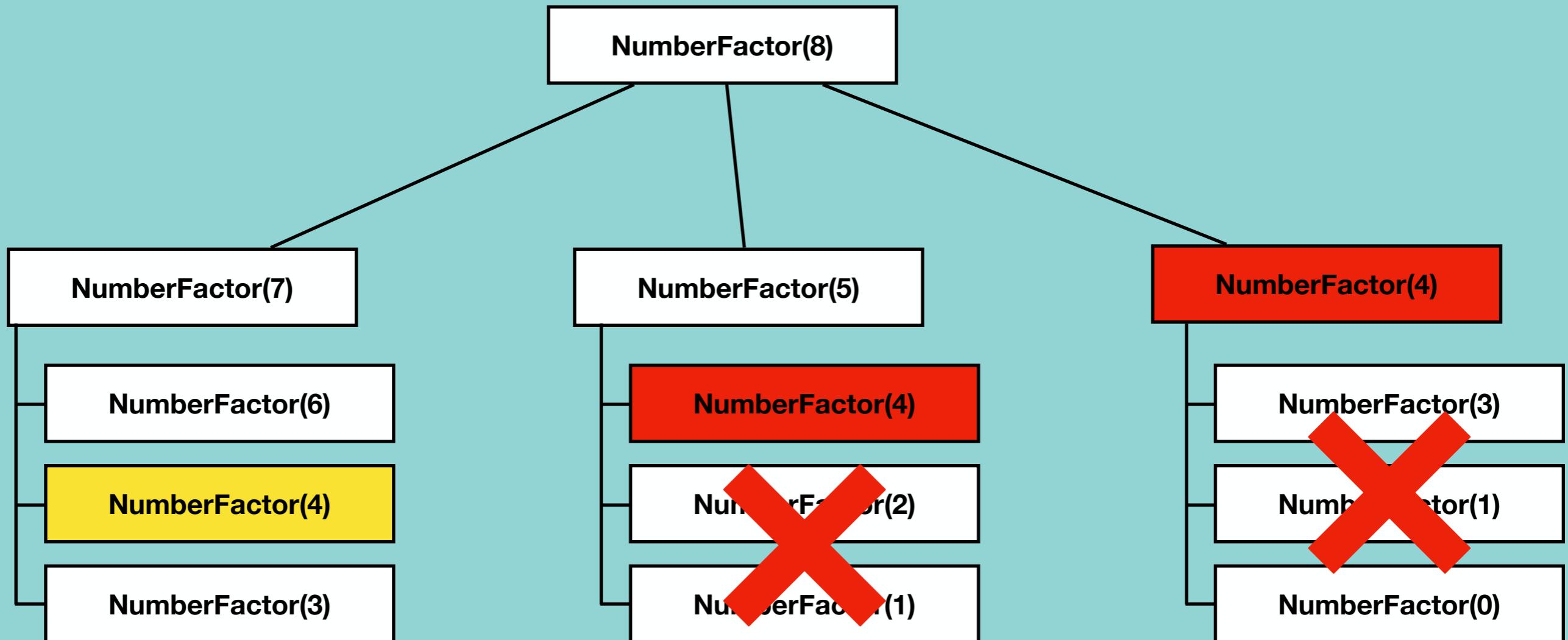
Given N, find the number of ways to express N as a sum of 1, 3 and 4.



# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.



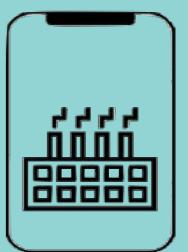
# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

NumberFactor(N):

```
If N in (0,1,2) return 1  
If N = 3 return 2  
  
Else  
    rec1 = NumberFactor(N-1)  
    rec2 = NumberFactor(N-3)  
    rec3 = NumberFactor(N-4)  
  
    return rec1 + rec2 + rec3
```



# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N, dp):
```

```
    If N in (0,1,2) return 1
```

```
    If N = 3 return 2
```

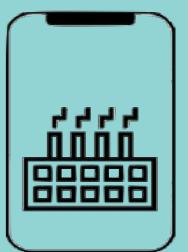
```
    Else
```

```
        rec1 = NumberFactor(N-1)
```

```
        rec2 = NumberFactor(N-3)
```

```
        rec3 = NumberFactor(N-4)
```

```
    return rec1 + rec2 + rec3
```



# Dynamic Programming - Number Factor problem

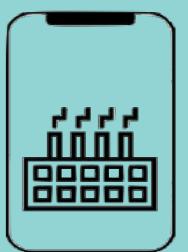
## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N, dp):
```

```
If N in (0,1,2) return 1
If N = 3 return 2
Elif N in dp return dp[N]
Else
    rec1 = NumberFactor(N-1)
    rec2 = NumberFactor(N-3)
    rec3 = NumberFactor(N-4)

    return rec1 + rec2 + rec3
```



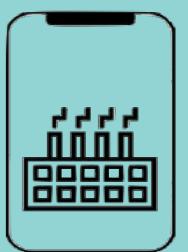
# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N, dp):
```

```
If N in (0,1,2) return 1
If N = 3 return 2
Elif N in dp return dp[N]
Else
    rec1 = NumberFactor(N-1)
    rec2 = NumberFactor(N-3)
    rec3 = NumberFactor(N-4)
    dp[N] = rec1 + rec2 + rec3
```



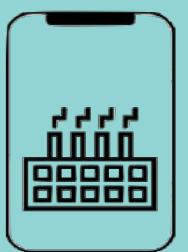
# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N, dp):
```

```
If N in (0,1,2) return 1
If N = 3 return 2
Elif N in dp return dp[N]
Else
    rec1 = NumberFactor(N-1)
    rec2 = NumberFactor(N-3)
    rec3 = NumberFactor(N-4)
    dp[N] = rec1 + rec2 + rec3
return dp[N]
```



# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

```
NumberFactor(N, dp): -----> Step 1
```

```
If N in (0,1,2) return 1
```

```
If N = 3 return 2
```

```
Elif N in dp return dp[N]
```

```
Else
```

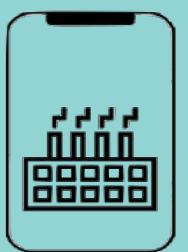
```
    rec1 = NumberFactor(N-1)
```

```
    rec2 = NumberFactor(N-3)
```

```
    rec3 = NumberFactor(N-4)
```

```
    dp[N] = rec1 + rec2 + rec3 -----> Step 3
```

```
    return dp[N] -----> Step 4
```



# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

## Top Down Approach

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2				NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	NF4+NF2+NF1	NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	NF5+NF3+NF2	NF6+NF4+NF3		

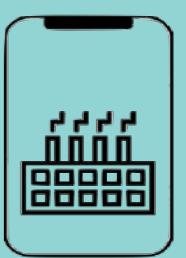
NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	4+1+1=6	NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	NF4+NF2+NF1	NF5+NF3+NF2	NF6+NF4+NF3	

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	NF3+NF2+NF0	NF4+NF2+NF1	NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	9+4+2=15



# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

## Bottom Up Approach

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	NF3+NF2+NF0			

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	NF4+NF2+NF1	NF6+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	NF4+NF2+NF1		

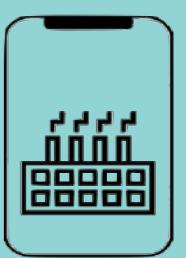
NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	4+1+1=6	NF6+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	NF3+NF2+NF0	NF4+NF2+NF1	NF6+NF3+NF2	

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	NF3+NF2+NF0	NF4+NF2+NF1	NF6+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
0	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	9+4+2=15



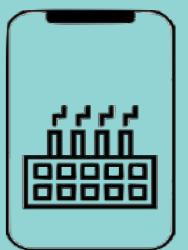
# Dynamic Programming - Number Factor problem

## Problem Statement:

Given N, find the number of ways to express N as a sum of 1, 3 and 4.

## Bottom Up Approach

```
def numberFactor(n):
    tb = [1,1,1,2]
    for i in range(4, n+1):
        tb.append(tb[i-1]+tb[i-3]+tb[i-4])
    return tb[n]
```



# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

## Example 1



## Answer

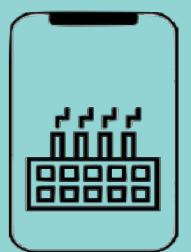
- Maximum amount = 41
- Houses that are stolen : 7, 30, 4

$$\text{Option1} = 6 + f(5)$$



$$\text{Max}(\text{Option1}, \text{Option2})$$

$$\text{Option2} = 0 + f(6)$$

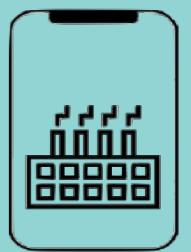


# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

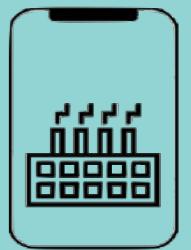
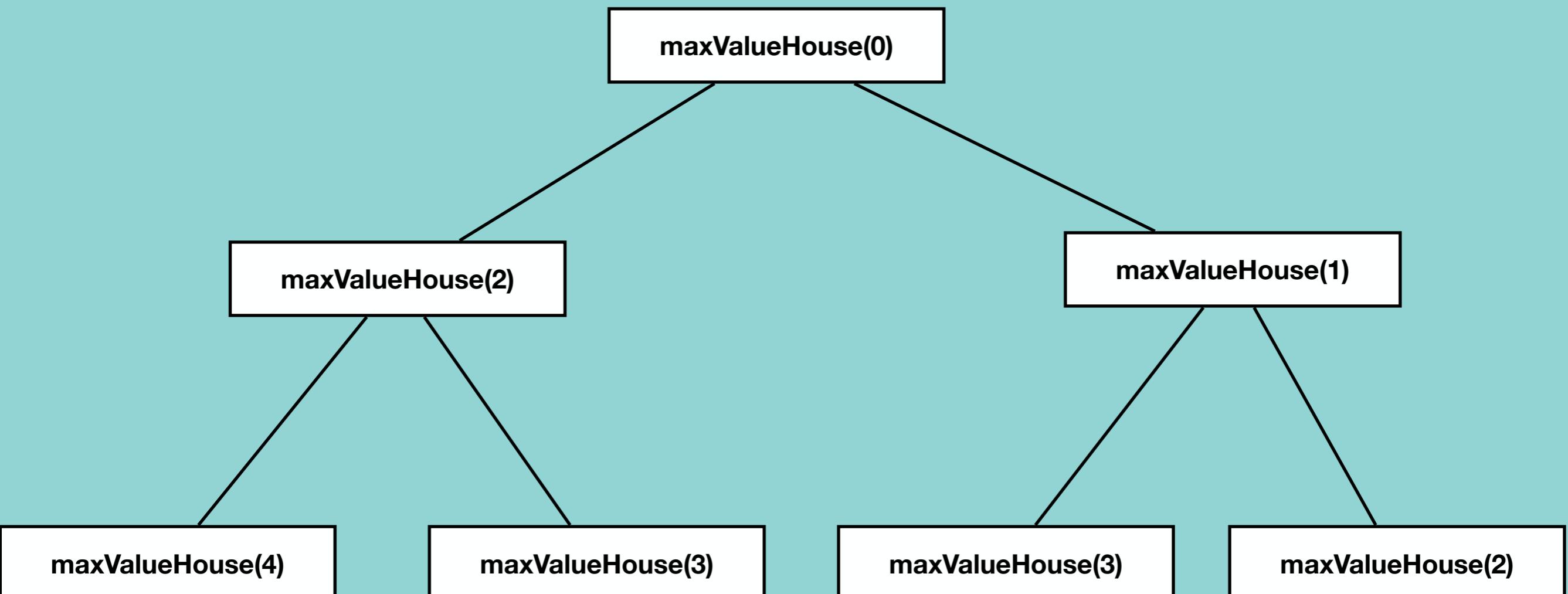
```
maxValueHouse(houses, currentHouse):  
    If currentHouse > length of houses  
        return 0  
    Else  
        stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)  
        skipFirstHouse = maxValueHouse(houses, currentHouse+1)  
        return max(stealFirstHouse, skipFirstHouse)
```



# House Robber

## Problem Statement:

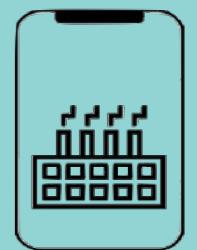
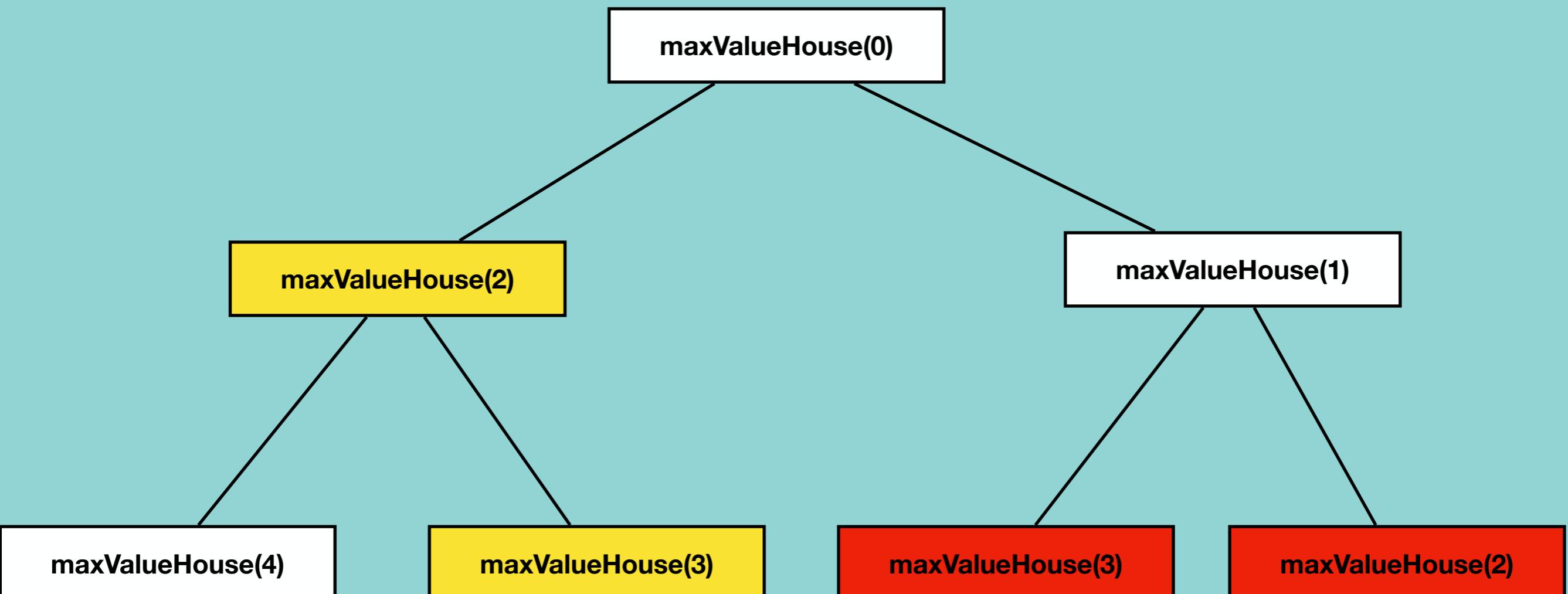
- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen



# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

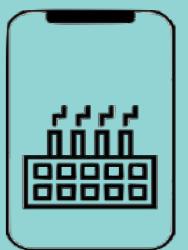


# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

```
maxValueHouse(houses, currentHouse):  
    If currentHouse > length of houses  
        return 0  
    Else  
        stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)  
        skipFirstHouse = maxValueHouse(houses, currentHouse+1)  
        return max(stealFirstHouse, skipFirstHouse)
```

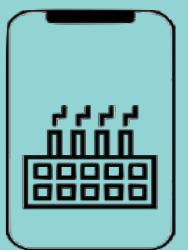


# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

```
maxValueHouse(houses, currentHouse, tempDict): -----> Step 1
    If currentHouse > length of houses
        return 0
    Else
        stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)
        skipFirstHouse = maxValueHouse(houses, currentHouse+1)
        return max(stealFirstHouse, skipFirstHouse)
```

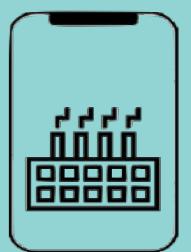


# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

```
maxValueHouse(houses, currentHouse, tempDict): -----> Step 1
    If currentHouse > length of houses
        return 0
    Else
        If currentHouse not in tempDict: -----> Step 2
            stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)
            skipFirstHouse = maxValueHouse(houses, currentHouse+1)
        return max(stealFirstHouse, skipFirstHouse)
```

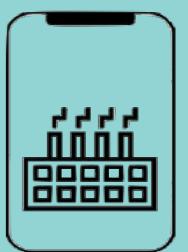


# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

```
maxValueHouse(houses, currentHouse, tempDict): -----> Step 1
    If currentHouse > length of houses
        return 0
    Else
        If currentHouse not in tempDict: -----> Step 2
            stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)
            skipFirstHouse = maxValueHouse(houses, currentHouse+1)
            tempDict[currentHouse] = max(stealFirstHouse, skipFirstHouse)-----> Step 3
        return tempDict[currentHouse] -----> Step 4
```



# House Robber

## Top Down Approach

### Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

H0	H1	H2	H3	H4	H5	H6
6	7	1	30	8	2	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)						

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)					

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)				

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)			

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)		

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	max(H6+HR8, HR7)

# House Robber

## Top Down Approach

### Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

H0	H1	H2	H3	H4	H5	H6
6	7	1	30	8	2	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	max(H6+HR8, HR7)

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(2+0, 4)=4	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(8+4,4)=12	4	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	12	4	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	34	12	4	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
41	41	34	34	12	4	4

# House Robber

## Bottom Up Approach

### Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

H0	H1	H2	H3	H4	H5	H6
6	7	1	30	8	2	4

HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	max(H6+HR8, HR7)	0	0
HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(H5+HR7, HR6)	4	0	0
HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	4	4	0	0
HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	12	4	4	0	0
HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	34	12	4	4	0	0
HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	34	34	12	4	4	0	0
HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
max(H0+HR2, HR1)	41	34	34	12	4	4	0	0
HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)	HR(7)	HR(8)
41	41	34	34	12	4	4	0	0

# House Robber

## Problem Statement:

- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen

## Bottom Up Approach

```
def houseRobberBU(houses, currentIndex):  
    tempAr = [0]*(len(houses)+2)  
    for i in range(len(houses)-1, -1, -1):  
        tempAr[i] = max(houses[i]+tempAr[i+2], tempAr[i+1])  
    return tempAr[0]
```

# Convert String

## Problem Statement:

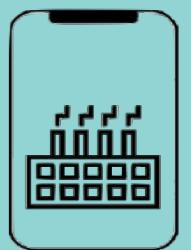
- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

## Example 1

- S1 = “catch”
- S2 = “carch”
- Output = 1
- Explanation : Replace “r” with “t”

## Example 2

- S1 = “table”
- S2 = “tbres”
- Output = 3
- Explanation : Insert “a” to second position, replace “r” with “l” and delete “s”



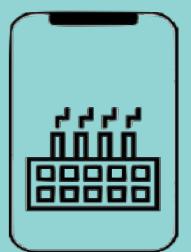
# Convert String

## Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

```
findMinOperation(s1, s2, index1, index2):
    If index1 == len(s1)
        return len(s2)-index2
    If index2 == len(s2)
        return len(s1)-index1
    If s1[index1] == s2[index2]
        return findMinOperation(s1, s2, index1+1, index2+1)

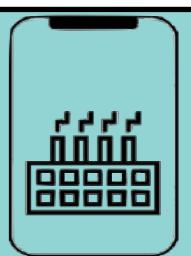
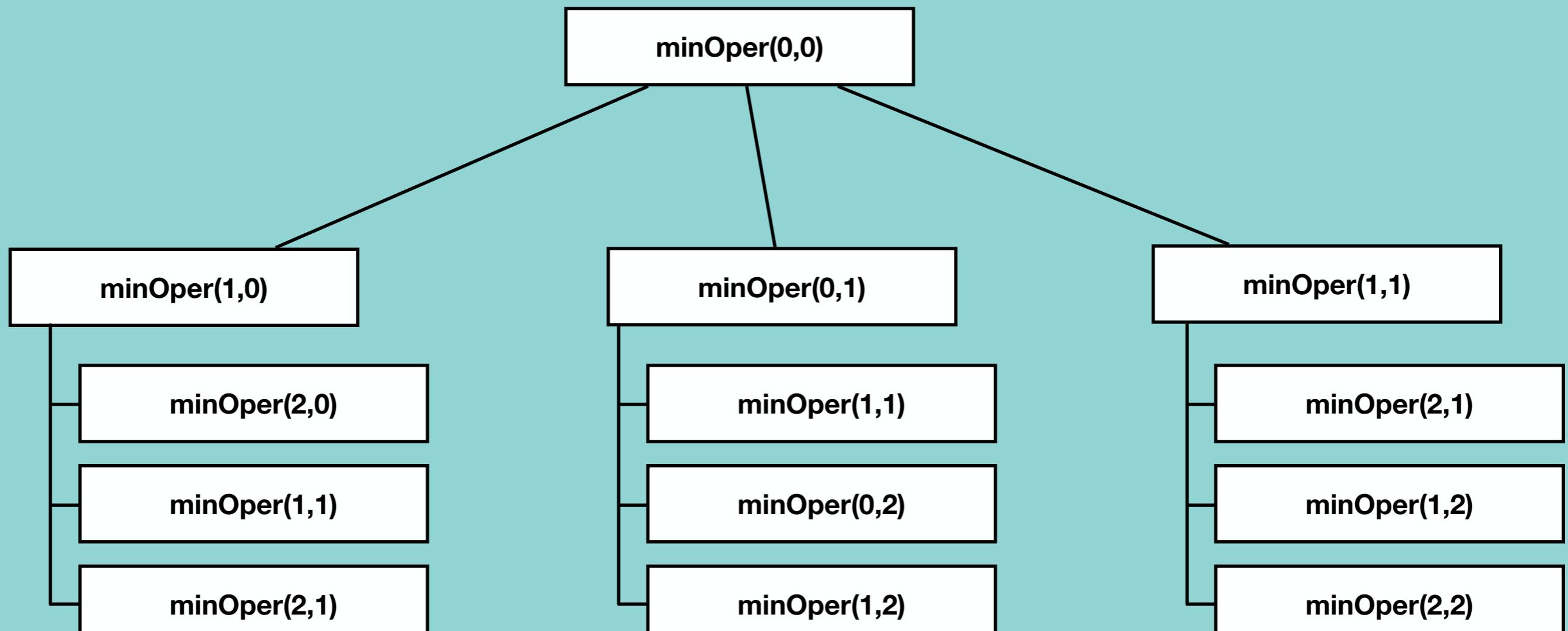
    Else
        deleteOp = 1 + findMinOperation(s1, s2, index1, index2+1)
        insertOp = 1 + findMinOperation(s1, s2, index1+1, index2)
        replaceOp = 1 + findMinOperation(s1, s2, index1+1, index2+1)
        return min(deleteOp, insertOp, replaceOp)
```



# Convert String

## Problem Statement:

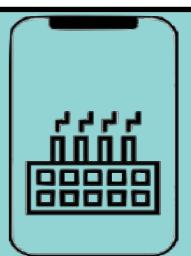
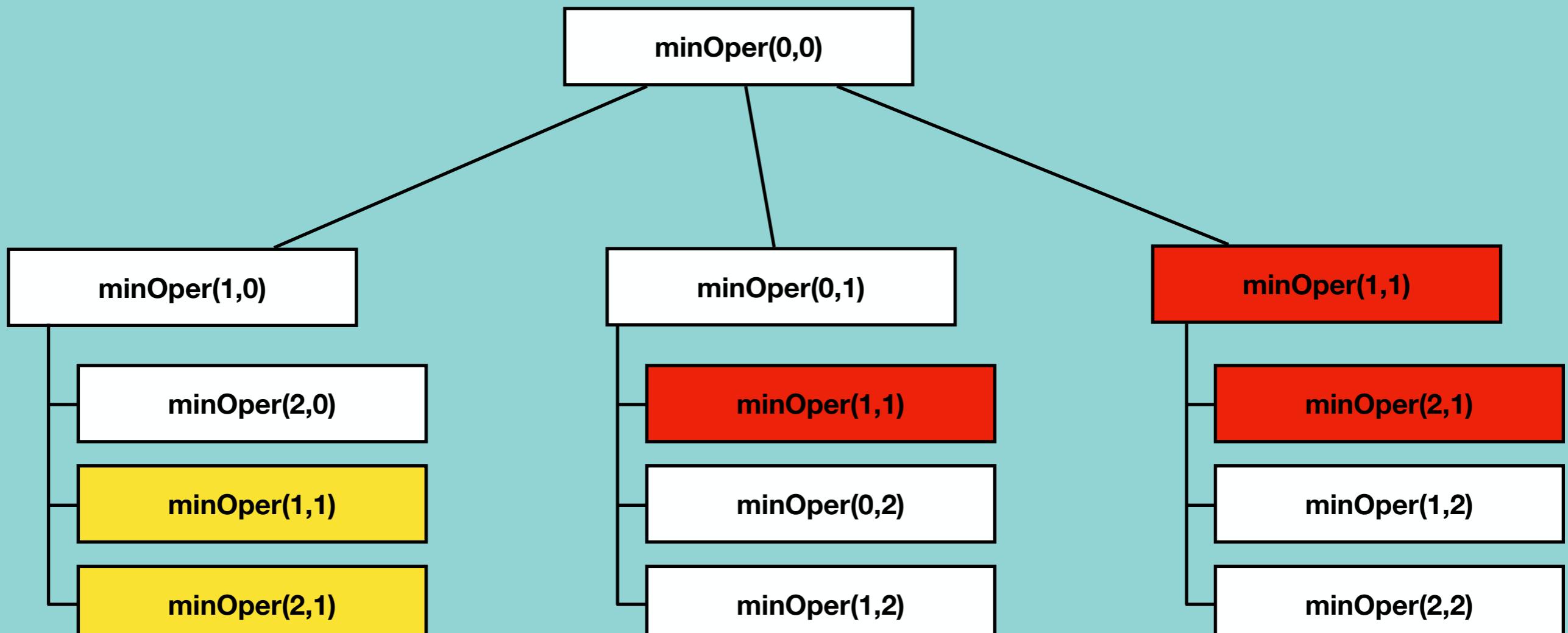
- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations



# Convert String

## Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations



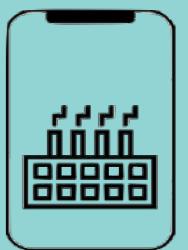
# Convert String

## Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

## Top Down Approach

```
def findMinOperation(s1, s2, index1, index2, tempDict): → Step 1
    if index1 == len(s1):
        return len(s2)-index2
    if index2 == len(s2):
        return len(s1)-index1
    if s1[index1] == s2[index2]:
        return findMinOperation(s1, s2, index1+1, index2+1, tempDict)
    else:
        dictKey = str(index1)+str(index2) → Step 2
        if dictKey not in tempDict:
            deleteOp = 1 + findMinOperation(s1, s2, index1, index2+1, tempDict)
            insertOp = 1 + findMinOperation(s1, s2, index1+1, index2, tempDict)
            replaceOp = 1 + findMinOperation(s1, s2, index1+1, index2+1, tempDict)
            tempDict[dictKey] = min (deleteOp, insertOp, replaceOp) → Step 3
        return tempDict[dictKey] → Step 4
```



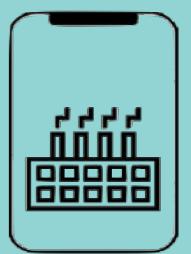
# Convert String

## Problem Statement:

- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

## Bottom Up Approach

```
def findMinOperationBU(s1, s2, tempDict):  
    for i1 in range(len(s1)+1):  
        dictKey = str(i1) + '0'  
        tempDict[dictKey] = i1  
    for i2 in range(len(s2)+1):  
        dictKey = '0' + str(i2)  
        tempDict[dictKey] = i2  
  
    for i1 in range(1, len(s1)+1):  
        for i2 in range(1, len(s2)+1):  
            if s1[i1-1] == s2[i2-1]:  
                dictKey = str(i1) + str(i2)  
                dictKey1 = str(i1-1) + str(i2-1)  
                tempDict[dictKey] = tempDict[dictKey1]  
            else:  
                dictKey = str(i1) + str(i2)  
                dictKeyD = str(i1-1) + str(i2)  
                dictKeyI = str(i1) + str(i2-1)  
                dictKeyR = str(i1-1) + str(i2-1)  
                tempDict[dictKey] = 1 + min(tempDict[dictKeyD], min(tempDict[dictKeyI], tempDict[dictKeyR]))  
    dictKey = str(len(s1)) + str(len(s2))  
    return tempDict[dictKey]
```



# Zero One Knapsack Problem

## Problem Statement:

- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken

## Example 1



**Mango**  
Weight : 3  
Profit : 31



**Apple**  
Weight : 1  
Profit : 26



**Orange**  
Weight : 2  
Profit : 17

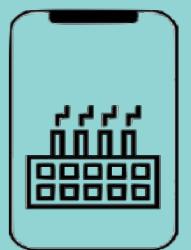


**Banana**  
Weight : 5  
Profit : 72

**Knapsack Capacity : 7**

## Answer Combinations

- Mango (W:3, P:31) + Apple (W:1,P:26) + Orange (W:2, P:17) = W:6, Profit:74
- Orange (W:2, P:17) + Banana (W:5,P:72) = W:7, Profit:89
- Apple (W:1,P:26) + Banana (W:5,P:72) = W:6, Profit:98



# Zero One Knapsack Problem

## Problem Statement:

- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken

## Example 1



**Mango**  
Weight : 3  
Profit : 31



**Apple**  
Weight : 1  
Profit : 26



**Orange**  
Weight : 2  
Profit : 17



**Banana**  
Weight : 5  
Profit : 72

**Knapsack Capacity : 7**

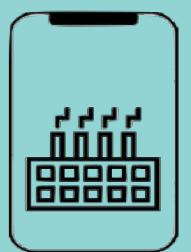
## Subproblems:

$$\text{Option1} = 31 + f(2,3,4)$$



**Max(Option1, Option2)**

$$\text{Option2} = 0 + f(2,3,4)$$

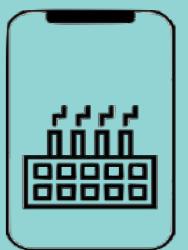


# Zero One Knapsack Problem

## Problem Statement:

- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken

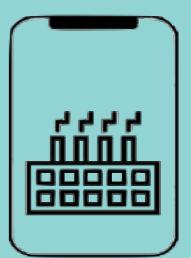
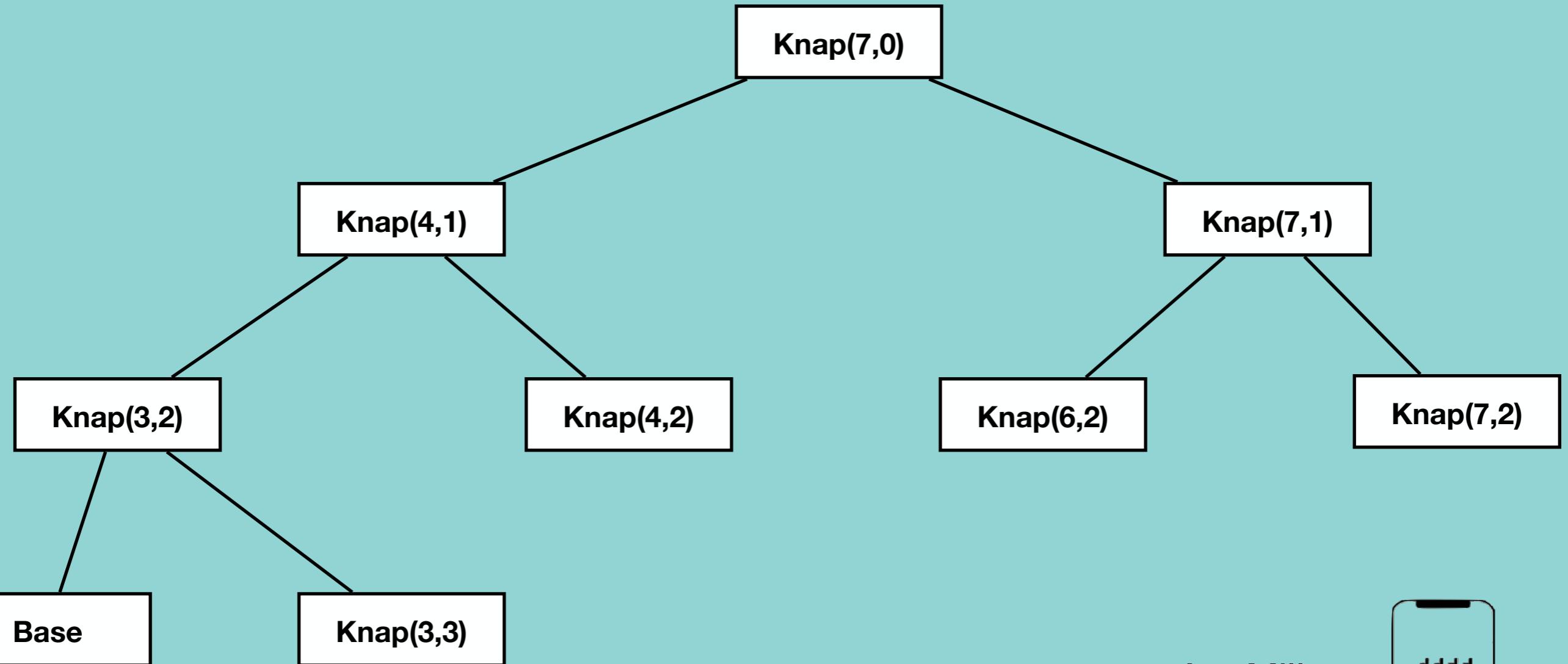
```
zoKnapsack(items, capacity, currentIndex):  
    If capacity<=0 or currentIndex<0 or currentIndex>len(profits)  
        return 0  
    Elif currentItemWeight <= capacity  
        Profit1 = curentItemProfit + zoKnapsack(items, capacity- currentItemsWeight, nextItem)  
        Profit2 = zoKnapsack(items, capacity, nextItem)  
        return max(Profit1, Profit2)  
    Else  
        return 0
```



# Zero One Knapsack Problem

## Problem Statement:

- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken



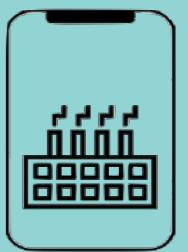
# Zero One Knapsack Problem

## Problem Statement:

- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken

## Top Down Approach

```
def zoKnapsack(items, capacity, currentIndex, tempDict): → Step 1
    dictKey = str(currentIndex) + str(capacity)
    if capacity <=0 or currentIndex < 0 or currentIndex >= len(items):
        return 0
    elif dictKey in tempDict:
        return tempDict[currentIndex] → Step 2
    elif items[currentIndex].weight <= capacity:
        profit1 = items[currentIndex].profit + zoKnapsack(items, capacity-items[currentIndex].weight, currentIndex+1, tempDict)
        profit2 = zoKnapsack(items, capacity, currentIndex+1, tempDict)
        tempDict[dictKey] = max(profit1, profit2) → Step 3
        return tempDict[dictKey] → Step 4
    else:
        return 0
```



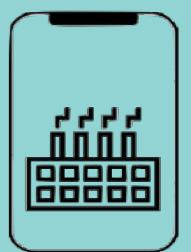
# Zero One Knapsack Problem

## Problem Statement:

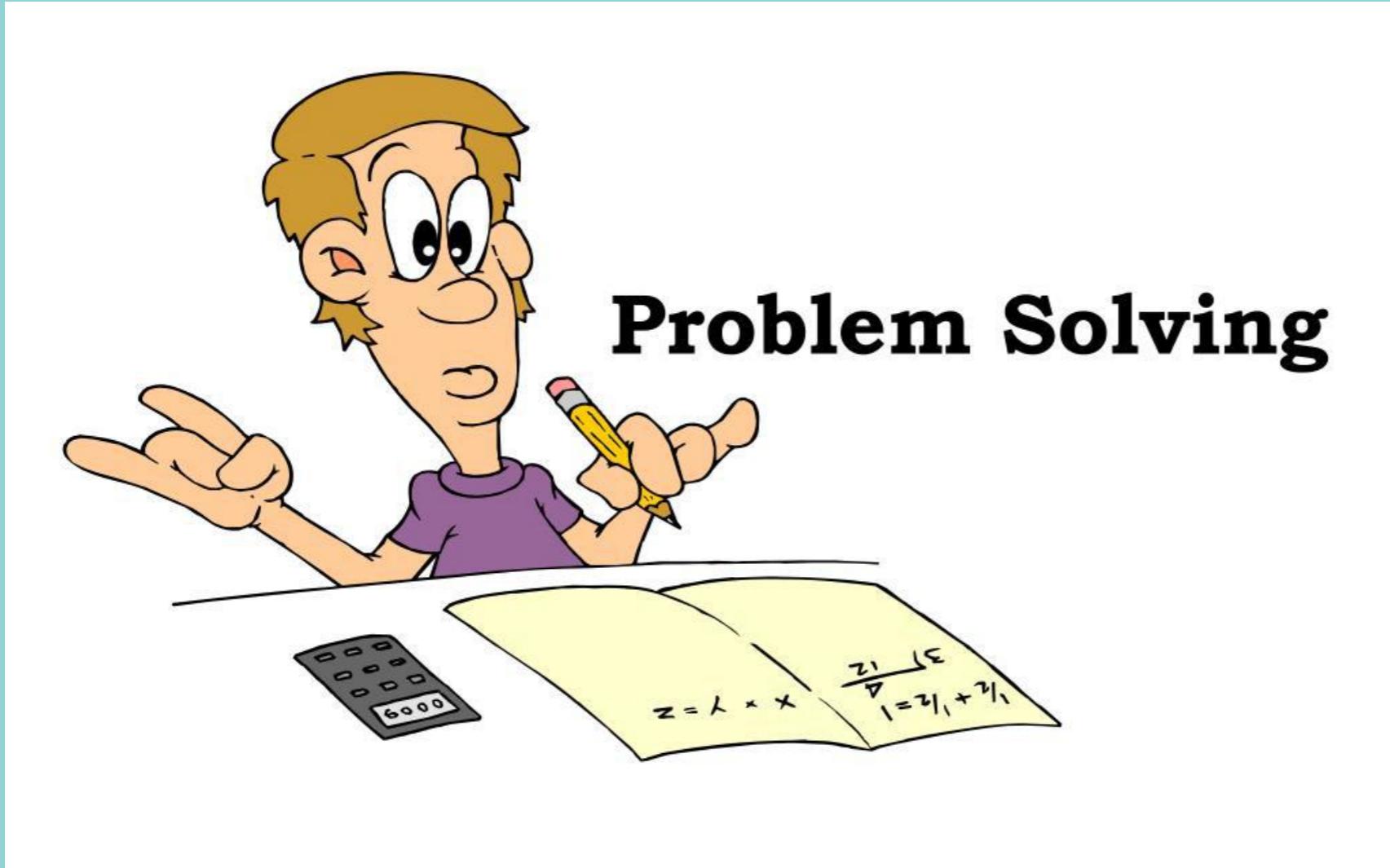
- Given the weights and profits of N items
- Find the maximum profit within given capacity of C
- Items cannot be broken

## Bottom Up Approach

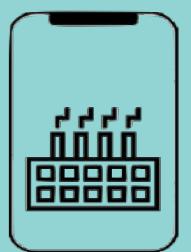
```
def zoKnapsackBU(profits, weights, capacity):
    if capacity <= 0 or len(profits) == 0 or len(weights) != len(profits):
        return 0
    numberOfRows = len(profits) + 1
    dp = [[0 for i in range(capacity+2)] for j in range(numberOfRows)]
    for row in range(numberOfRows-2, -1, -1):
        for column in range(1, capacity+1):
            profit1 = 0
            profit2 = 0
            if weights[row] <= column:
                profit1 = profits[row] + dp[row + 1][column - weights[row]]
            profit2 = dp[row + 1][column]
            dp[row][column] = max(profit1, profit2)
    return dp[0][capacity]
```



# A Recipe for Problem Solving



## Problem Solving

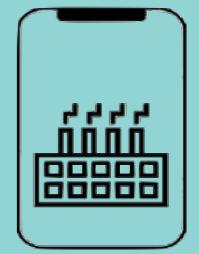


# A Recipe for Problem Solving

Algorithm : is a set of steps to accomplish a certain task

$$(x + a)^n = \sum_{k=0}^n \binom{n}{k} x^k a^{n-k}$$

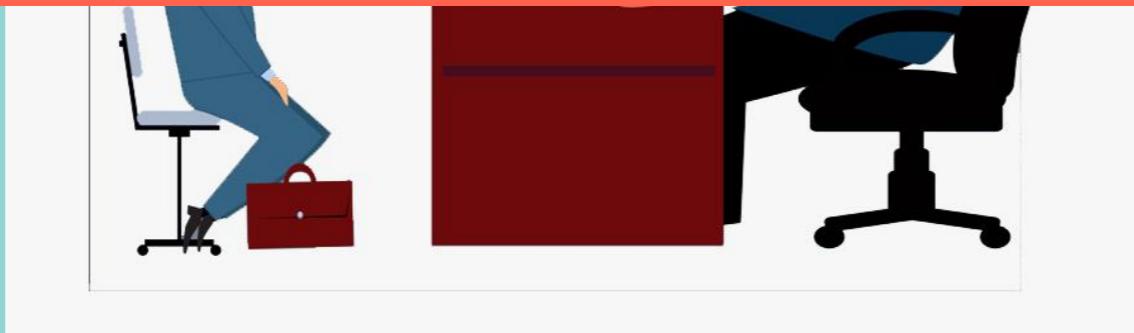
$$\begin{aligned} & x^2 + 2x - 1 \quad y = \frac{1}{2}(-12x^2 + 6x + 2) \\ & 1) \quad y = 1 \quad (\cos x + 1) + x \quad (-\sin x) = \cos x - x \sin x + 1 \\ & x_0 = ? \quad y(x_0) = 1 \quad y = 6x^2 - 4x + 1 \quad 0+1 \\ & x_2 = \frac{2}{3}, A(0-1) \quad B(\frac{2}{3} - 1) \quad y = \frac{3}{2} \\ & x_0 = ? \quad y(x_0) = 3 \quad 6x^2 - 4x + 1 = 3 \\ & x = -\frac{1}{3}, A(1; 0) \quad B(-\frac{1}{3} - 1) \quad 4x - 2 = 0 \quad 3x^2 \\ & x = 1 - 0, 1 - 3 \quad x(t) = \frac{1}{3}x - 1 \\ & x_0 = 1 \quad f = \frac{x}{x^2} = \frac{1}{x} \\ & 3 \cdot \frac{2}{3} \cdot x(3) = 2 \cdot 2 \cdot 3 = \frac{2}{3} \cdot 1 \cdot 3 = 0, 084 \\ & \text{and } x + 1 \quad y(1) = 1 \quad \log a = y(1) = 1 \\ & y = \frac{2}{x+1} \quad y = -\frac{2}{(x+1)^2} \quad \int \frac{2}{x^2} dx \\ & \int_0^3 x \sqrt{x} dx = \int_0^3 \frac{4}{x^2} dx \geq \left. \frac{3}{2} x^2 \right|_0^3 = \frac{27}{2} \end{aligned}$$



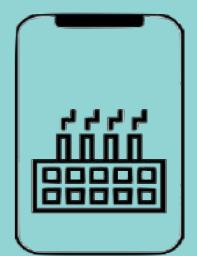
# A Recipe for Problem Solving



## 5 STEPS FOR PROBLEM SOLVING



AppMillers  
[www.appmillers.com](http://www.appmillers.com)



# A Recipe for Problem Solving

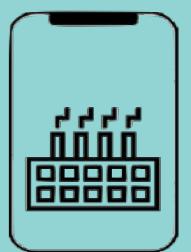
UNDERSTAND THE PROBLEM

EXPLORE EXAMPLES

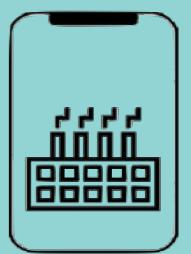
BREAK IT DOWN

SOLVE / SIMPLIFY

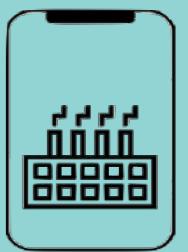
LOOK BACK REFACTOR



# Step 1 - UNDERSTAND THE PROBLEM

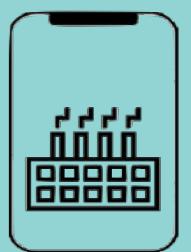


# Step 1 - UNDERSTAND THE PROBLEM



# Step 1 - UNDERSTAND THE PROBLEM

1. Can we restate the problem in our own words?
2. What are the inputs that go into the problem?
3. What are the outputs that come from the problem?
4. Can the outputs be determined from the inputs? In other words do we have enough information to solve this problem?
5. What should I label the important piece of data that are the part of a problem?



# Step 1 - UNDERSTAND THE PROBLEM

**Write a function that takes two numbers and returns their sum**

1. Can we restate the problem in our own words?

Implement addition

2. What are the inputs that go into the problem?

Integer? Float? Or?

3. What are the outputs that come from the problem?

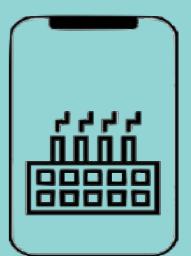
Integer? Float? Or?

4. Can the outputs be determined from the inputs? In other words do we have enough information to solve this problem?

Yes

5. What should I label the important piece of data that are the part of a problem?

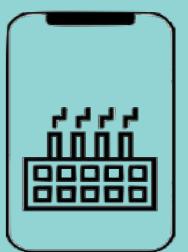
Add, Sum



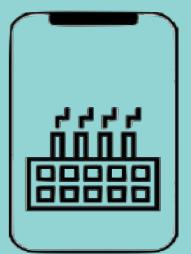
# A Recipe for Problem Solving

UNDERSTAND THE PROBLEM

EXPLORE EXAMPLES

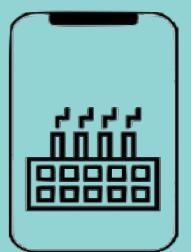


## Step 2 - EXPLORE EXAMPLES



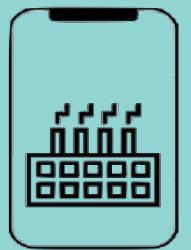
## Step 2 - EXPLORE EXAMPLES

1. Start with simple examples
2. Progress to more complex examples
3. Explore examples with empty
4. Explore the examples with invalid inputs



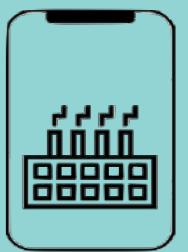
## Step 2 - EXPLORE EXAMPLES

Write a function with takes in a string and returns count of each character in the string



## Step 3 - BREAK IT DOWN

Write out the steps that you need to take

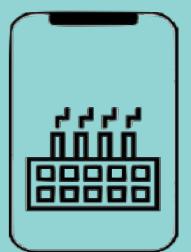


# SOLVE / SIMPLIFY

Solve the Problem

If you cannot...

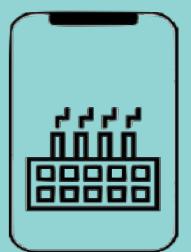
Simplify the Problem



# SOLVE / SIMPLIFY

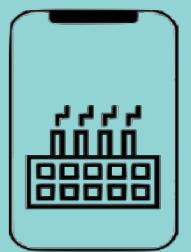
## Simplify the Problem

- Find the core difficulty
- Temporarily ignore that difficulty
- Write a simplified solution
- Then incorporate that difficulty



## LOOK BACK REFACTOR

- Can we check the result?
- Can we drive the result differently?
- Can we understand it at a glance?
- Can we use the result or method for some other problem?
- Can you improve the performance of your solution?
- How other people solve this problem?



# Summarize

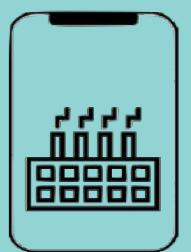
UNDERSTAND THE PROBLEM

EXPLORE EXAMPLES

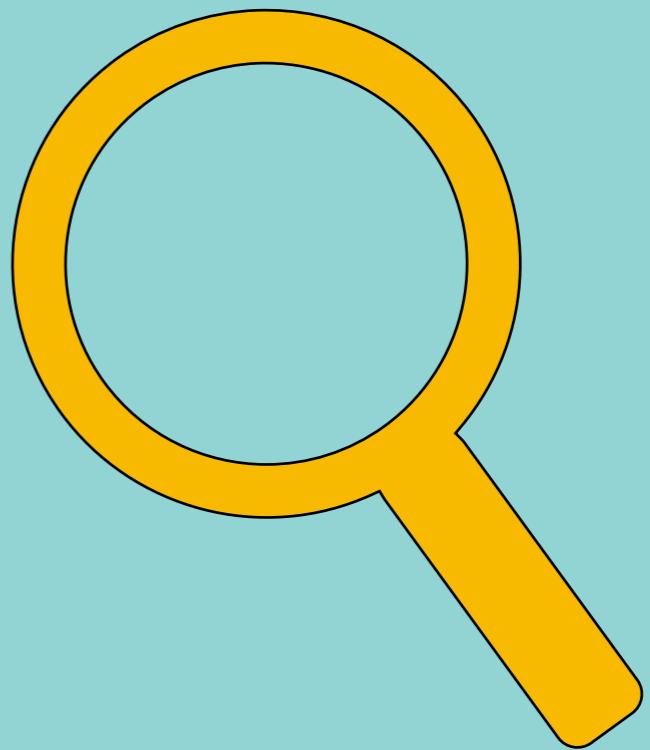
BREAK IT DOWN

SOLVE / SIMPLIFY

LOOK BACK REFACTOR



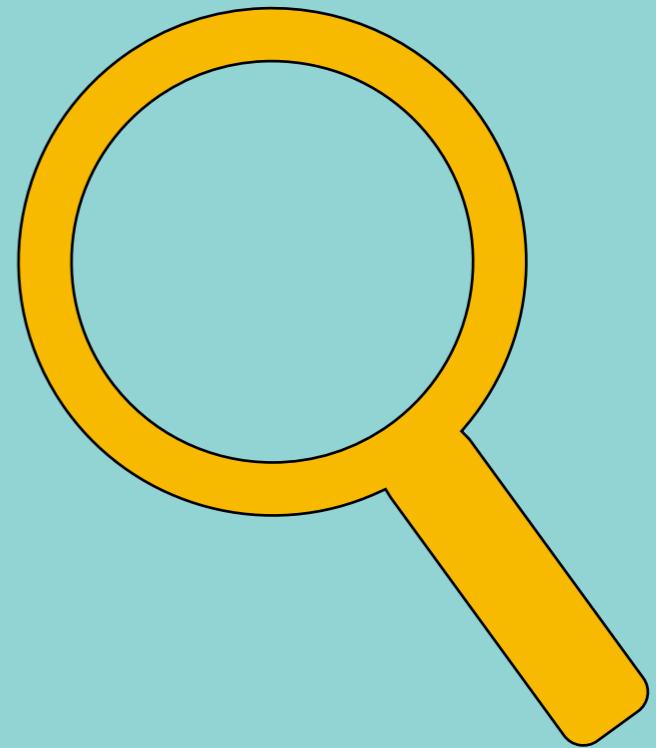
# Searching Algorithm



# Searching Algorithm

**Linear Search**

**Binary Search**



# Searching Algorithm

python

All Images Videos Books News More Settings Tools

About 420,000,000 results (0.96 seconds)

[www.python.org](http://www.python.org) ▾  
**Welcome to Python.org**  
The official home of the **Python** Programming Language.

Search python.org

**Downloads**  
Windows - Python 3.9.0 - Python 3.8.6 - Mac OS X - Python 3.7.9

**Python 3.9.0**  
Python 3.9.0. Release Date: Oct. 5, 2020. This is the stable release ...

**Python For Beginners**  
BeginnersGuide/Download - Python for Programmers - Books

**Tutorial**  
1. Whetting Your Appetite - 5. Data Structures - Classes - 6. Modules

**Documentation**  
Python's documentation, tutorials, and guides are constantly ...

**Windows**  
Python 3.9.0 - Python 3.8.5 - Python 2.7.18 - Python 3.8.6

[en.wikipedia.org › wiki › Python\\_\(programming\\_lang... ▾](https://en.wikipedia.org/wiki/Python_(programming_language))  
**Python (programming language) - Wikipedia**  
Python is an interpreted, high-level and general-purpose programming language . Created by Guido van Rossum and first released in 1991, Python's design ...  
**Developer:** Python Software Foundation      **Designed by:** Guido van Rossum  
**Paradigm:** Multi-paradigm: functional, imperative, procedural, object-oriented, reflective  
**Typing discipline:** Duck, dynamic, gradual...

 **python**   [More images](#)

**Python**  
High-level programming language

Python is an interpreted, high-level and general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. [Wikipedia](#)

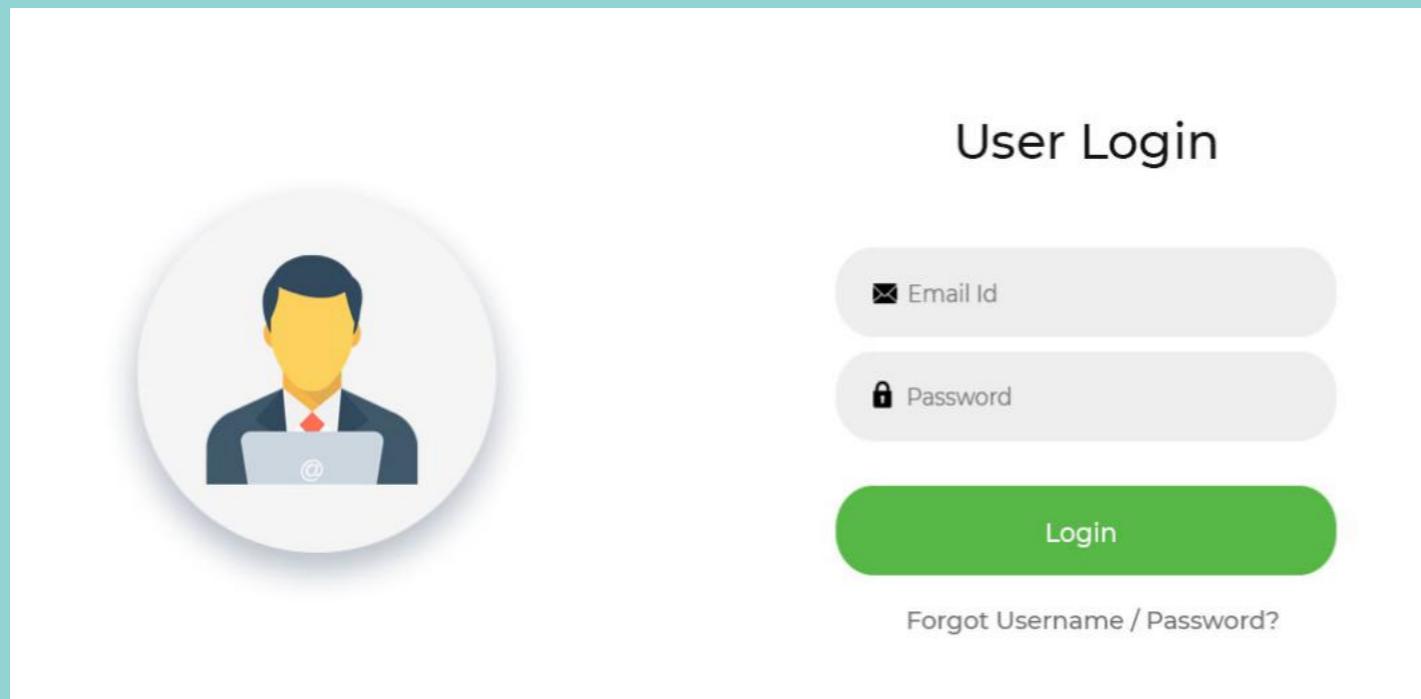
**Typing discipline:** Duck, dynamic, gradual (since 3.5)  
**Stable release:** 3.9.0 / [5 October 2020](#); 32 days ago  
**Preview release:** 3.10.0a1 / [5 October 2020](#); 32 days ago  
**Designed by:** Guido van Rossum  
**OS:** Linux, macOS, Windows; and more  
**Filename extensions:** .py,.pyi,.pyc,.pyd,.pyo (prior to 3.5),.pyw,.pyz (since 3.5)

Range

Function

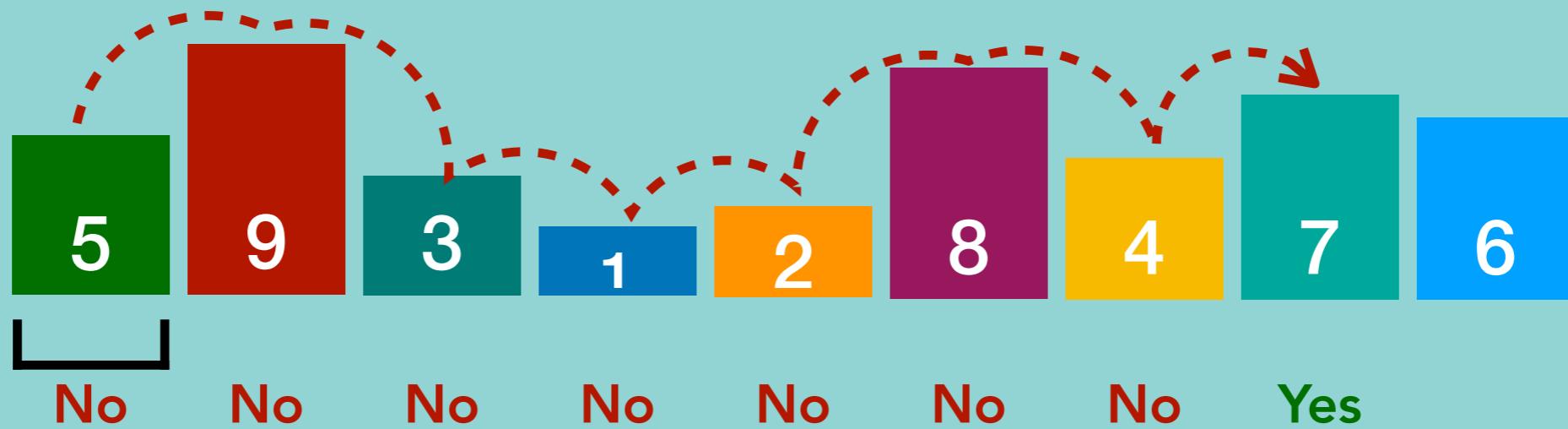
Absolute value

# Searching Algorithm



# Linear Search

Search for 7



Time complexity :  $O(N)$

Space complexity :  $O(1)$

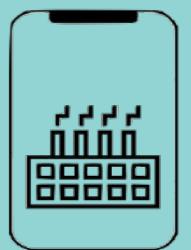
# Linear Search



# Linear Search in Python

# Linear Search Pseudocode

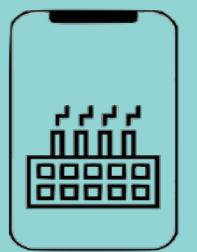
- Create function with two parameters which are an array and a value
- Loop through the array and check if the current array element is equal to the value
- If it is return the index at which the element is found
- If the value is never found return -1



# Binary Search

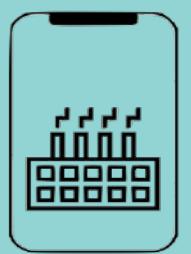
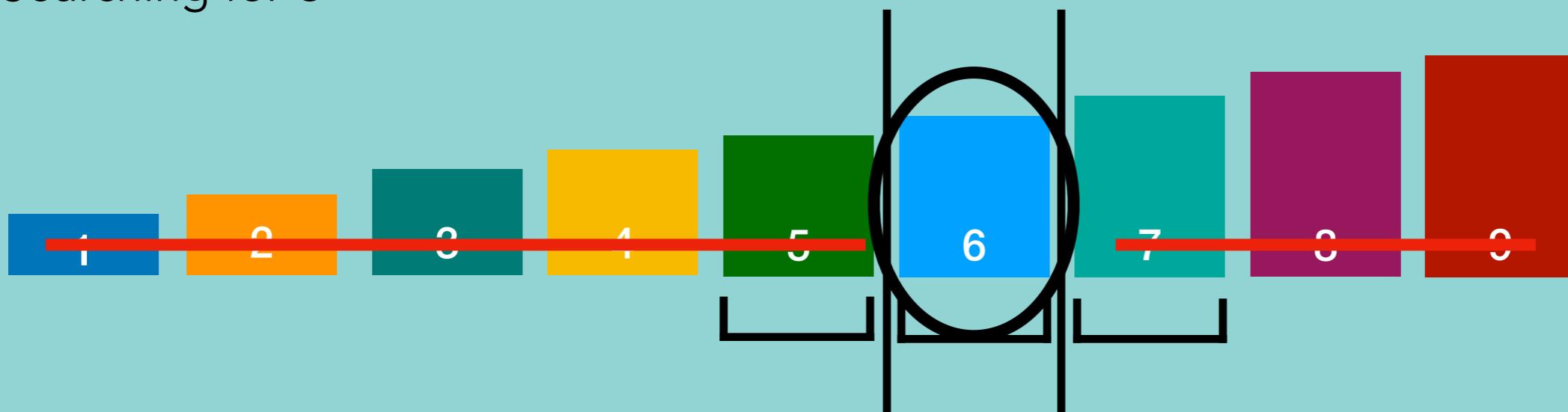
# Binary Search

- Binary Search is faster than Linear Search
- Half of the remaining elements can be eliminated at a time, instead of eliminating them one by one
- Binary Search only works for **sorted arrays.**



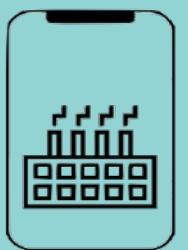
# Binary Search

Searching for **6**



# Binary Search Pseudocode

- Create function with two parameters which are a sorted array and a value
- Create two pointers : a left pointer at the start of the array and a right pointer at the end of the array.
- Based on left and right pointers calculate middle pointer
- While middle is not equal to the value and start<=end loop:
  - if the middle is greater than the value move the right pointer down
  - if the middle is less than the value move the left pointer up
- If the value is never found return -1



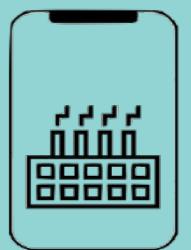
# Binary Search Time Complexity

Worst and Average Case

$O(\log n)$

Best Case

$O(1)$



# Binary Search Time Complexity

Searching for 12

[2,4,5,9,11,13,14,15,19,20,22,23,27,30,32,39]

Step 1 [2,4,5,9,11,13,14,15,19,20,22,23,27,30,32,39]

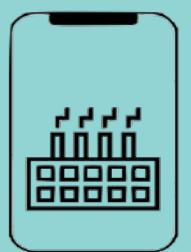
Step 2 [2,4,5,9,11,13,14]

Step 3 [11,13,14]

Not Found

16 elements = 4 Steps

Step 4 [11]



# Binary Search Time Complexity

## Searching for 72

[2,4,5,9,11,13,14,15,19,20,22,23,27,30,32,39,42,44,45,49,51,53,54,55,59,60,62,63,67,70,72,79]

**Step 1** [2,4,5,9,11,13,14,15,19,20,22,23,27,30,32,39,42,44,45,49,51,53,54,55,59,60,62,63,67,70,72,79]

**Step 2** [42,44,45,49,51,53,54,55,59,60,62,63,67,70,72,79]

**Step 3** [59,60,62,63,67,70,72,79]

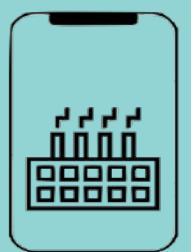
**72 Found**

32 elements = 5 Steps

**Step 4** [67,70,72,79]

$\log_2 N$        $2^x = N$

**Step 5** [72,79]



# Binary Search Time Complexity

Searching for 12

[2,4,5,9,11,13,14,15,19,20,22,23,27,30,32,39]

Step 1 [2,4,5,9,11,13,14,15,19,20,22,23,27,30,32,39]

Step 2 [2,4,5,9,11,13,14]

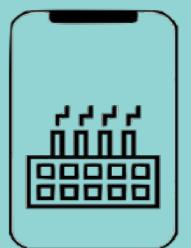
Step 3 [11,13,14]

Not Found

16 elements = 4 Steps

Step 4 [11]

$$\log_2 16 = 2^x = 16 = 2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16$$



# Binary Search Time Complexity

## Searching for 72

[2,4,5,9,11,13,14,15,19,20,22,23,27,30,32,39,42,44,45,49,51,53,54,55,59,60,62,63,67,70,72,79]

**Step 1** [2,4,5,9,11,13,14,15,19,20,22,23,27,30,32,39,42,44,45,49,51,53,54,55,59,60,62,63,67,70,72,79]

**Step 2** [42,44,45,49,51,53,54,55,59,60,62,63,67,70,72,79]

**Step 3** [59,60,62,63,67,70,72,79]

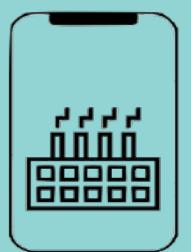
**72 Found**

32 elements = 5 Steps

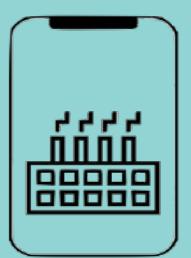
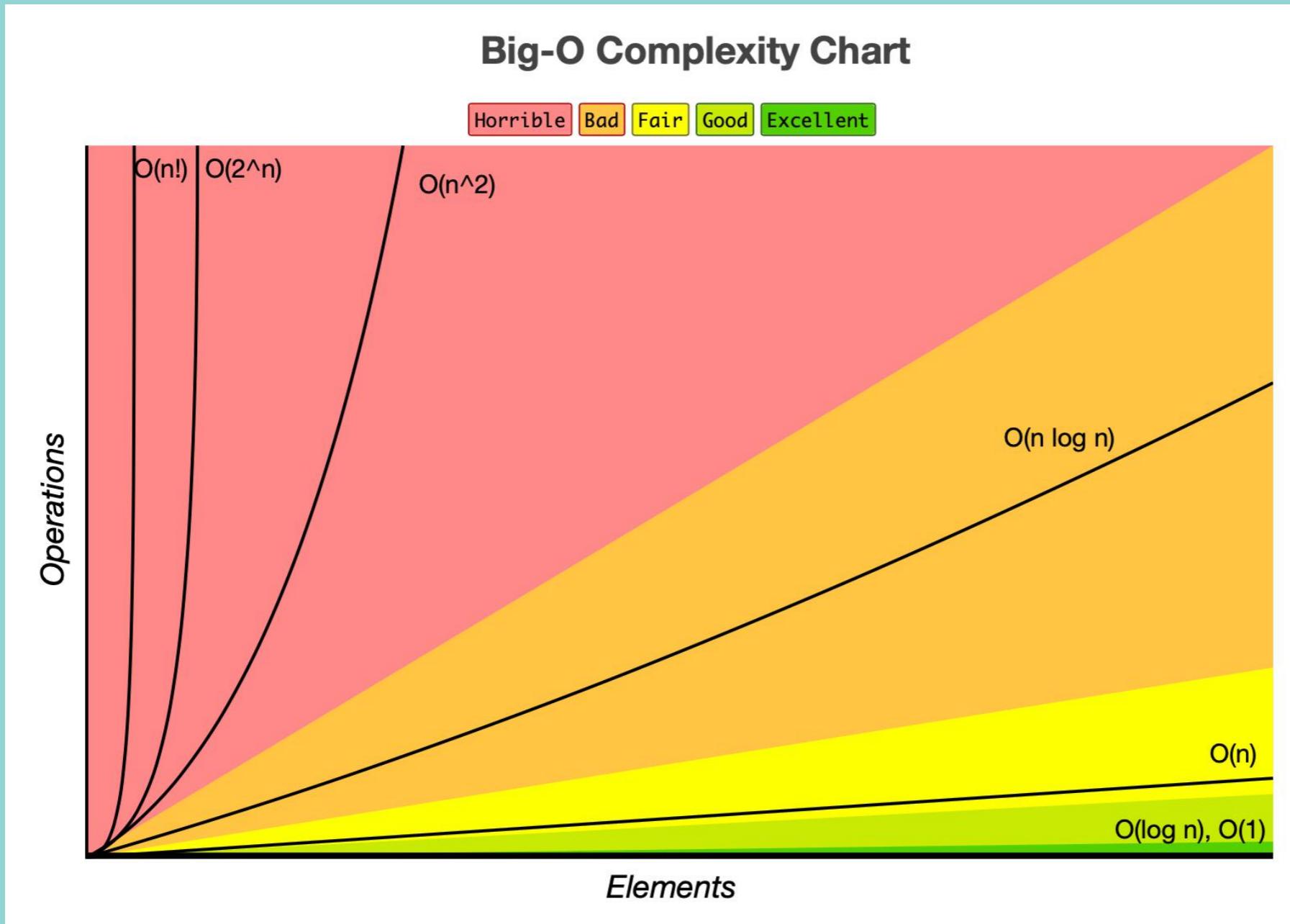
**Step 4** [67,70,72,79]

$$\log_2 32 = 2^x = 32 = 2^5 = 2 * 2 * 2 * 2 * 2 = 32$$

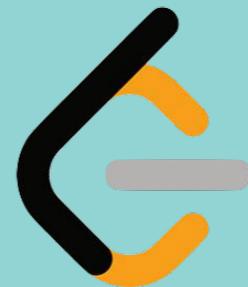
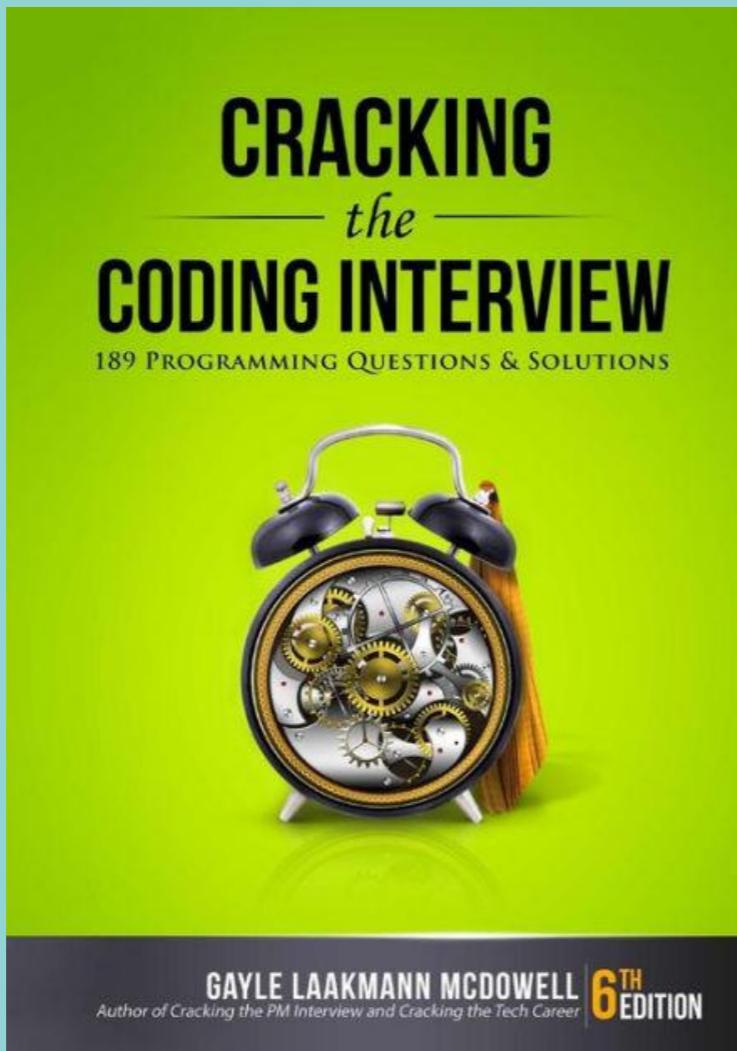
**Step 5** [72,79]



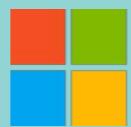
# Binary Search Time Complexity



# Cracking Tree and Graph Interview Questions



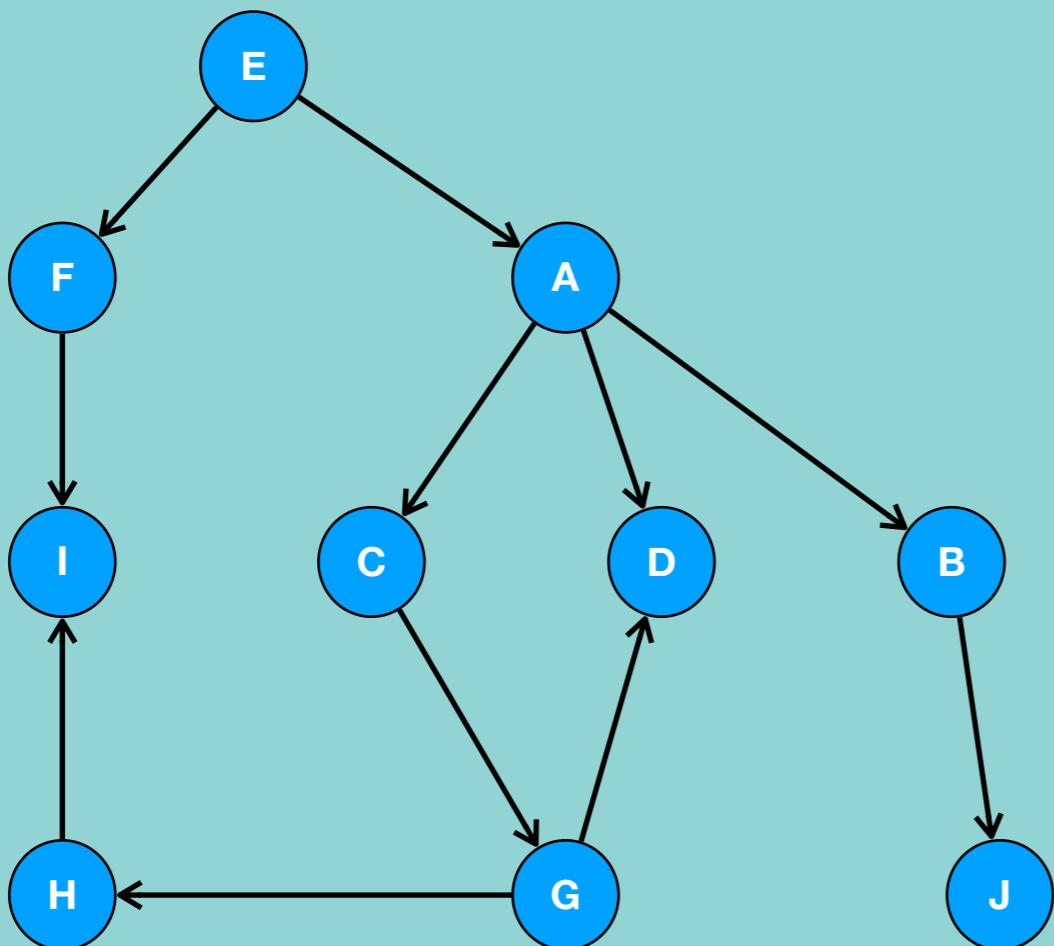
# LeetCode



# Route Between Nodes

## Problem Statement:

Given a directed graph and two nodes (S and E), design an algorithm to find out whether there is a route from S to E.



$A \rightarrow E$

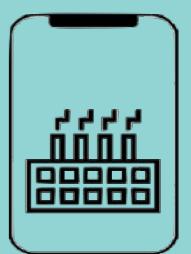
**False**

$A \rightarrow J$

**True**

$A \rightarrow I$

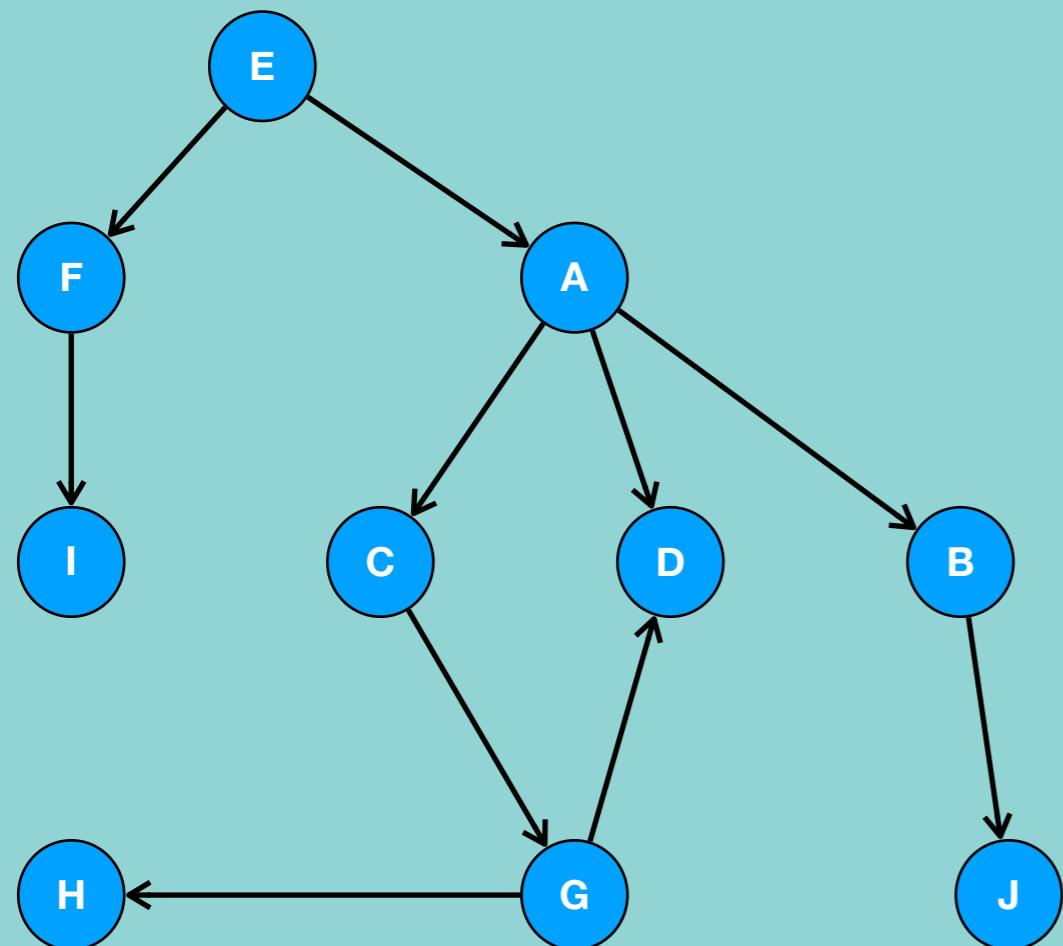
**True**



# Route Between Nodes

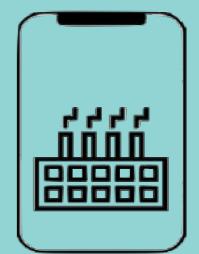
## Problem Statement:

Given a directed graph and two nodes (S and E), design an algorithm to find out whether there is a route from S to E.



## Pseudocode

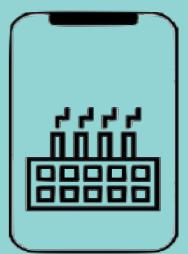
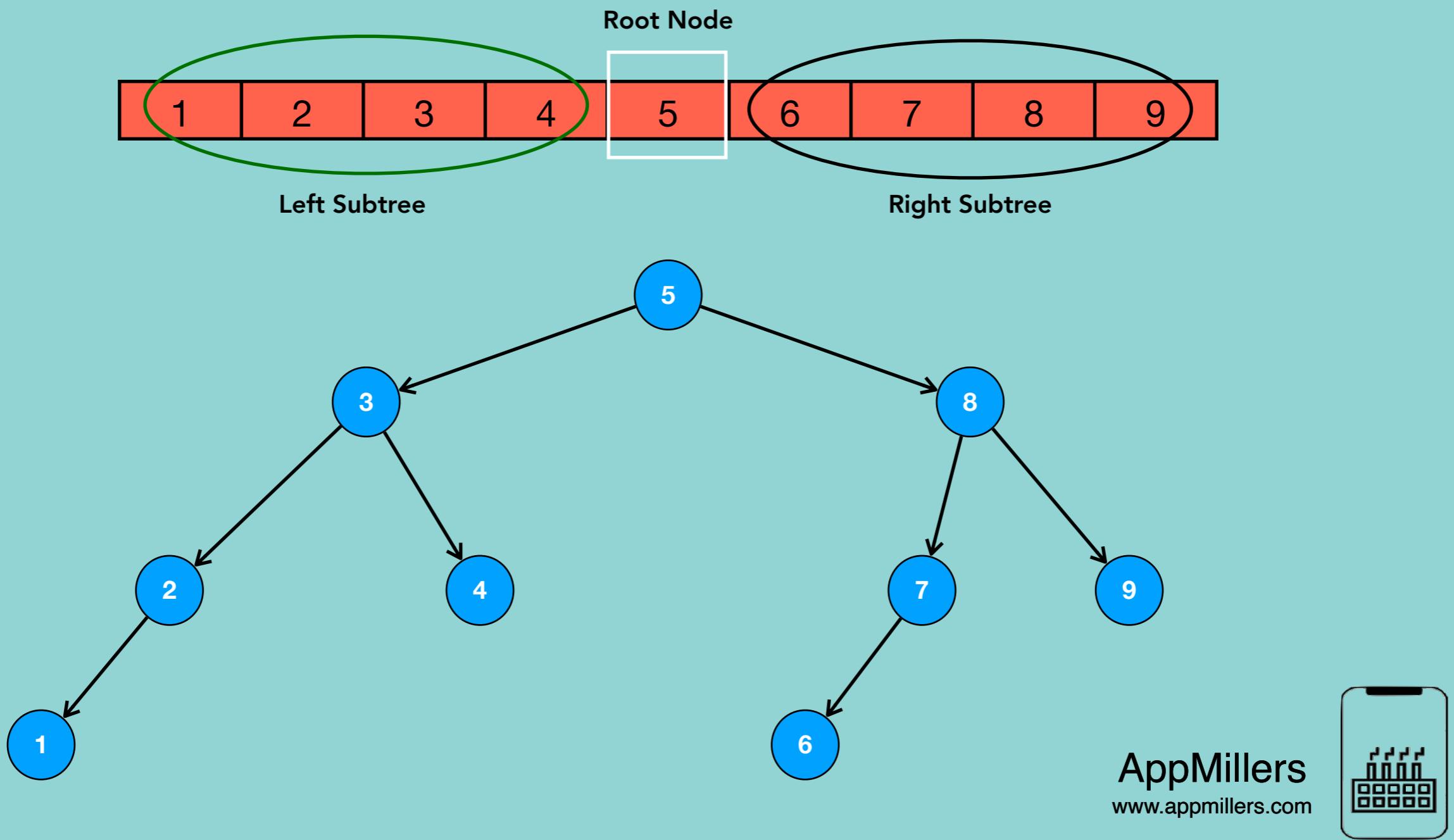
- Create function with two parameters start and end nodes
- Create queue and enqueue start node to it
- Find all the neighbors of the just enqueued node and enqueue them into the queue
- Repeat this process until the end of elements in graph
- If during the above process at some point in time we encounter the destination node, we return True.
- Mark visited nodes as visited



# Minimal Tree

## Problem Statement:

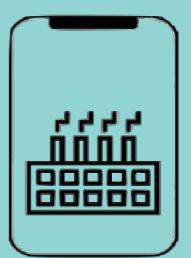
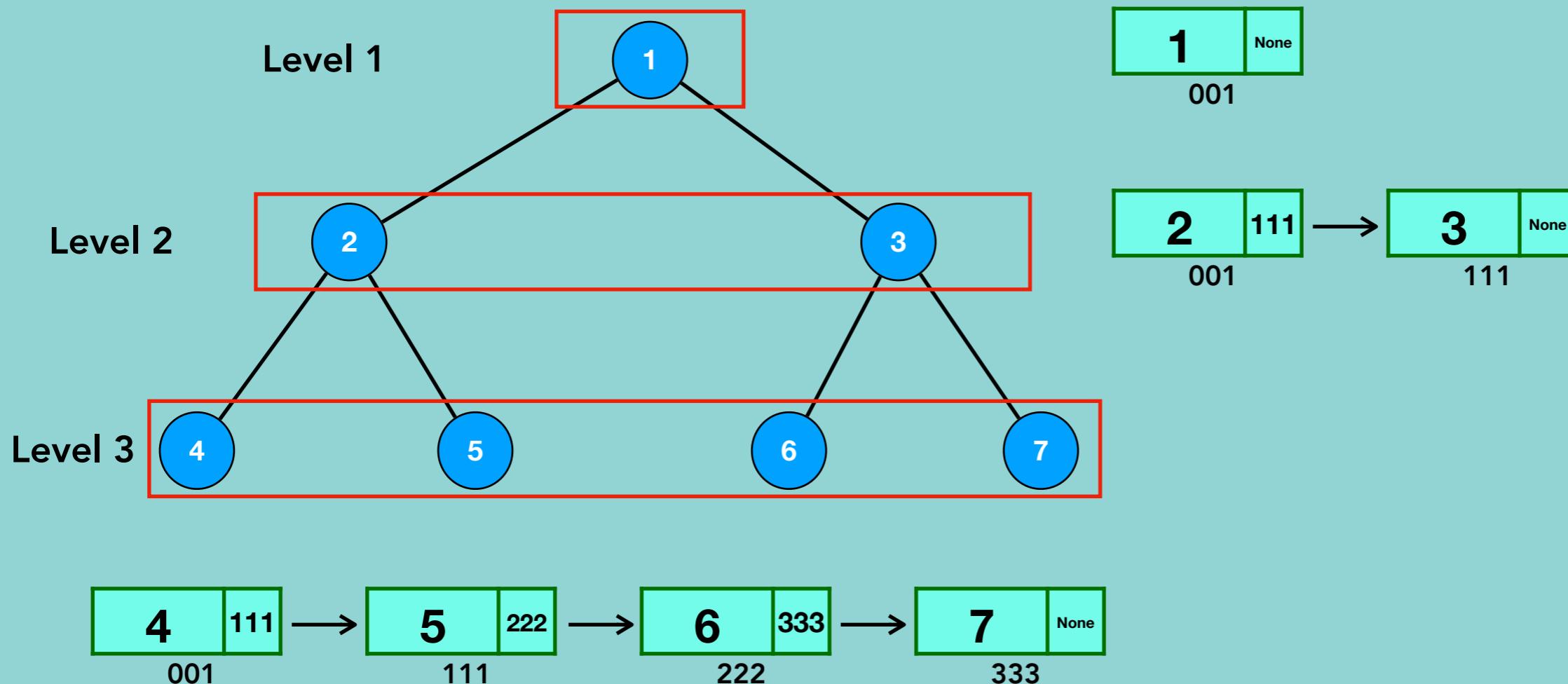
Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.



# List of Depths

## Problem Statement:

Given a binary search tree, design an algorithm which creates a linked list of all the nodes at each depth (i.e., if you have a tree with depth D, you'll have D linked lists)

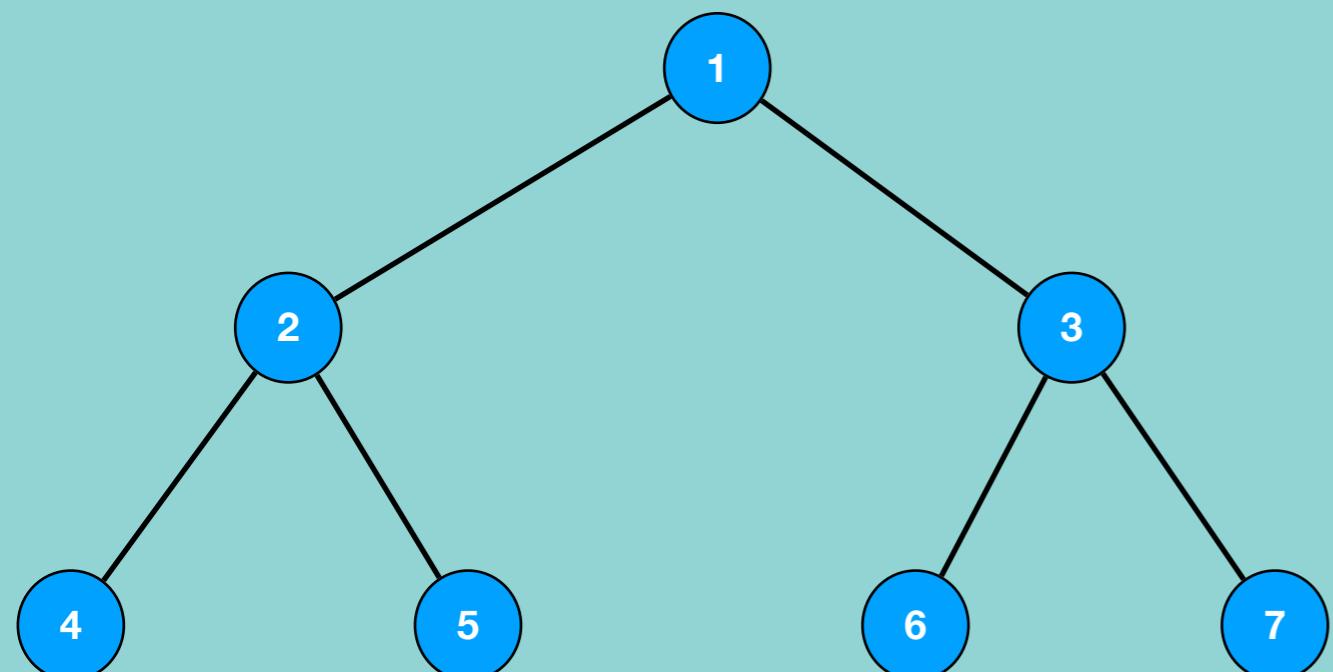


# List of Depths

## Problem Statement:

Given a binary search tree, design an algorithm which creates a linked list of all the nodes at each depth (i.e., if you have a tree with depth D, you'll have D linked lists)

## Pre Order Traversal

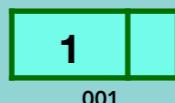


custDict

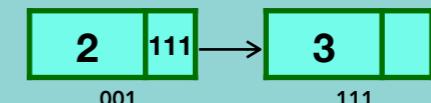
Keys

3

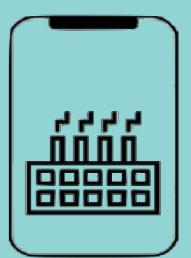
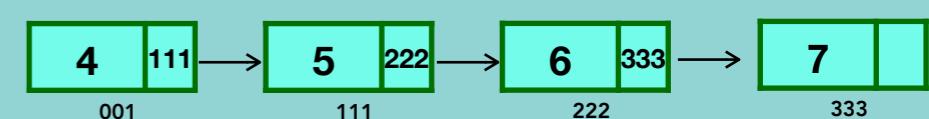
Values



2



1

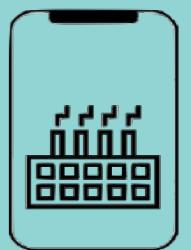
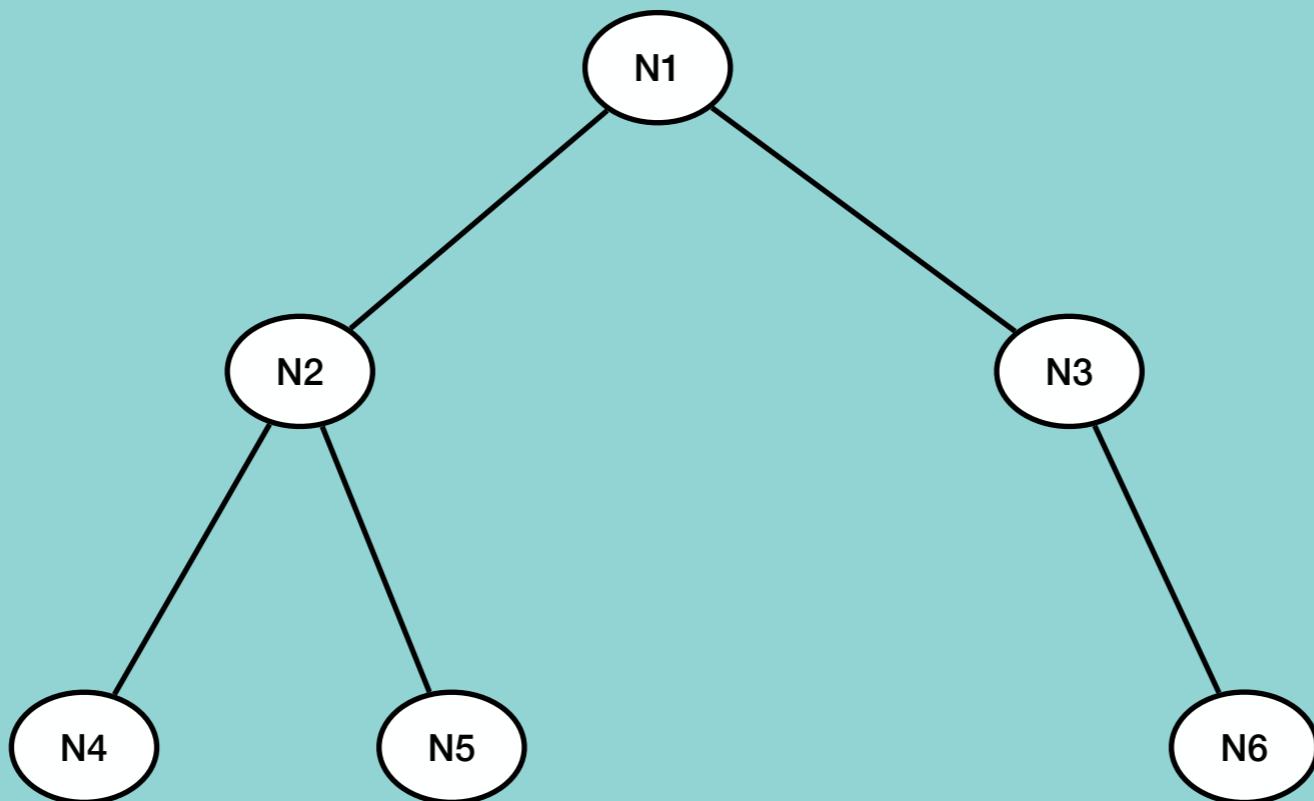


# Check Balanced

## Problem Statement:

Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

## Balanced Binary Tree

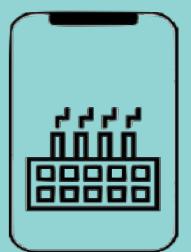
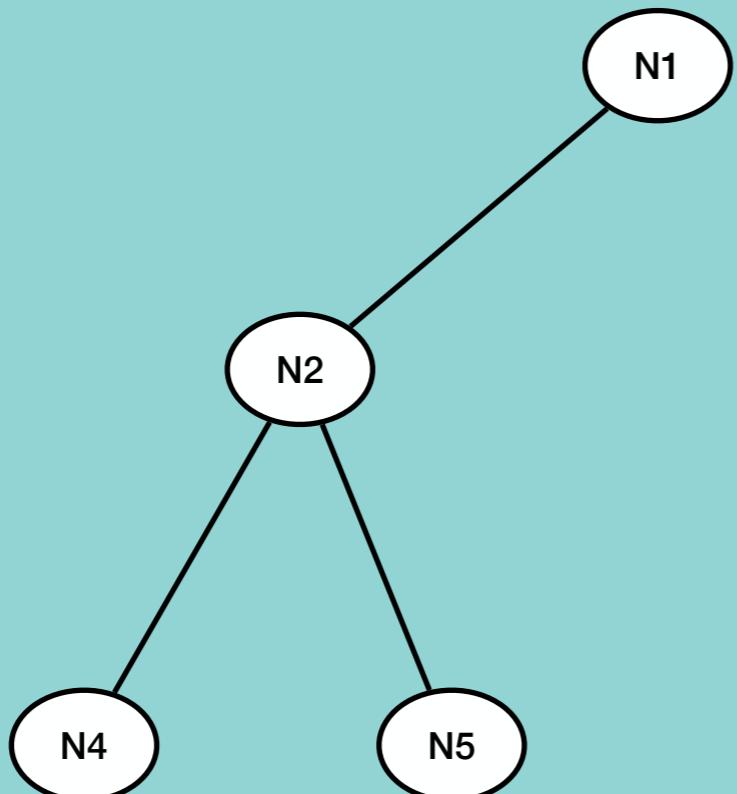


# Check Balanced

## Problem Statement:

Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

## Balanced Binary Tree



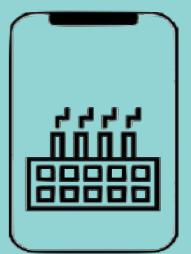
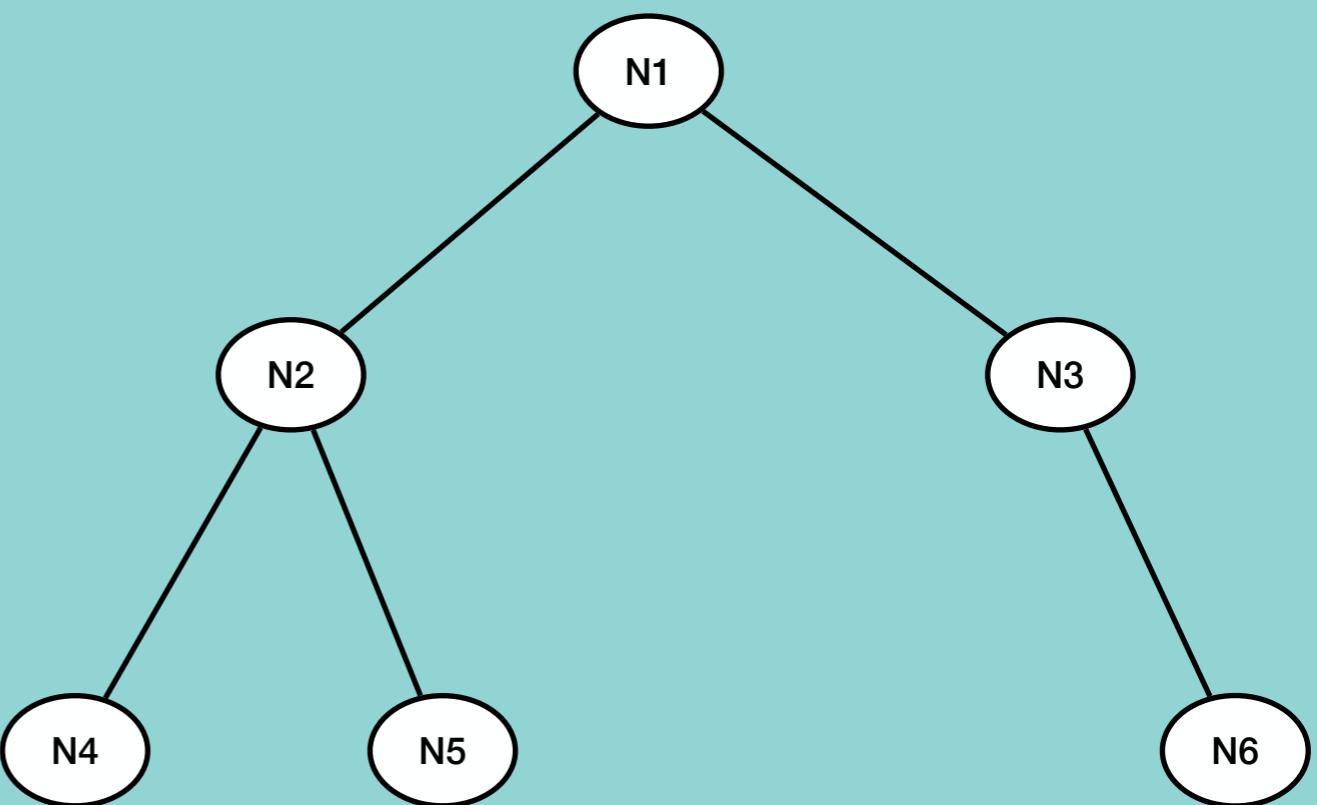
# Check Balanced

## Problem Statement:

Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.

The Binary Tree is Balanced if:

- The right subtree is balanced
- The left subtree is balanced
- The difference between the height of the left subtree and the right subtree is at most 1



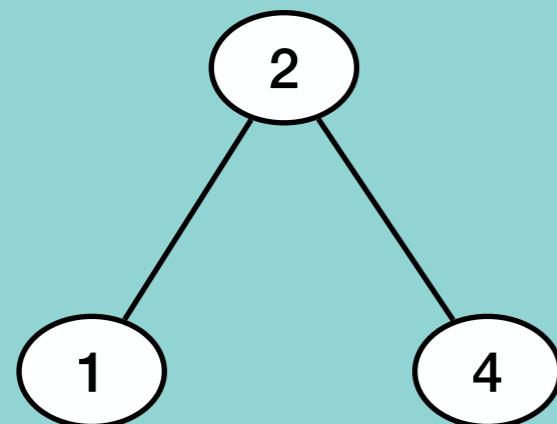
# Validate BST

## Problem Statement:

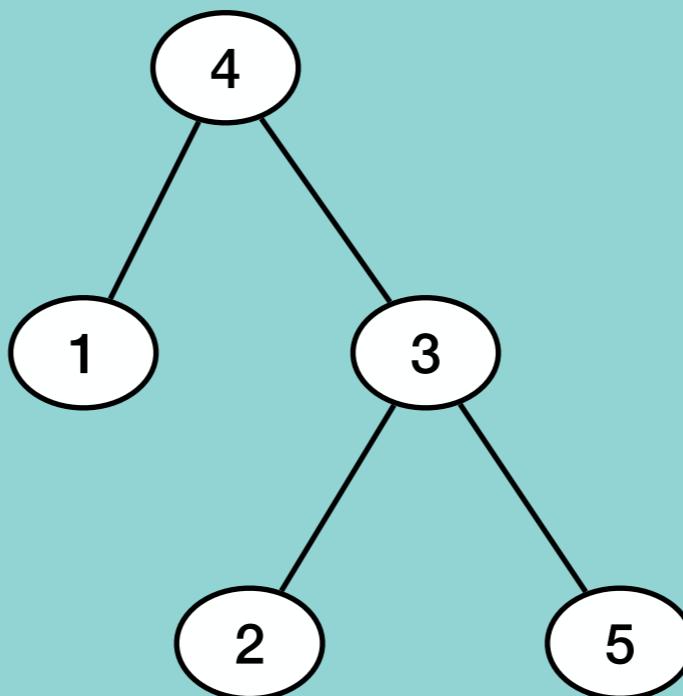
Implement a function to check if a binary tree is a Binary Search Tree.

The Binary Tree is Binary Search Tree if:

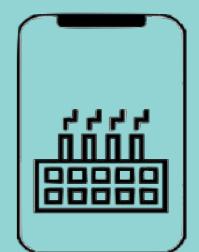
- The left subtree of a node contains only nodes with keys less than the node's key
- The right subtree of a node contains only nodes with keys greater than the node's key
- These conditions are applicable for both left and right subtrees



True



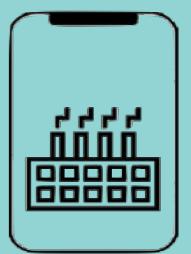
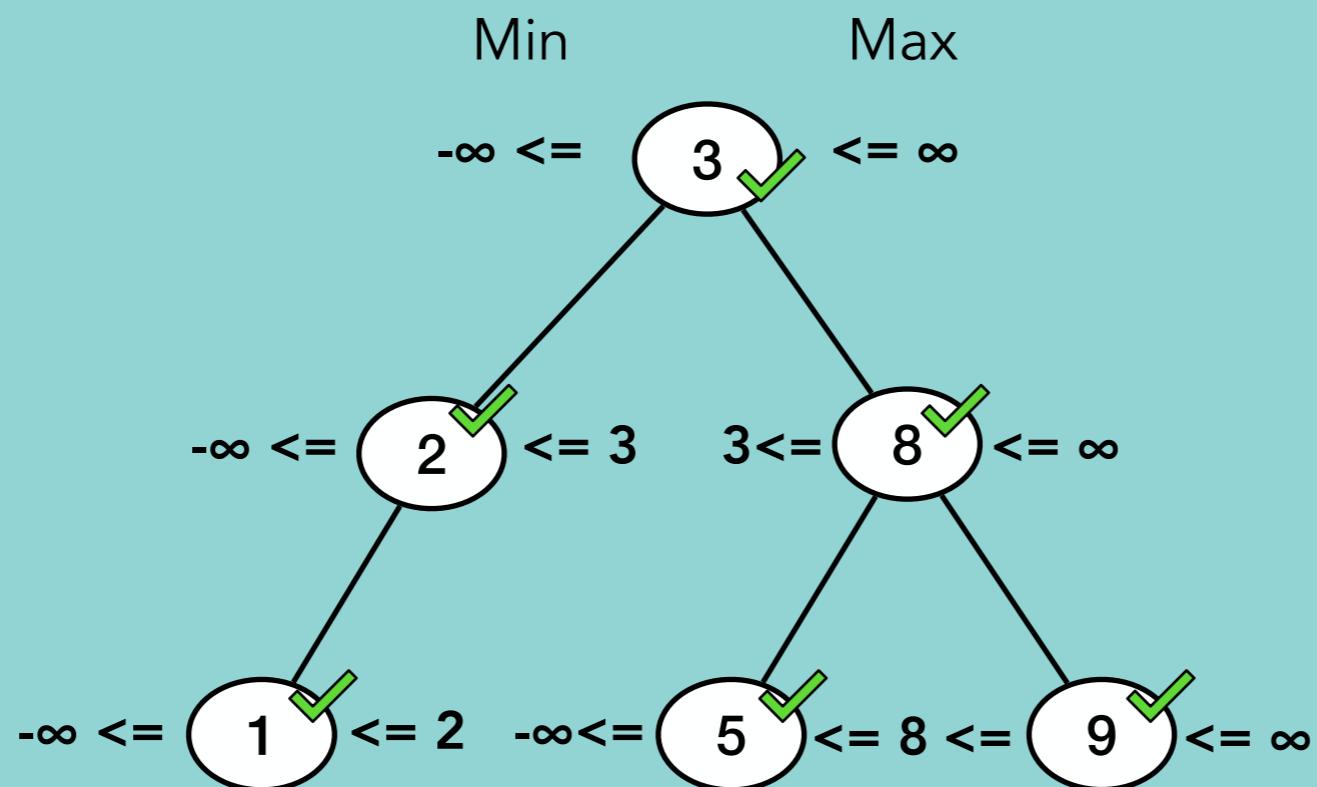
False



# Validate BST

## Problem Statement:

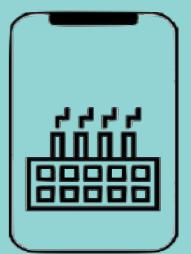
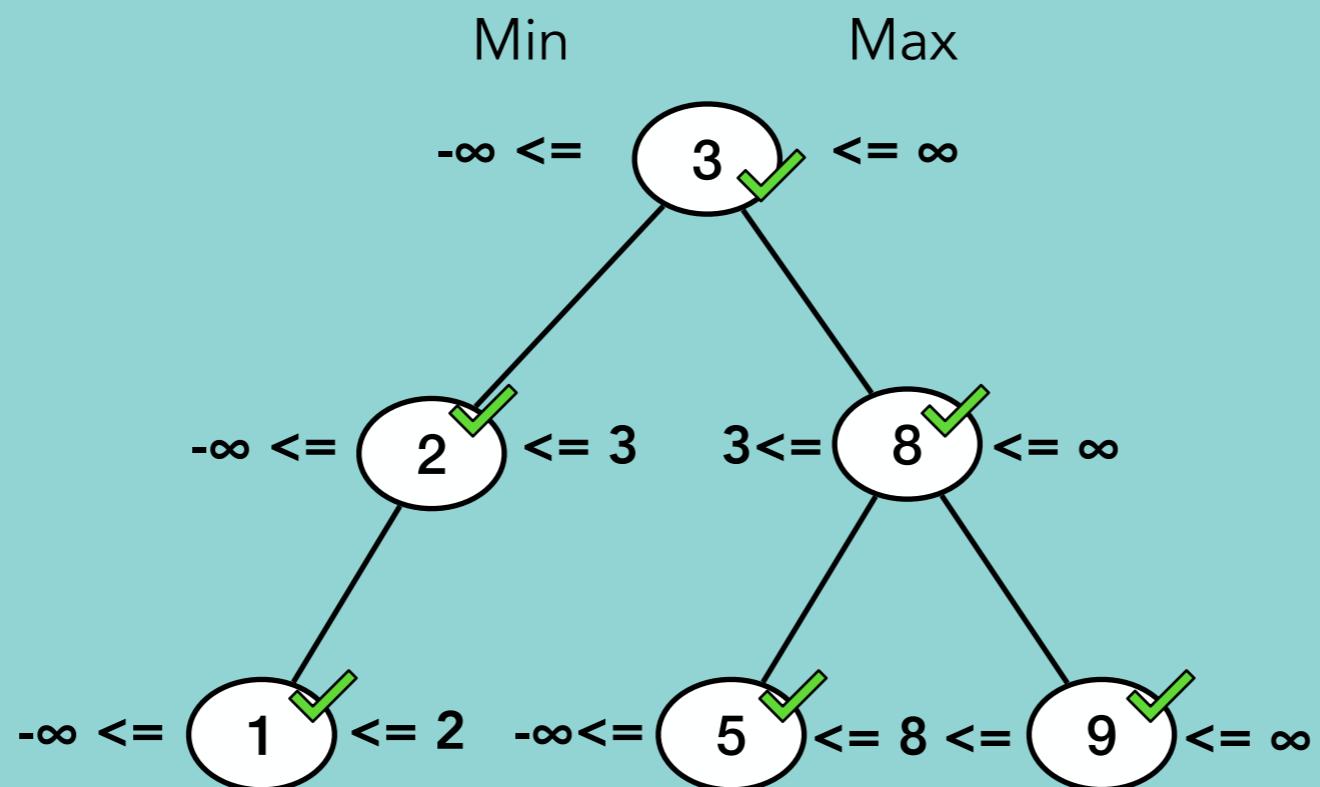
Implement a function to check if a binary tree is a Binary Search Tree.



# Validate BST

## Problem Statement:

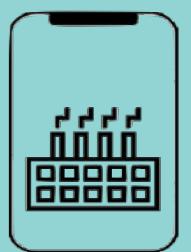
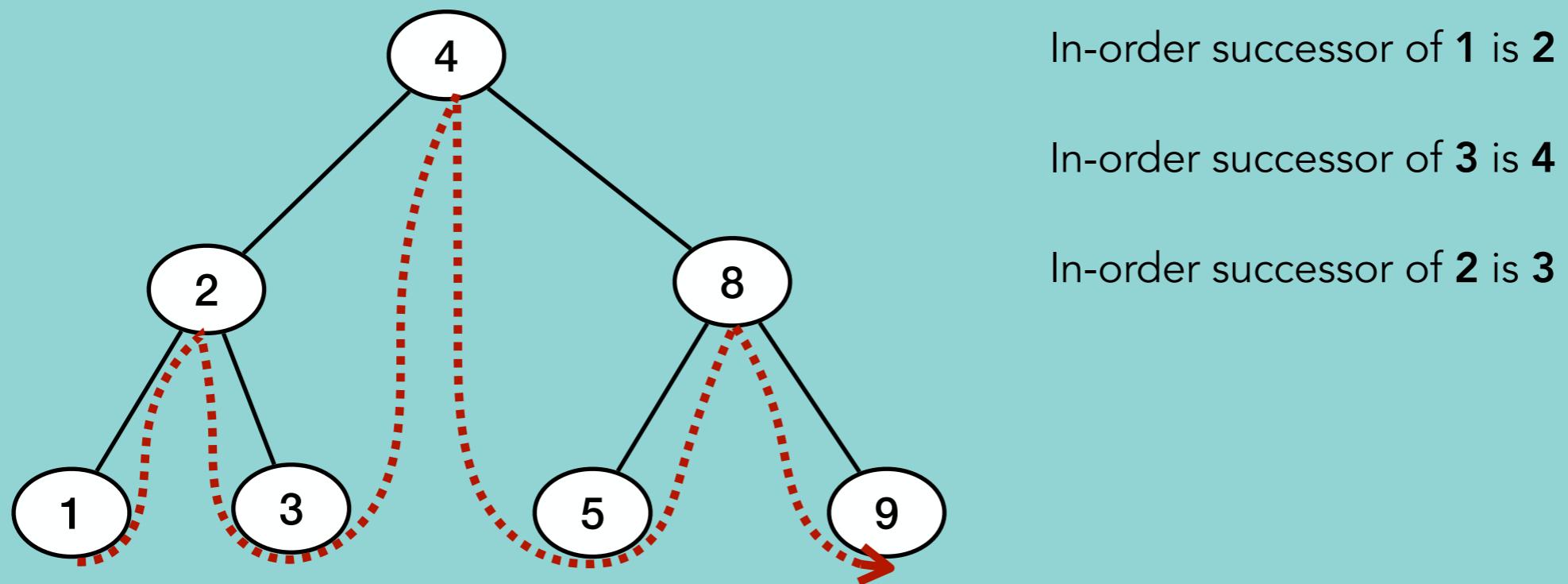
Implement a function to check if a binary tree is a Binary Search Tree.



# Successor

## Problem Statement:

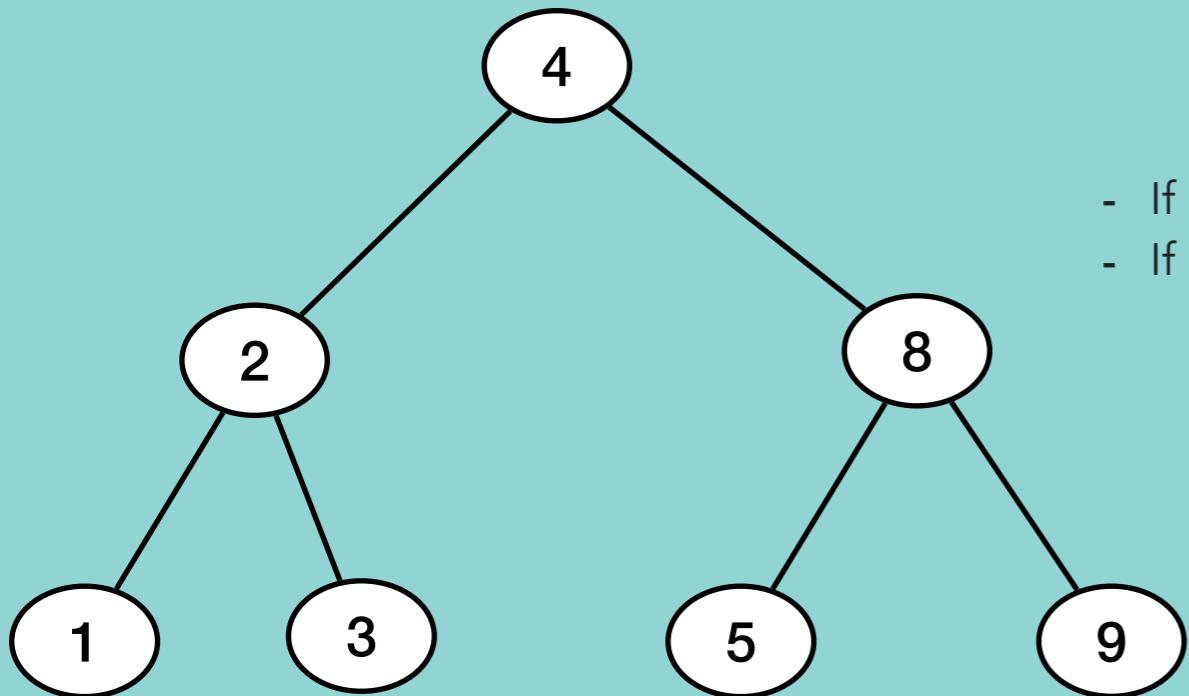
Write an algorithm to find the next node (i.e in-order successor) of given node in a binary search tree. You may assume that each node has a link to its parent.



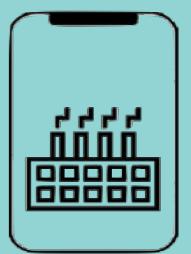
# Successor

## Problem Statement:

Write an algorithm to find the next node (i.e in-order successor) of given node in a binary search tree. You may assume that each node has a link to its parent.



- If right subtree of node is **not None**, then successor lies in right subtree.
- If right subtree of node is **None**, then successor is one of the ancestors.



# Build Order

## Problem Statement:

You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project's dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.

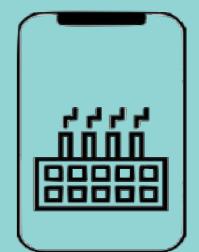
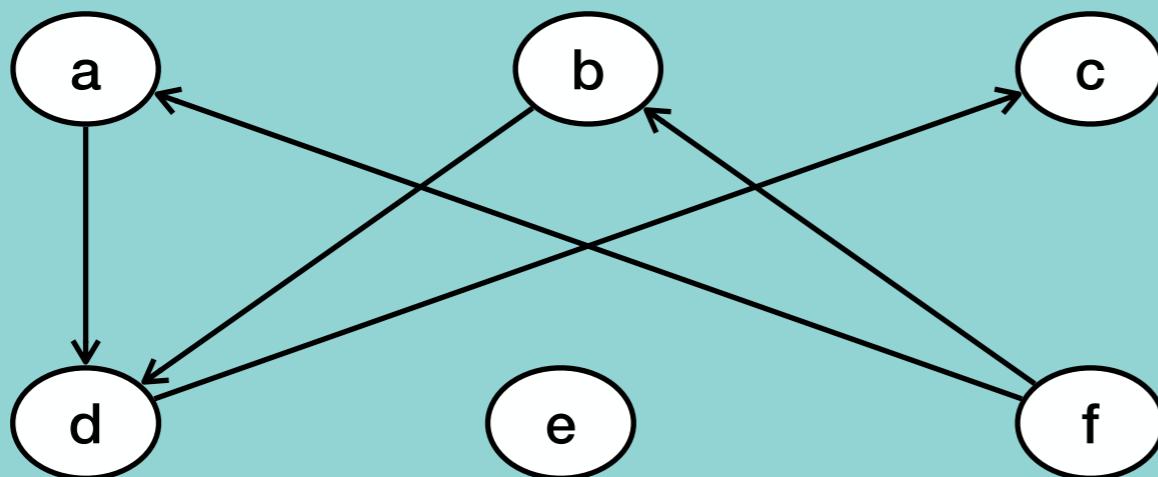
### **Input:**

Projects : a, b, c, d, e, f

Dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)

### **Output:**

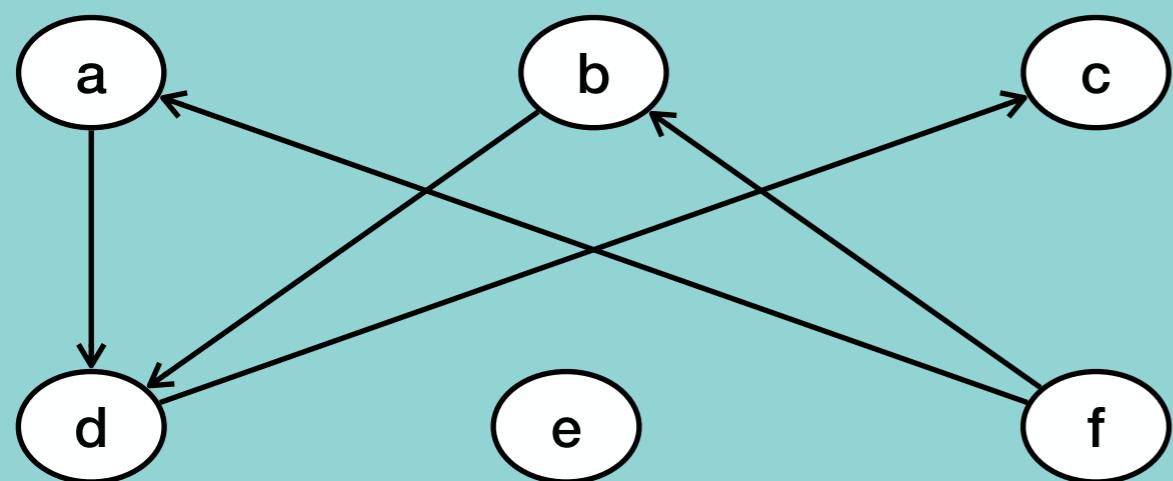
e, f, a, b, d, c



# Build Order

## Problem Statement:

You are given a list of projects and a list of dependencies (which is a list of pairs of projects, where the second project is dependent on the first project). All of a project's dependencies must be built before the project is. Find a build order that will allow the projects to be built. If there is no valid build order, return an error.



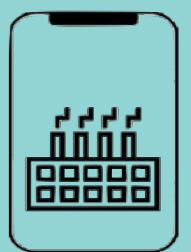
Find nodes with dependencies

a, b, c, d

Find nodes **without** dependencies

e, f

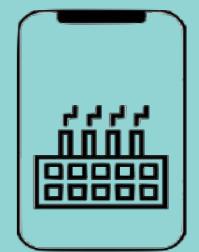
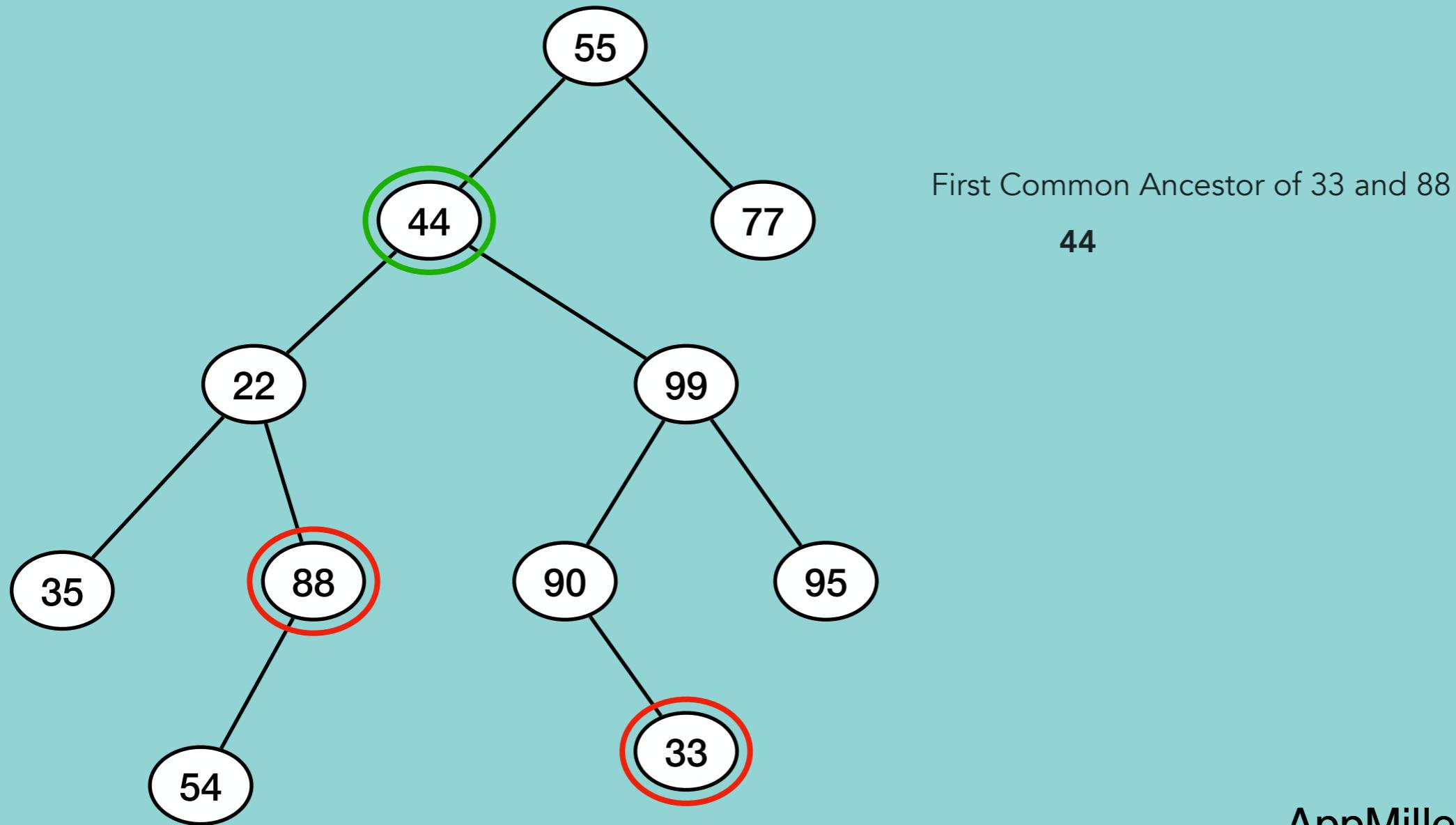
Find order by taking out nodes **without** dependencies.



# First Common Ancestor

## Problem Statement:

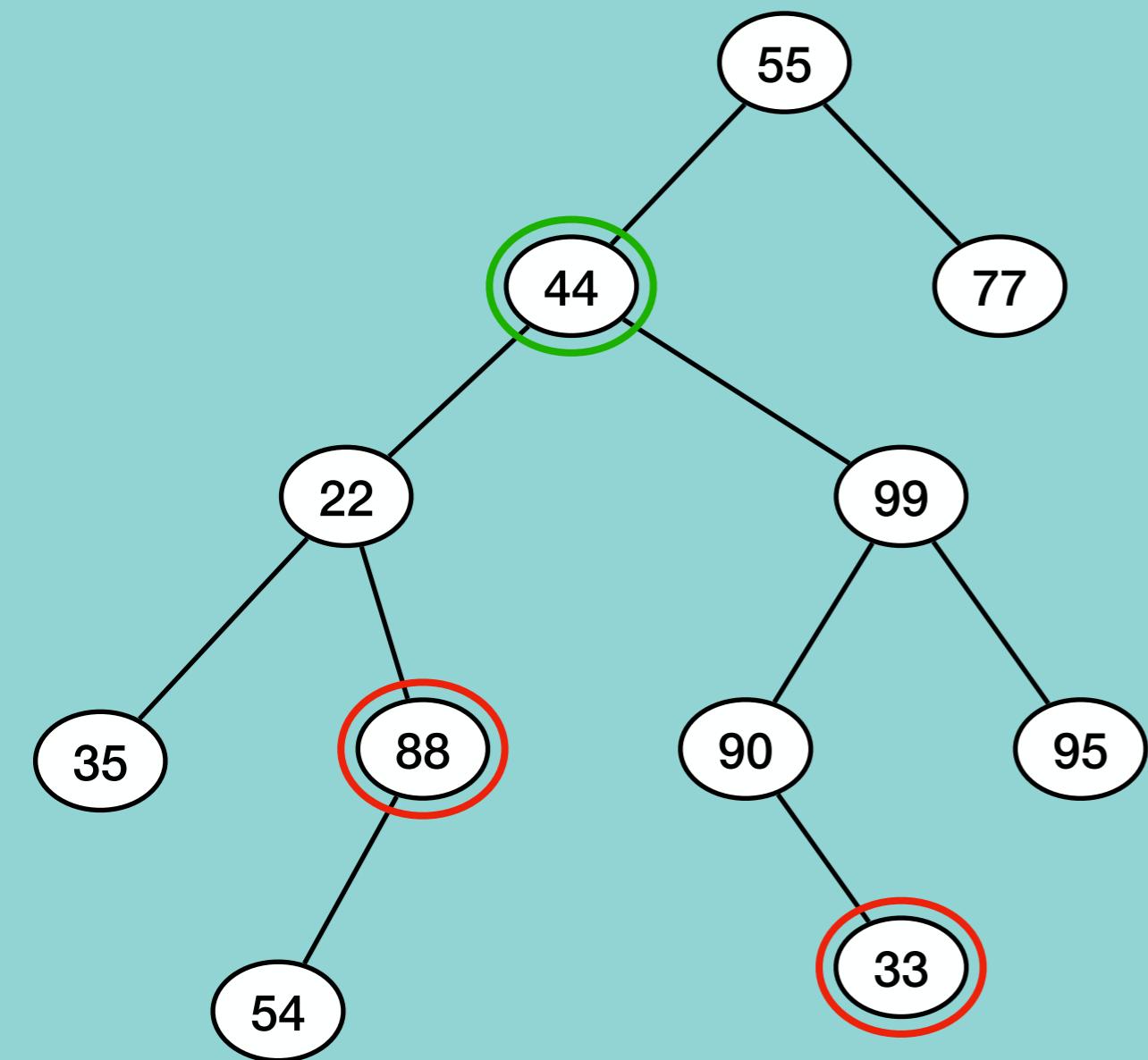
Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.



# First Common Ancestor

## Problem Statement:

Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.



findNodeinTree()

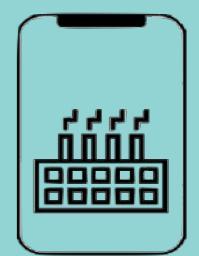
findNodeinTree(88, left-44) → True

findNodeinTree(33, left-44) → True

findNodeinTree(88, left-22) → True

findNodeinTree(33, left-22) → False

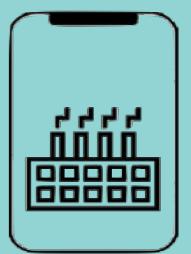
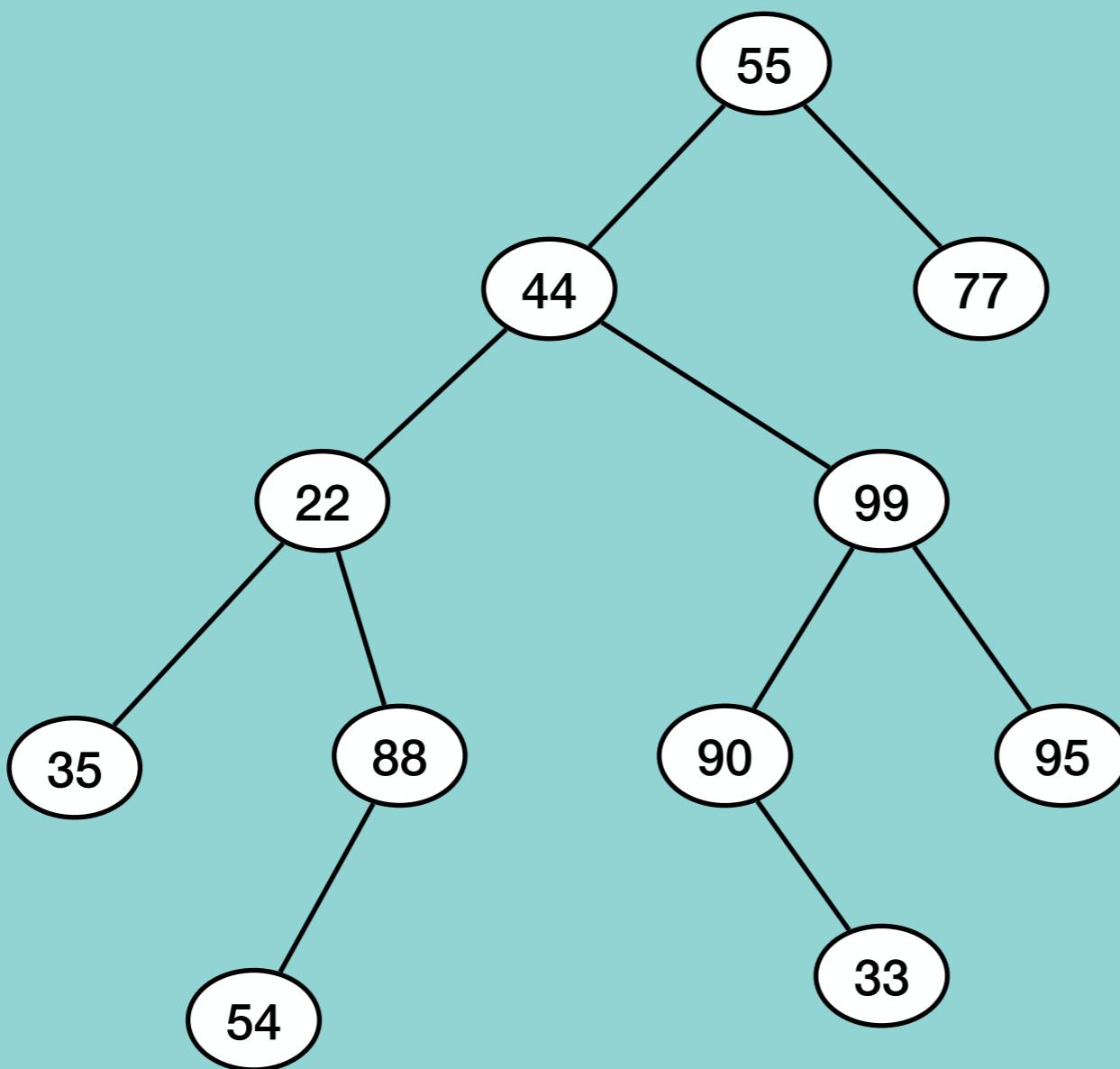
**Return root (44)**



# First Common Ancestor

## Problem Statement:

Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.





# Data Structures and Algorithms in Swift

Implement Stacks, Queues,  
Dictionaries, and Lists in Your Apps

—  
Elshad Karimov

Apress®