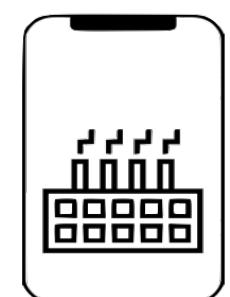


# **Java Data Structures and Algorithms Masterclass**

**Learning Data Structures and Algorithms**

Elshad Karimov from AppMillers



# Java Data Structures and Algorithms Masterclass

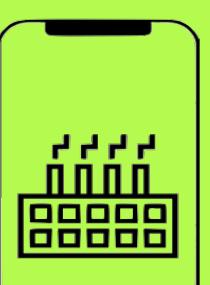
43 Sections

400+ Lectures

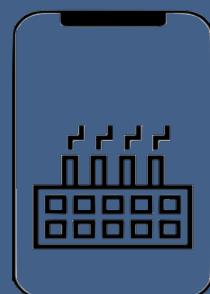
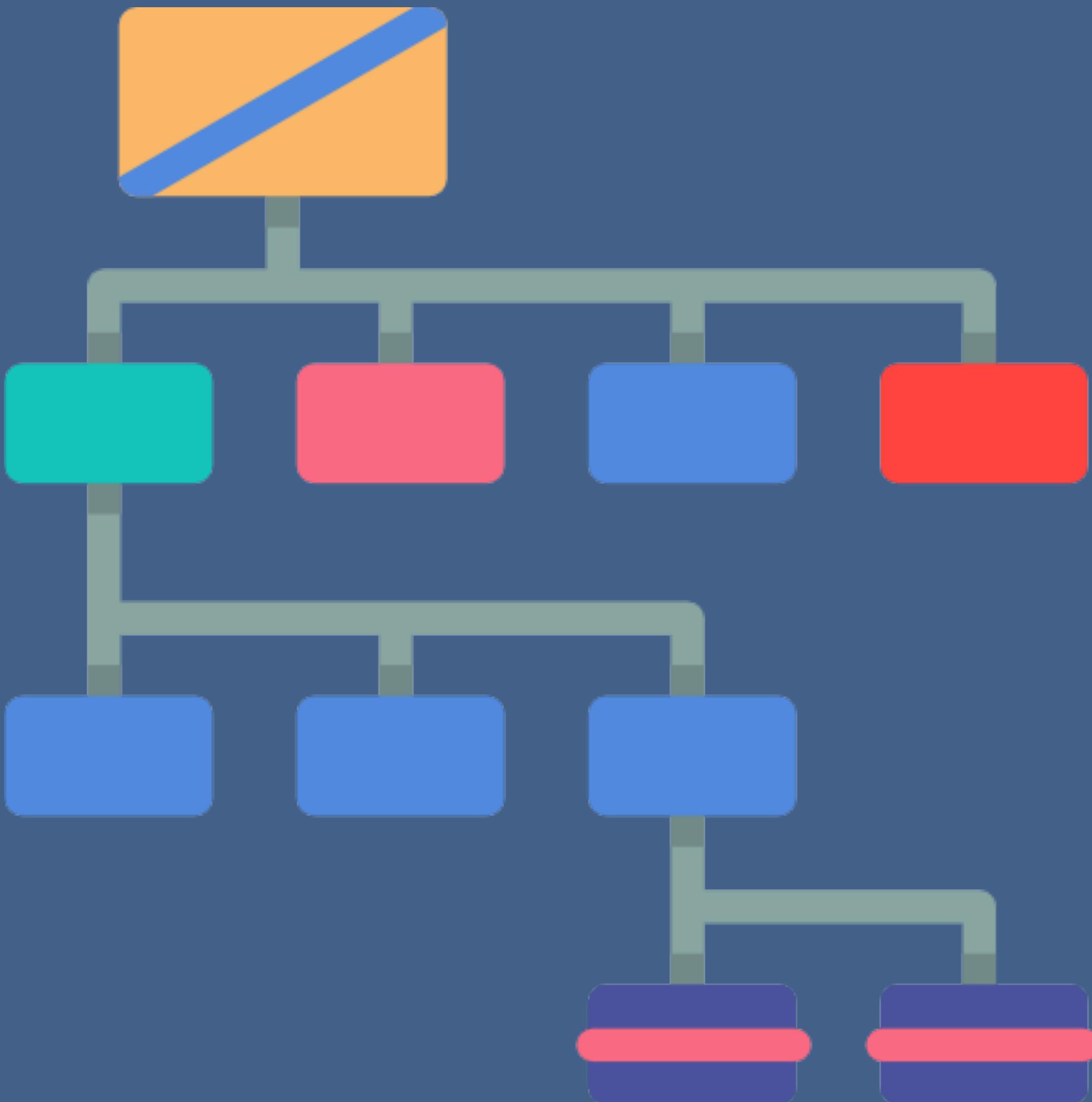
45+ hours of Content

100+ Top Interview Questions

100+ Downloadable resources

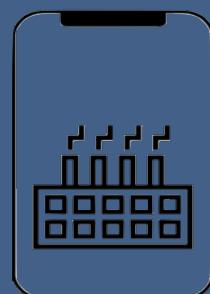


# What is a Data Structure?



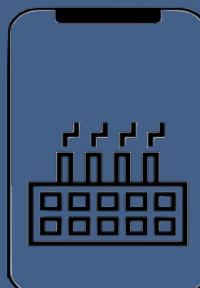
# What is a Data Structure?

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



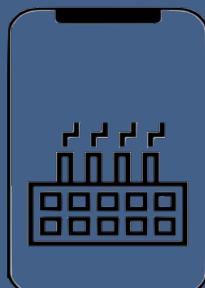
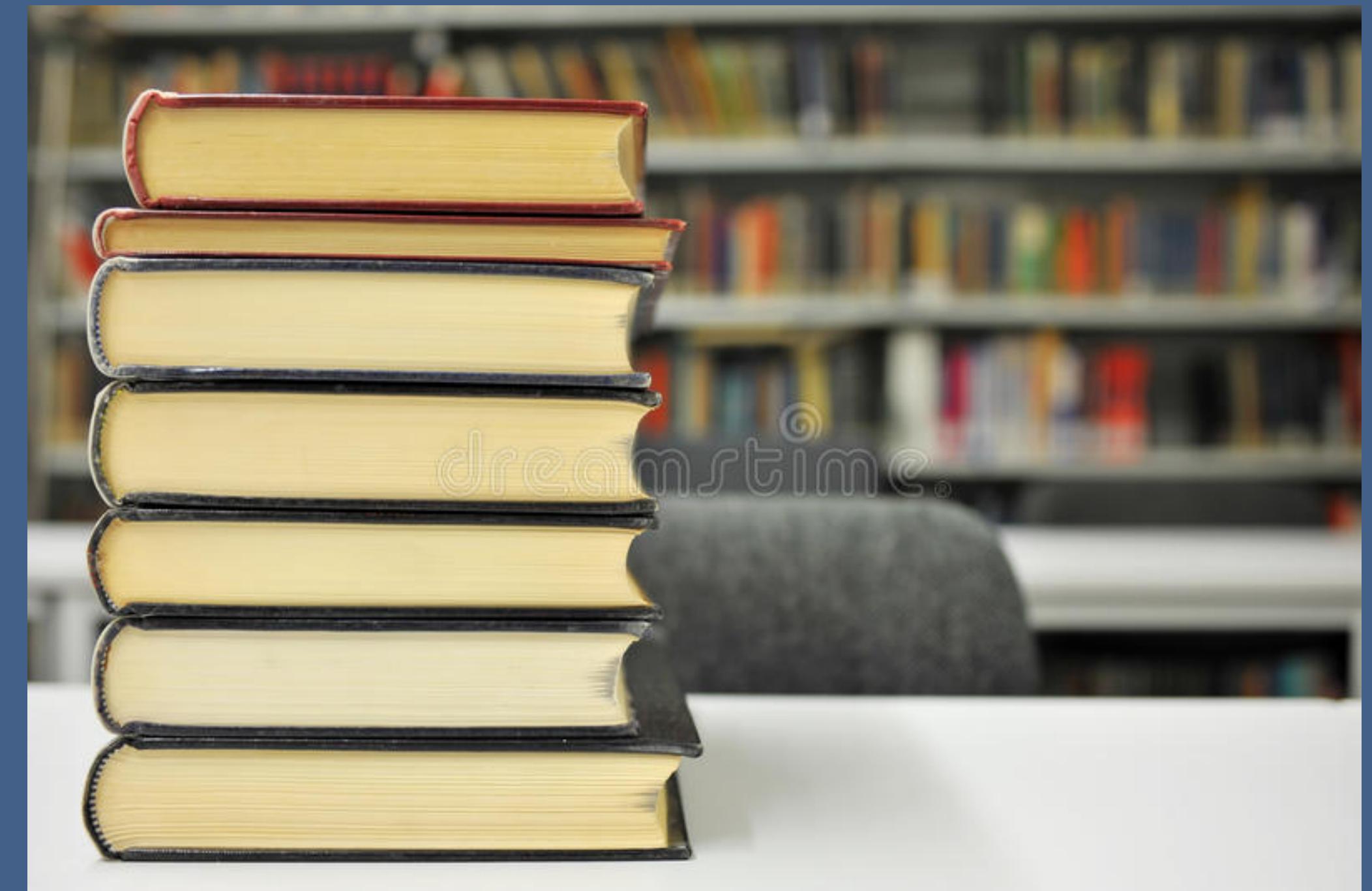
# What is a Data Structure?

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

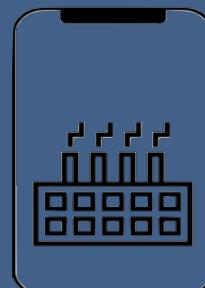
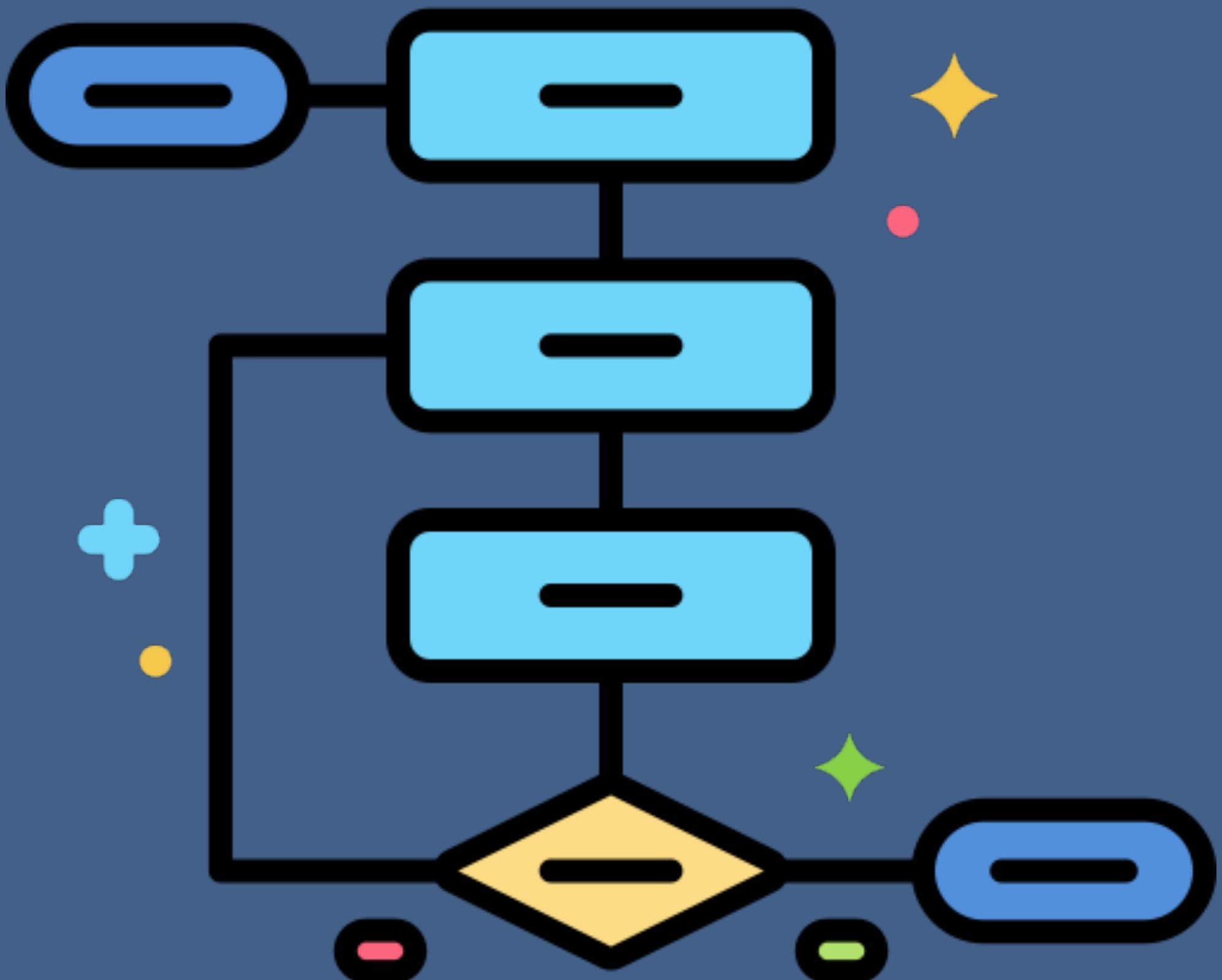


# What is a Data Structure?

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

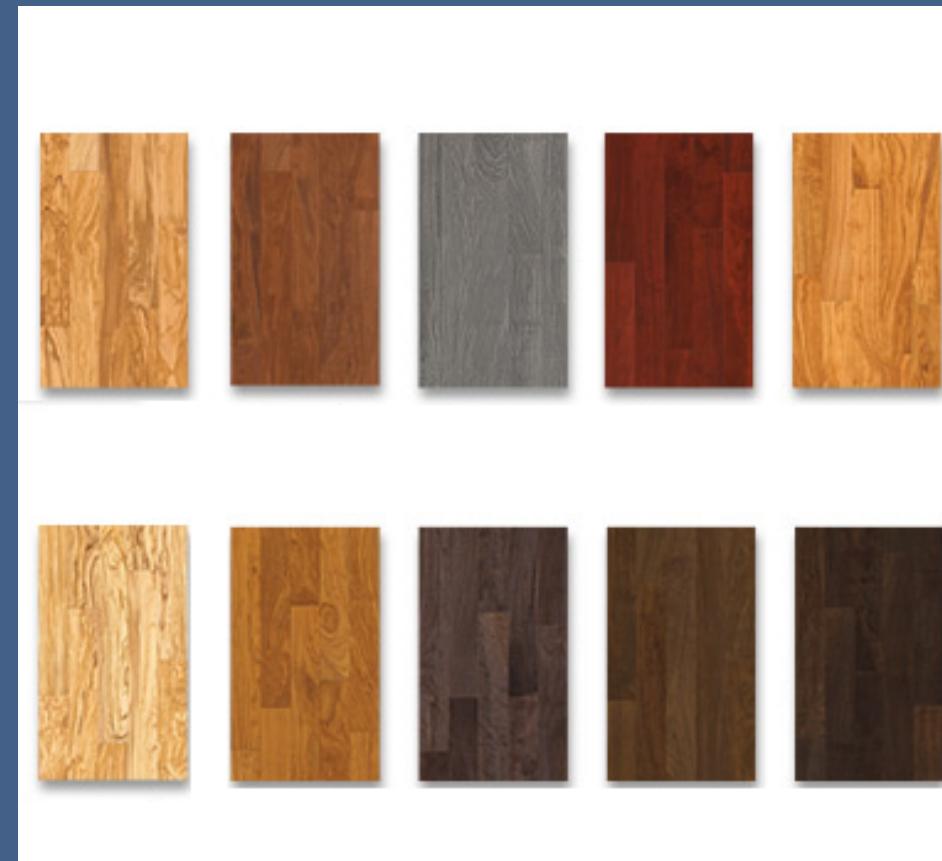


# What is an Algorithm?

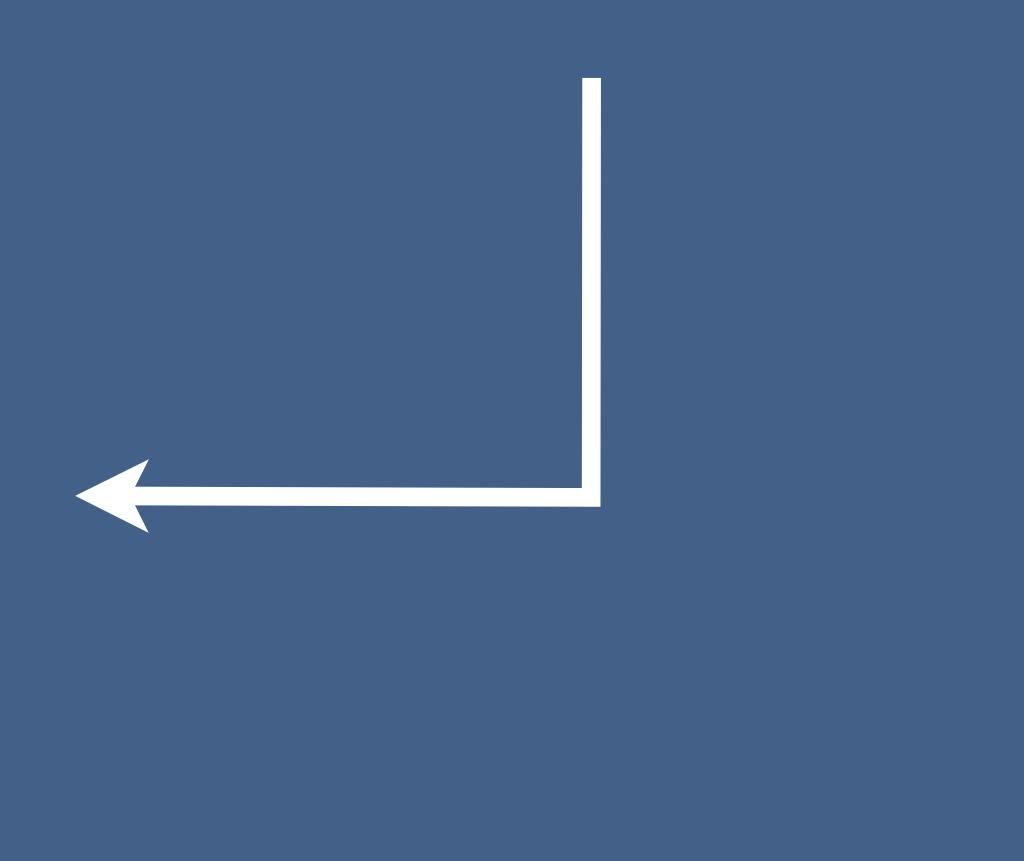
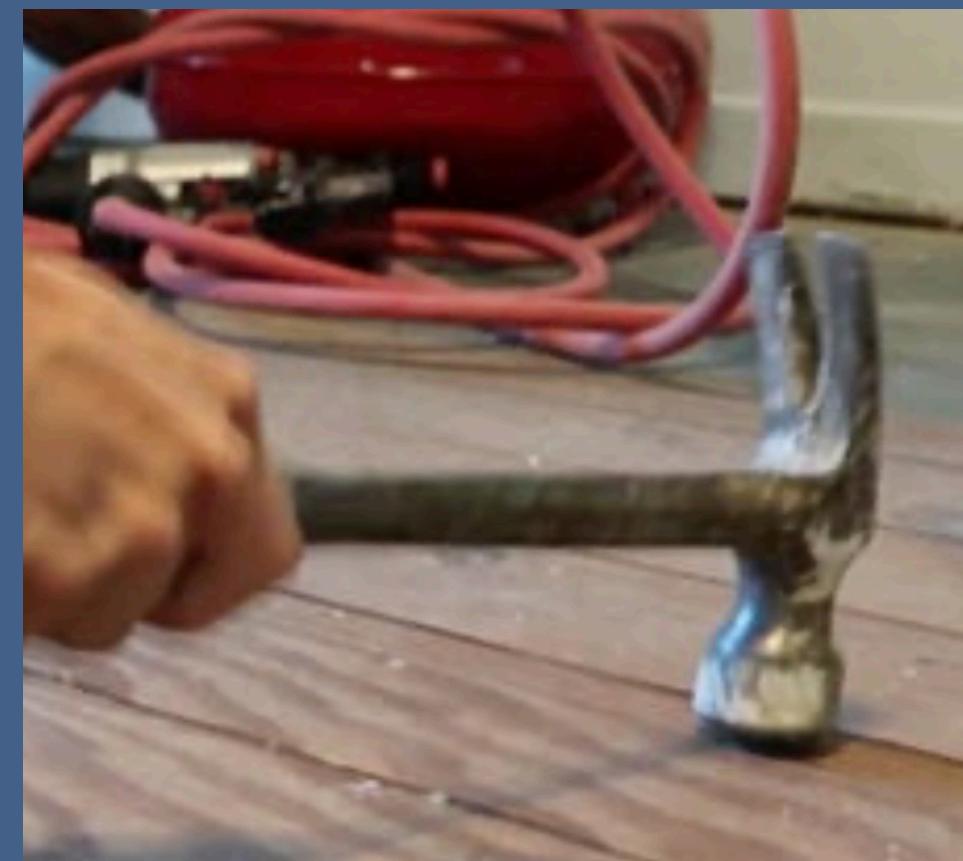
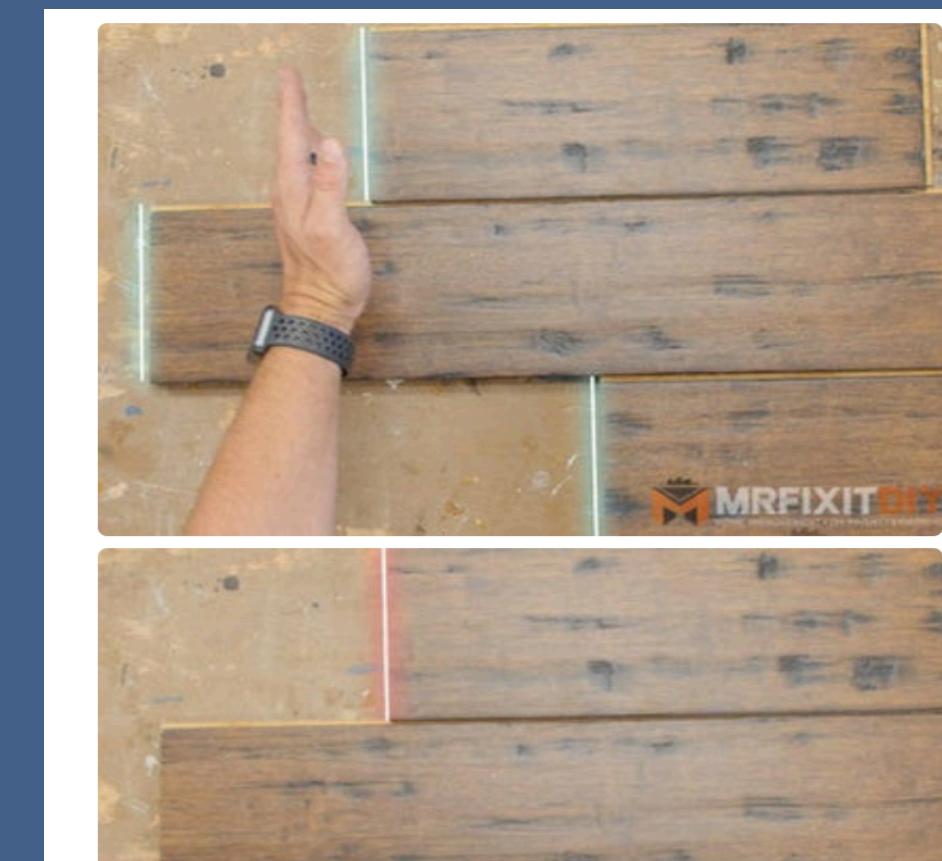


# What is an Algorithm?

Set of instructions to perform a Task



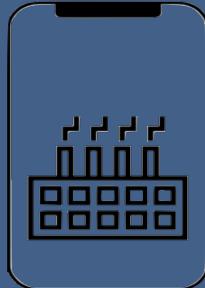
Step 1: Choosing flooring



Step 5: Trim door casing

Step 4: Determine the layout

Step 3: Prepare sub flooring



# What is an Algorithm?

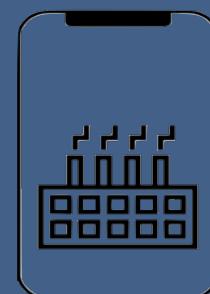
Algorithms in our daily lives



Step 1 : Go to bus stop

Step 2 : Take a bus

Step 3: Go to office



# What is an Algorithm?

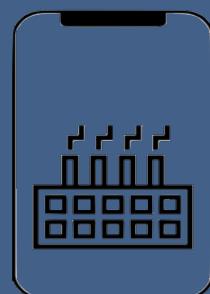
Algorithms in our daily lives



Step 1 : Go to Starbucks

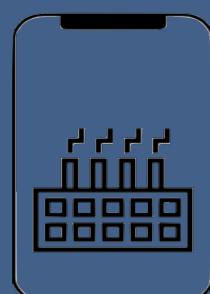
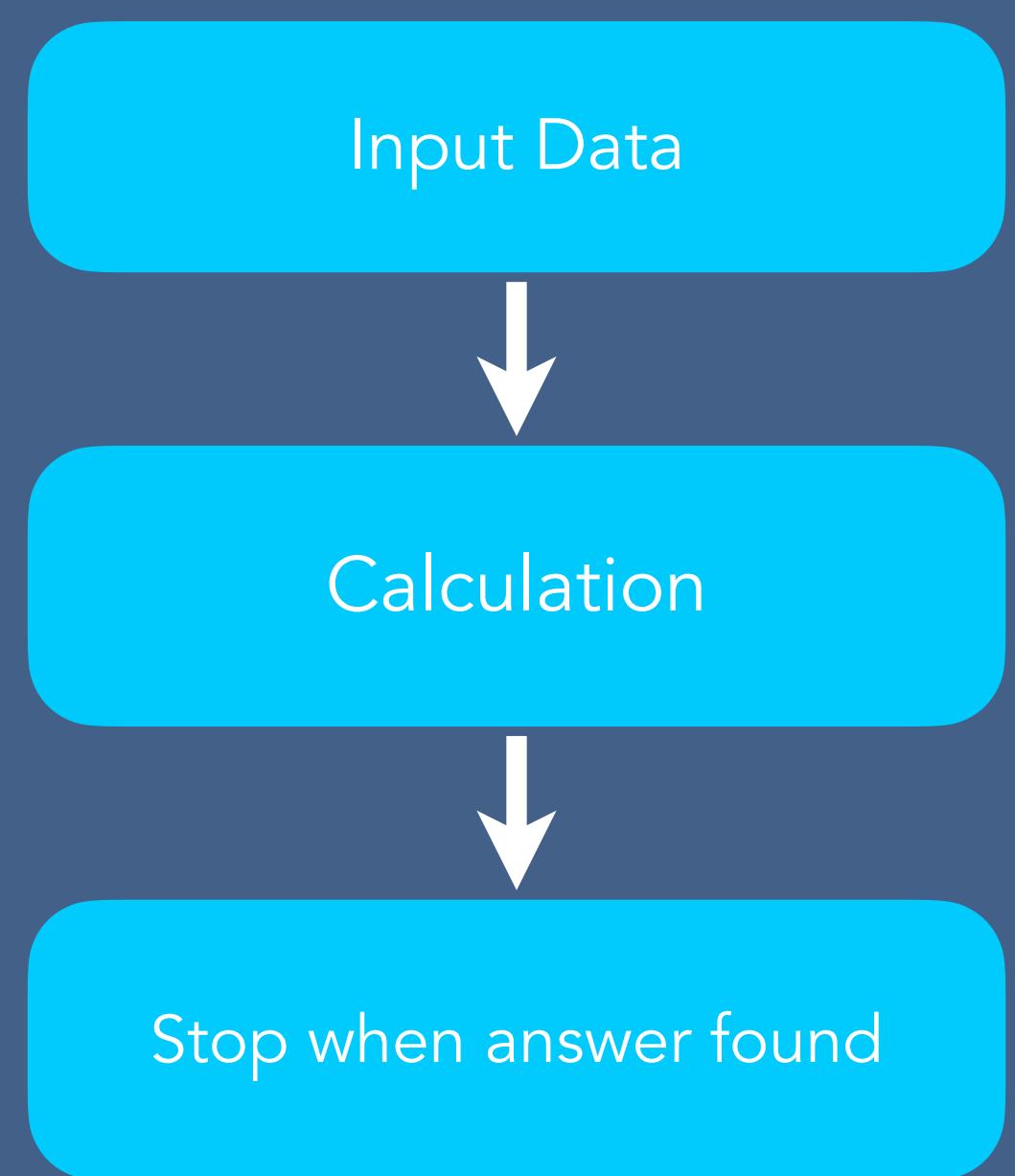
Step 2 : Pay money

Step 3: Take a coffee



# What is an Algorithm?

Algorithms in Computer Science : Set of rules for a computer program to accomplish a Task

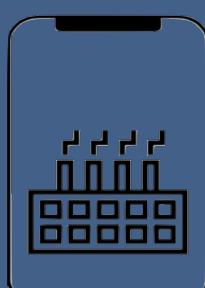
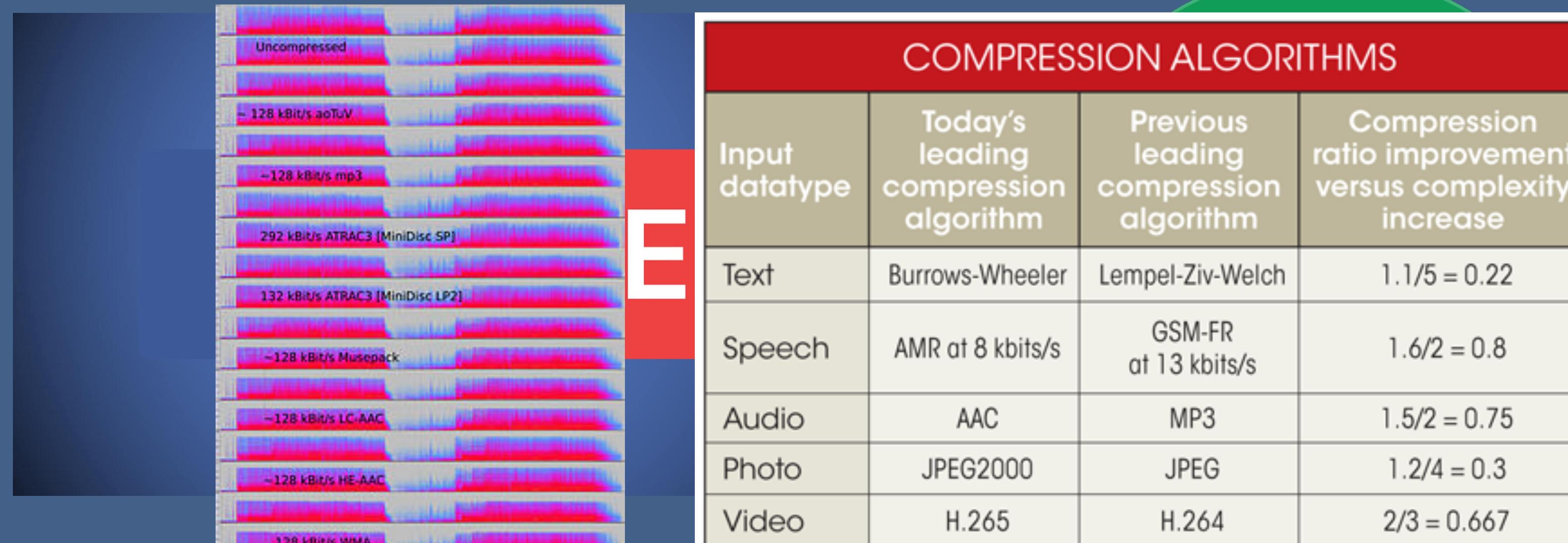


# What is an Algorithm?

Sample Algorithms that are used by BIG Companies

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

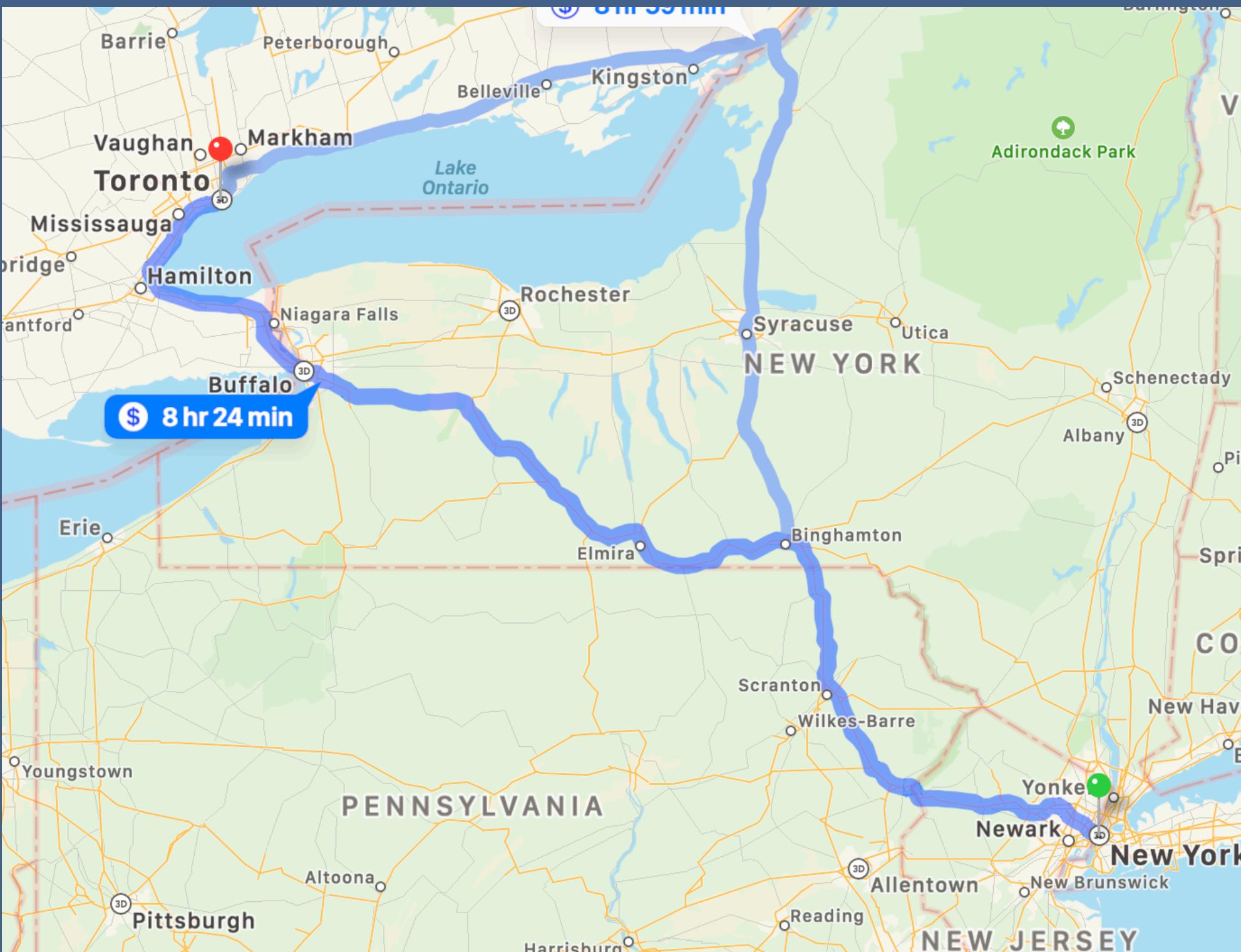
## Compression algorithms



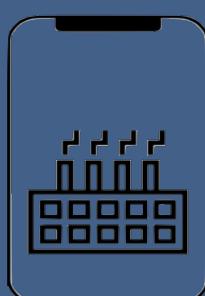
# What is an Algorithm?

Sample Algorithms that are used by BIG Companies

How to find the shortest path on the map?



Graph algorithms



# What is an Algorithm?

Sample Algorithms that are used by BIG Companies

How to arrange solar panels on the International Space Station?

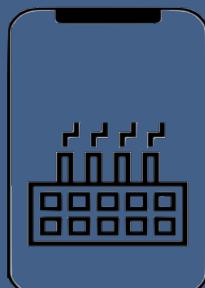


# What is an Algorithm?

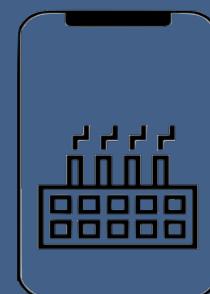
What makes a good algorithm?

- Correctness

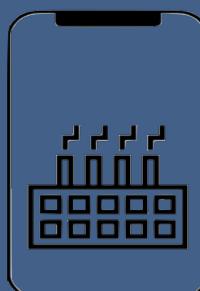
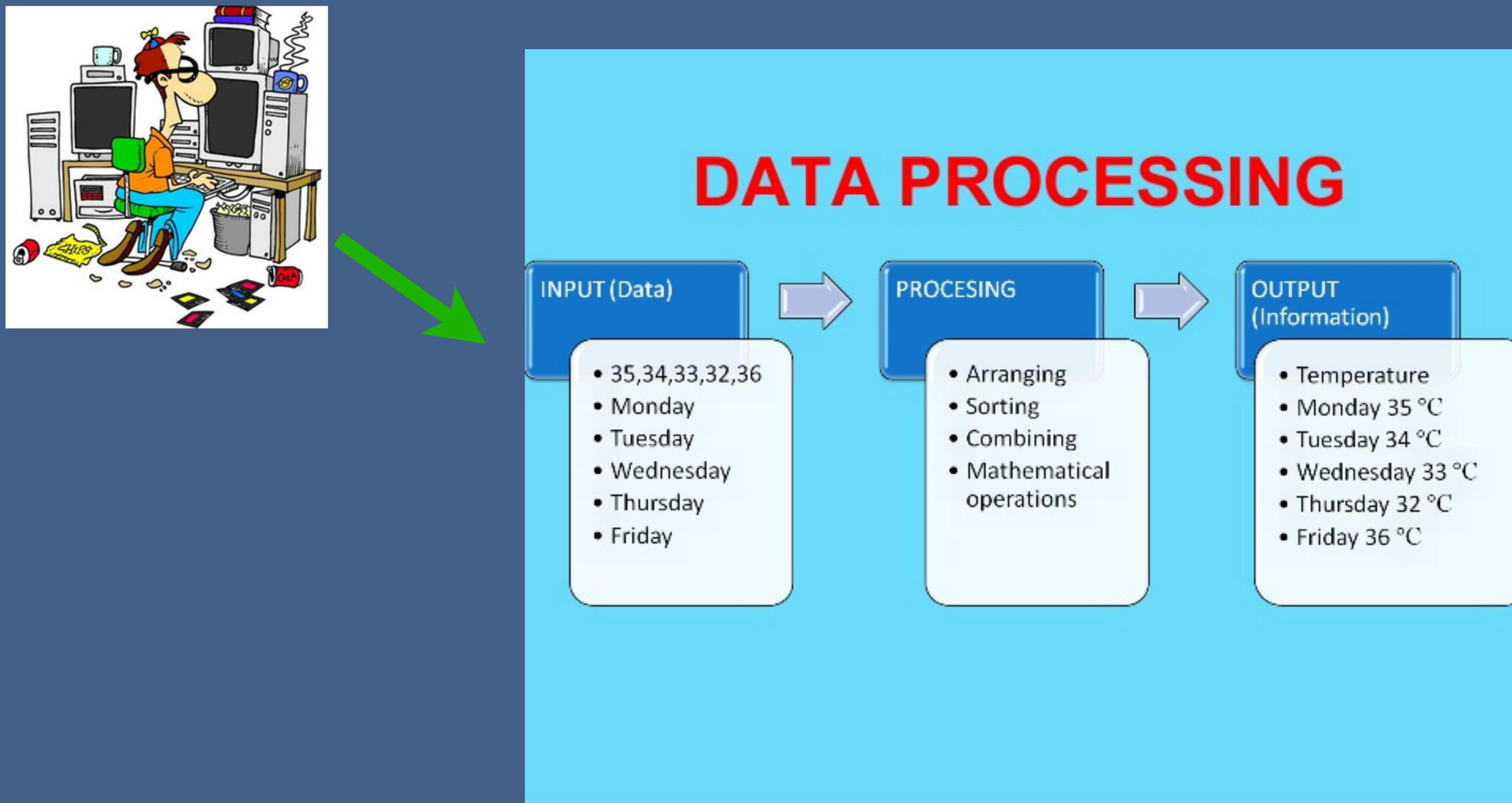
- Efficiency



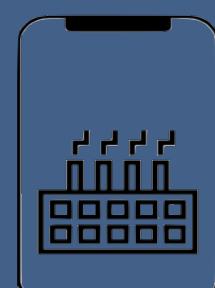
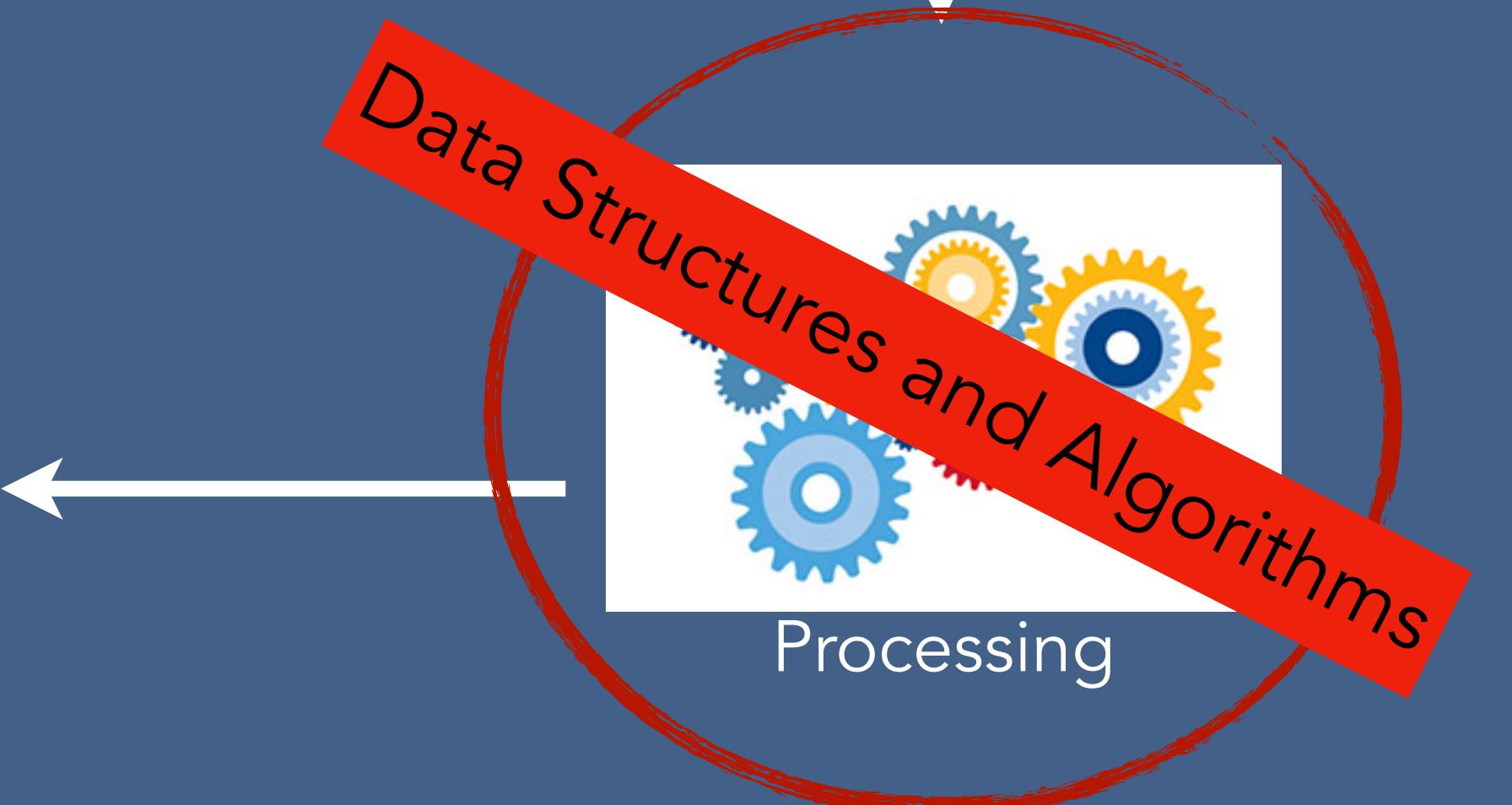
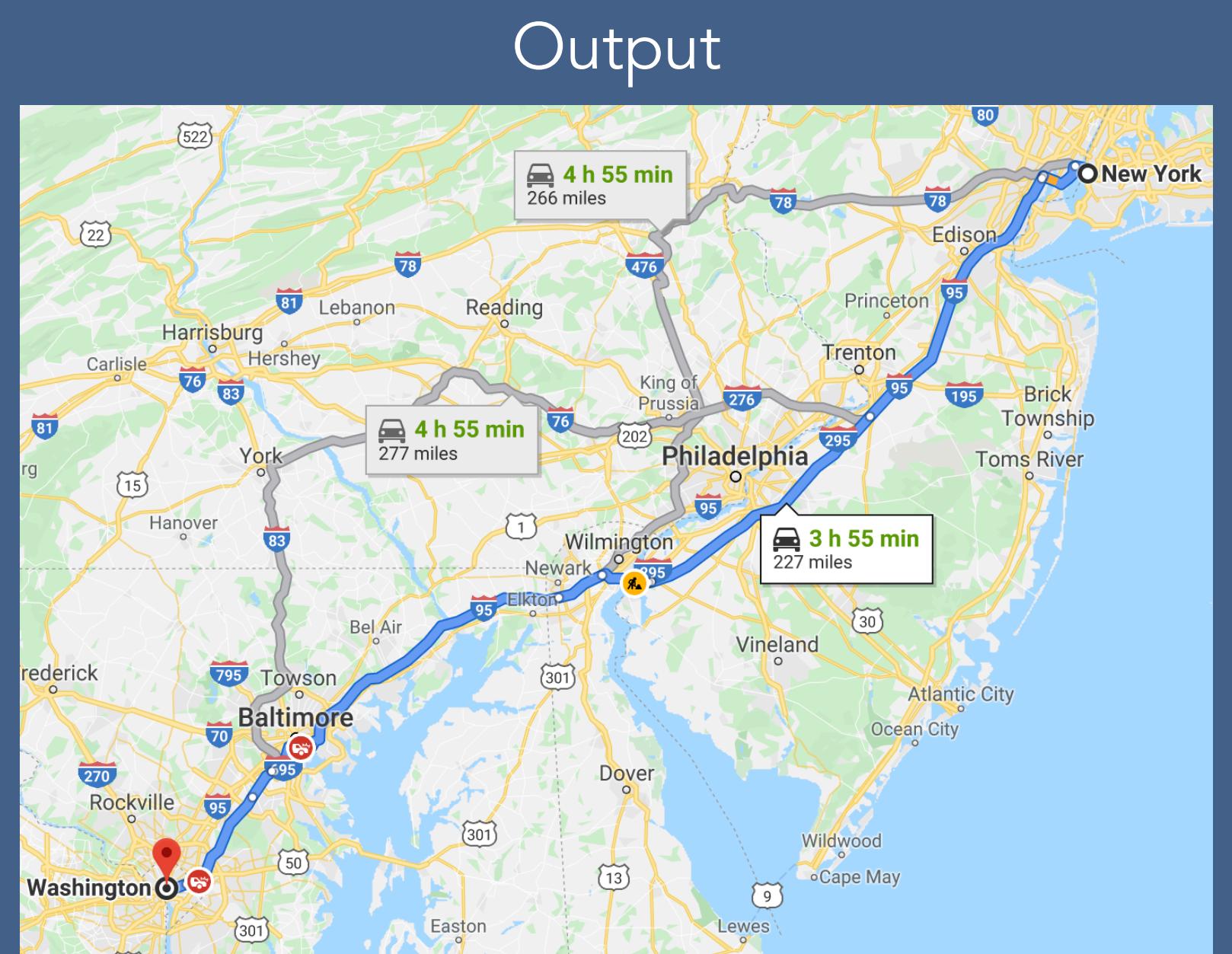
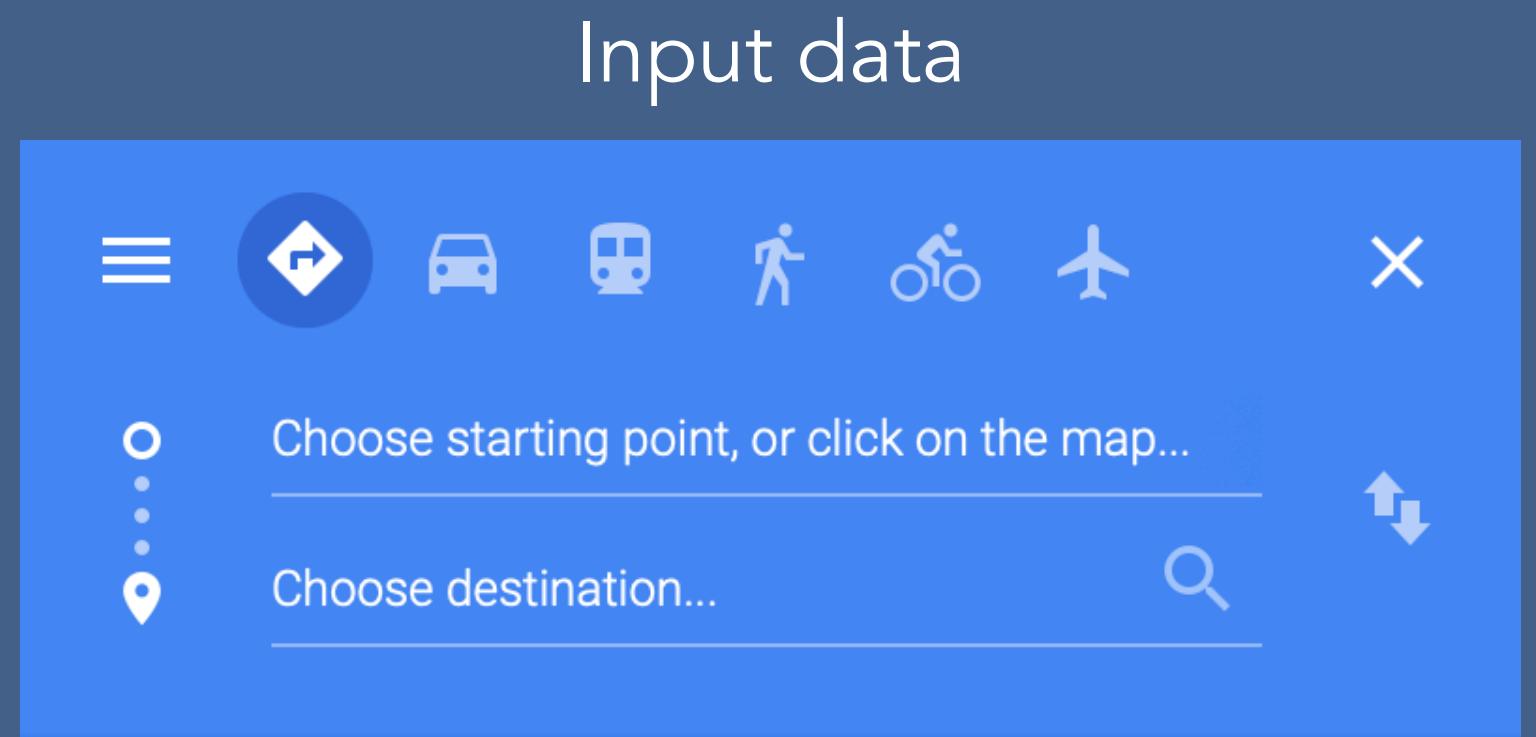
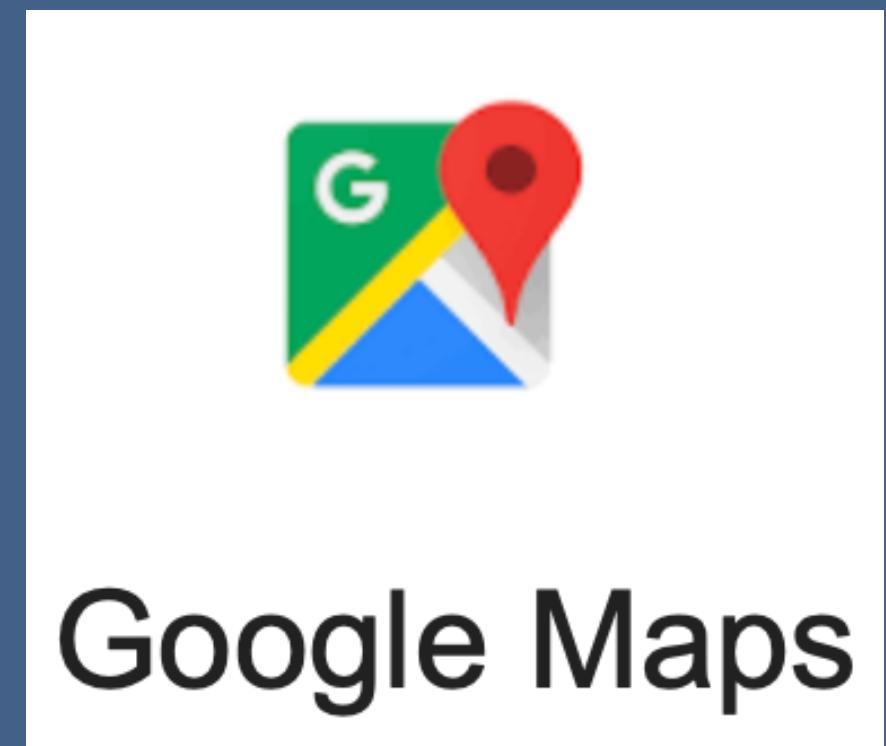
Why are Data Structures and Algorithms important?



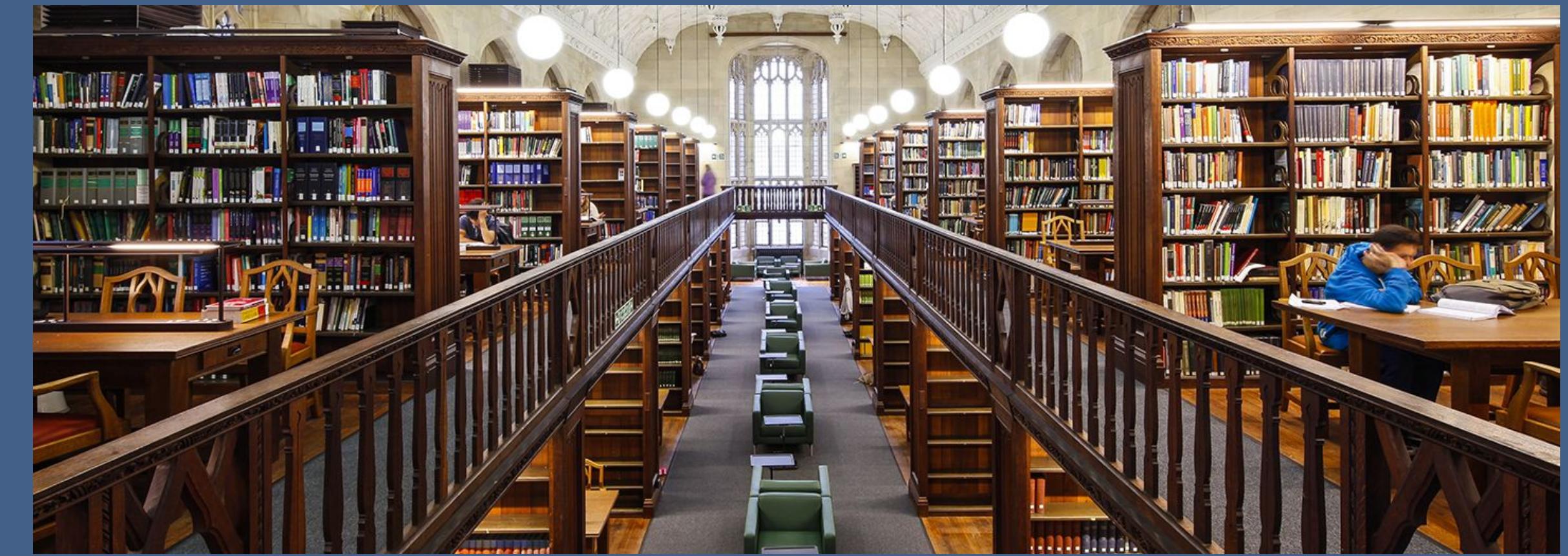
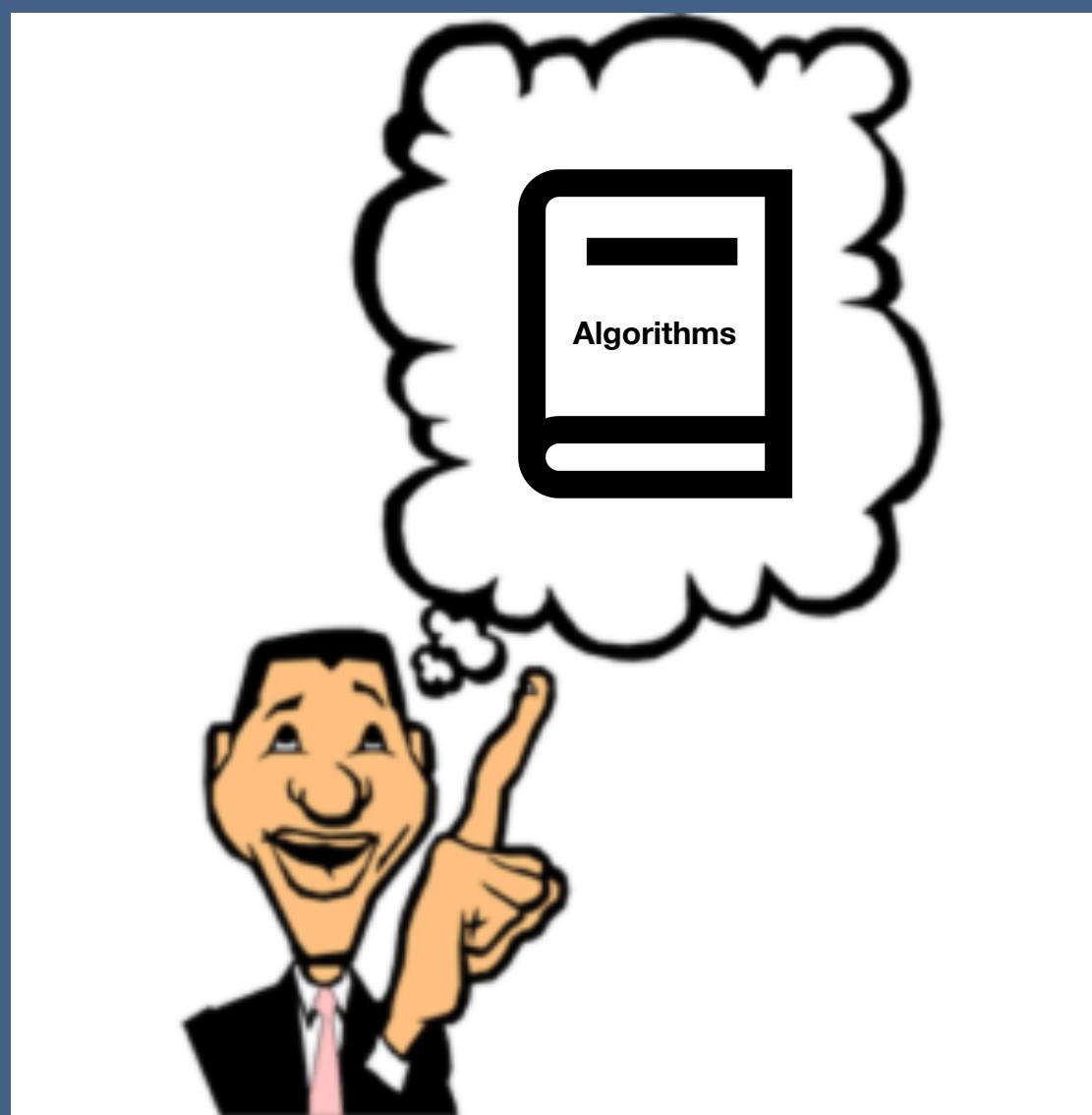
# Why are Data Structures and Algorithms important?



# Why are Data Structures and Algorithms important?

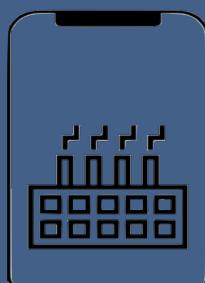
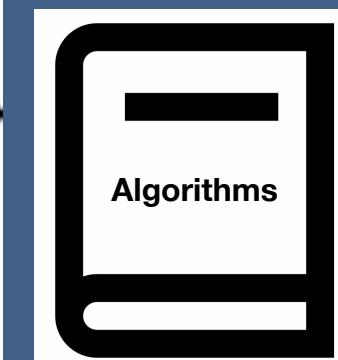


# Why are Data Structures and Algorithms important?

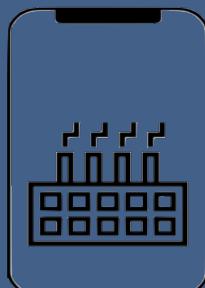


COMPUTER SCIENCE

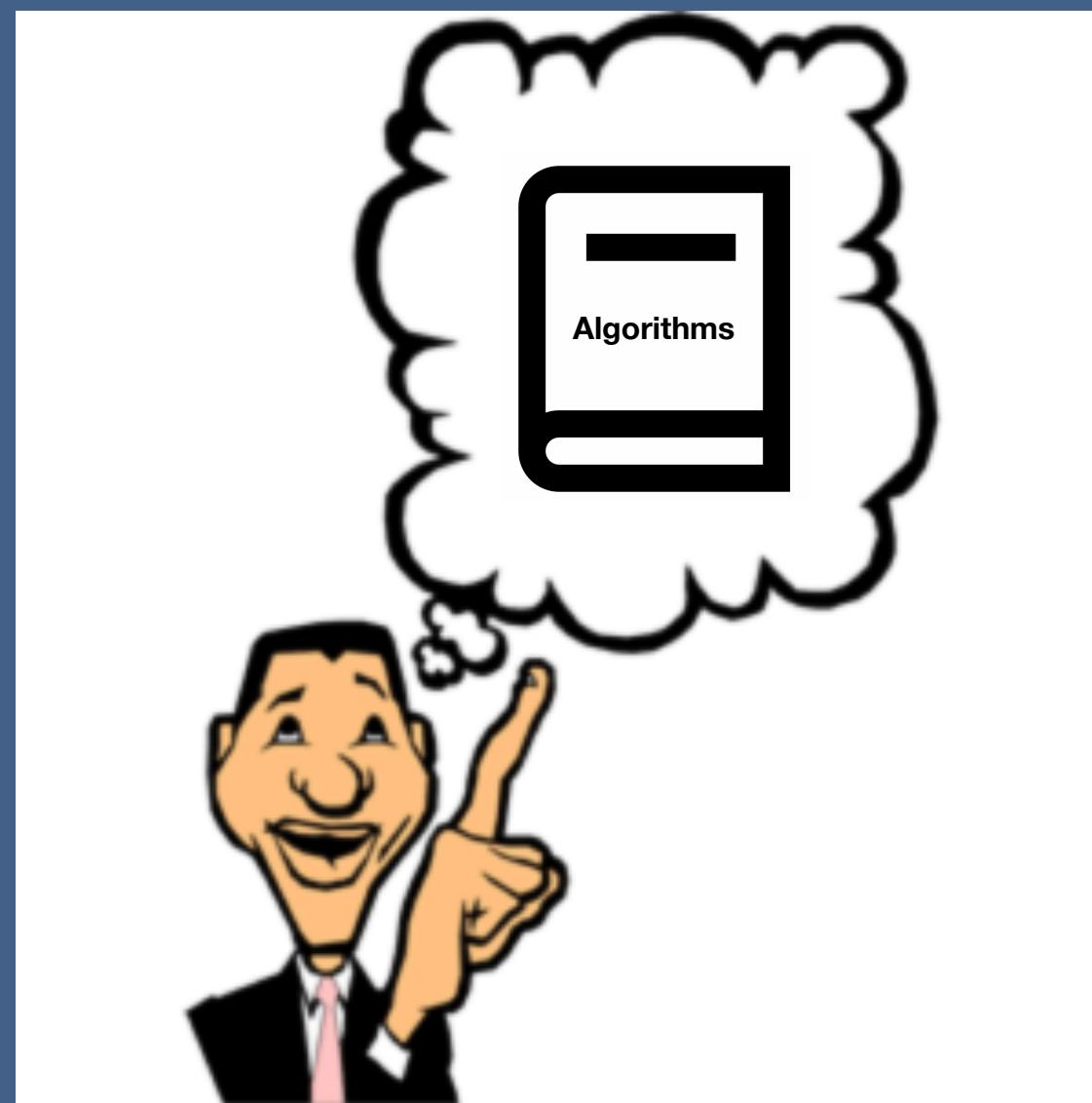
ALGORITHMS



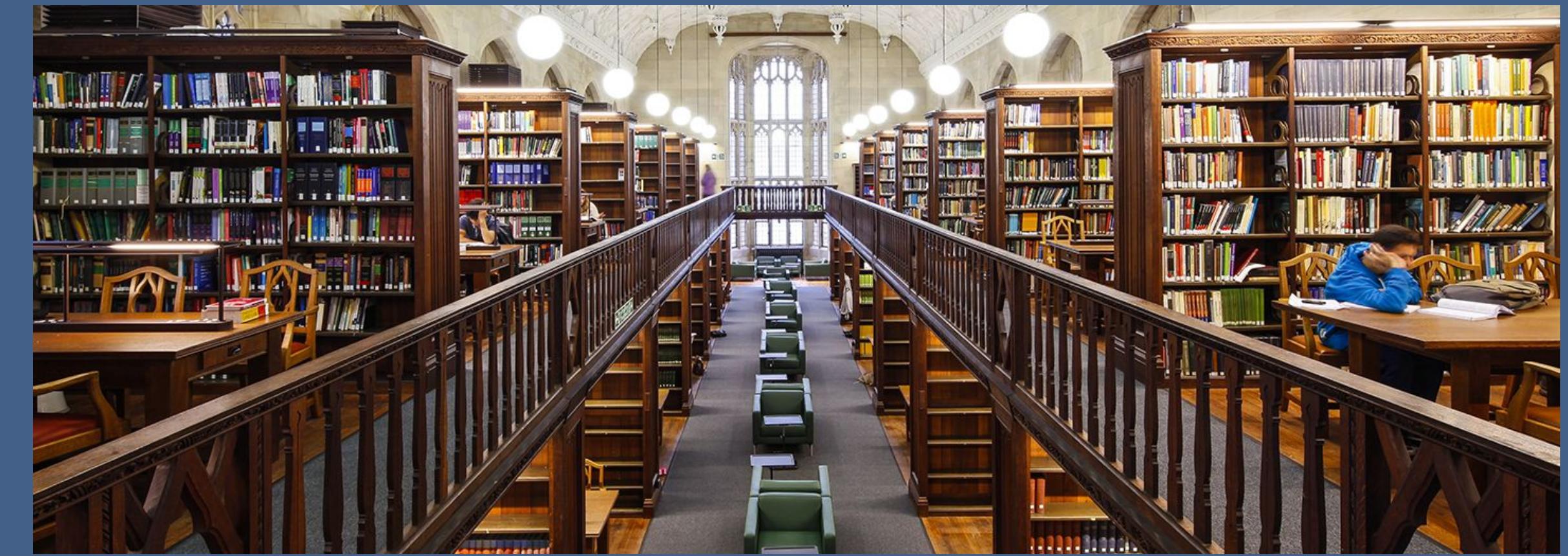
# Why are Data Structures and Algorithms important?



# Why are Data Structures and Algorithms important?



Data

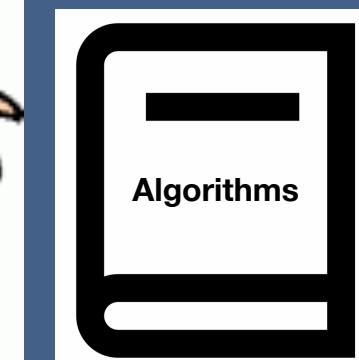


Data Structure

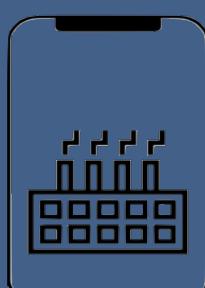


COMPUTER SCIENCE

ALGORITHMS



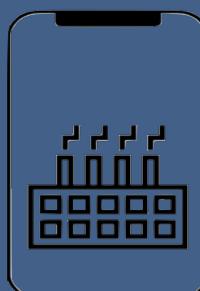
Algorithm



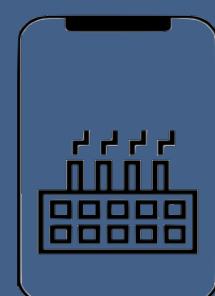
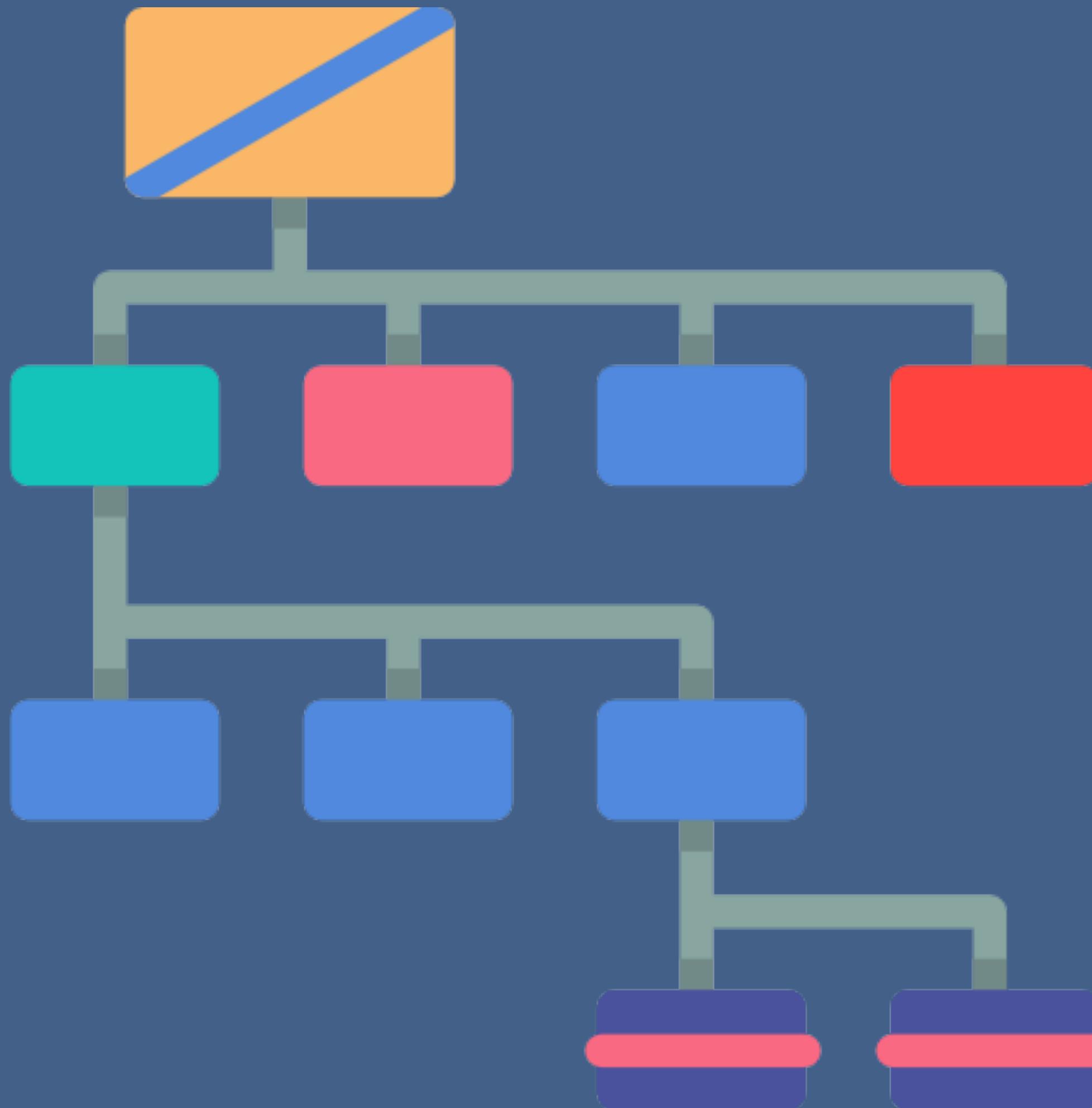
# Why are Data Structures and Algorithms in INTERVIEWS?



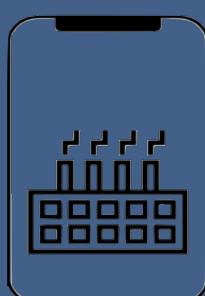
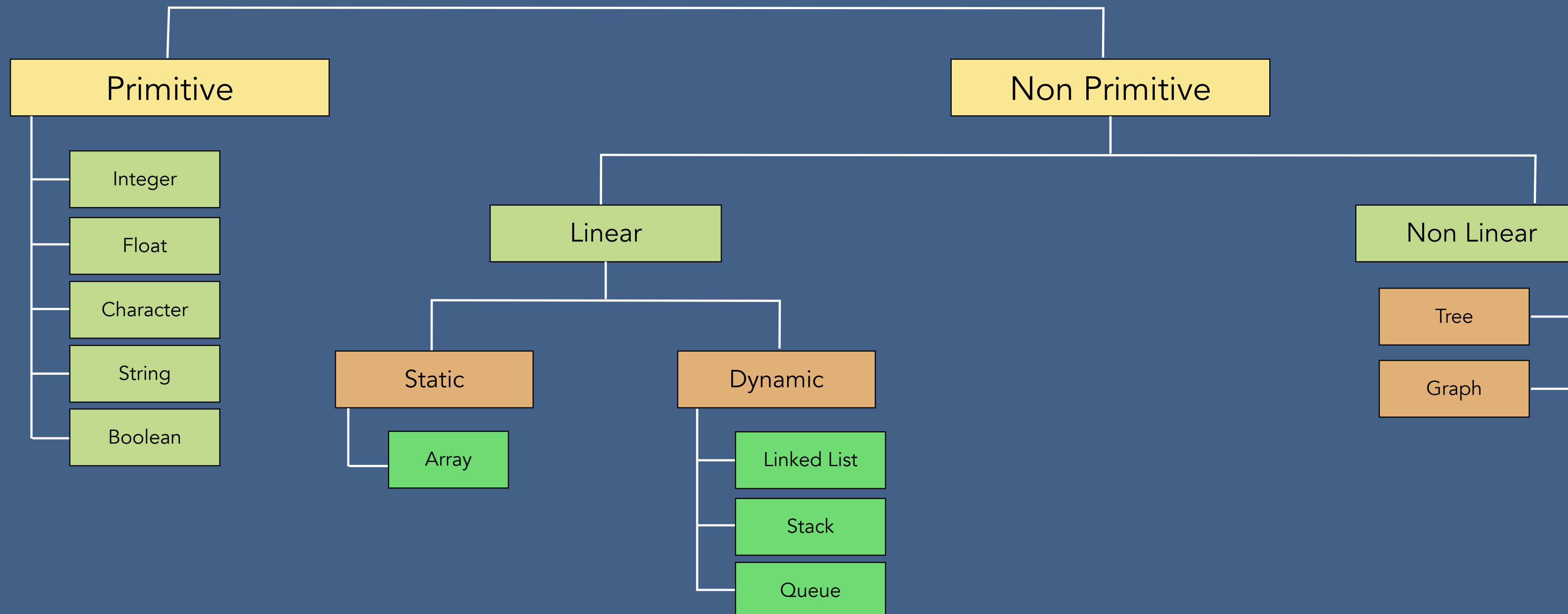
- Problem solving skills
- Fundamental concepts of programming in limited time



# Types of Data Structure

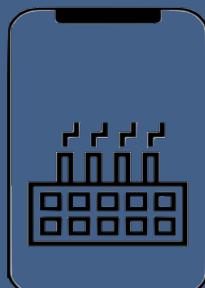


# Types of Data Structure

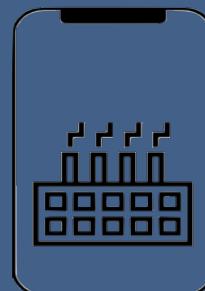
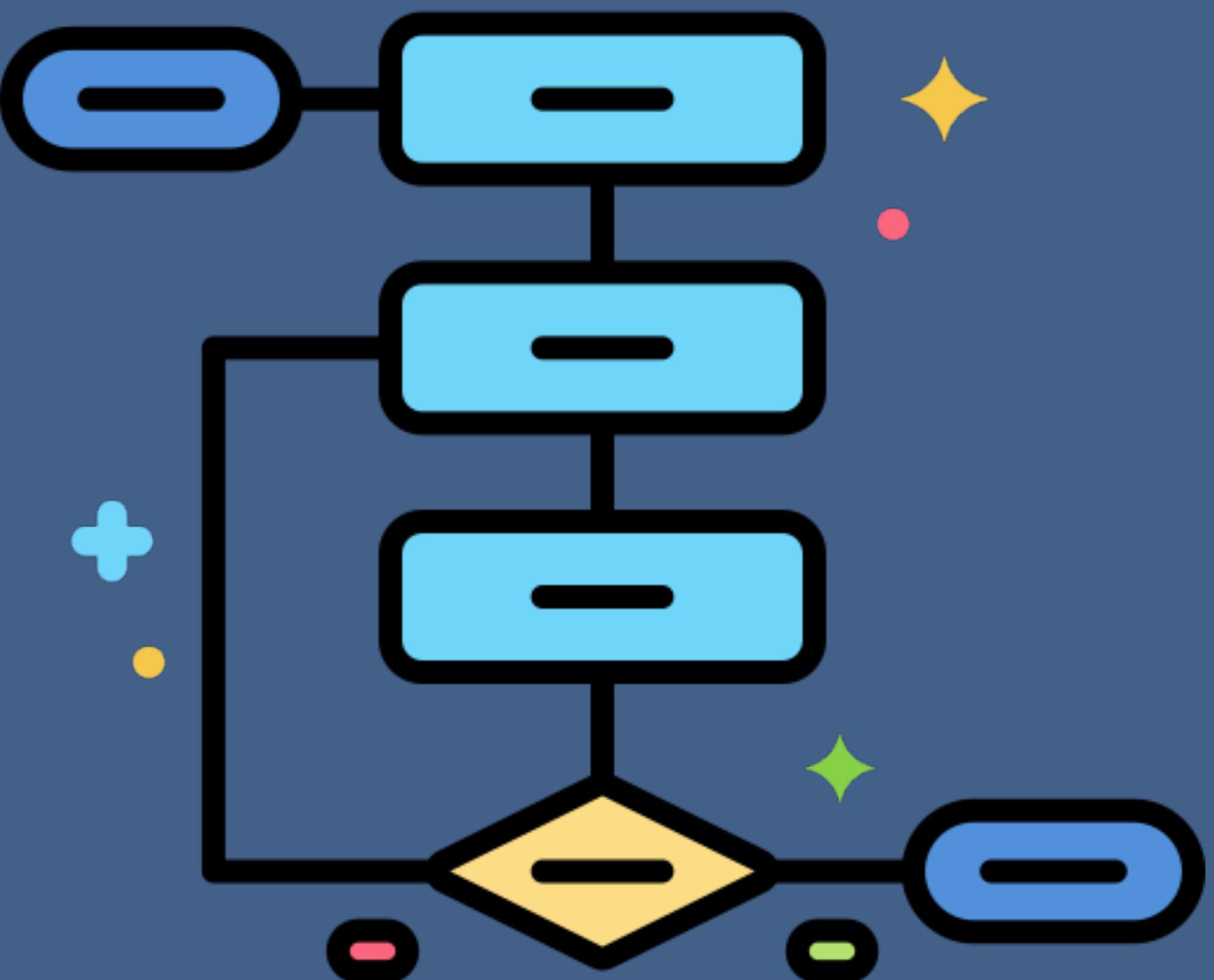


# Types of Data Structure

DATA STRUCTURE	Description	Example
INTEGER	Numbers with our 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

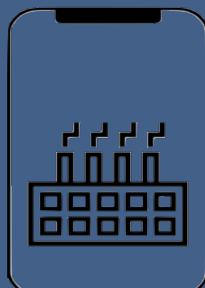


# Types of Algorithms



# Types of Algorithms

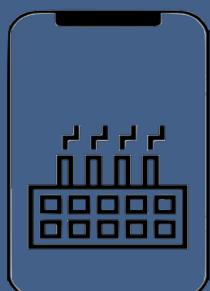
- Simple recursive algorithms
- Divide and conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Brute force algorithms
- Randomized algorithms



# 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

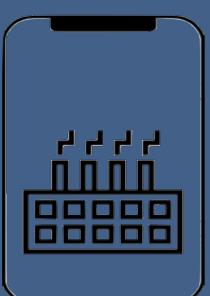
## Divide and conquer algorithms

- 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

## Dynamic programming algorithms

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



# Types of Algorithms

## Greedy algorithms

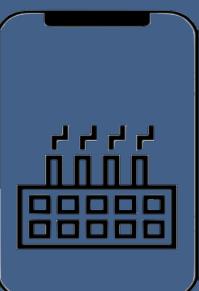
- We take the best we can without worrying about future consequences.
- We hope that by choosing a local optimum solution at each step, we will end up at a global optimum solution

## Brute force algorithms

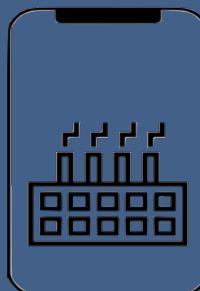
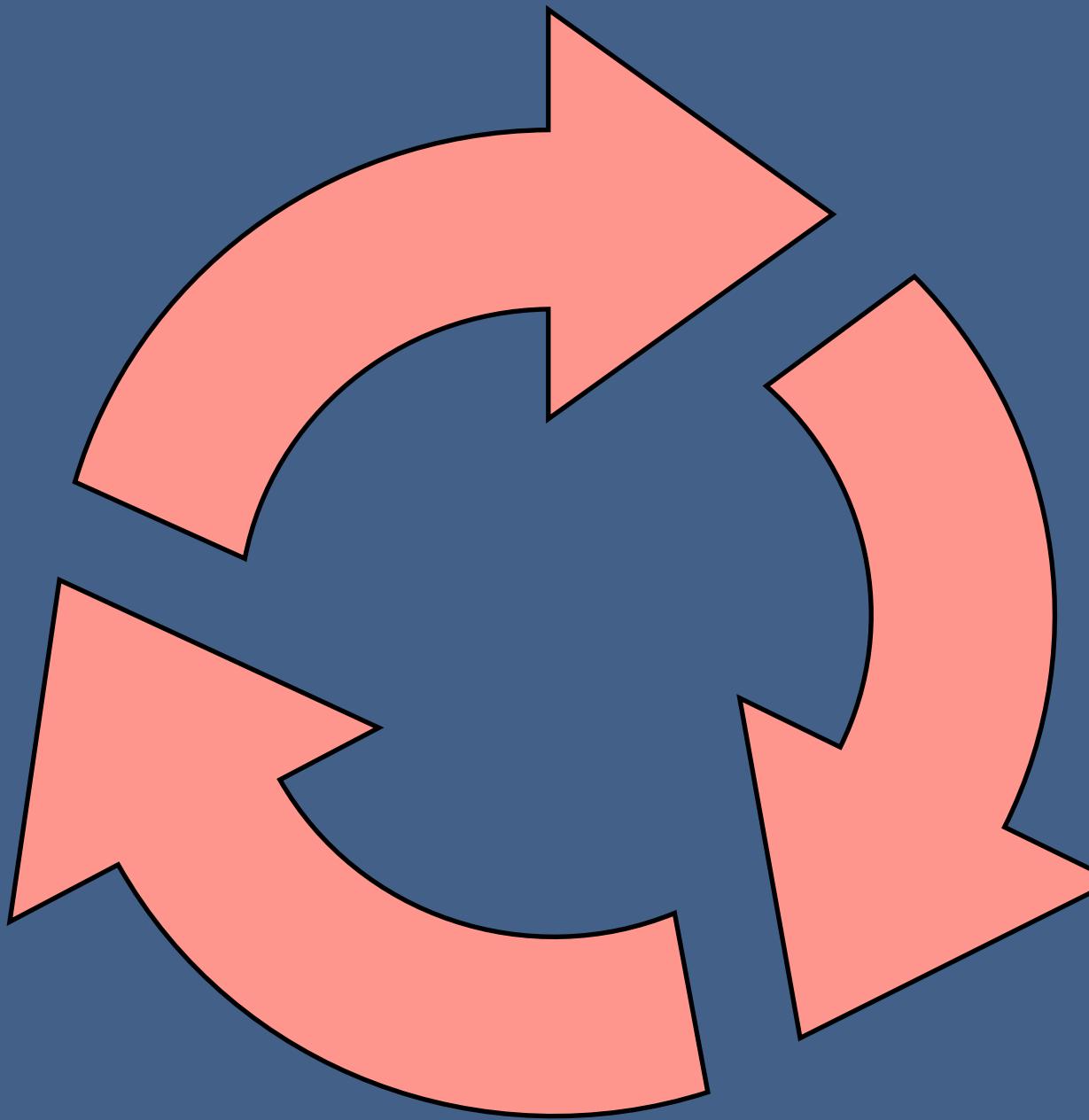
- It simply tries all possibilities until a satisfactory solution is found

## Randomized algorithms

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



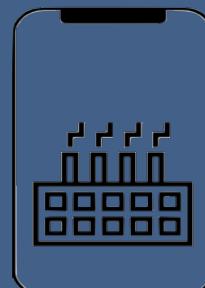
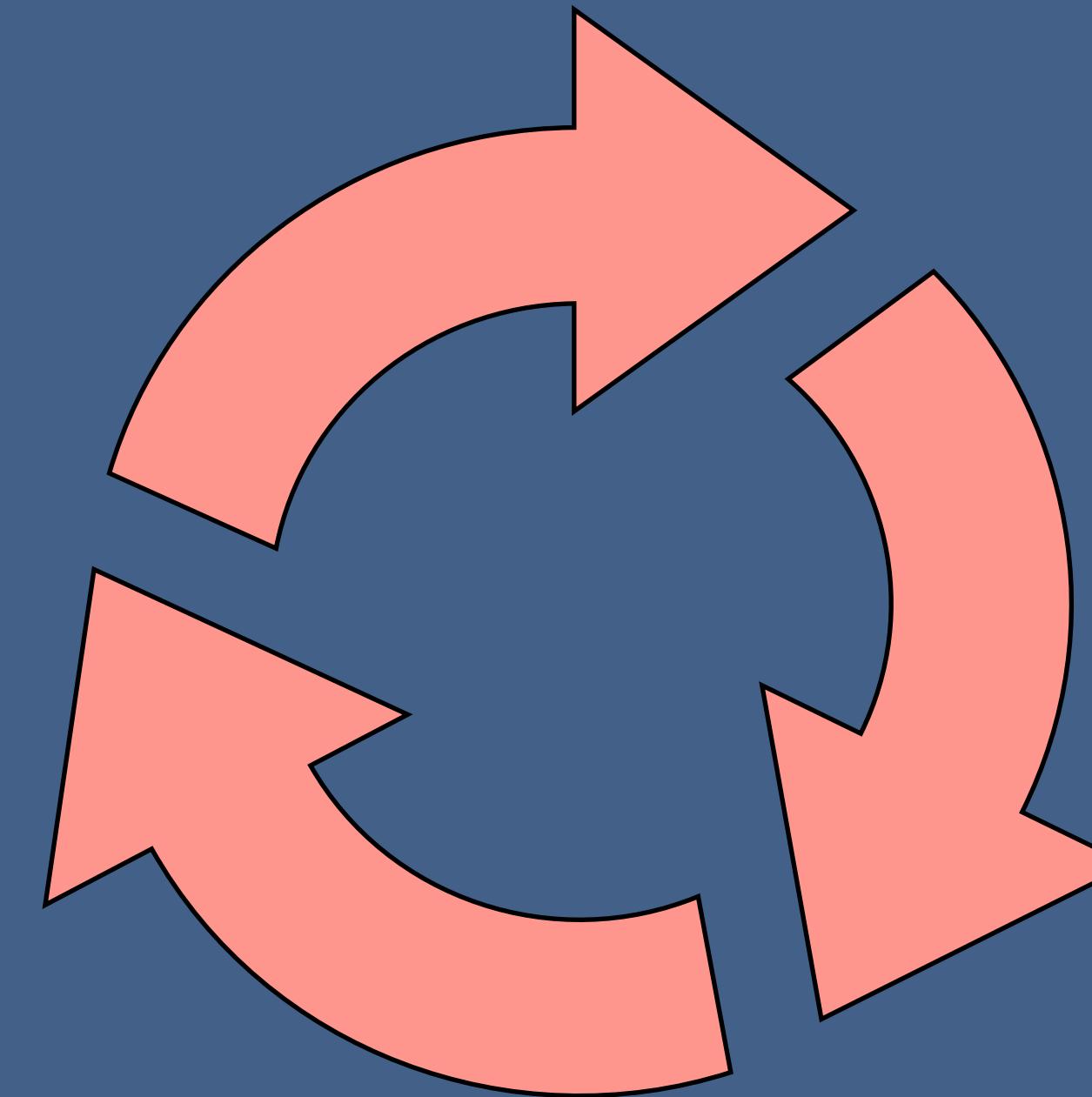
# Recursion



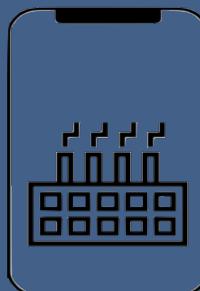
# Recursion

## Objectives:

- What is Recursion?
- Why do we need Recursion?
- The Logic behind the Recursion
- Recursive vs Iterative Solutions
- How to Write Recursion in 3 Steps?
- Find Fibonacci Series using Recursion



# What is Recursion?

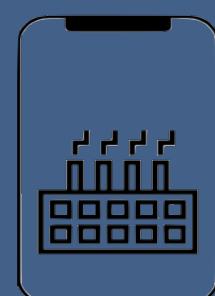


# What is Recursion?

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



...

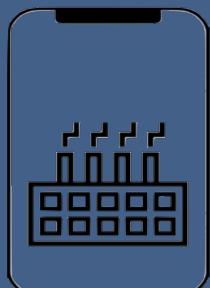


# What is Recursion?

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.

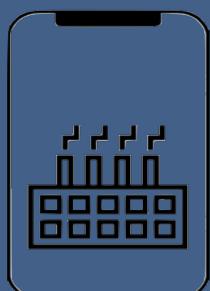


# What is Recursion?

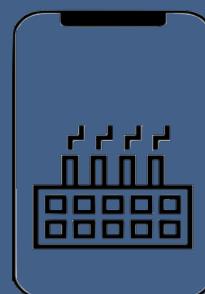
Recursion : a way of solving a problem by having a function calling itself



```
static void openRussianDoll(int doll) {  
    if (doll == 1) {  
        System.out.println("All dolls are opened");  
    } else {  
        openRussianDoll(doll-1);  
    }  
}
```

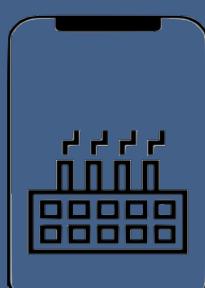


# Why we need Recursion?

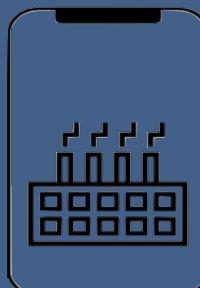


# Why we need Recursion?

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)



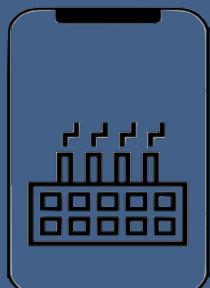
# The Logic Behind Recursion



# The Logic Behind Recursion

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

```
static string recursionMethod(String[] parameters) {  
    if (exit from condition satisfied) {  
        return some value;  
    } else {  
        recursionMethod(modified parameters);  
    }  
}
```



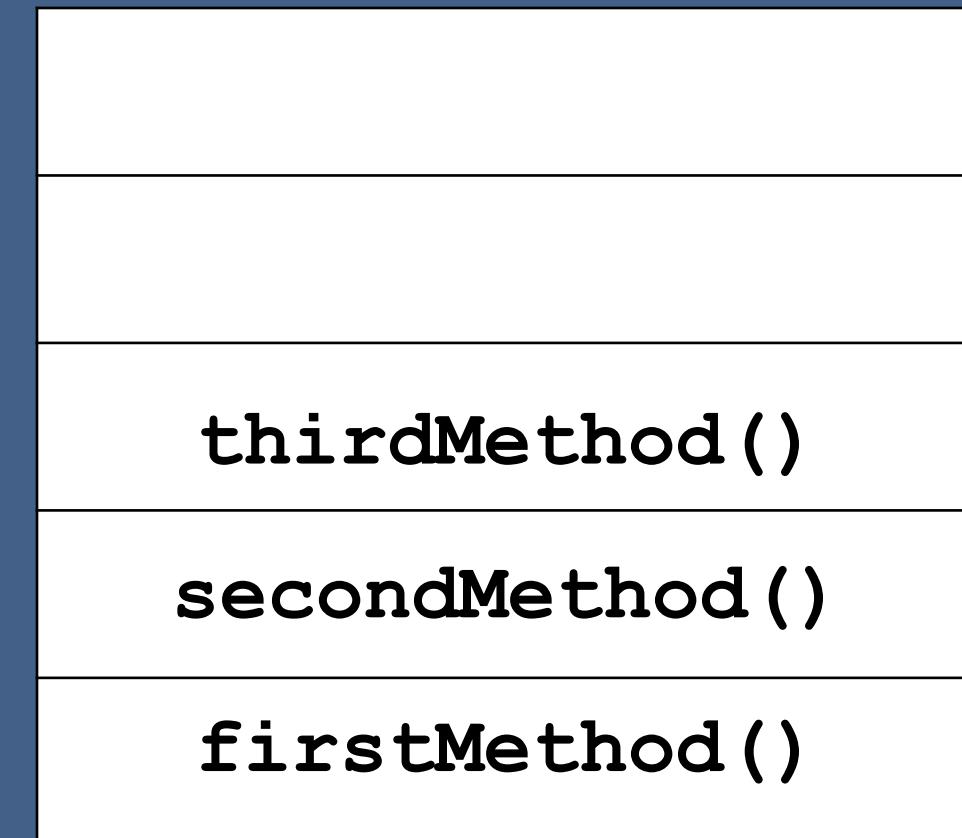
# The Logic Behind Recursion

```
static void firstMethod() {  
    secondMethod();  
    System.out.println("I am the first Method");  
}
```

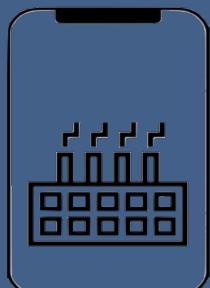
```
static void secondMethod() {  
    thirdMethod();  
    System.out.println("I am the second Method");  
}
```

```
static void thirdMethod() {  
    fourthMethod();  
    System.out.println("I am the third Method");  
}
```

```
static void fourthMethod() {  
    System.out.println("I am the fourth Method");  
}
```

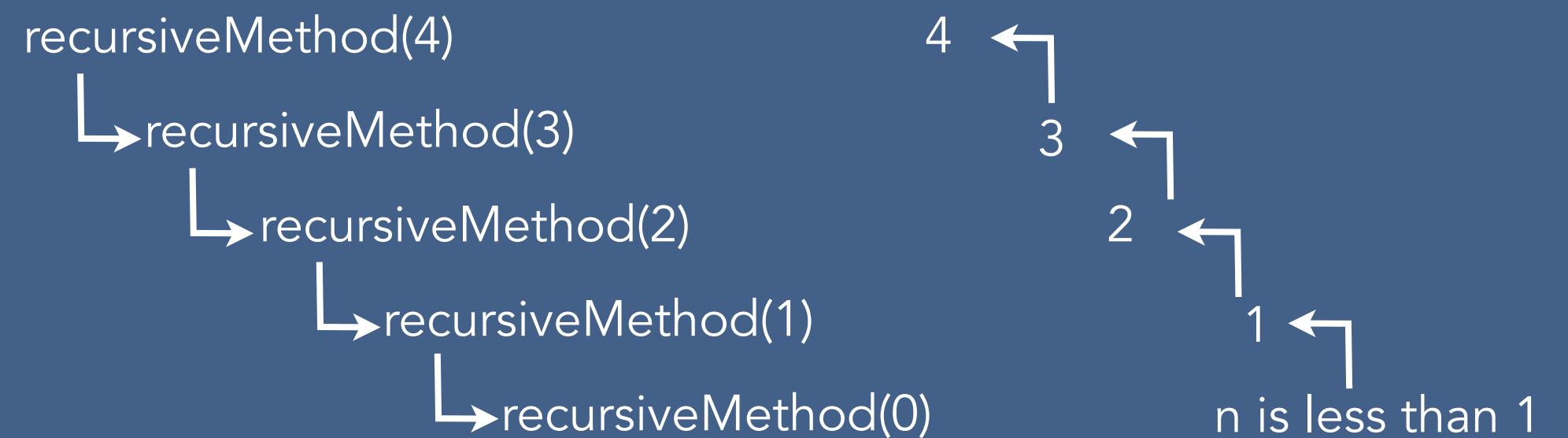


STACK Memory



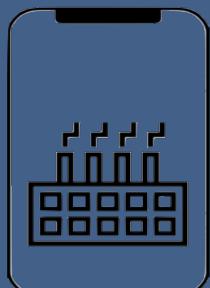
# The Logic Behind Recursion

```
static void recursiveMethod(int n) {  
    if (n<1) {  
        System.out.println("n is less than 1");  
    } else {  
        recursiveMethod(n-1);  
        System.out.println(n);  
    }  
}
```

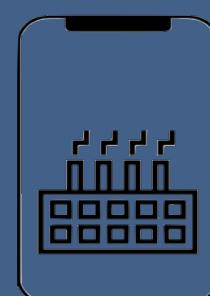
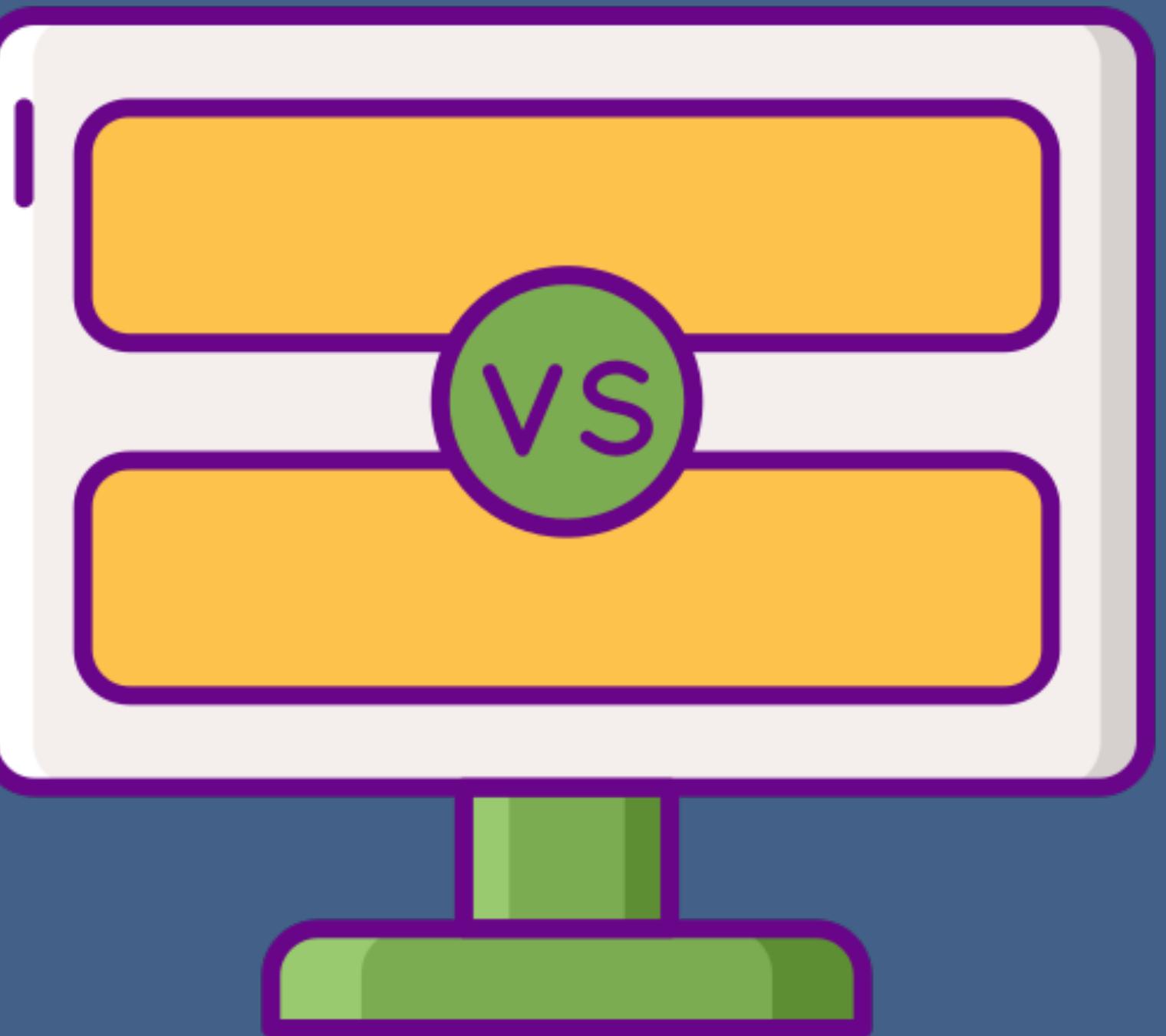


<b>recursiveMethod(1)</b>
<b>recursiveMethod(2)</b>
<b>recursiveMethod(3)</b>
<b>recursiveMethod(4)</b>

**STACK Memory**



# Recursive vs Iterative Solutions



# Recursive vs Iterative Solutions

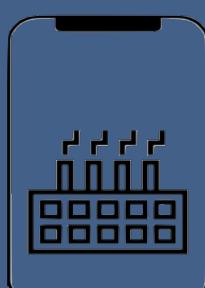
## Recursive

```
static int powerOfTwo(int n) {  
    if (n==0) {  
        return 1;  
    } else {  
        var power = 2*powerOfTwo(n-1);  
        return power;  
    }  
}
```

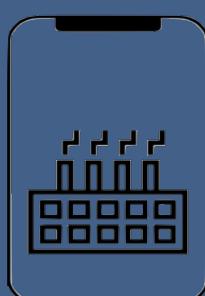
## Iterative

```
static int powerOfTwoIT(int n) {  
    var i = 0;  
    var 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.



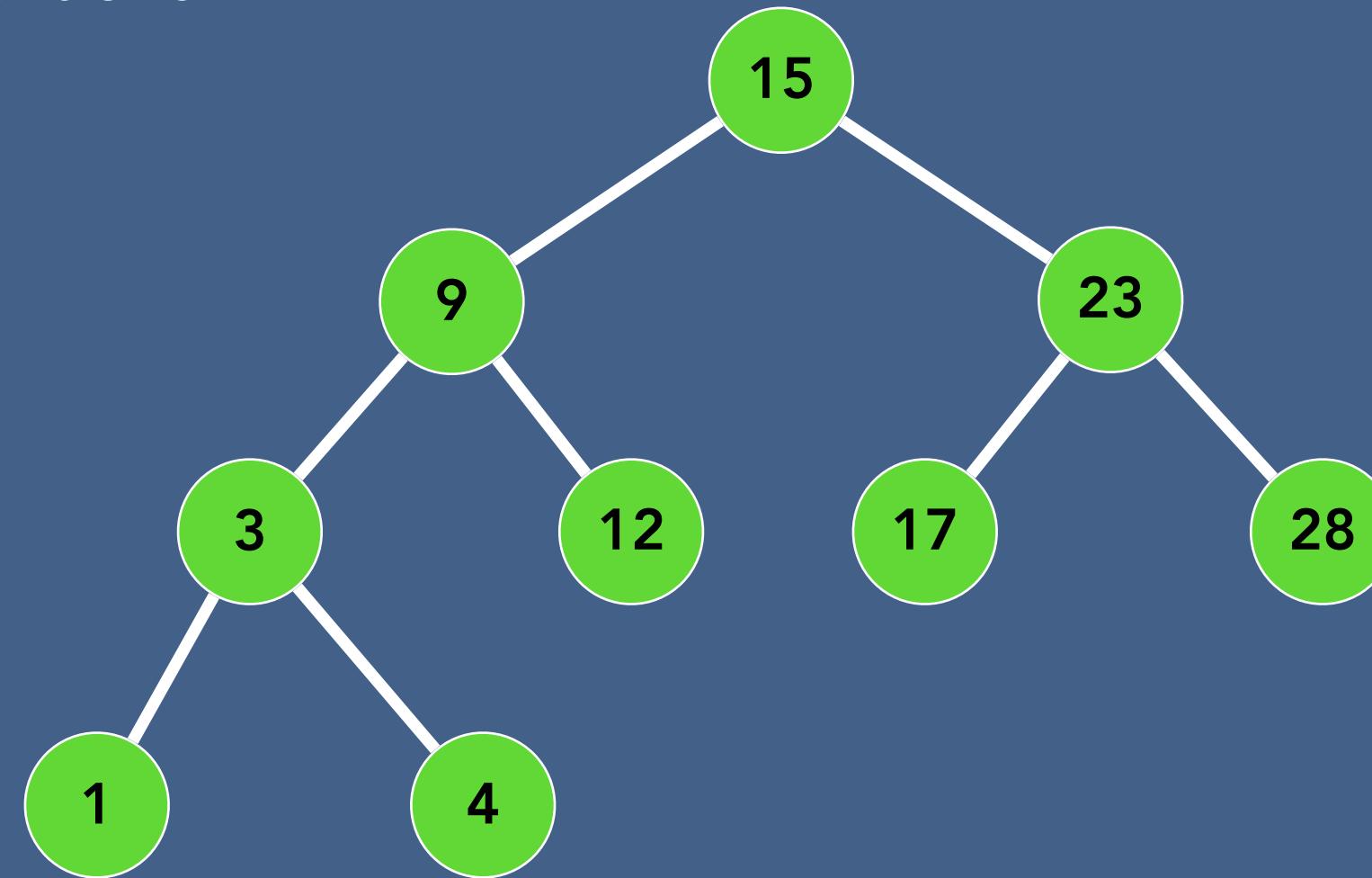
# When to Use/Avoid Recursion?



# When to Use/Avoid Recursion?

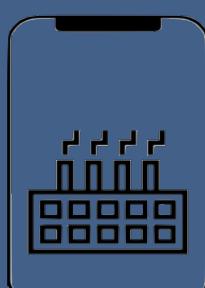
## When to use it?

- 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

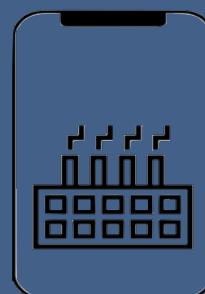
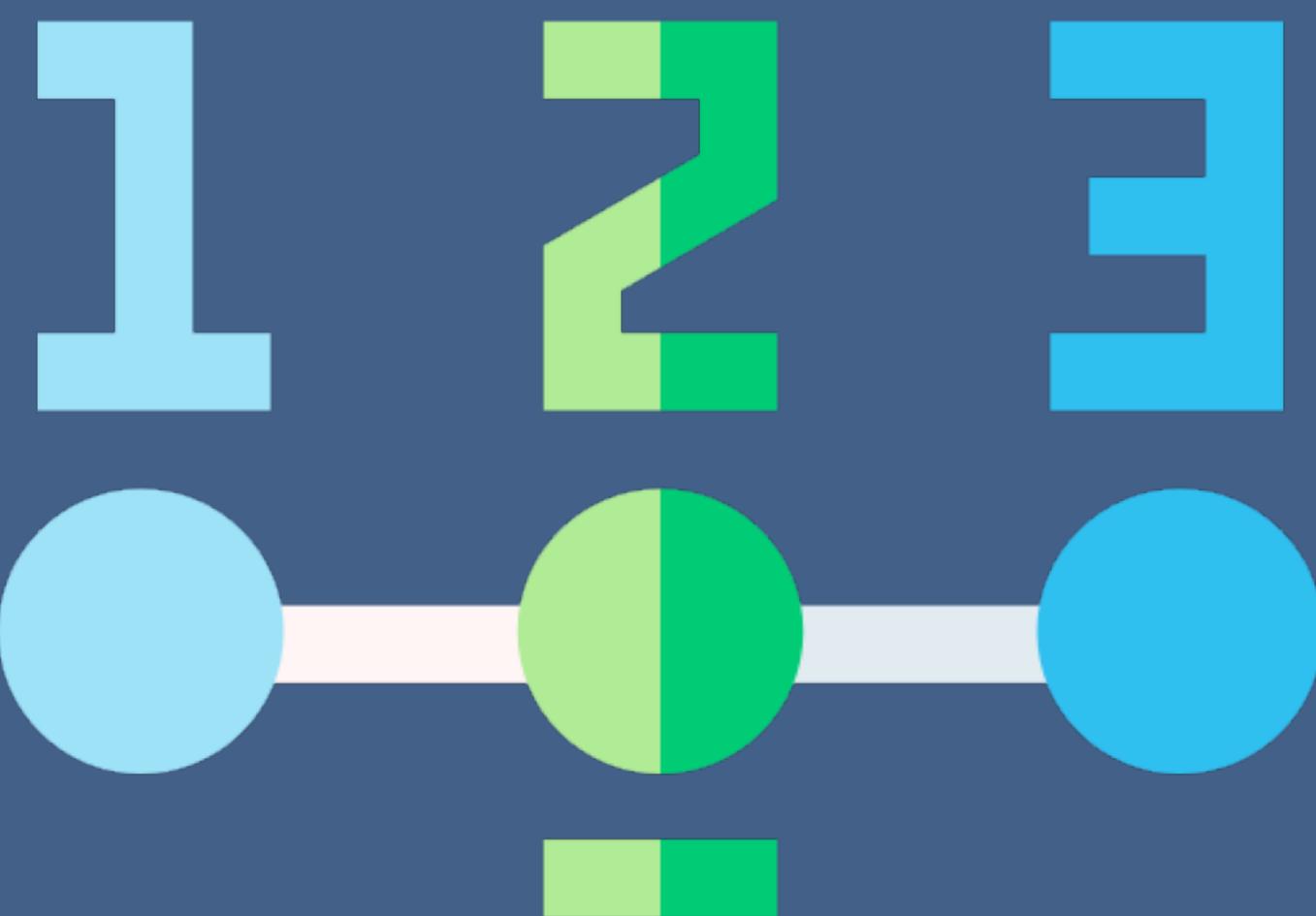


## When avoid it?

- 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



# Write Recursion in 3 Steps



# 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.
- $0!=1$ .



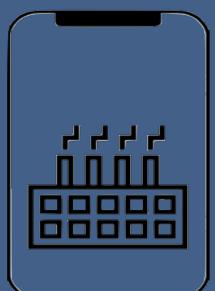
### Example 1

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

### Example 2

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

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



# Write Recursion in 3 Steps

## Step 1 : Recursive case - the flow

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

↓

$$(n-1)!$$

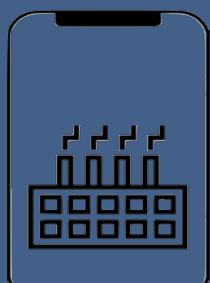
$$(n-1)! = (n-1) * (n-1-1) * (n-1-2) * \dots * 2 * 1 = (n-1) * (n-2) * (n-3) * \dots * 2 * 1$$

## Step 2 : Base case - the stopping criterion

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

## Step 3 : Unintentional case - the constraint

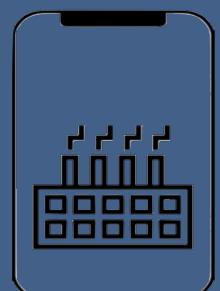
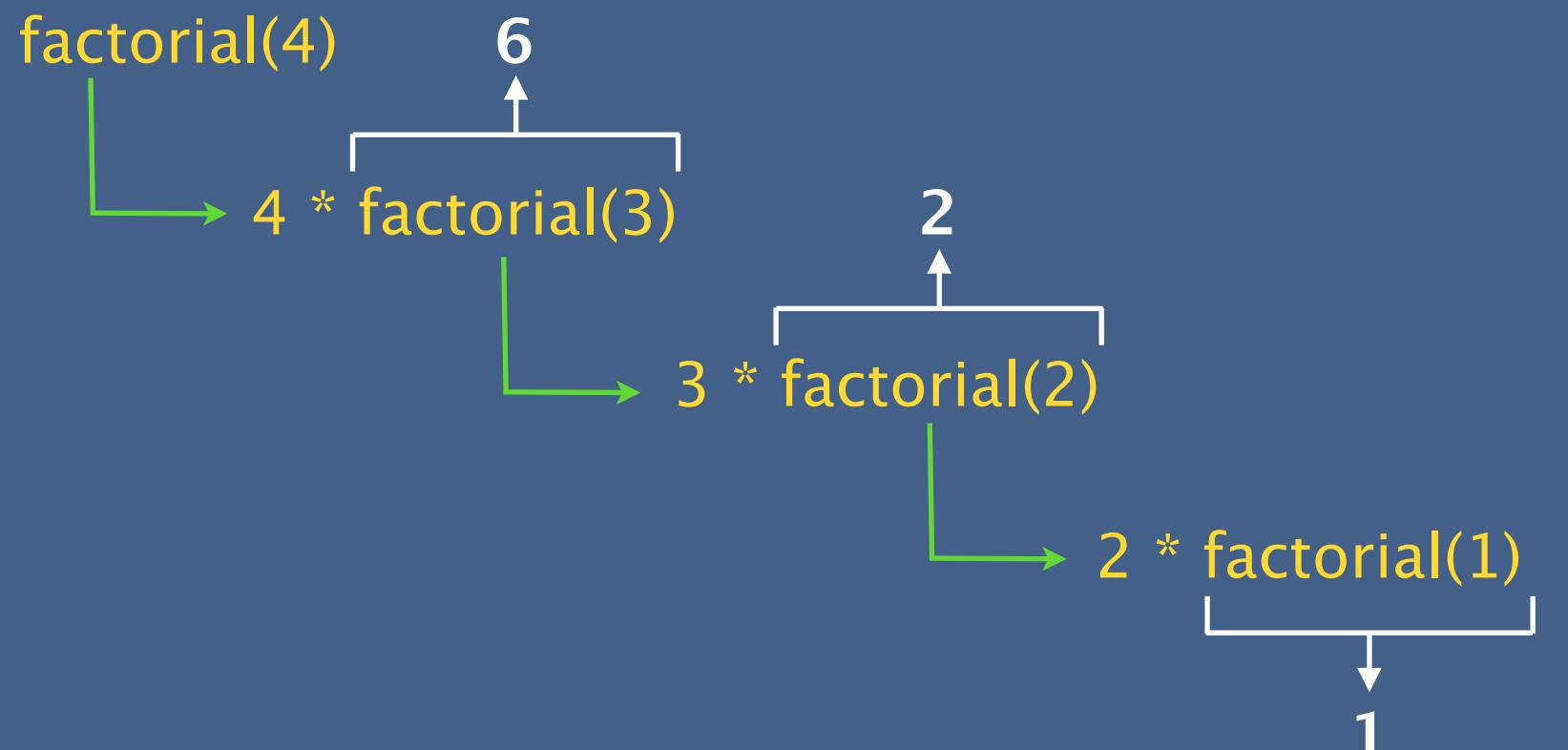
- $\text{factorial}(-1)$  ??
- $\text{factorial}(-2)$  ??



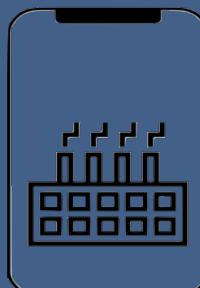
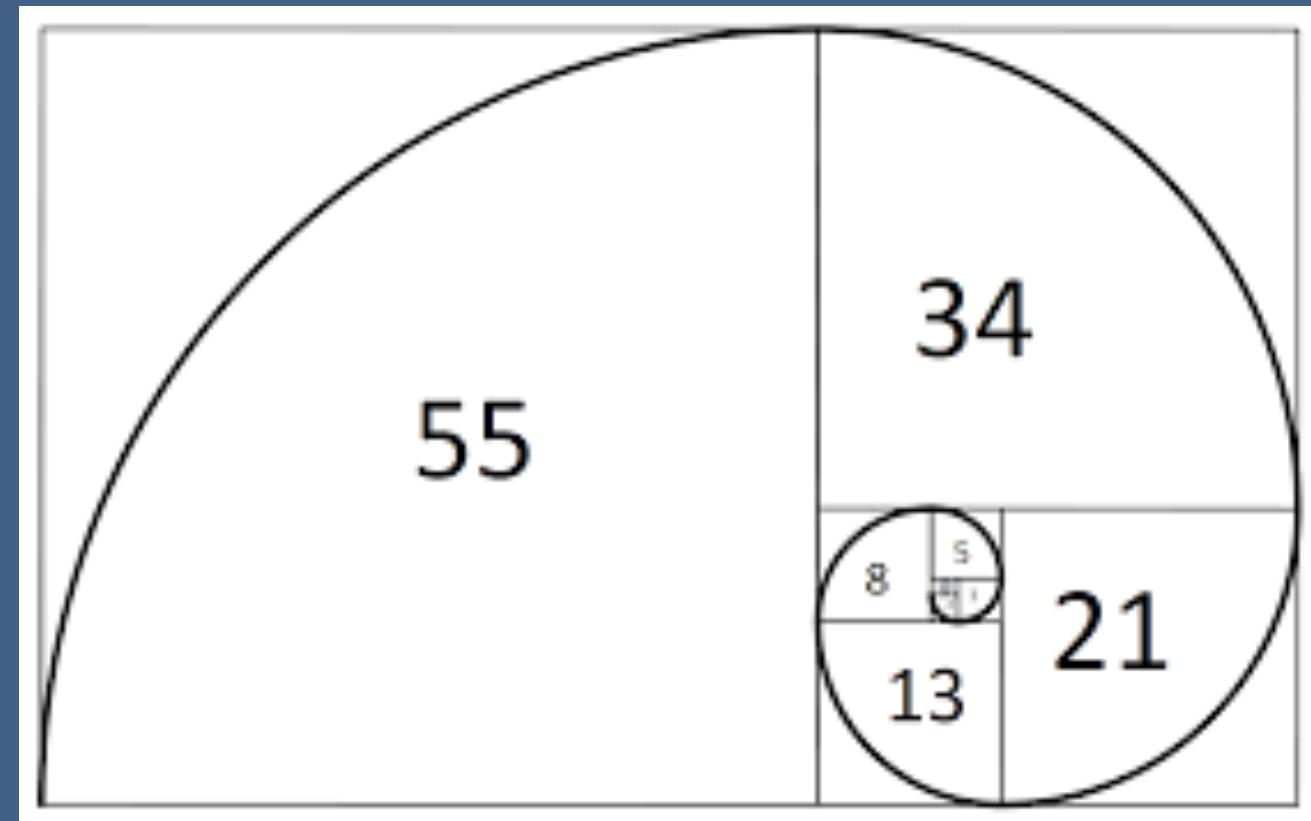
# Write Recursion in 3 Steps

```
public int factorial(int n) {  
    if (n<0) {  
        return -1;  
    }  
    if (n == 0 || n == 1) {  
        return 1;  
    }  
    return n * factorial(n-1);  
}
```

factorial(4) = 24



# Fibonacci using Recursion



# Write Fibonacci Recursion in 3 Steps

## Fibonacci numbers

- 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

$$5 = 3 + 2$$

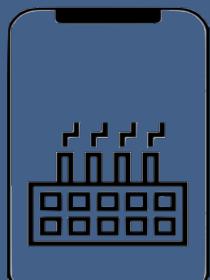
$$f(n) = f(n-1) + f(n-2)$$

### Step 2 : Base case - the stopping criterion

- 0 and 1

### Step 3 : Unintentional case - the constraint

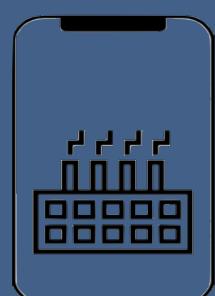
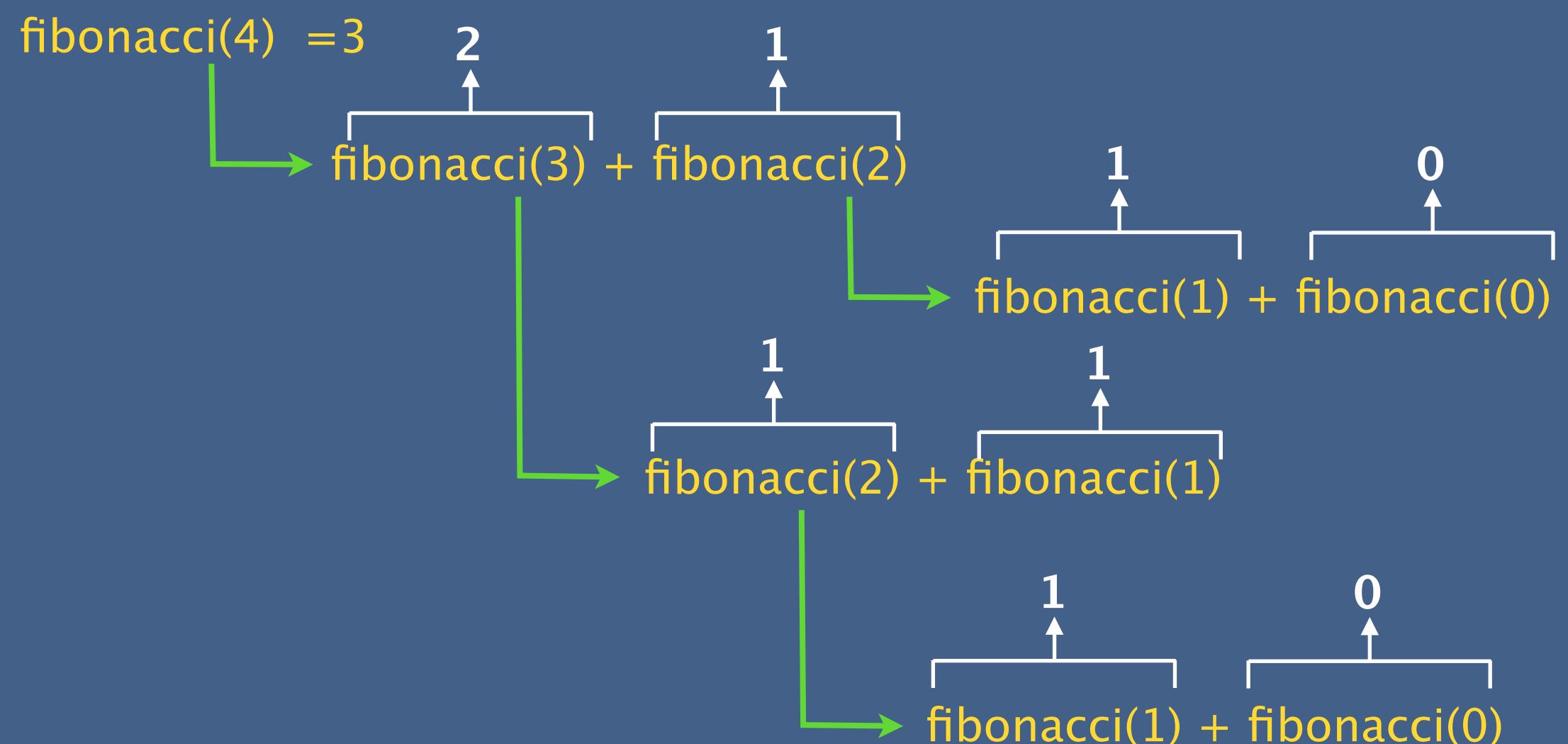
- fibonacci(-1) ??
- fibonacci(1.5) ??



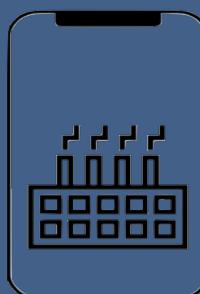
# Write Fibonacci Recursion in 3 Steps

```
public int fibonacci(int n) {  
    if (n < 0) {  
        return -1;  
    }  
    if (n == 0 || n == 1) {  
        return n;  
    }  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```

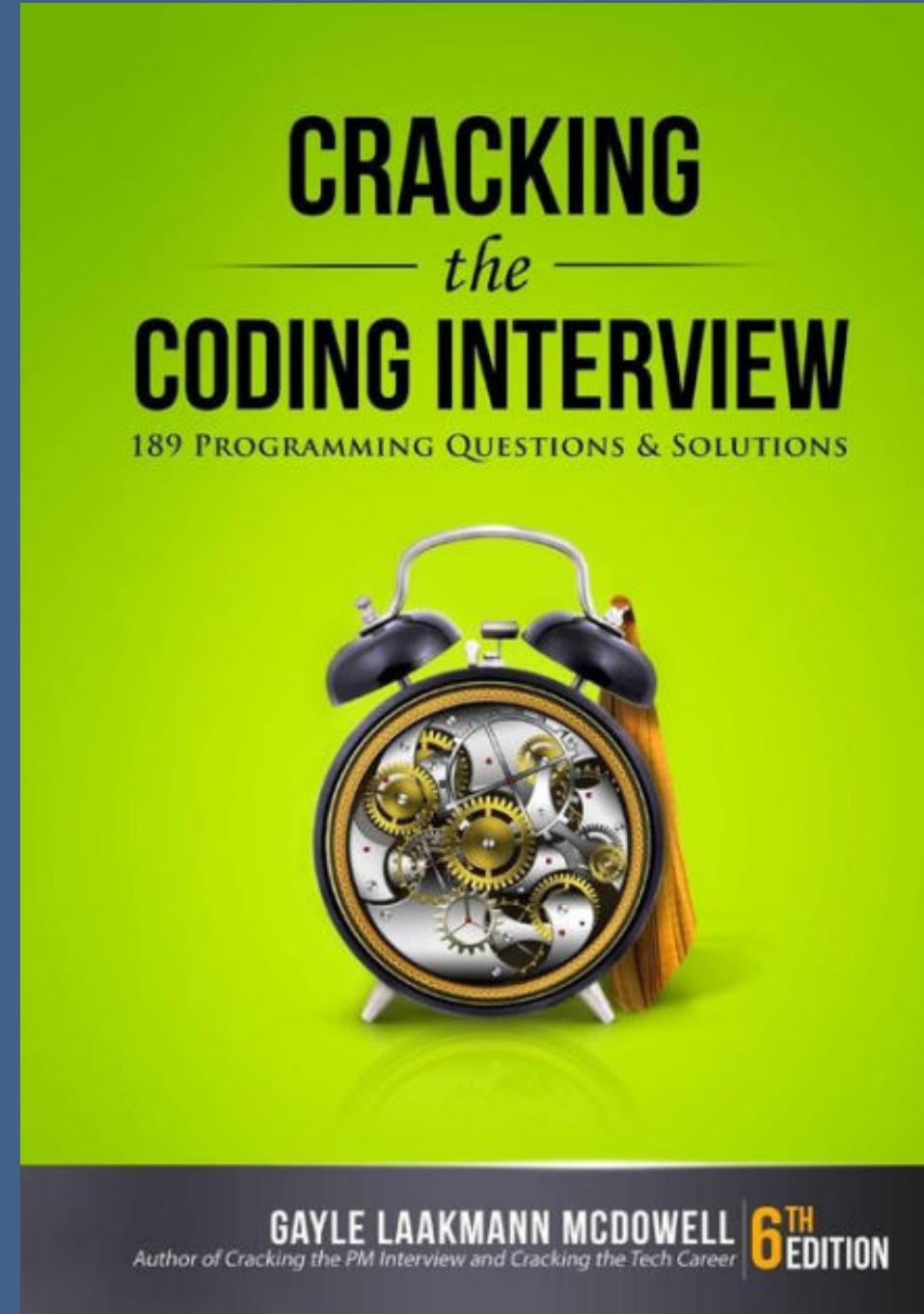
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...



# Recursion Interview Questions

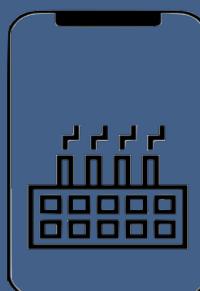
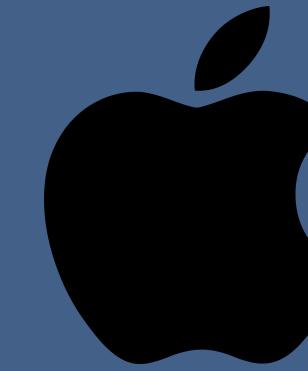


# Recursion Interview Questions



Google  
amazon

Microsoft



# Recursion 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

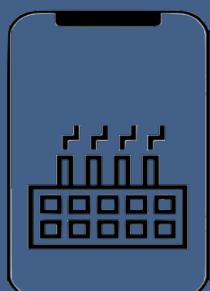
$11/10 = 1$  and Remainder = 1

**Step 2 : Base case - the stopping criterion**

–  $n = 0$

**Step 3 : Unintentional case - the constraint**

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



# Recursion Interview Questions - 2

How to calculate power of a number using recursion?

**Step 1 : Recursive case - the flow**

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

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

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

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

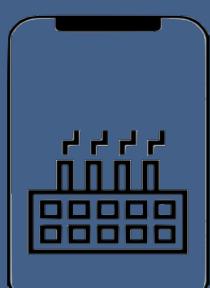
$$x^3 * x^4 = x^{3+4}$$

**Step 2 : Base case - the stopping criterion**

- $n = 0$
- $n = 1$

**Step 3 : Unintentional case - the constraint**

- $\text{power}(2, 1.2)$  ??
- $\text{power}(2, -1)$  ??



# Recursion 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**

$$\gcd(48,18)$$

$$\text{Step 1 : } 48/18 = 2 \text{ remainder } 12$$

$$\text{Step 2 : } 18/12 = 1 \text{ remainder } 6$$

$$\text{Step 3 : } 12/6 = 2 \text{ remainder } 0$$

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

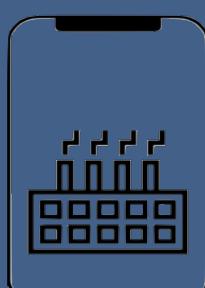
$$\gcd(a, b) = \gcd(b, a \bmod b)$$

**Step 2 : Base case - the stopping criterion**

- $b = 0$

**Step 3 : Unintentional case - the constraint**

- Positive integers



# Recursion 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

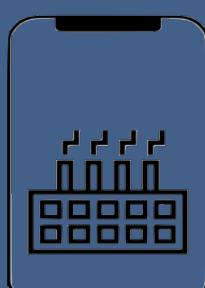
10 to binary

1000

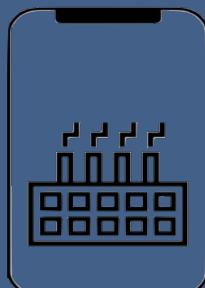
$$f(n) = n \bmod 2 + 10 * f(n/2)$$

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

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



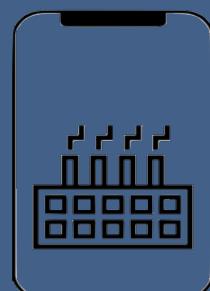
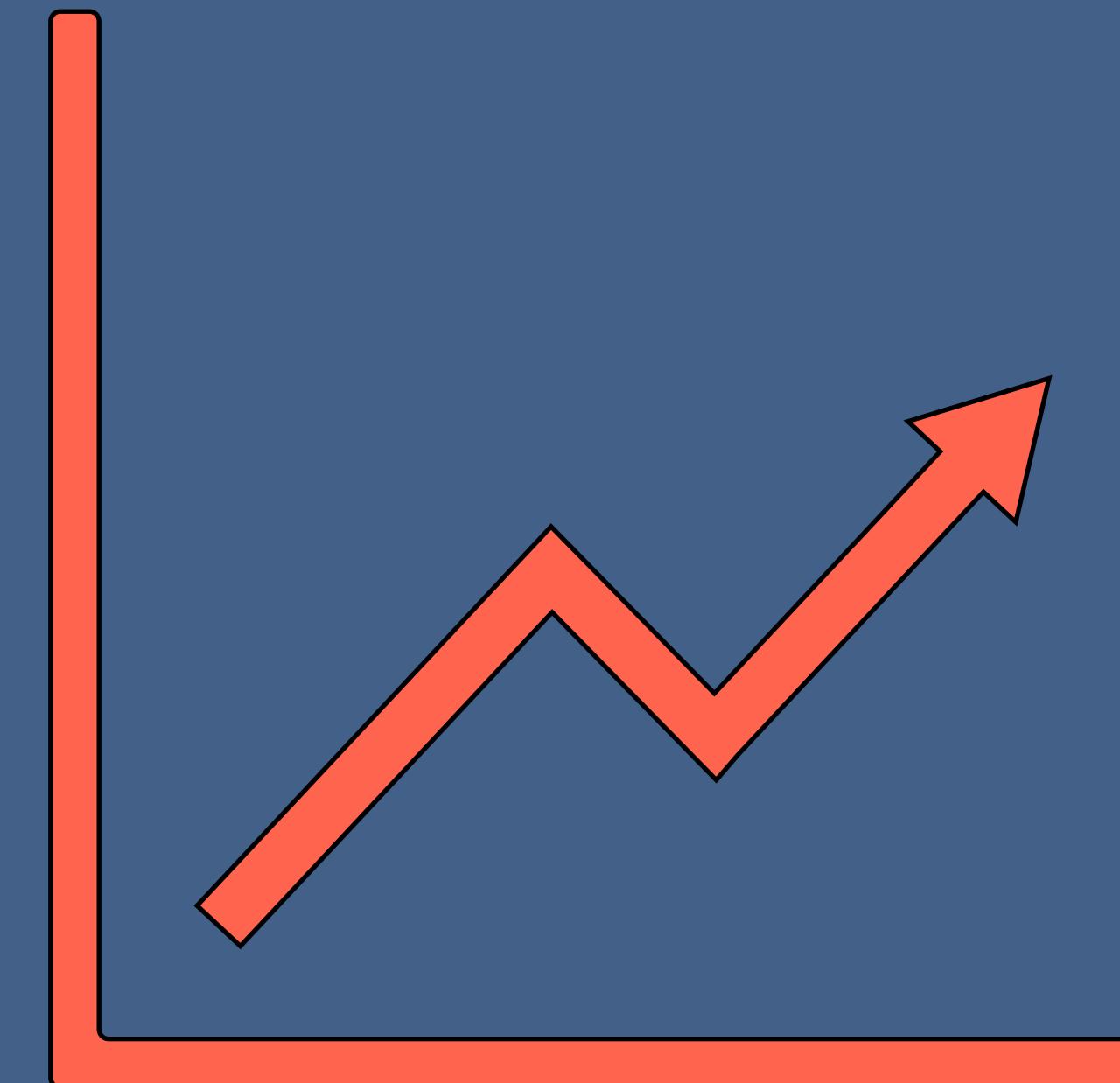
# Big O



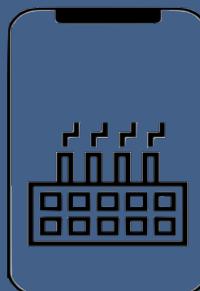
# Big O

## Objectives:

- What is Recursion?
- Why do we need Recursion?
- The Logic behind the Recursion
- Recursive vs Iterative Solutions
- How to Write Recursion in 3 Steps?
- Find Fibonacci Series using Recursion

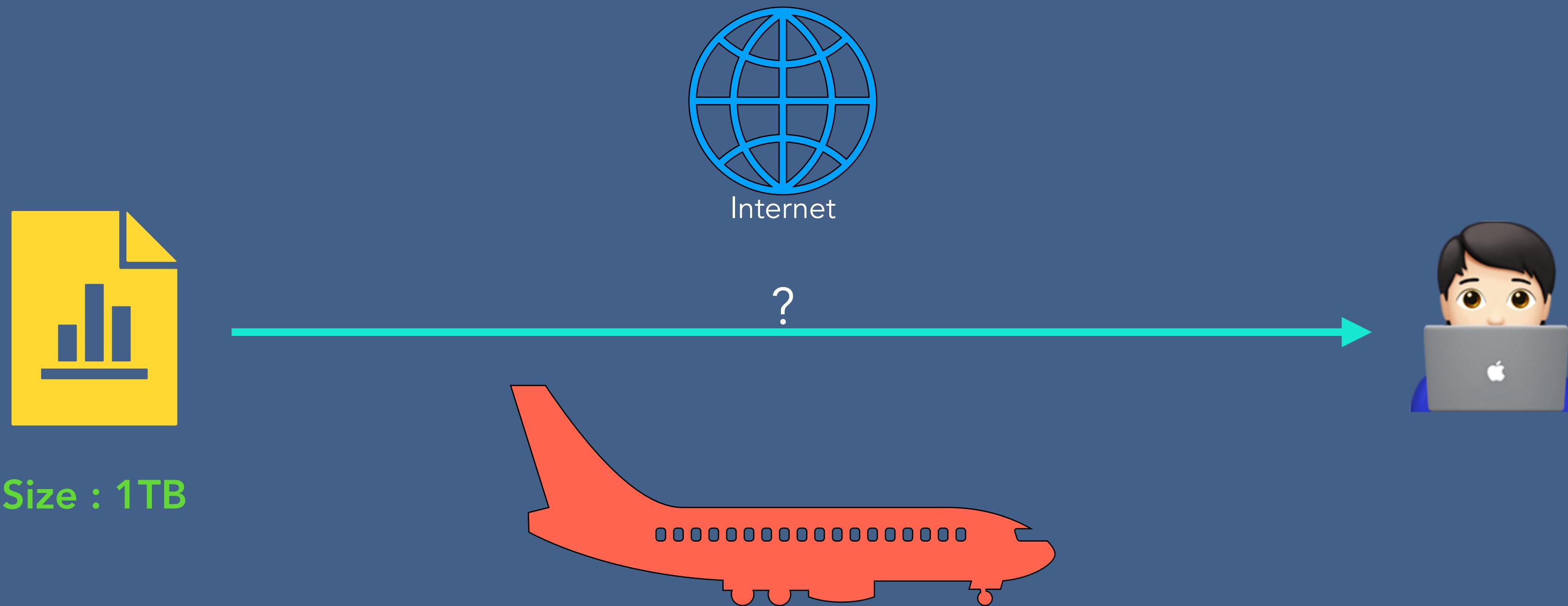


# What is Big O?

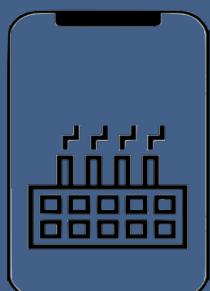


# What is Big O?

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

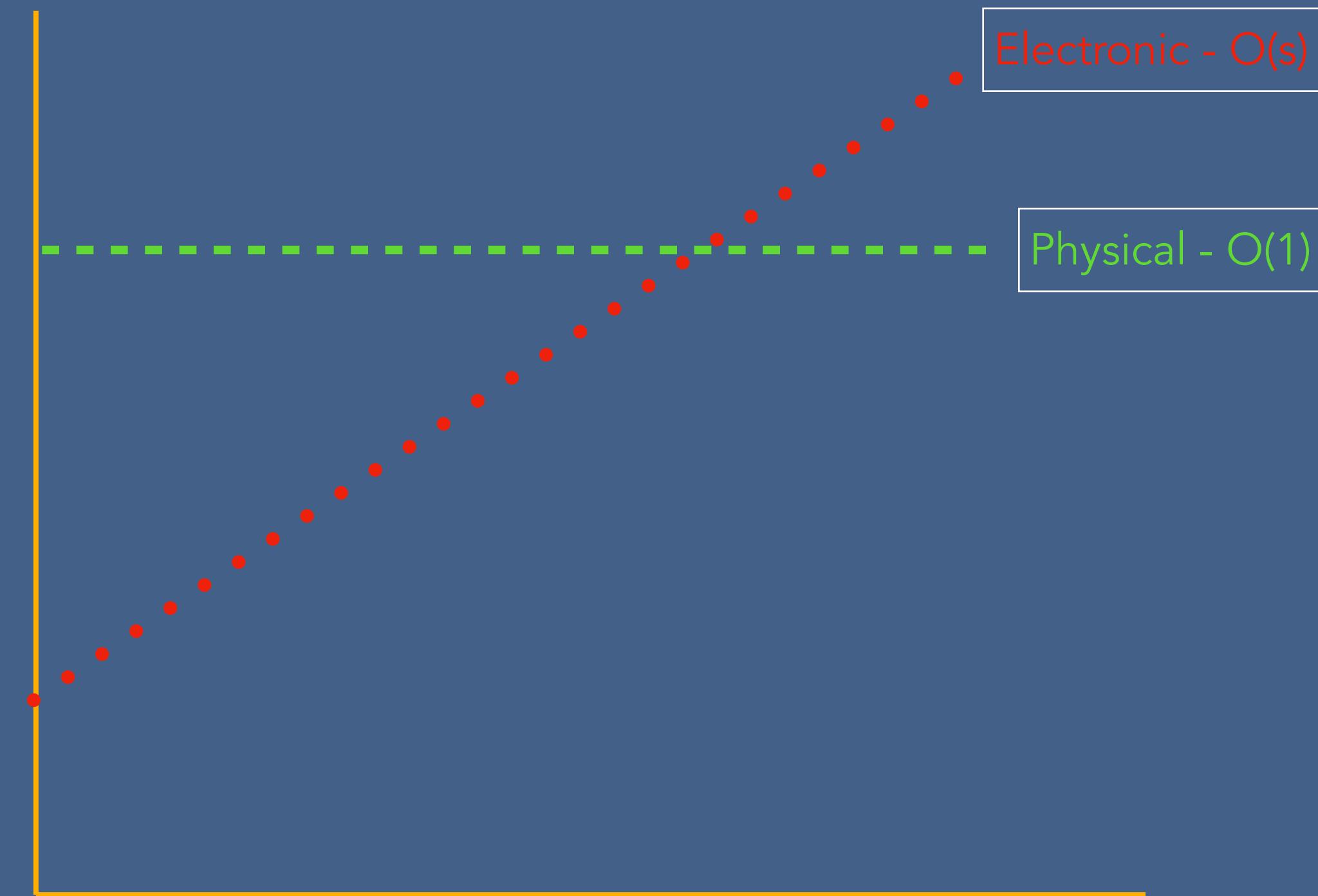


Time Complexity : A way of showing how the runtime of a function increases as the size of input increases.

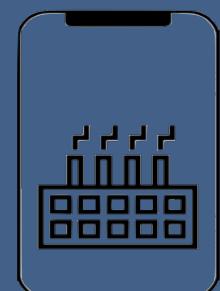


# What is Big O?

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



Time Complexity : A way of showing how the runtime of a function increases as the size of input increases.



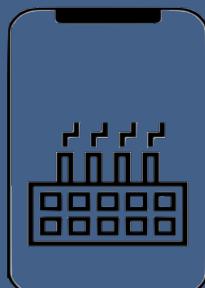
# What is Big O?

Types of Runtimes:

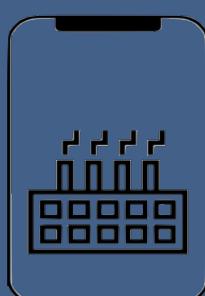
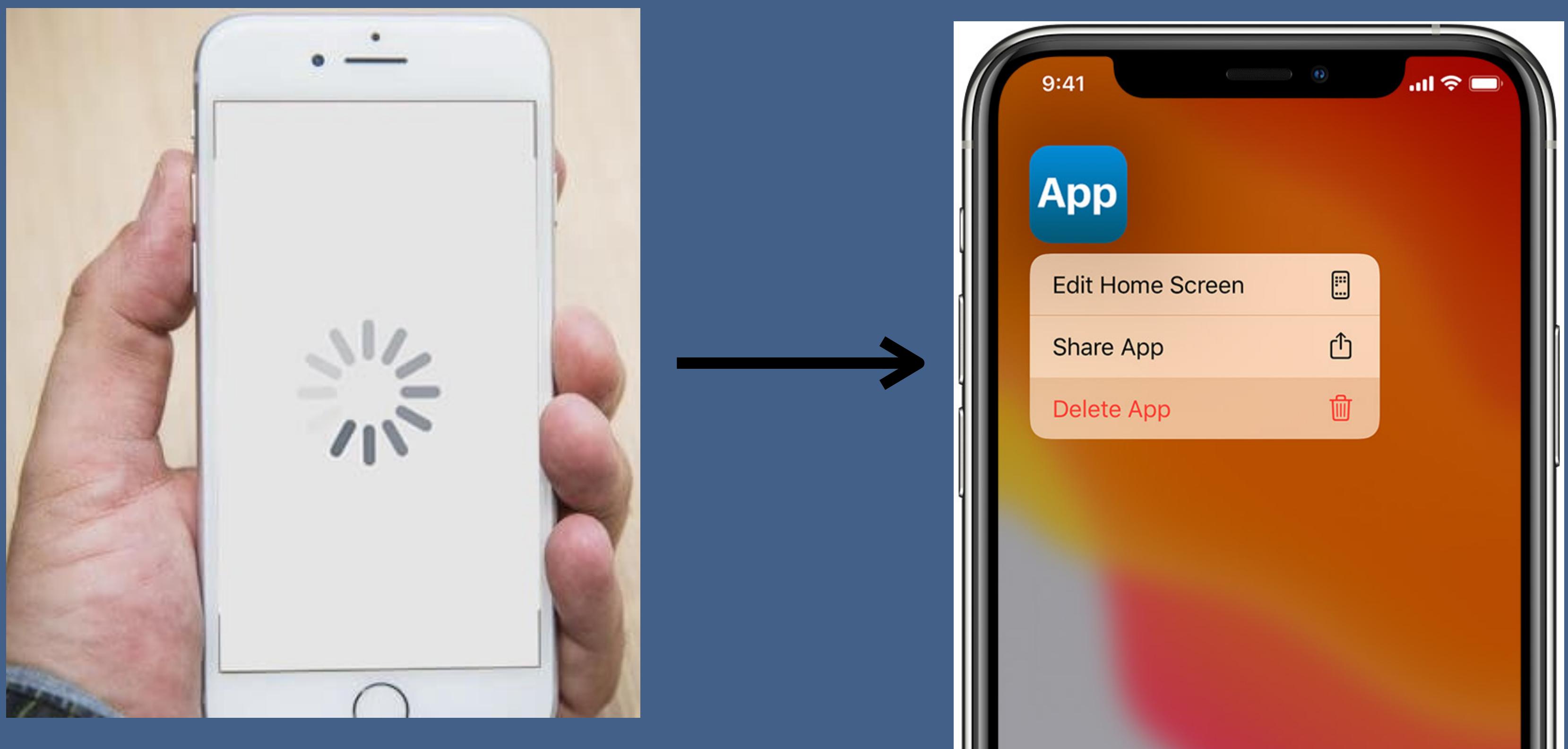
$O(N)$ ,  $O(N^2)$ ,  $O(2^N)$



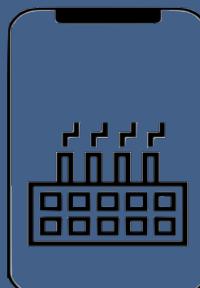
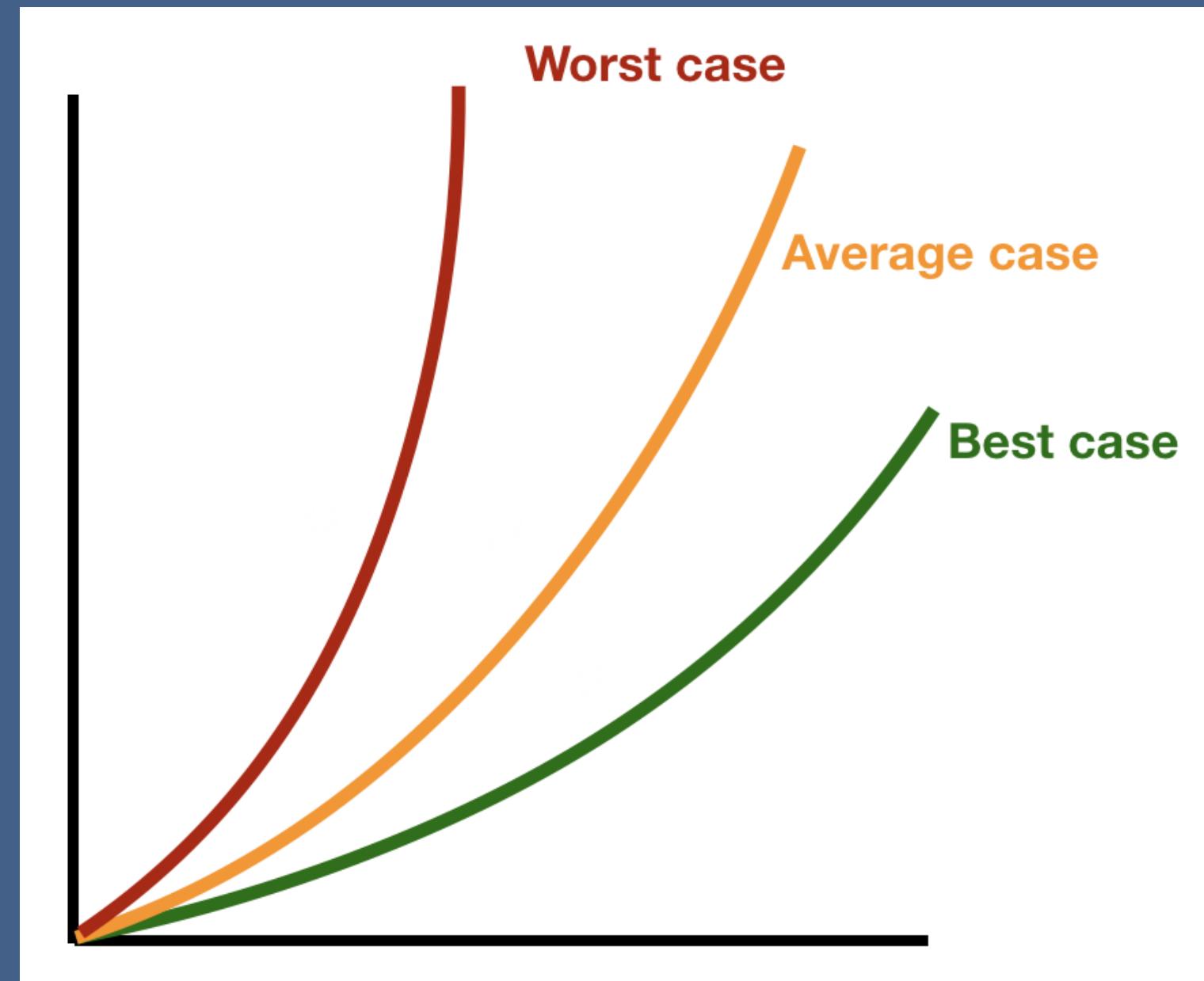
Time complexity :  $O(wh)$



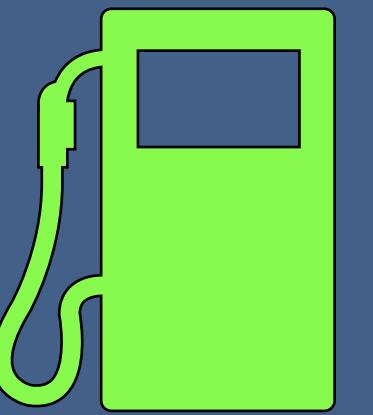
# What is Big O?



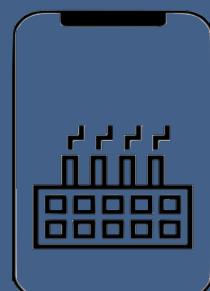
# Big O Notations



# Big O Notations

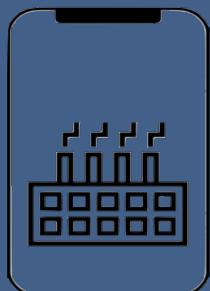


- City traffic - 20 liters
- Highway - 10 liters
- Mixed condition - 15 liters



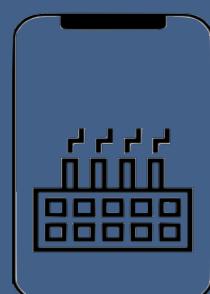
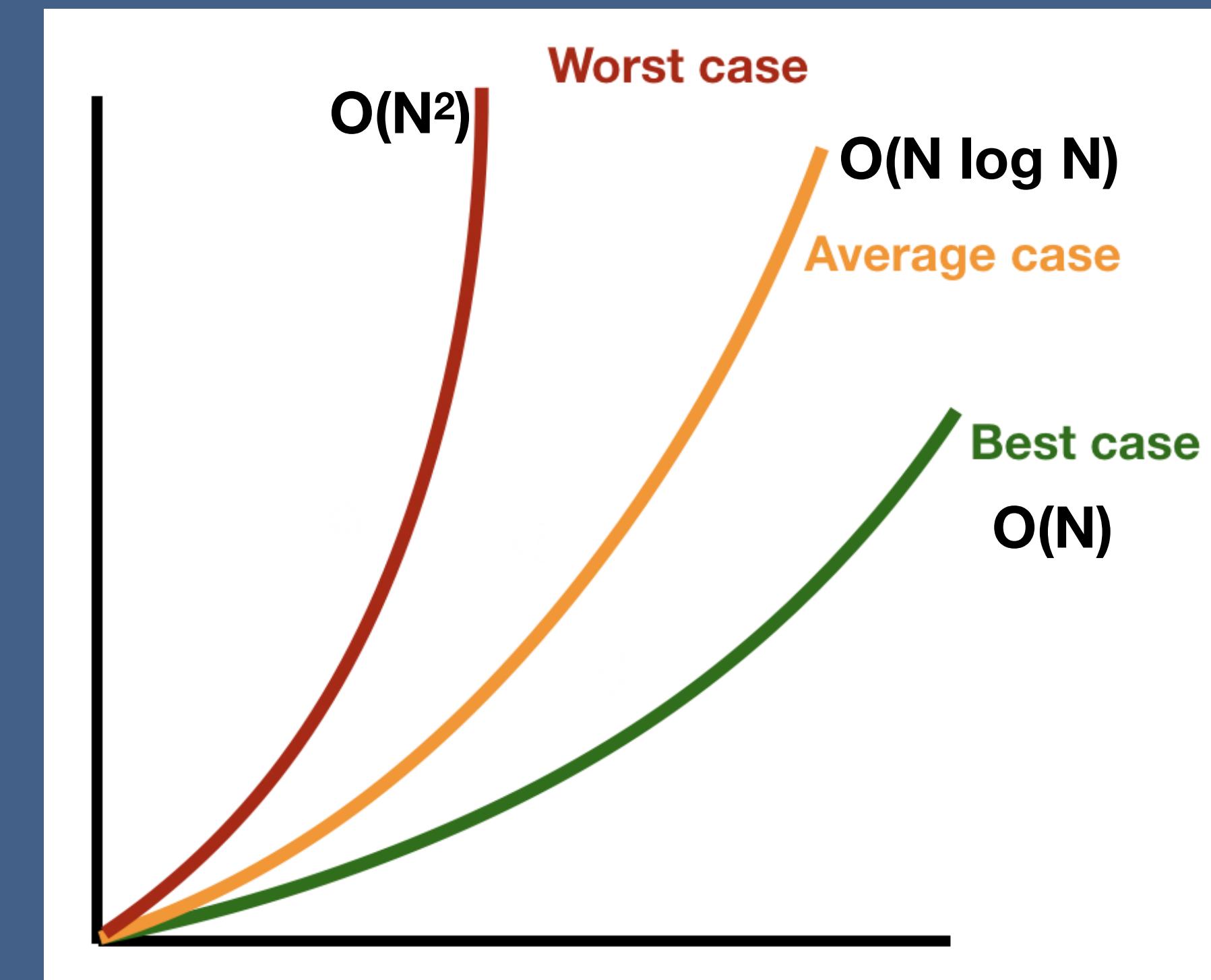
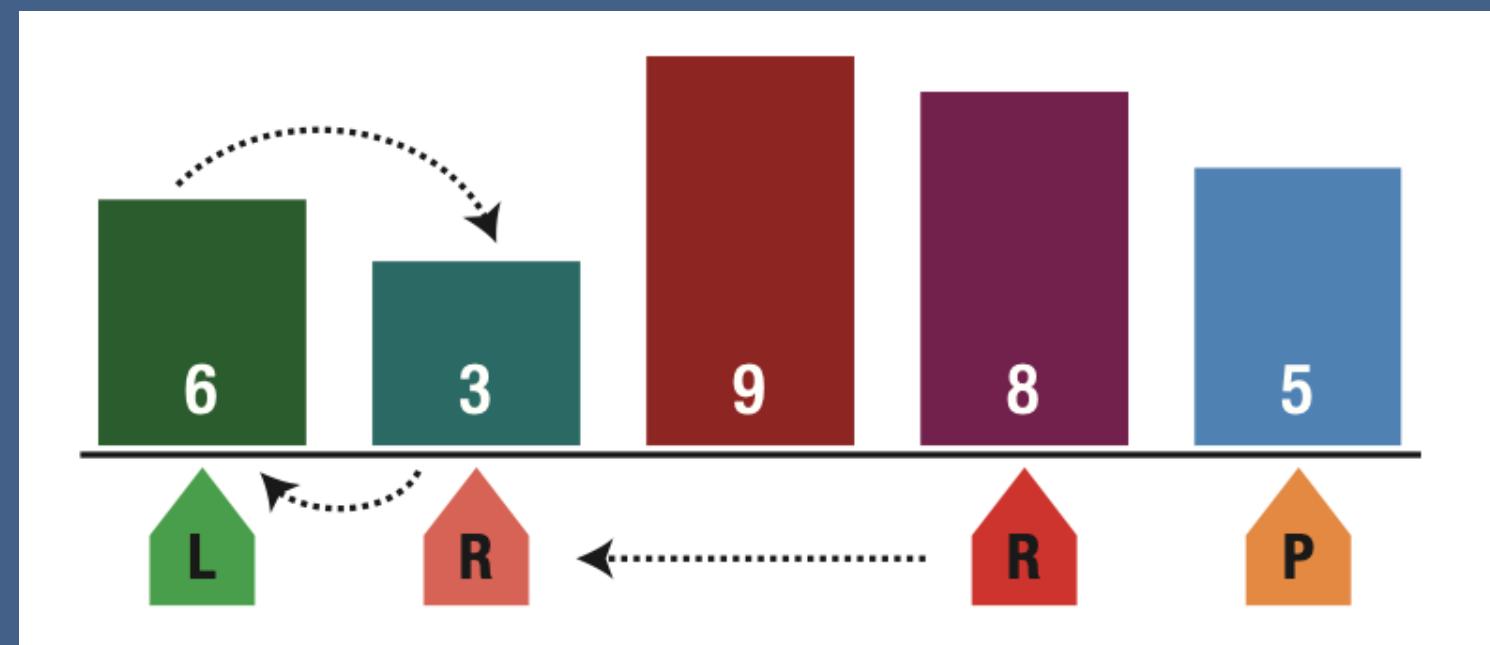
# Big O Notations

- Best case
- Average case
- Worst case



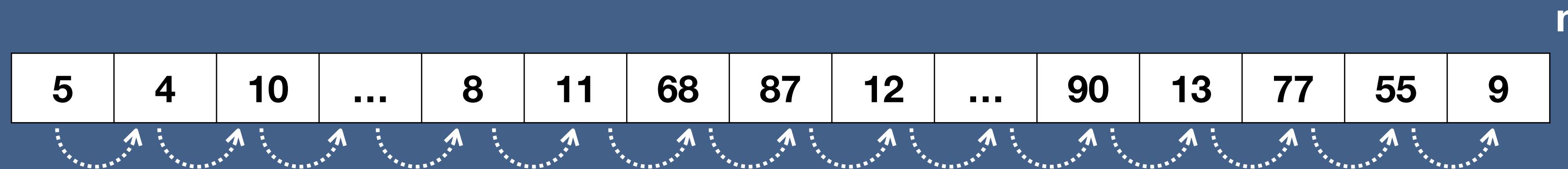
# Big O Notations

Quick sort algorithm



# Big O Notations

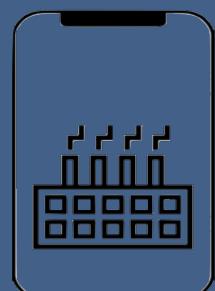
- **Big O** : It is a complexity that is going to be less or equal to the worst case.
- **Big -  $\Omega$  (Big-Omega)** : It is a complexity that is going to be at least more than the best case.
- **Big Theta (Big -  $\Theta$ )** : It is a complexity that is within bounds of the worst and the best cases.



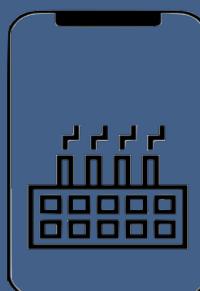
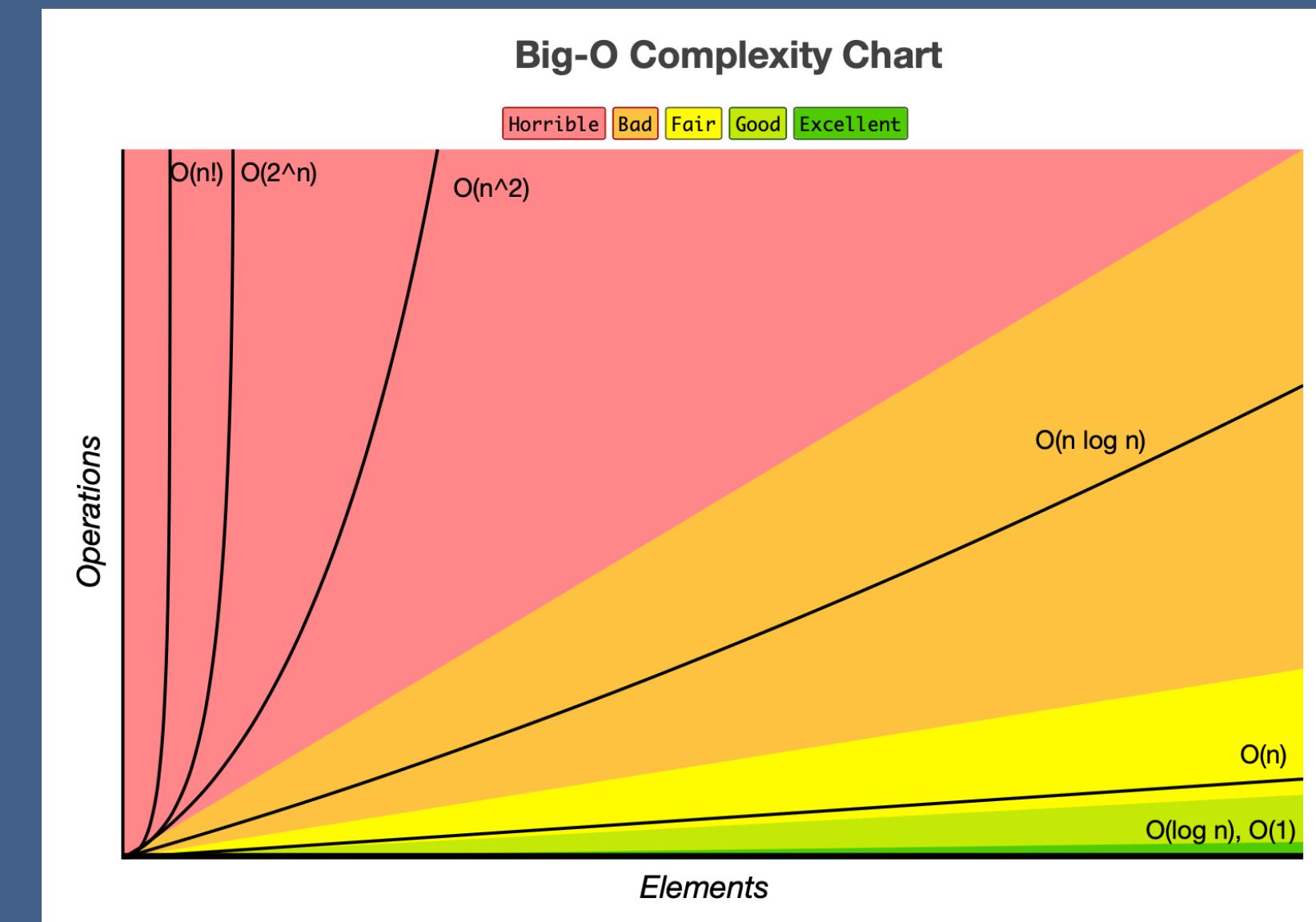
Big O -  $O(N)$

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

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



# Runtime Complexities

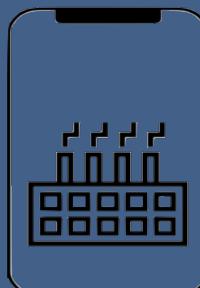


# Runtime Complexities

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

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



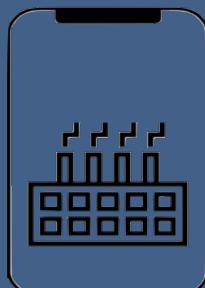
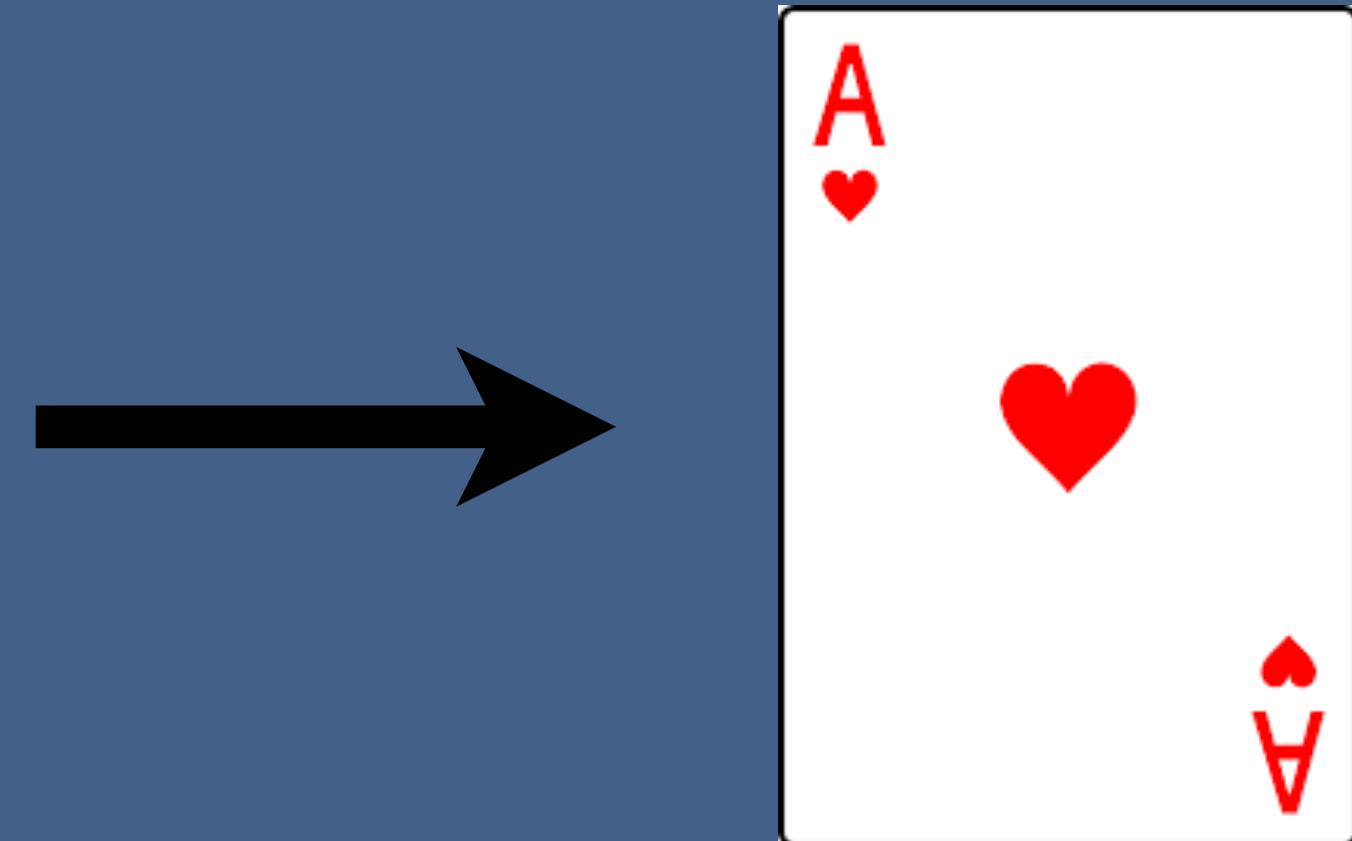
# Runtime Complexities

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



random card

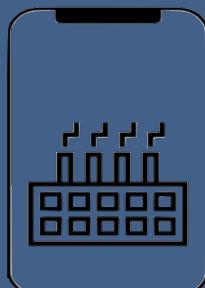


# Runtime Complexities

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

```
int[] custArray = {1, 2, 3, 4, 5}
for (int i = 0; i < custArray.length; i++) {
    System.out.println(custArray[i]);
}
//linear time since it is visiting every element of array
```



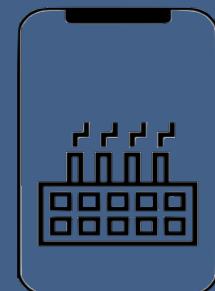
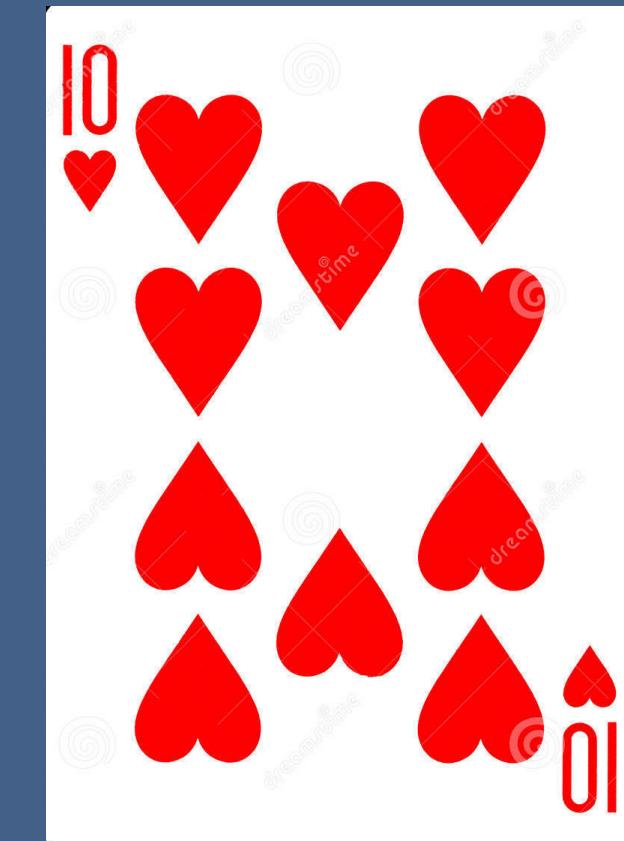
# Runtime Complexities

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

$O(N)$  - Linear time



Specific card

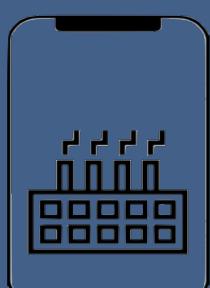


# Runtime Complexities

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

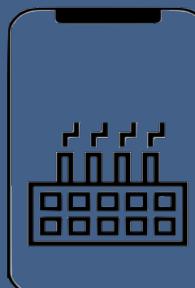
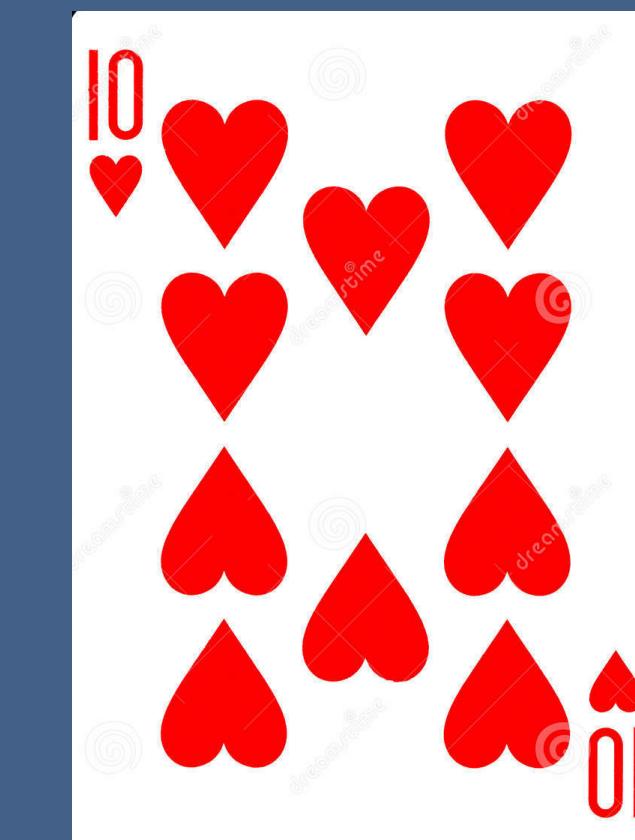
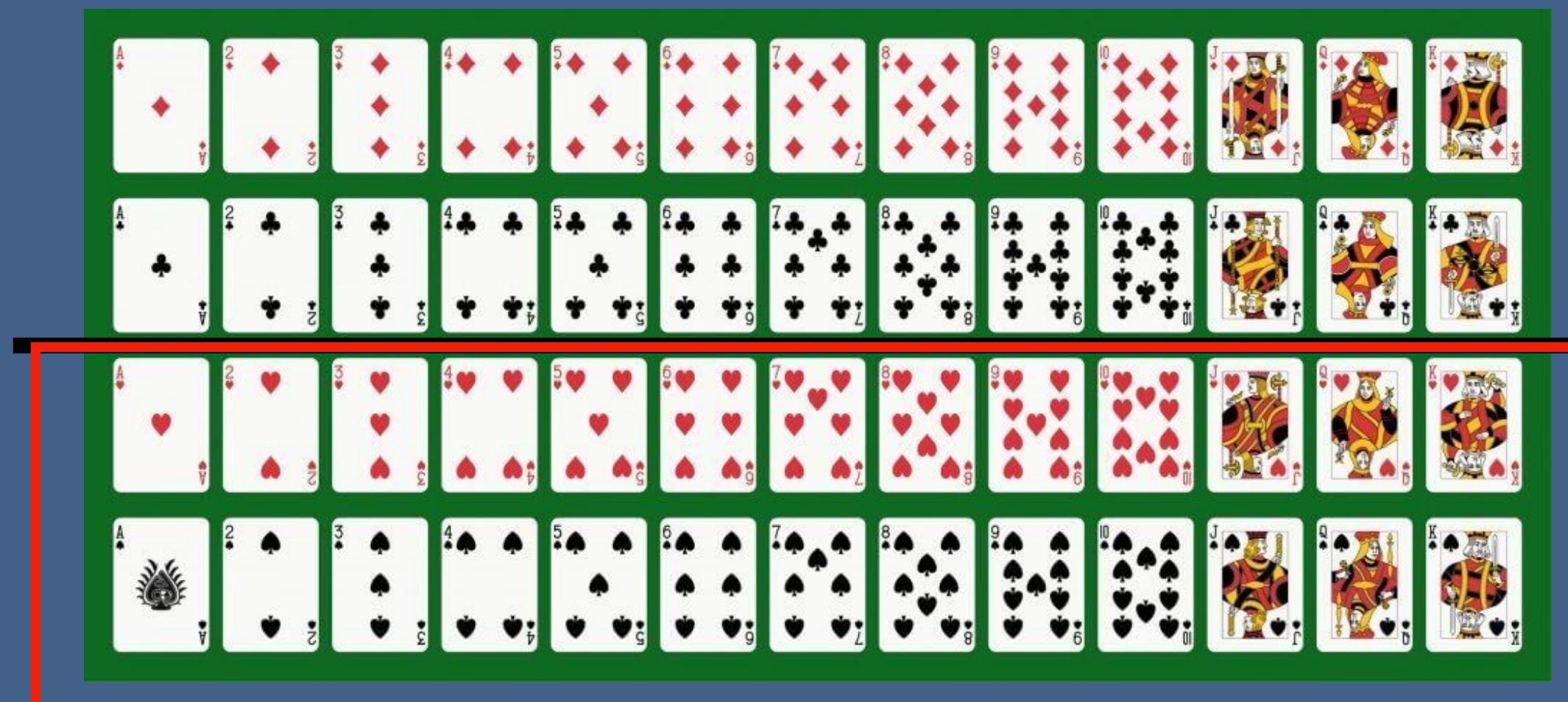
```
int[] custArray = {1, 2, 3, 4, 5}
for (int i = 0; i < custArray.length; i+3) {
    System.out.println(custArray[i]);
}
//logarithmic time since it is visiting only some elements
```



# Runtime 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



# Runtime 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

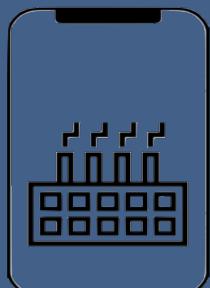
## $O(\log N)$ - Logarithmic time

### 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 */
```

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

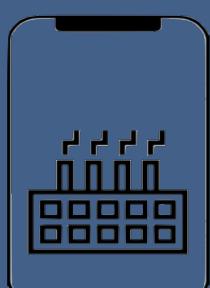


# Runtime Complexities

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

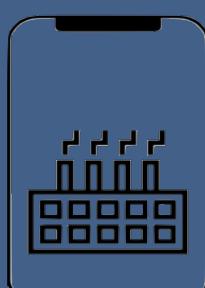
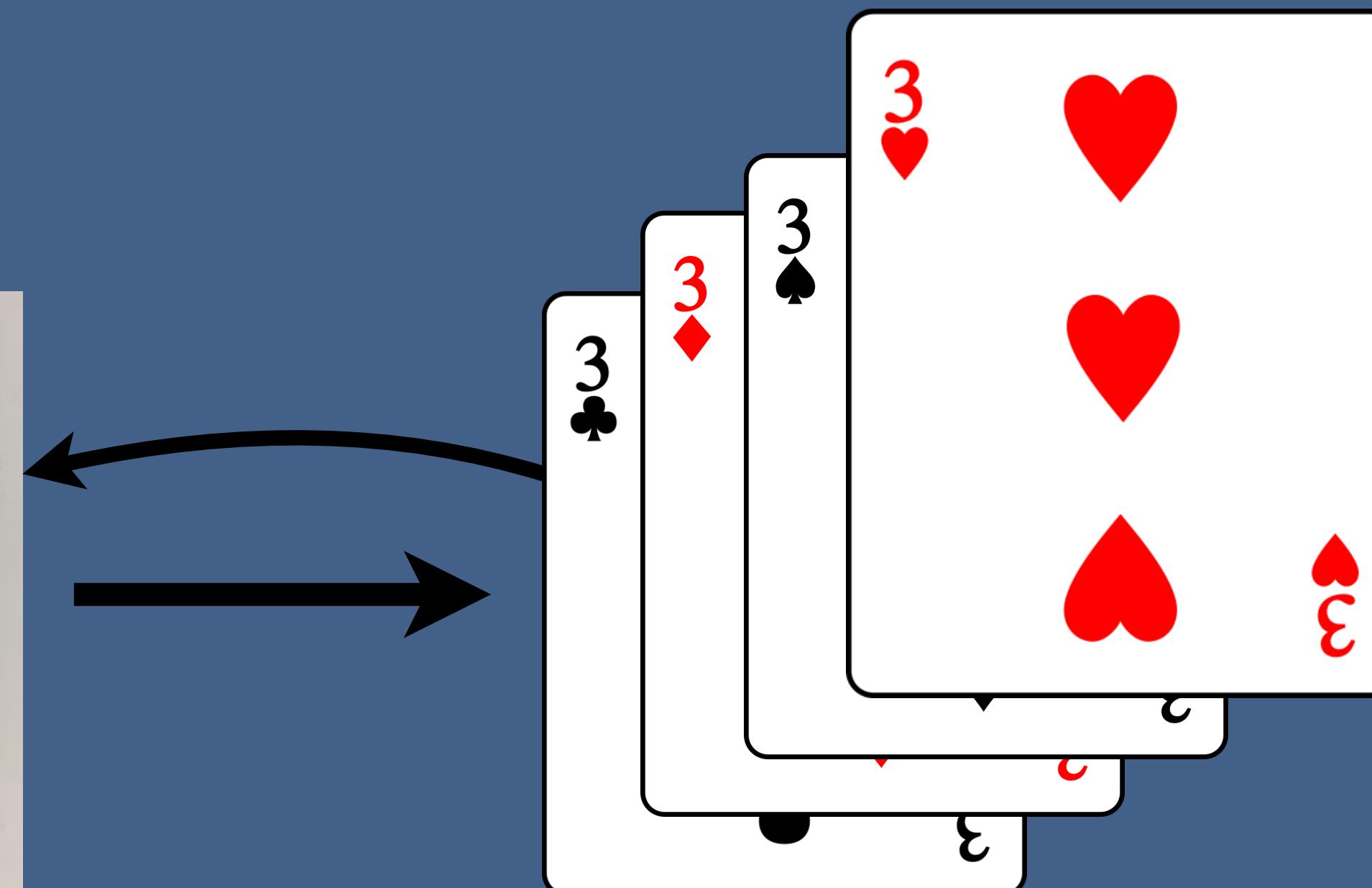
```
int[] custArray = {1, 2, 3, 4, 5}
for (int i = 0; i < custArray.length; i++) {
    for (int j = 0; j < custArray.length; j++) {
        System.out.println(custArray[i]);
    }
}
```



# Runtime 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

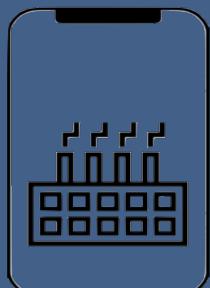


# Runtime Complexities

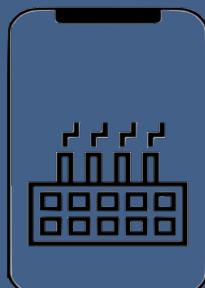
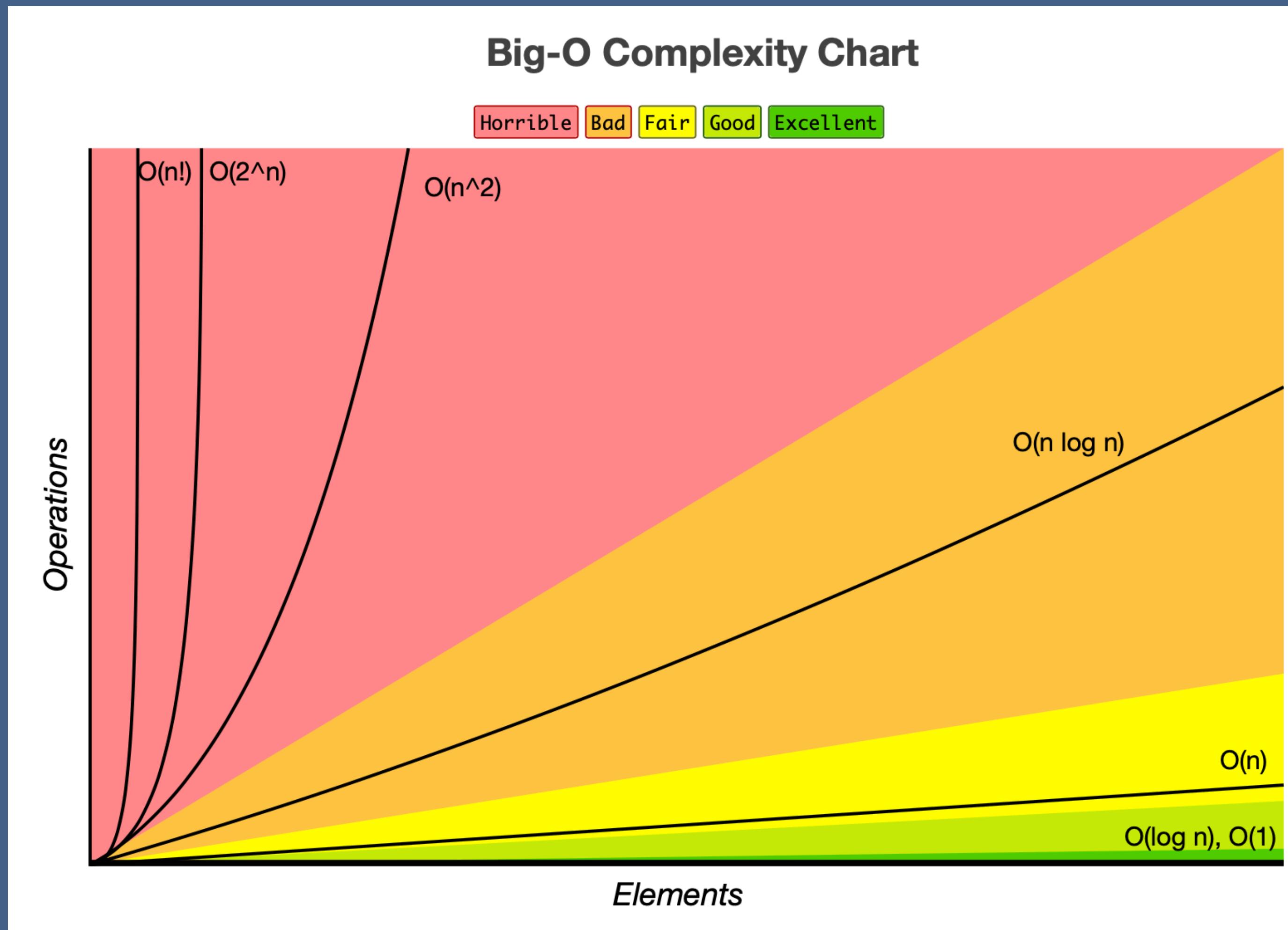
Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

## $O(2^N)$ - Exponential time

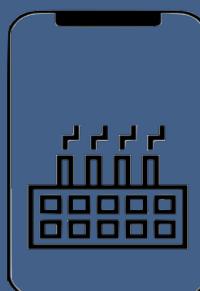
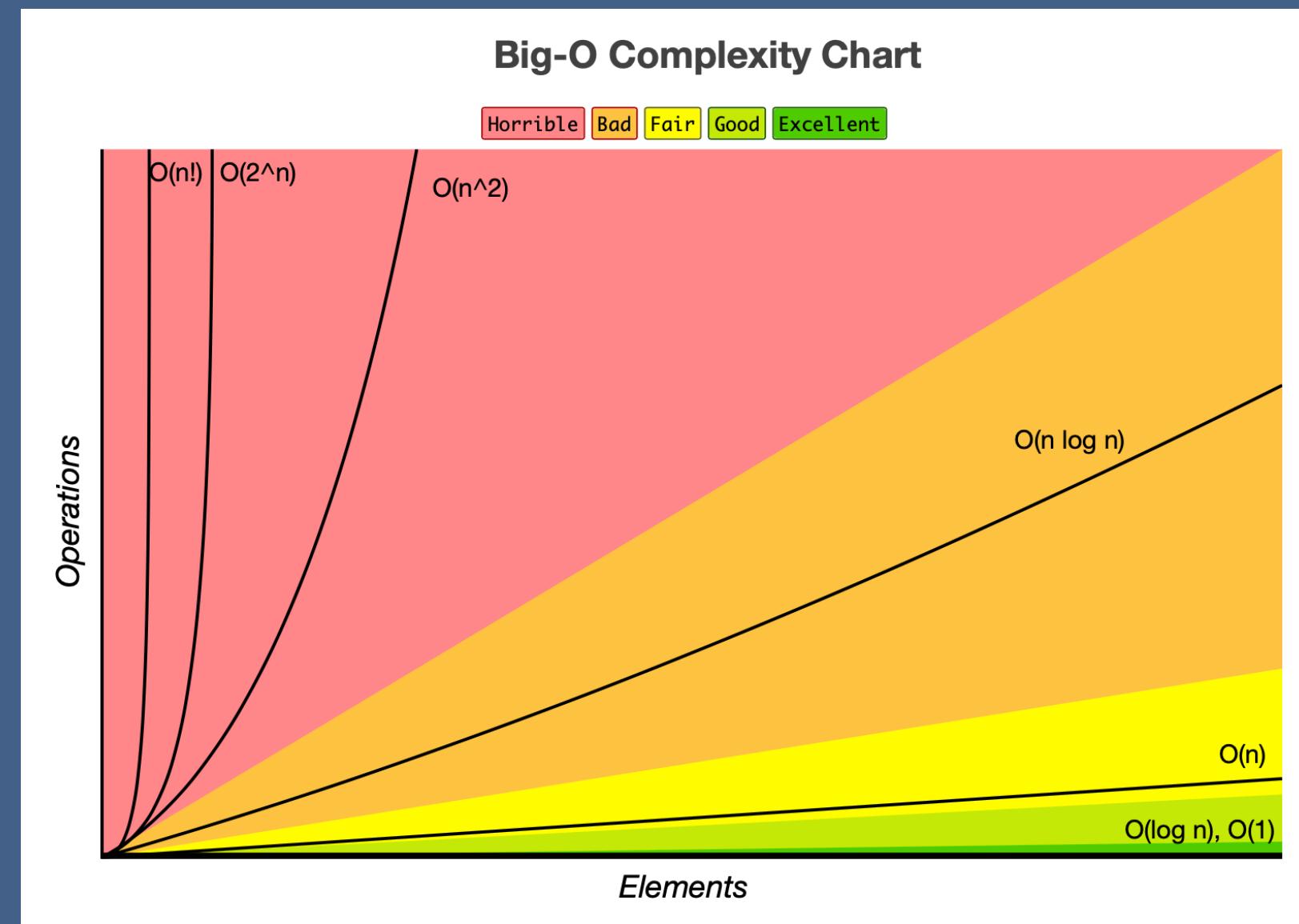
```
public int fibonacci(int n) {  
  
    if (n==0 || n==1) {  
        return n;  
    }  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```



# Runtime Complexities



# Space Complexity



# Space Complexity

an array of size  $n$

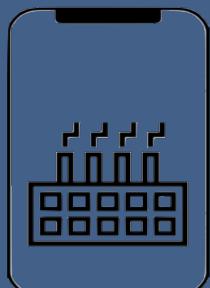
$$a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

$O(n)$

an array of size  $n \times n$

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

$O(n^2)$

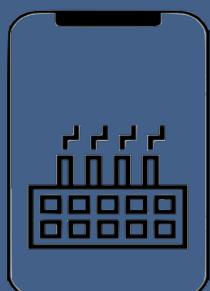


# Space Complexity

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

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

Space complexity : O(n)

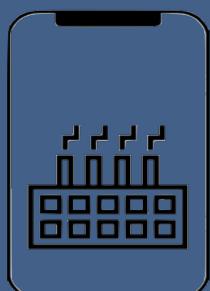


# Space Complexity

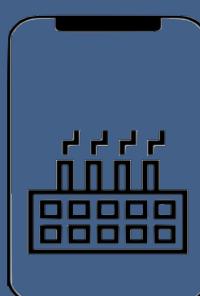
```
static int pairSumSequence(int n) {  
    var sum = 0;  
    for (int i = 0; i <= n; i++) {  
        sum = sum + pairSum(i, i+1);  
    }  
    return sum;  
}
```

```
static int pairSum(int a, int b) {  
    return a + b;  
}
```

Space complexity : O(1)



# Drop Constants and Non Dominant Terms



# Drop Constants and Non Dominant Terms

## Drop Constant

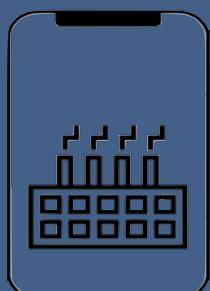
$$O(2N) \longrightarrow O(N)$$

## Drop Non Dominant Terms

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

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

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



# 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

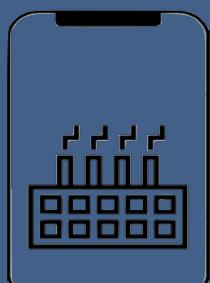


Slow computer  
Slow memory access  
Higher constant

- 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

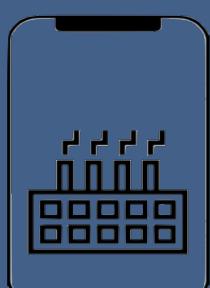
```
for (a=0; arrayA.length; a++) {  
    System.out.println(arrayA[a]);  
}  
  
for (b=0; arrayB.length; b++) {  
    System.out.println(arrayB[b]);  
}
```

```
for (a=0; arrayA.length; a++) {  
    for (b=0; arrayB.length; b++) {  
        System.out.println(arrayB[b] + arrayA[a]);  
    }  
}
```

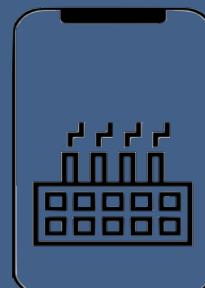
**Add the Runtimes:  $O(A + B)$**

- 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.

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



# How to measure the codes using Big O?



# How to measure the codes using Big O?

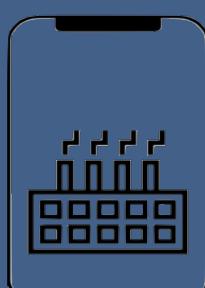
No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	$O(1)$
Rule 2	A simple “for” loop from 0 to n ( with no internal loops)	$O(n)$
Rule 3	A nested loop of the same type takes quadratic time complexity	$O(n^2)$
Rule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

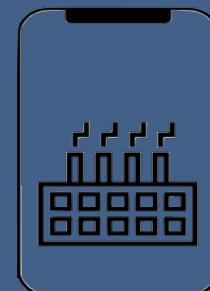


```
Public static void findBiggestNumber( [] sampleArray) {  
    var biggestNumber = sampleArray[0]; ..... → O(1)  
    for (index=1; sampleArray.length; index++) { ..... → O(n)  
        if (sampleArray[index] > biggestNumber) { ..... → O(1) } ..... → O(n)  
            biggestNumber = sampleArray[index]; ..... → O(1) } ..... → O(1)  
    } ..... → O(1)  
    System.out.println(biggestNumber); ..... → O(1)  
}
```

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



# How to measure Recursive Algorithm?



# How to measure Recursive Algorithm?

sampleArray

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

```
public int findMaxNumRec(int [] sampleArray, int n):
    if (n == 1) {
        return sampleArray[0];
    }
    return max(sampleArray[n-1], findMaxNumRec(sampleArray, n-1));
```

**Explanation:**

$$A = \begin{array}{|c|c|c|c|} \hline 11 & 4 & 12 & 7 \\ \hline \end{array} \quad n = 4$$

`findMaxNumRec(A, 4)`  $\longrightarrow$  `max(A[4-1], 12)`  $\longrightarrow$  `max(7, 12)=12`



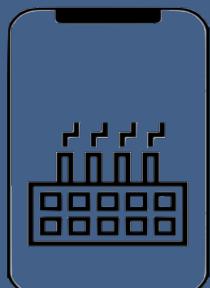
`findMaxNumRec(A, 3)`  $\longrightarrow$  `max(A[3-1], 11)`  $\longrightarrow$  `max(12, 11)=12`



`findMaxNumRec(A, 2)`  $\longrightarrow$  `max(A[2-1], 11)`  $\longrightarrow$  `max(4, 11)=11`



`findMaxNumRec(A, 1)`  $\longrightarrow$  `A[0]=11`



# How to measure Recursive Algorithm?

sampleArray

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

```
public int findMaxNumRec(int [] sampleArray, int 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)
```

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

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

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

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



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

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

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

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

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

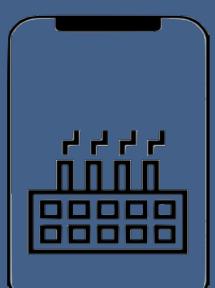
.

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

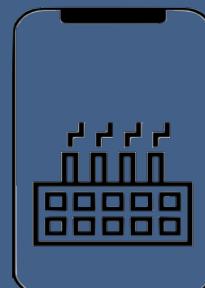
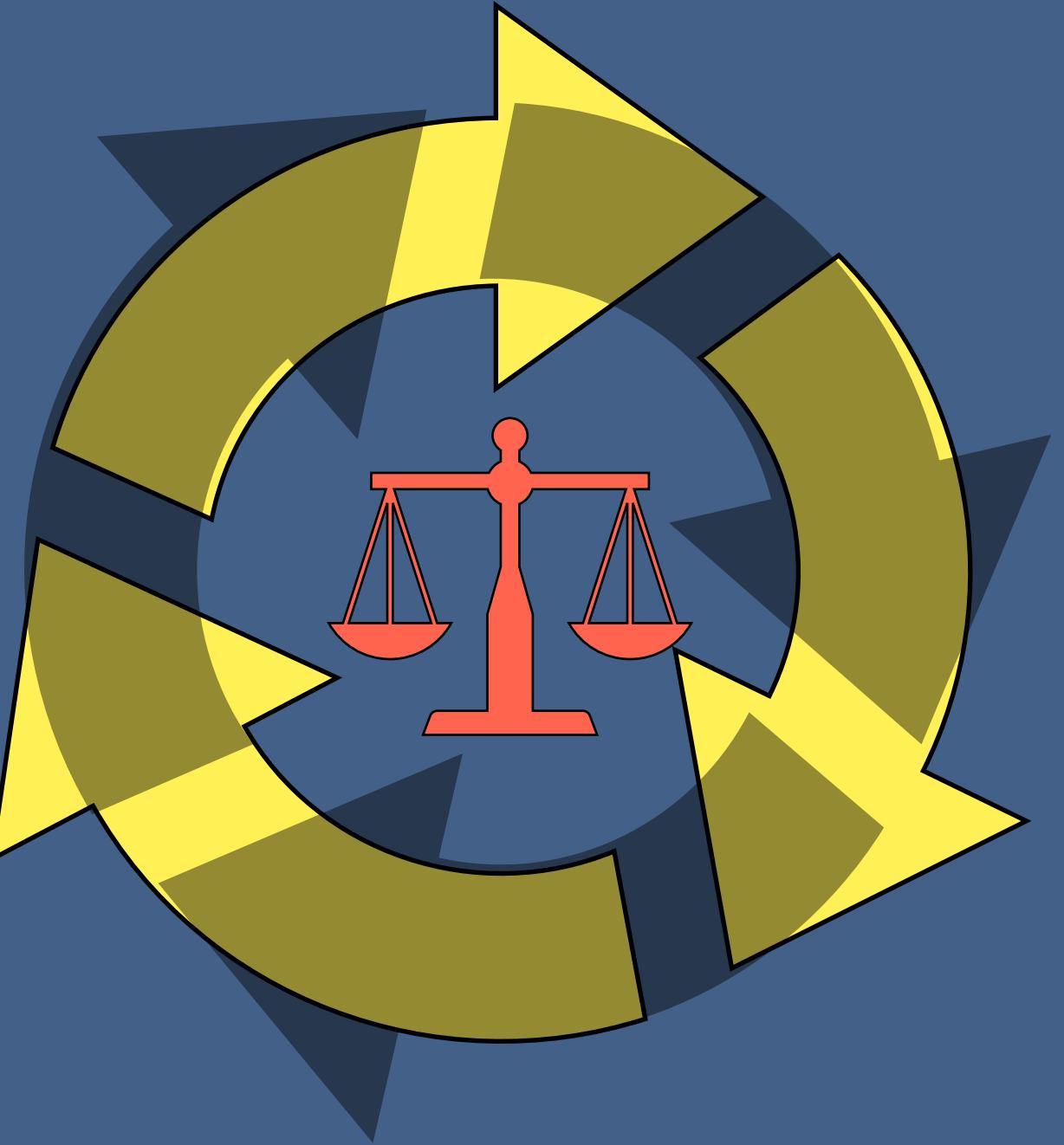
$$=n-1+M(n-(n-1))$$

$$=n-1+1$$

$$=n$$



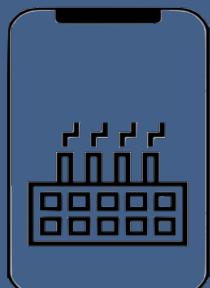
# How to measure Recursive Algorithm with multiple calls?



# How to measure Recursive Algorithm with multiple calls?

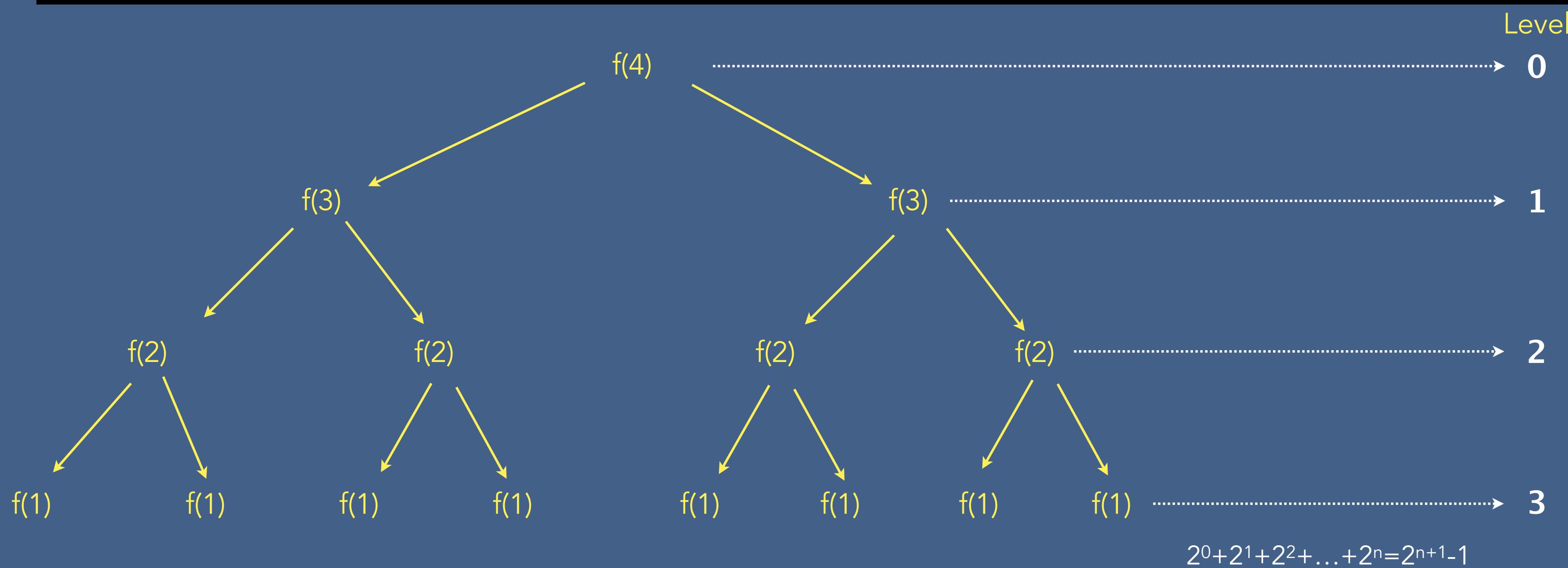
```
public int findMaxNumRec(int [] sampleArray, int n):  
    if (n == 1) {  
        return sampleArray[0]; }  
    return max(sampleArray[n-1], findMaxNumRec(sampleArray, n-1));
```

```
public int f(int n):  
    if (n <= 1) {  
        return 1; }  
    return f(n-1) + f(n-1);
```



# How to measure Recursive Algorithm with multiple calls?

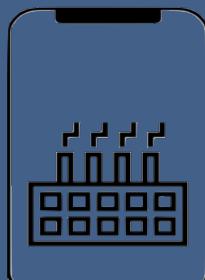
```
public int f(int n):
    if (n <= 1) {
        return 1;
    }
    return f(n-1) + f(n-1);
```



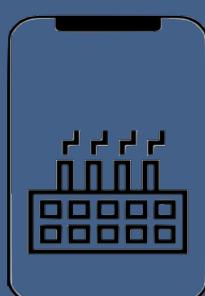
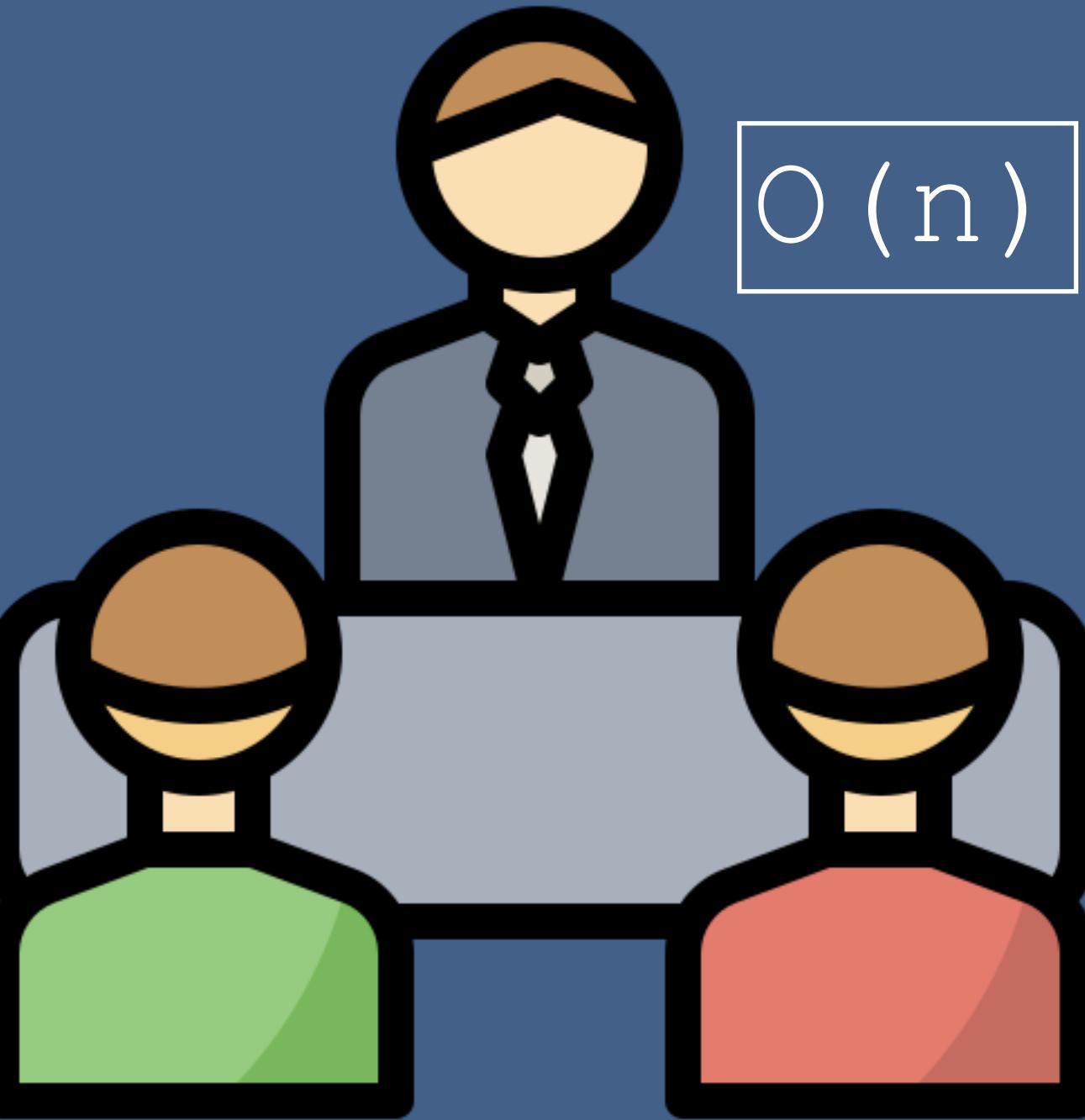
N	Level	Node#	Also can be expressed..	or..
4	0	1		$2^0$
3	1	2	$2 * \text{previous level} = 2$	$2^1$
2	2	4	$2 * \text{previous level} = 2 * 2^1 = 2^2$	$2^2$
1	3	8	$2 * \text{previous level} = 2 * 2^2 = 2^3$	$2^3$

O(branches<sup>depth</sup>)

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

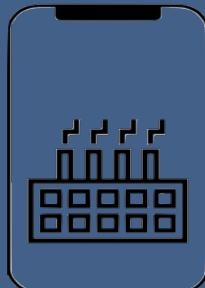


# Big O Interview Questions



# Interview Question 1

- Create a function which calculates the sum and product of elements of array
- Find the time complexity for created method.



# Interview Question 2

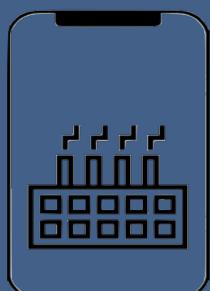
- Create a function which prints to the console the pairs from given array.
- Find the time complexity for created method.

[1,3,4,5]      11, 13, 14, 15

31, 33, 34, 35

41, 43, 44, 45

51, 53, 54, 55



# Interview Question 3

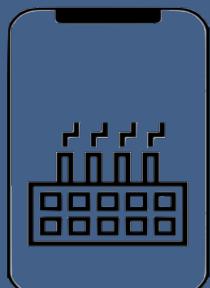
- What is the time complexity for this method?

```
void printUnorderedPairs(int[] array) {  
    for (int i=0; i<array.length; i++) {  
        for (int j=i+1; j<array.length; j++) {  
            System.out.println(array[i] + ", " + array[j]);  
        }  
    }  
}
```

[1,3,4,5]      13, 14, 15

34, 35

45



# Interview Question 3

- What is the time complexity for this method?

```
void printUnorderedPairs(int[] array) {  
    for (int i=0; i<array.length; i++) {  
        for (int j=i+1; j<array.length; j++) {  
            System.out.println(array[i] + ", " + array[j]);  
        }  
    }  
}
```

## 1. Counting the iterations

1st  $\longrightarrow n-1$

2nd  $\longrightarrow n-2$

.

.

1

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

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

$=n(n-1)/2$

$=n^2/2 - n/2$

$=n^2$

## 2. Average Work

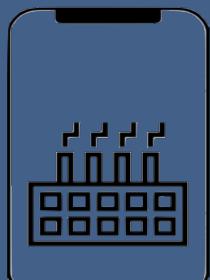
Outer loop - N times

Inner loop?

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

$\left. \begin{matrix} \\ \\ \\ \end{matrix} \right\} = 5 \longrightarrow 10/2$   
 $n \longrightarrow n/2$

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



Time Complexity :  $O(N^2)$

# Interview Question 4

What is the runtime of the below code?

```
void printUnorderedPairs (int[] arrayA, int[] arrayB) {  
    for (int i=0; i < arrayA.length; i++) {  
        for (int j=0; j < arrayB.length; j++) {  
            if (arrayA[i] < arrayB[j]) {  
                System.out.println(arrayA[i] + "," + arrayB[j]);  
            }  
        }  
    }  
}
```

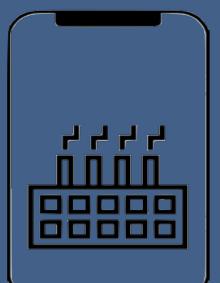
$O(ab)$

```
void printUnorderedPairs (int[] arrayA, int[] arrayB) {  
    for (int i=0; i < arrayA.length; i++) {  
        for (int j=0; j < arrayB.length; j++) {  
            O(1)  
        }  
    }  
}
```

a = arrayA.length;

Time Complexity :  $O(ab)$

b = arrayB.length;



# Interview Question 5

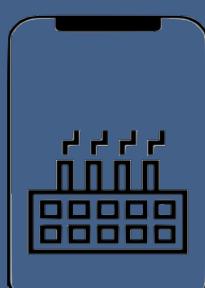
What is the runtime of the below code?

```
void printUnorderedPairs (int[] arrayA, int[] arrayB) {  
    for (int i=0; i < arrayA.length; i++) {  
        for (int j=0; j < arrayB.length; j++) {  
            for (int k = 0; k<1000000; k++) {  
                System.out.println(arrayA[i] + "," + arrayB[j]);  
            }  
        }  
    }  
}
```

b = arrayB.length;  
a = arrayA.length;

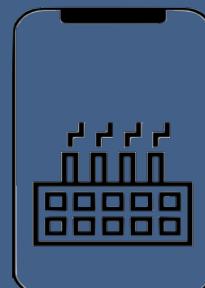
Time Complexity :  $O(ab)$

100,000 units of work is still constant



# Interview Question 6

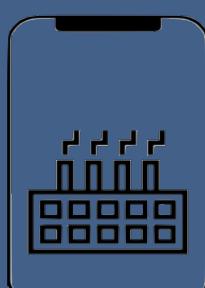
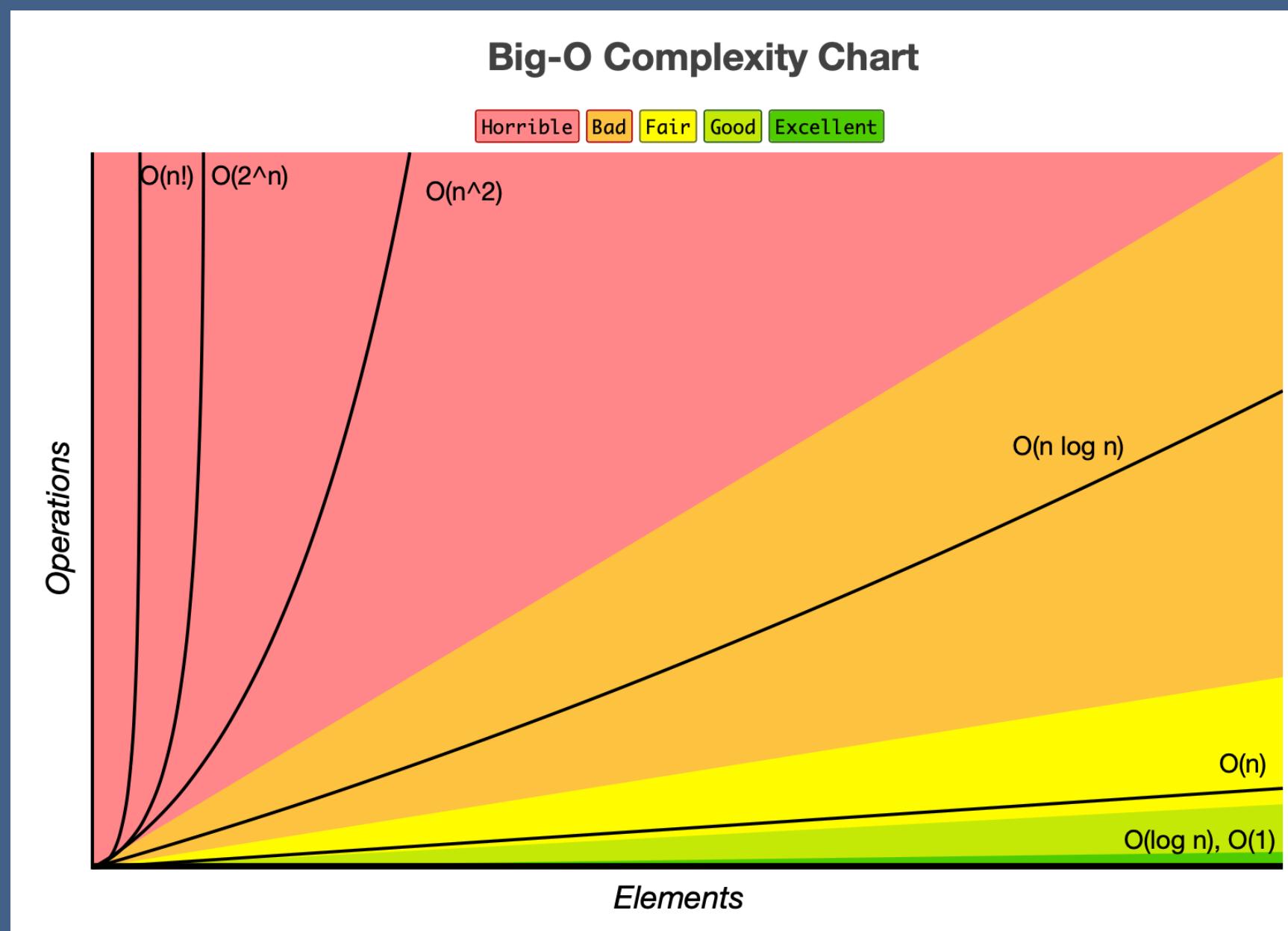
- Create a method which takes an array as a parameter and reverses it.
- Find the runtime of the created method?



# Interview Question 7

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

1.  ~~$O(N + P)$ , where  $P < N/2$~~   $\longrightarrow O(N)$  ✓
2.  ~~$O(2N)$~~   $\longrightarrow O(N)$  ✓
3.  ~~$O(N+\log N)$~~   $\longrightarrow O(N)$  ✓
4.  ~~$O(N + N\log N)$~~   $\longrightarrow O(N\log N)$  ✗
5.  $O(N+M)$  ✗



# Interview Question 8

What is the runtime of the below code?

```
static int factorial(int n) {  
    if (n < 0) {  
        return -1;  
    } else if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```

→ **M(n)**  
→ **O(1)**  
→ **M(n-1)**

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

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

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

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

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

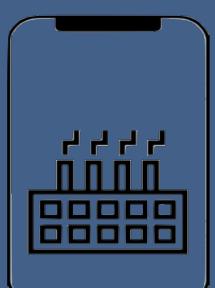
$$M(n-2) = O(1) + M((n-2)-1)$$

$$\begin{aligned} M(n) &= 1 + M(n-1) \\ &= 1 + (1 + M((n-1)-1)) \end{aligned}$$

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

$$\begin{aligned} &= n + M(n-n) \\ &= n+1 \\ &= n \end{aligned}$$

**Time Complexity : O(N)**



# Interview Question 9

What is the runtime of the below code?

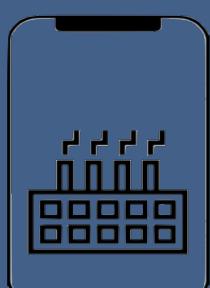
```
void allFib(int n) {  
    for (i=0; i<n; i++) {  
        fib(i);  
    }  
}  
  
static int fib(int n) {  
    if (n<=0) {  
        return 0;  
    } else if (n==1) {  
        return 1;  
    } else {  
        return fib(n-1) + fib(n-2);  
    }  
}
```

**branches<sup>depth</sup>** .....  $\rightarrow O(2^N)$

fib(1)  $\longrightarrow$   $2^1$  steps  
fib(2)  $\longrightarrow$   $2^2$  steps  
fib(3)  $\longrightarrow$   $2^3$  steps  
fib(4)  $\longrightarrow$   $2^4$  steps  
...  
fib(n)  $\longrightarrow$   $2^n$  steps

$\longrightarrow$  Total work =  $2^1+2^2+2^3+2^4+\dots+2^n$   
 $= 2^{n+1}-2$

**Time Complexity :  $O(2^N)$**



# Interview Question 10

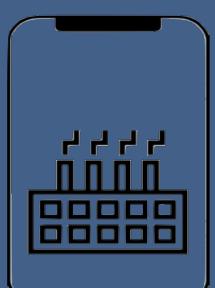
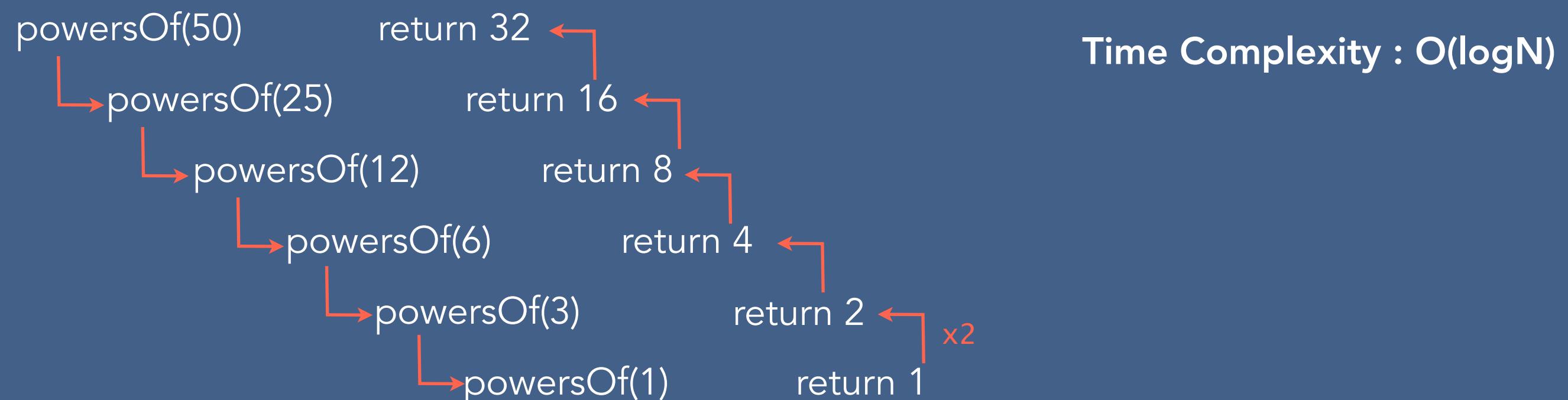
What is the runtime of the below code?

```
static int powersOf2(int n) {  
    if (n<1) {  
        return 0;  
    } else if (n==1) {  
        System.out.println(1);  
        return 1;  
    } else {  
        var prev = powersOf2(n/2);  
        var curr = prev*2;  
        System.out.println(curr);  
        return curr;  
    }  
}
```

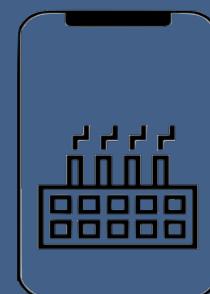
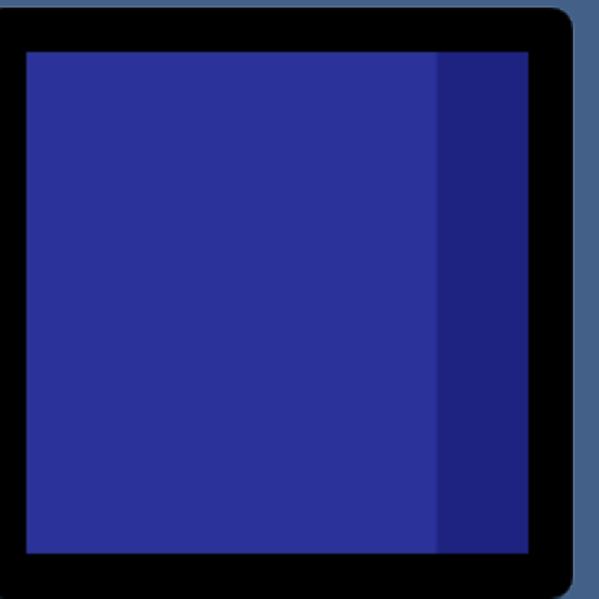
→ O(1)

→ O(1)

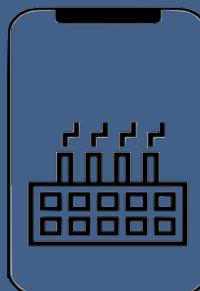
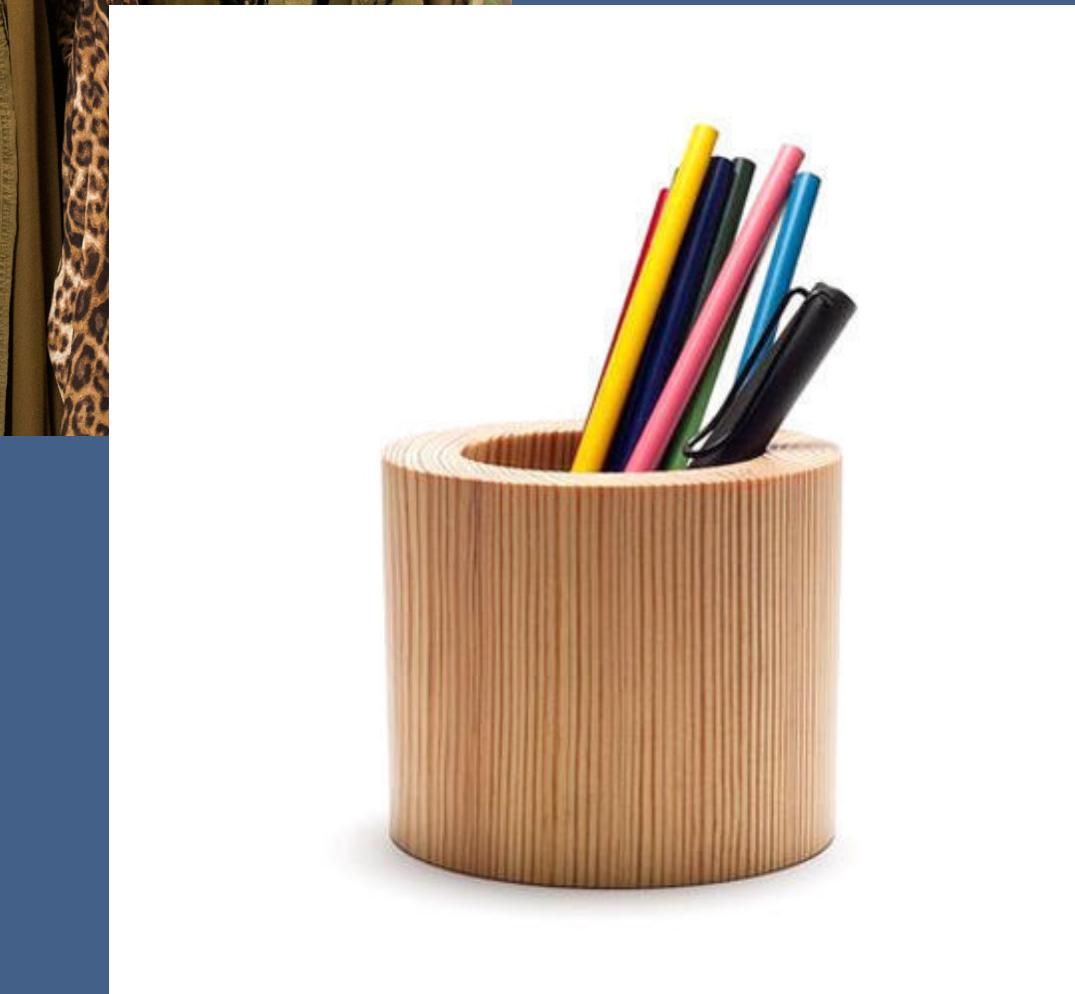
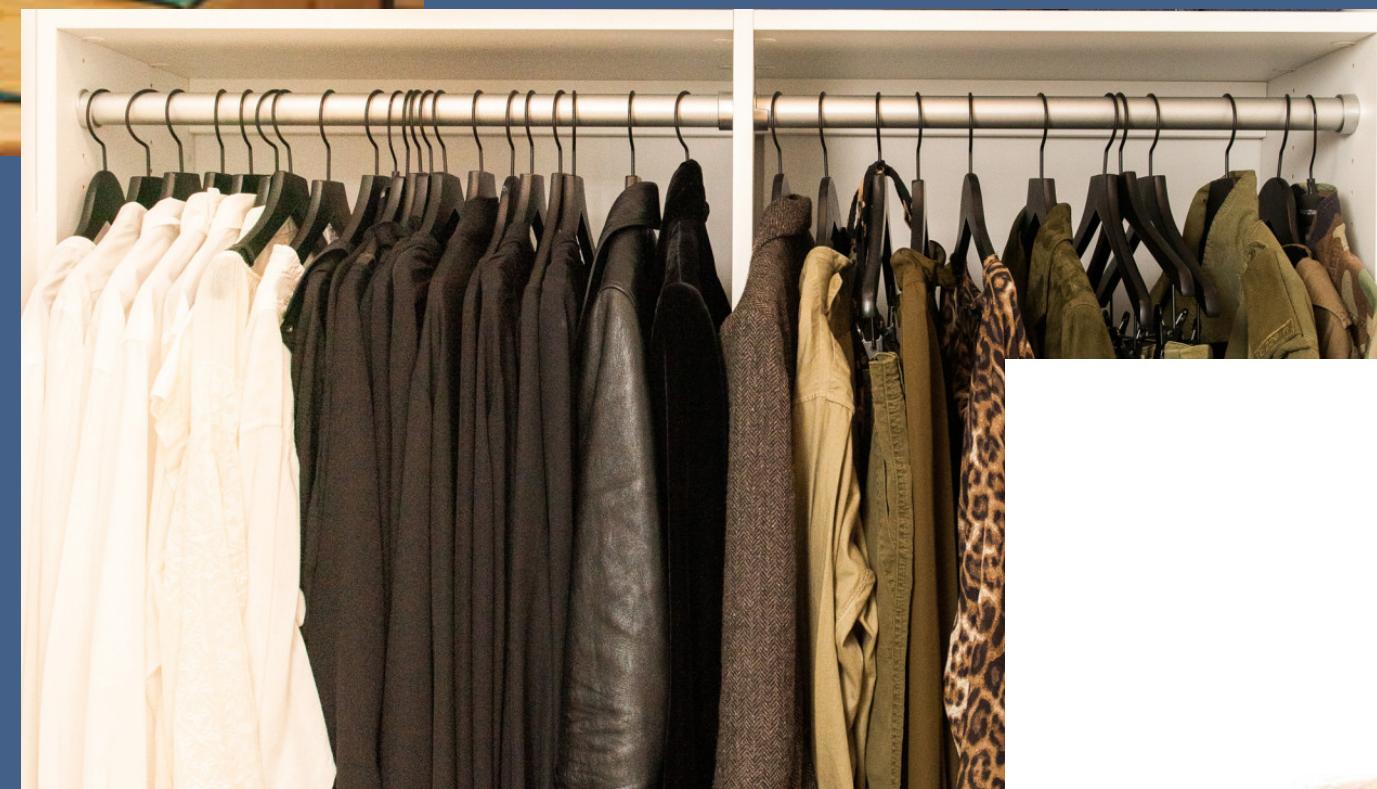
n=50



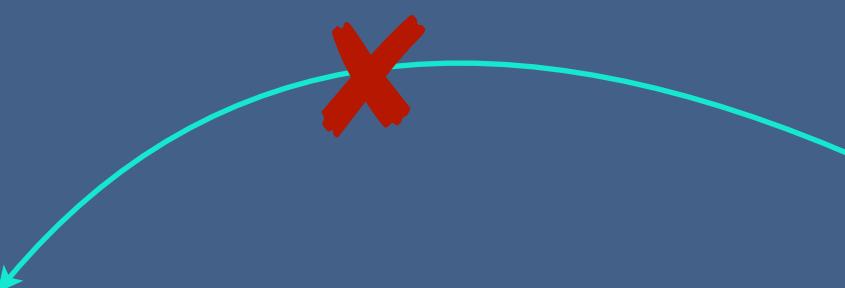
# Arrays



# Arrays



# Arrays

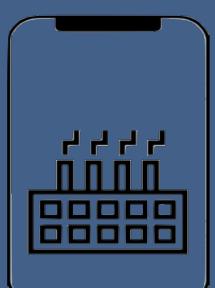


- It is a box of macaroons.
- All macaroons in this box are next to each other
- Each macaroon can be identified uniquely based on their location
- The size of box cannot be changed



"a"

- 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

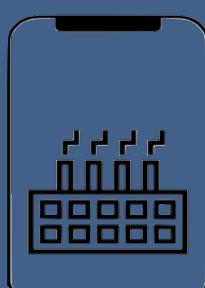
number1

number2

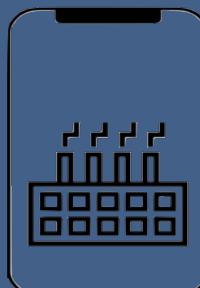
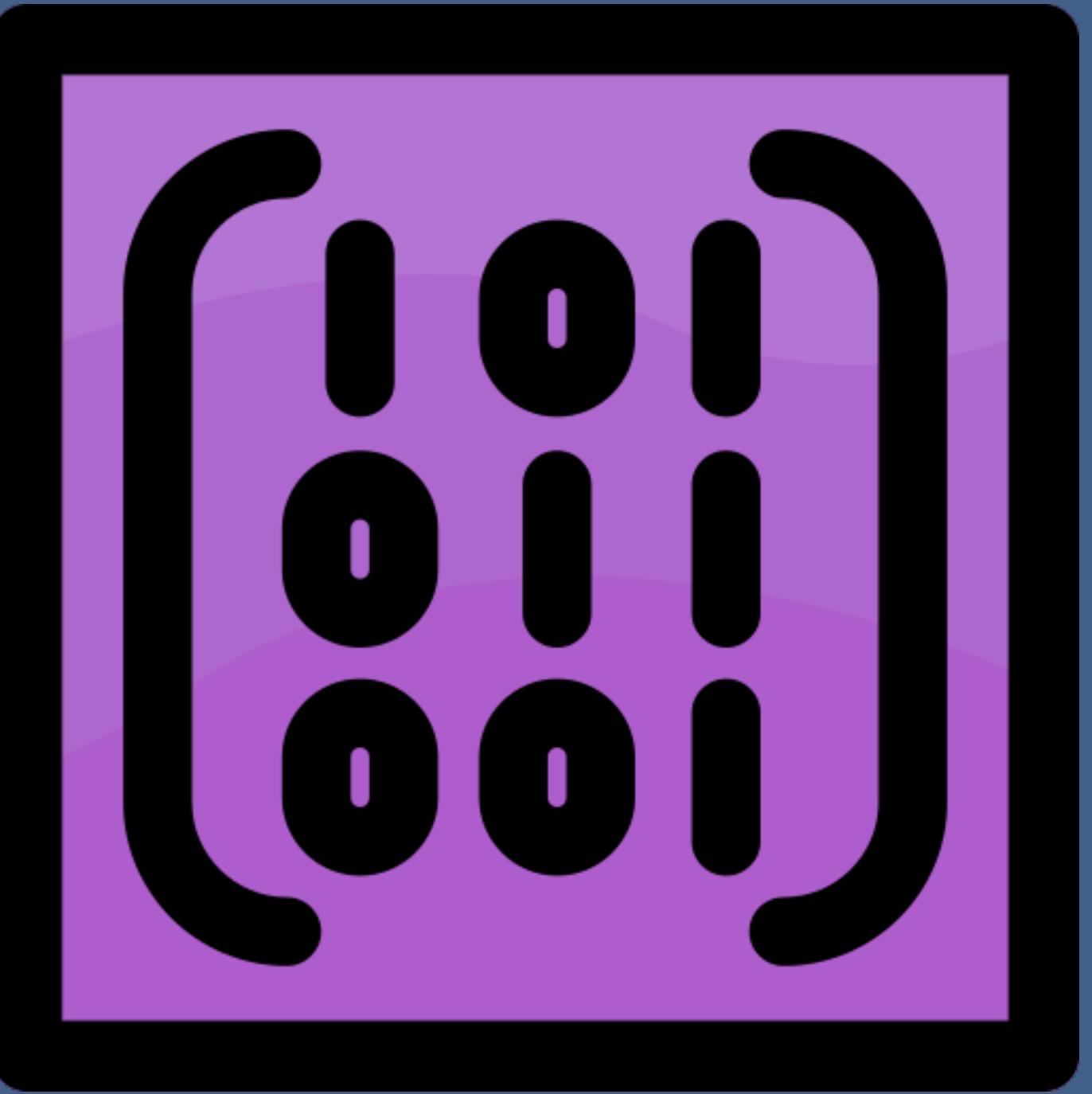
number3

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

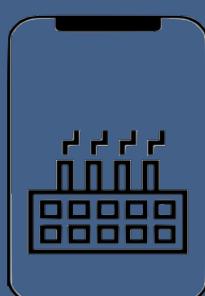
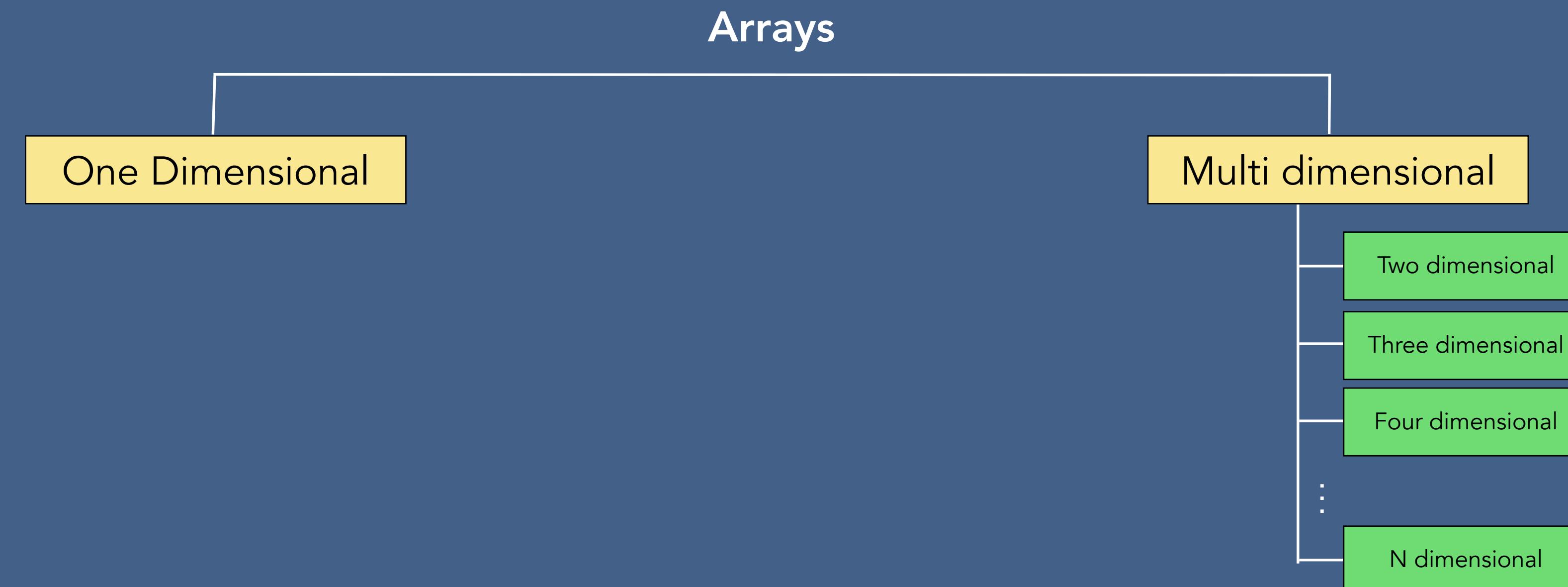
The answer is an **ARRAY**



# Types of Arrays



# Types of Arrays



# Types of Arrays

**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.

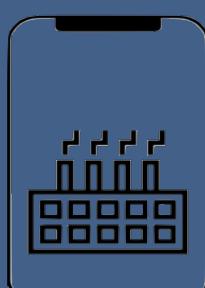
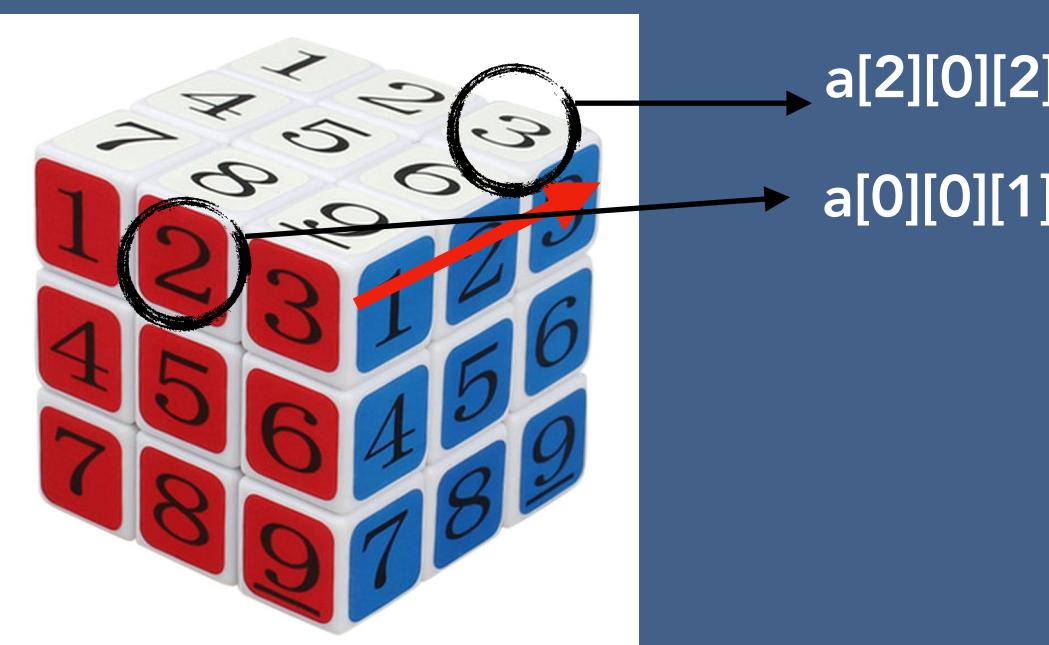
$a[i][j] \rightarrow i \text{ and } j \text{ between } 0 \text{ and } n$

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

Arrows point from  $a[0][4]$  to the value 20 and from  $a[2][5]$  to the value 30.

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

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



# Types of Arrays

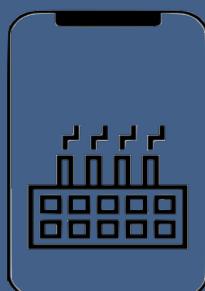
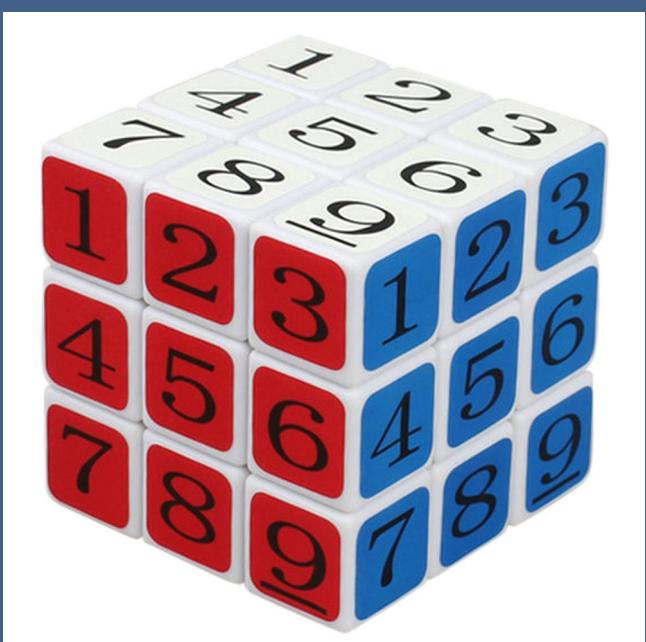
One dimensional array :

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

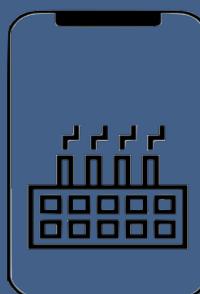
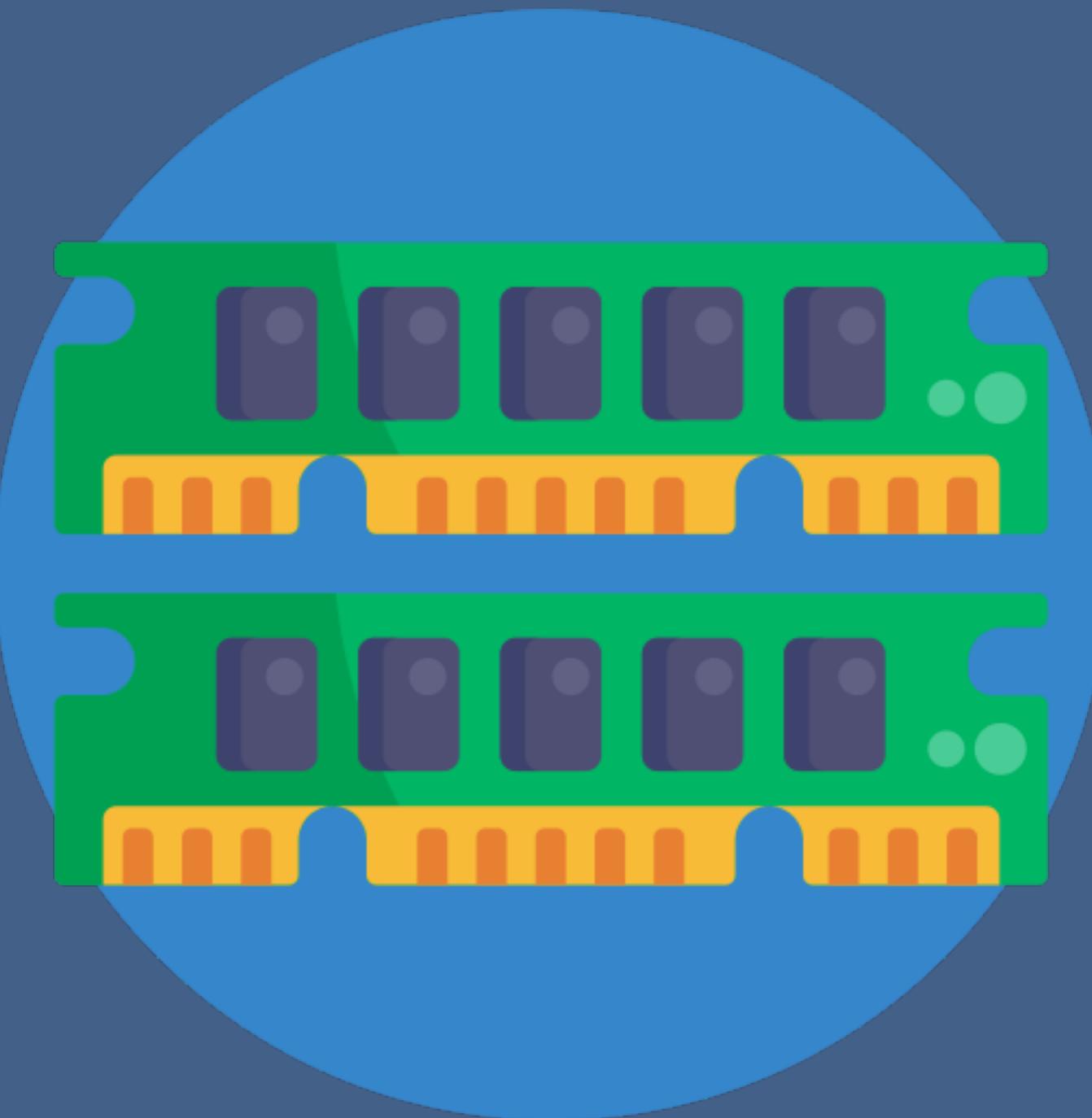
Two dimensional array :

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

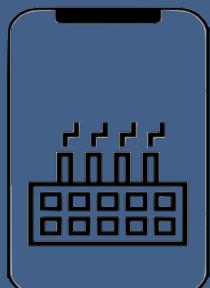
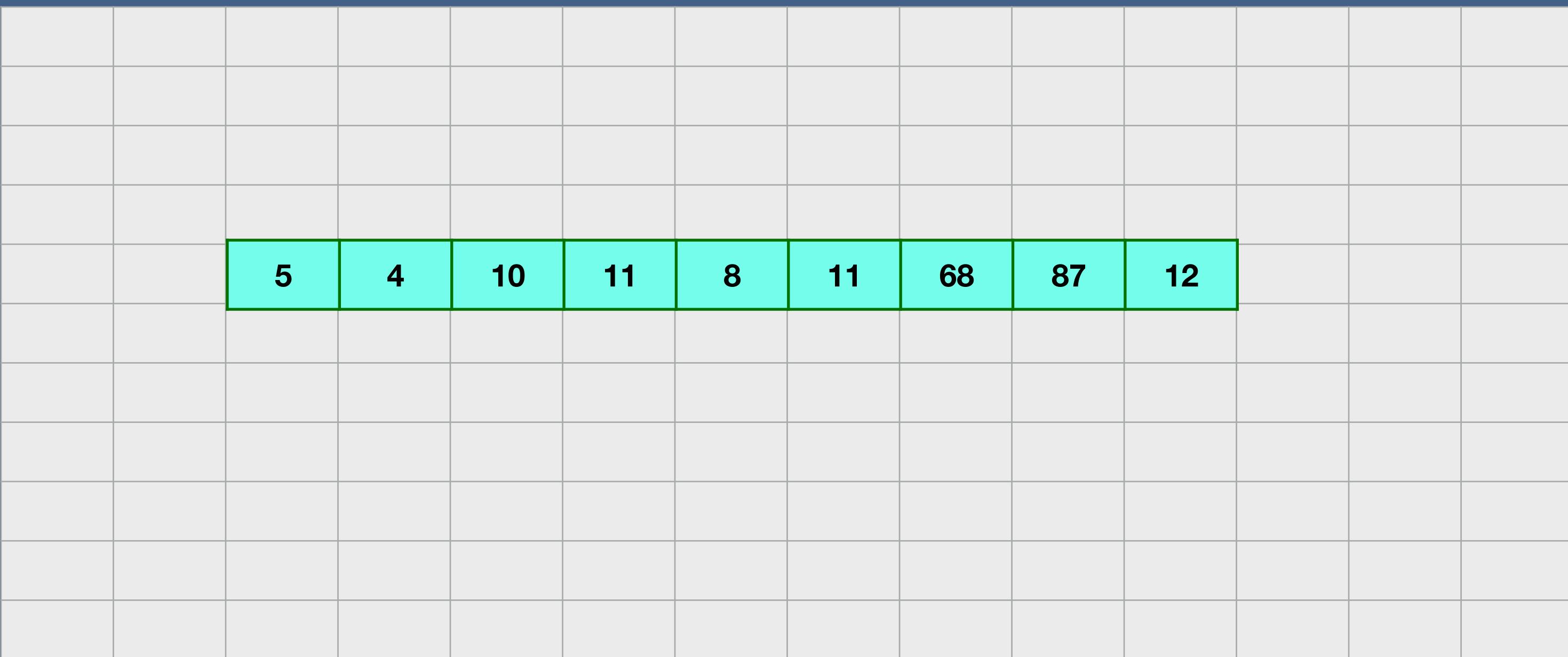


# Arrays in Memory

## One Dimensional

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

## Memory

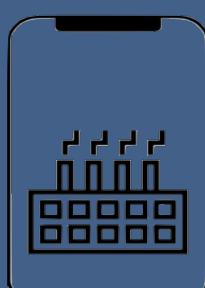
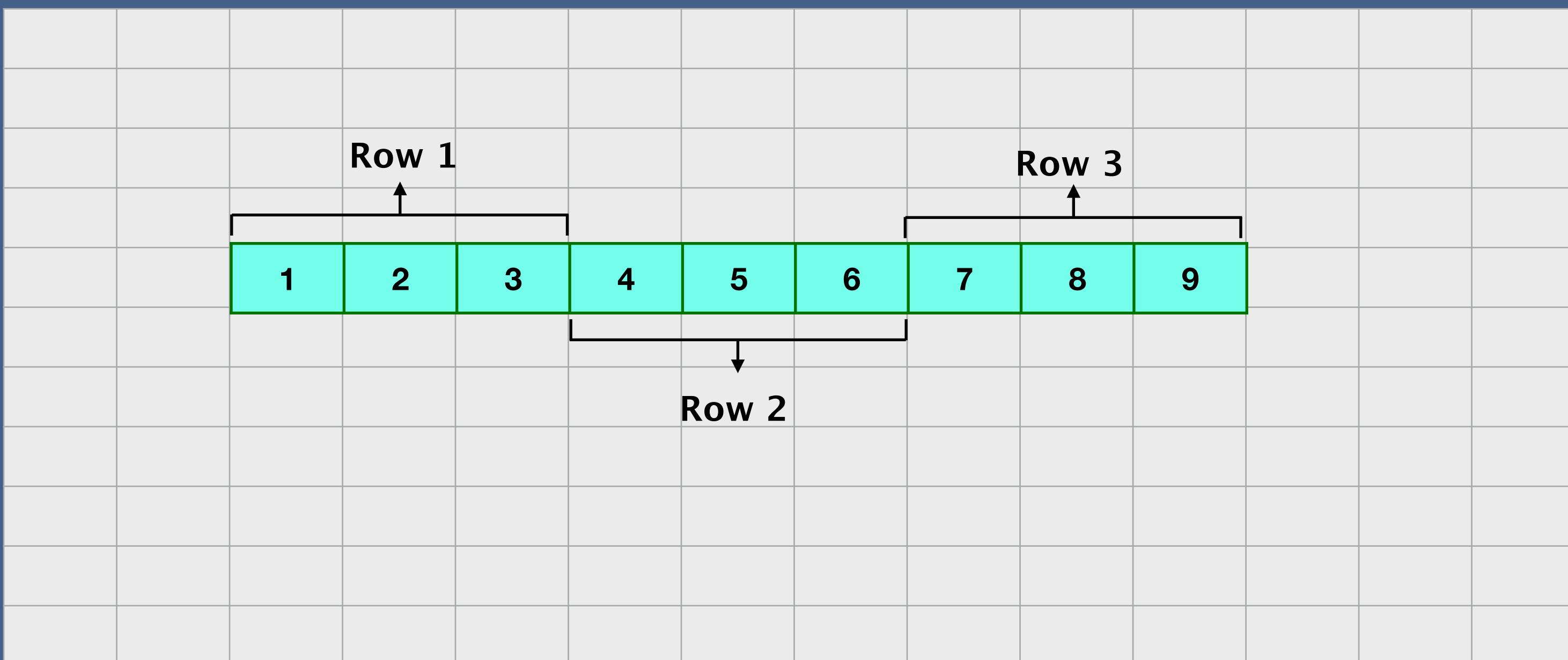


# Arrays in Memory

Two Dimensional array

1	2	3
4	5	6
7	8	9

Memory

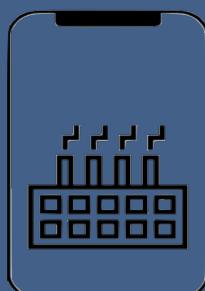
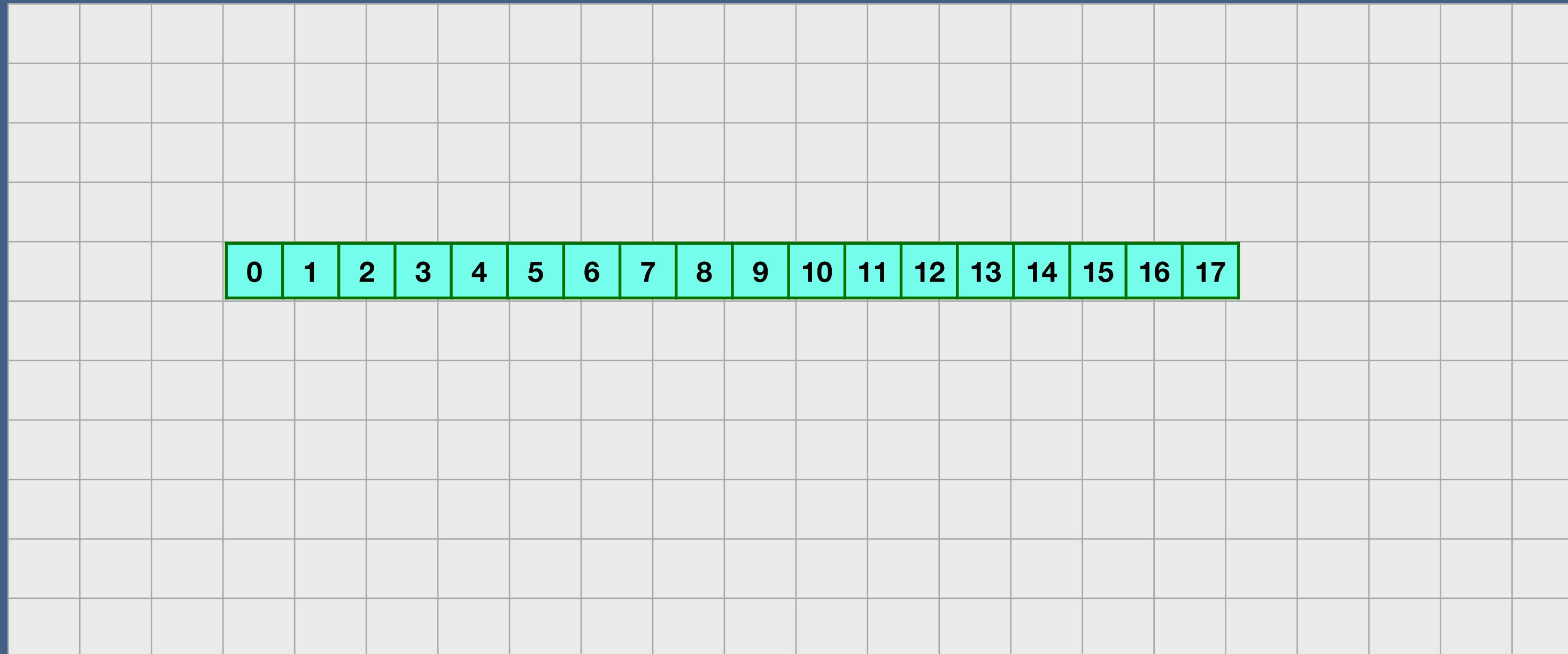


# Arrays in Memory

## Three Dimensional array

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

## Memory

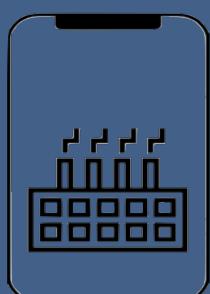
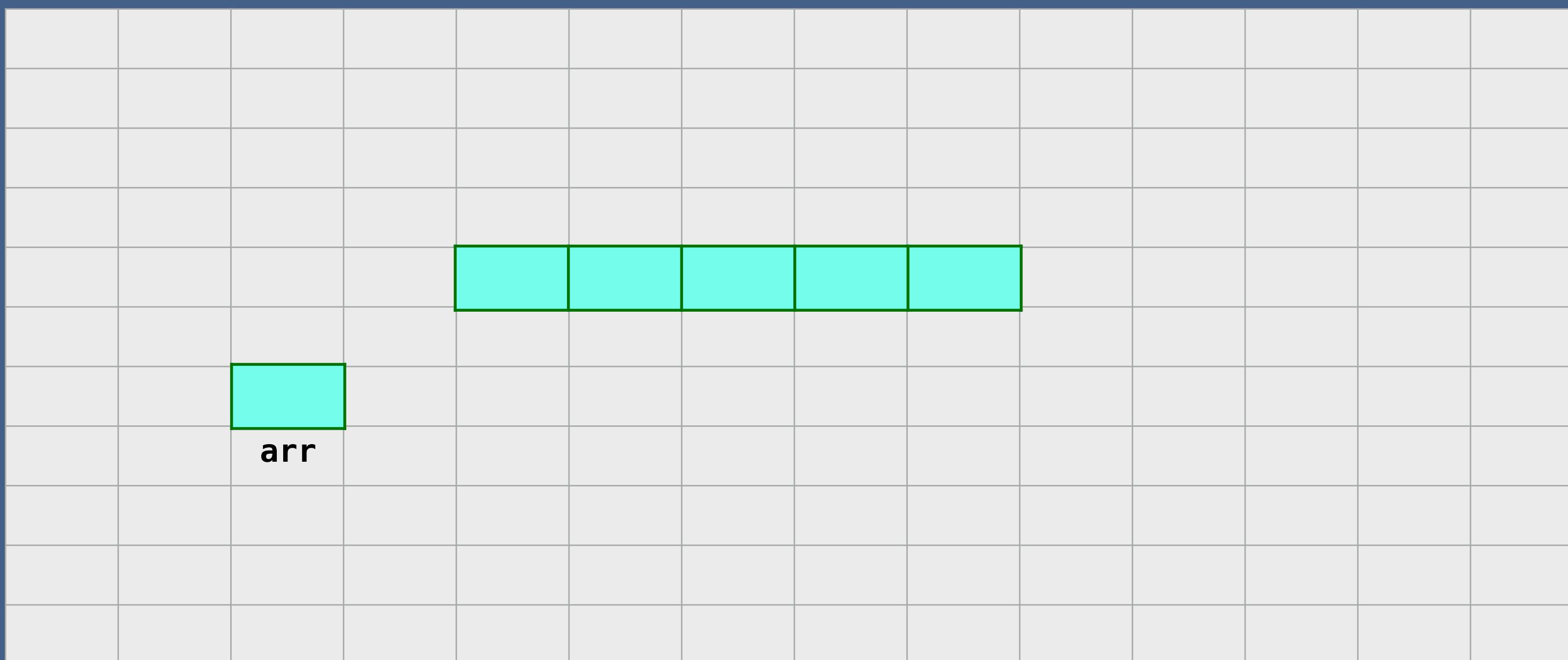


# Creating an Array

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

- Declare - creates a reference to array
- Instantiation of an array - creates an array
- Initialization - assigns values to cells in array

**Memory**

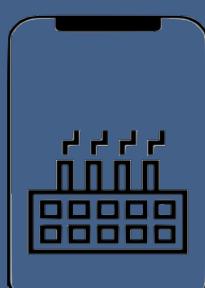
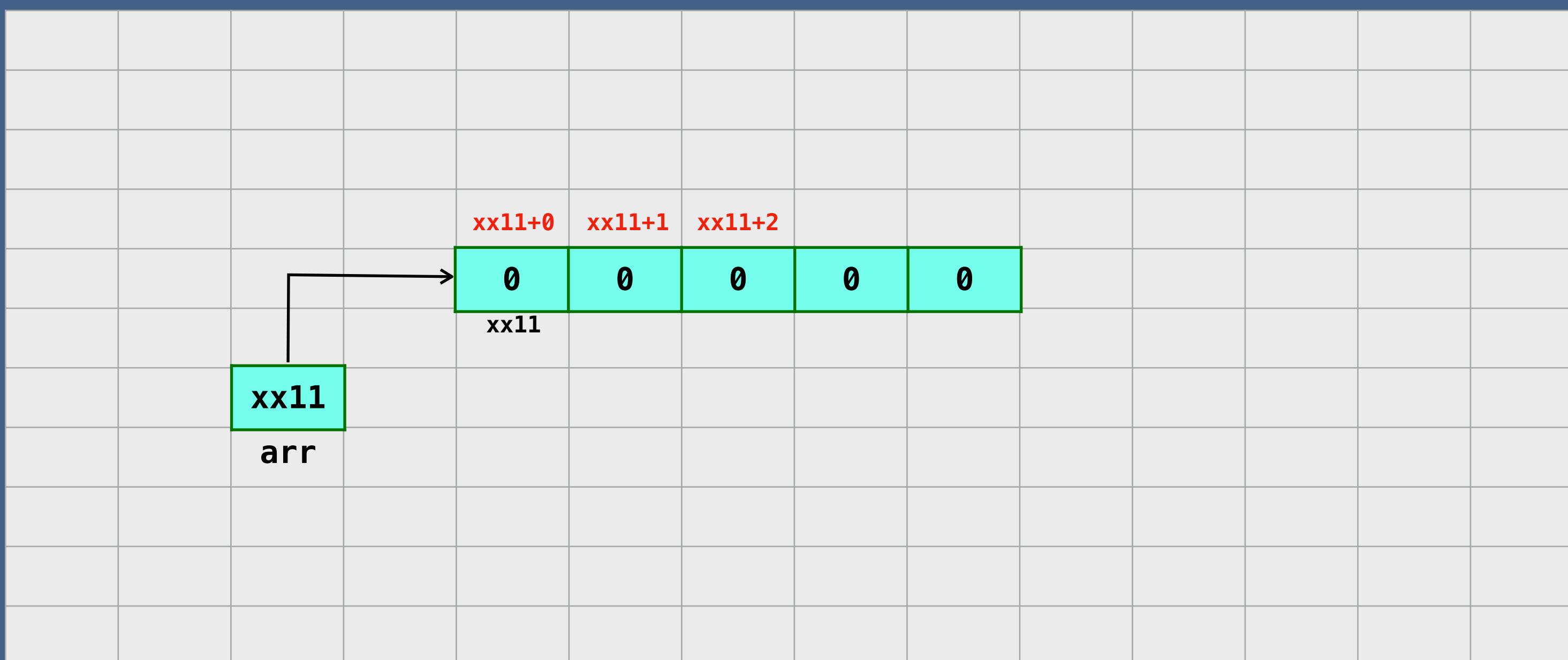


# Creating an Array

When we create an array , we:

- Declare - creates a reference to array
- Instantiation of an array - creates an array
- Initialization - assigns values to cells in array

Memory

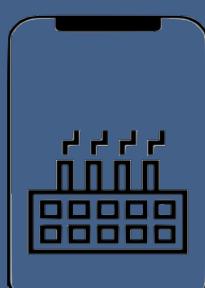
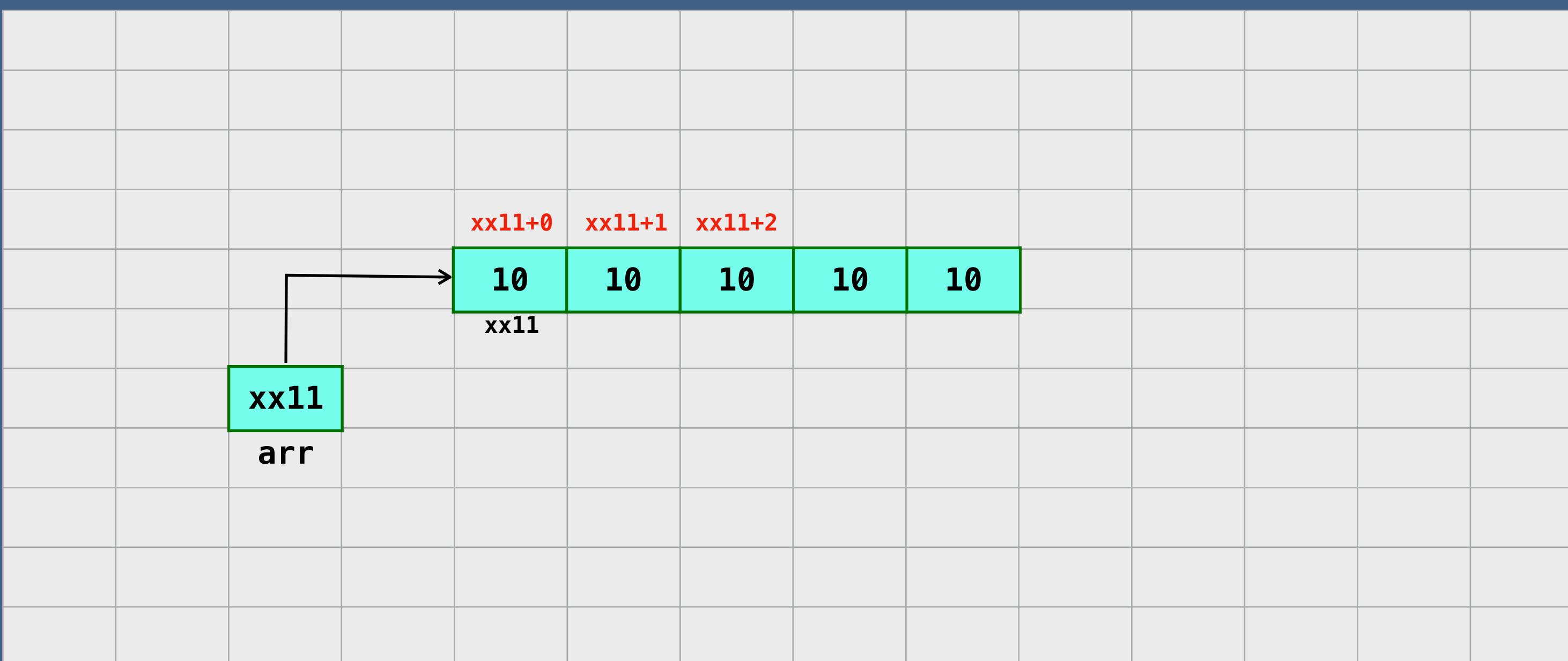


# Creating an Array

When we create an array , we:

- Declare - creates a reference to array
- Instantiation of an array - creates an array
- Initialization - assigns values to cells in array

Memory



# Creating an Array

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

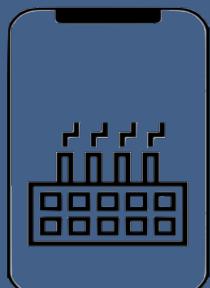
- Declare - creates a reference to array
- Instantiation of an array - creates an array
- Initialization - assigns values to cells in array

```
dataType[] arr
```

```
arr = new dataType[]
```

```
arr[0] = 1
```

```
arr[1] = 2
```



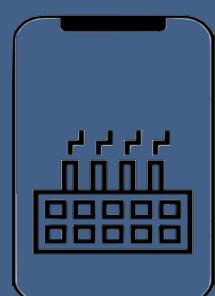
# Insertion in Array

myArray =

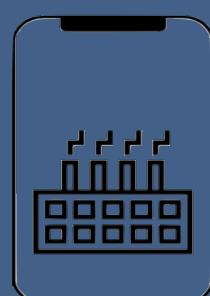
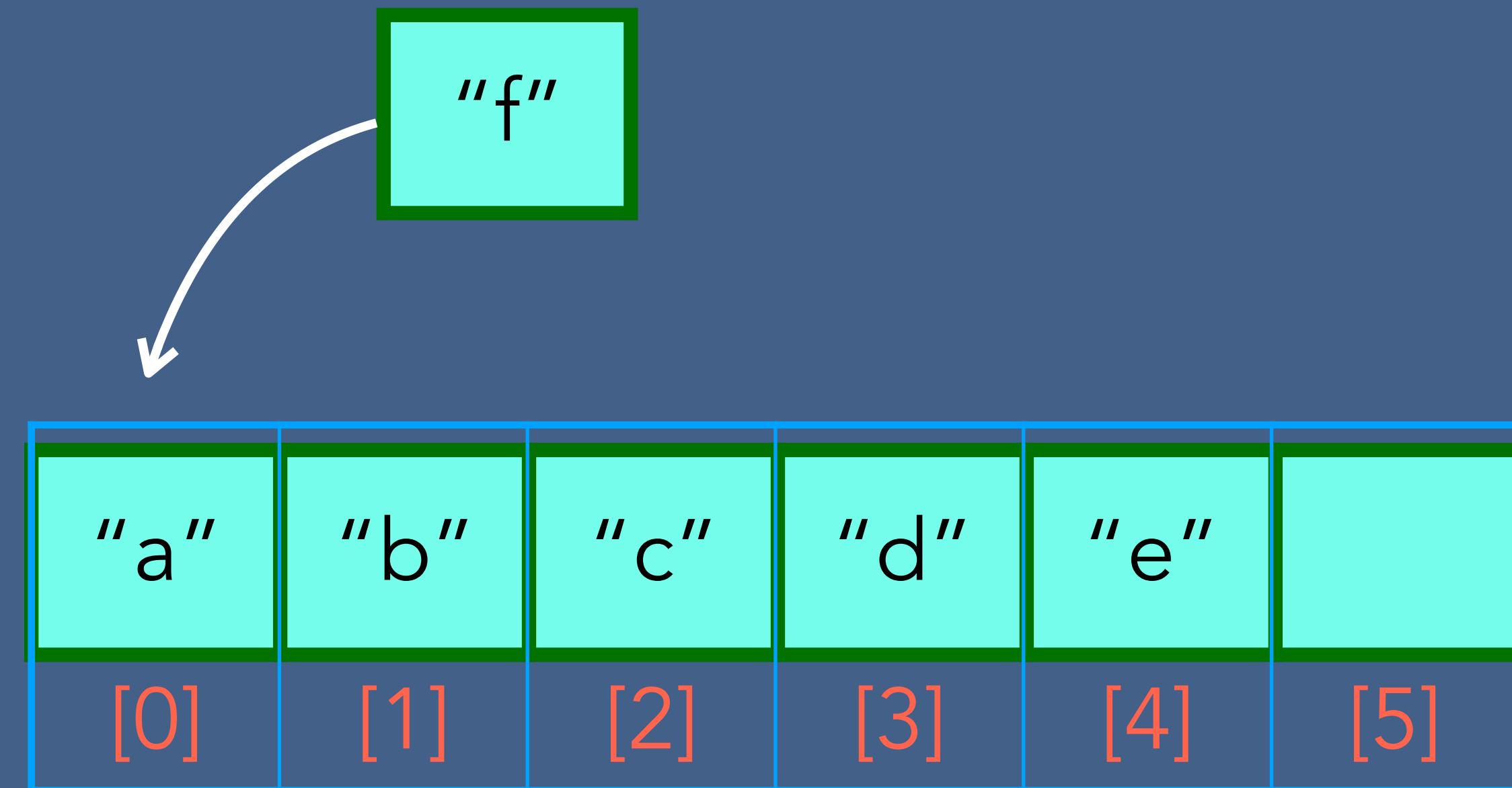
“a”	“b”	“c”	“d”		“f”
[ 0 ]	[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]

myArray[3] = “d”

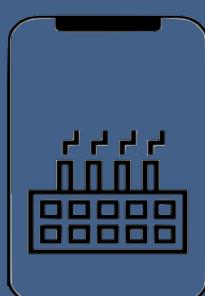
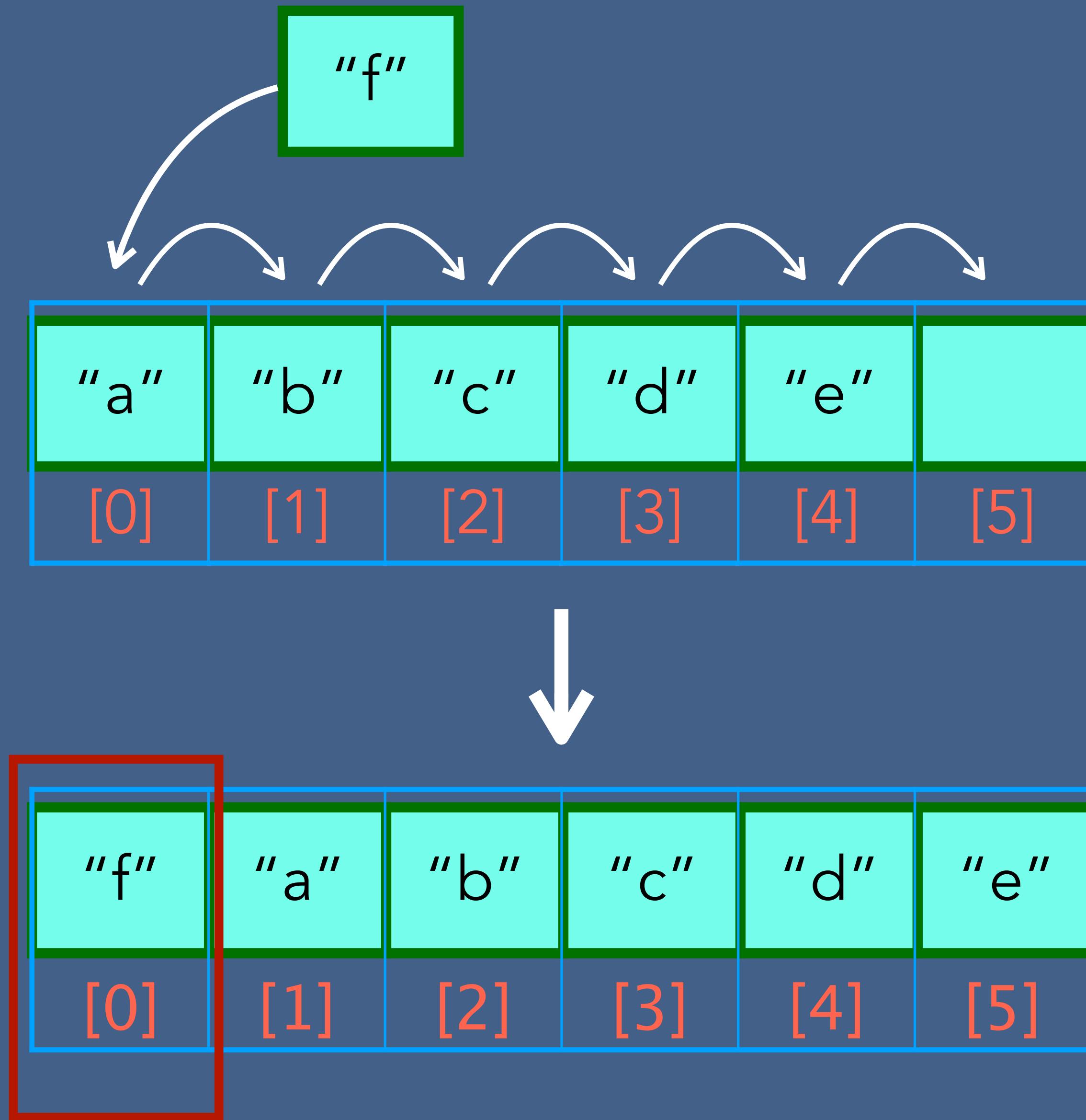
myArray[5] = “f”



# Insertion in Array



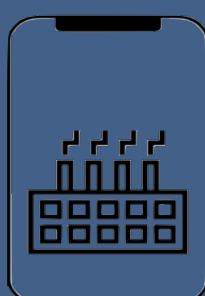
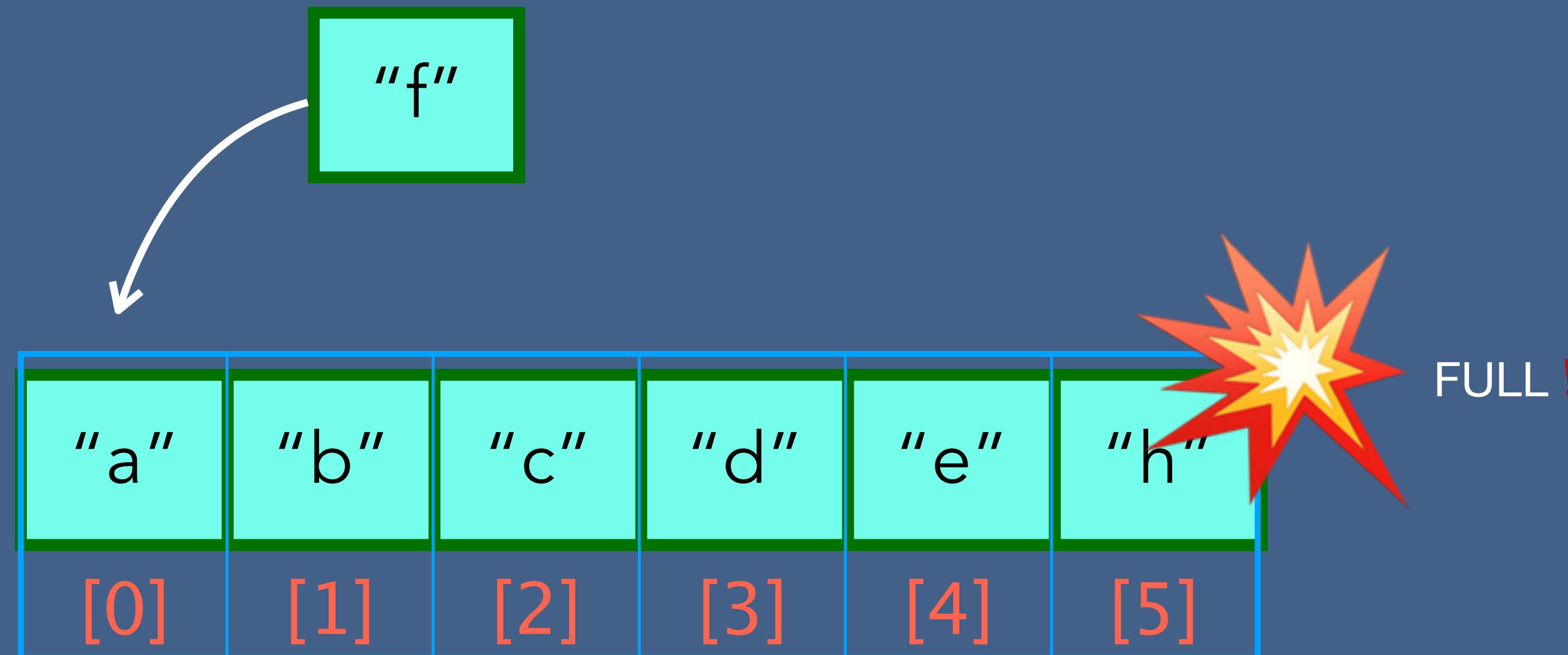
# Insertion in Array



# Insertion in Array

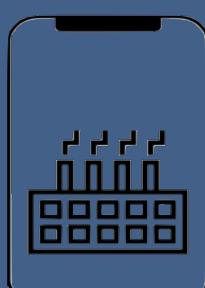
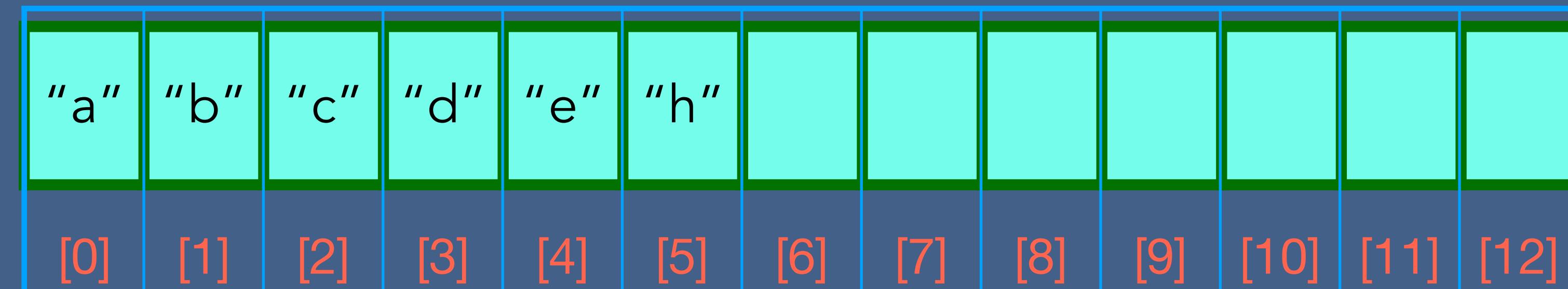
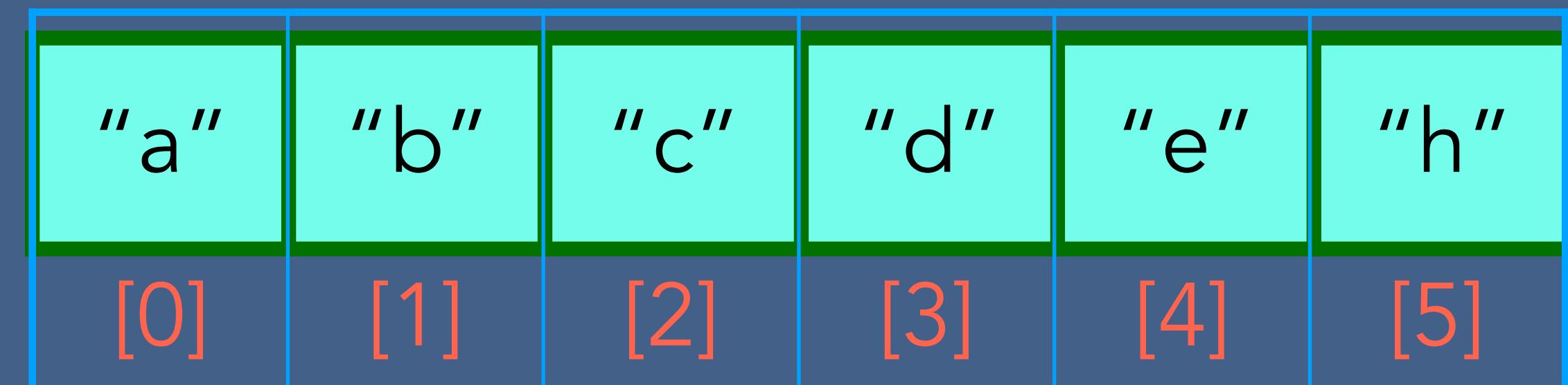


Wait A minute! What Happens if the Array is Full?

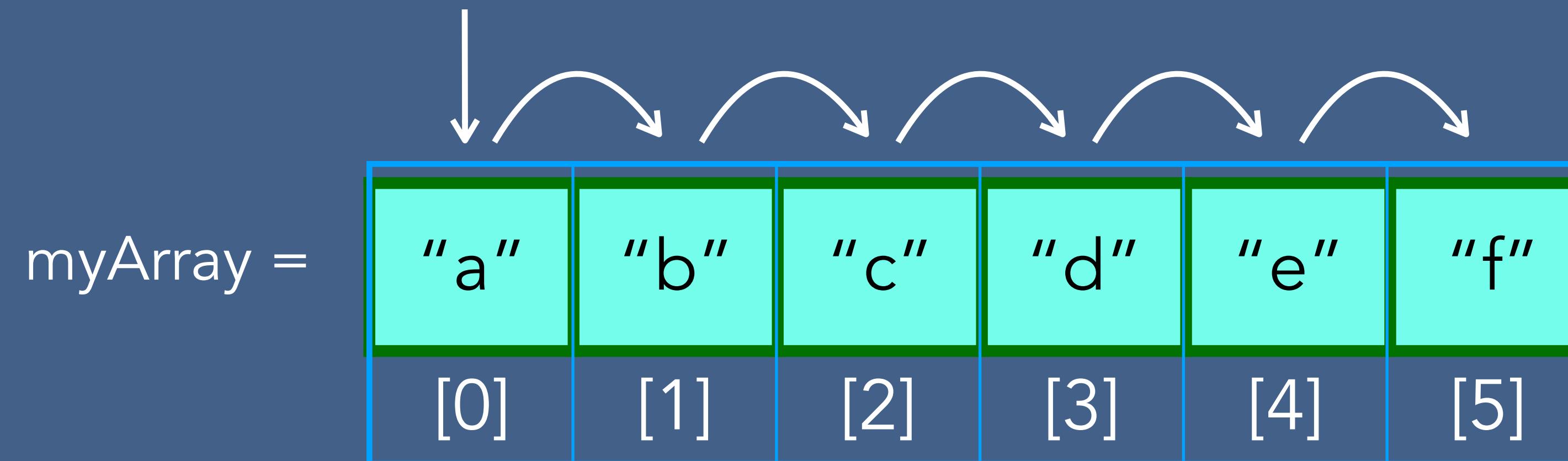


# Insertion in Array

Insertion , when an array is full.



# Array Traversal



myArray[0] = "a"

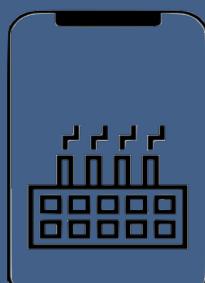
myArray[1] = "b"

myArray[2] = "c"

myArray[3] = "d"

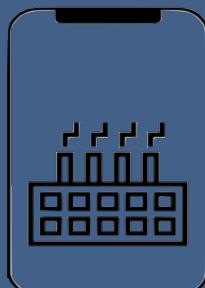
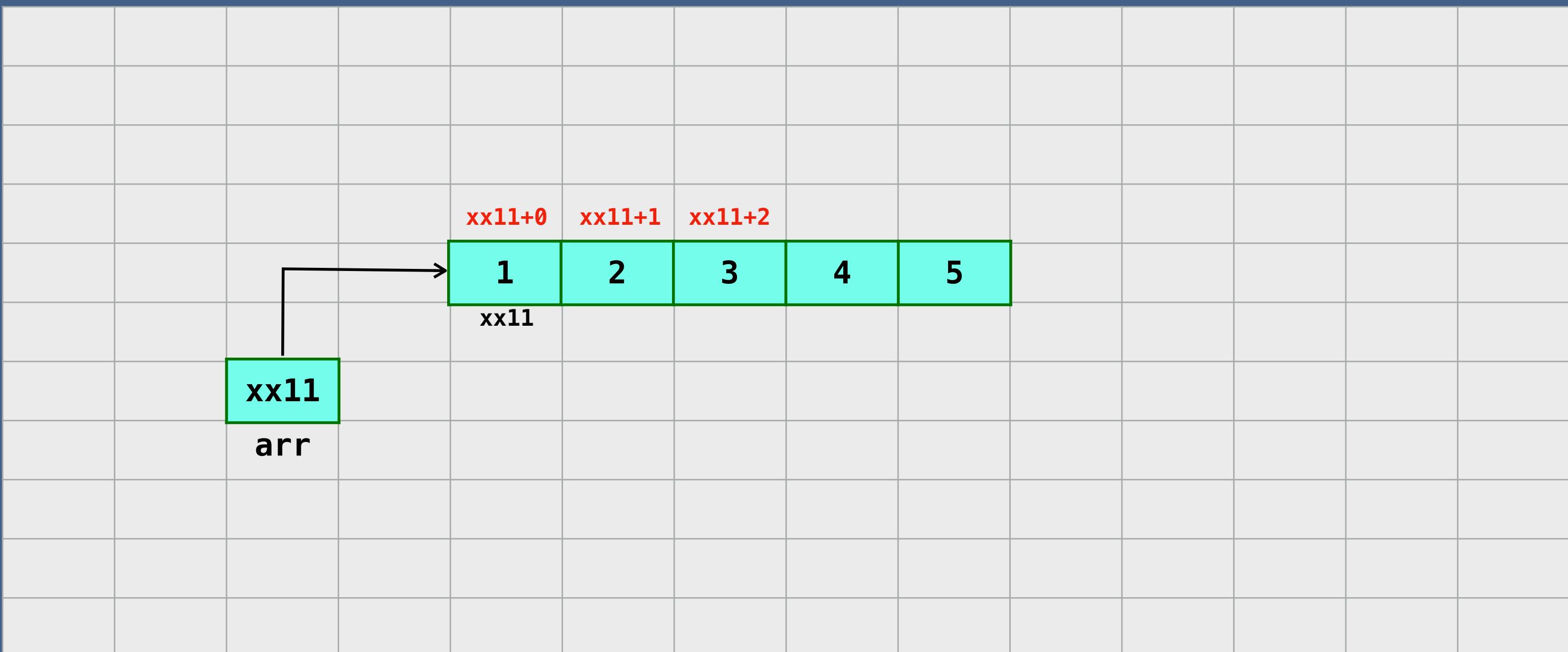
myArray[4] = "e"

myArray[5] = "f"

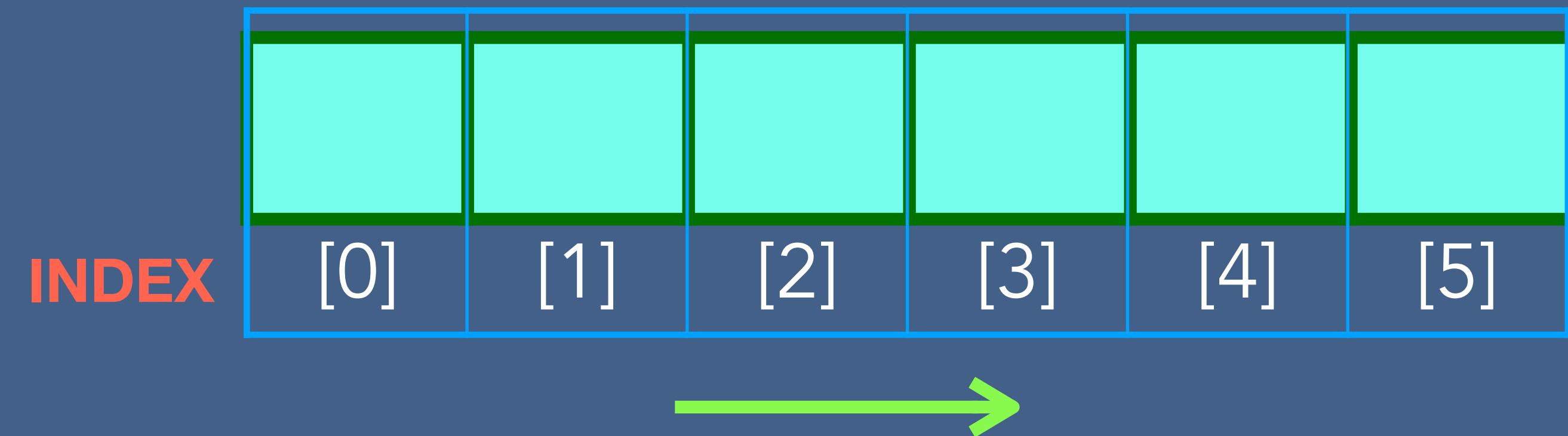


# Accessing Array Element

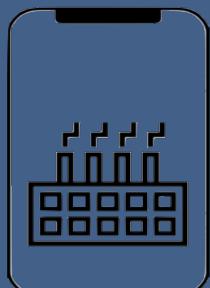
## Memory



# Accessing Array Element



<arrayName>[index]



# Accessing Array Element

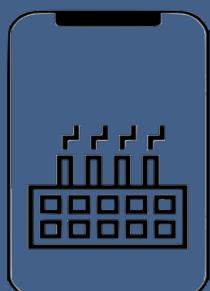
For example:

myArray =

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

myArray[0] = “a”

myArray[3] = “d”

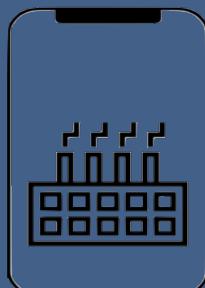
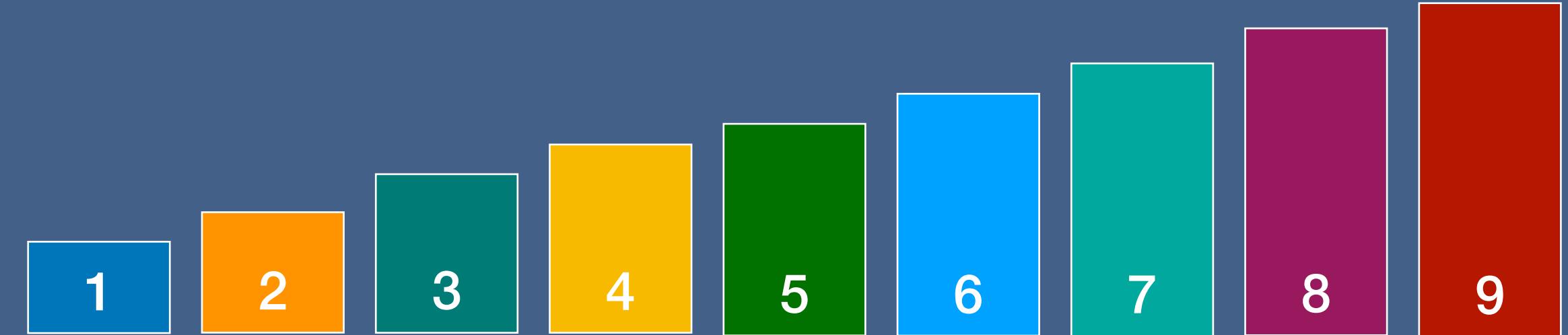


# Finding Array Element

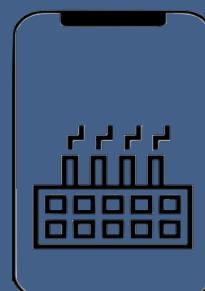
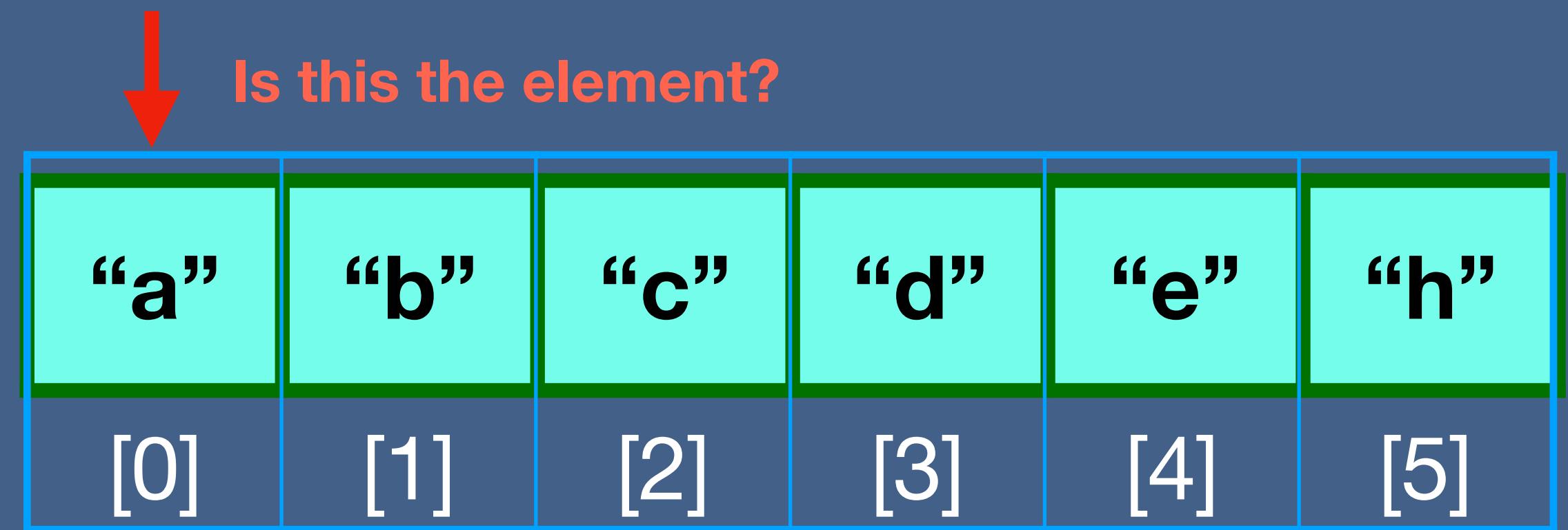
myArray =

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

myArray2 =



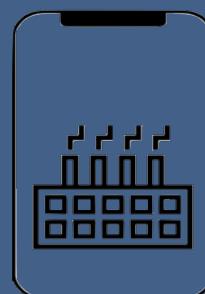
# Finding Array Element



# Deleting Array Element

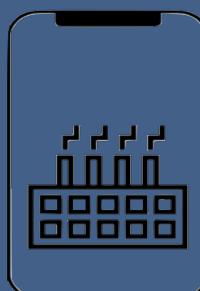
Integer.MIN\_VALUE

10	20	-2 <sup>31</sup>	40	-2 <sup>31</sup>	-2 <sup>31</sup>
[0]	[1]	[2]	[3]	[4]	[5]



# Time and Space Complexity of 1D Arrays

Operation	Time complexity	Space complexity
Creating an empty array	$O(1)$	$O(n)$
Inserting a value in an array	$O(1)$	$O(1)$
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	$O(1)$	$O(1)$



# Find Number of Days Above Average Temperature

How many day's temperature?

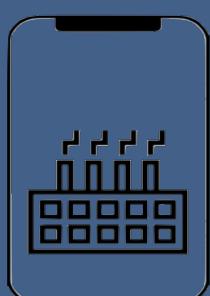
2

Day 1's high temp: 1

Day 2's high temp: 2

## Output

Average = 1.5  
1 day(s) above average



# 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

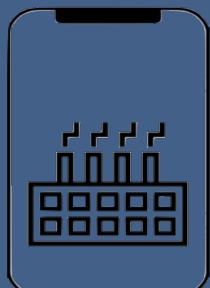
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[0]	1	33	55	91	20	51	62	74	13
[1]	5	4	10	11	8	11	68	87	12
[2]	24	50	37	40	48	30	59	81	93

Day 1 - 11, 15, 10, 6

Day 2 - 10, 14, 11, 5

Day 3 - 12, 17, 12, 8

Day 4 - 15, 18, 14, 9



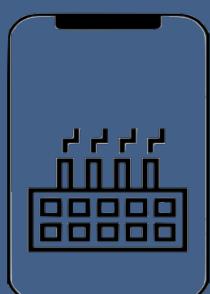
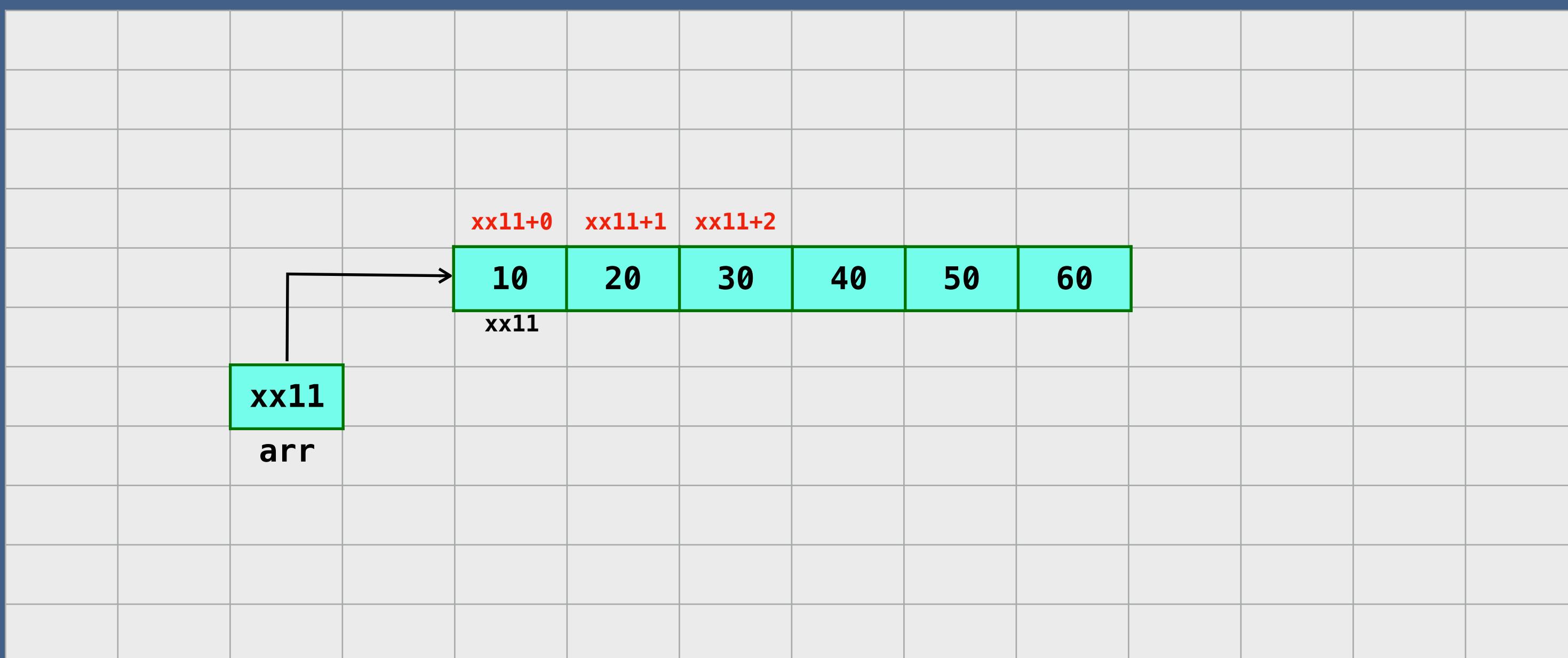
# Two Dimensional Array

When we create an array , we:

- Declare - creates a reference to array
- Instantiation of an array - creates an array
- Initialization - assigns values to cells in array

10	20	30
40	50	60

Memory



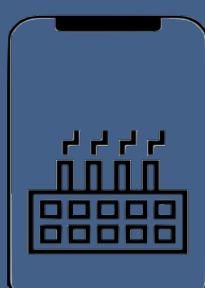
# Insertion - Two Dimensional Array

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[0]	50								
[1]									
[2]						30			

arrayName[2][5] = 30

arrayName[0][0] = 50

arrayName[0][0] = 60 → Occupied



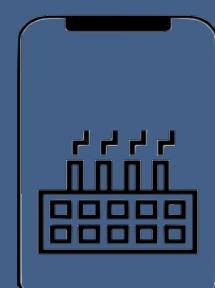
# Access an element of Two Dimensional Array

$a[i][j]$  —> i is row index and j is column index

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

$a[2][5]$

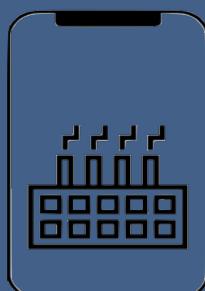
$a[0][4]$



# 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



# Searching Two Dimensional Array

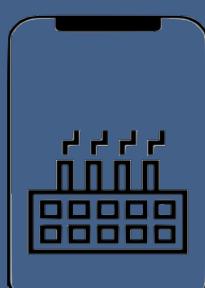


Is this the element?

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

Searching for 44

The element is found at [1][2]

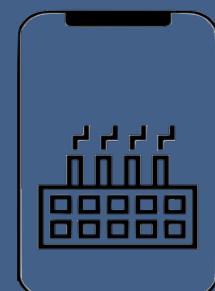


# Deleting Array Element in 2D Array

	[0]	[1]	[2]	[3]
[0]	1	33	55	91
[1]	5	-2 <sup>31</sup>	10	11
[2]	24	50	37	-2 <sup>31</sup>

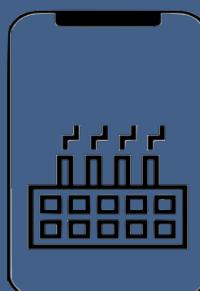
```
arrayName[2][3] = Integer.MIN_VALUE
```

```
arrayName[1][1] = Integer.MIN_VALUE
```



# Time and Space Complexity of 2D Array

Operation	Time complexity	Space complexity
Creating an empty array	$O(1)$	$O(mn)$
Inserting a value in an array	$O(1)$	$O(1)$
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	$O(1)$	$O(1)$



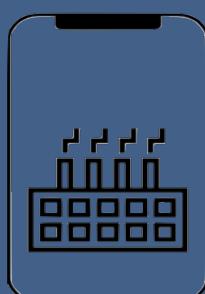
# When to use/avoid Arrays

## When to use

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

## When to avoid

- Same data type elements
- Reserve memory



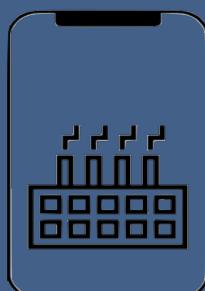
# When to use/avoid Arrays

## When to use

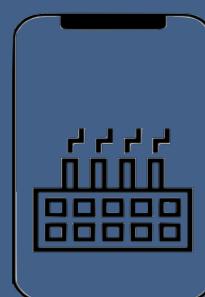
- To store multiple variables of same data type
- Random access

## When to avoid

- Same data type elements
- Reserve memory



# Array Project



# Array Project

Find Number of Days Above Average Temperature

How many day's temperature?

2

Day 1's high temp:

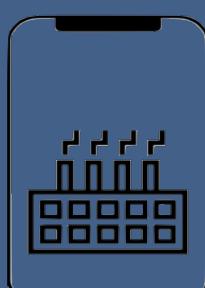
1

Day 2's high temp:

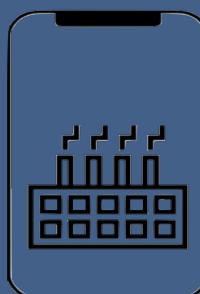
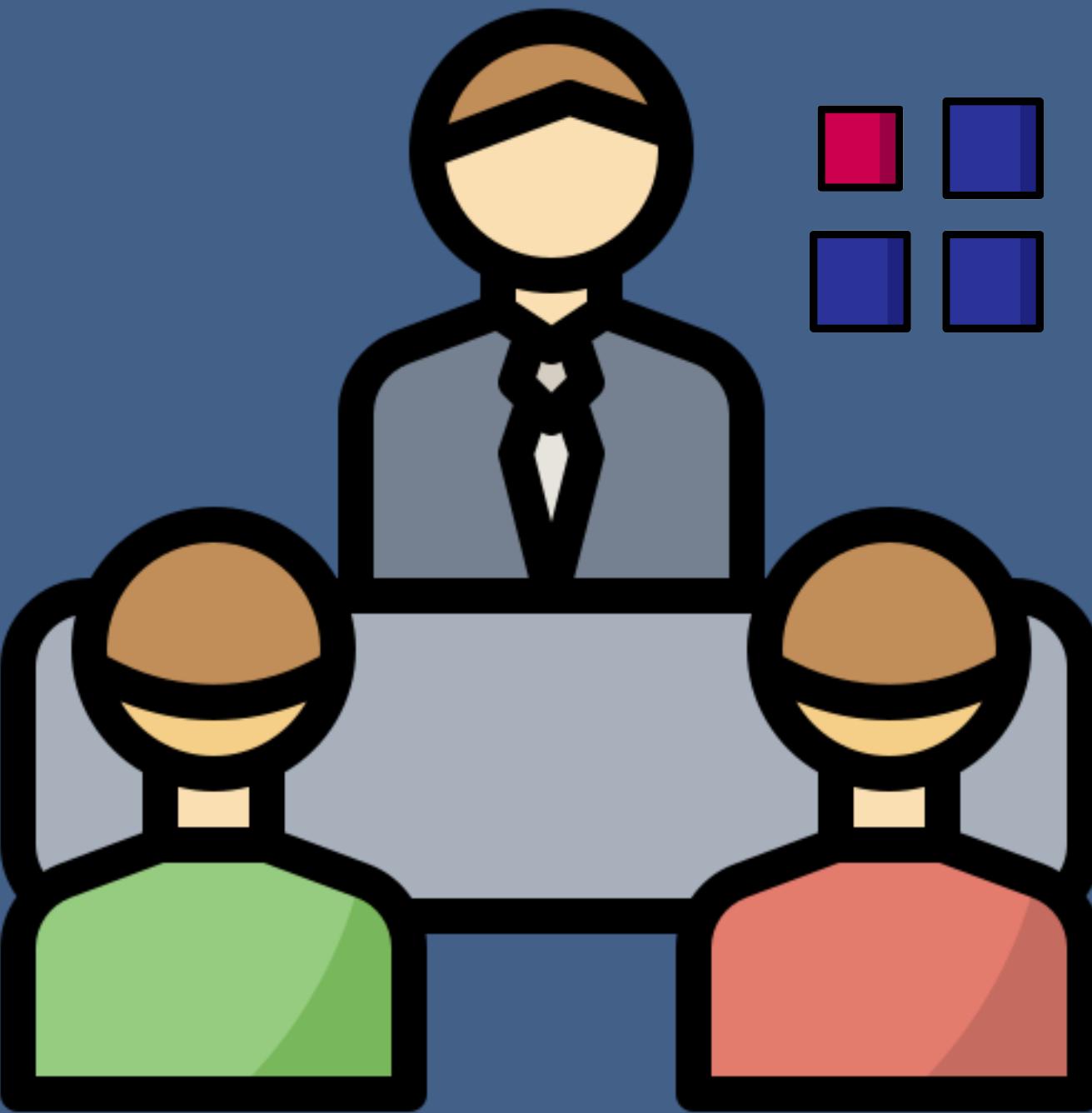
2

## Output

Average = 1.5  
1 day(s) above average



# Array Interview Questions



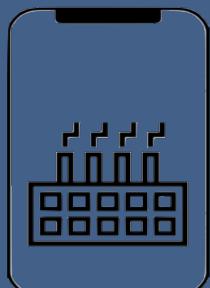
# Missing Number

Find the missing number in an integer array of 1 to 100

1,2,3,4,5,6...50,51,52..100

$$\begin{array}{l} 1,2,3,4,5,6,7,8,9,10 = 55 \\ \longrightarrow \qquad \qquad \qquad 7 \\ 1,2,3,4,5,6,8,9,10 = 48 \end{array}$$

$$1,2,3,4,5,6...n = n(n+1)/2$$



# Pairs / Two Sum

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]

**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:**

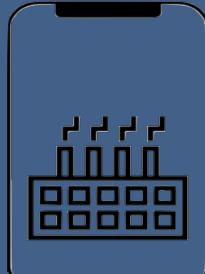
```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Output: Because nums[0] + nums[1] == 9, we return [0, 1].
```

**Example 2:**

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

**Example 3:**

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

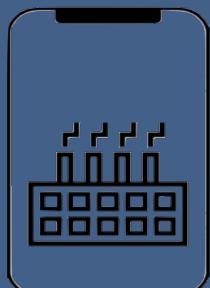


# Pairs / Two Sum

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]

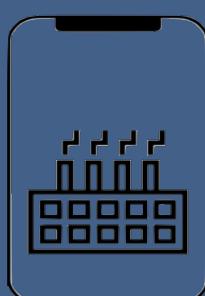
- Does array contain only positive or negative numbers?
- What if the same pair repeats twice, should we print it every time?
- If the reverse of the pair is acceptable e.g. can we print both (4,1) and (1,4) if the given sum is 5.
- Do we need to print only distinct pairs? does (3, 3) is a valid pair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



# Search for a Value

Write a program to check if an array contains a number in Java

Searching for 40 —————> Found at the index of 4

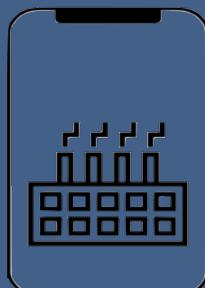


# Max Product of Two Integers

Write a program to find maximum product of two integers in the array where all elements are positive

maxProduct = 0

20	10	30	50	40	60
[0]	[1]	[2]	[3]	[4]	[5]

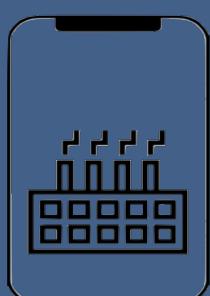


# Is Unique

Write a program to check if an array is unique or not.

isUnique → True

20	10	30	50	40	60
[0]	[1]	[2]	[3]	[4]	[5]



# Permutation

You are given two integer arrays. Write a program to check if they are permutation of each other.

Permutation  $\longrightarrow$  True

2	1	3	5	4	6
---	---	---	---	---	---

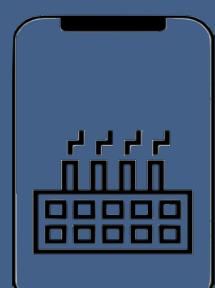
1	3	2	4	6	5
---	---	---	---	---	---

sum1 = 21

sum2 = 21

mul1 = 720

mul2 = 720

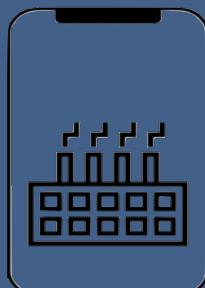


# Rotate Matrix

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

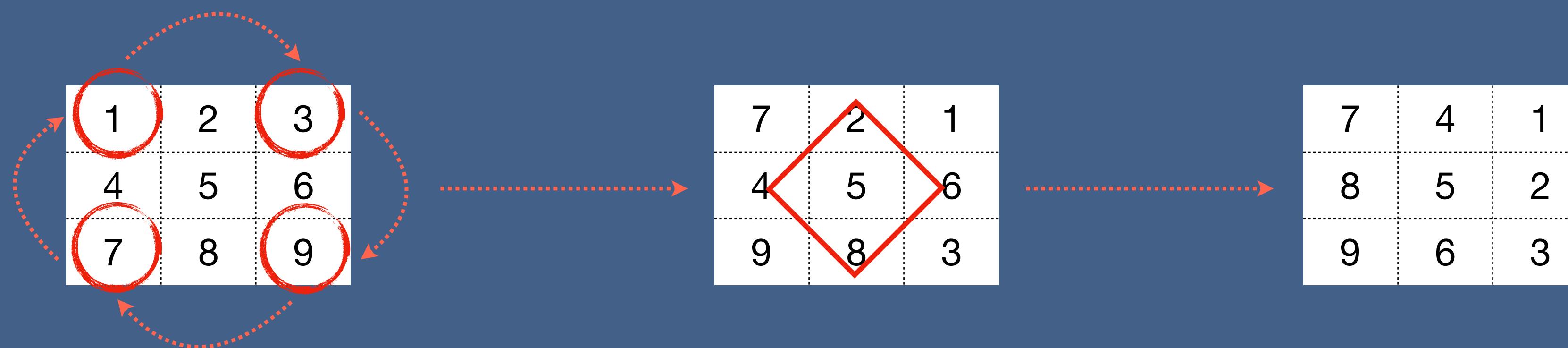
1	2	3
4	5	6
7	8	9

Rotate 90 degrees 

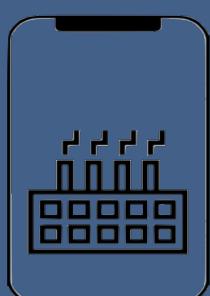


# Rotate Matrix

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
```



# Rotate Matrix

LeetCode Day 26 Explore Problems Mock Contest Discuss Store

Description Solution | Discuss (999+) | Submissions

## 48. Rotate Image

Medium 4008 295 Add to List Share

You are given an  $n \times n$  2D matrix representing an image, rotate the image by 90 degrees (clockwise).

You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

**Example 1:**

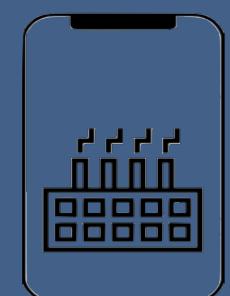
1	2	3
4	5	6
7	8	9

→

7	4	1
8	5	2
9	6	3

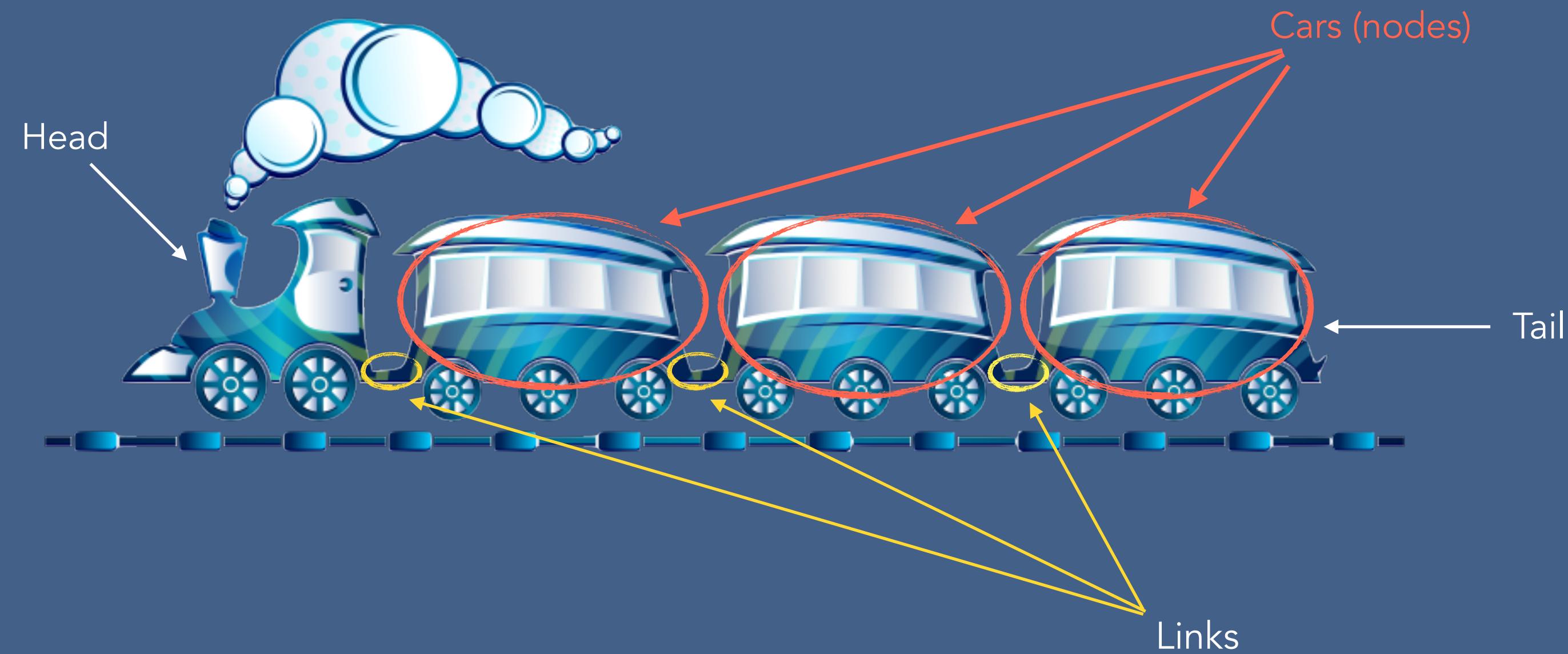
**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]  
**Output:** [[7,4,1],[8,5,2],[9,6,3]]

# Linked List

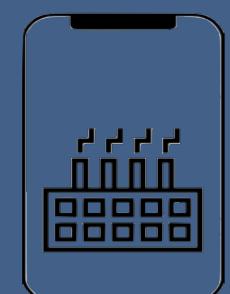


# What is a Linked List?

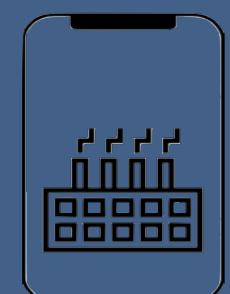
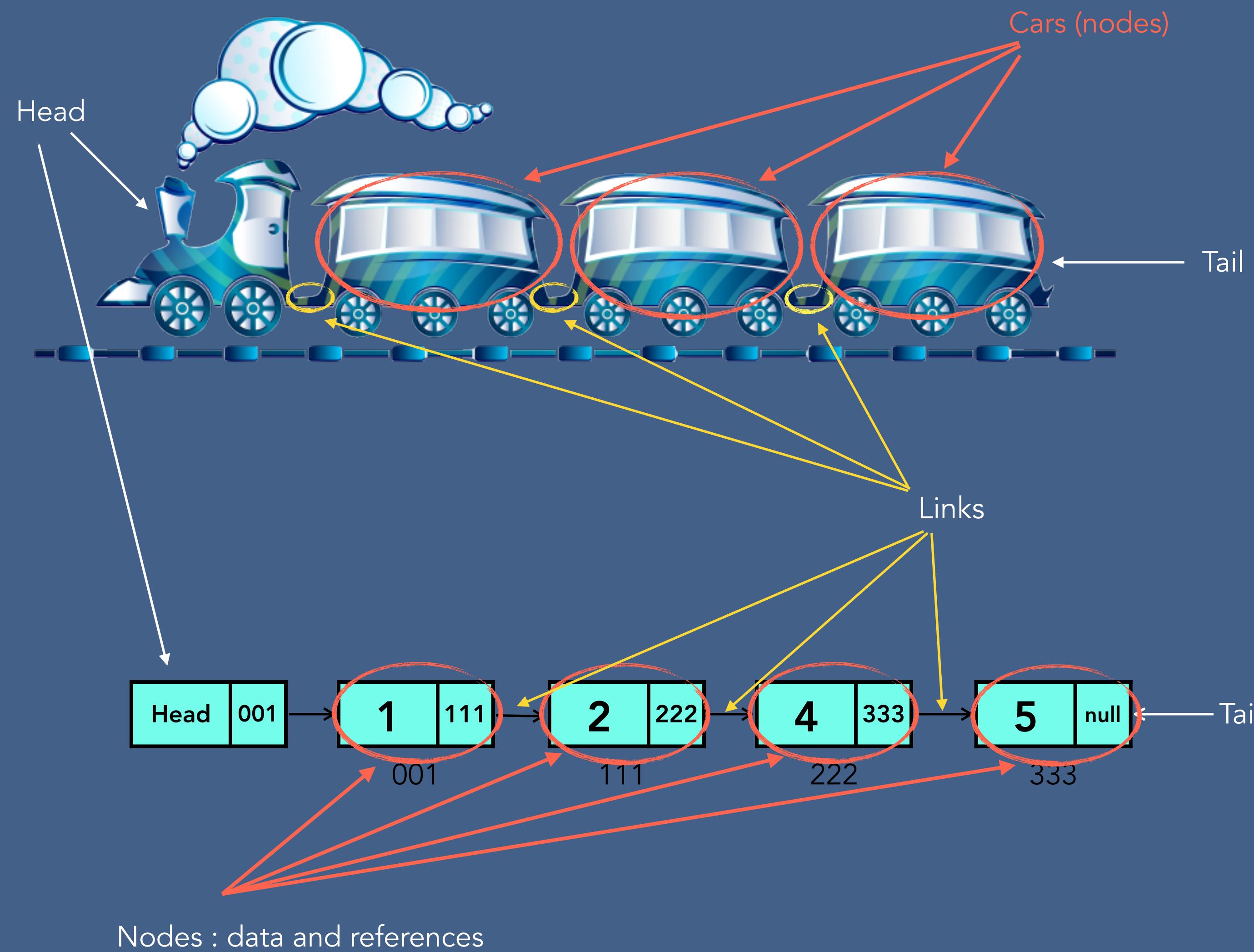
Linked List is a form of a sequential collection and it does not have to be in order. A Linked list is made up of independent nodes that may contain any type of data and each node has a reference to the next node in the link.



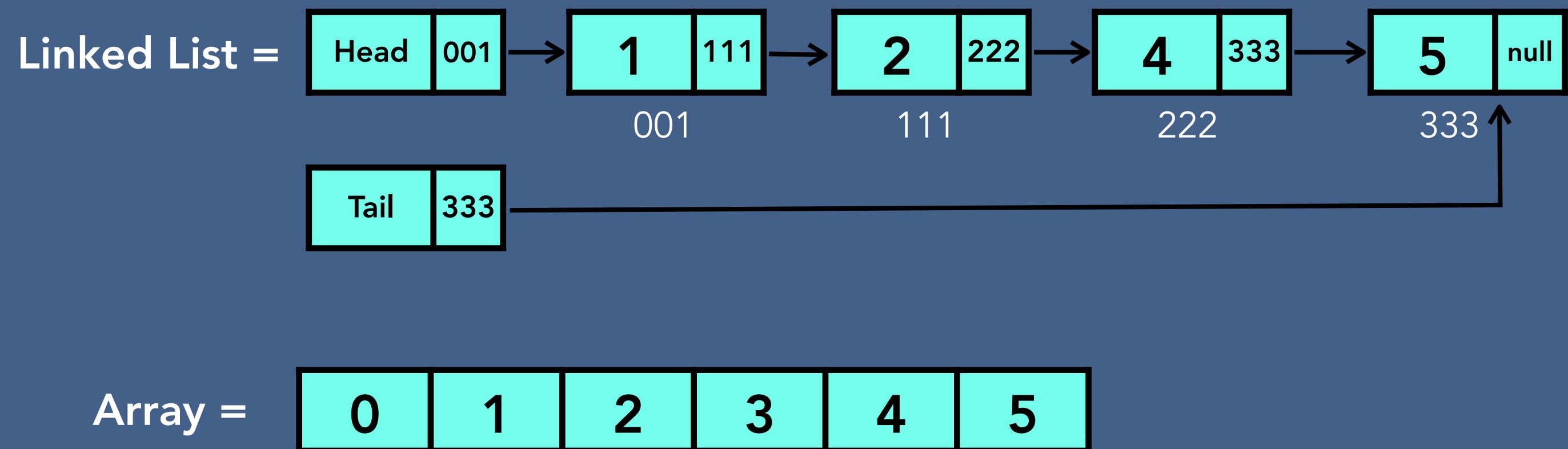
- Each car is independent
- Cars : passengers and links (nodes: data and links)



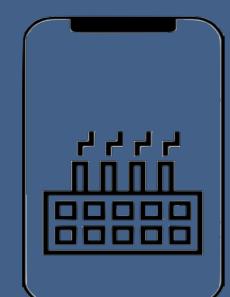
# What is a Linked List?



# Linked List vs Array



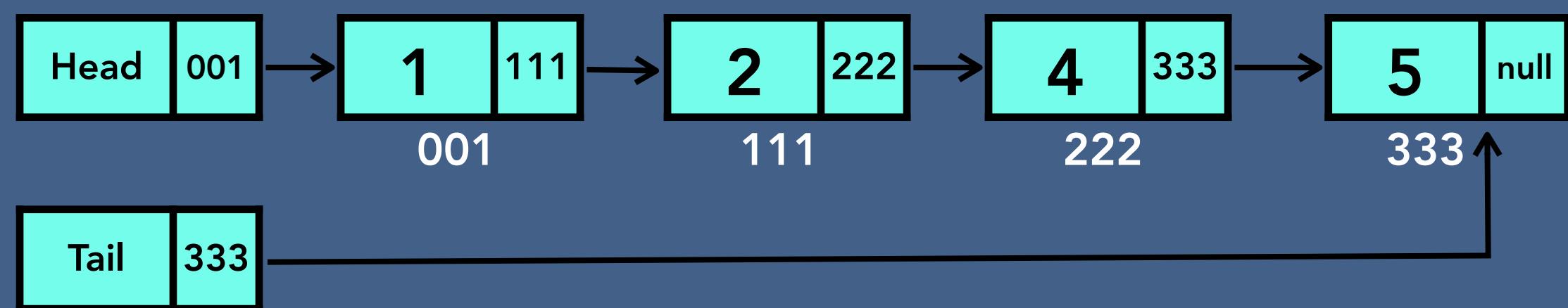
- Elements of Linked list are independent objects
- Variable size - the size of a linked list is not predefined
- Random access - accessing an element is very efficient in arrays



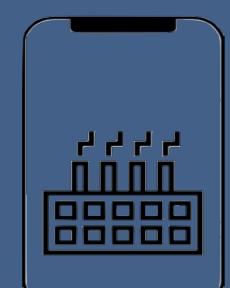
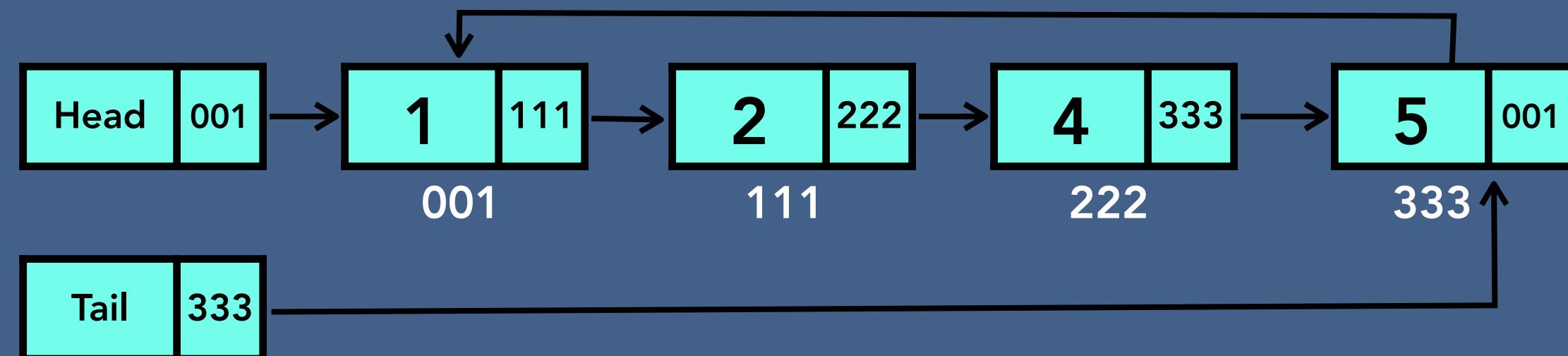
# Types of Linked List

- Singly Linked List
- Circular Singly Linked List
- Doubly Linked List
- Circular Doubly Linked List

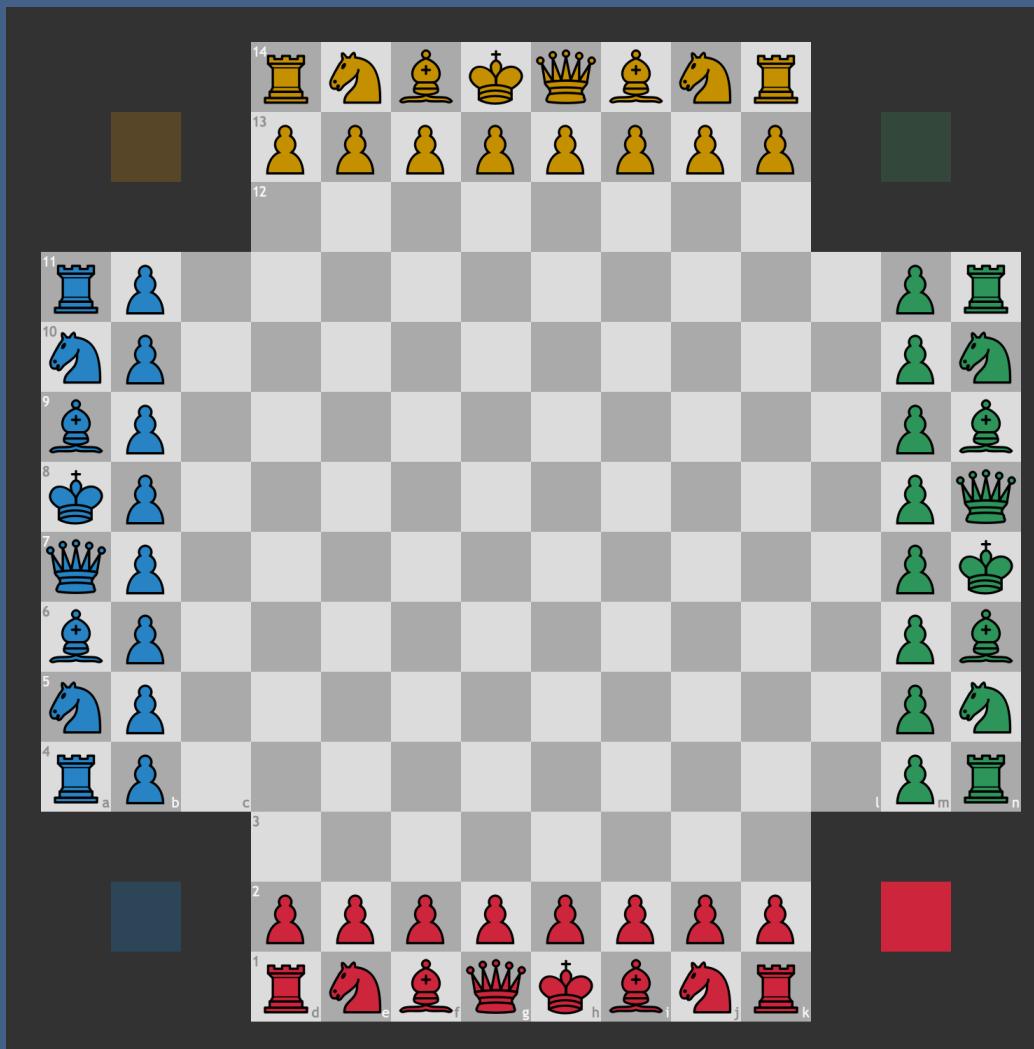
**Singly Linked List**



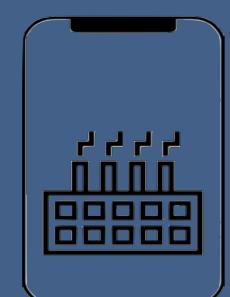
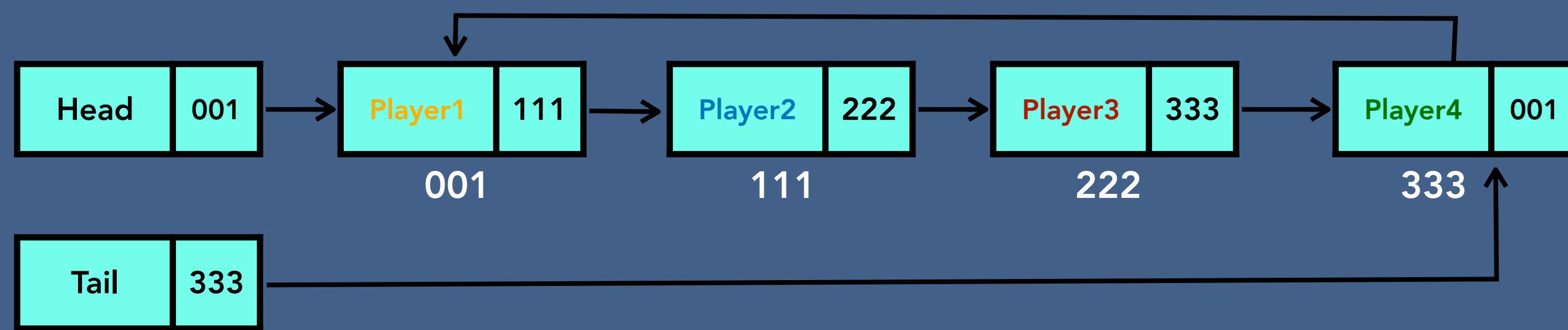
**Circular Singly Linked List**



# Types of Linked List

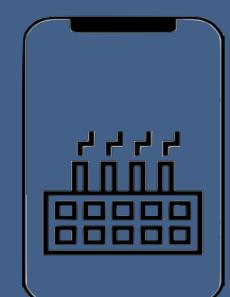
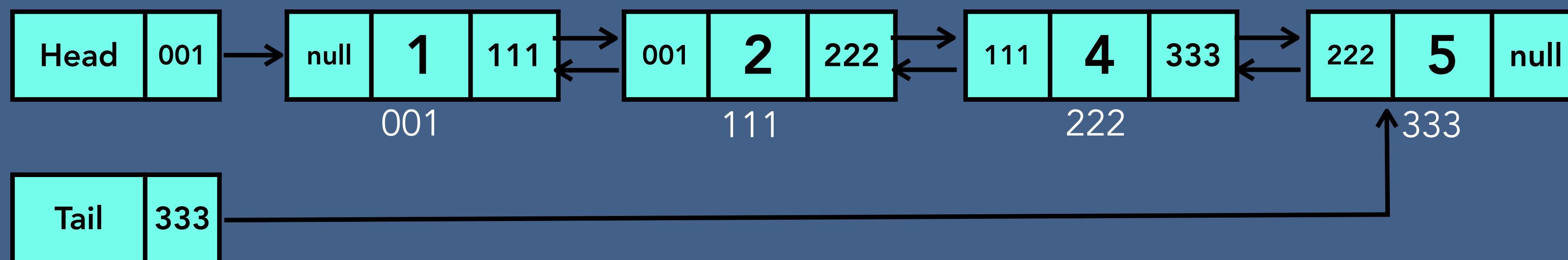


**Circular Singly Linked List**

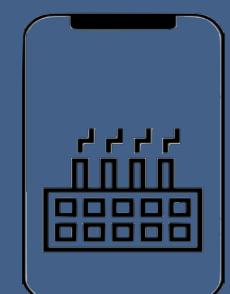
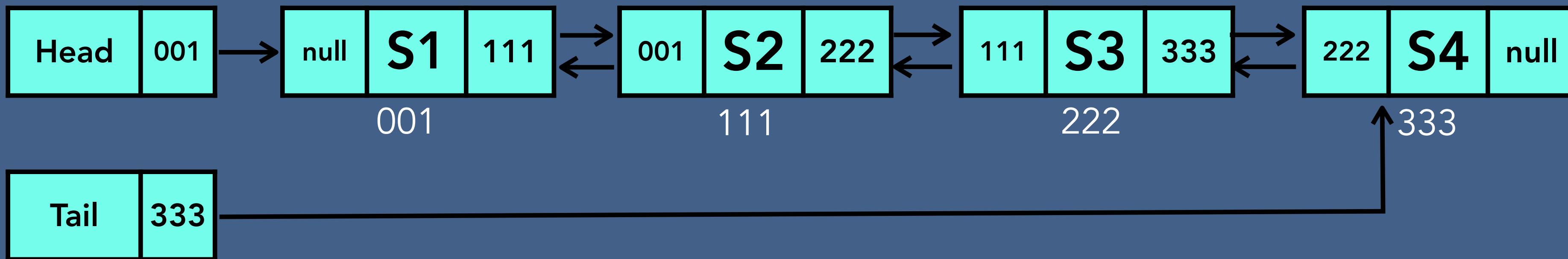
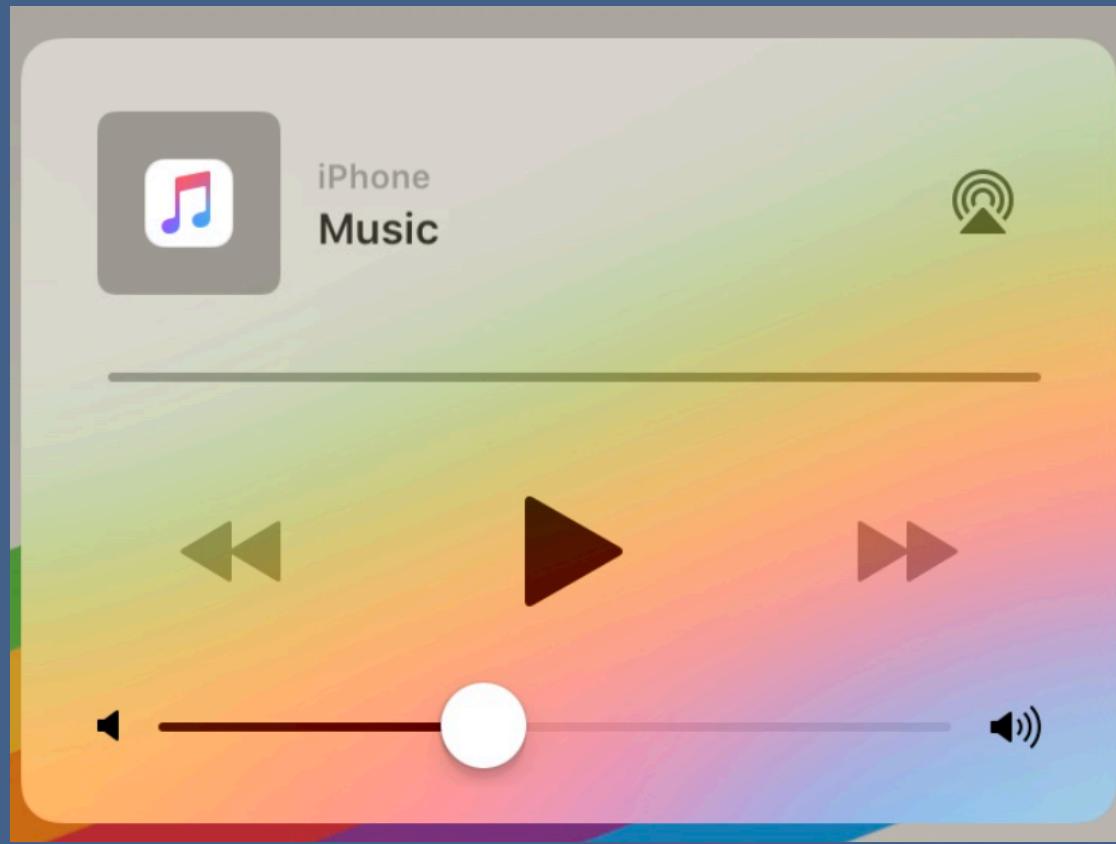


# Types of Linked List

## Doubly Linked List

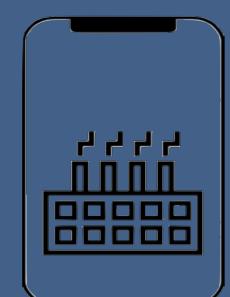
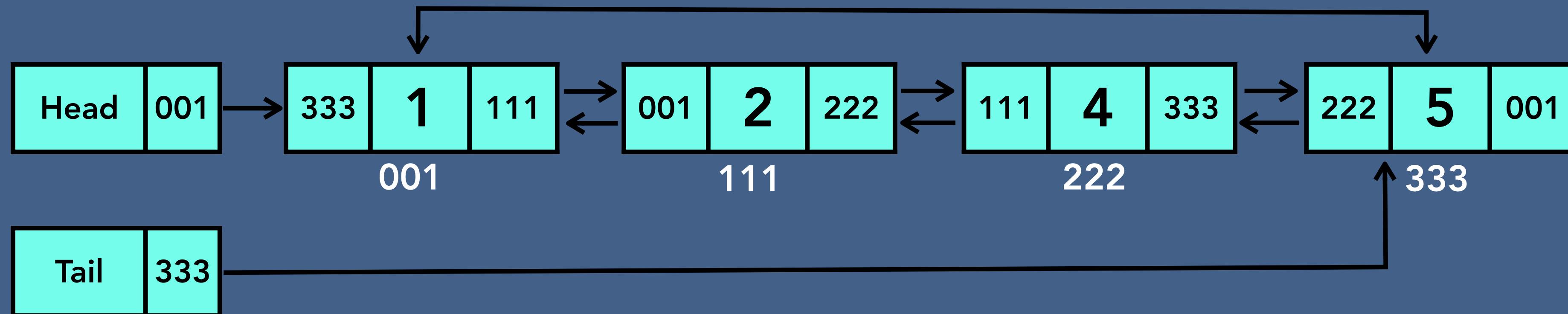


# Types of Linked List



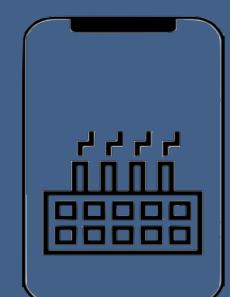
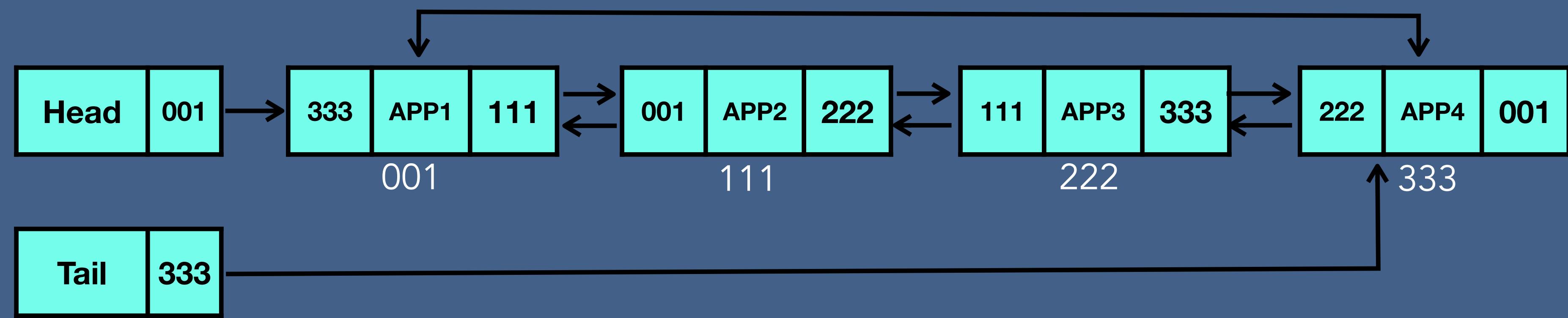
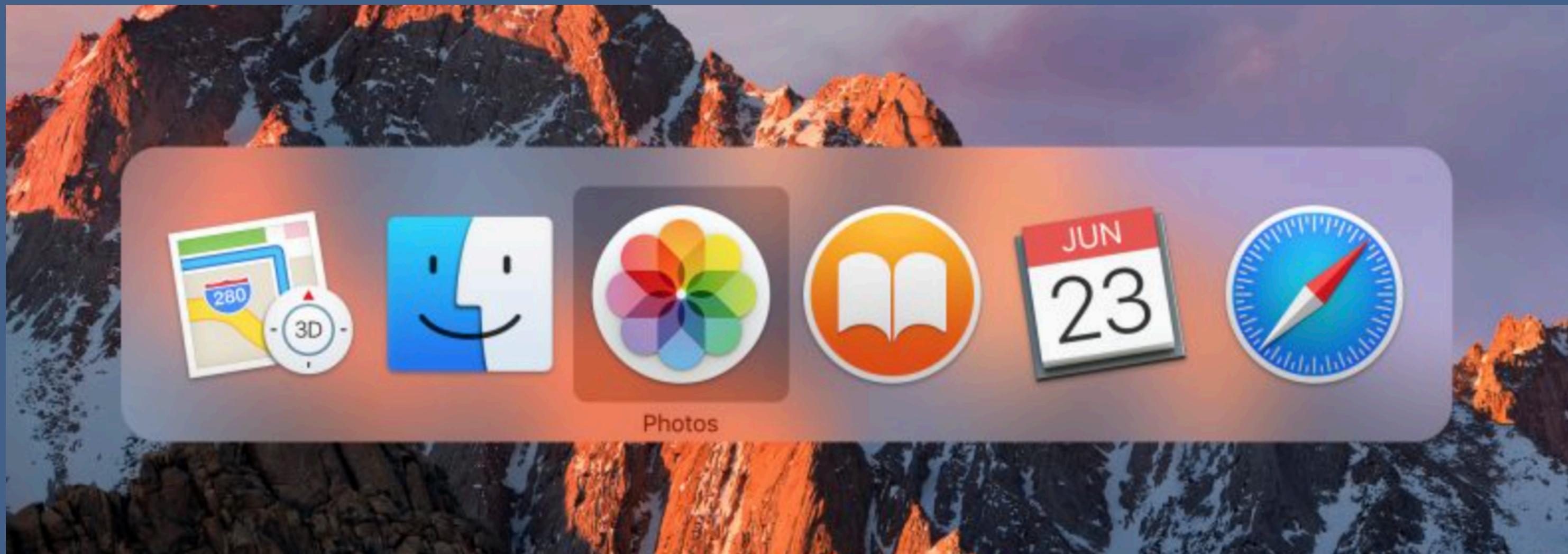
# Types of Linked List

## Circular Doubly Linked List



# Types of Linked List

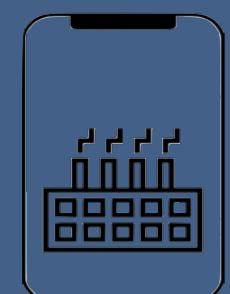
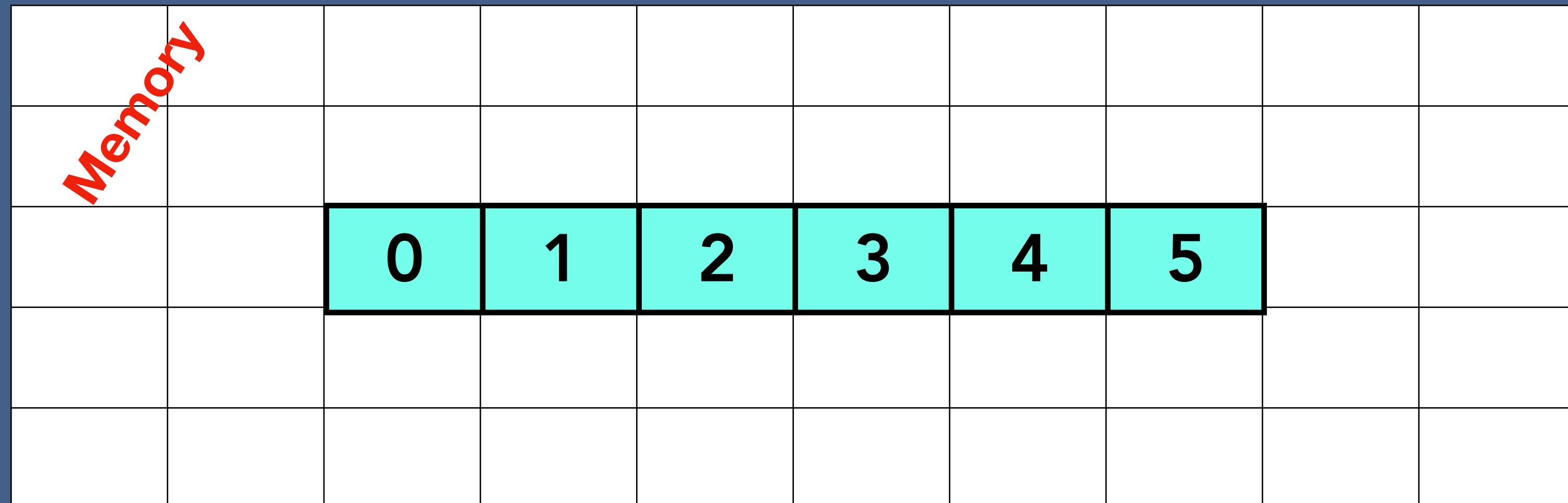
Cmd+shift+tab



# Linked List in Memory

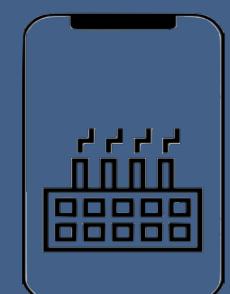
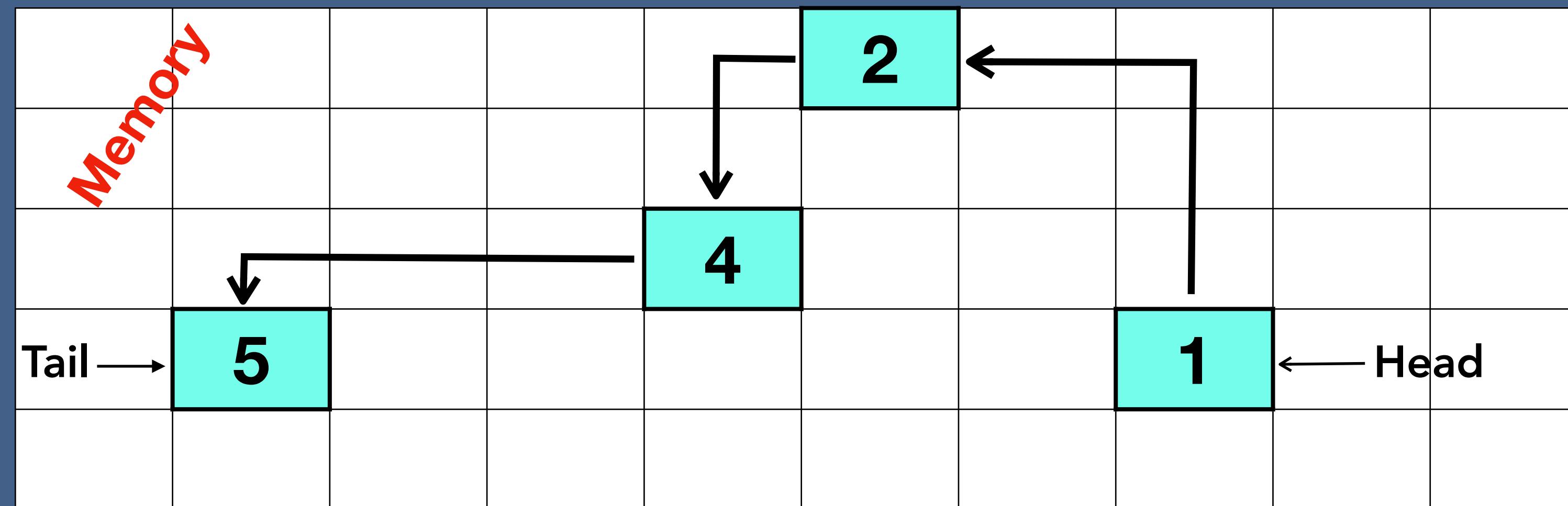
Arrays in memory :

0	1	2	3	4	5
---	---	---	---	---	---

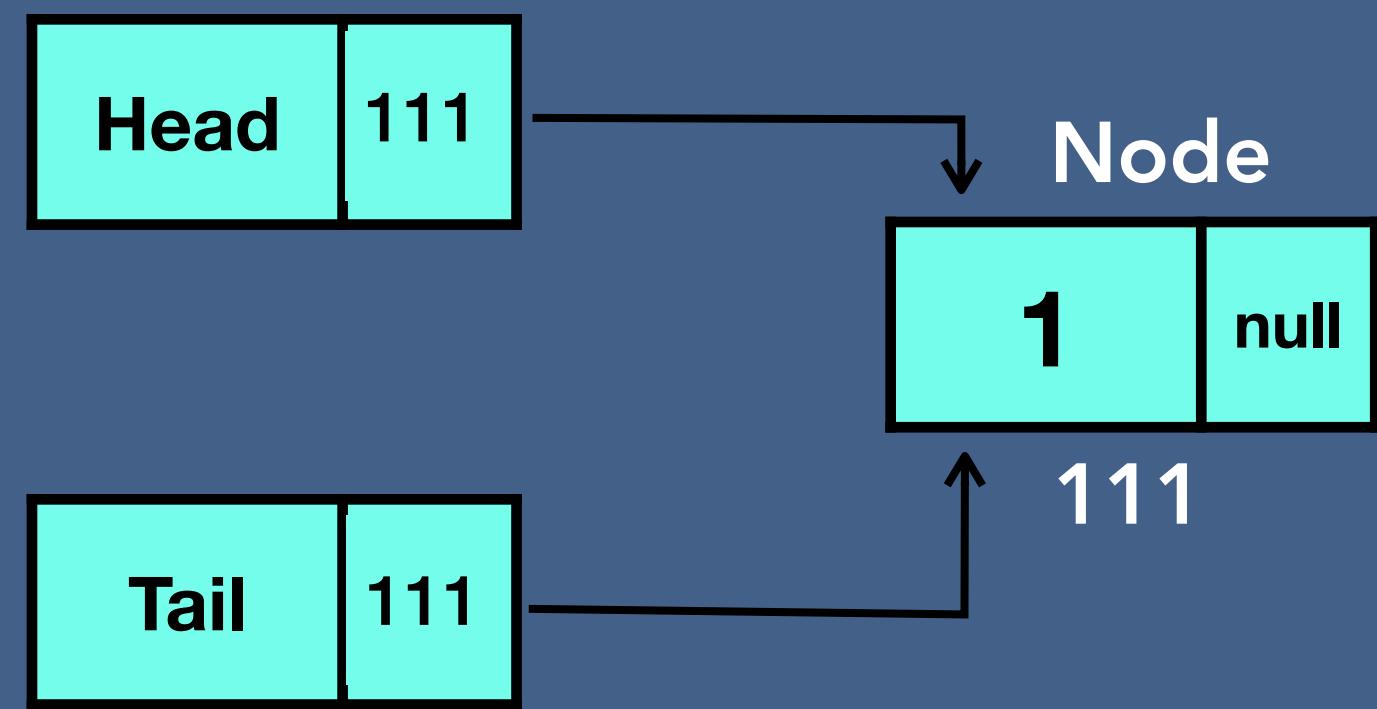


# Linked List in Memory

Linked list:



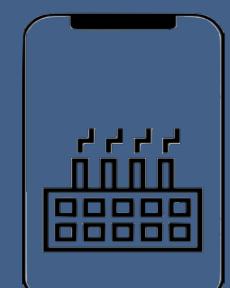
# Creation of Singly Linked List



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

$O(1)$



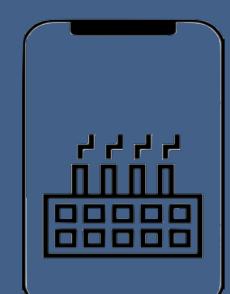
Create a blank Node and assign a value to it  
and reference to null.

$O(1)$



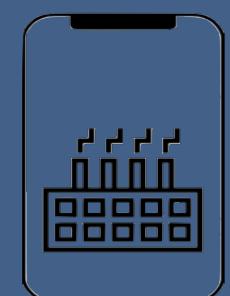
Link Head and Tail with these Node

$O(1)$



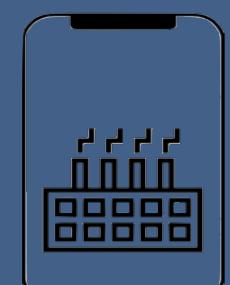
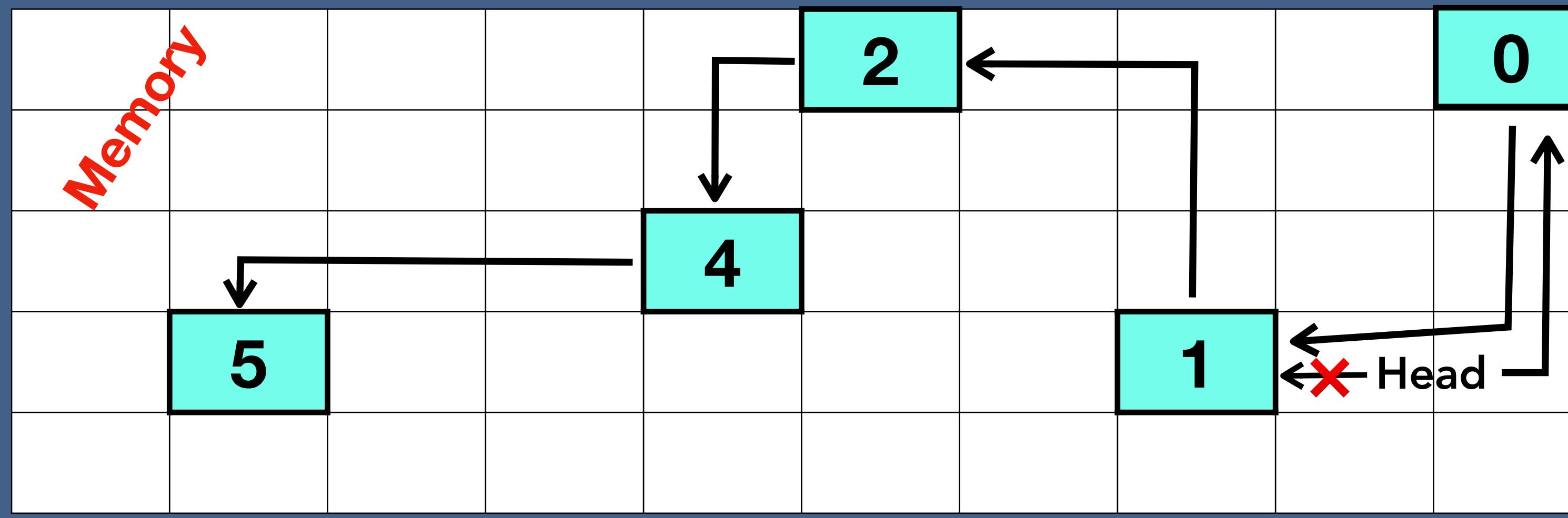
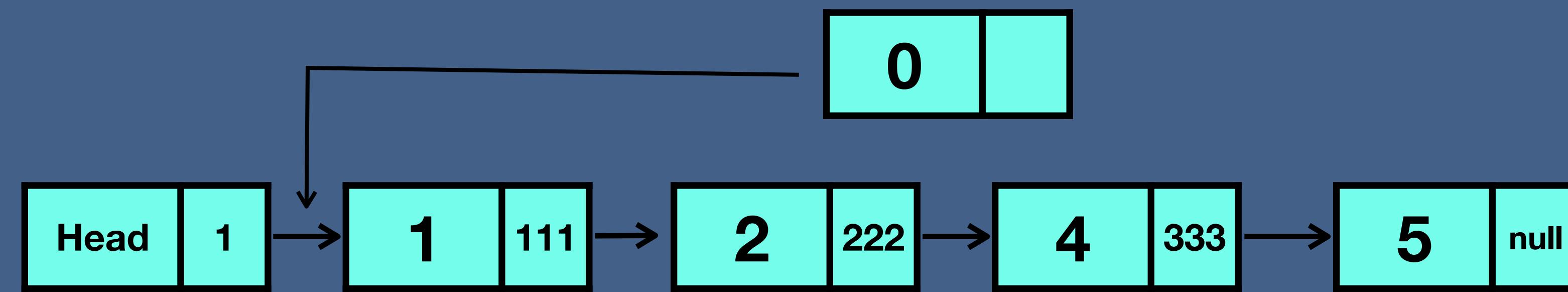
# Insertion to Linked List in Memory

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.



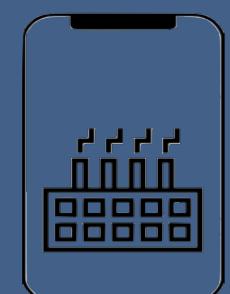
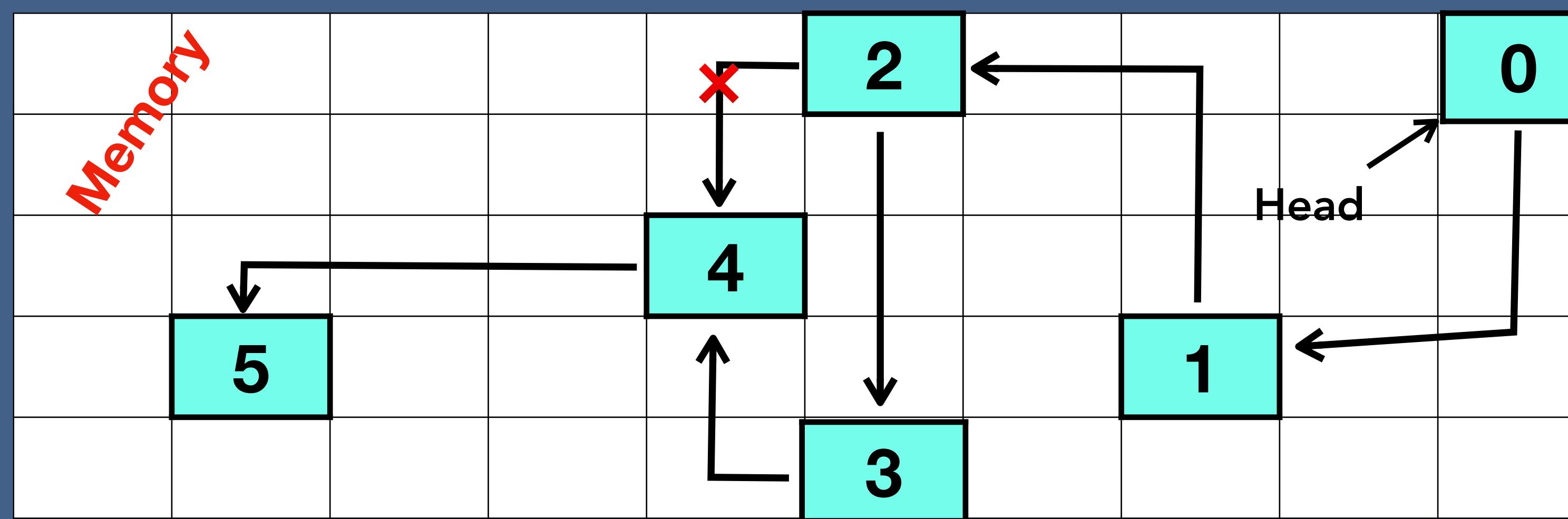
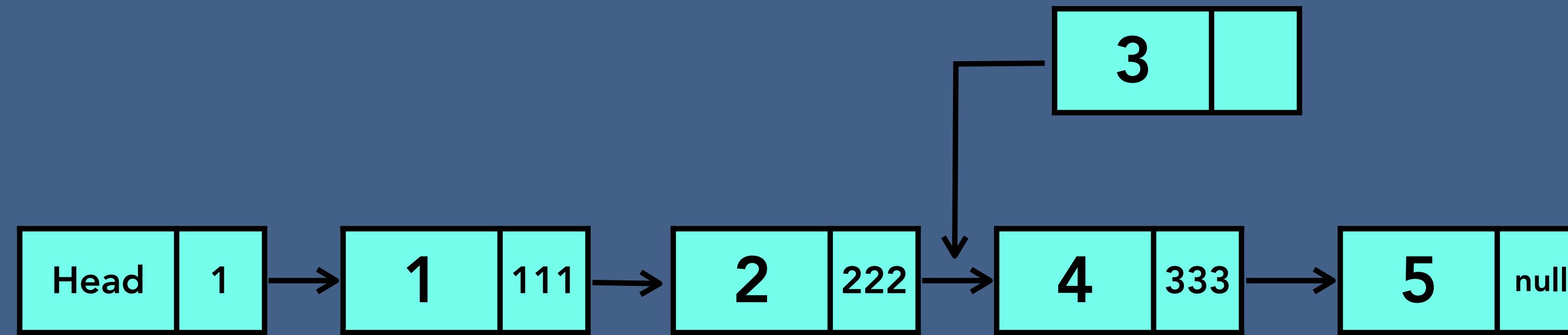
# Insertion to Linked List in Memory

Insert a new node at the beginning



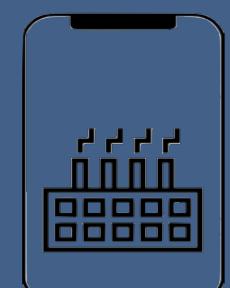
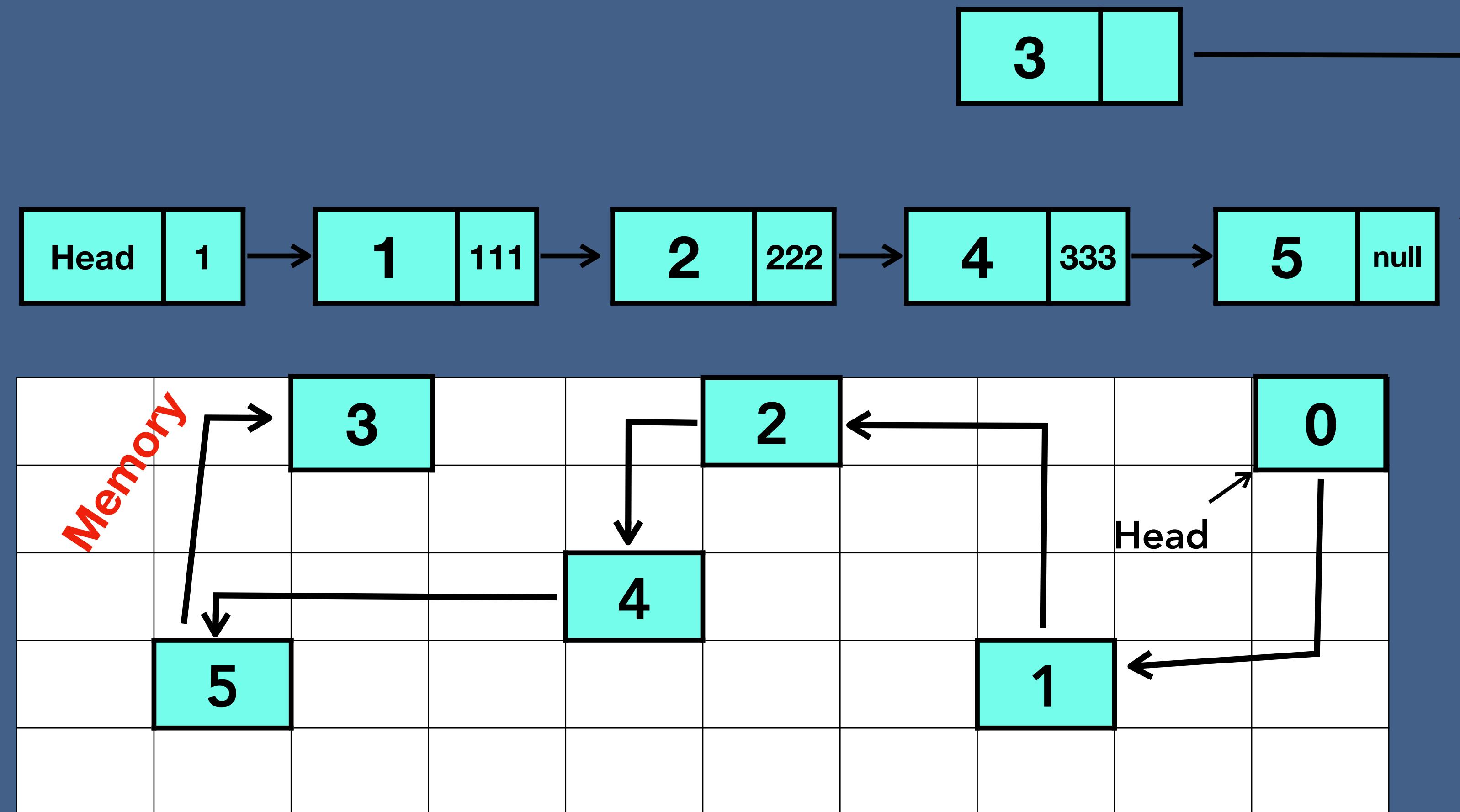
# Insertion to Linked List in Memory

Insert a new node after a node

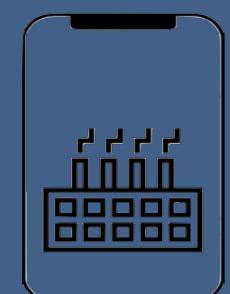
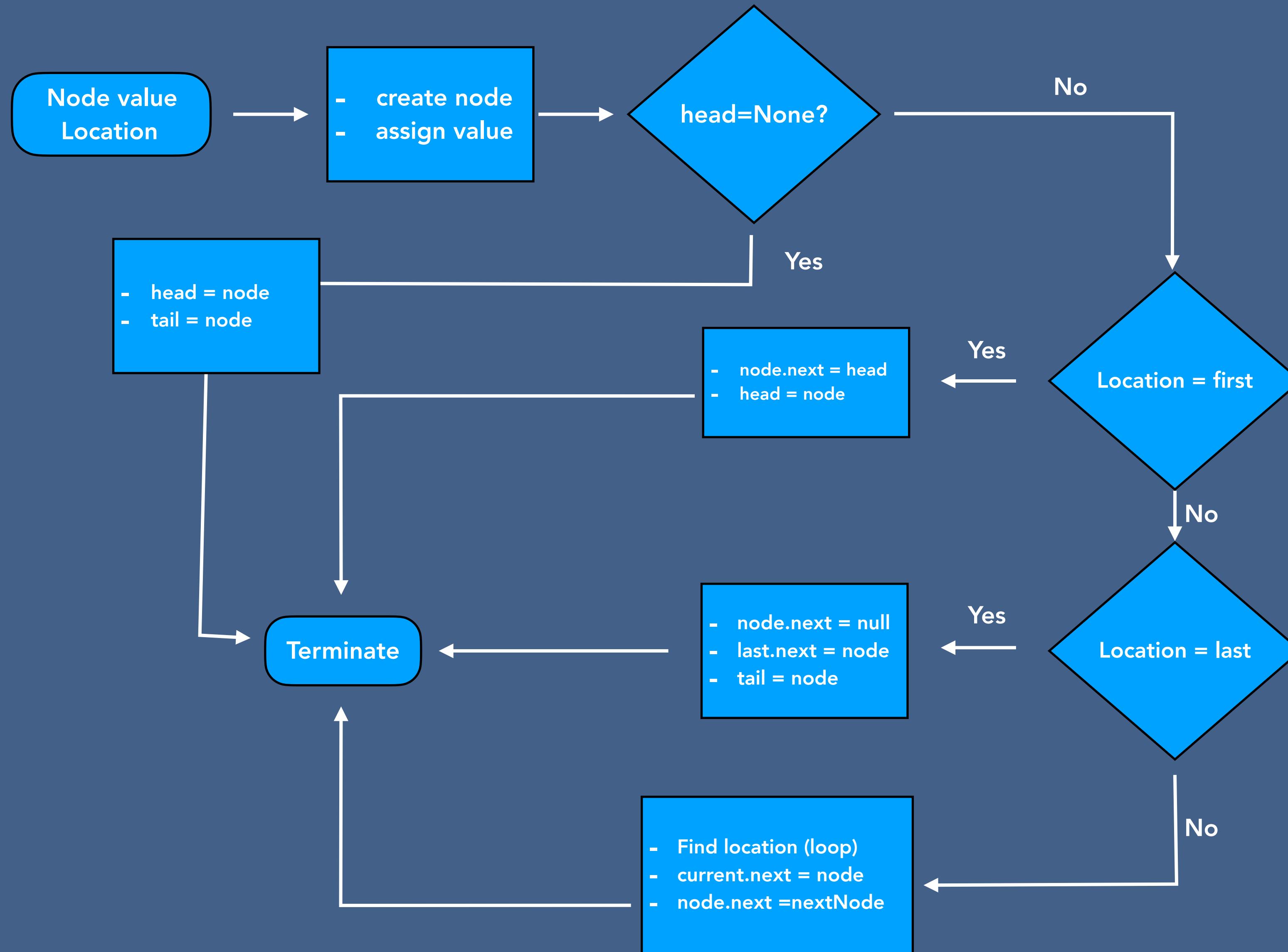


# Insertion to Linked List in Memory

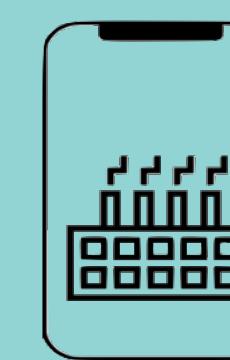
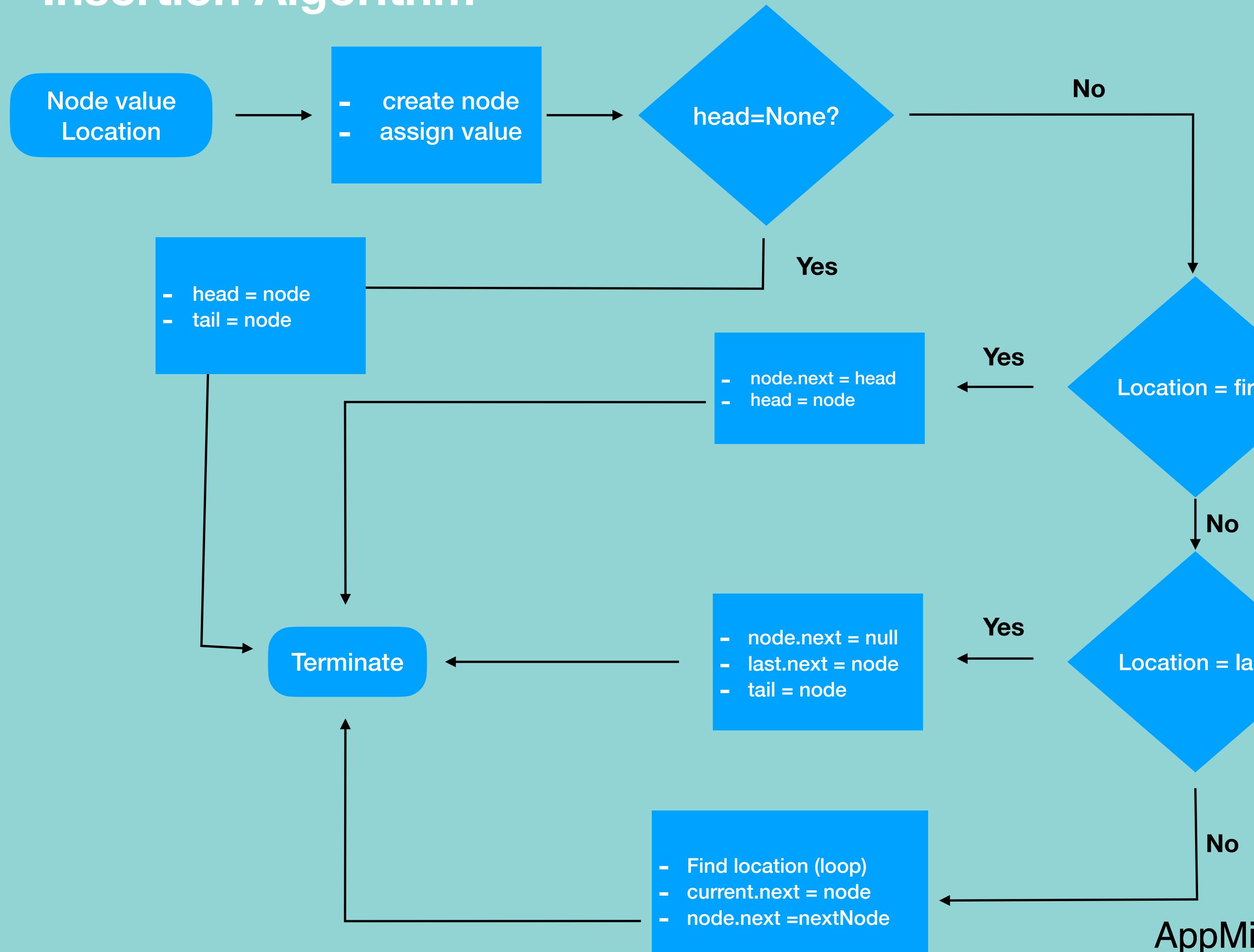
Insert a new node at the end of linked list



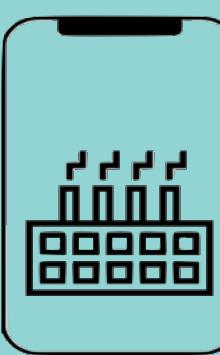
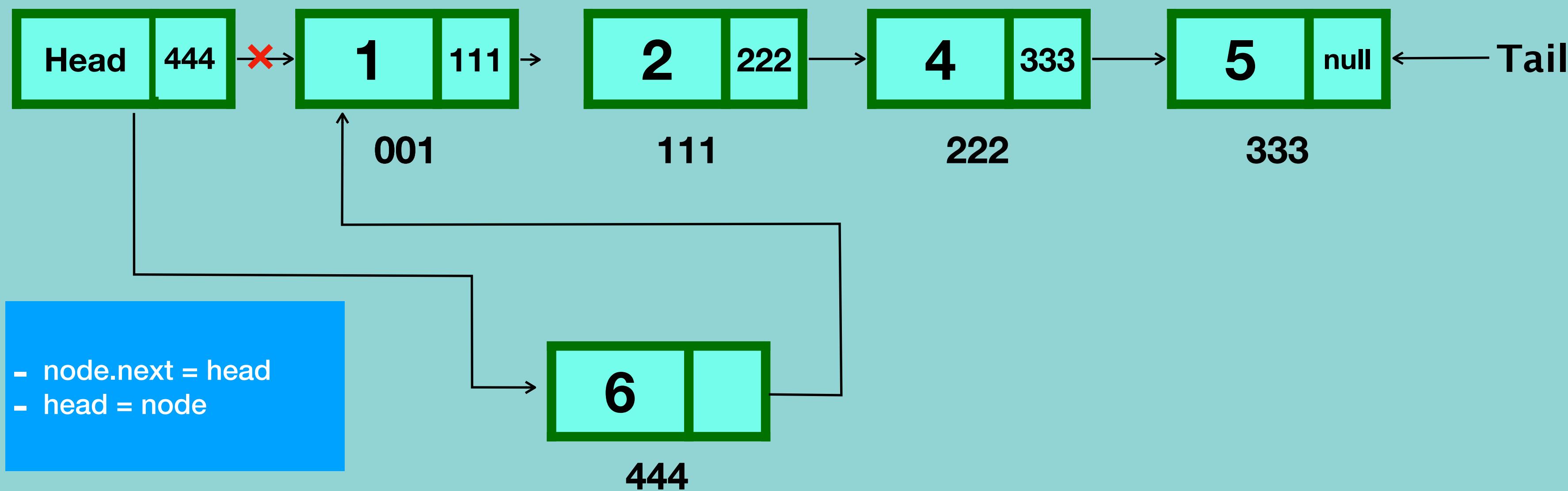
# Insertion Algorithm



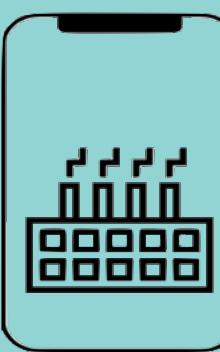
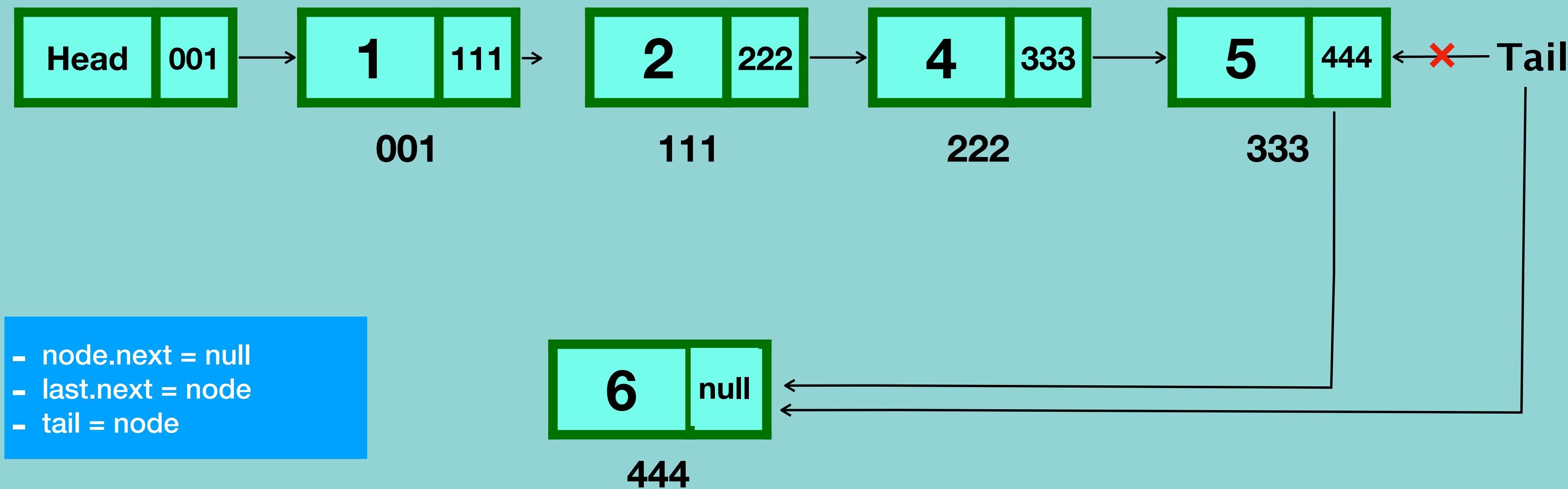
# Insertion Algorithm



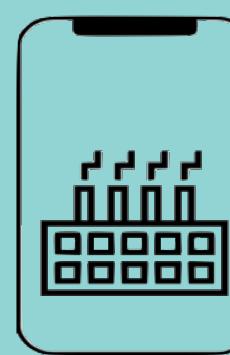
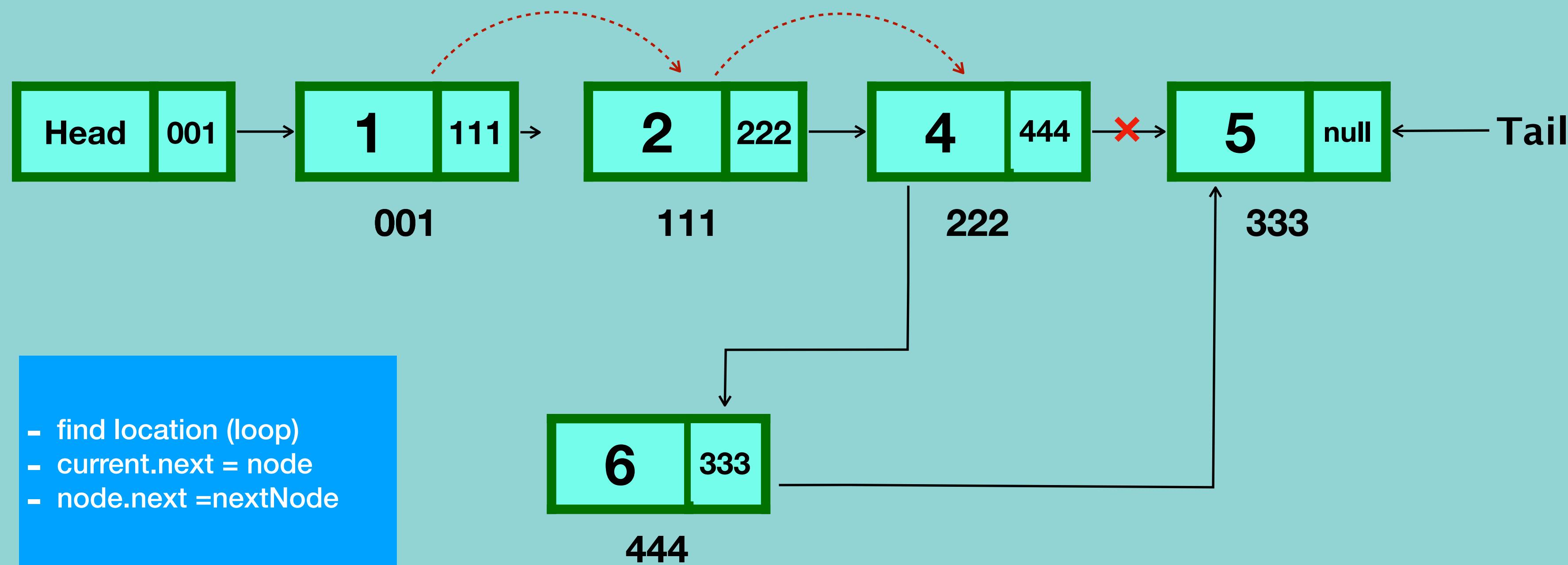
## Singly Linked List Insertion at the beginning



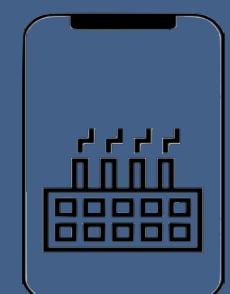
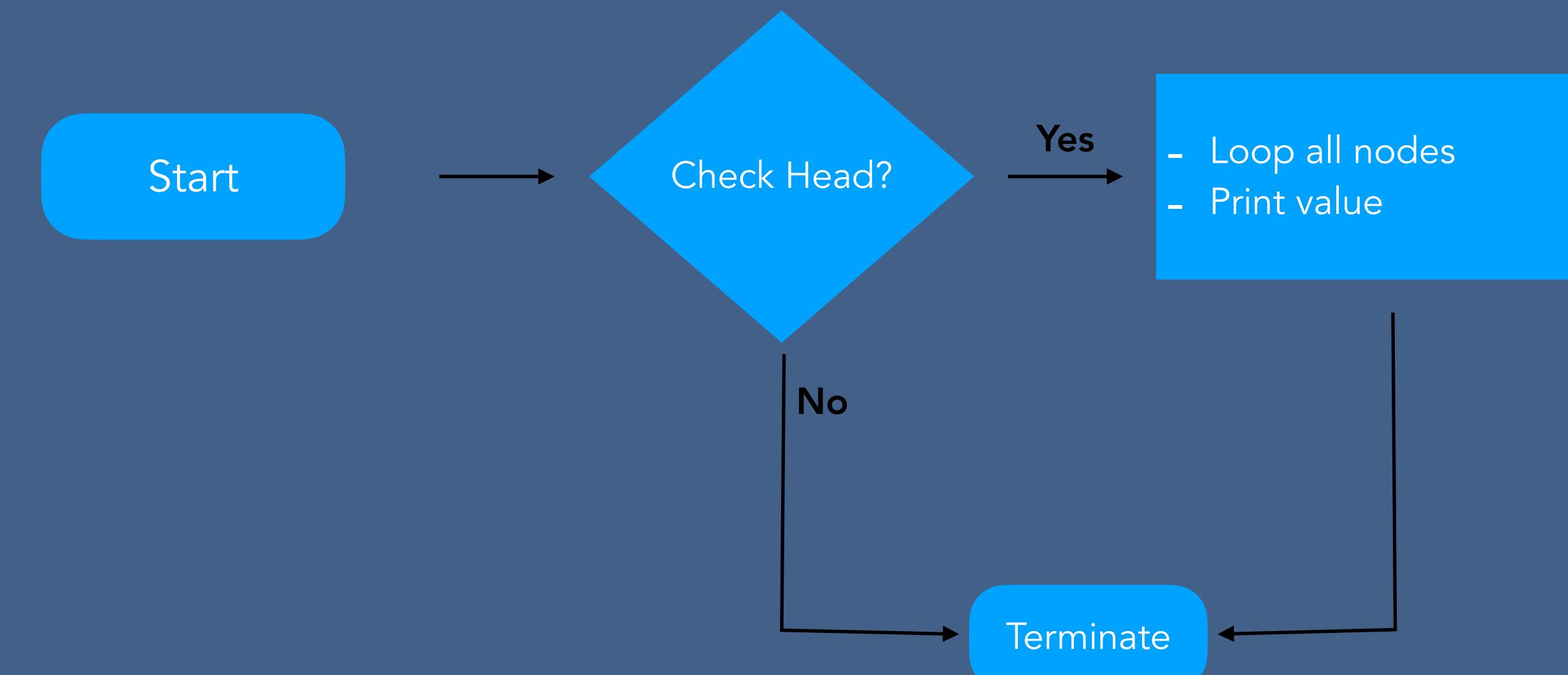
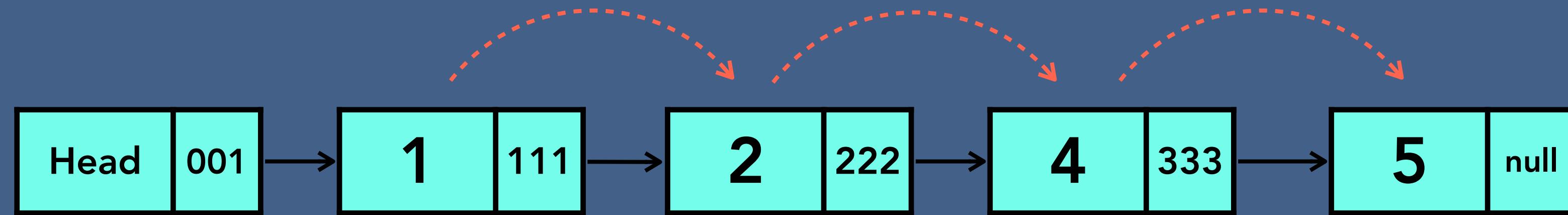
## Singly Linked List Insertion at the end



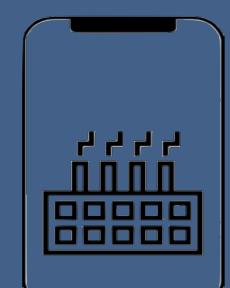
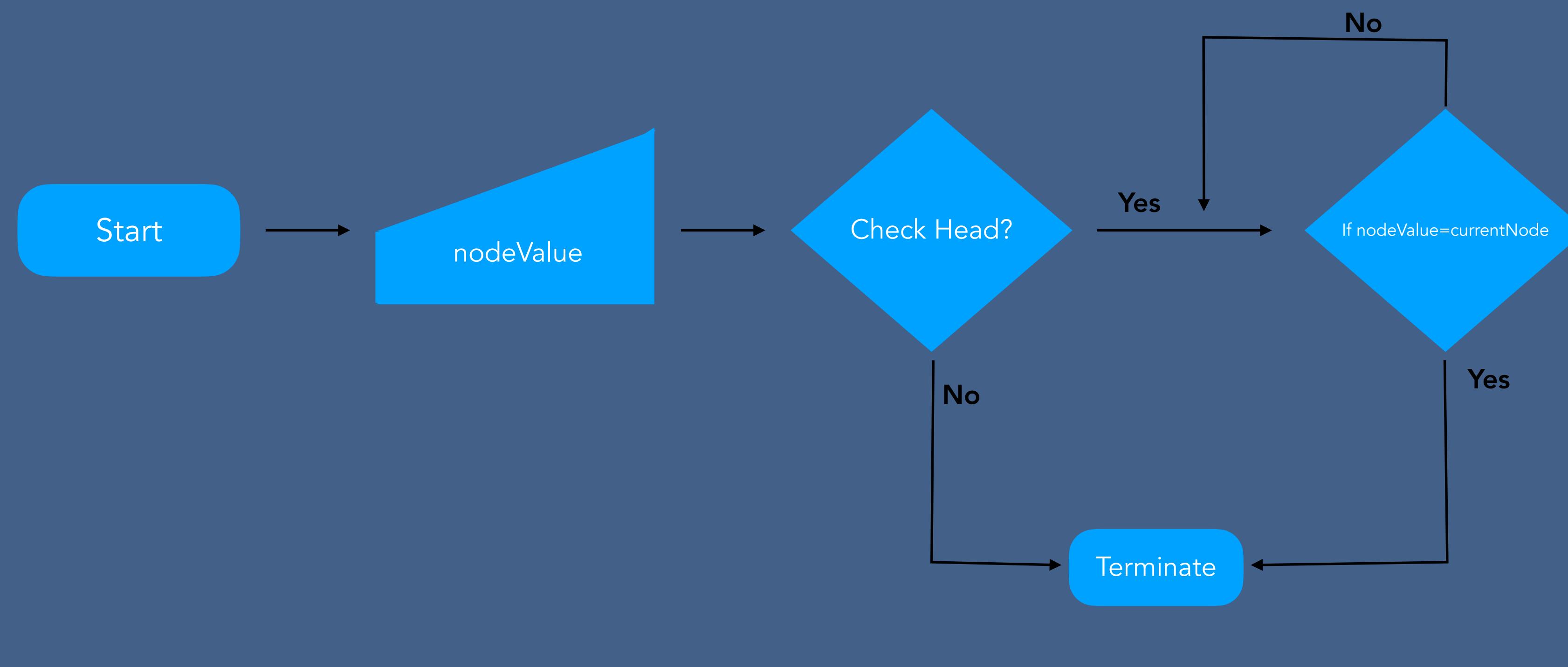
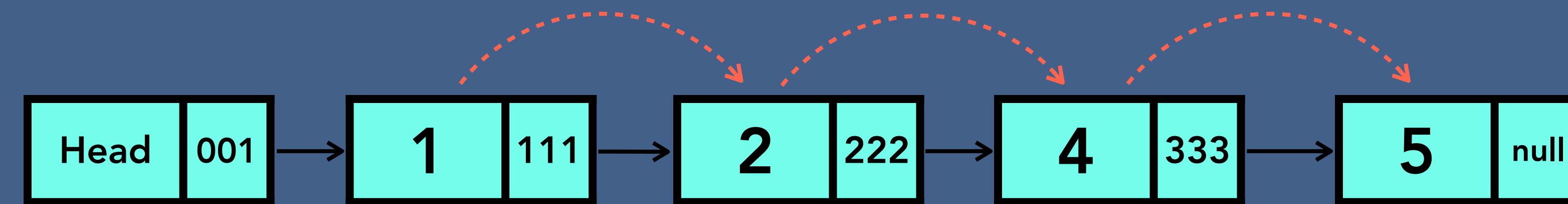
# Singly Linked List Insertion in the middle



# Traversal of Singly Linked List

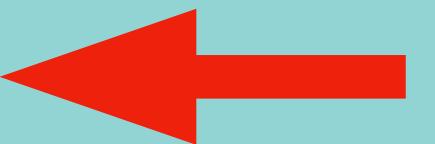


# Search in Singly Linked List

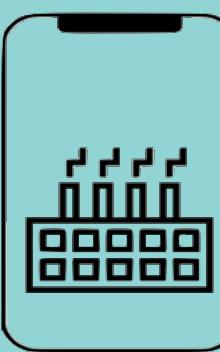
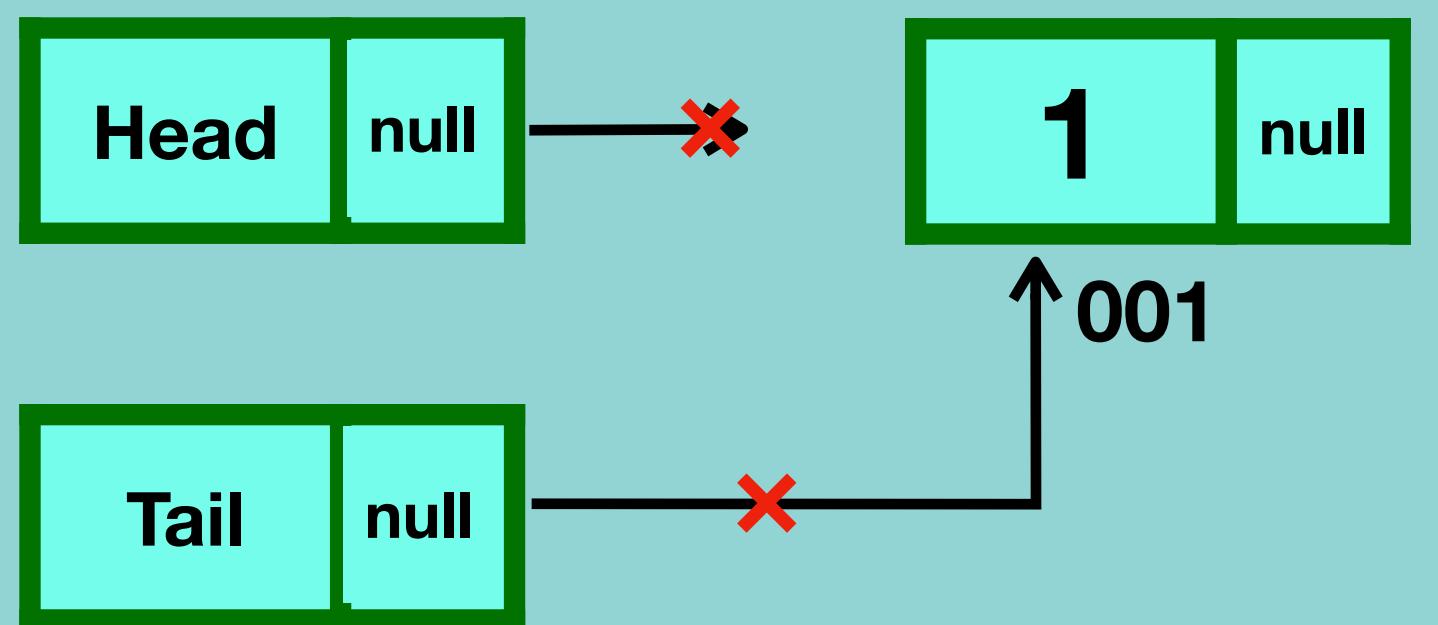


# Singly Linked list Deletion

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

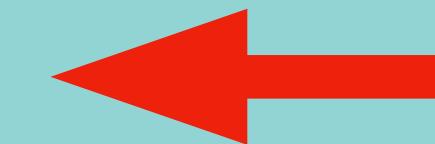


## Case 1 - one node

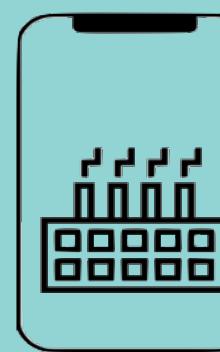
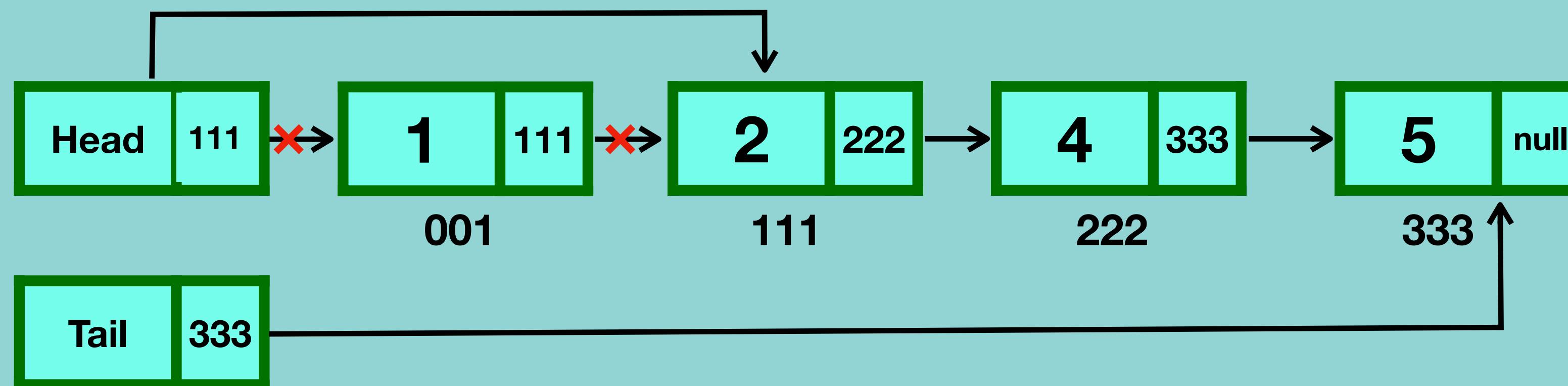


# Singly Linked list Deletion

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

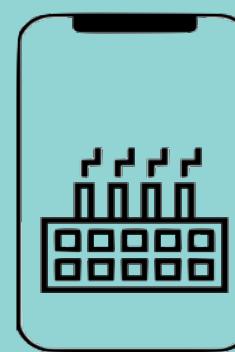
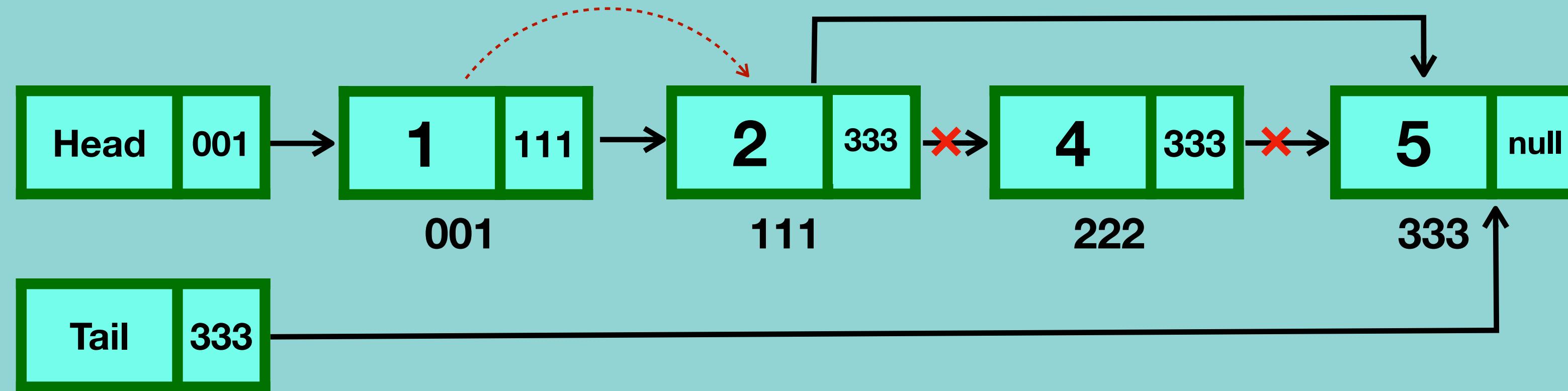


## Case 2 - more than one node



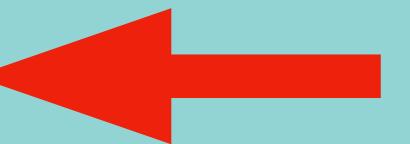
# Singly Linked list Deletion

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

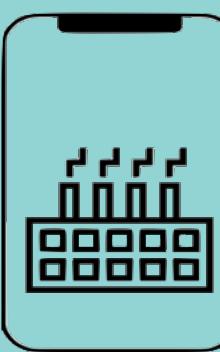
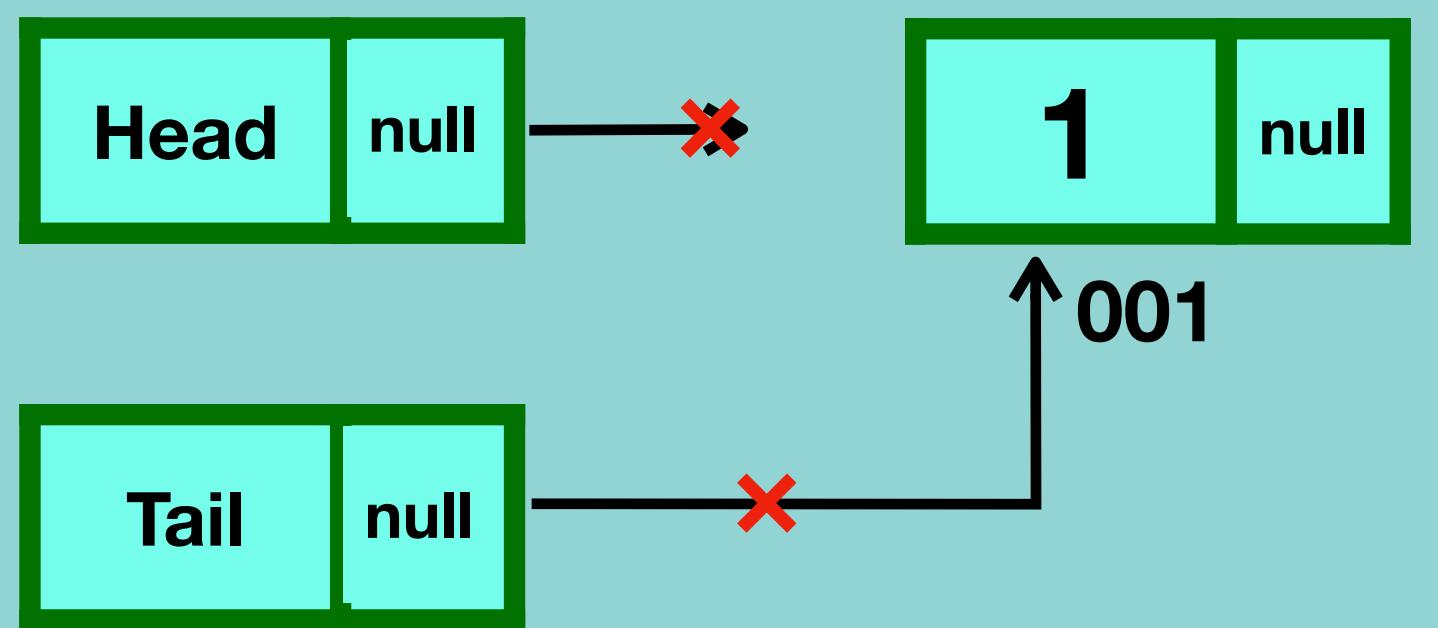


# Singly Linked list Deletion

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

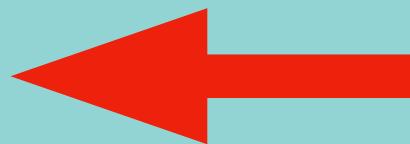


## Case 1 - one node

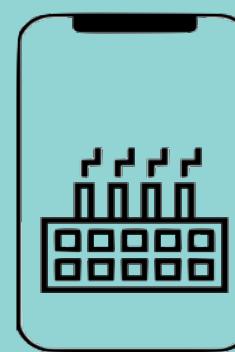
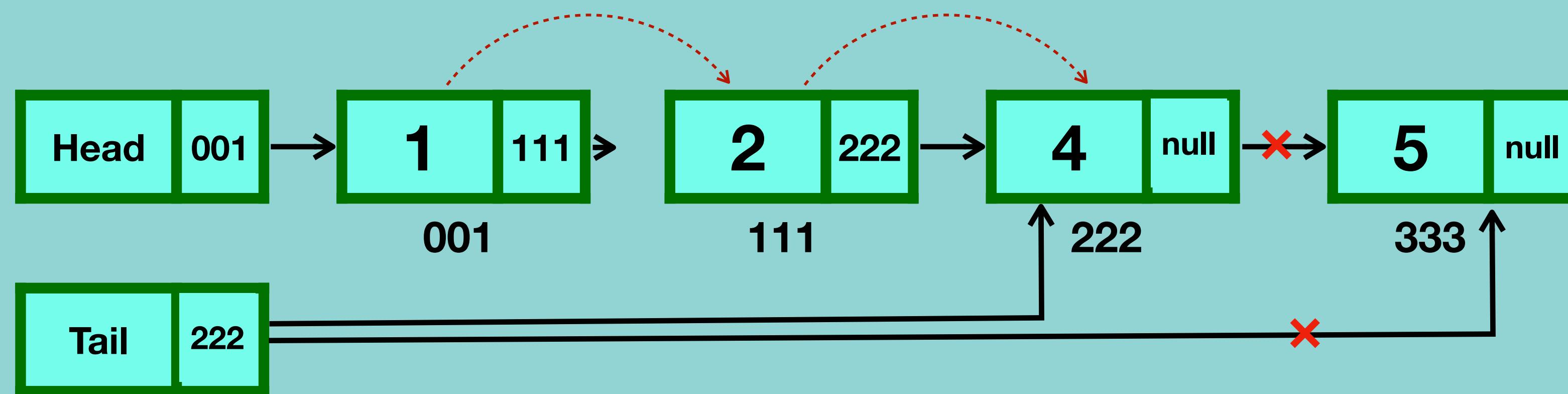


# Singly Linked list Deletion

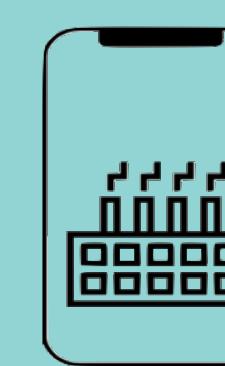
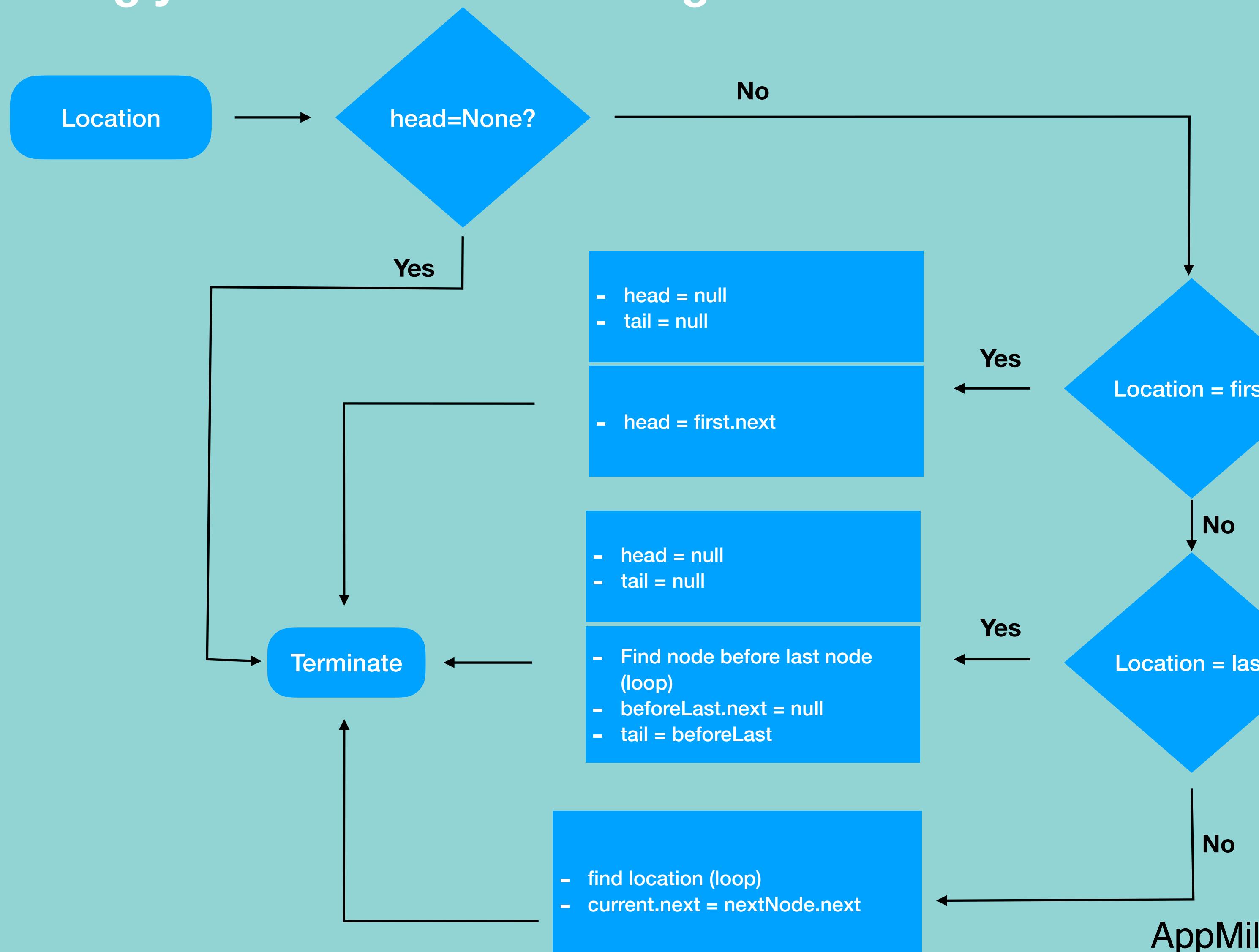
- Deleting the first node
- Deleting any given node
- Deleting the last node



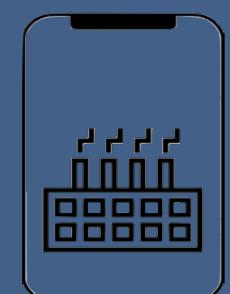
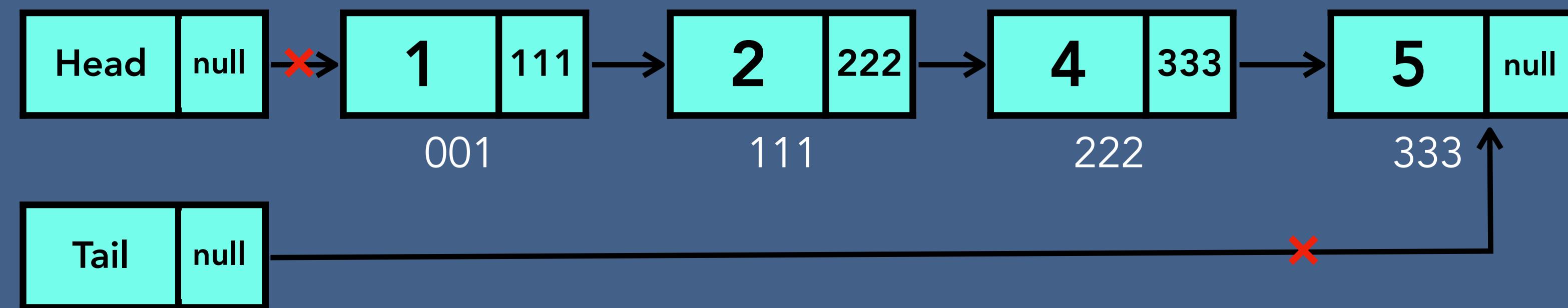
## Case 2 - more than one node



# Singly Linked list Deletion Algorithm

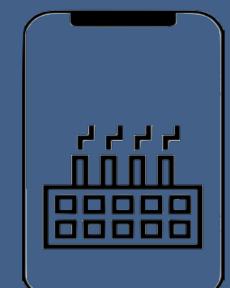
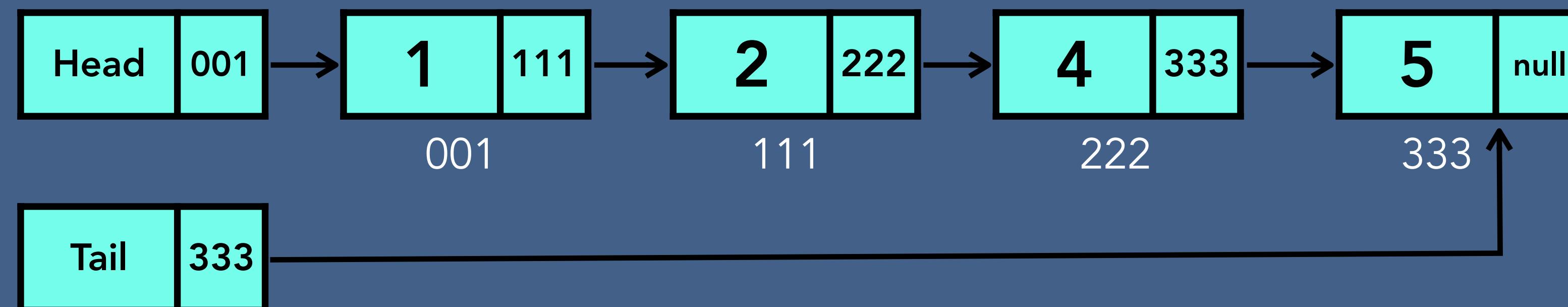


# Delete entire Singly Linked List

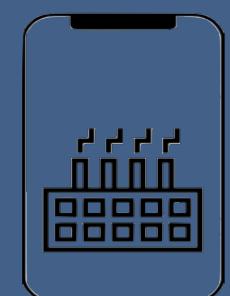
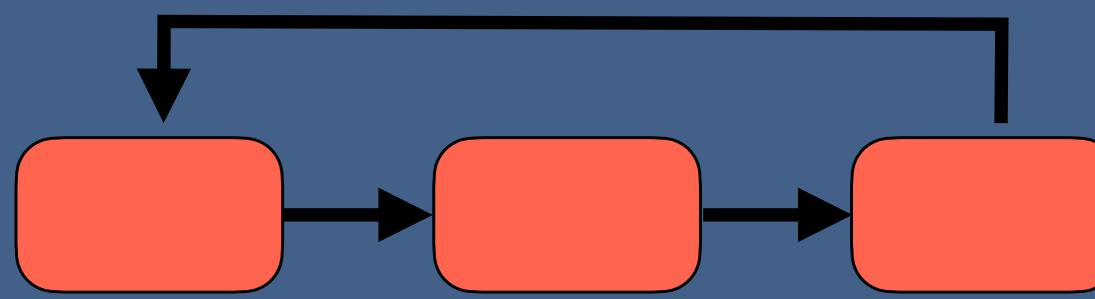


# Time and Space Complexity of Singly Linked List

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	$O(1)$	$O(1)$

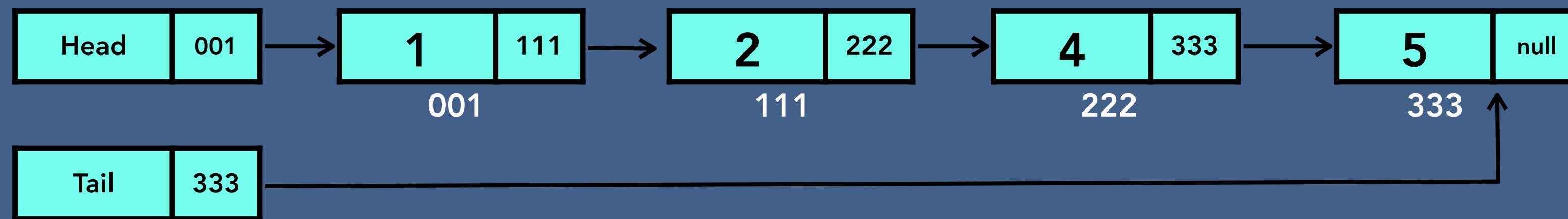


# Circular Singly Linked List

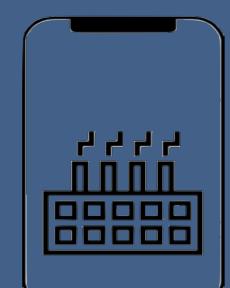
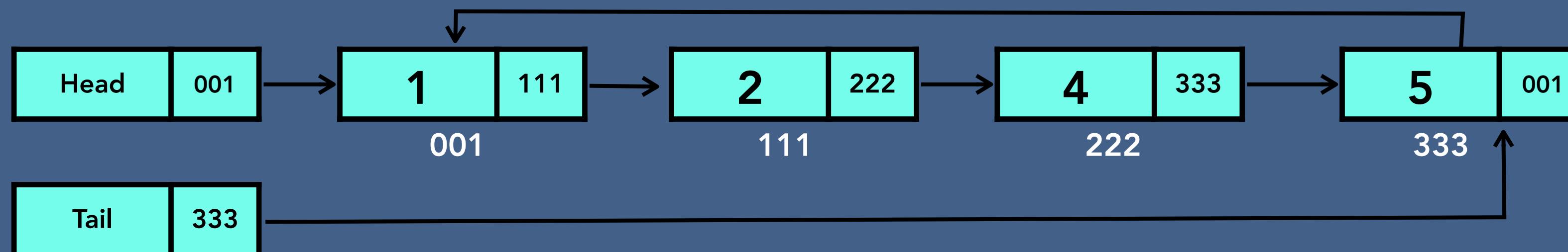


# Circular Singly Linked List

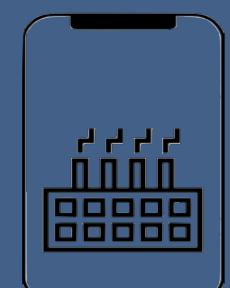
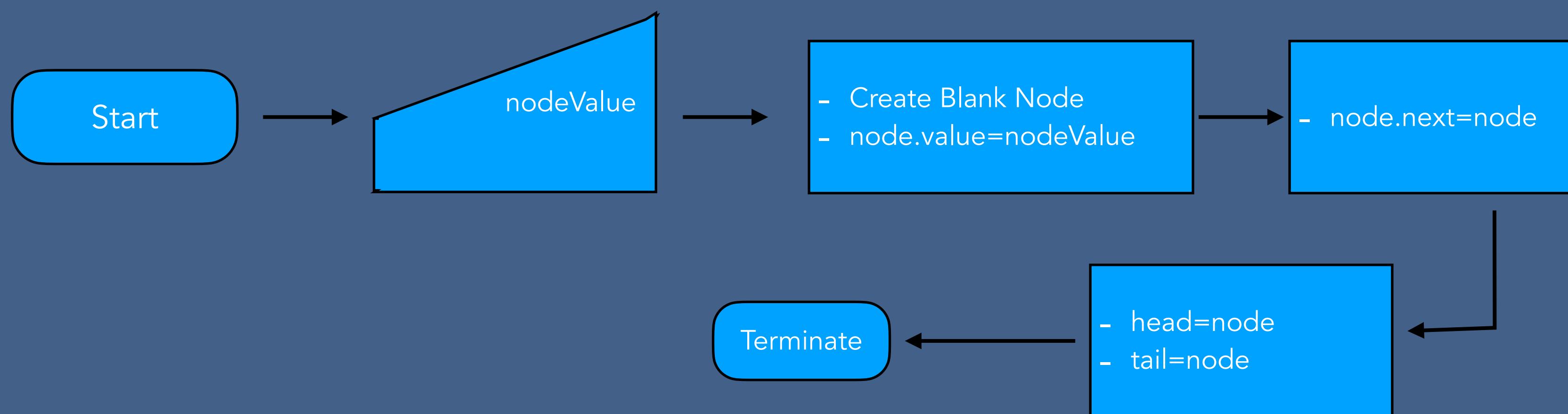
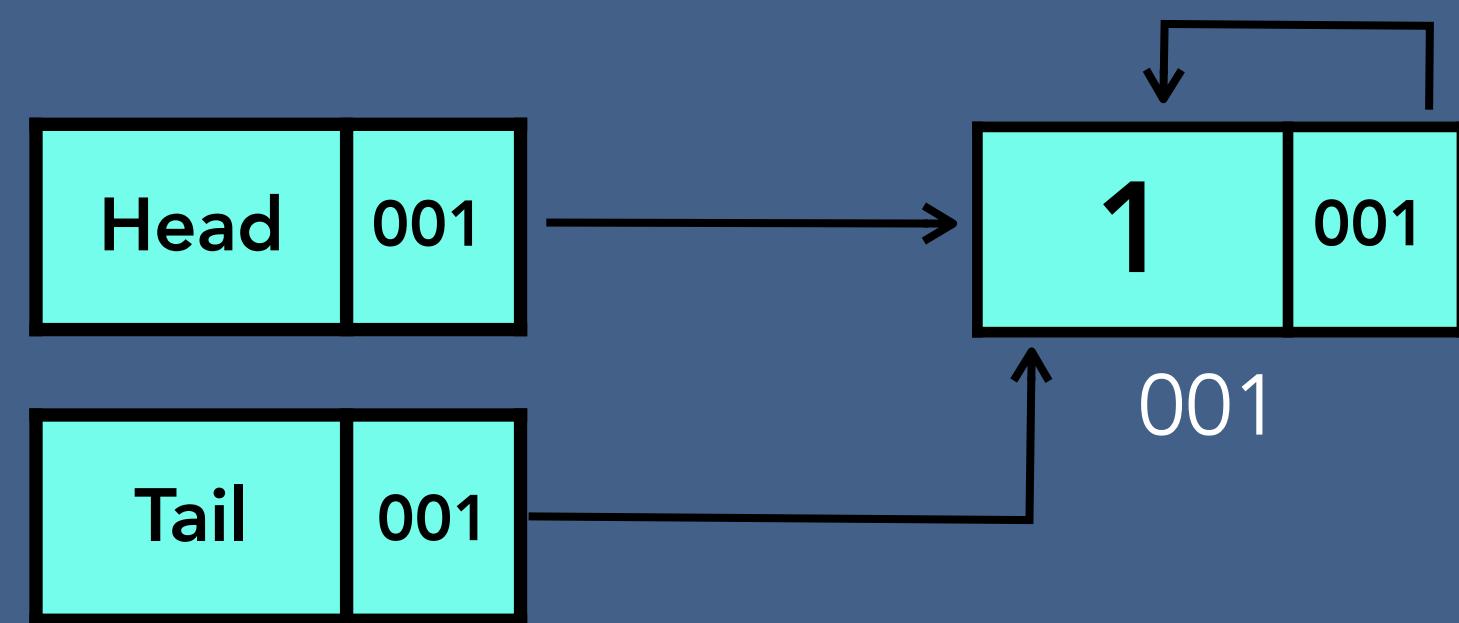
Singly Linked List



Circular Singly Linked List

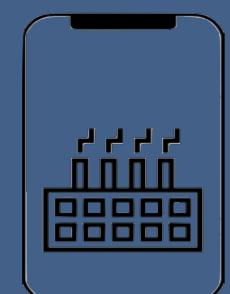
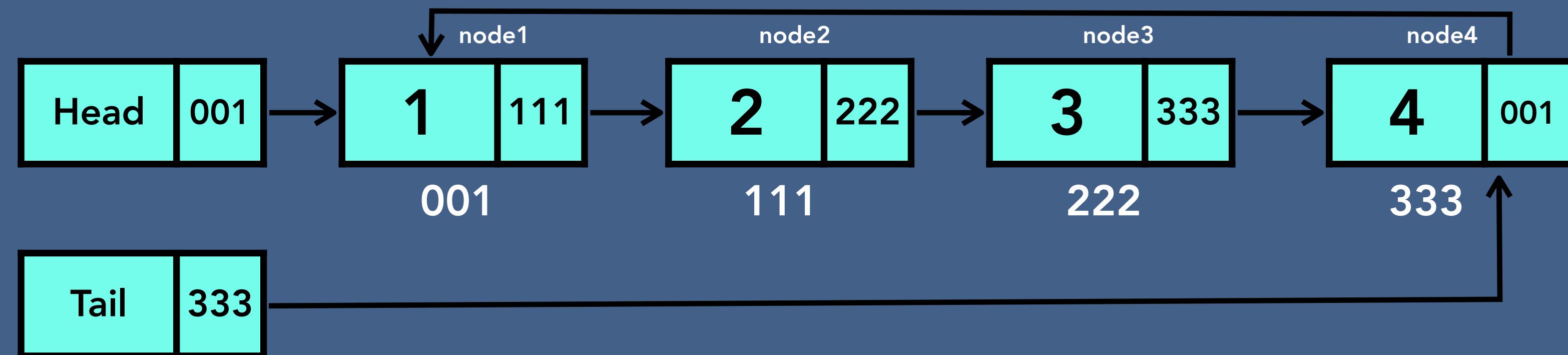


# Create - Circular Singly Linked List



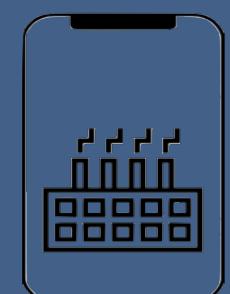
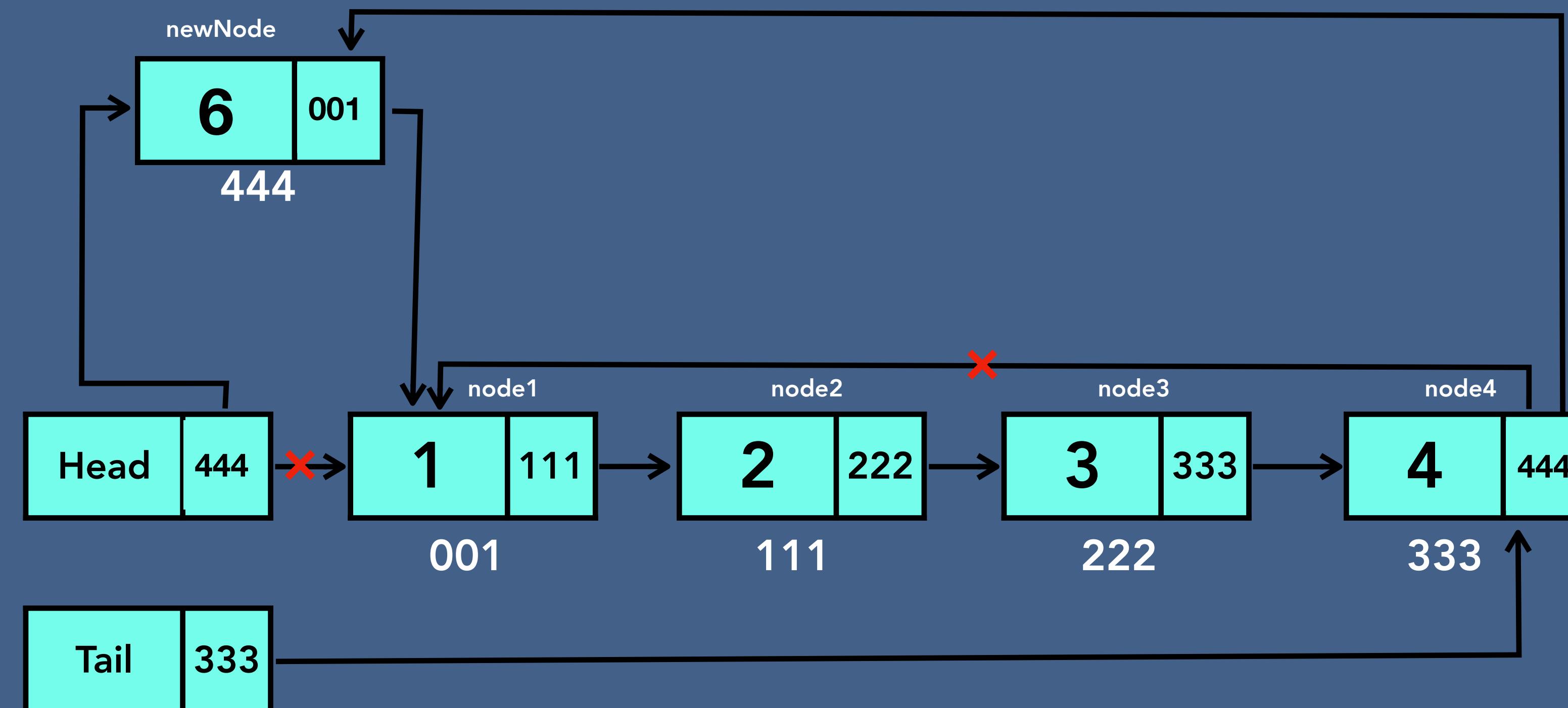
# Insertion - Circular Singly Linked List

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



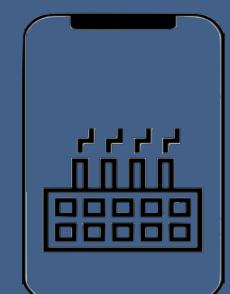
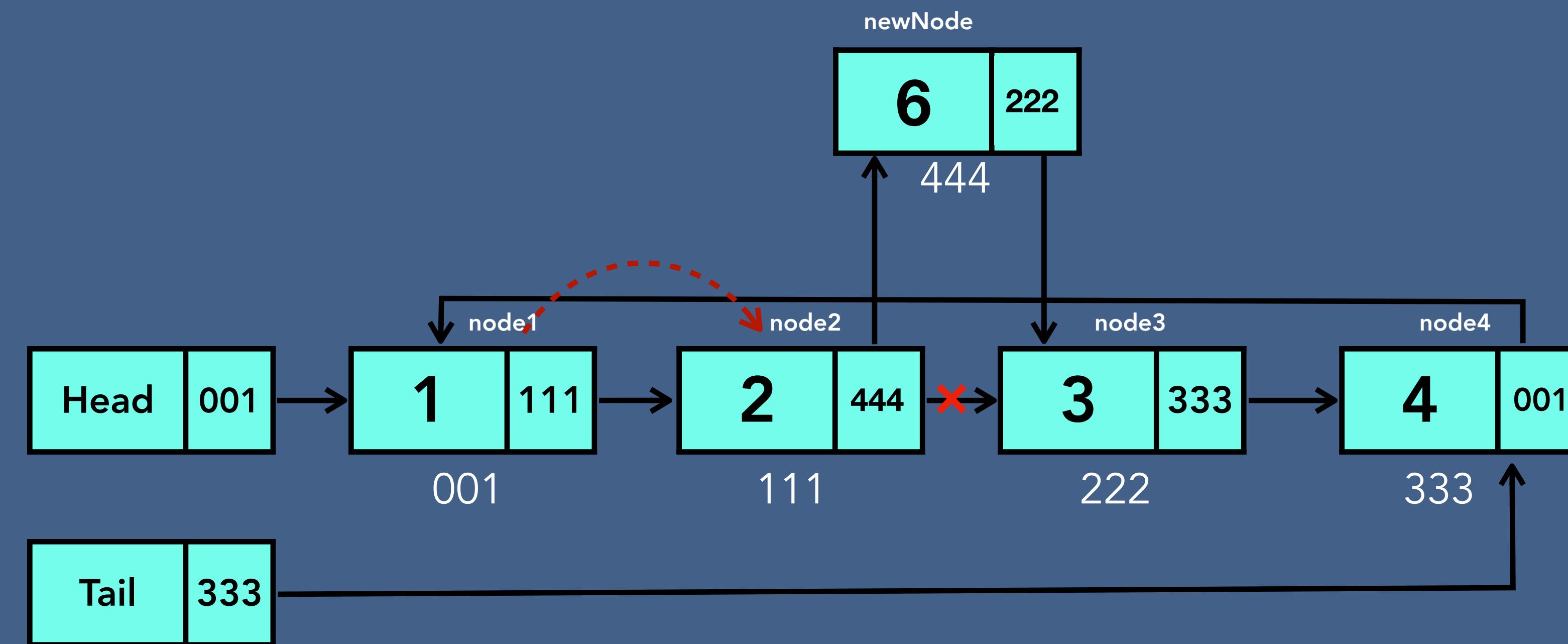
# Insertion - Circular Singly Linked List

- Insert at the beginning of linked list



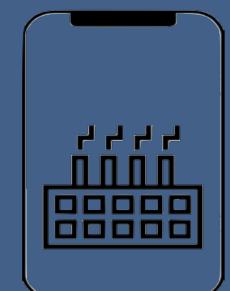
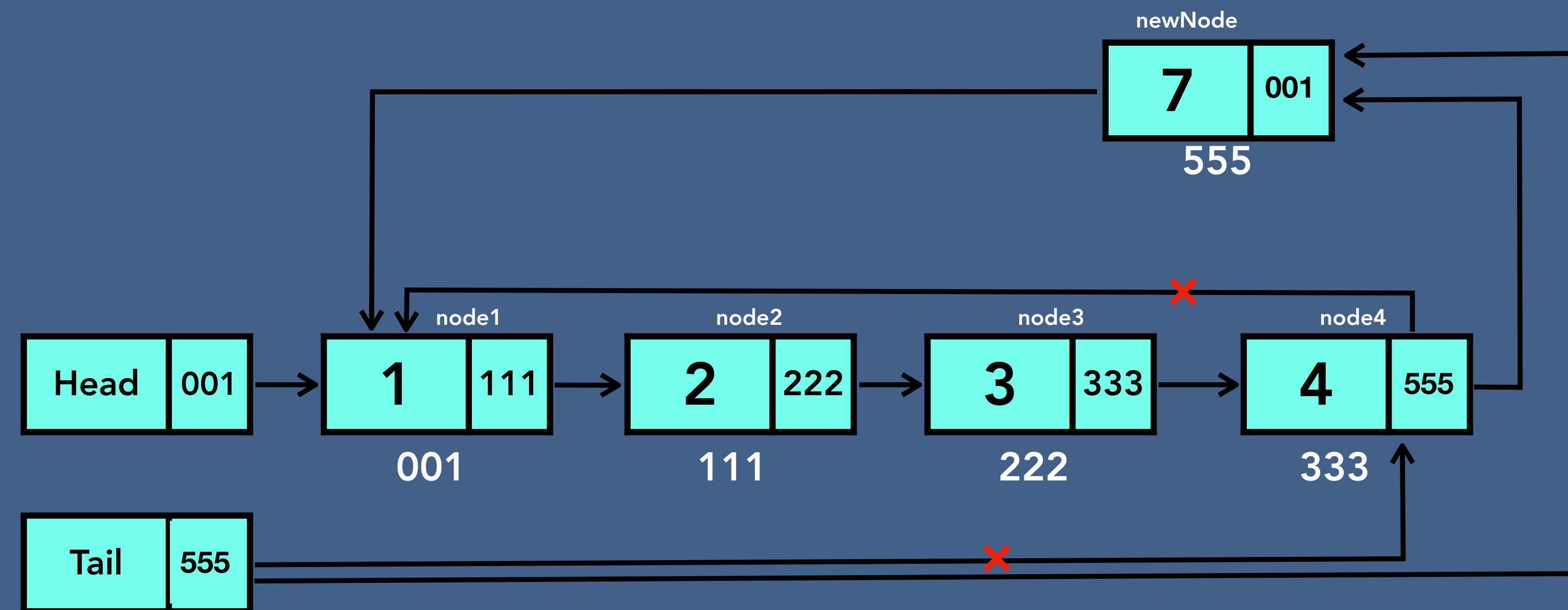
# Insertion - Circular Singly Linked List

- Insert at the specified location of linked list

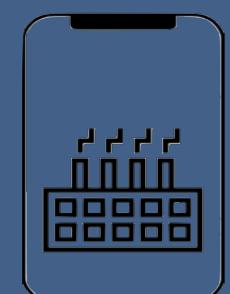
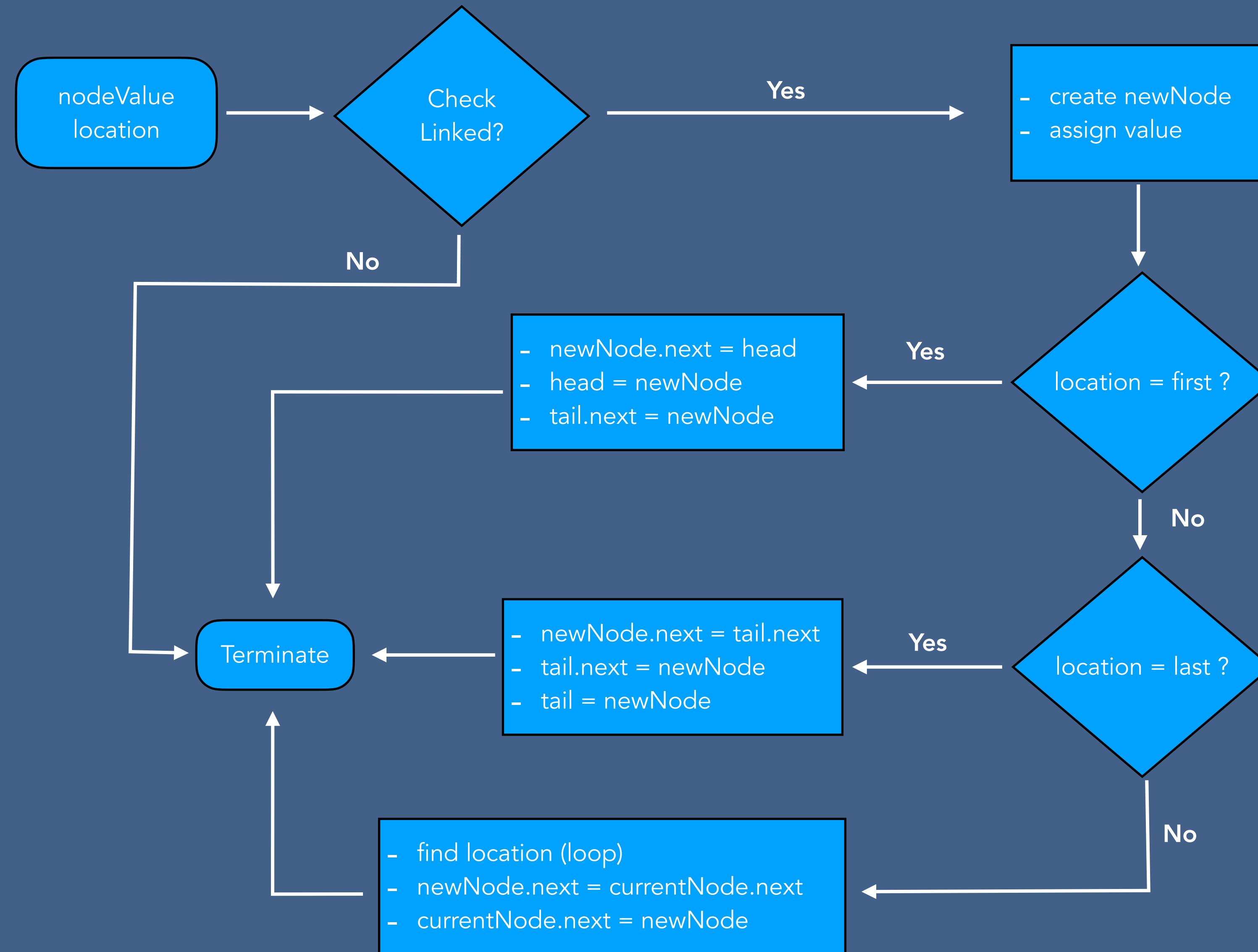


# Insertion - Circular Singly Linked List

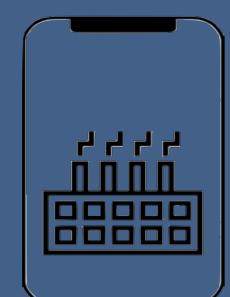
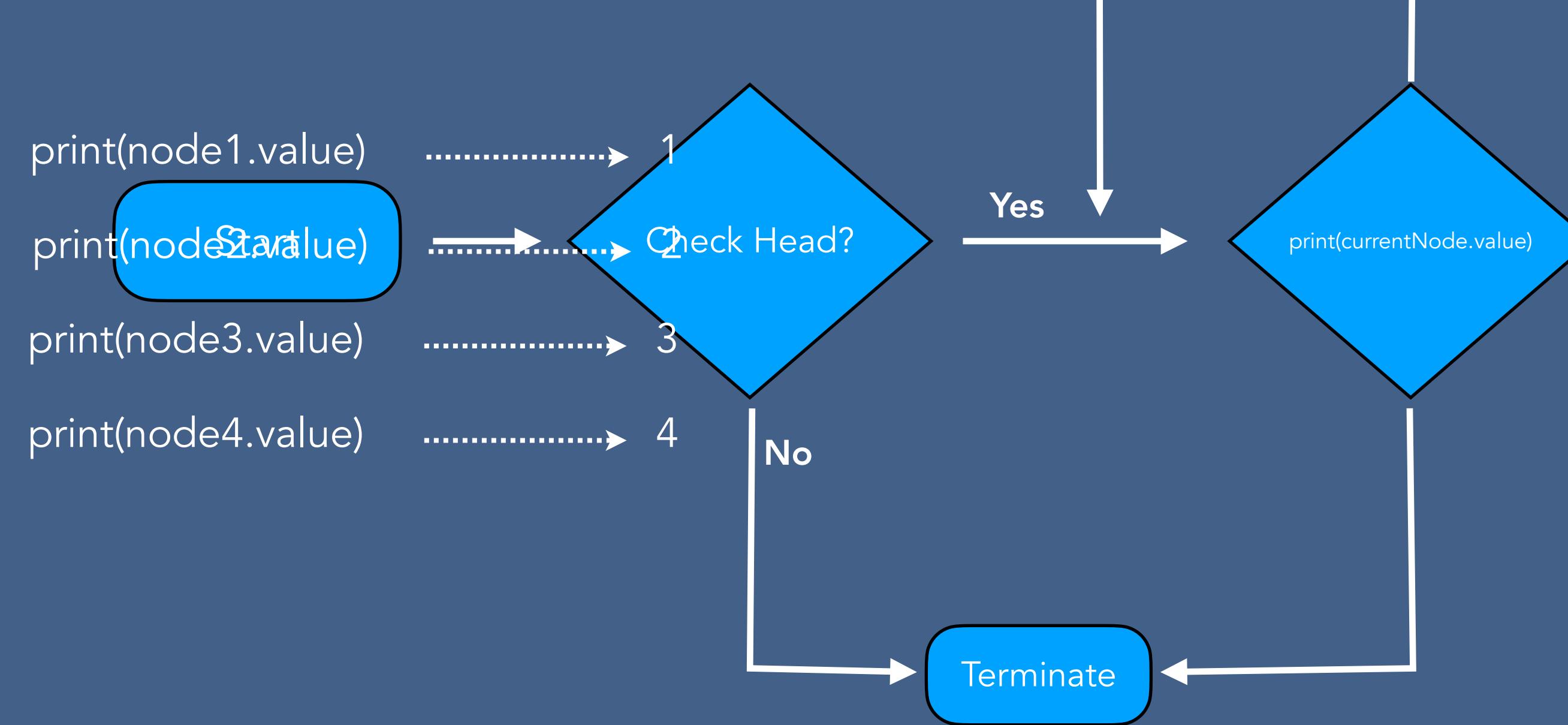
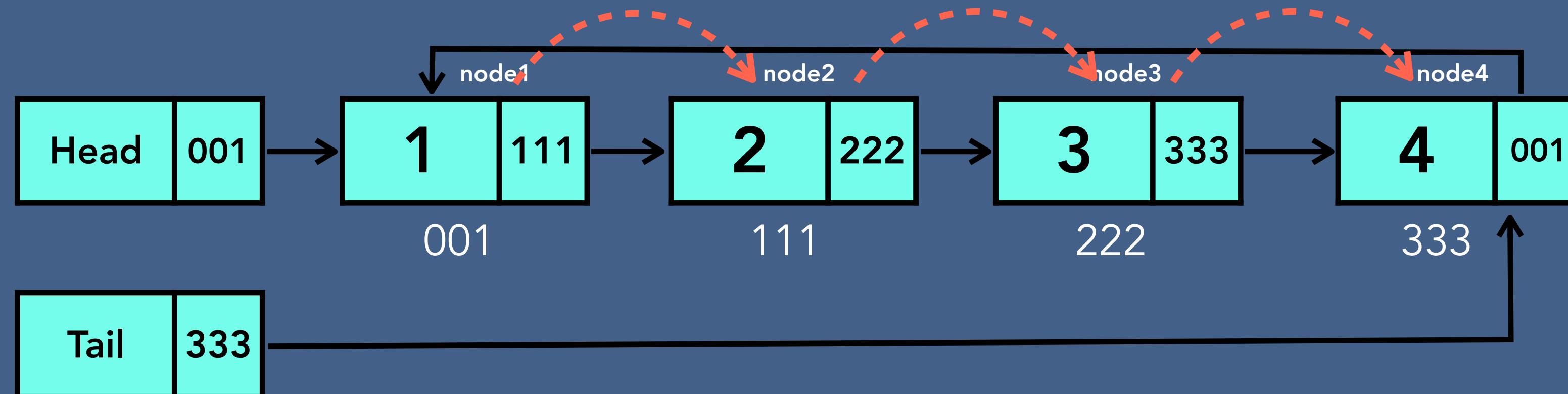
- Insert at the end of linked list



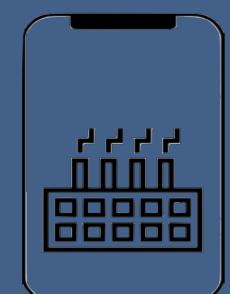
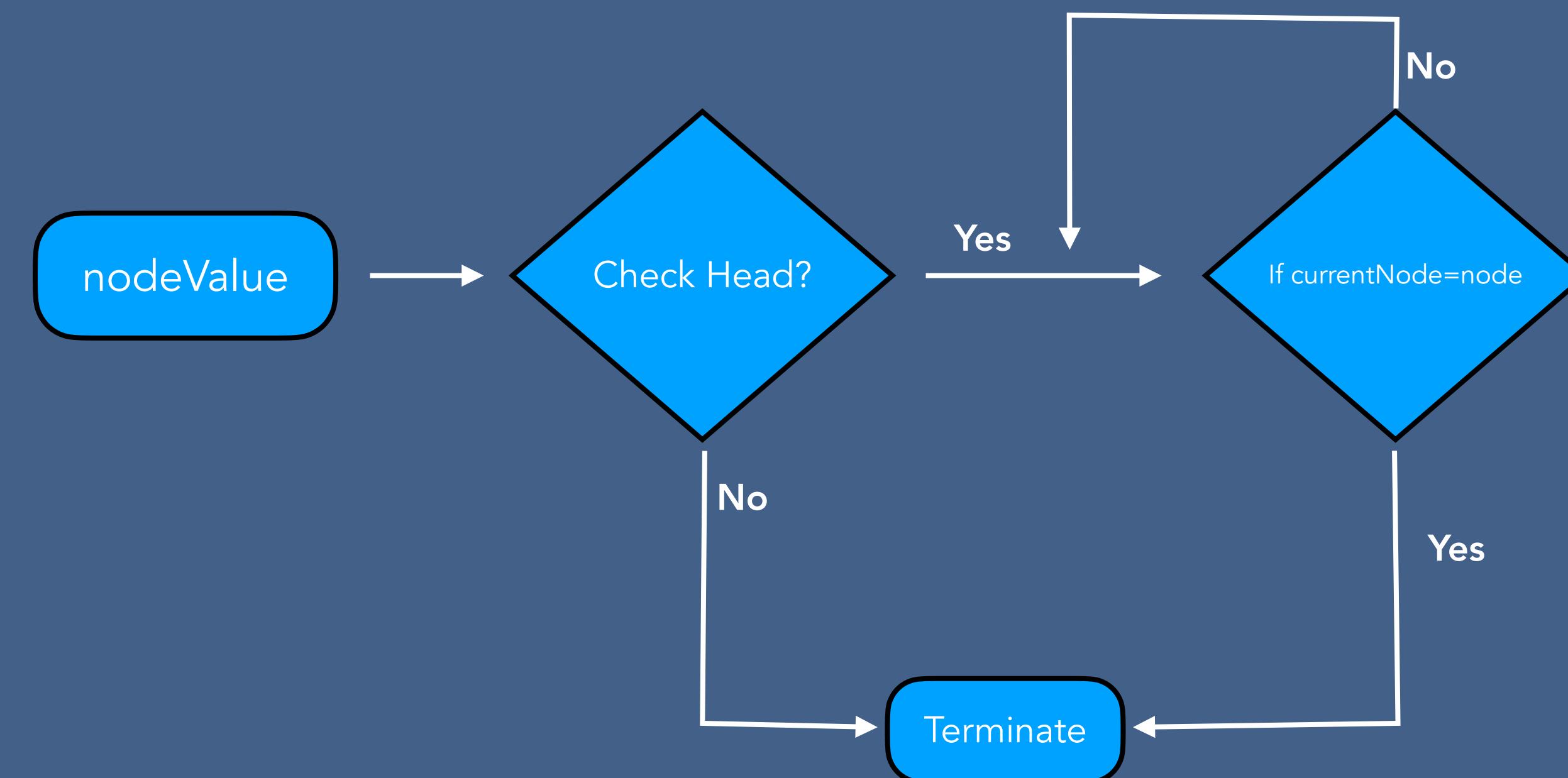
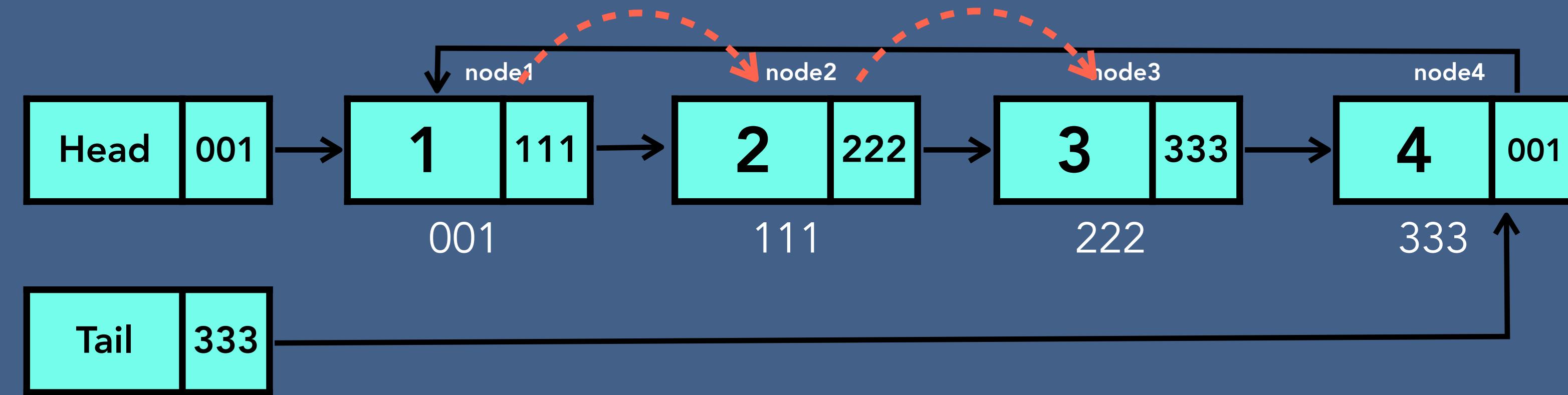
# Insertion Algorithm - Circular Singly Linked List



# Traversal - Circular Singly Linked List

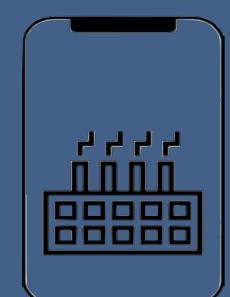
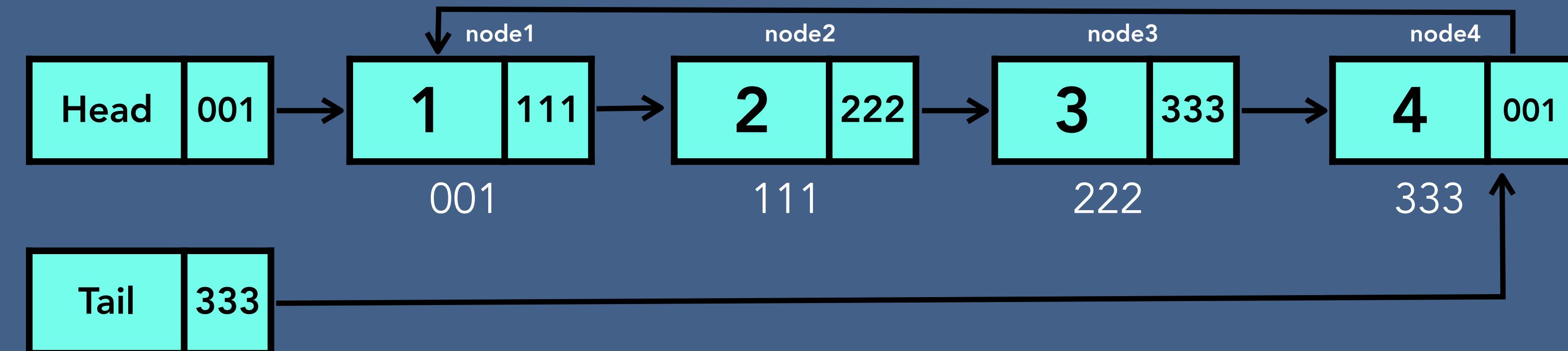


# Searching - Circular Singly Linked List



# Deletion - Circular Singly Linked List

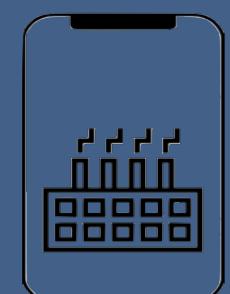
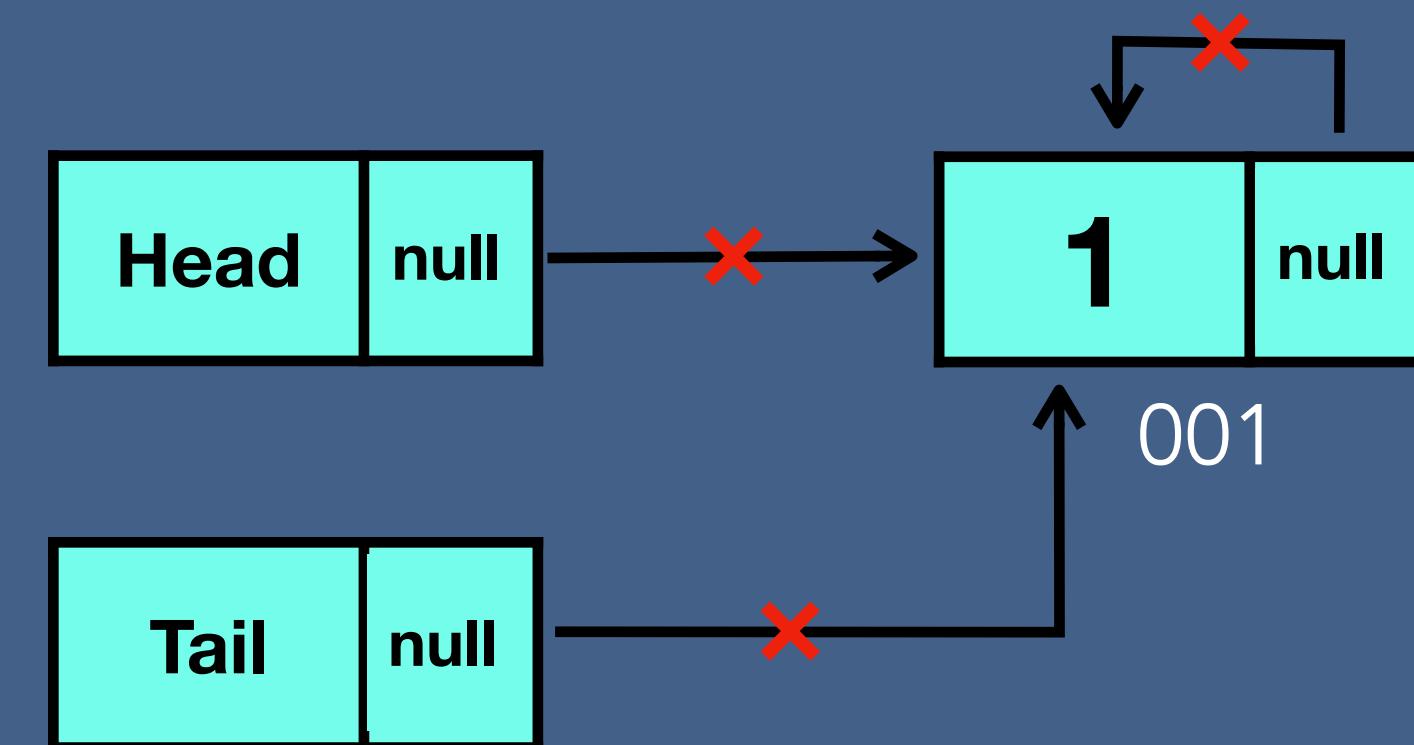
- Deleting the first node
- Deleting any given node
- Deleting the last node



# Deletion - Circular Singly Linked List

Deleting the first node

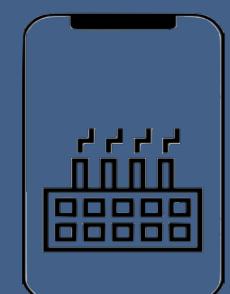
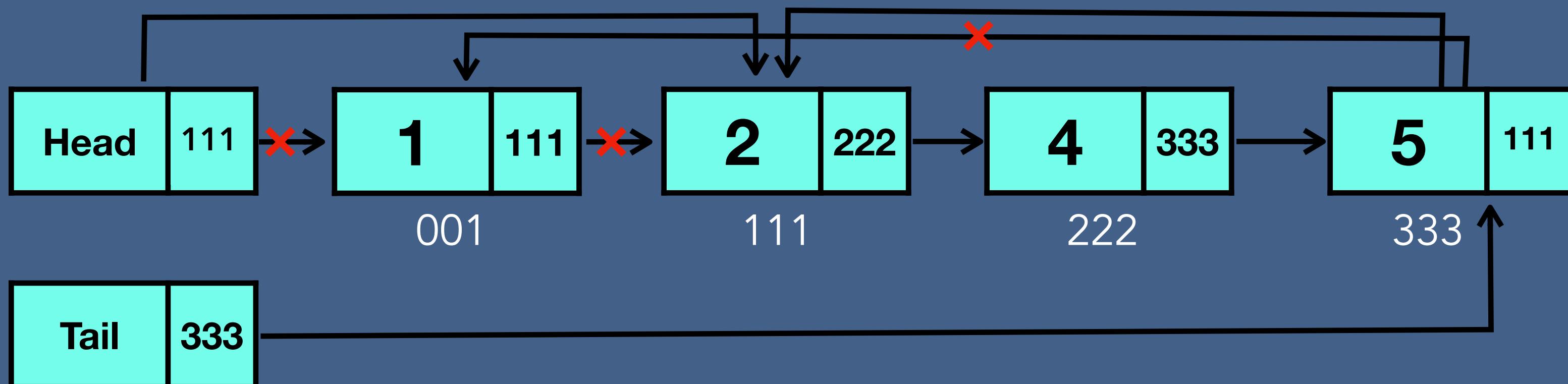
Case 1 - one node



# Deletion - Circular Singly Linked List

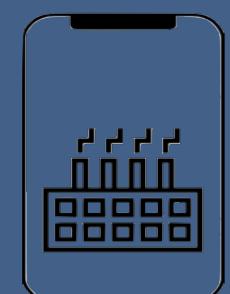
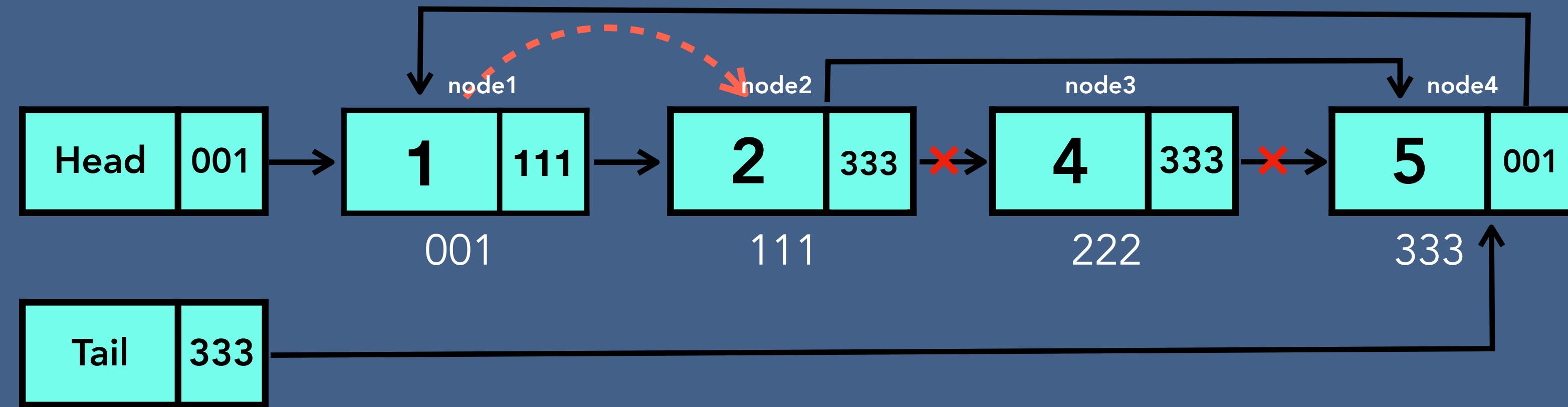
Deleting the first node

Case 2 - more than one node



# Deletion - Circular Singly Linked List

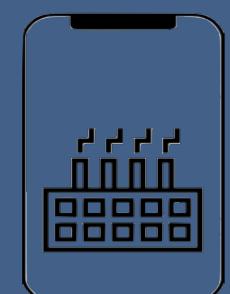
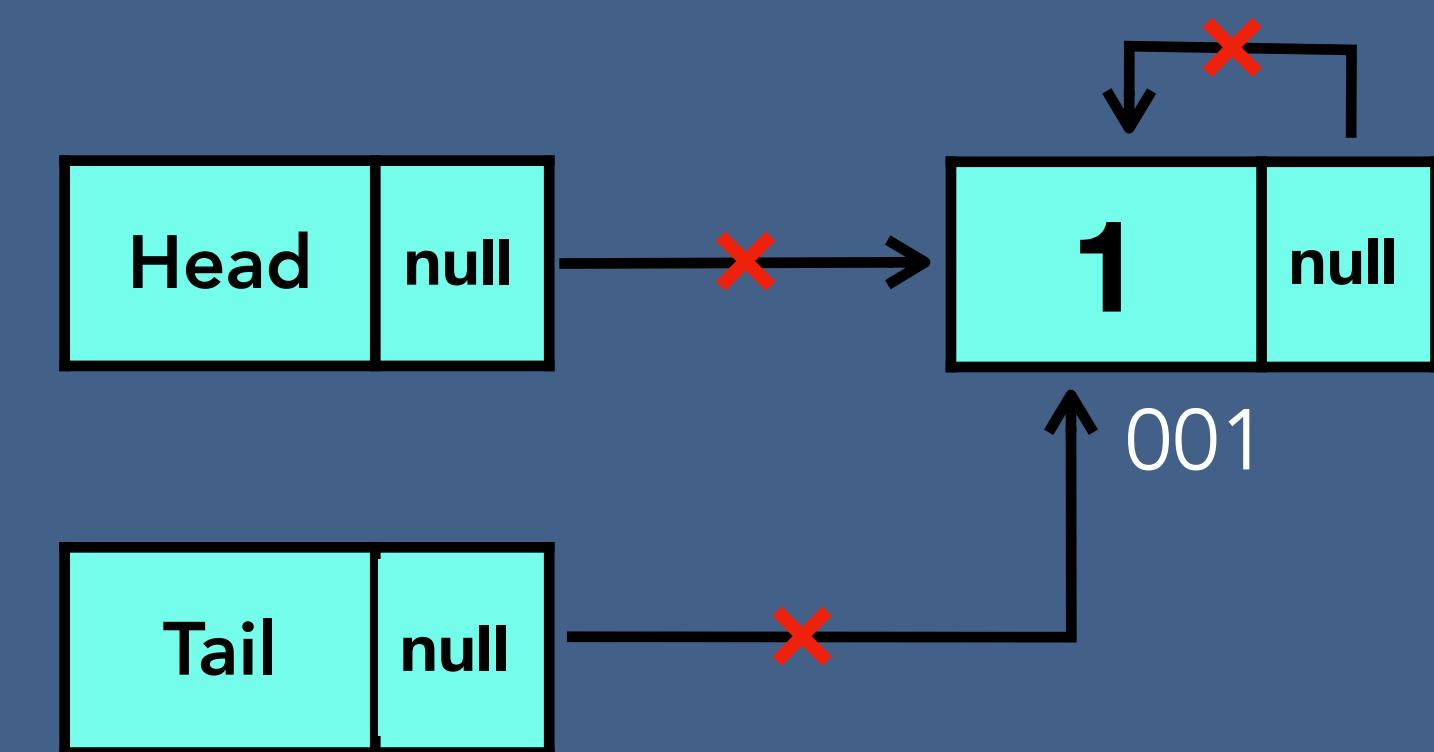
Deleting any given node



# Deletion - Circular Singly Linked List

Deleting the last node

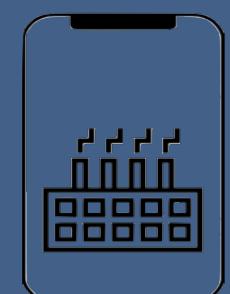
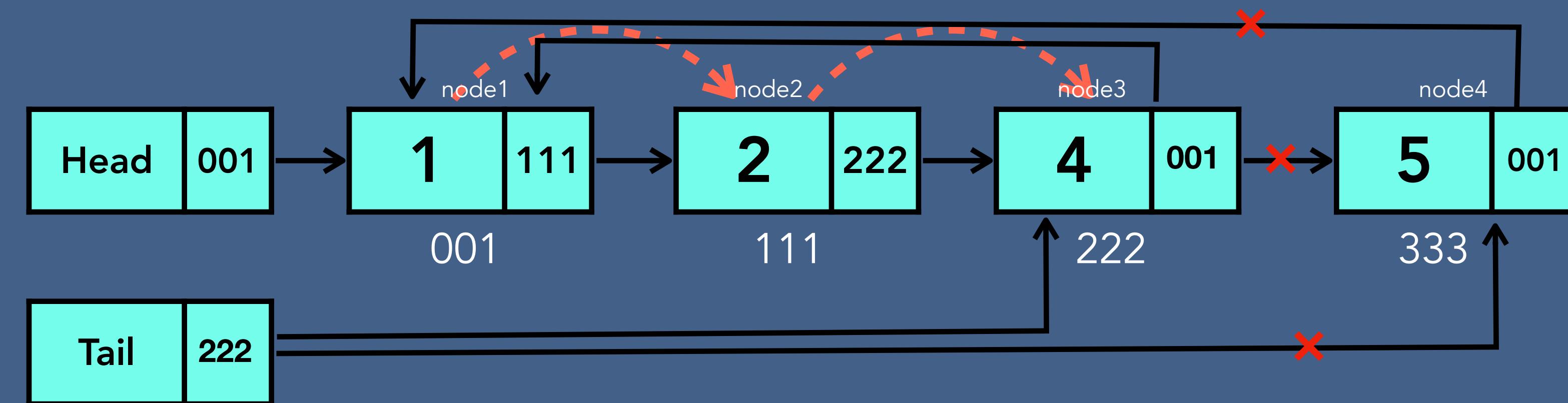
Case 1 - one node



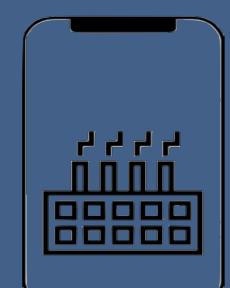
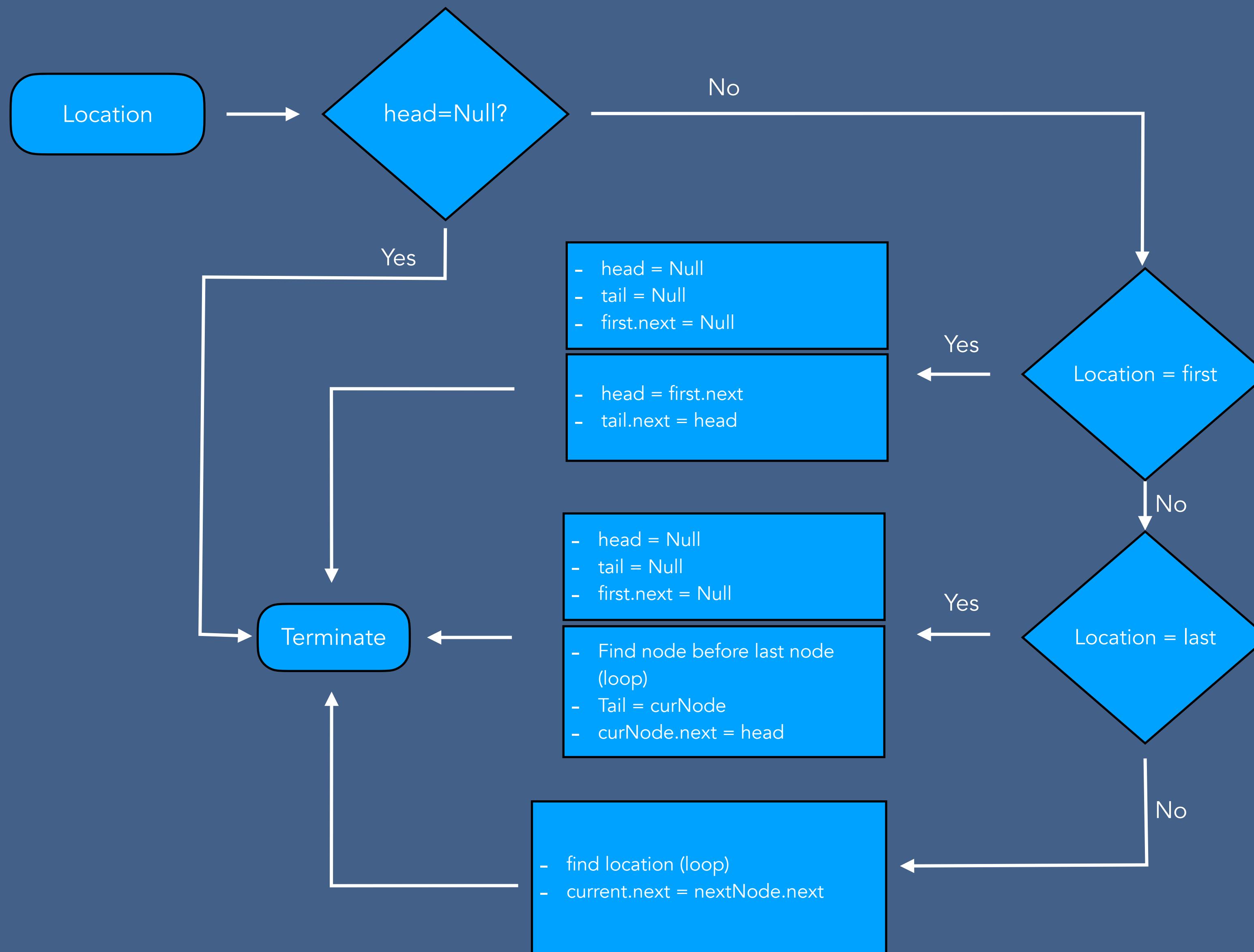
# Deletion - Circular Singly Linked List

Deleting the last node

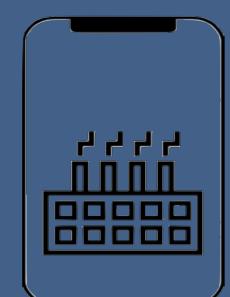
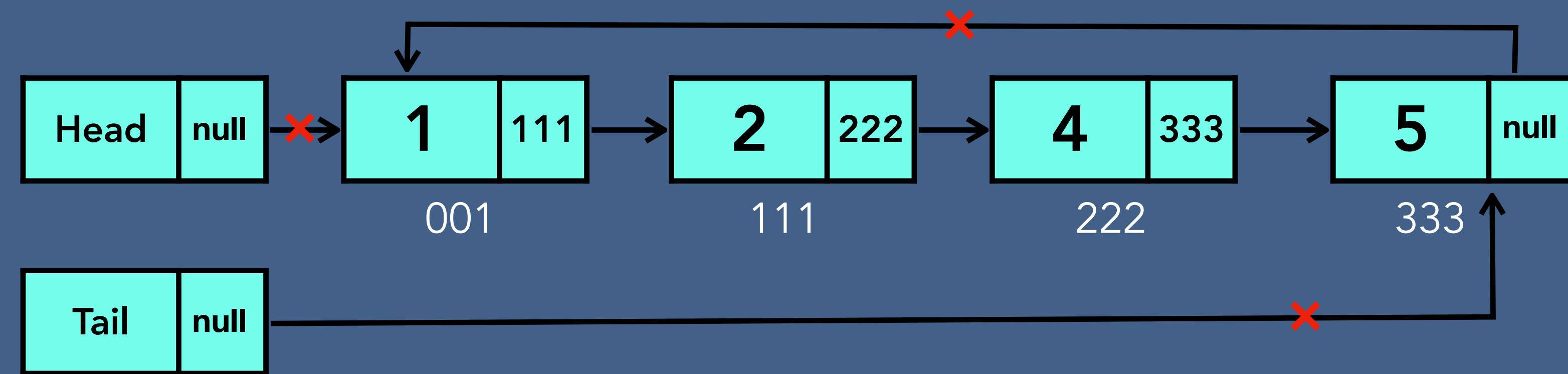
Case 2 - more than one node



# Deletion Algorithm - Circular Singly Linked List

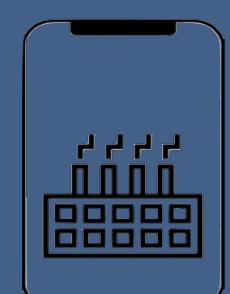
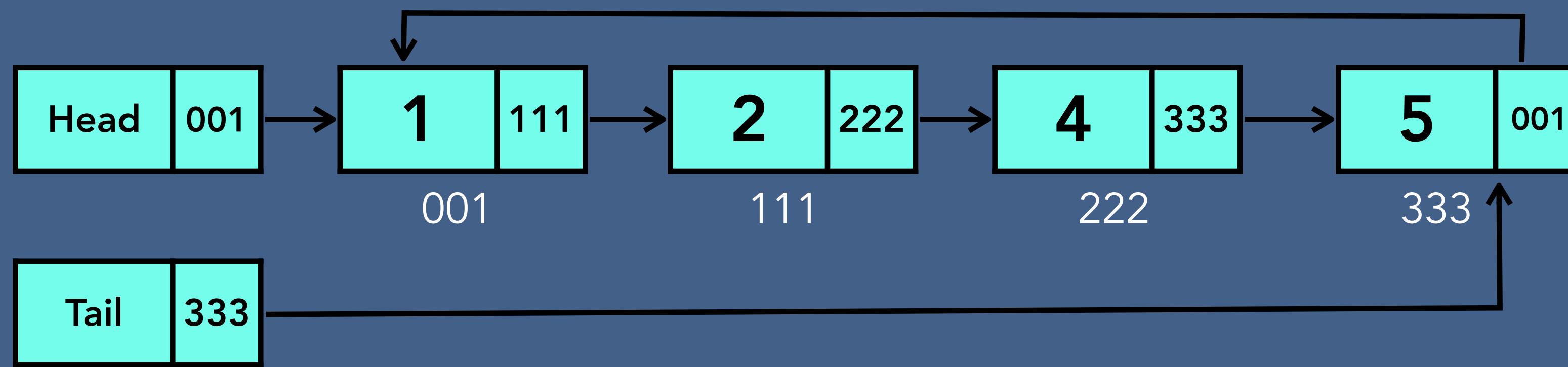


# Delete Entire Circular Singly Linked List

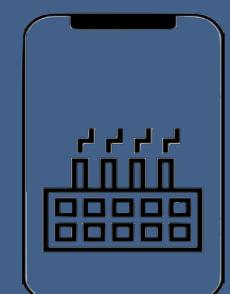


# Time and Space Complexity of Circular Singly Linked List

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 CSLL	$O(1)$	$O(1)$

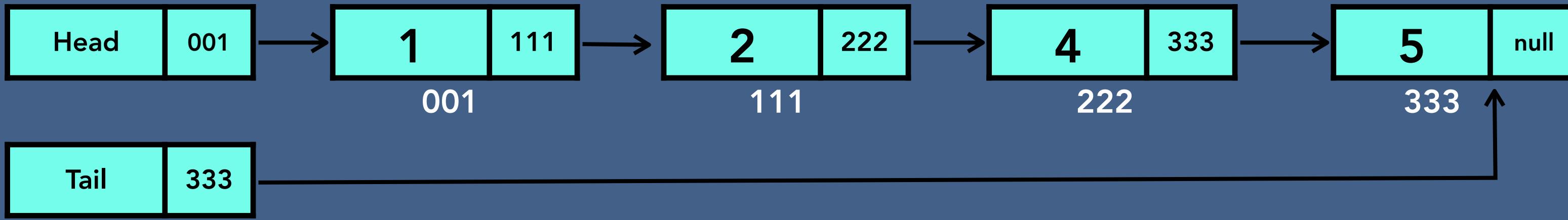


# Doubly Linked List

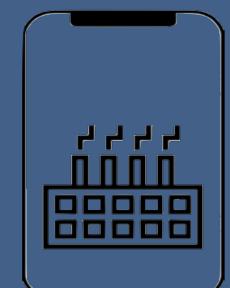
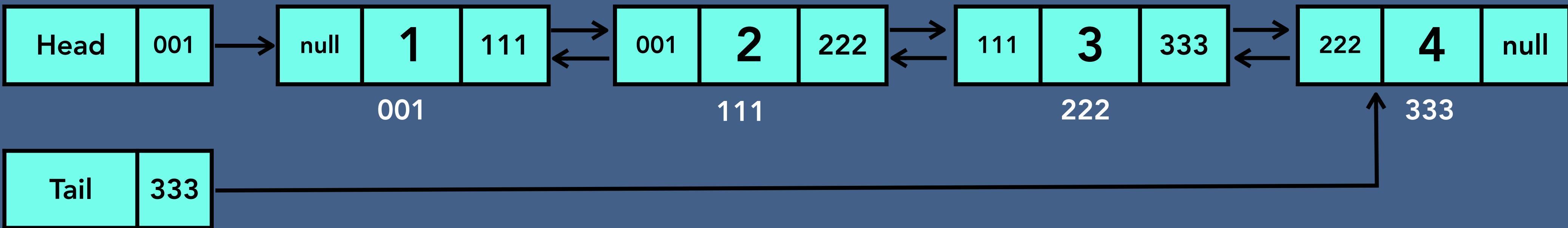


# Doubly Linked List

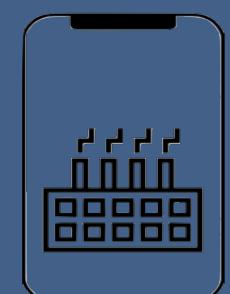
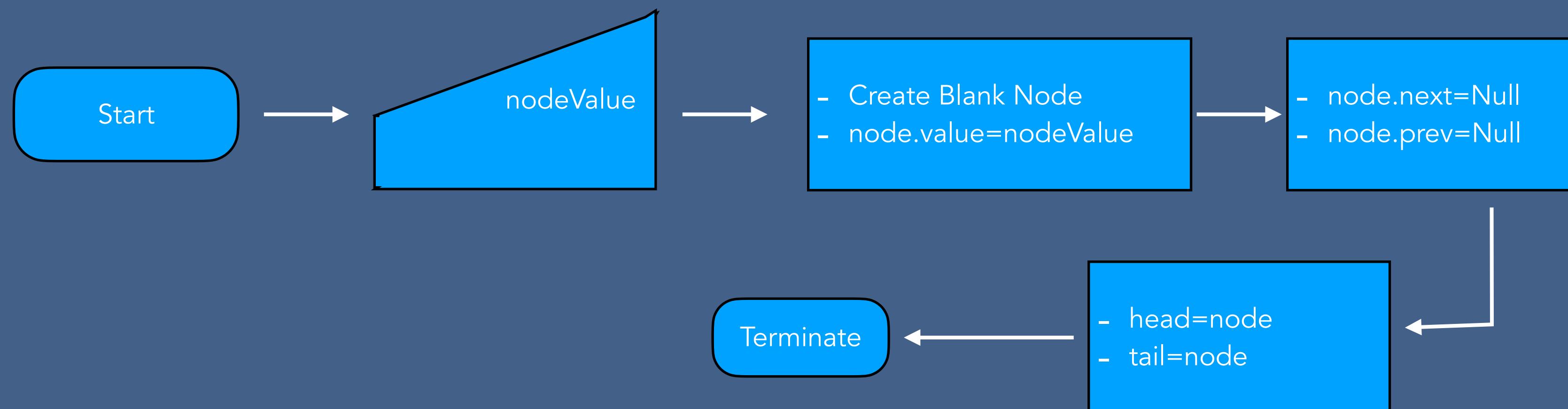
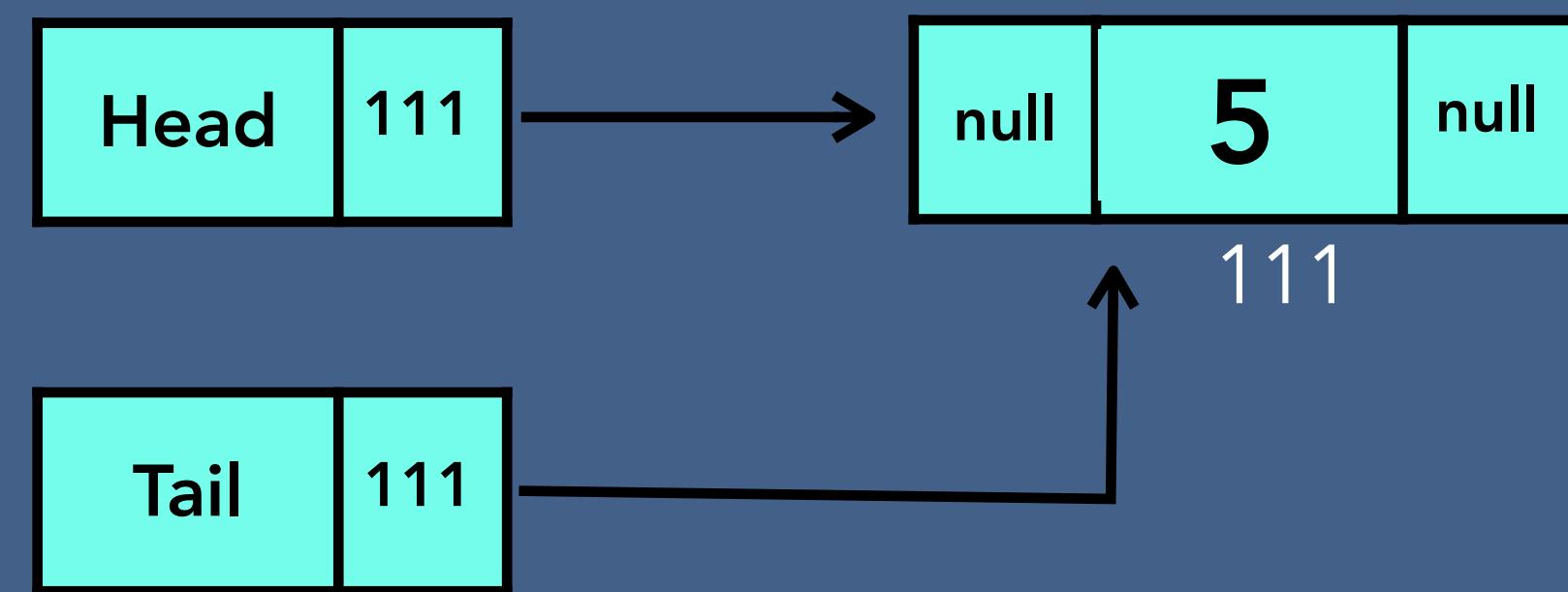
Singly Linked List



Doubly Linked List

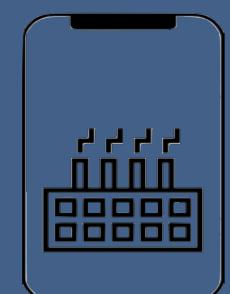
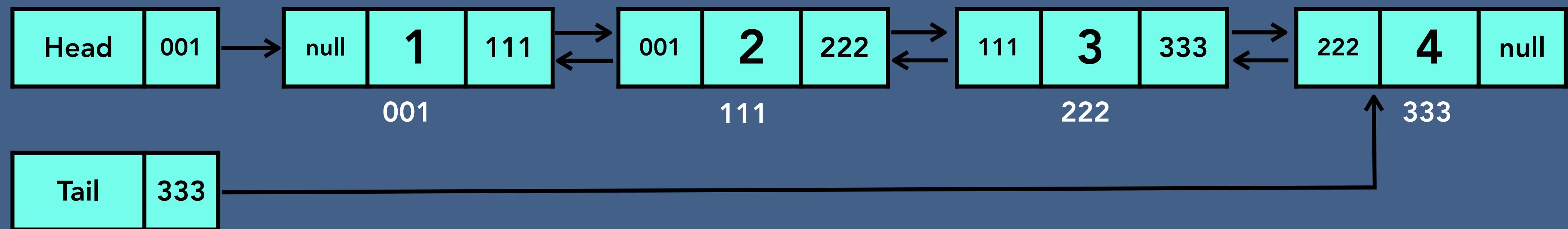


# Create - Doubly Linked List



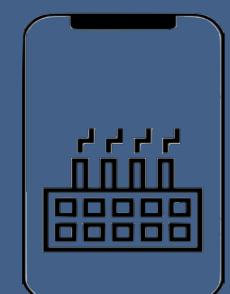
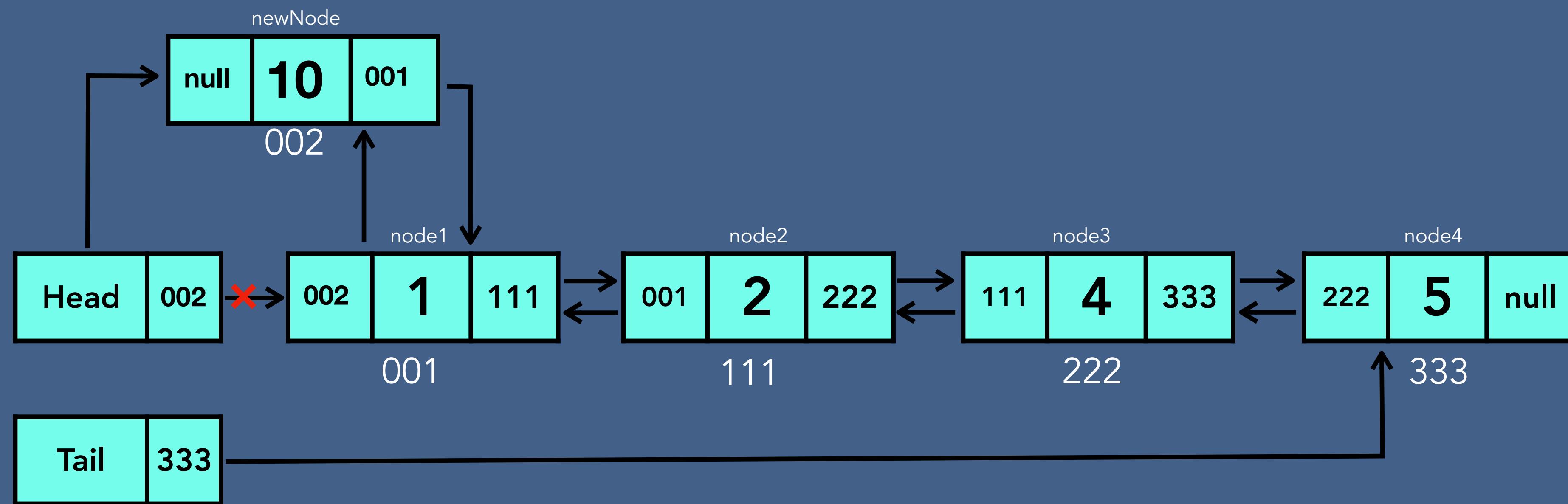
# Insertion - Doubly Linked List

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



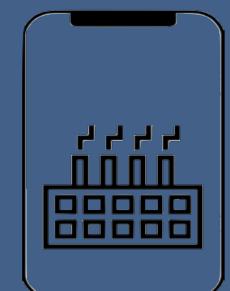
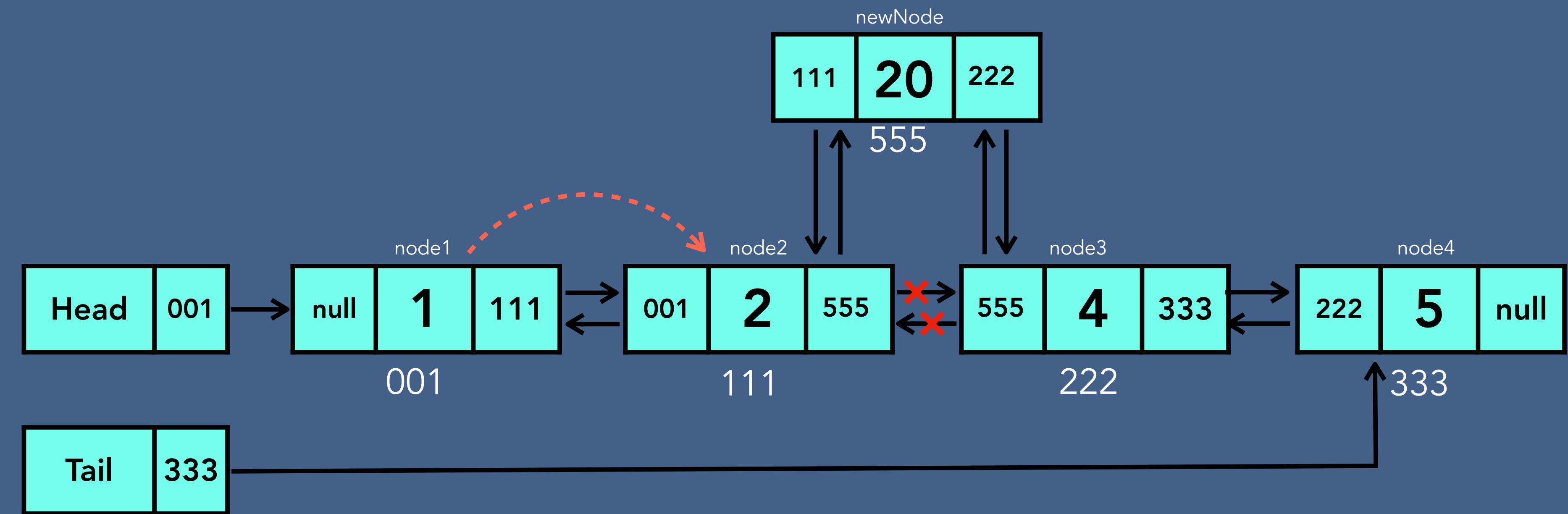
# Insertion - Doubly Linked List

- Insert at the beginning of linked list



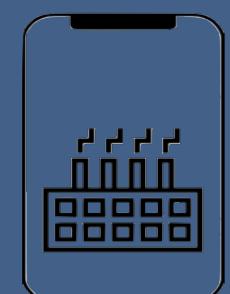
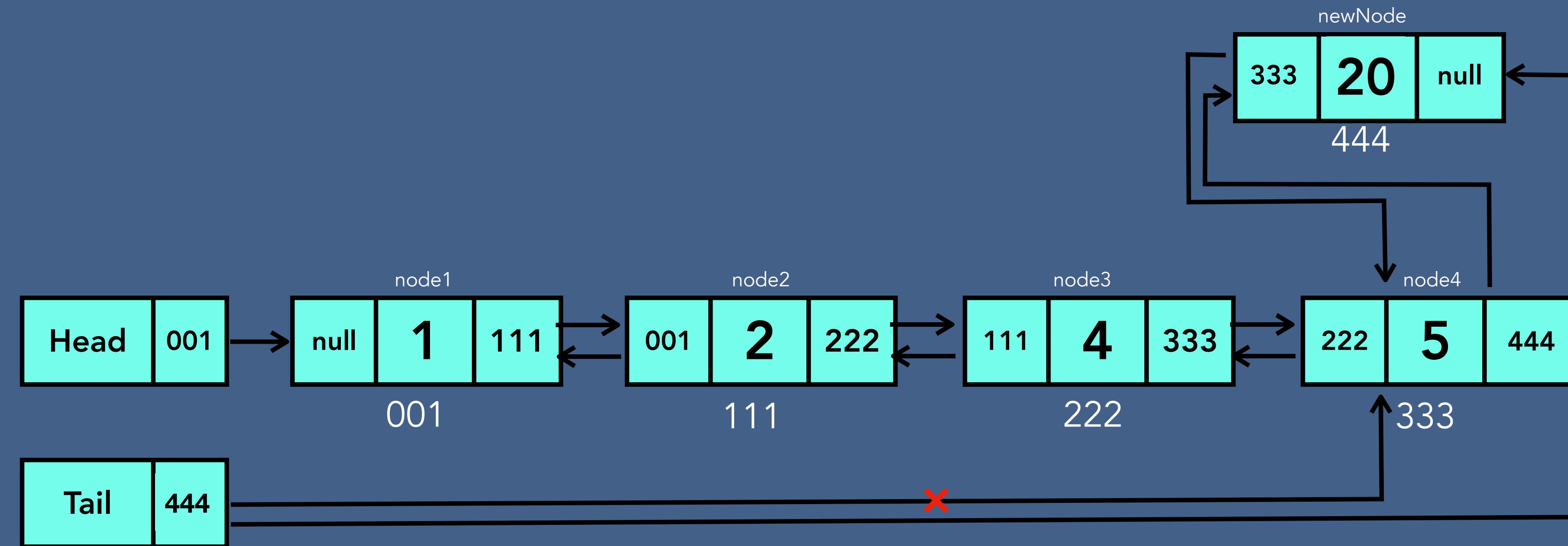
# Insertion - Doubly Linked List

- Insert at the specified location of linked list

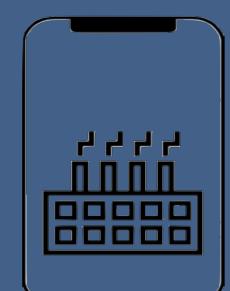
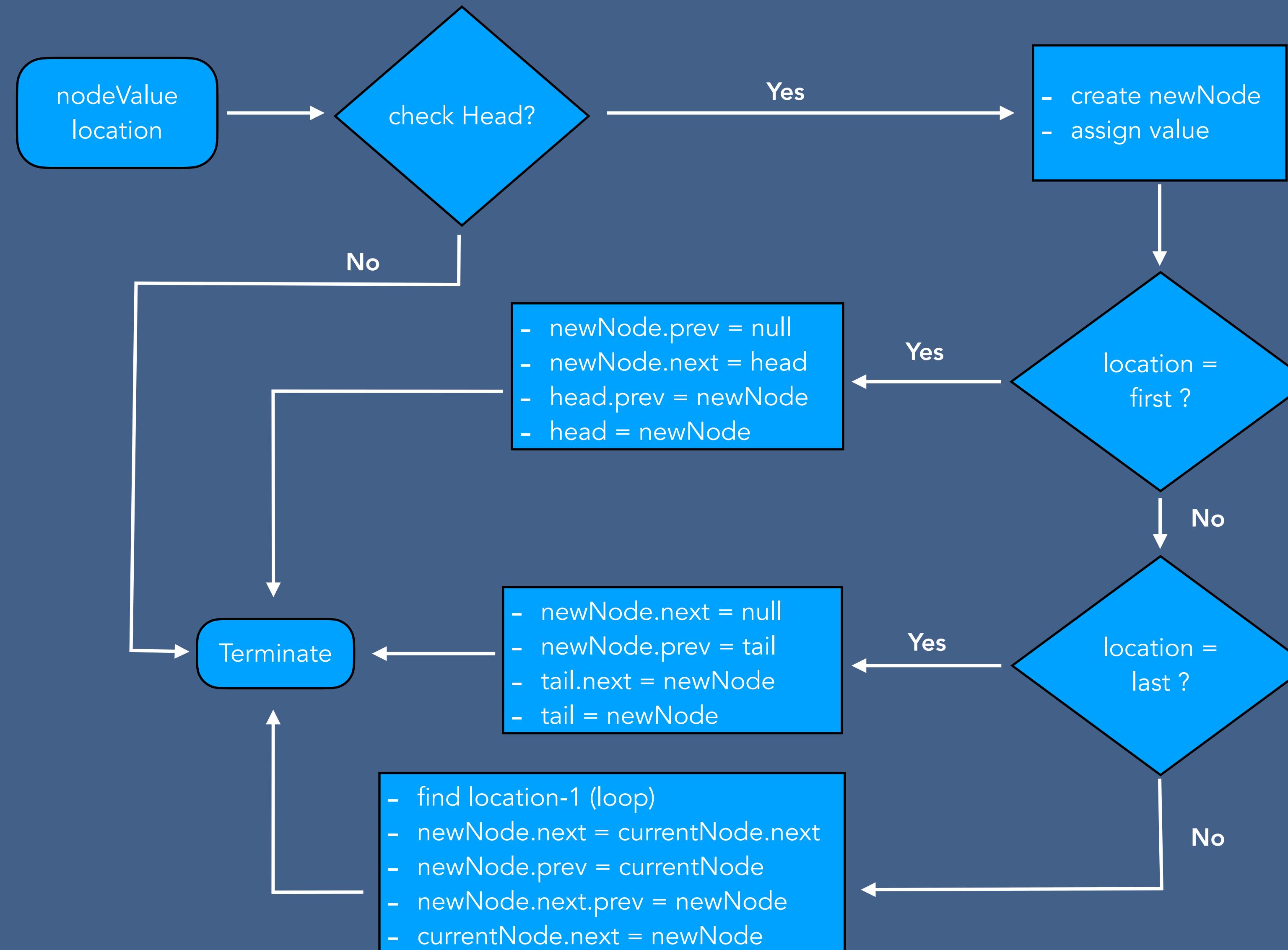


# Insertion - Doubly Linked List

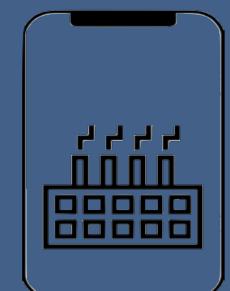
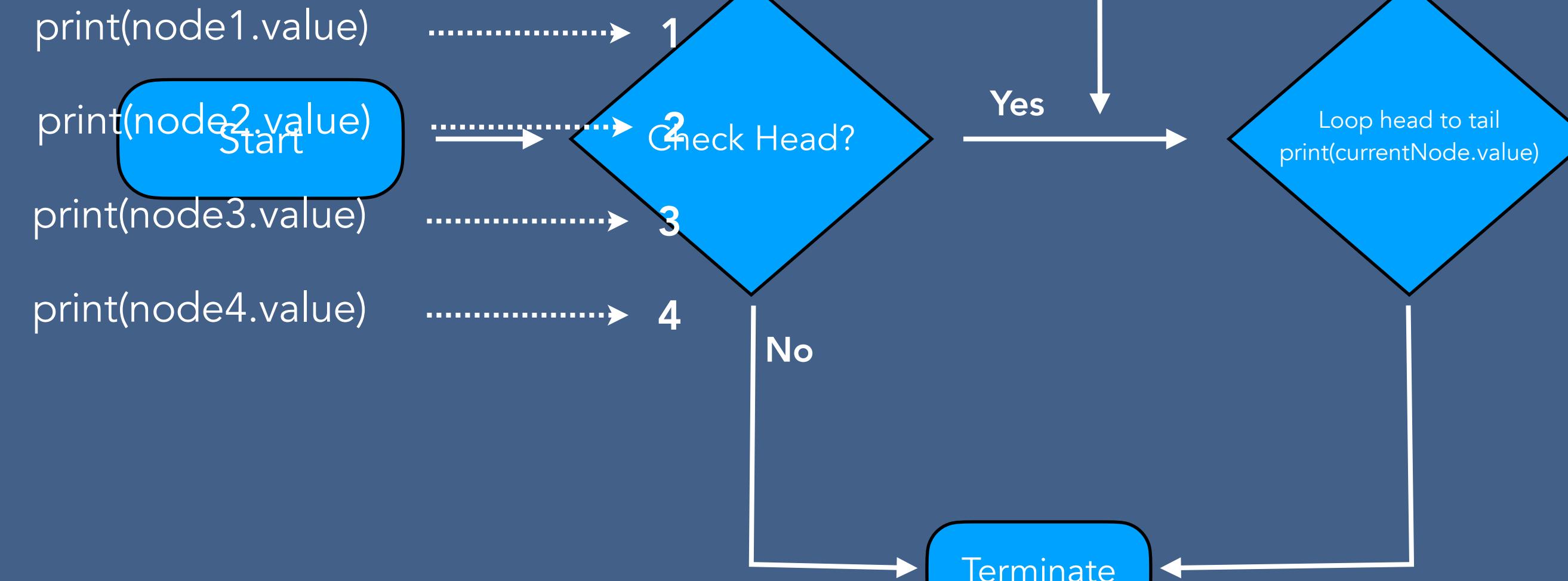
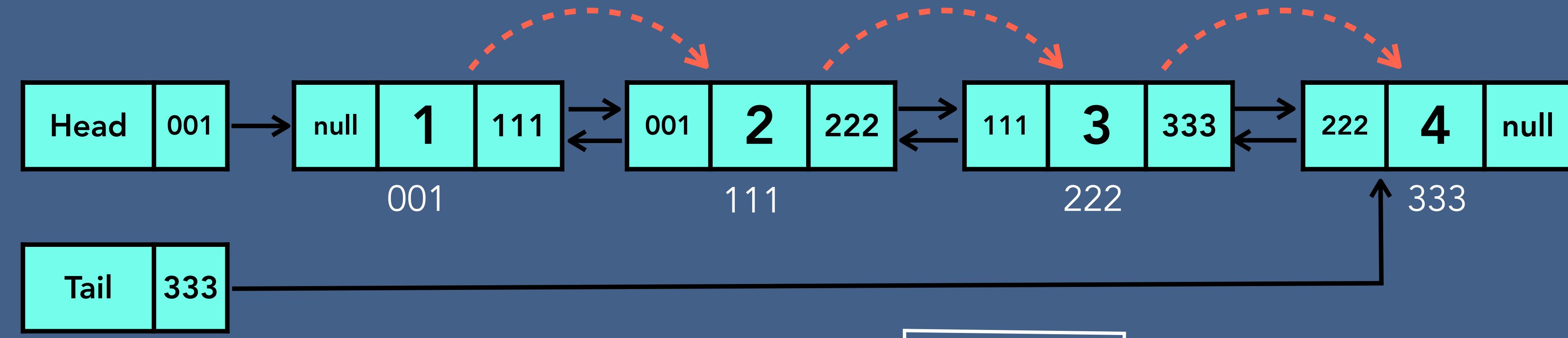
- Insert at the end of linked list



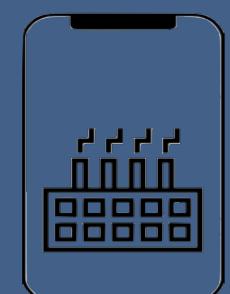
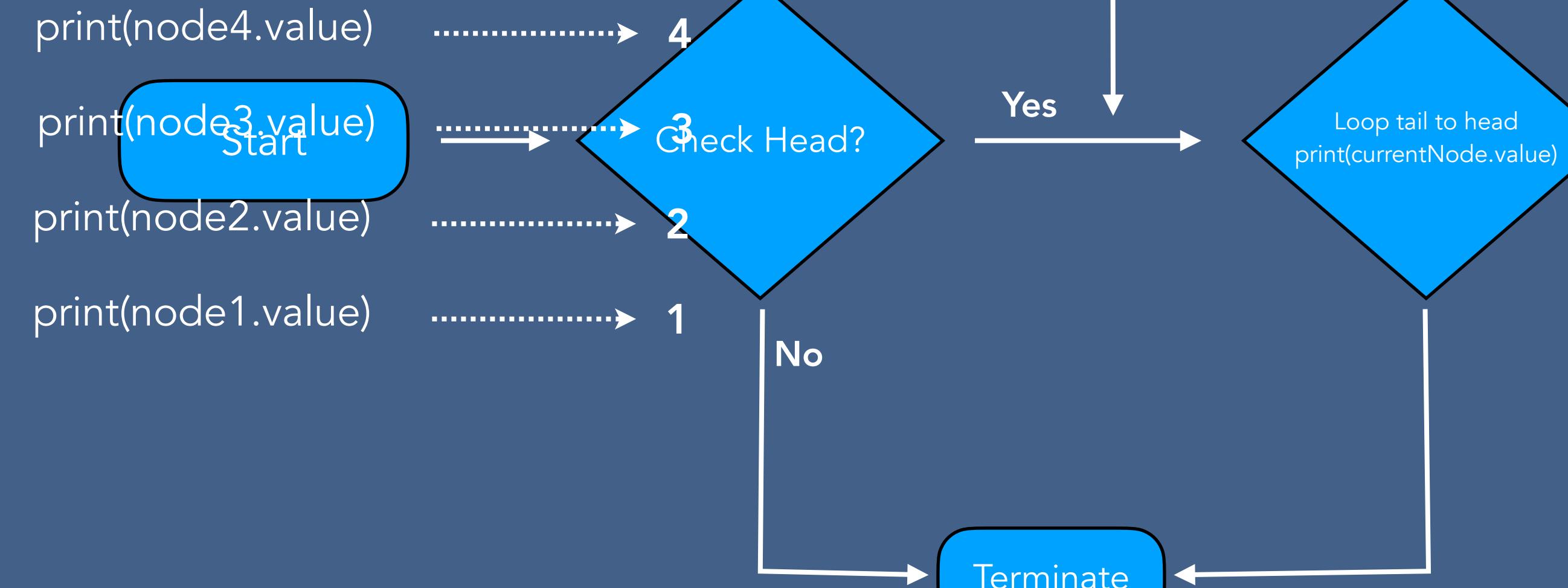
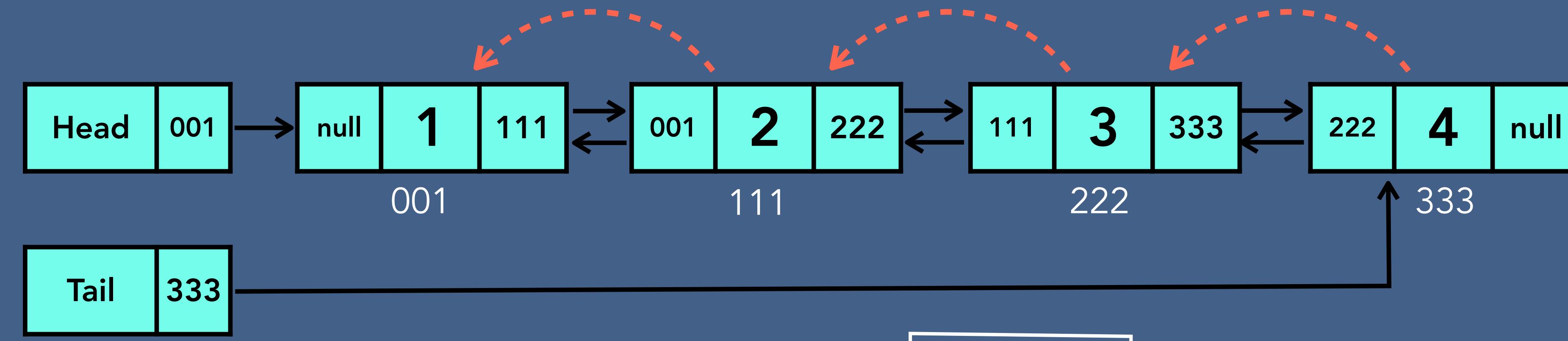
# Insertion Algorithm - Doubly Linked List



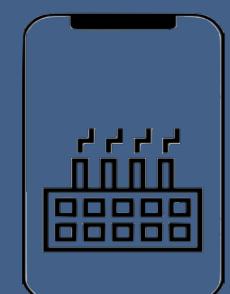
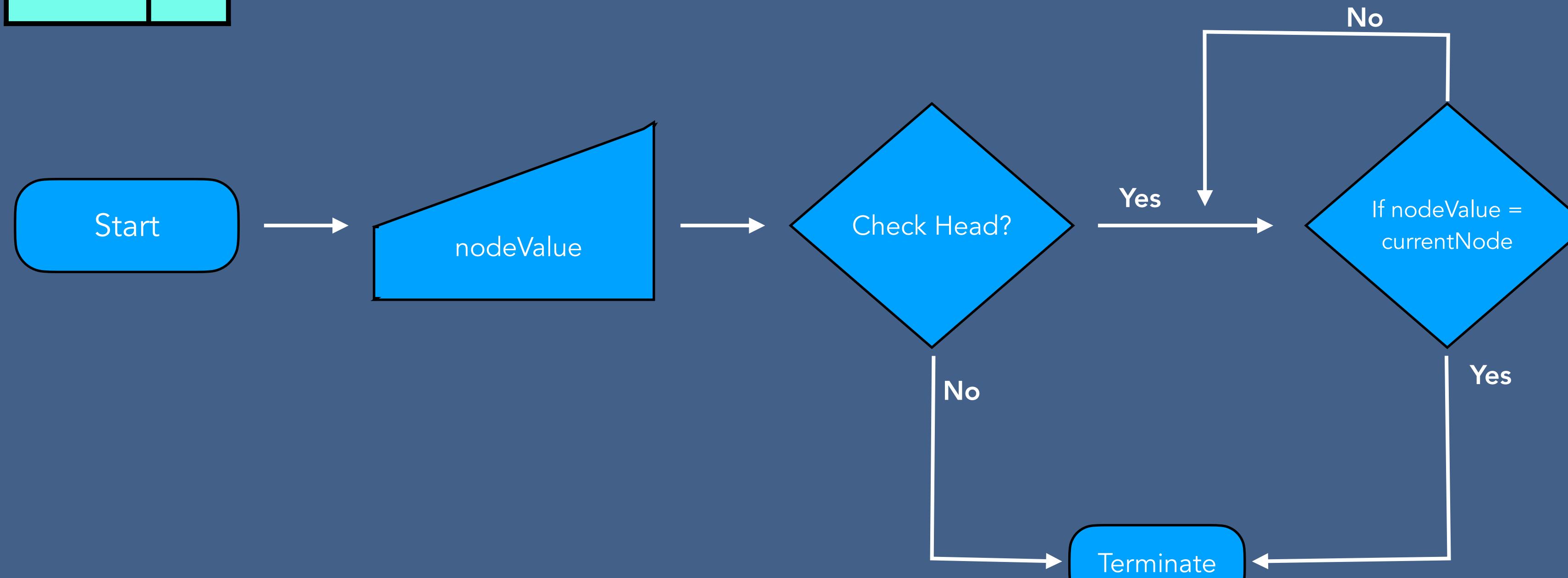
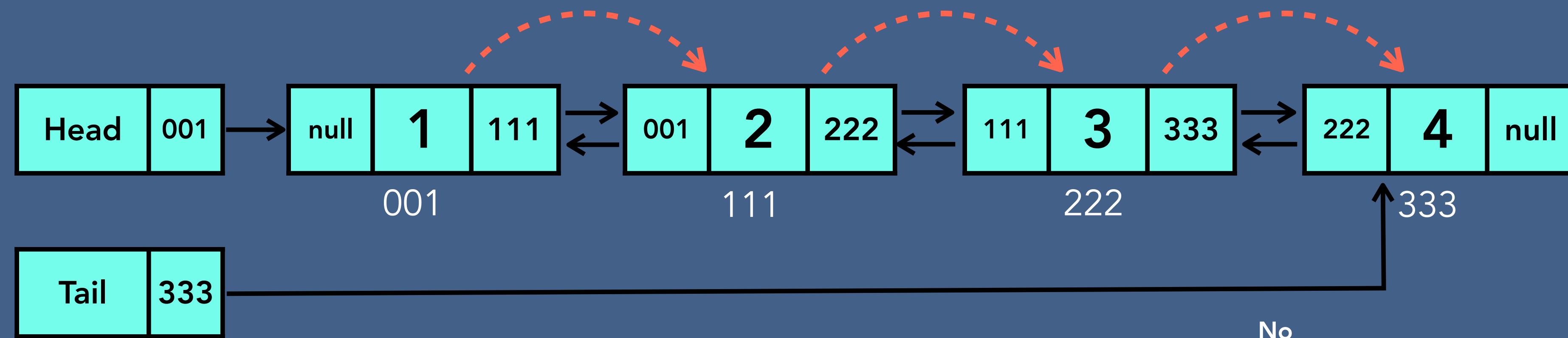
# Traversal - Doubly Linked List



# Reverse Traversal - Doubly Linked List

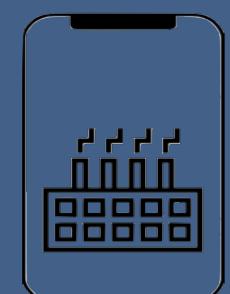
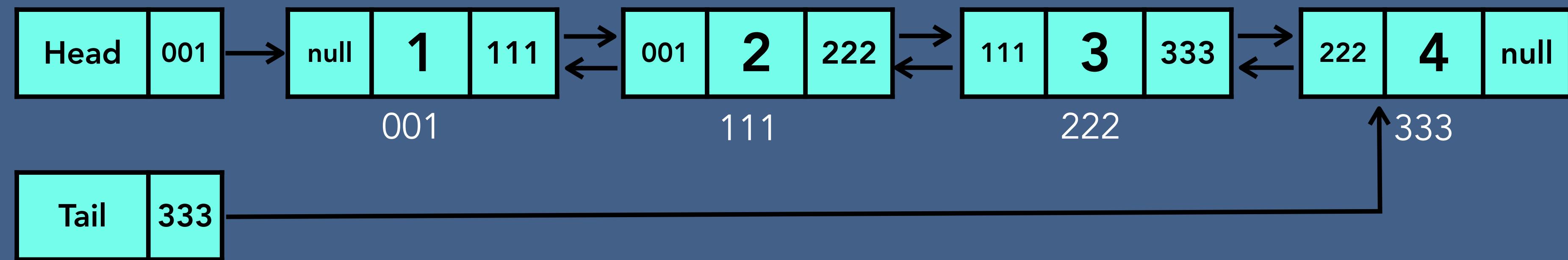


# Searching- Doubly Linked List



# Deletion - Doubly Linked List

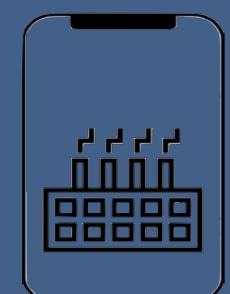
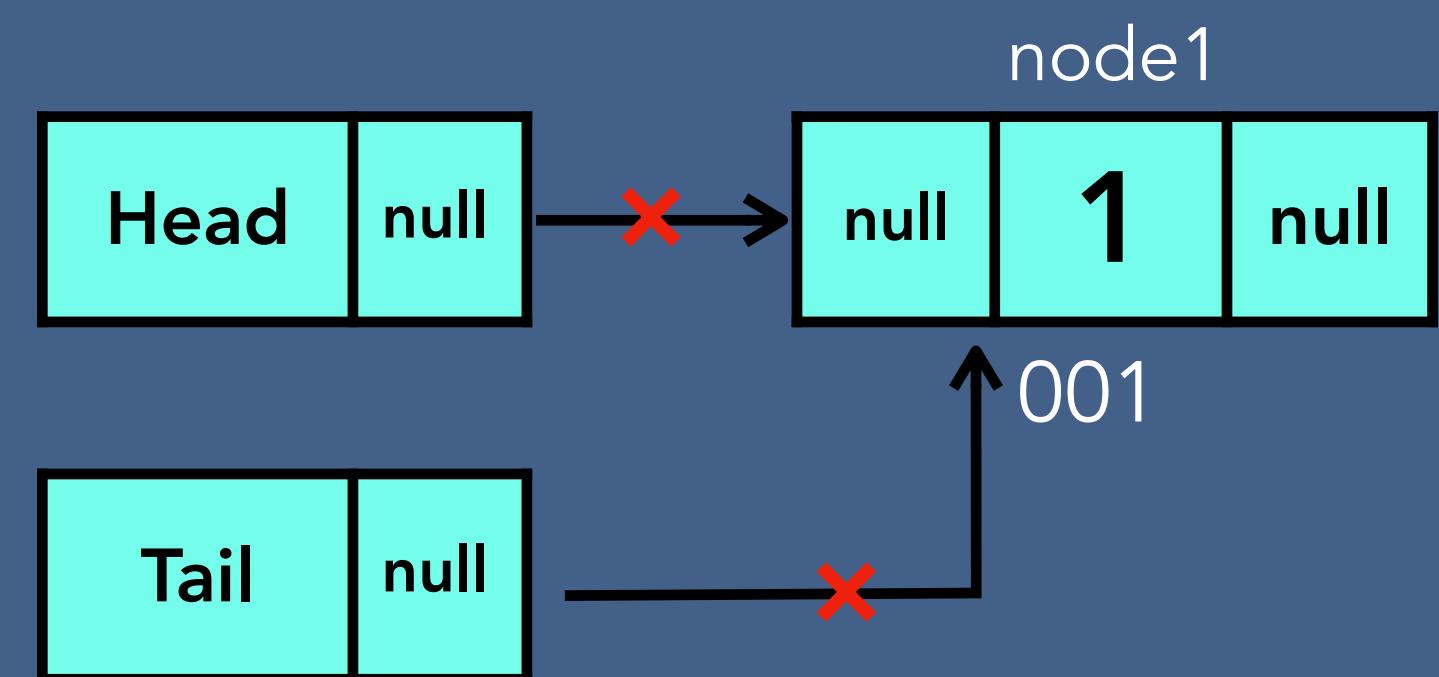
- Deleting the first node
- Deleting any given node
- Deleting the last node



# Deletion - Doubly Linked List

Deleting the first node

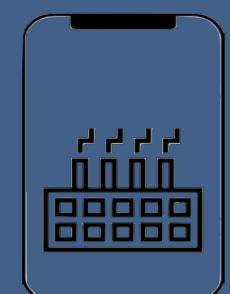
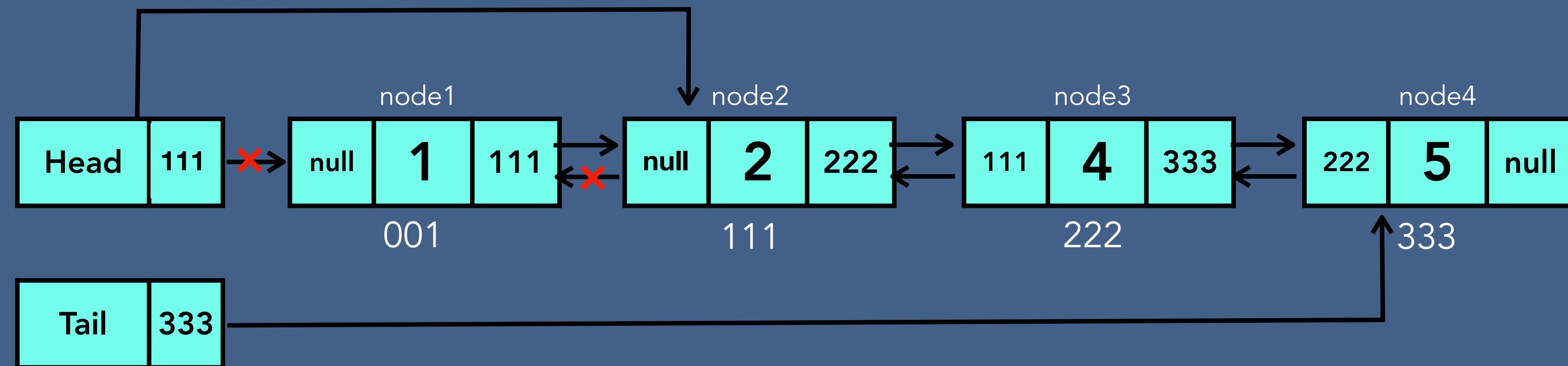
Case 1 - one node



# Deletion - Doubly Linked List

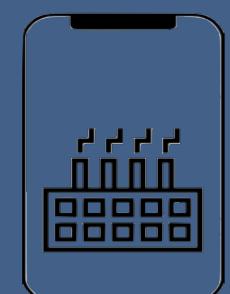
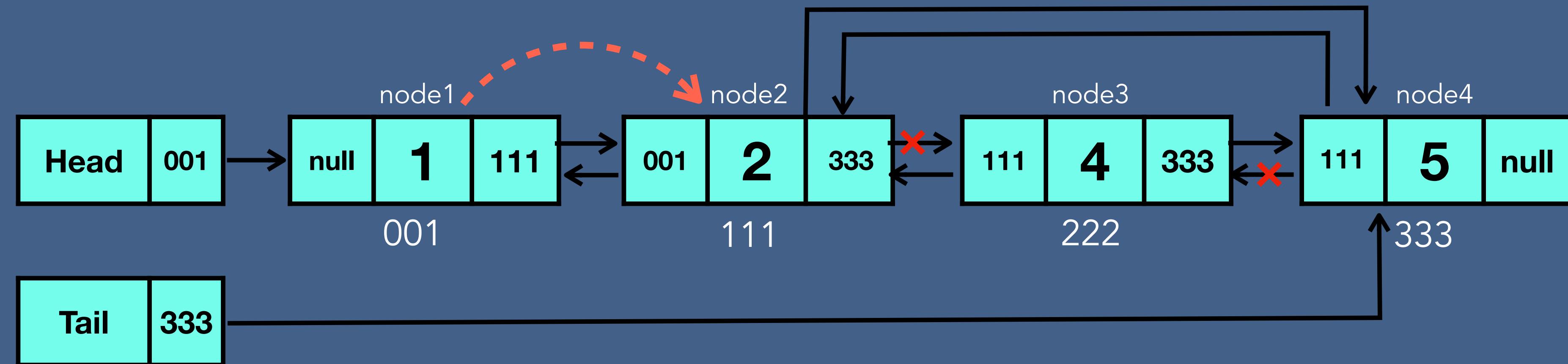
Deleting the first node

Case 2 - more than one node



# Deletion - Doubly Linked List

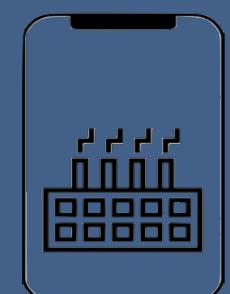
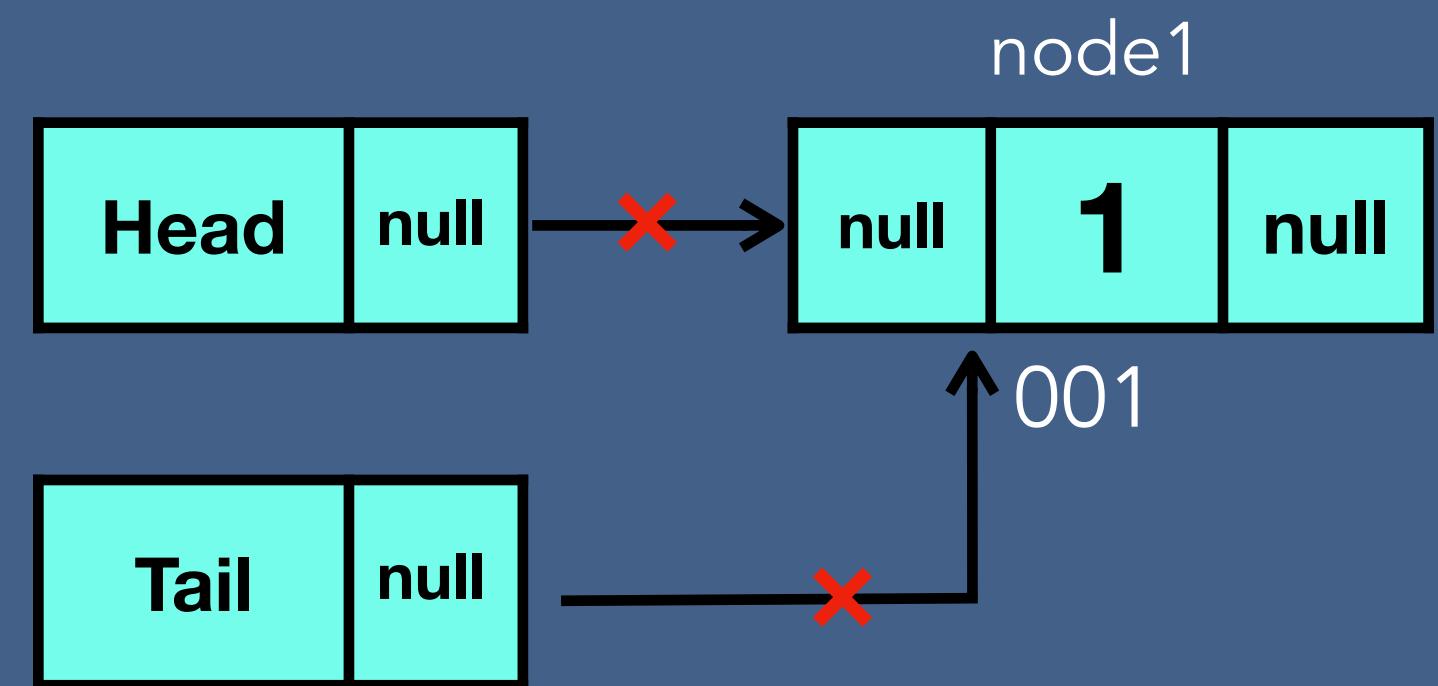
Deleting any given node



# Deletion - Doubly Linked List

Deleting the last node

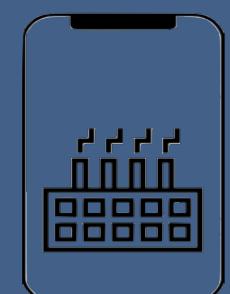
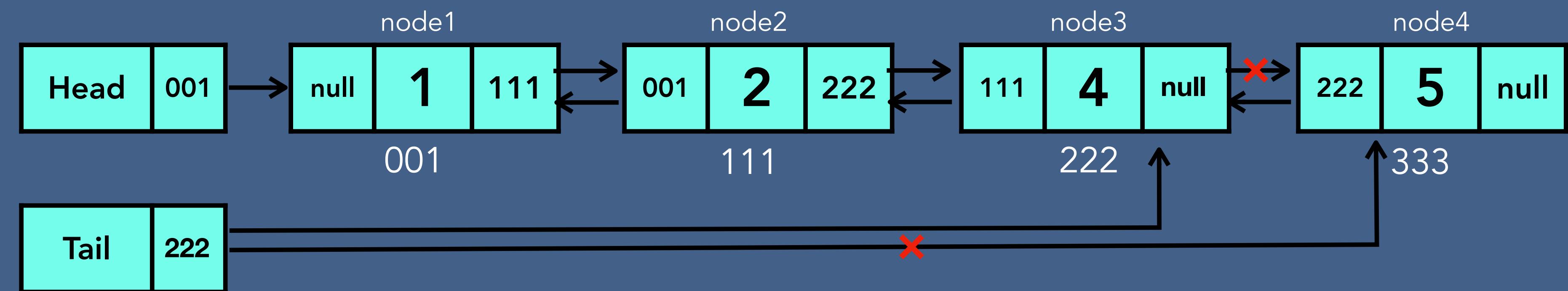
Case 1 - one node



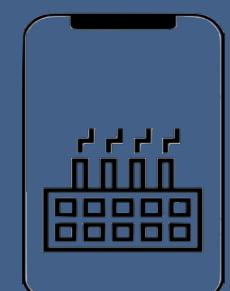
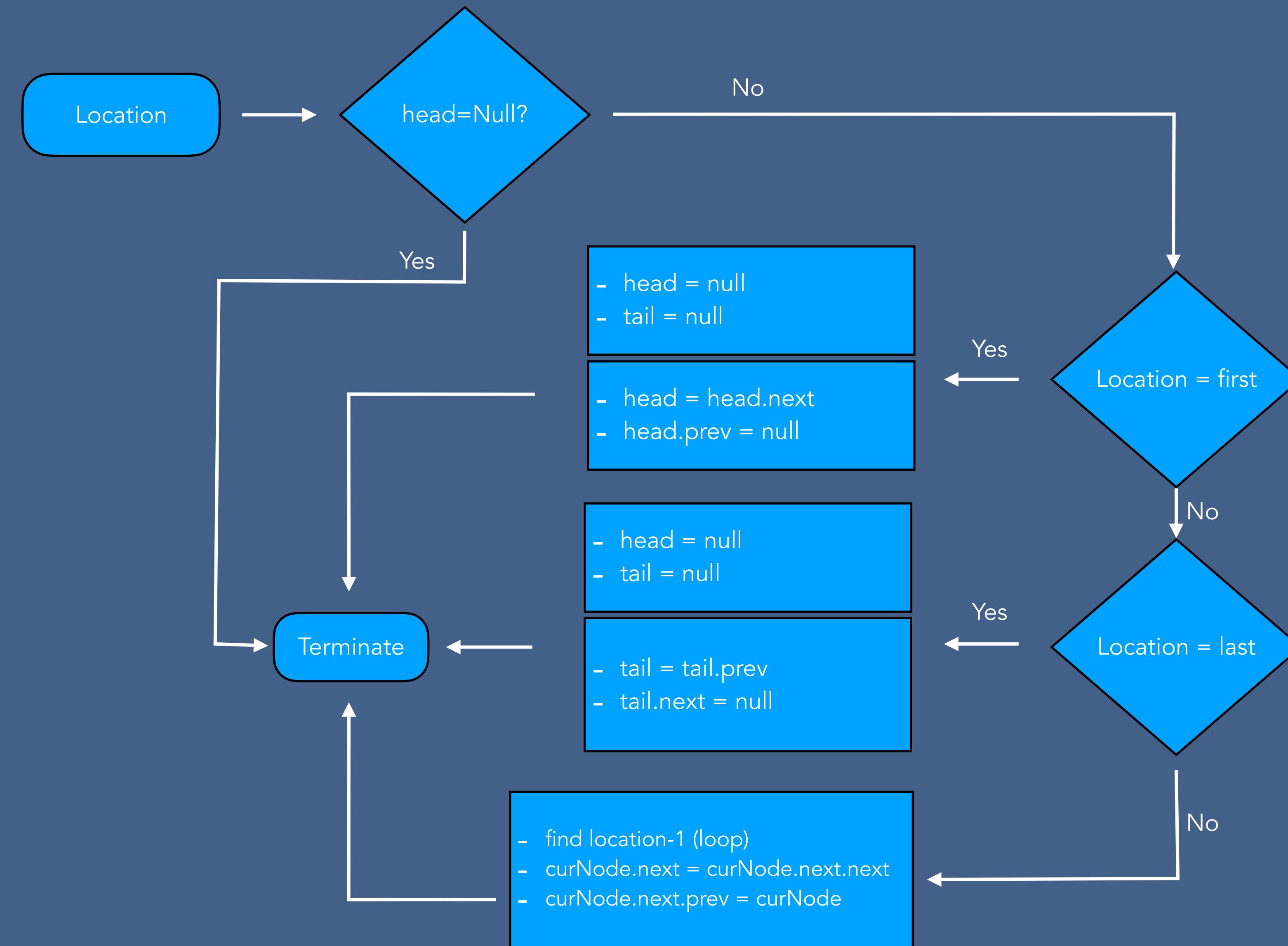
# Deletion - Doubly Linked List

Deleting the last node

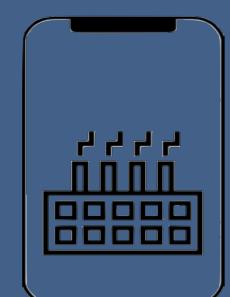
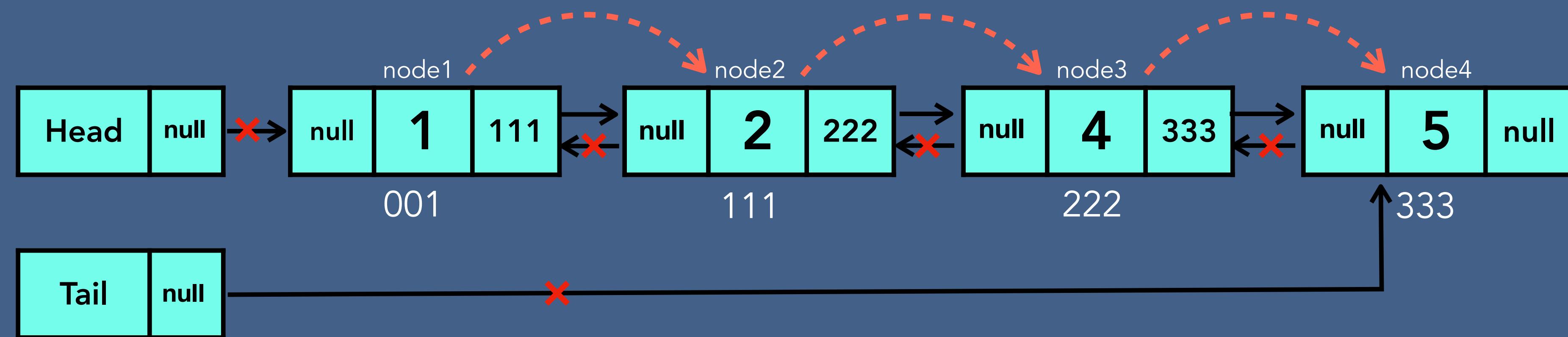
Case 2 - more than one node



# Deletion Algorithm - Doubly Linked List

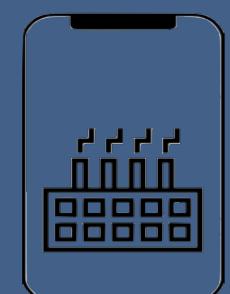
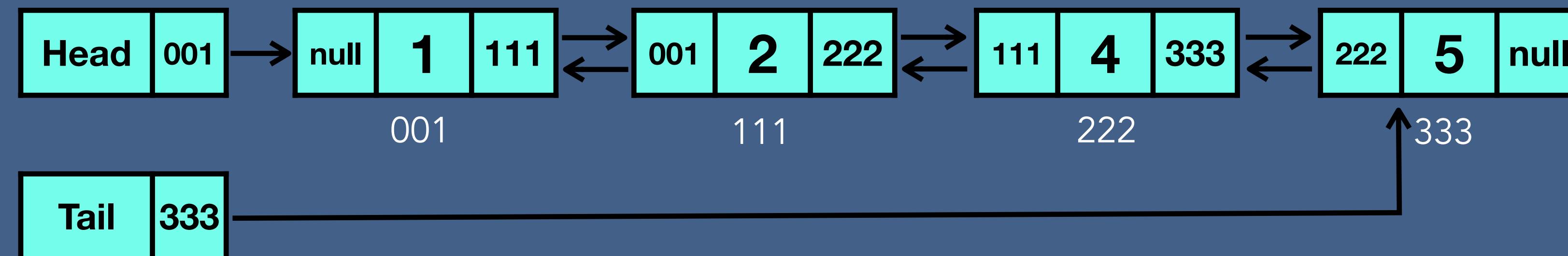


# Delete Entire Doubly Linked List

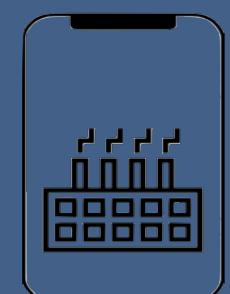
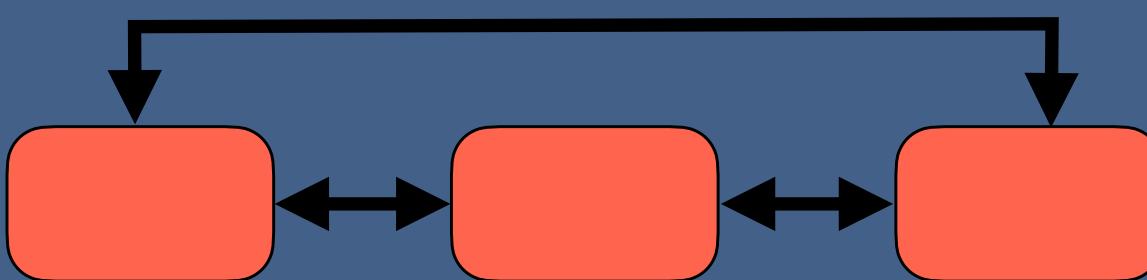


# Time and Space Complexity of Doubly Linked List

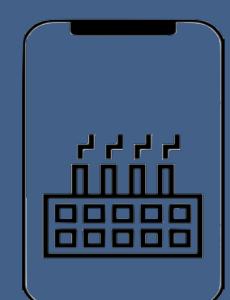
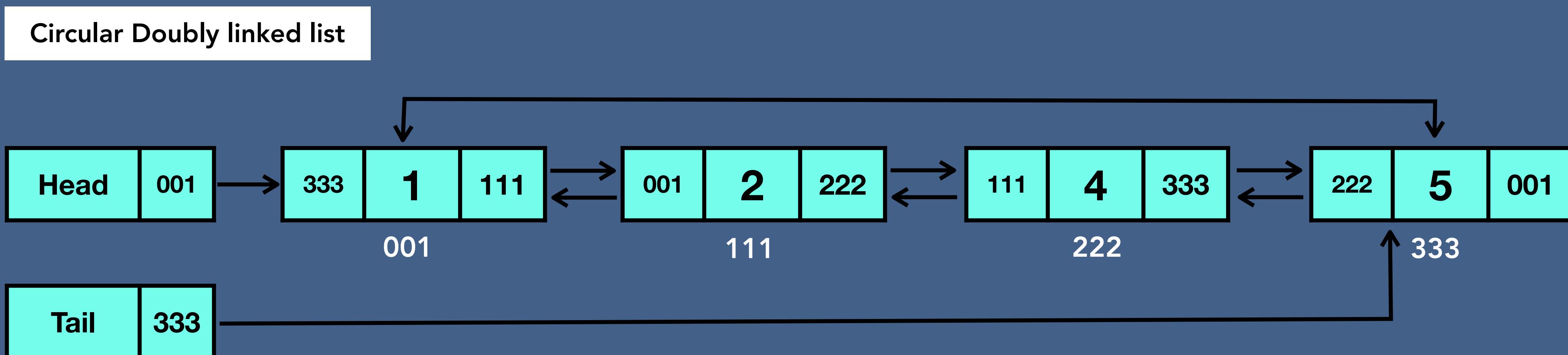
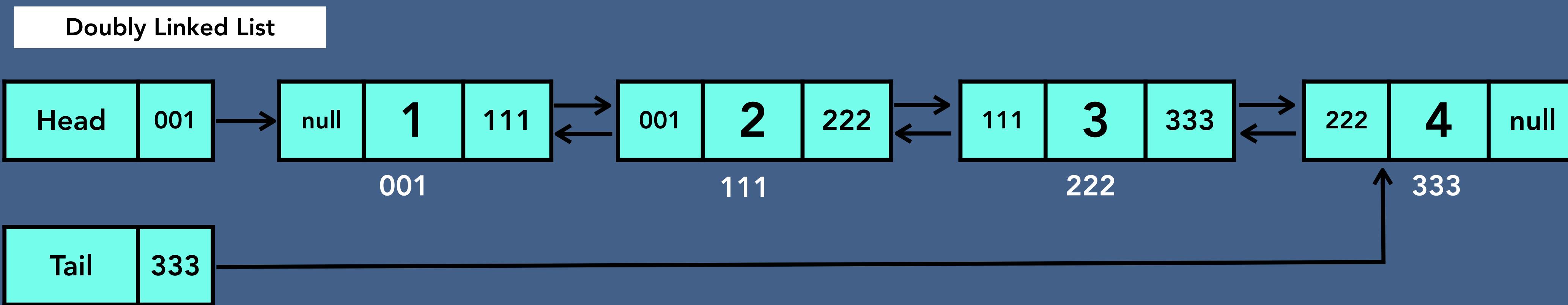
Doubly Linked List	Time complexity	Space complexity
Creation	$O(1)$	$O(1)$
Insertion	$O(n)$	$O(1)$
Searching	$O(n)$	$O(1)$
Traversing (forward ,backward)	$O(n)$	$O(1)$
Deletion of a node	$O(n)$	$O(1)$
Deletion of DLL	$O(n)$	$O(1)$



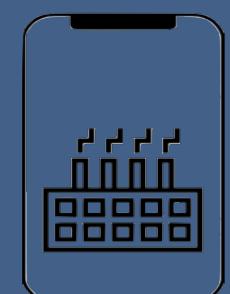
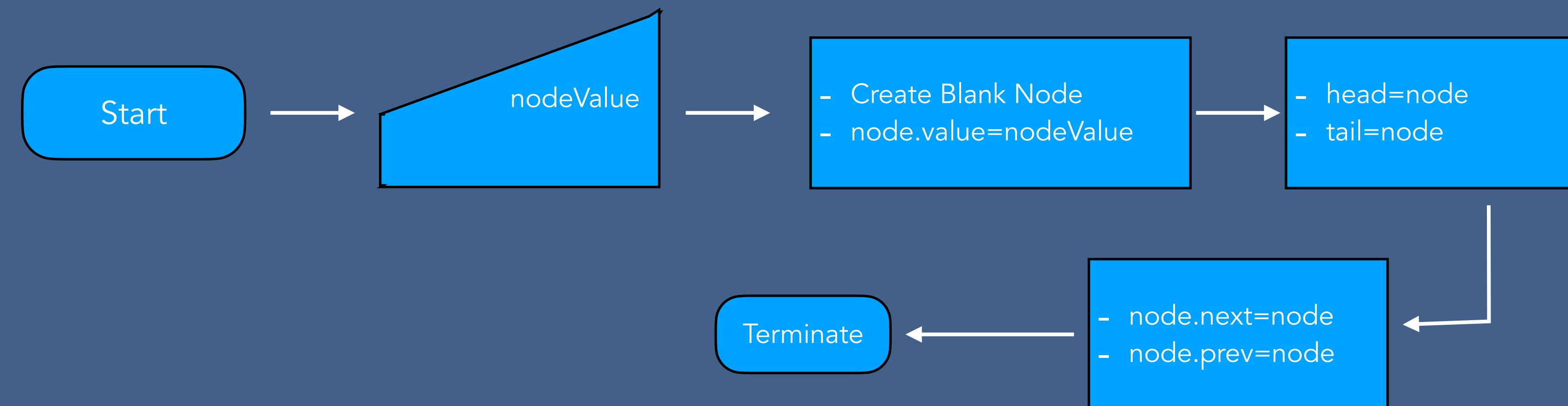
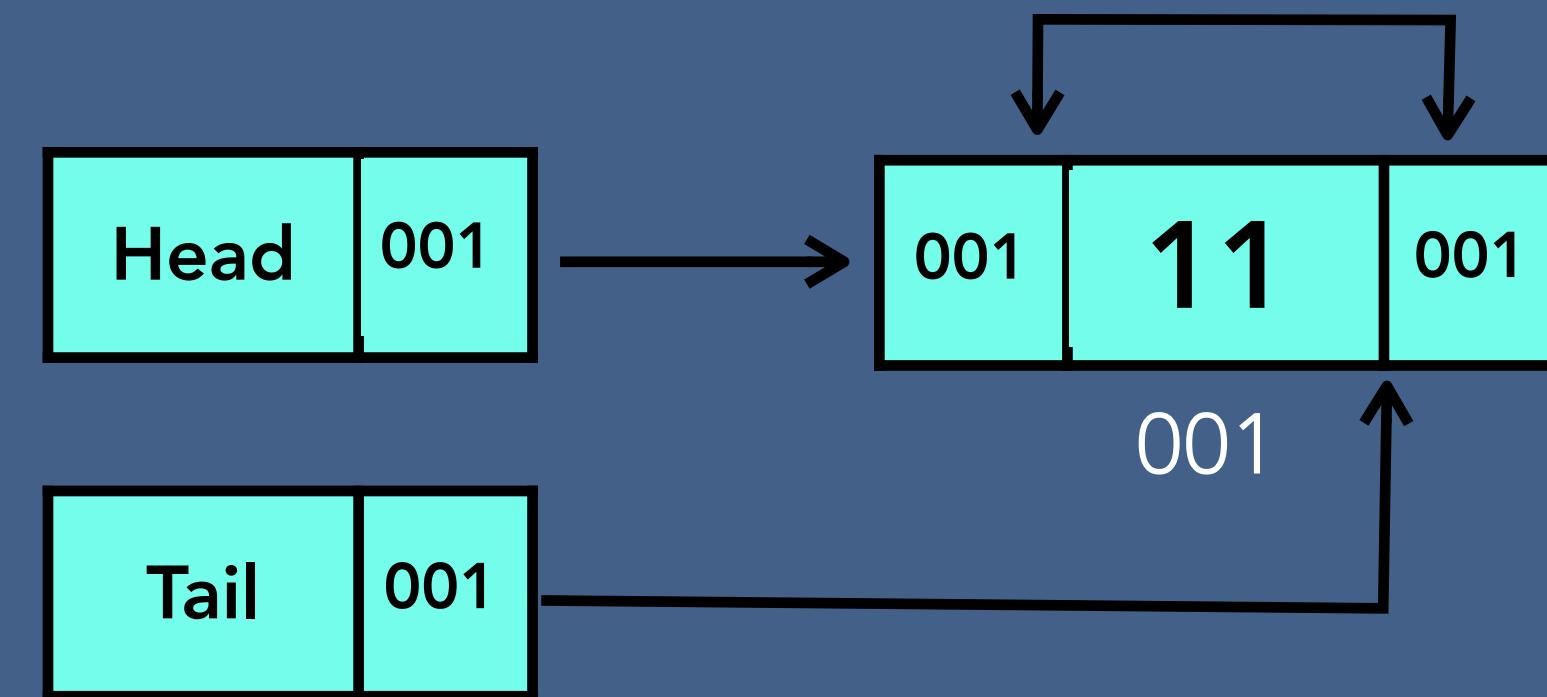
# Circular Doubly Linked List



# Circular Doubly Linked List

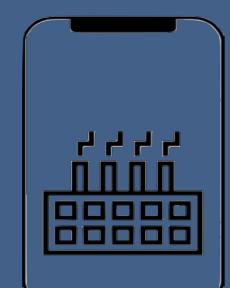
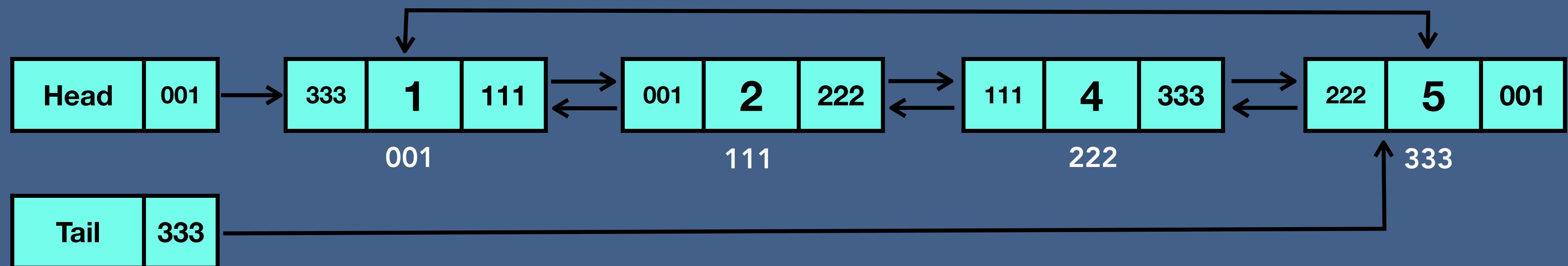


# Create - Circular Doubly Linked List



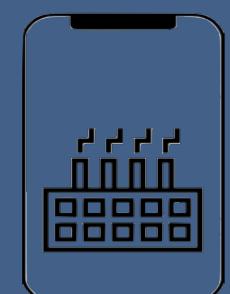
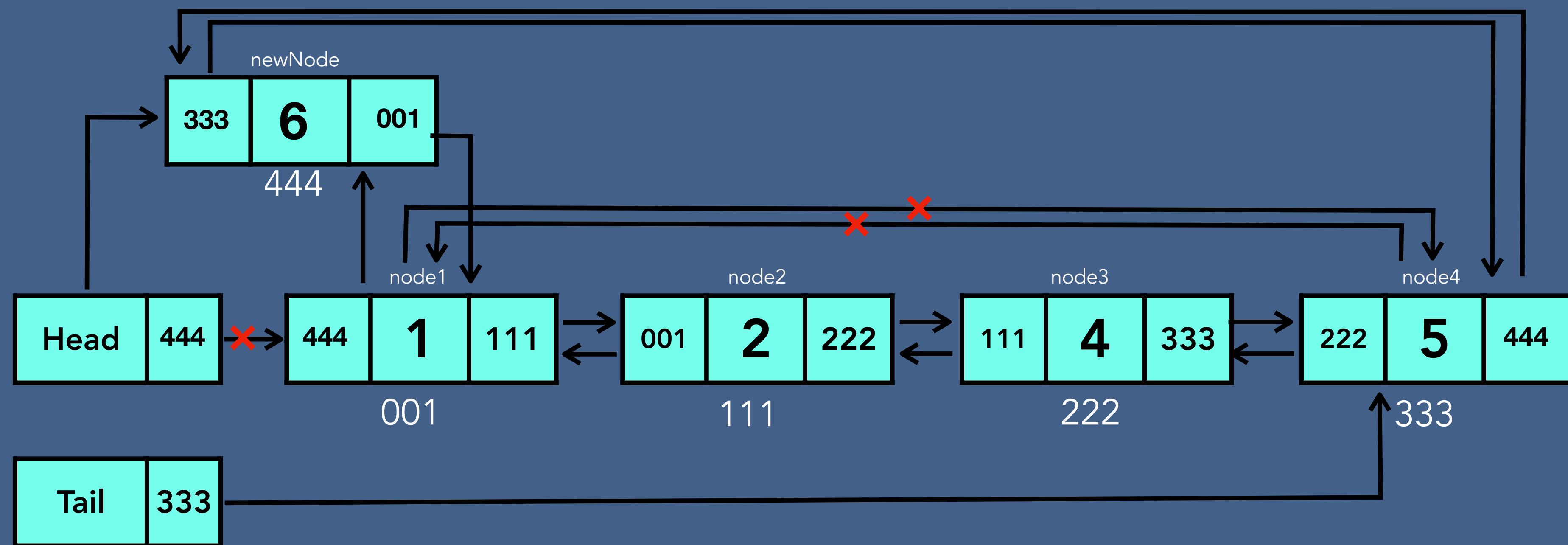
# Insertion - Circular Doubly Linked List

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



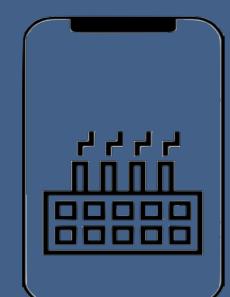
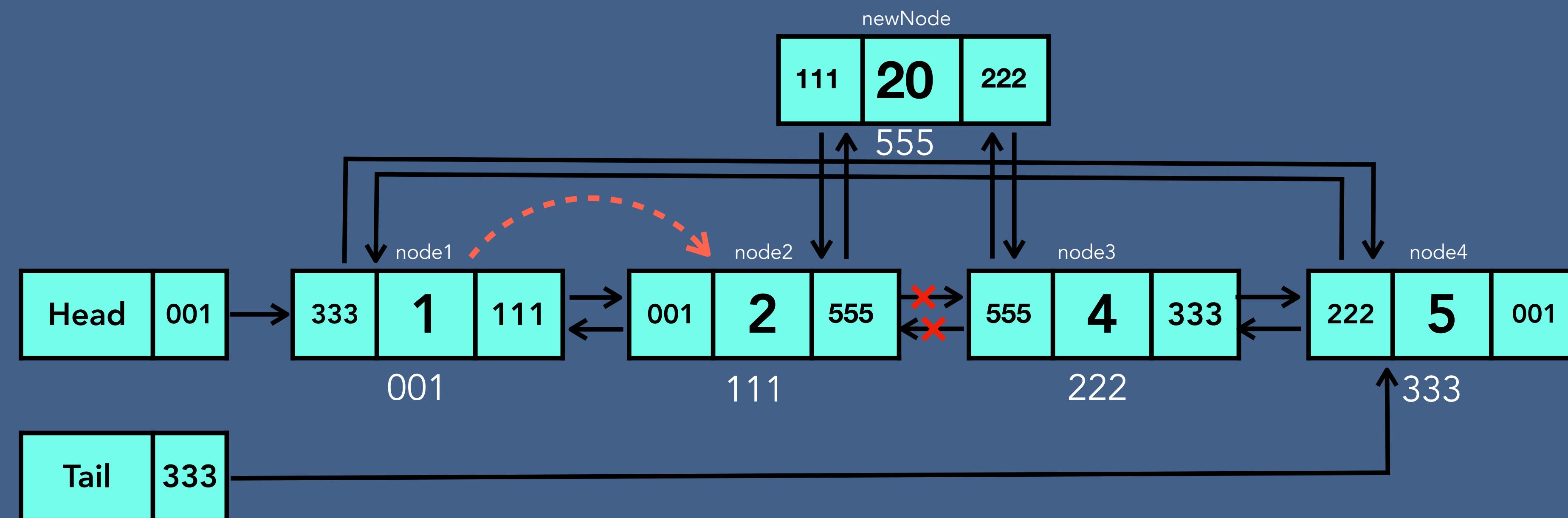
# Insertion - Circular Doubly Linked List

- Insert at the beginning of linked list



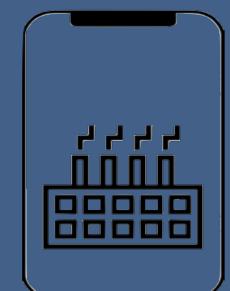
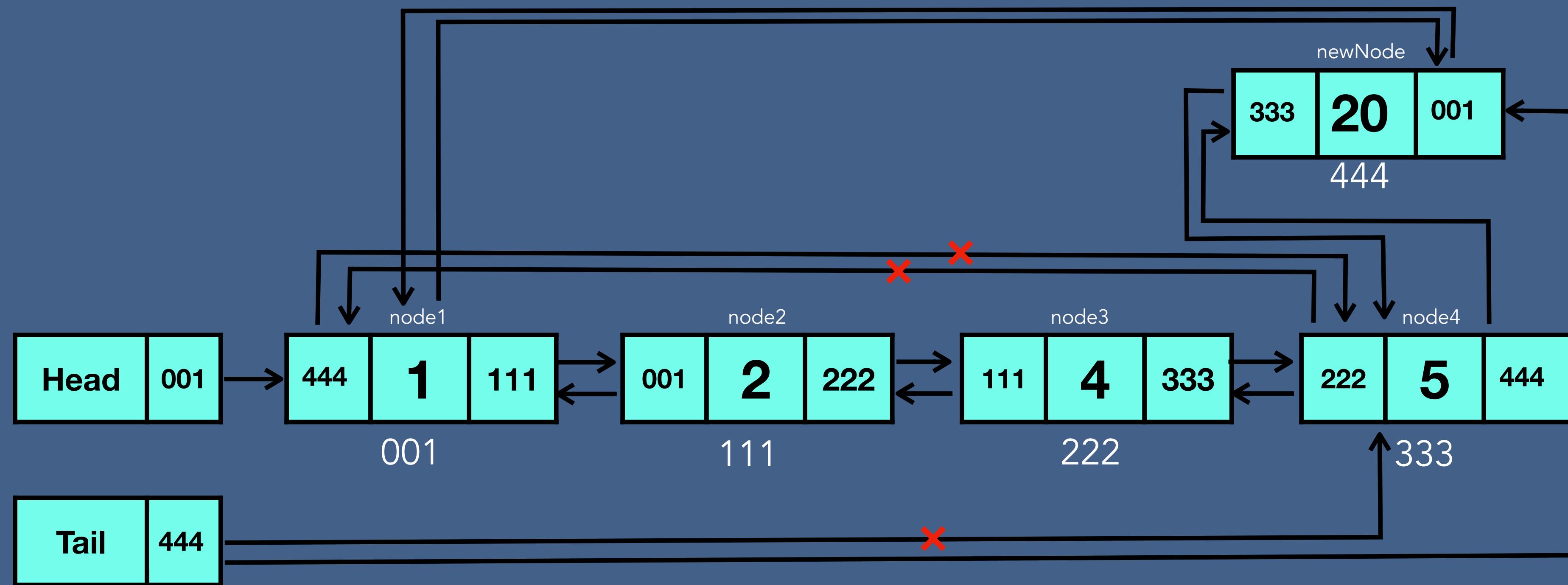
# Insertion - Circular Doubly Linked List

- Insert at the specified location of linked list

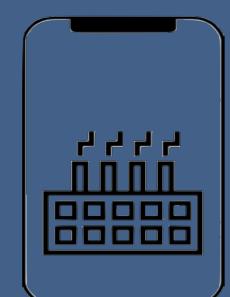
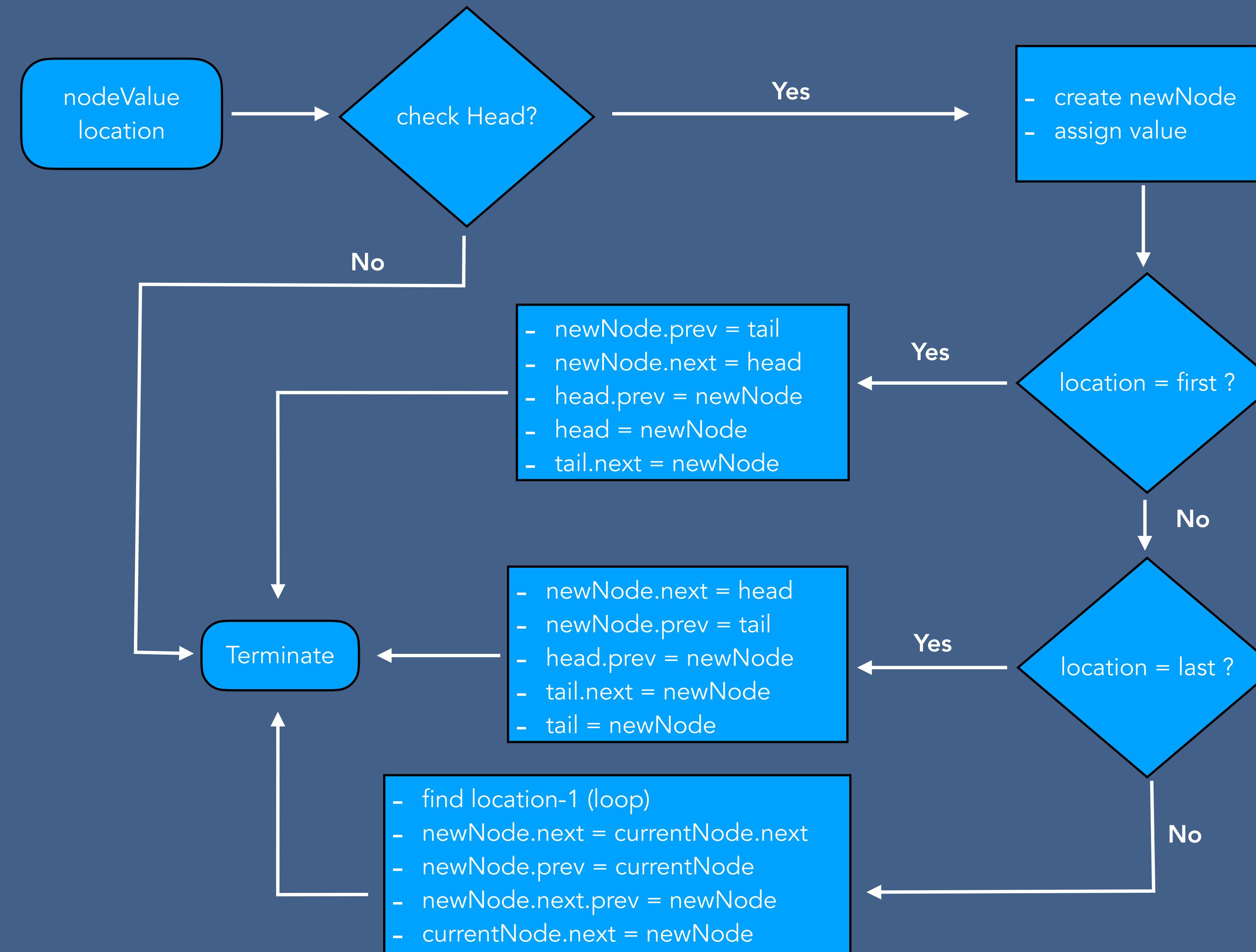


# Insertion - Circular Doubly Linked List

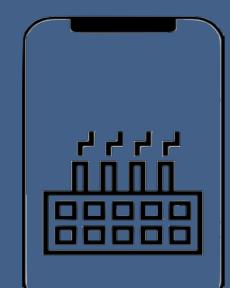
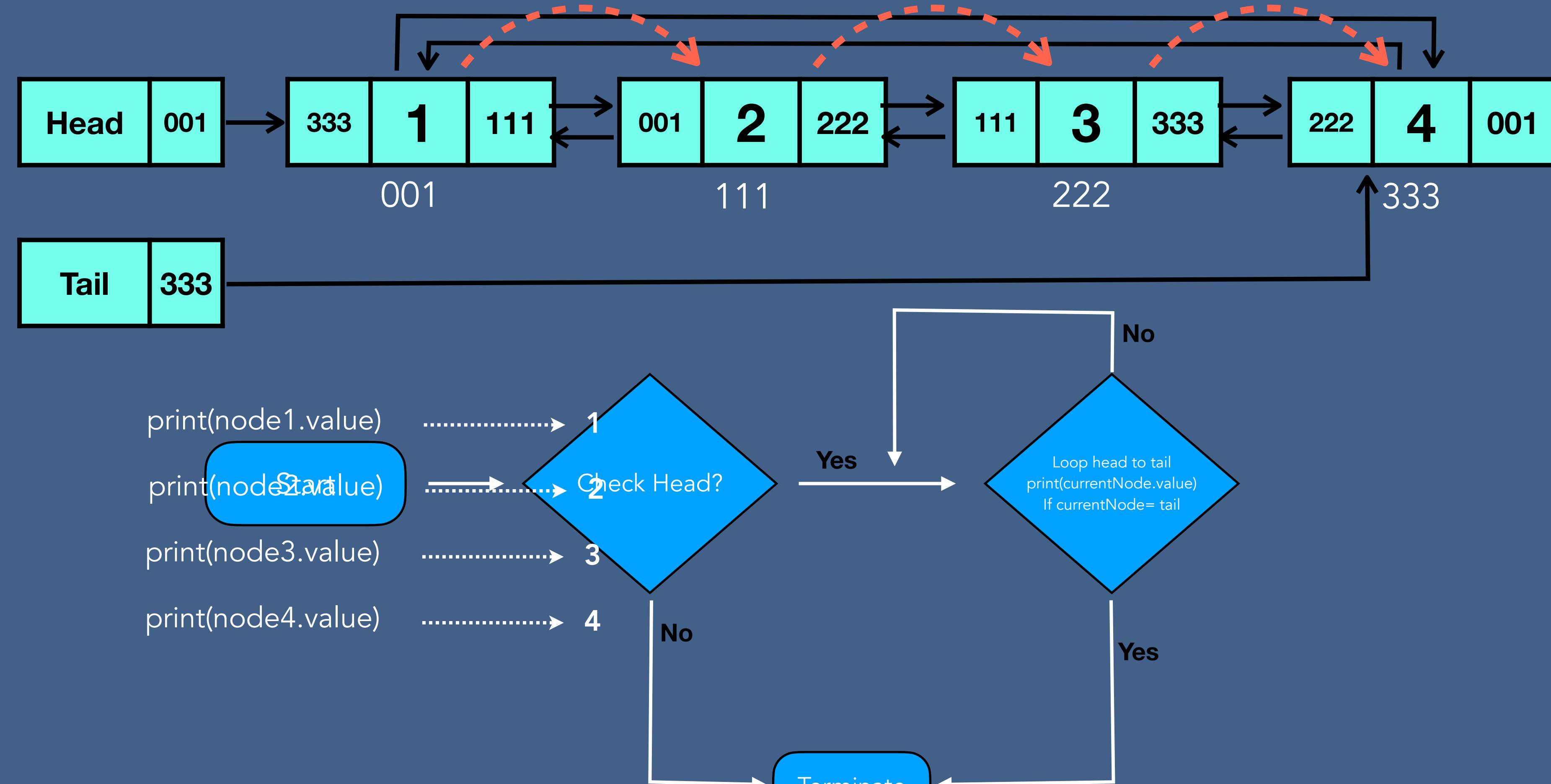
- Insert at the end of linked list



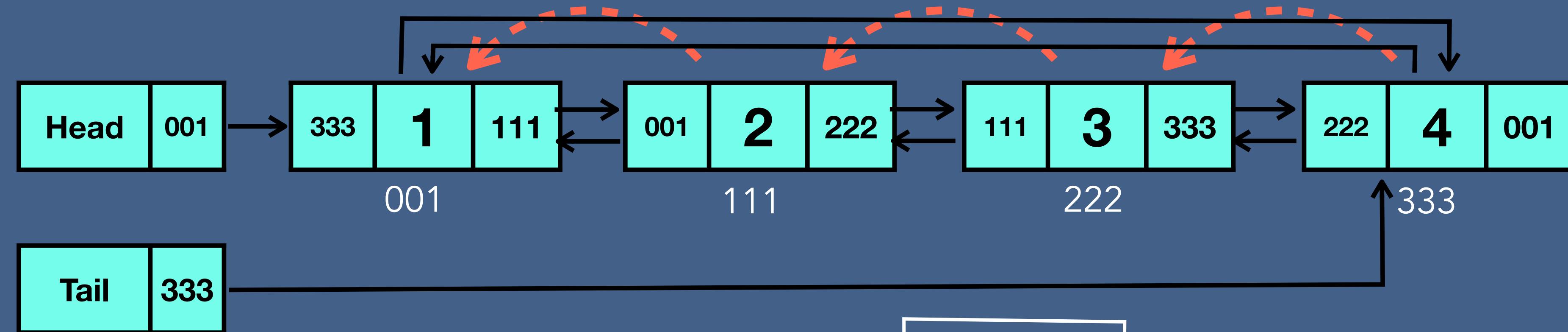
# Insertion Algorithm - Circular Doubly Linked List



# Traversal - Circular Doubly Linked List



# Reverse Traversal - Circular Doubly Linked List

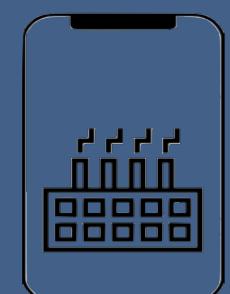
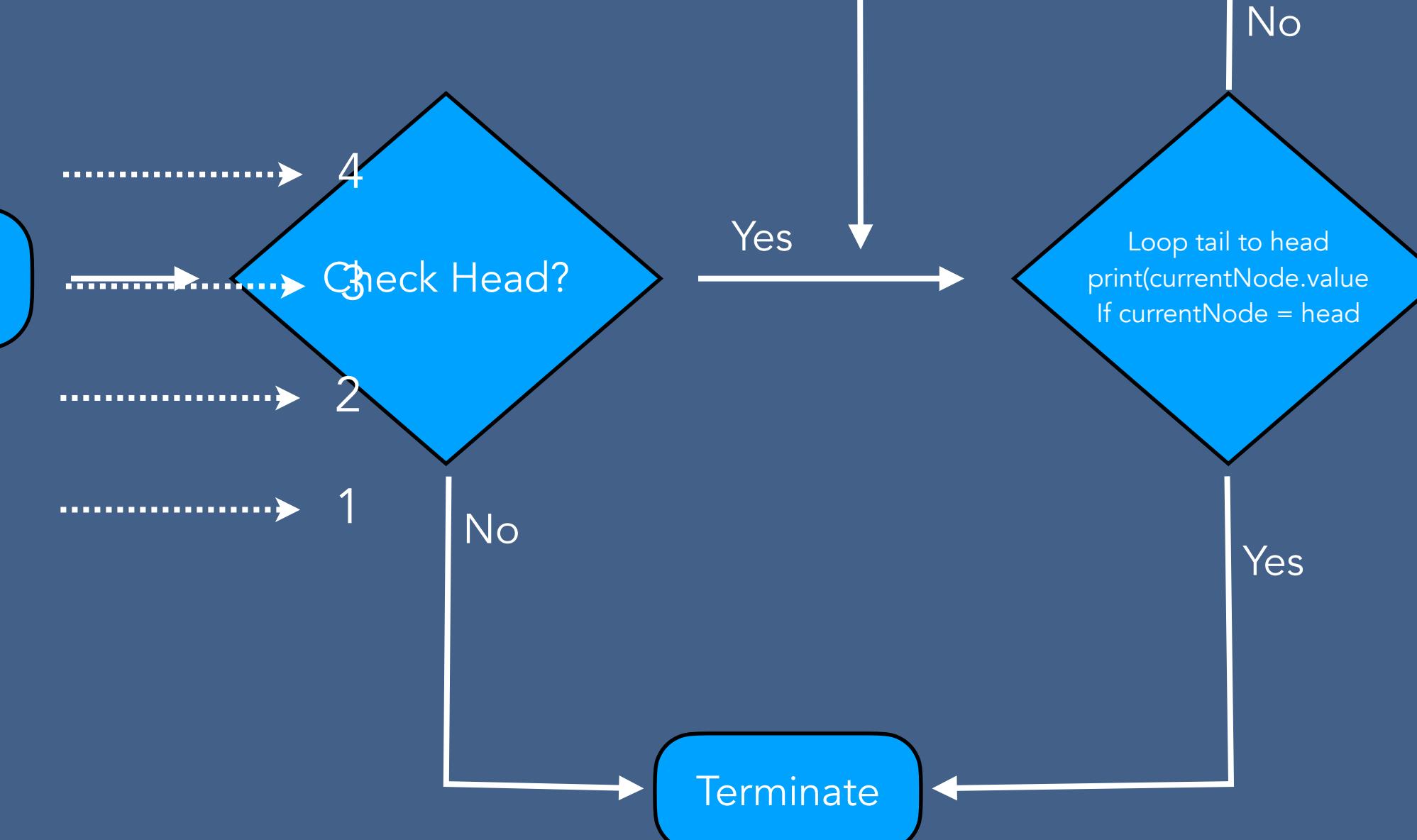


print(node4.value)

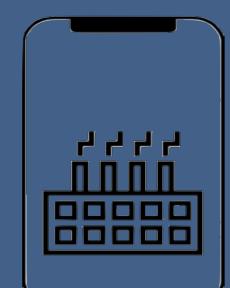
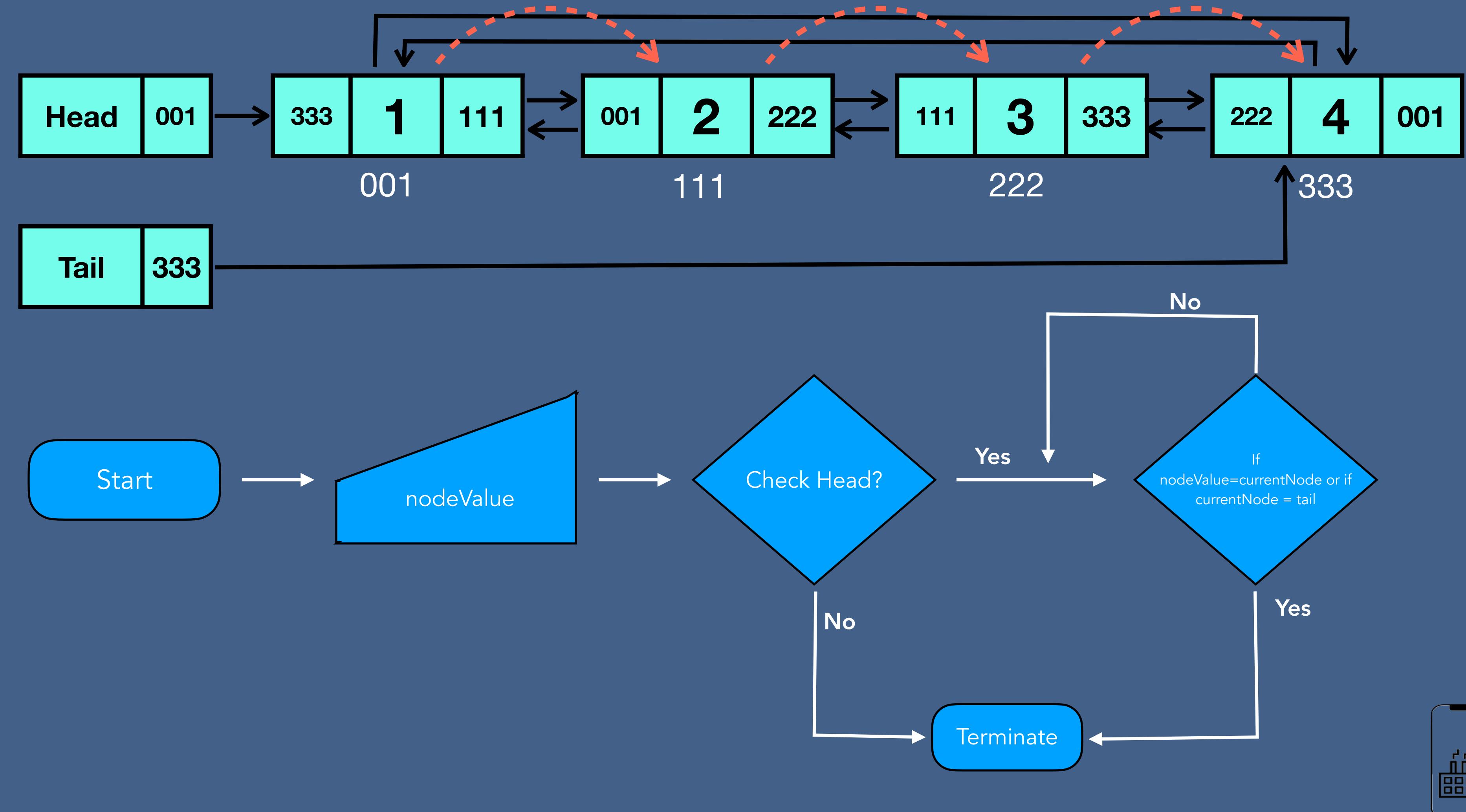
print(node3.value)

print(node2.value)

print(node1.value)

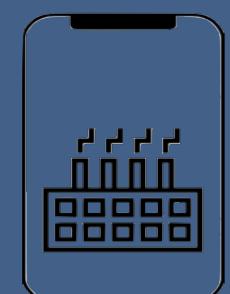
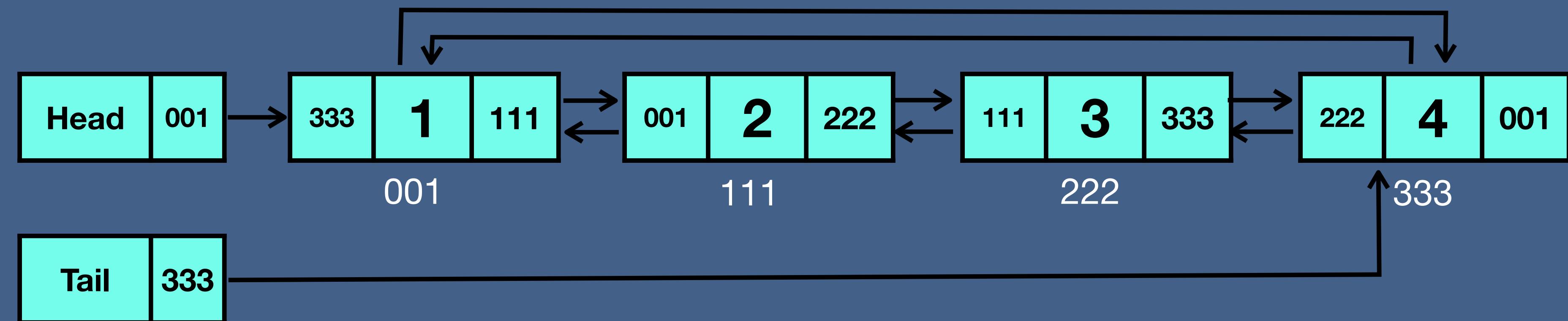


# Searching- Circular Doubly Linked List



# Deletion - Circular Doubly Linked List

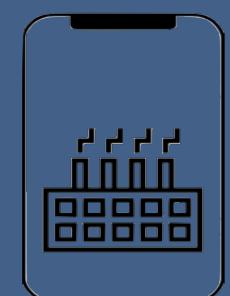
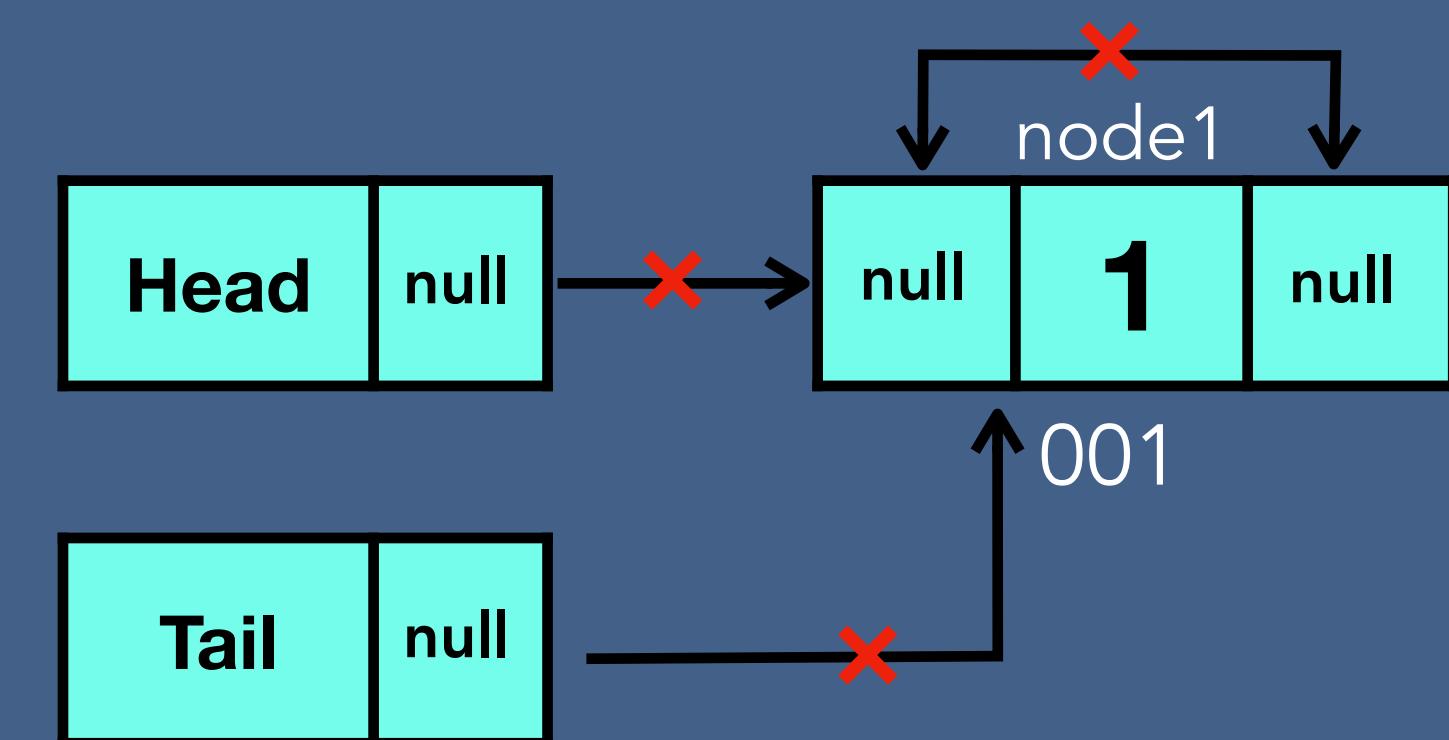
- Deleting the first node
- Deleting any given node
- Deleting the last node



# Deletion - Circular Doubly Linked List

Deleting the first node

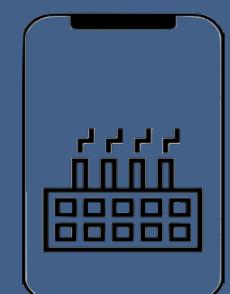
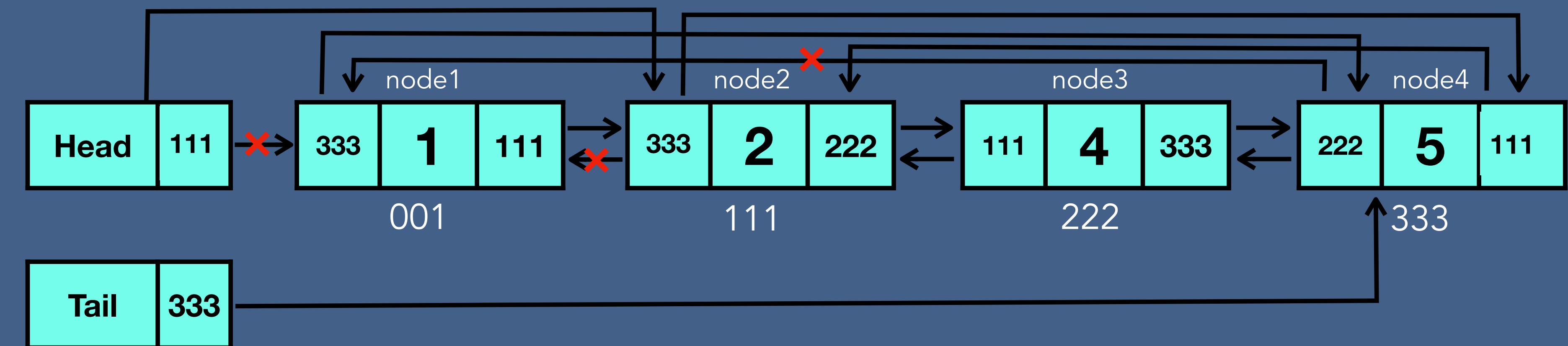
Case 1 - one node



# Deletion - Circular Doubly Linked List

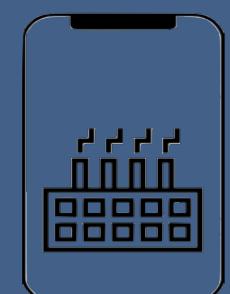
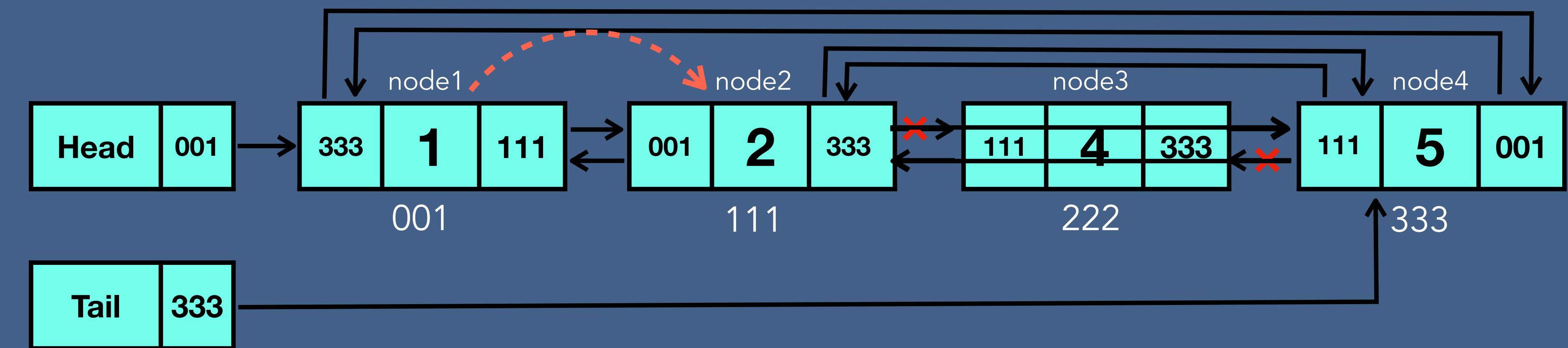
Deleting the first node

Case 2 - more than one node



# Deletion - Circular Doubly Linked List

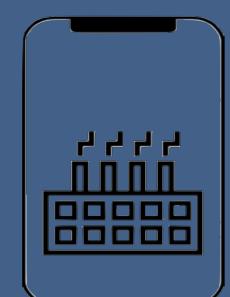
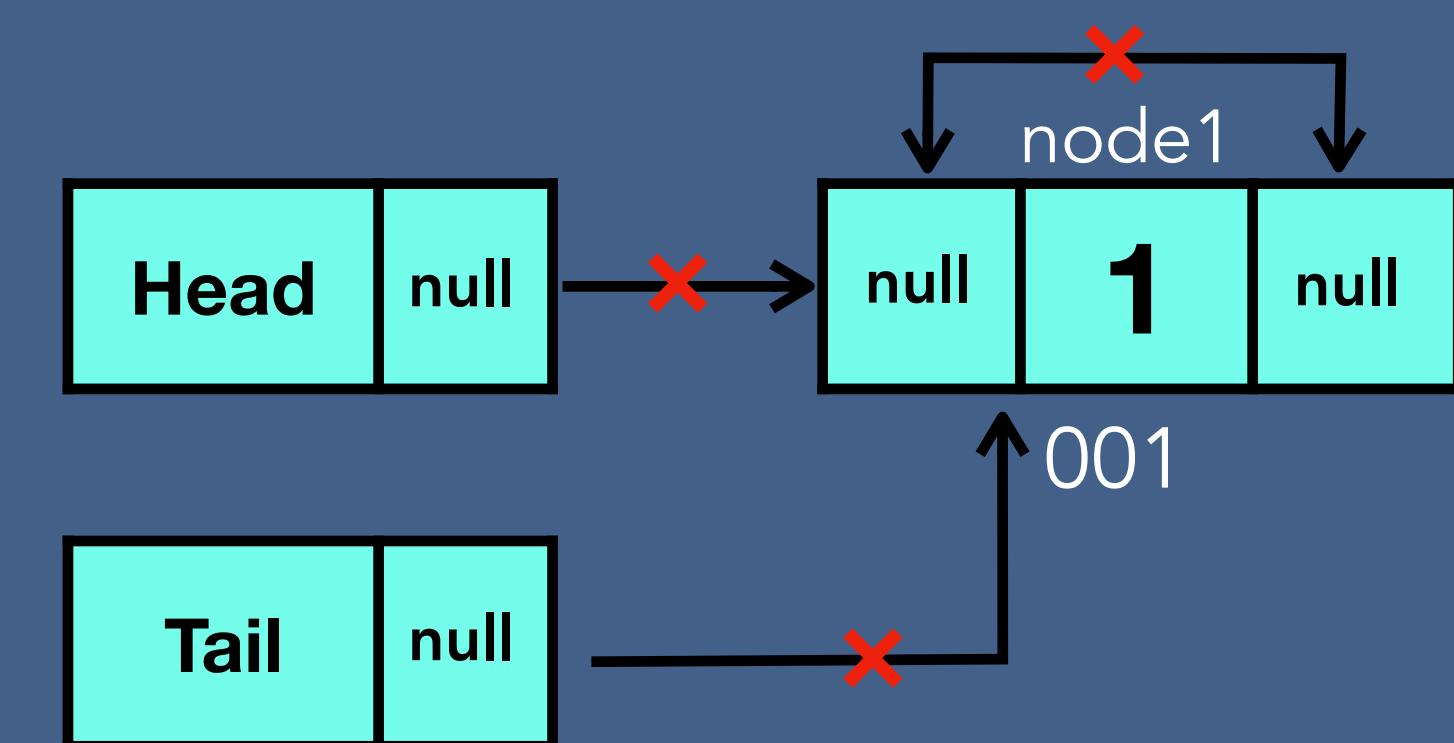
Deleting any given node



# Deletion - Circular Doubly Linked List

Deleting the last node

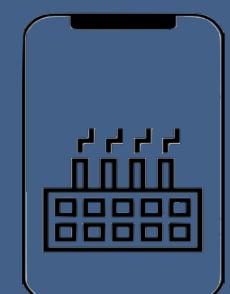
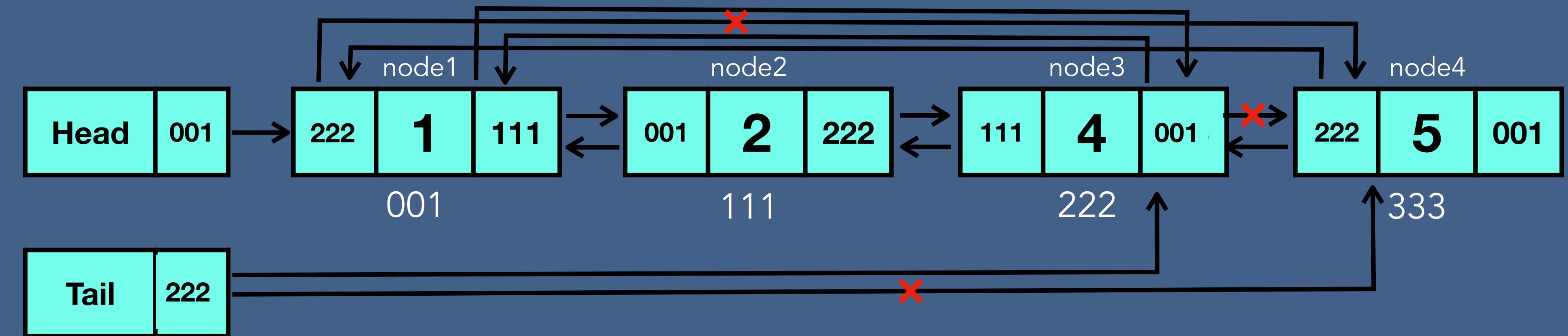
Case 1 - one node



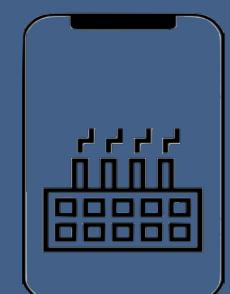
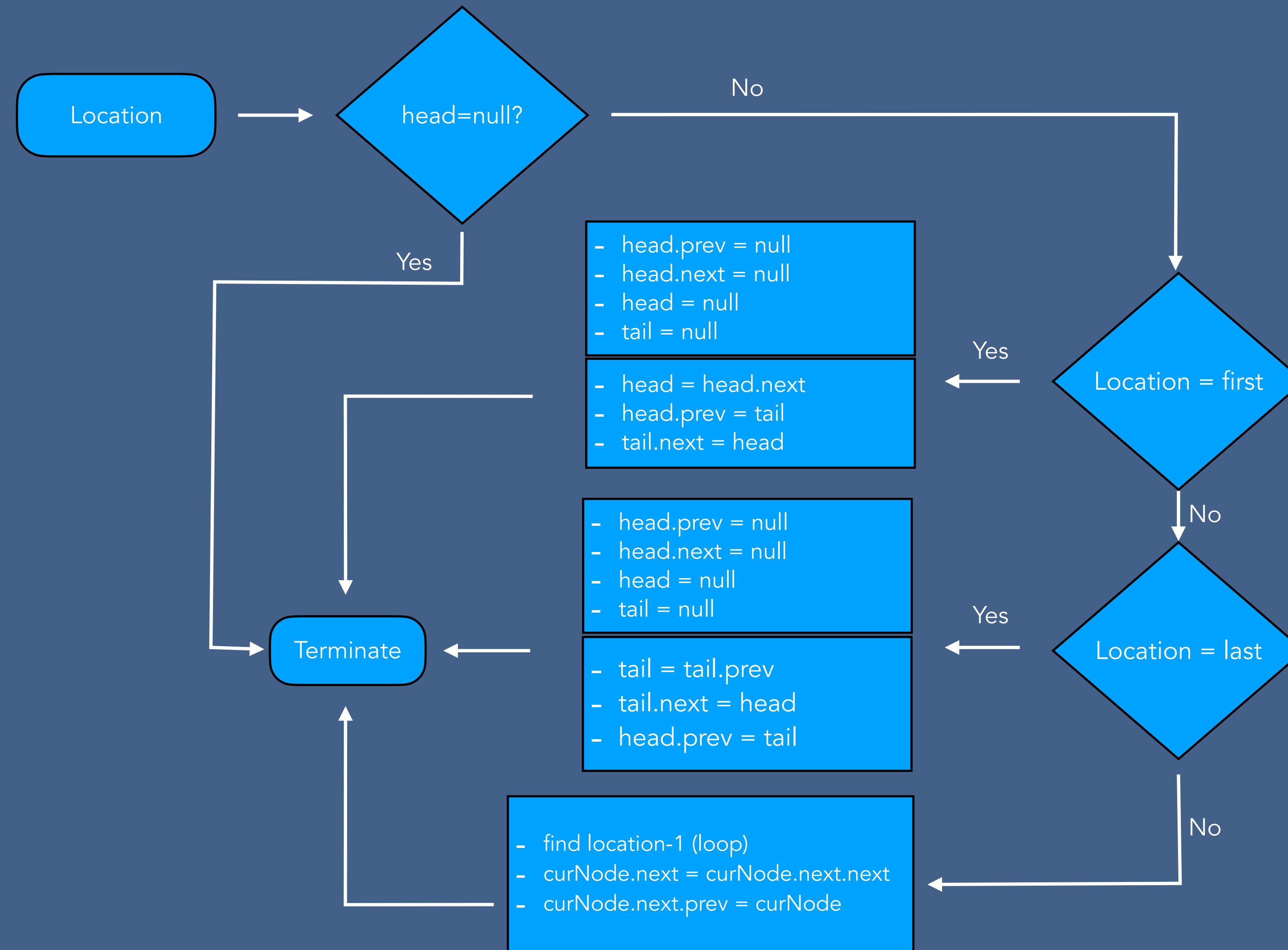
# Deletion - Circular Doubly Linked List

Deleting the last node

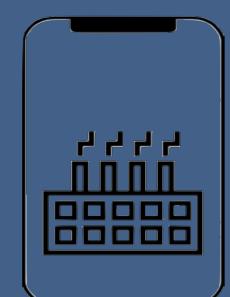
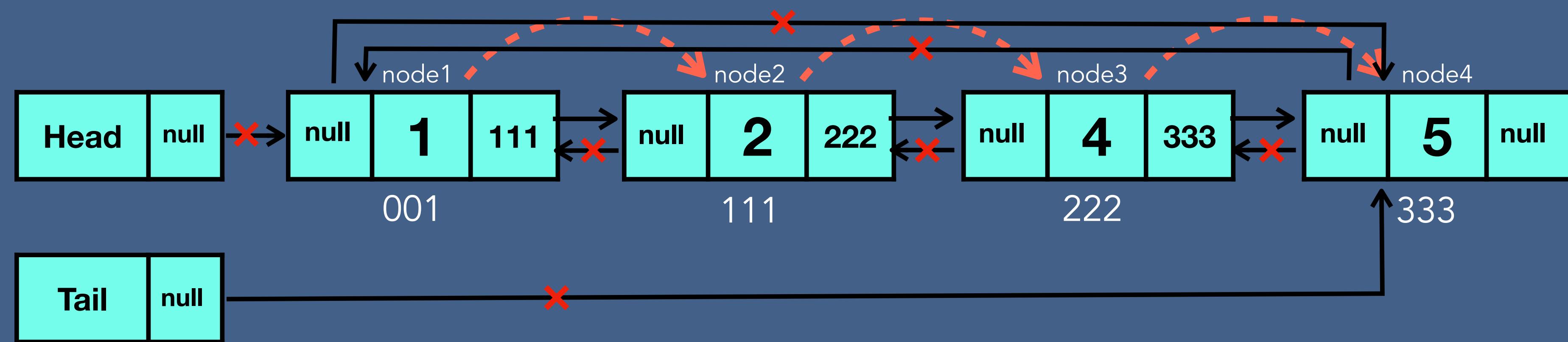
Case 2 - more than one node



# Deletion Algorithm - Circular Doubly Linked List

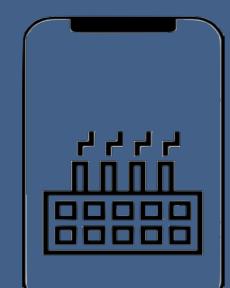
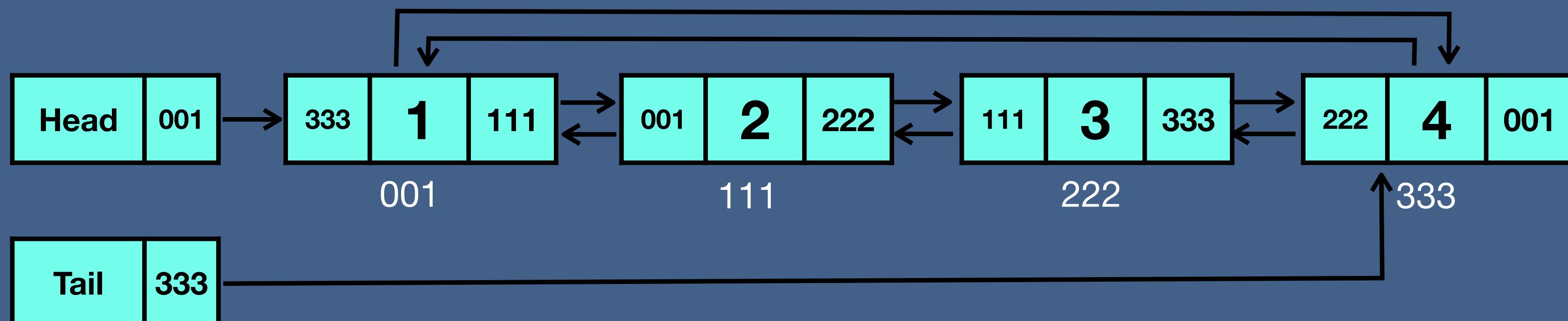


# Delete Entire Circular Doubly Linked List



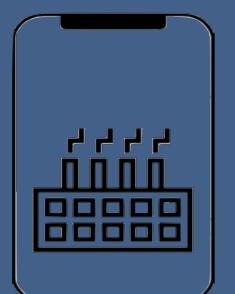
# Time and Space Complexity of Circular Doubly Linked List

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 CDLL	$O(n)$	$O(1)$

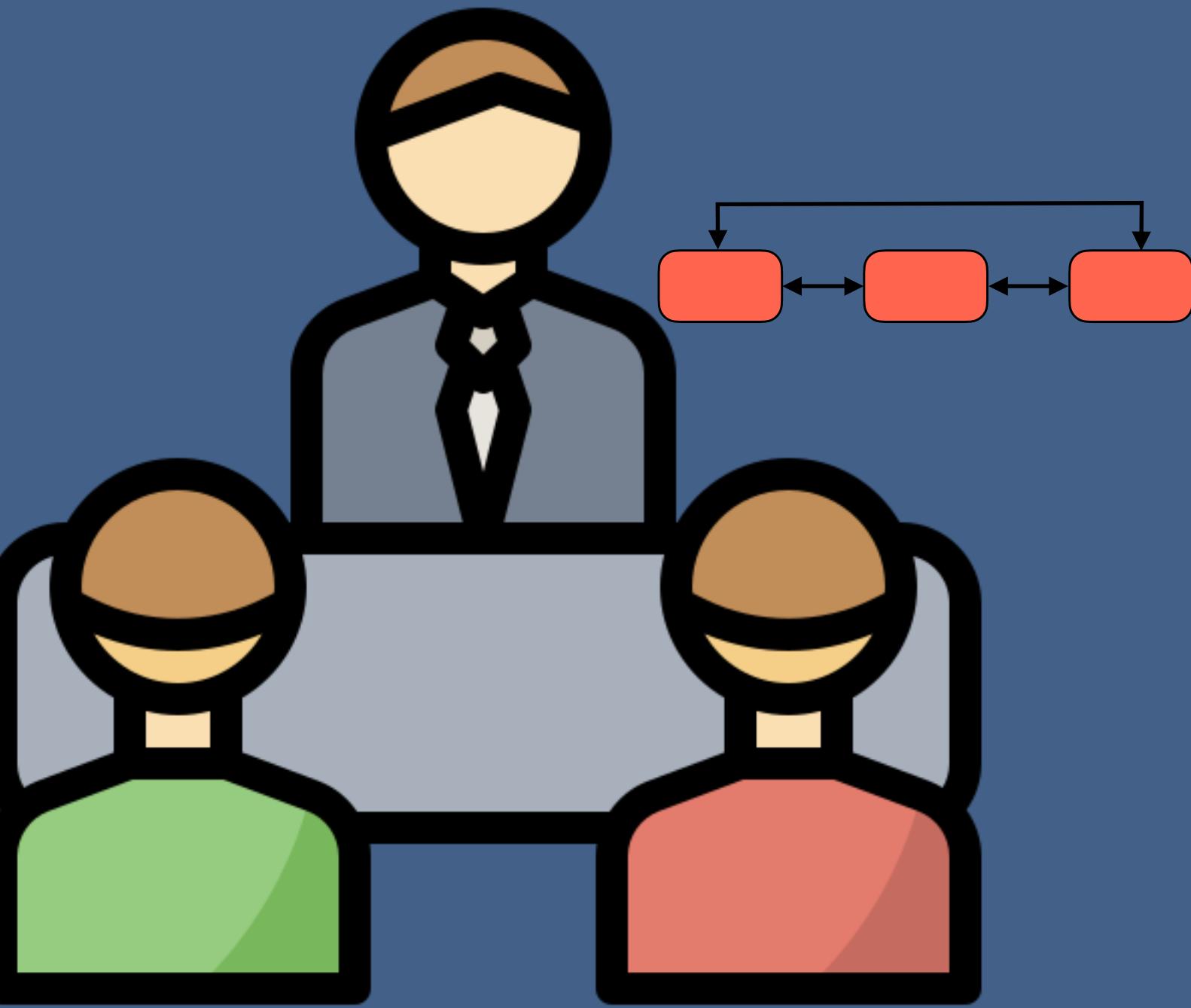


# Time Complexity of Array vs Linked List

	<b>Array</b>	<b>Linked List</b>
Creation	$O(1)$	$O(1)$
Insertion at first position	$O(1)$	$O(1)$
Insertion at last position	$O(1)$	$O(1)$
Insertion at $n^{\text{th}}$ position	$O(1)$	$O(n)$
Searching in Unsorted data	$O(n)$	$O(n)$
Searching in Sorted data	$O(\log n)$	$O(n)$
Traversing	$O(n)$	$O(n)$
Deletion at first position	$O(1)$	$O(1)$
Deletion at last position	$O(1)$	$O(n)/O(1)$
Deletion at $n^{\text{th}}$ position	$O(1)$	$O(n)$
Deletion of array/linked list	$O(1)$	$O(n)/O(1)$
Access $n^{\text{th}}$ element	$O(1)$	$O(n)$

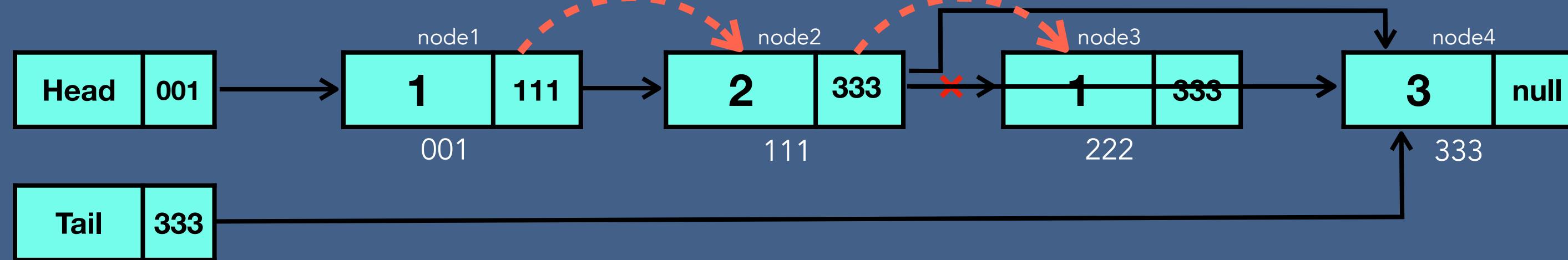


# Linked List Interview Questions

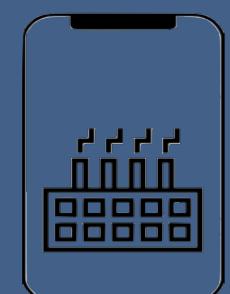


# Remove Duplicates

Write a method to remove duplicates from an unsorted linked list.

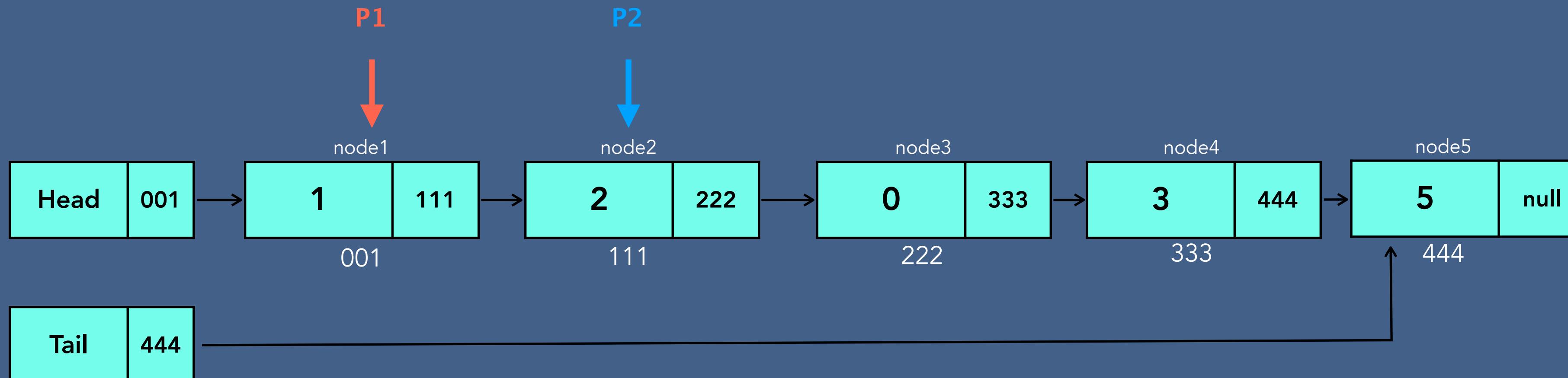


```
currentNode = node1  
hashSet = {1} → {1,2} → {1,2,3}  
while currentNode.next is not Null  
    If next node's value is in hashSet  
        Delete next node  
    Otherwise add it to hashSet
```



# Return Nth to Last

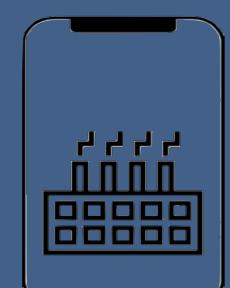
Implement an algorithm to find the nth to last element of a singly linked list.



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

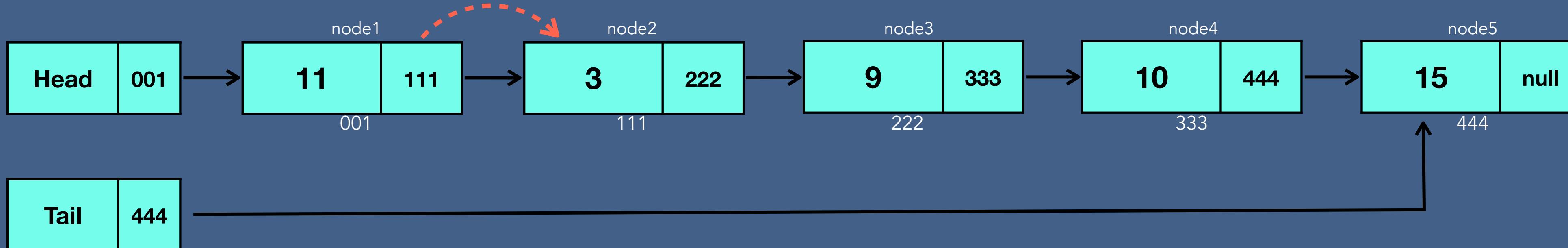
```
pointer1 = node1  
= node2  
= node3  
= node4
```

```
pointer2 = node2  
= node3  
= node4  
= node5
```



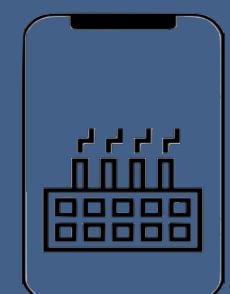
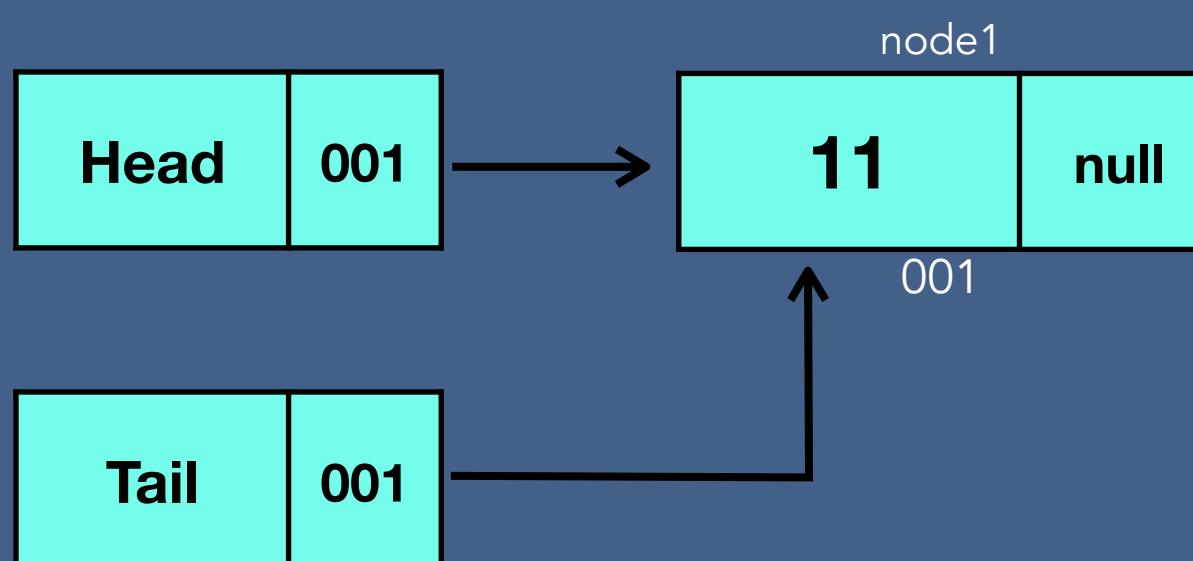
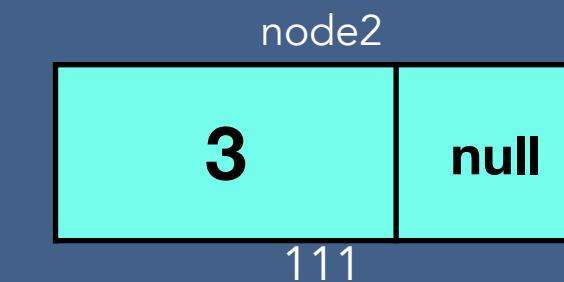
# Partition

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  $x$ .



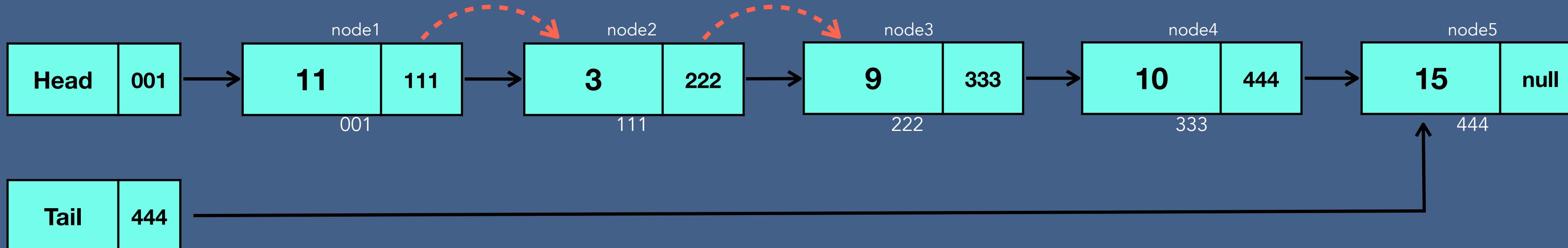
$x = 10$

currentNode = node1  
Tail = node1  
currentNode.next = null



# 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  $x$ .

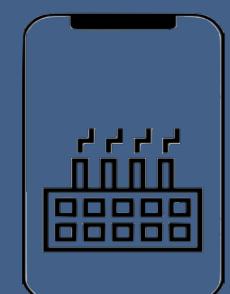
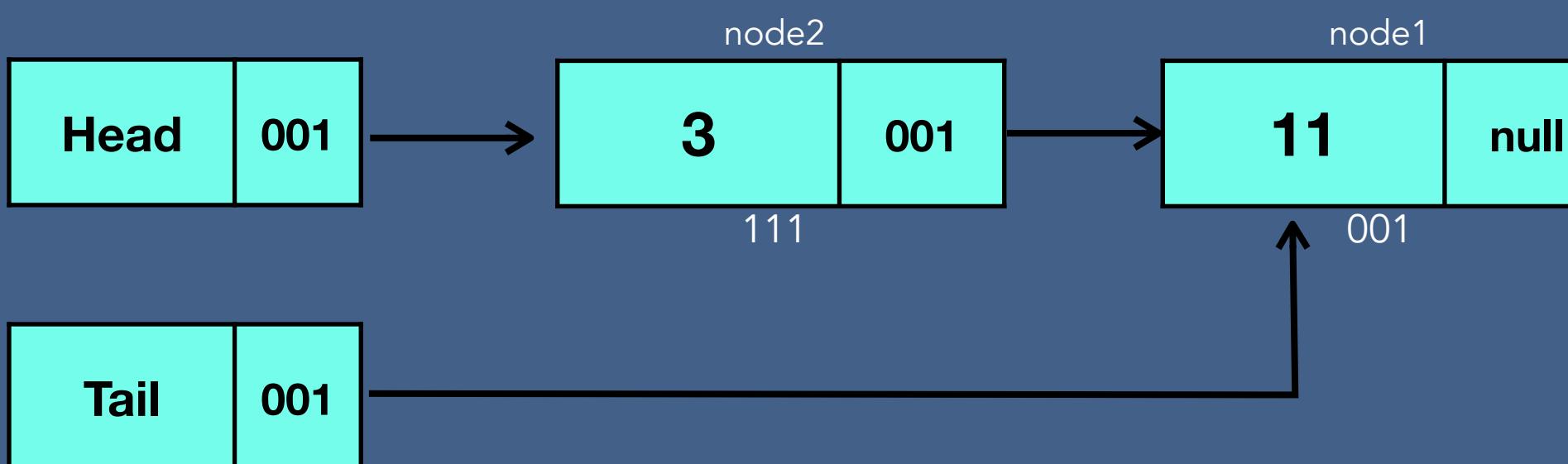


$x = 10$

currentNode = node1

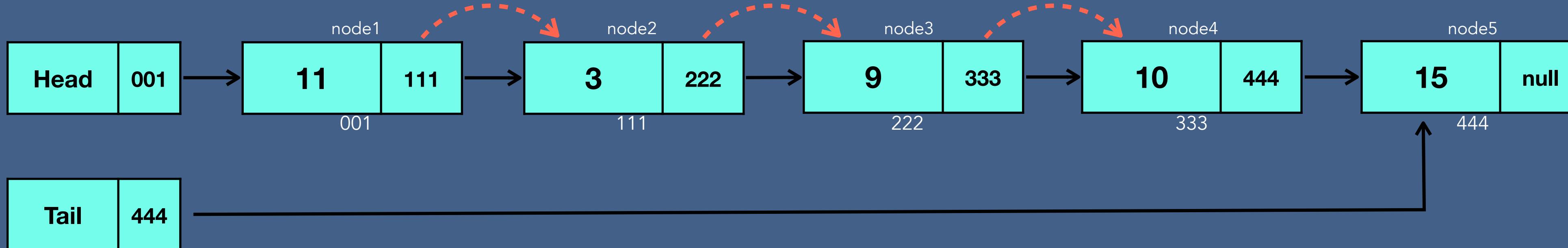
Tail = node1

currentNode.next = null



# 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  $x$ .

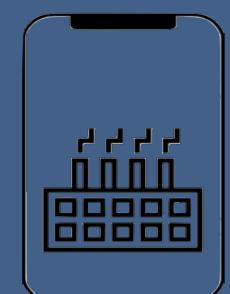
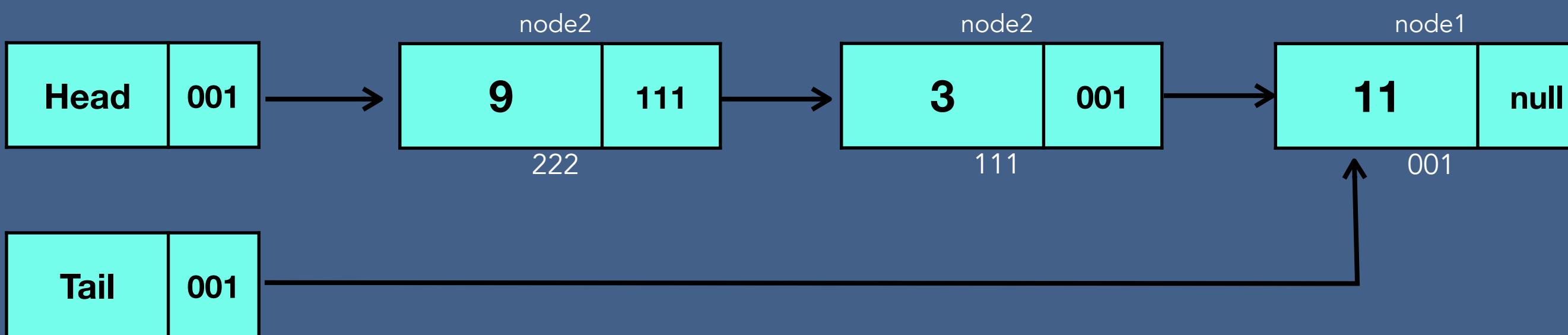
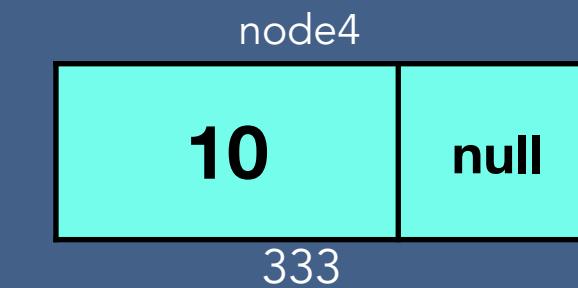


$x = 10$

currentNode = node1

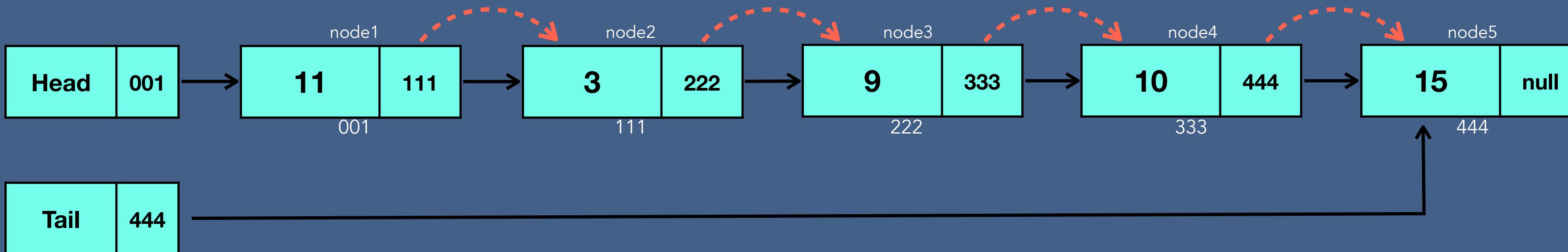
Tail = node1

currentNode.next = null



# 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  $x$ .

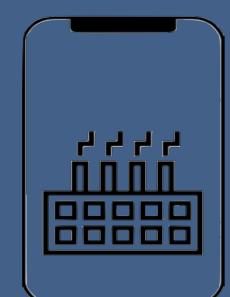
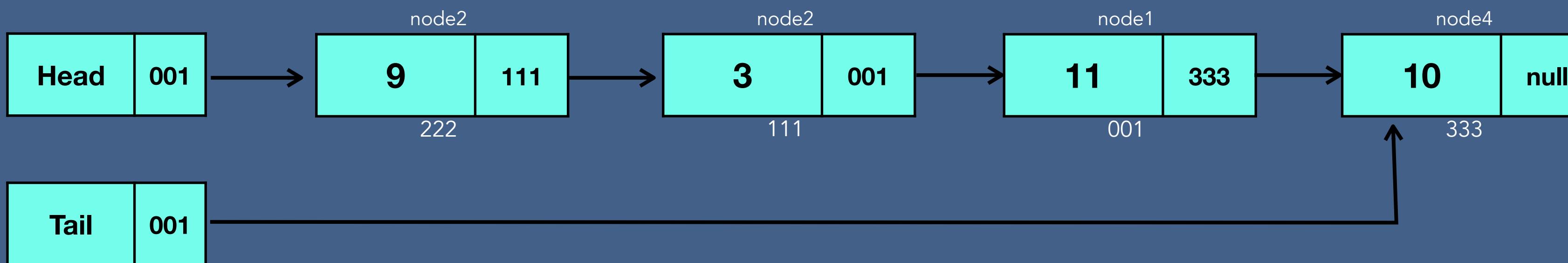


$x = 10$

`currentNode = node1`

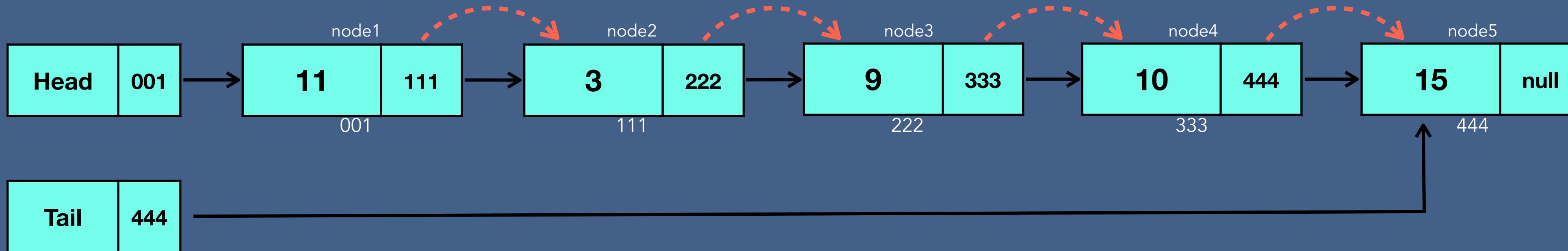
`Tail = node1`

`currentNode.next = null`



# 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  $x$ .

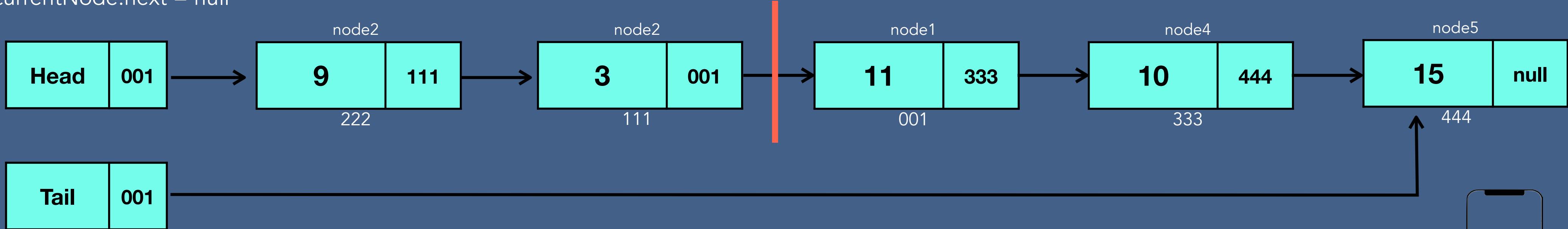


$x = 10$

`currentNode = node1`

`Tail = node1`

`currentNode.next = null`

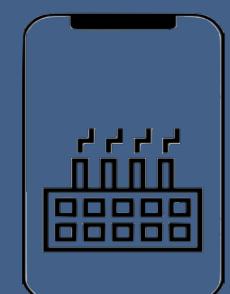
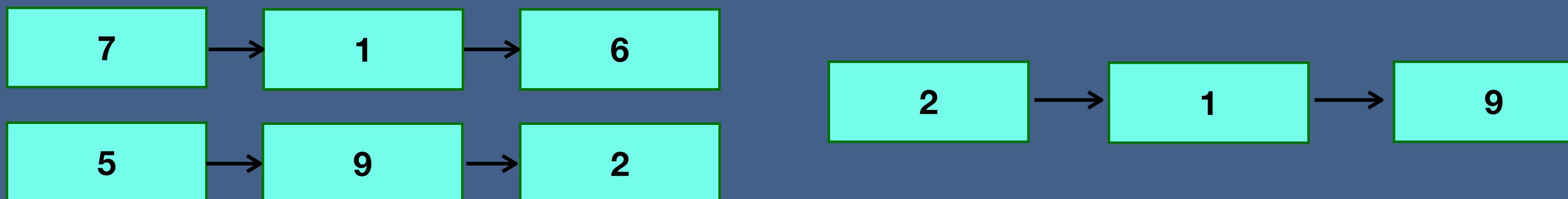


# Sum Lists

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

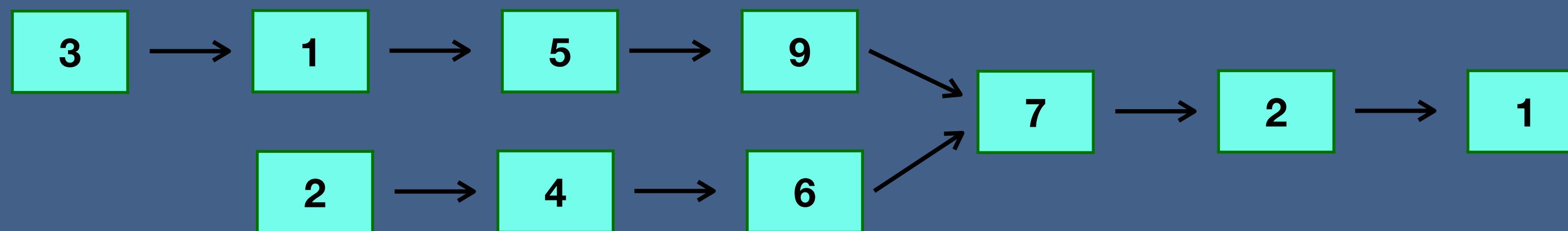
$$\begin{array}{r} 617 \\ + 295 \\ \hline 912 \end{array} \quad \begin{array}{l} 7 + 5 = 12 \\ 1+9+1 = 11 \\ 6+2+1 = 9 \end{array}$$



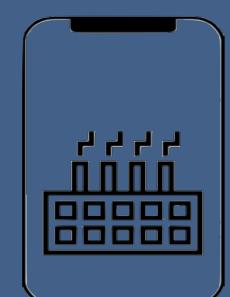
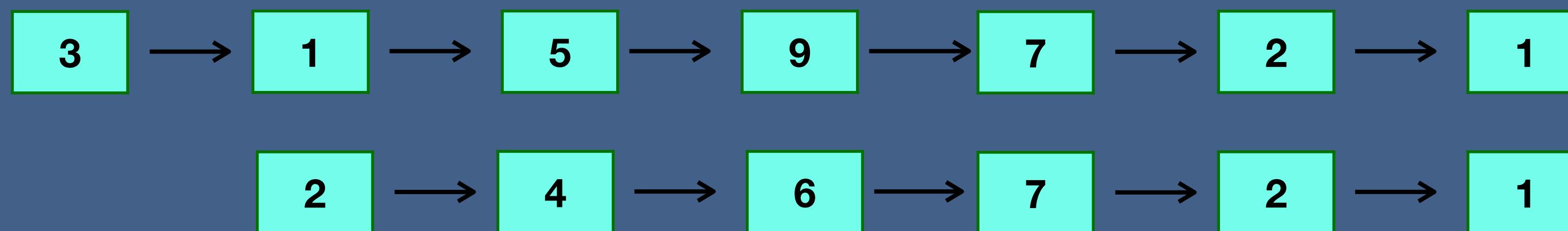
# Intersection

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



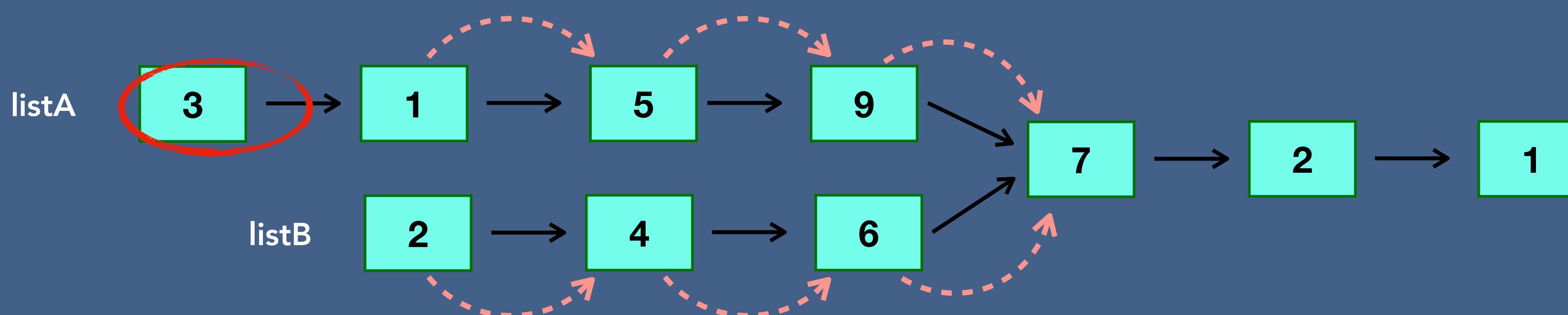
Non - intersecting linked lists



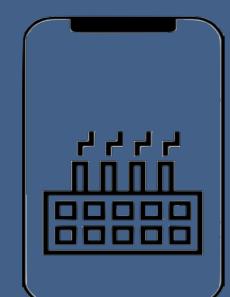
# Intersection

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  $k$ th node of the first linked list is the exact same node (by reference) as the  $j$ th node of the second linked list, then they are intersecting.

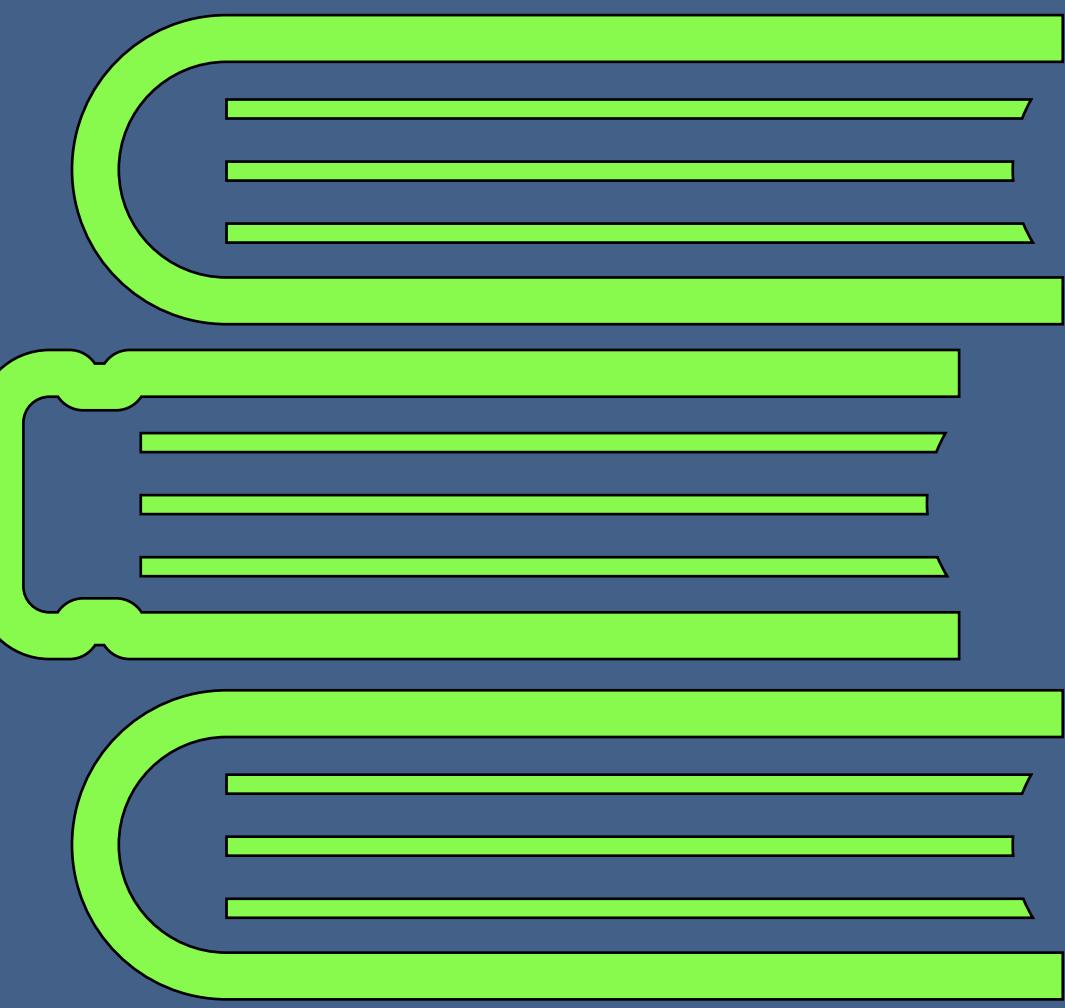
## Intersecting linked lists



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

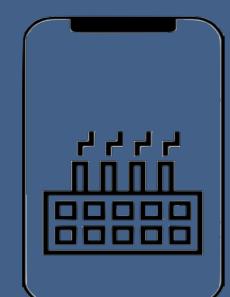
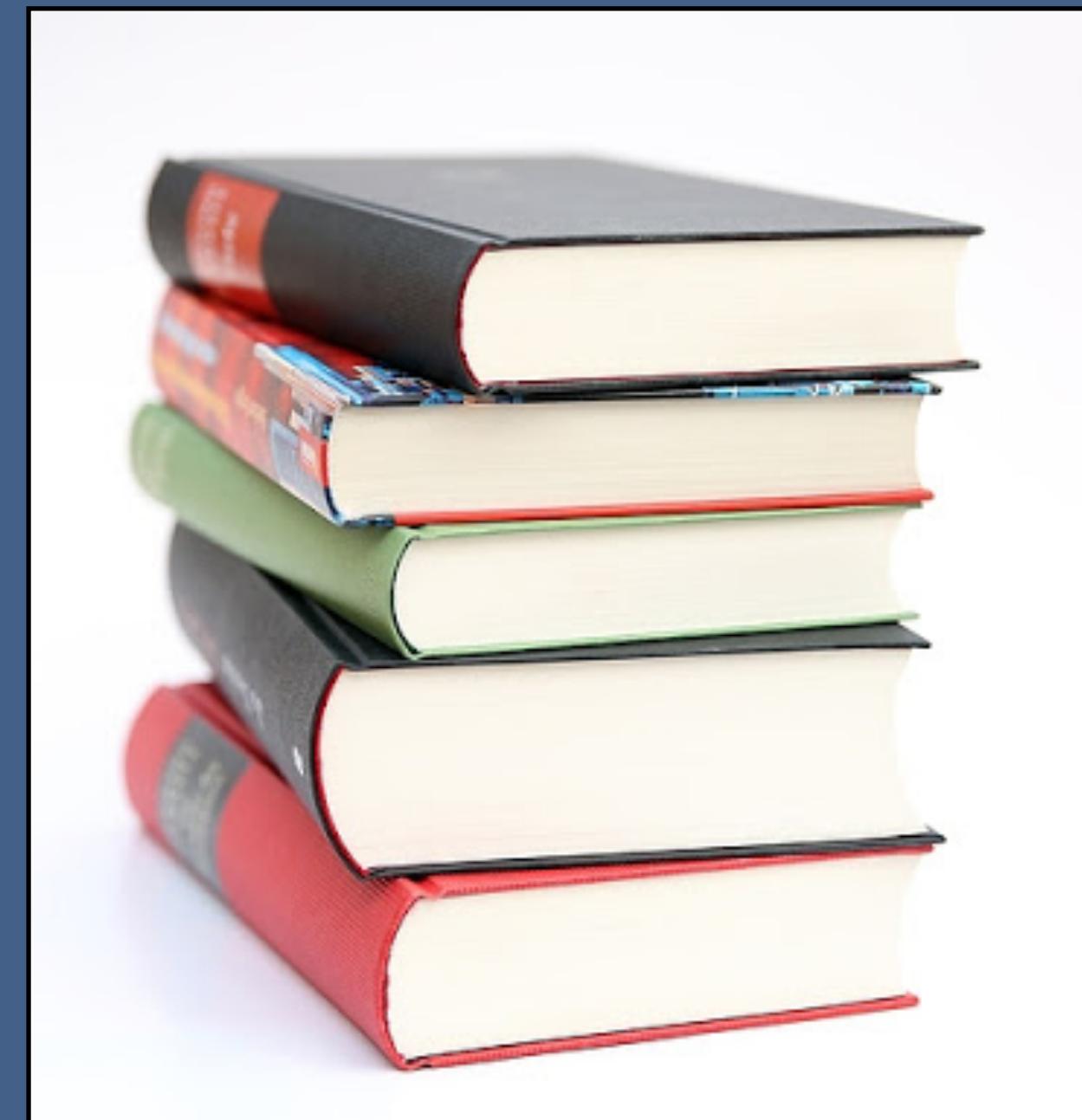


# Stack



# What is a Stack?

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

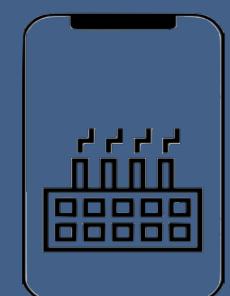


# What is a Stack?

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

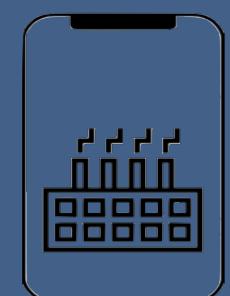
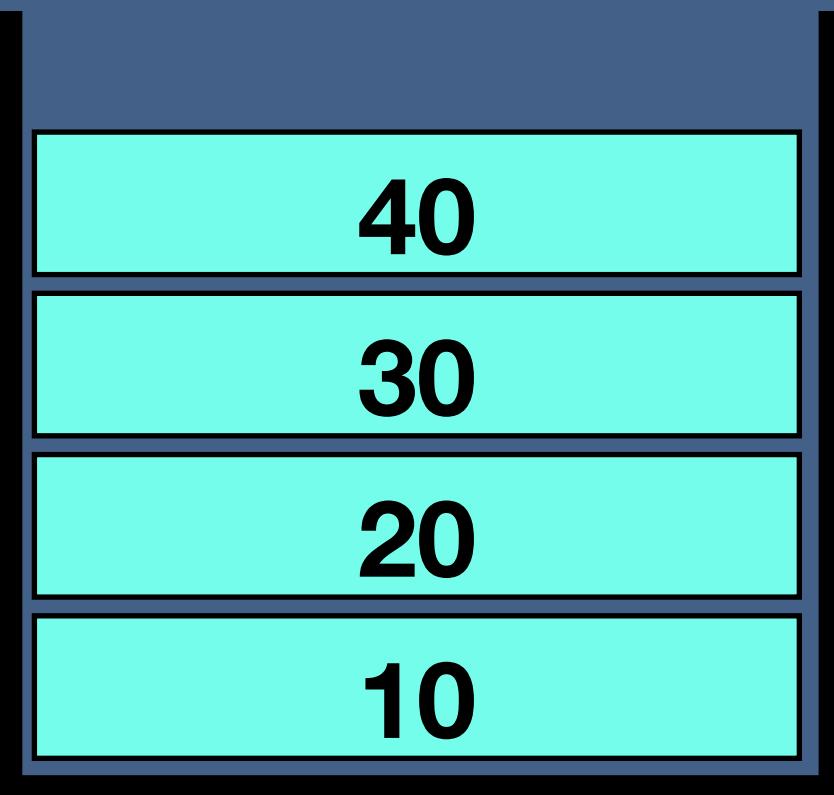


LIFO method

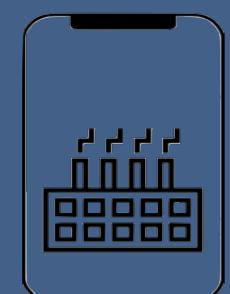
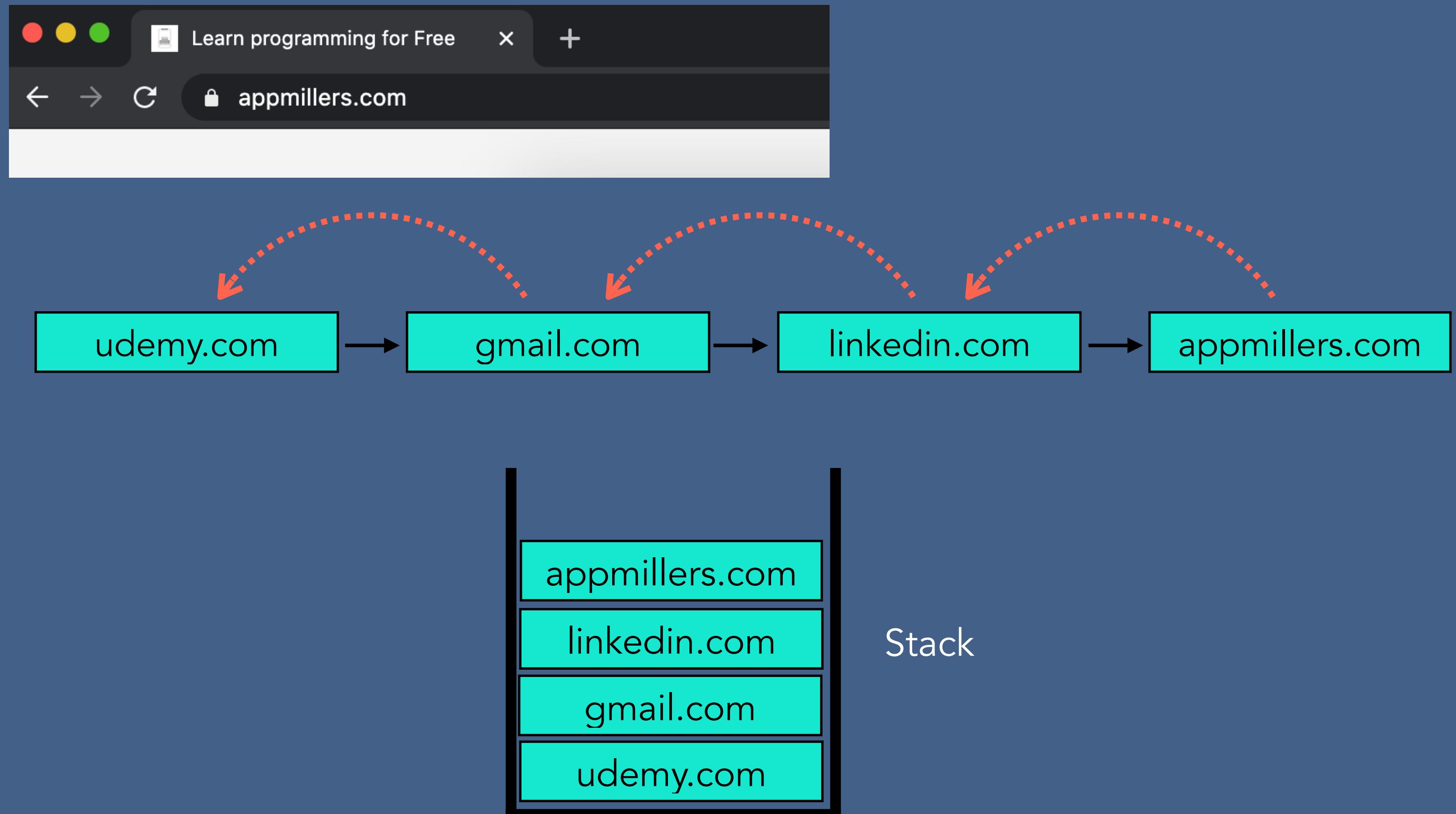


# What is a Stack?

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



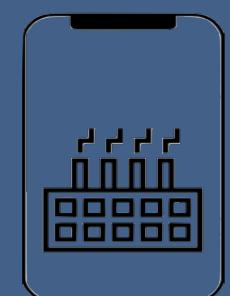
# Why do we need Stack?



# Stack Operations

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

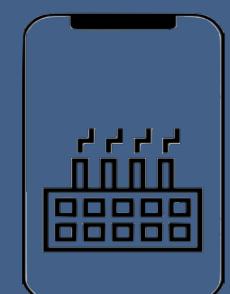
customStack()



# Push Method

```
customStack = [ ]
```

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

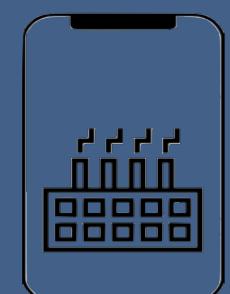



# Push Method

customStack = [1]

customStack.push(2)

					1				

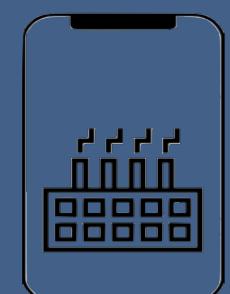


# Push Method

customStack = [1,2]

customStack.push(3)

					2				
					1				

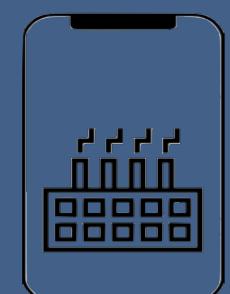


# Push Method

customStack = [1,2,3]

customStack.push(4)

					3				
					2				
					1				

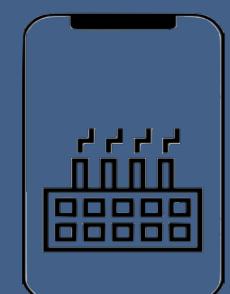


# Push Method

customStack = [1,2,3,4]

customStack.push(4)

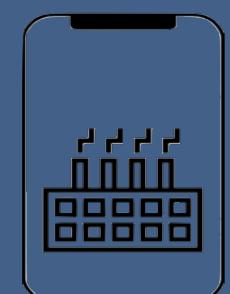
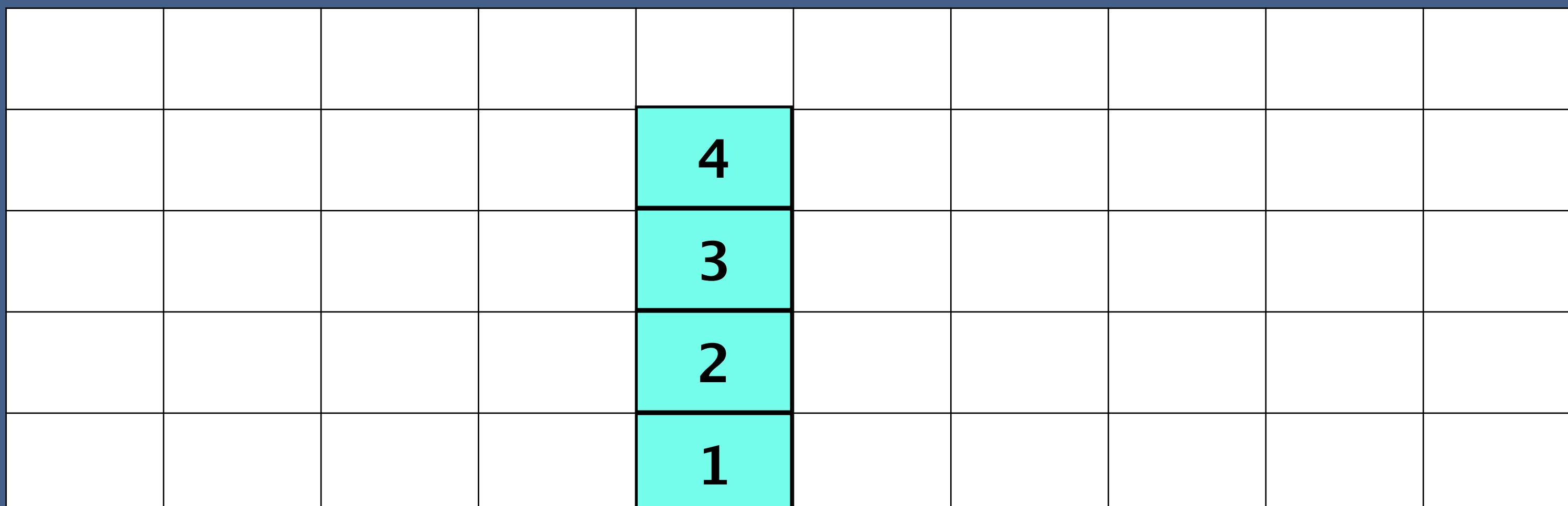
				4					
				3					
				2					
				1					



# Pop Method

customStack = [1,2,3]4]

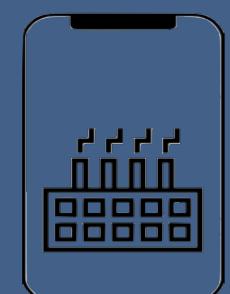
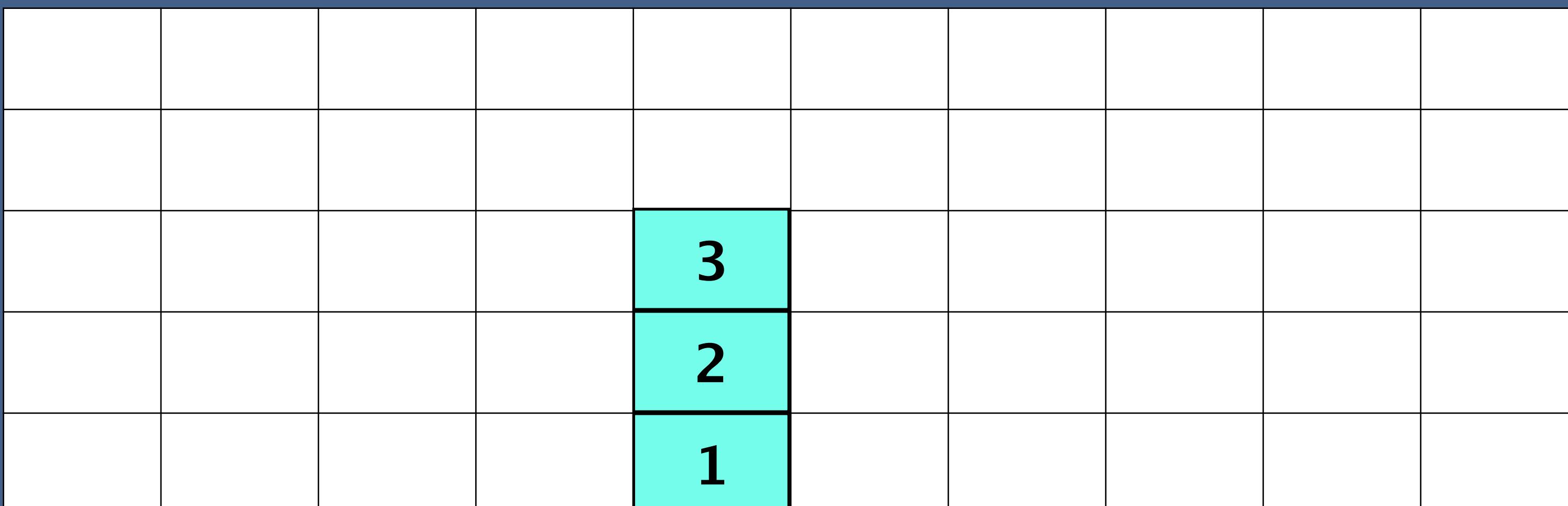
customStack.pop() → 4



# Pop Method

customStack = [1,2]3]

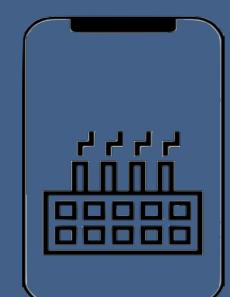
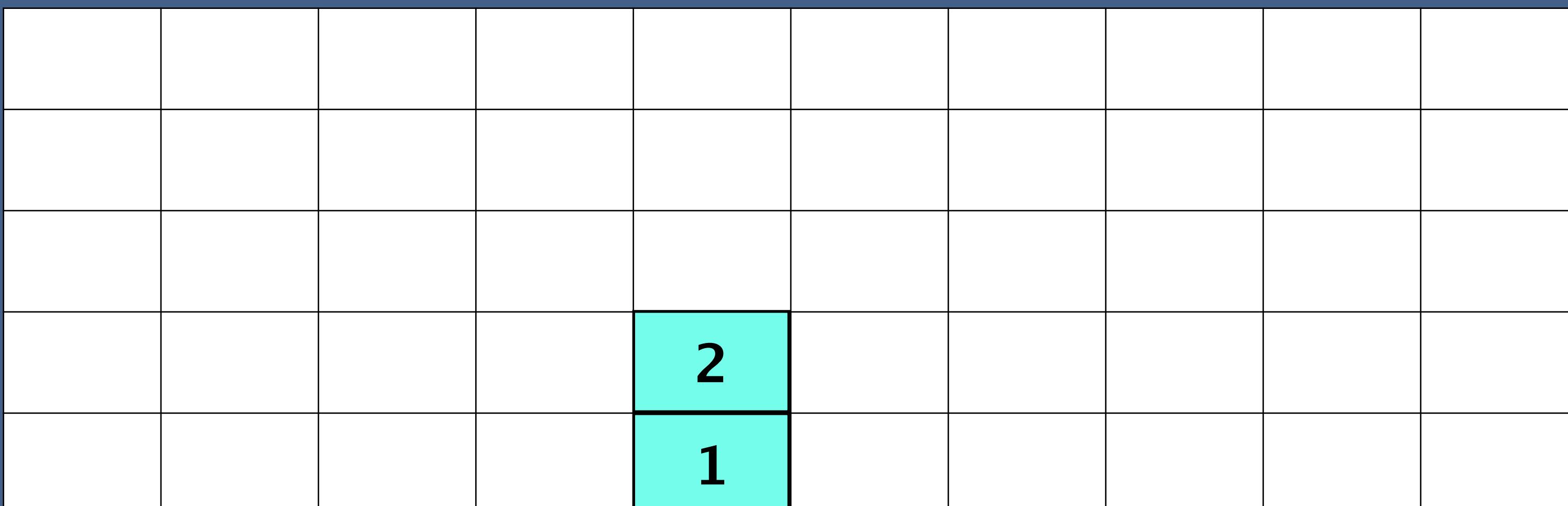
customStack.pop() → 3



# Pop Method

customStack = [1]2

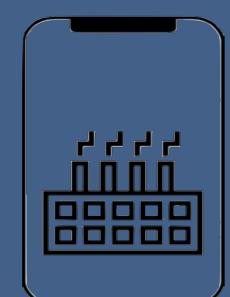
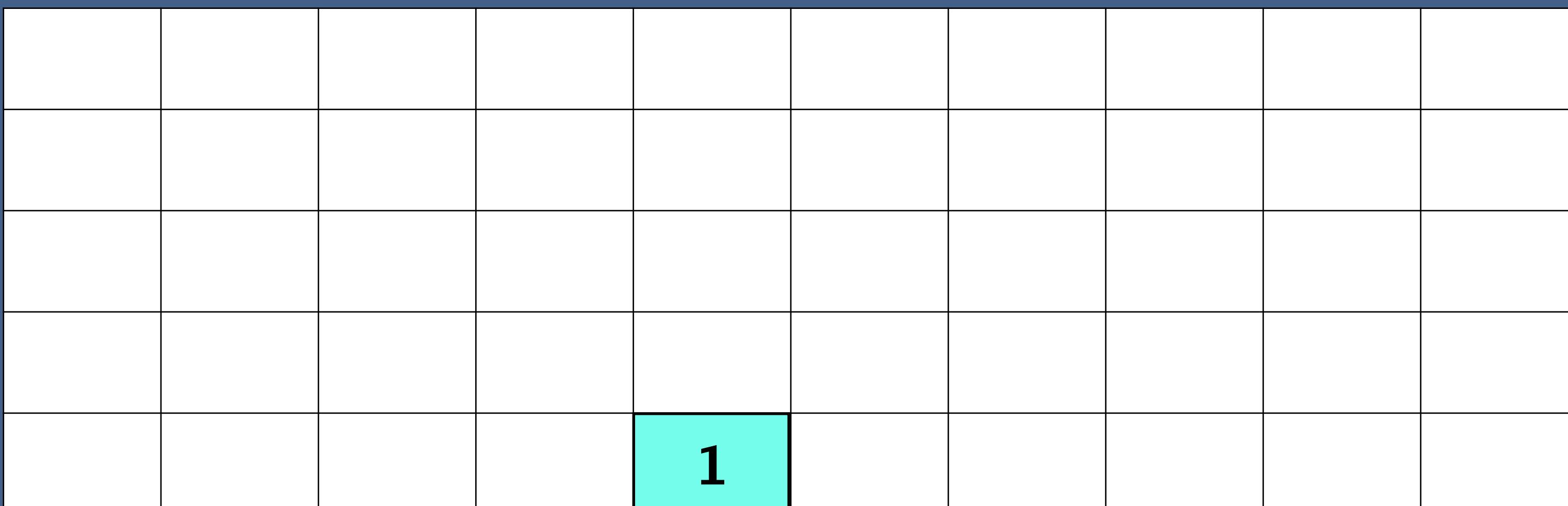
customStack.pop() → 2



# Pop Method

customStack = [ ]

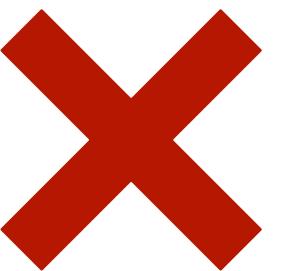
customStack.pop() → 1



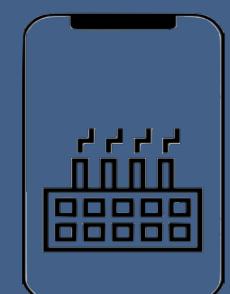
# Pop Method

```
customStack = []
```

```
customStack.pop()
```



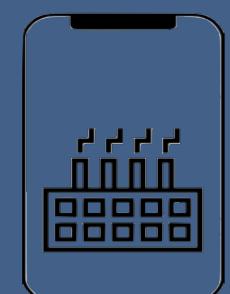
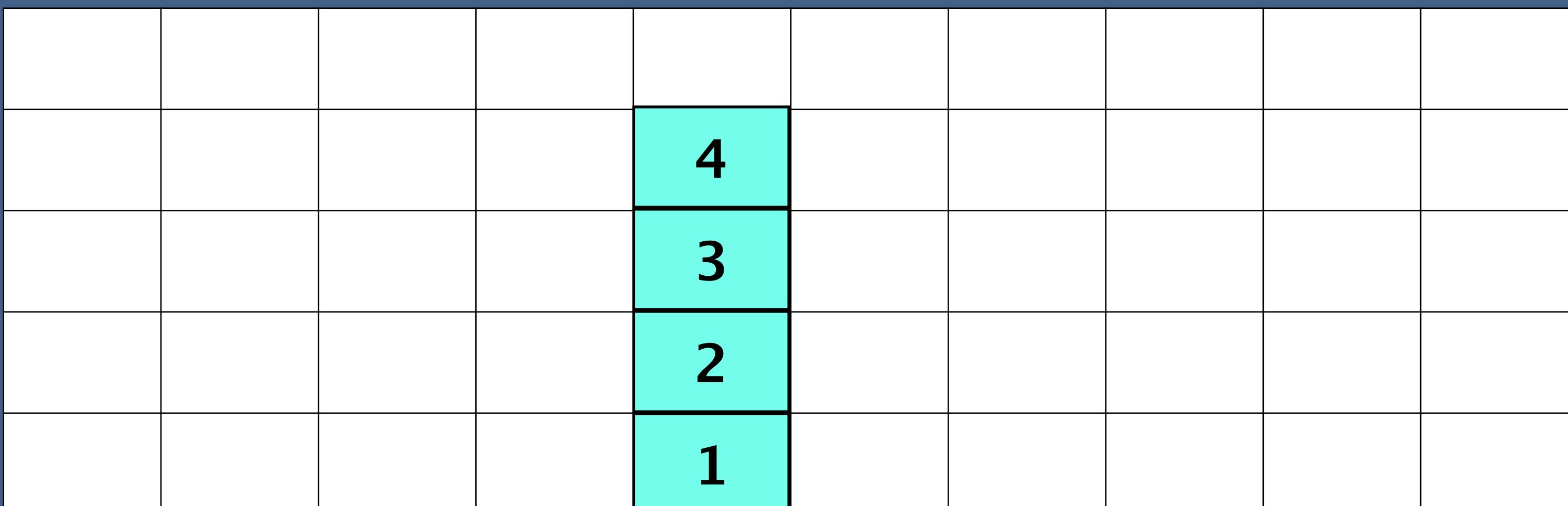
The stack is Empty

# Peek Method

customStack = [1,2,3,4]

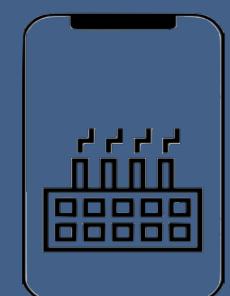
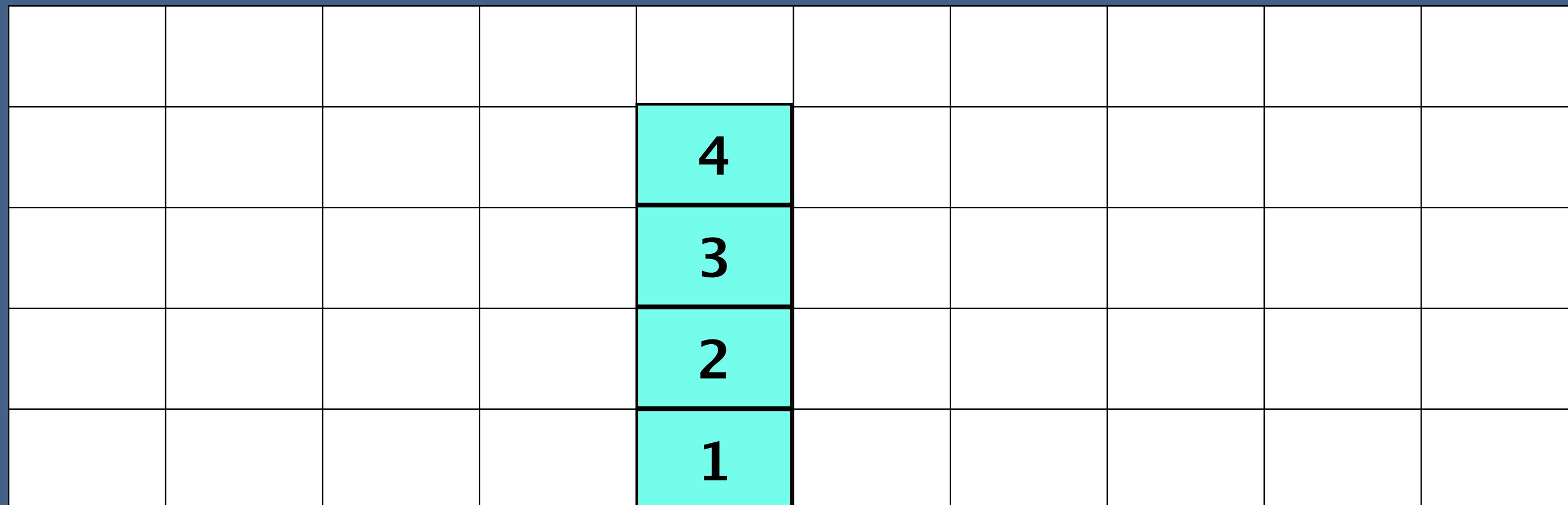
customStack.peek() —————→ 4



# isEmpty Method

customStack = [1,2,3,4]

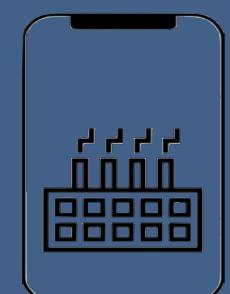
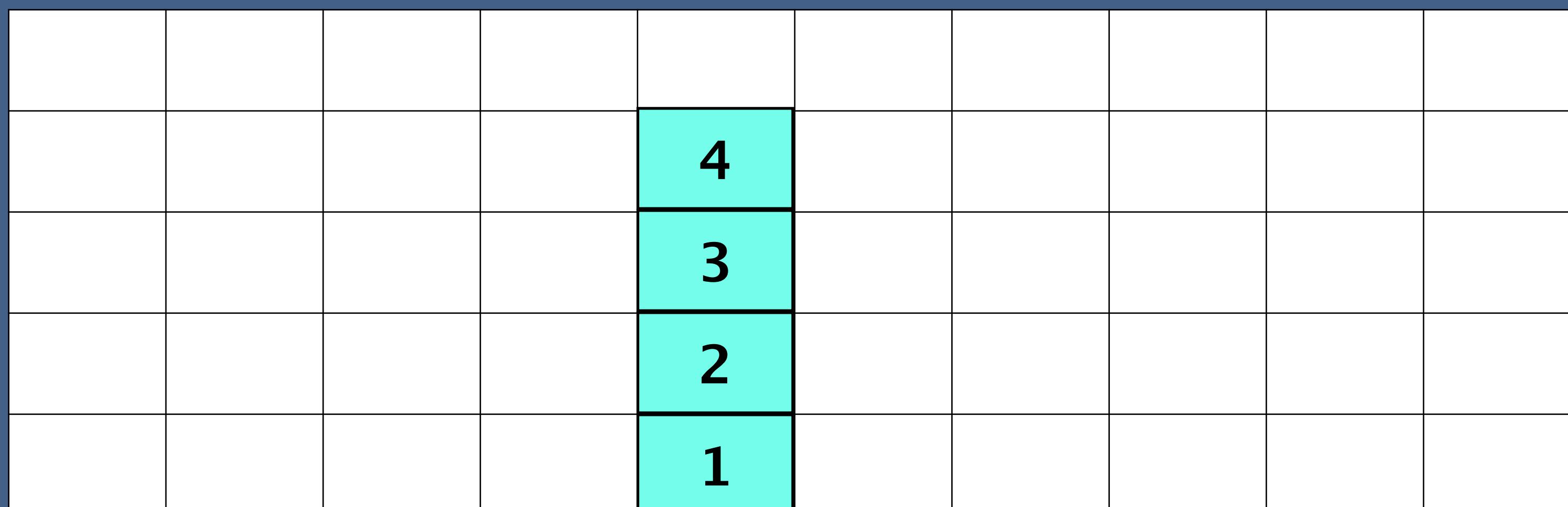
customStack.isEmpty() → False



# isFull Method

customStack = [1,2,3,4]

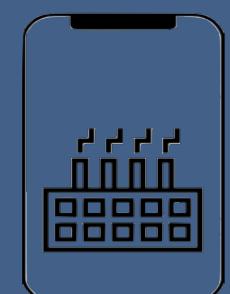
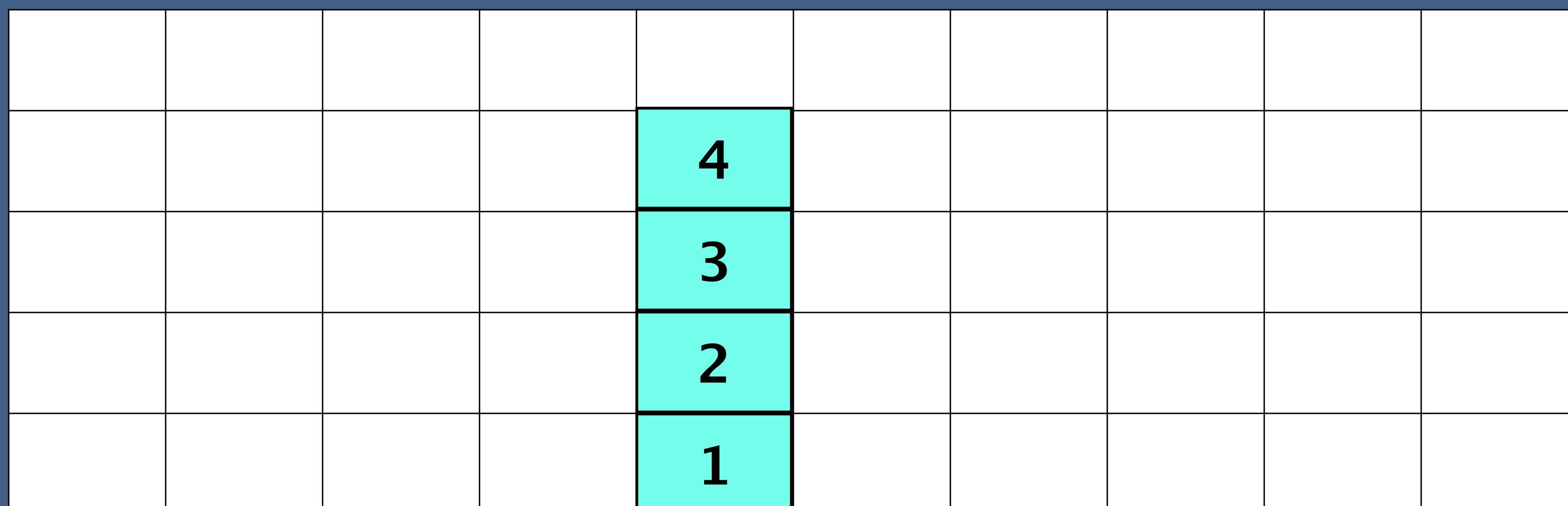
customStack.isFull() → False



# Delete Method

customStack = [1,2,3,4]

customStack.delete()



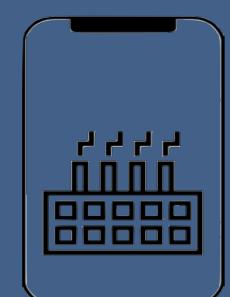
# Stack Creation - Array vs Linked List

## Stack using Array

- Easy to implement
- Fixed size

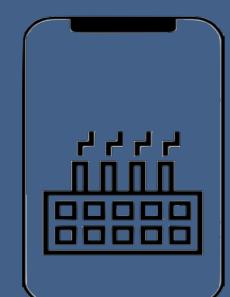
## Stack using Linked List

- Variable size
- Implementation is not easy



# Time and Space Complexity of Stack using Array

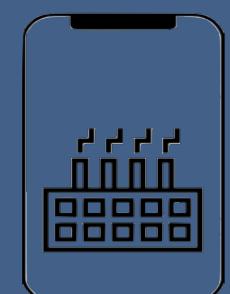
Stack	Time complexity	Space complexity
Create Stack	$O(1)$	$O(n)$
Push	$O(1)$	$O(1)$
Pop	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
Delete Entire Stack	$O(1)$	$O(1)$



# Stack using Linked List

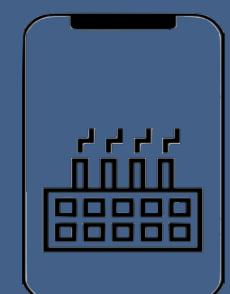
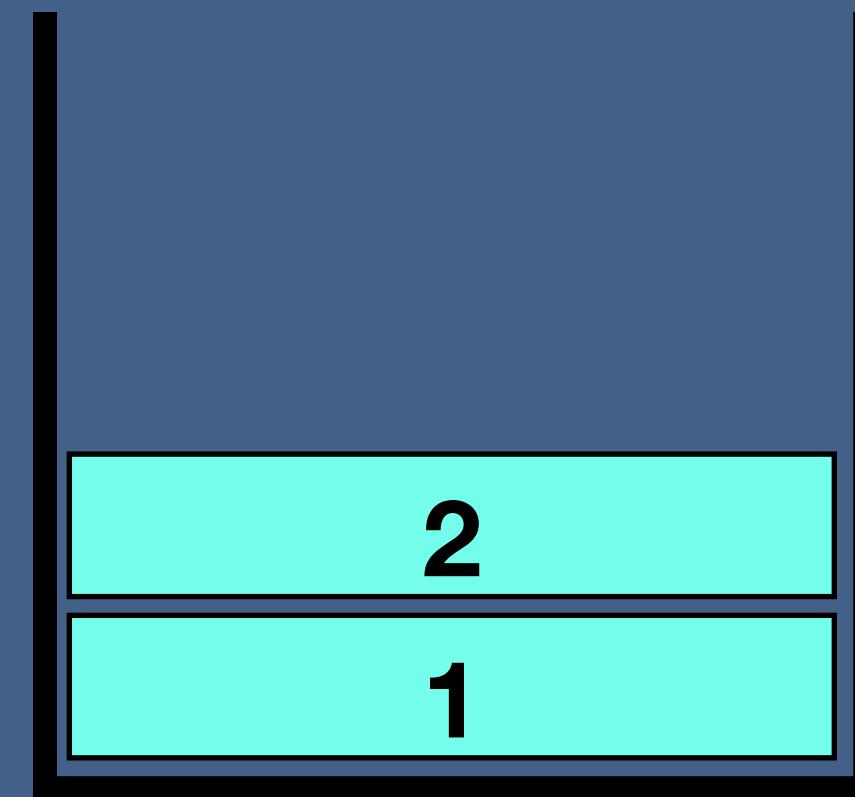
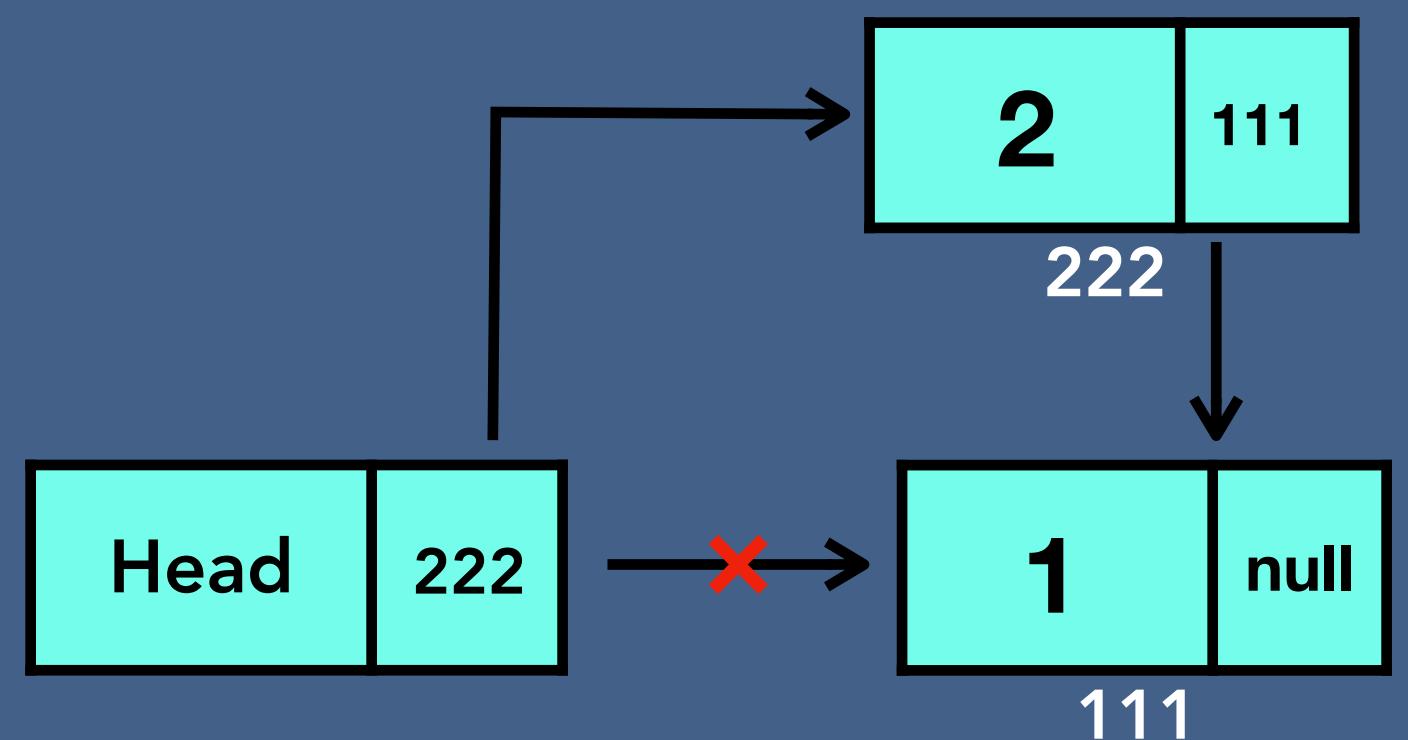
## Create a Stack

Create an object of Linked List class



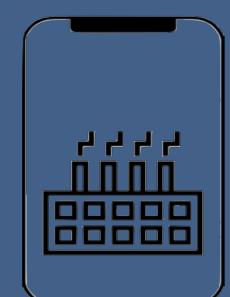
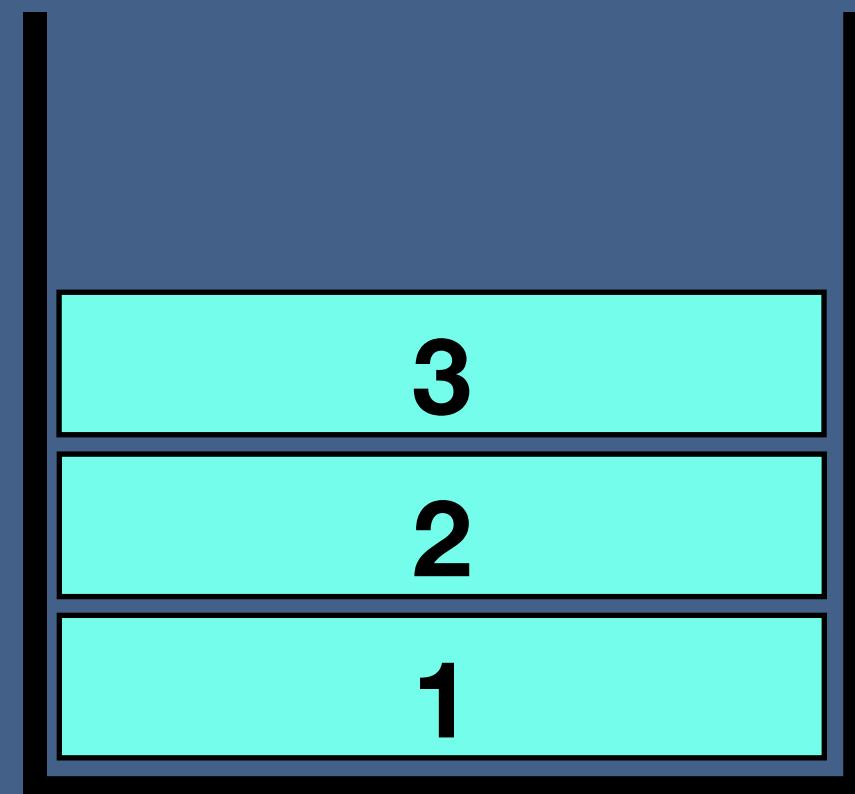
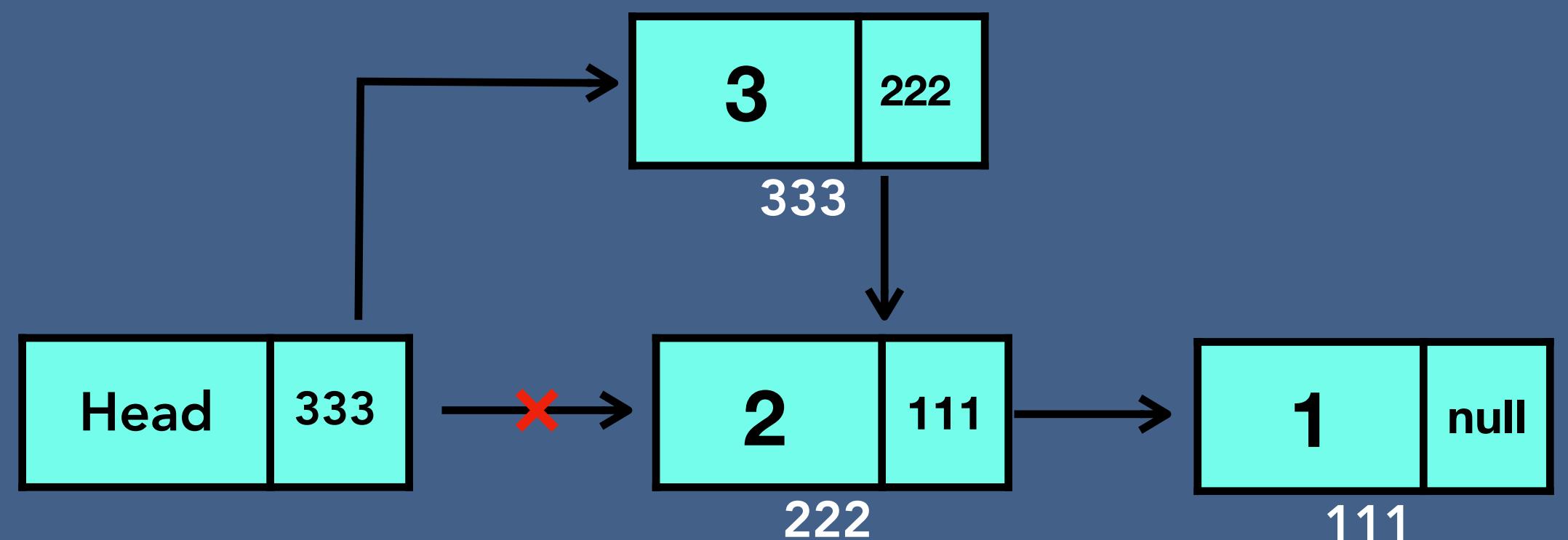
# Stack using Linked List

## Push() Method



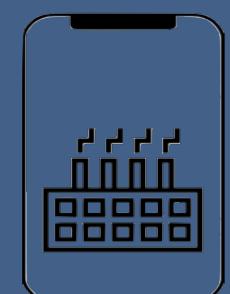
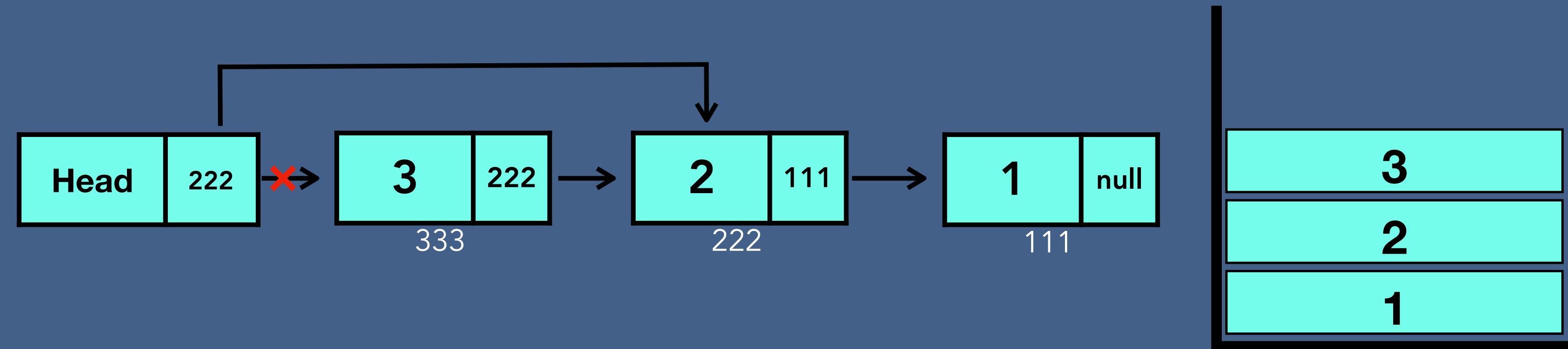
# Stack using Linked List

## Push() Method



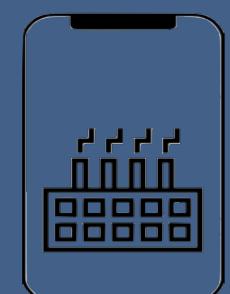
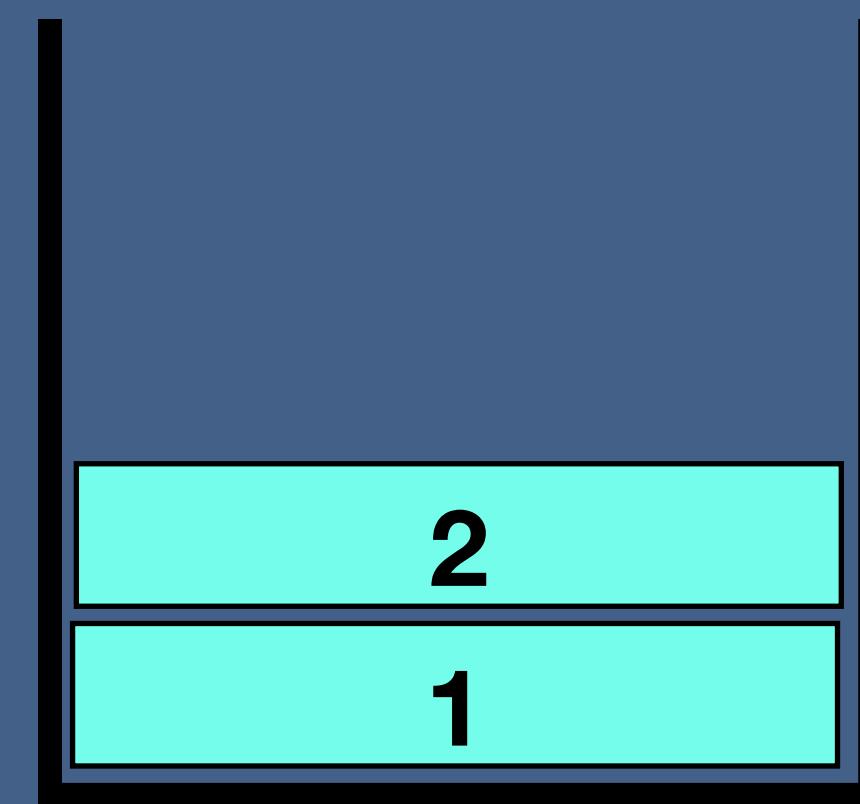
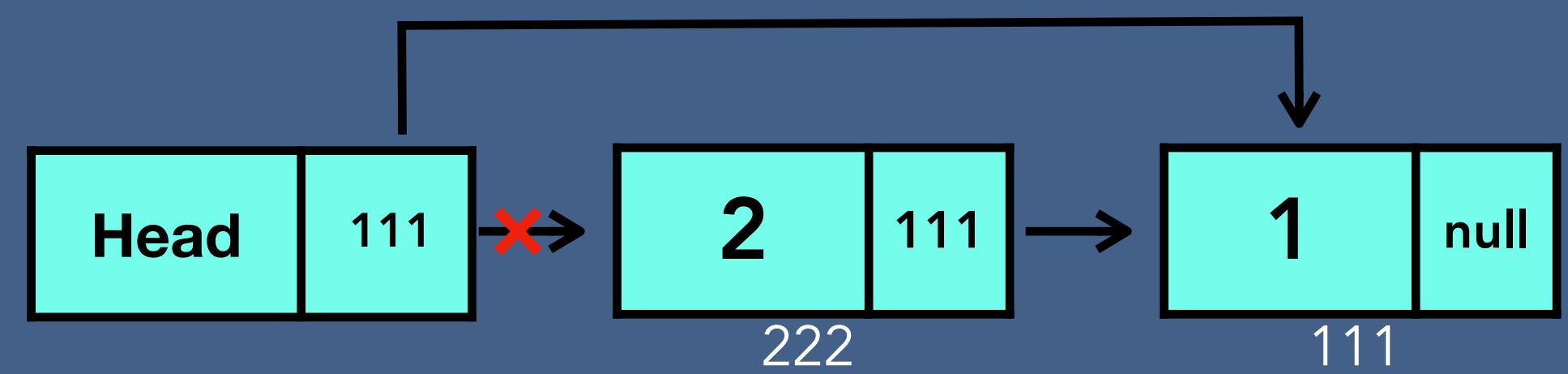
# Stack using Linked List

## Pop() Method



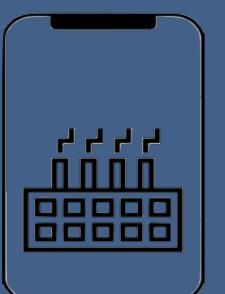
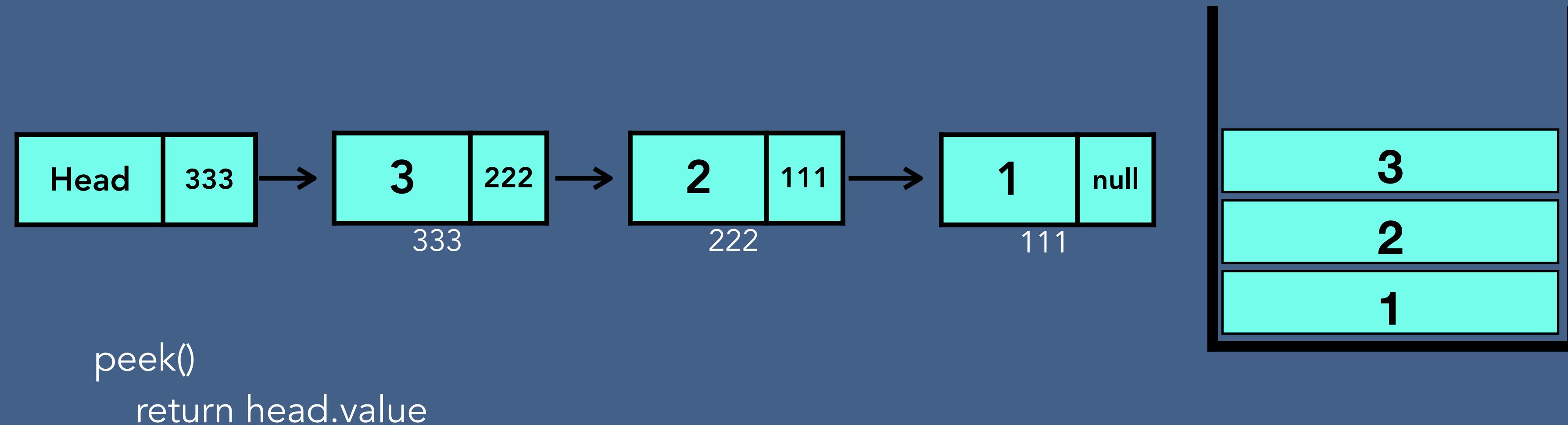
# Stack using Linked List

## Pop() Method



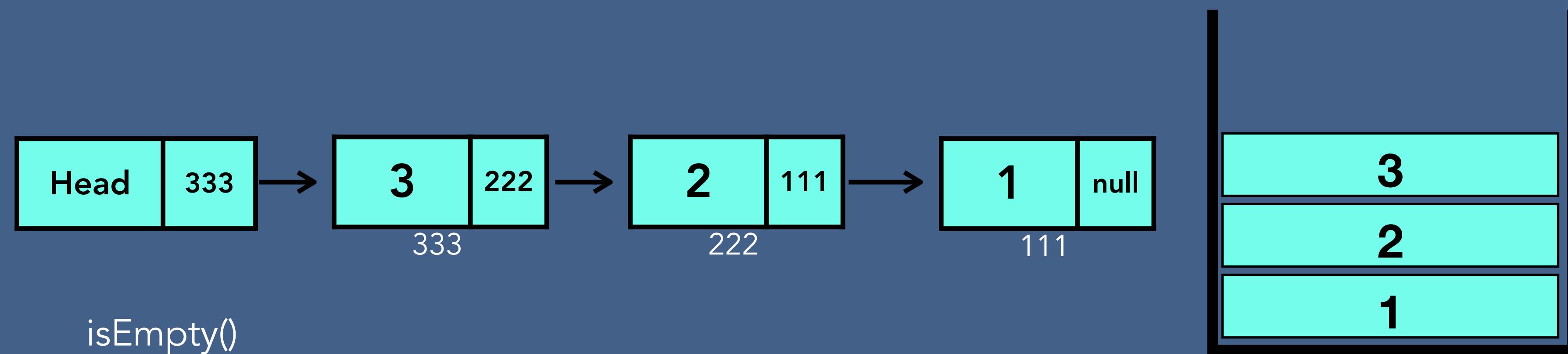
# Stack using Linked List

## Peek() Method

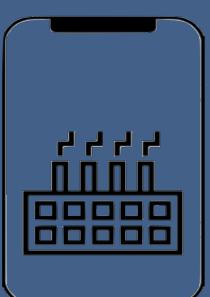


# Stack using Linked List

## isEmpty() Method

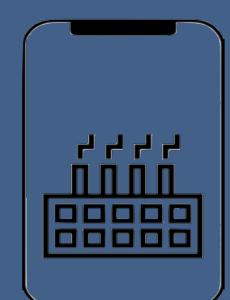
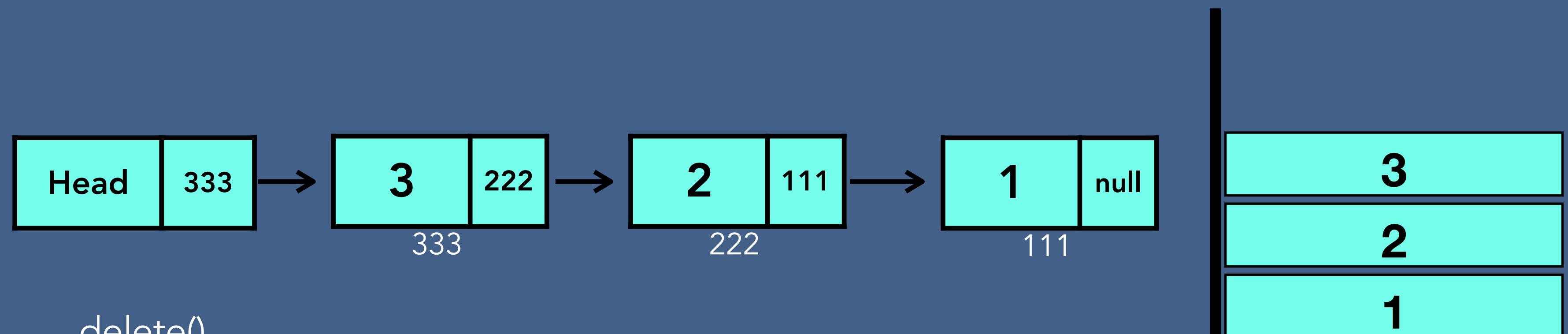


```
isEmpty()  
If head==null {  
    true  
}
```



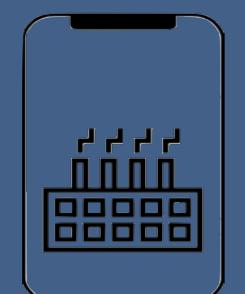
# Stack using Linked List

## delete() Method



# Time and Space Complexity of Stack using Linked List

Stack	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	$O(1)$	$O(1)$



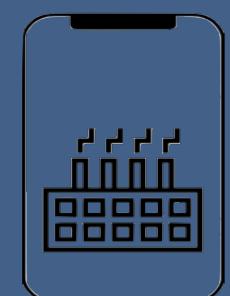
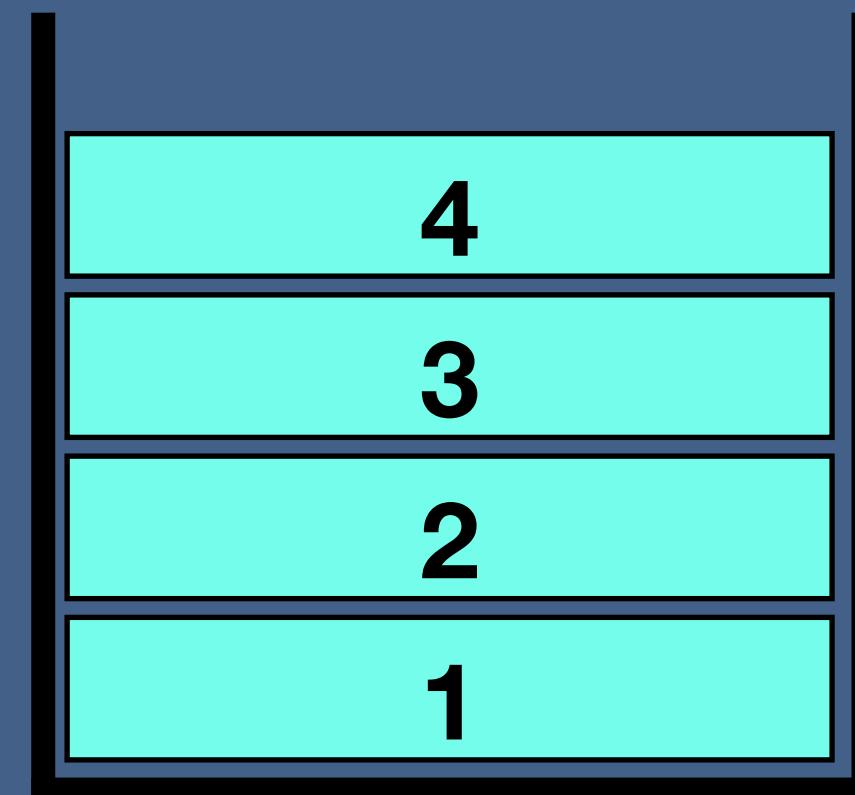
# When to Use/Avoid Stack

## Use:

- LIFO functionality
- The chance of data corruption is minimum

## Avoid:

- Random access is not possible



# Queue



# What is a Queue?

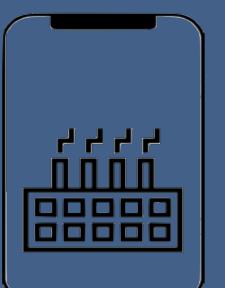
Queue is a data structure that stores items in a First-In/First-Out manner.



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

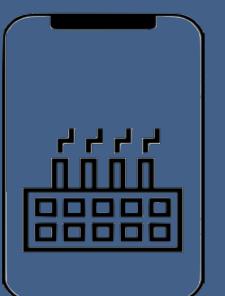
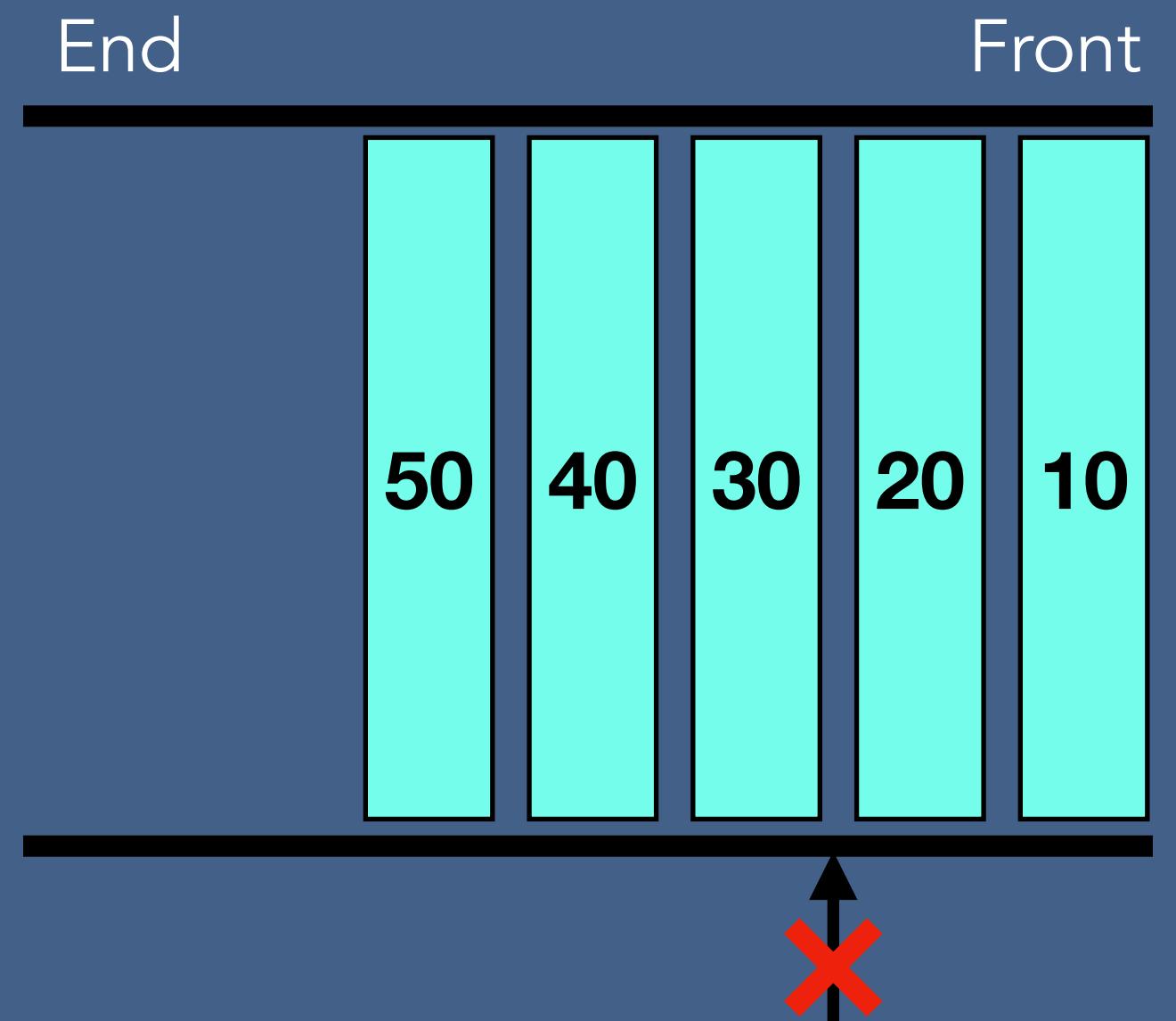
First person in the queue will be served first

FIFO method - First in First Out



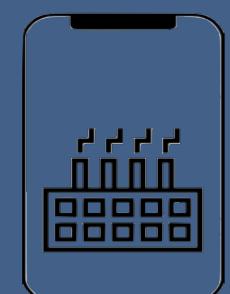
# What is a Queue?

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



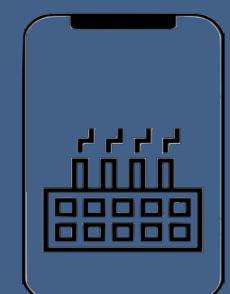
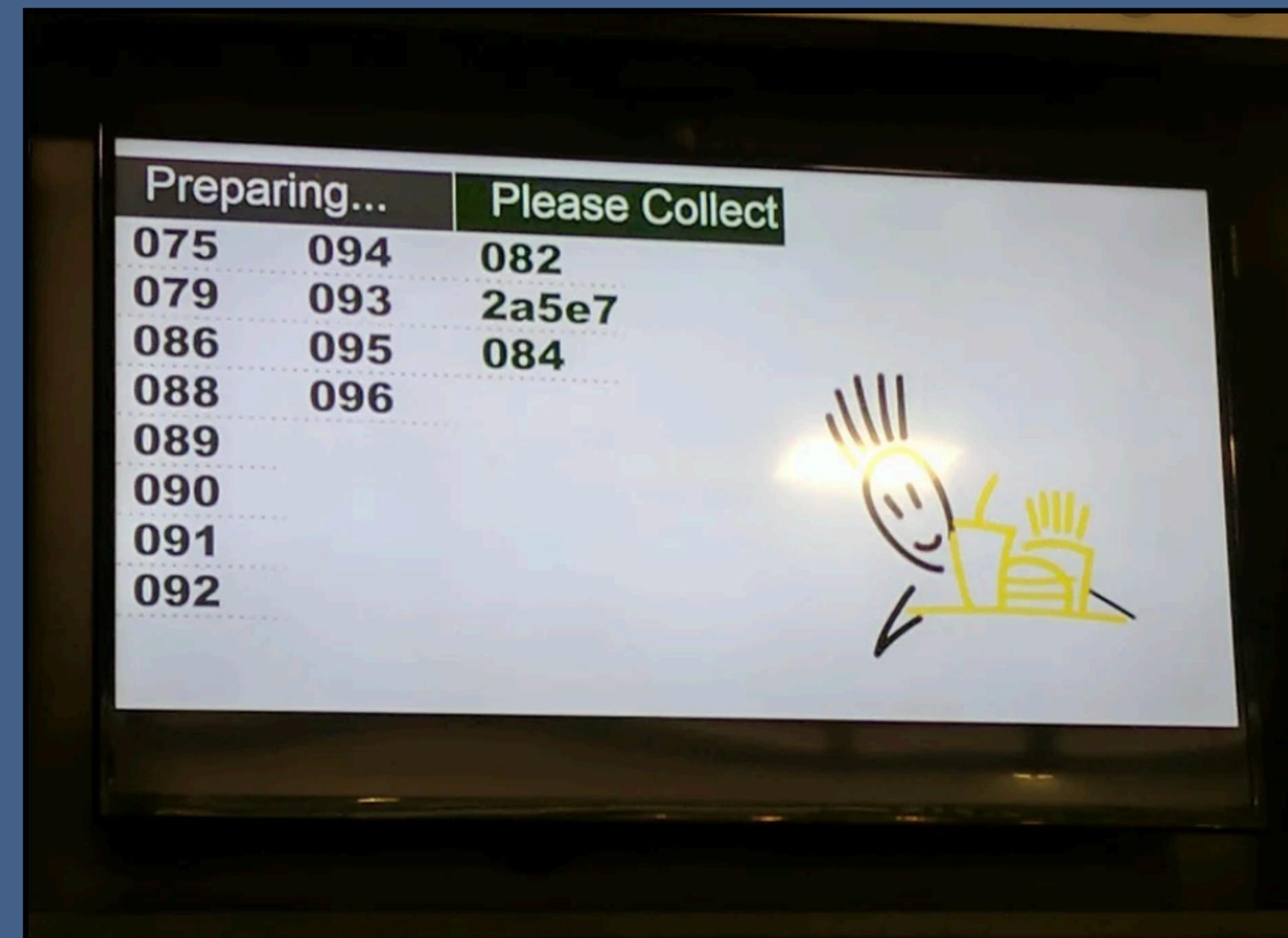
# Why do we need Queue?

- Utilize first coming data first , while others wait for their turn.
- FIFO method - First In First Out



# Why do we need Queue?

Point sale system of a restaurant

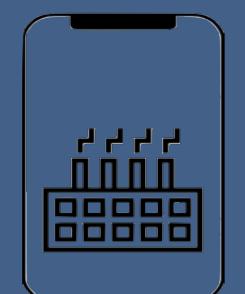


# Why do we need Queue?

Printer queue

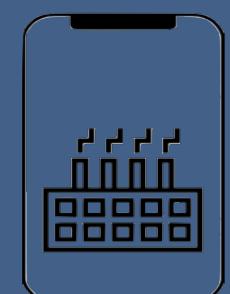
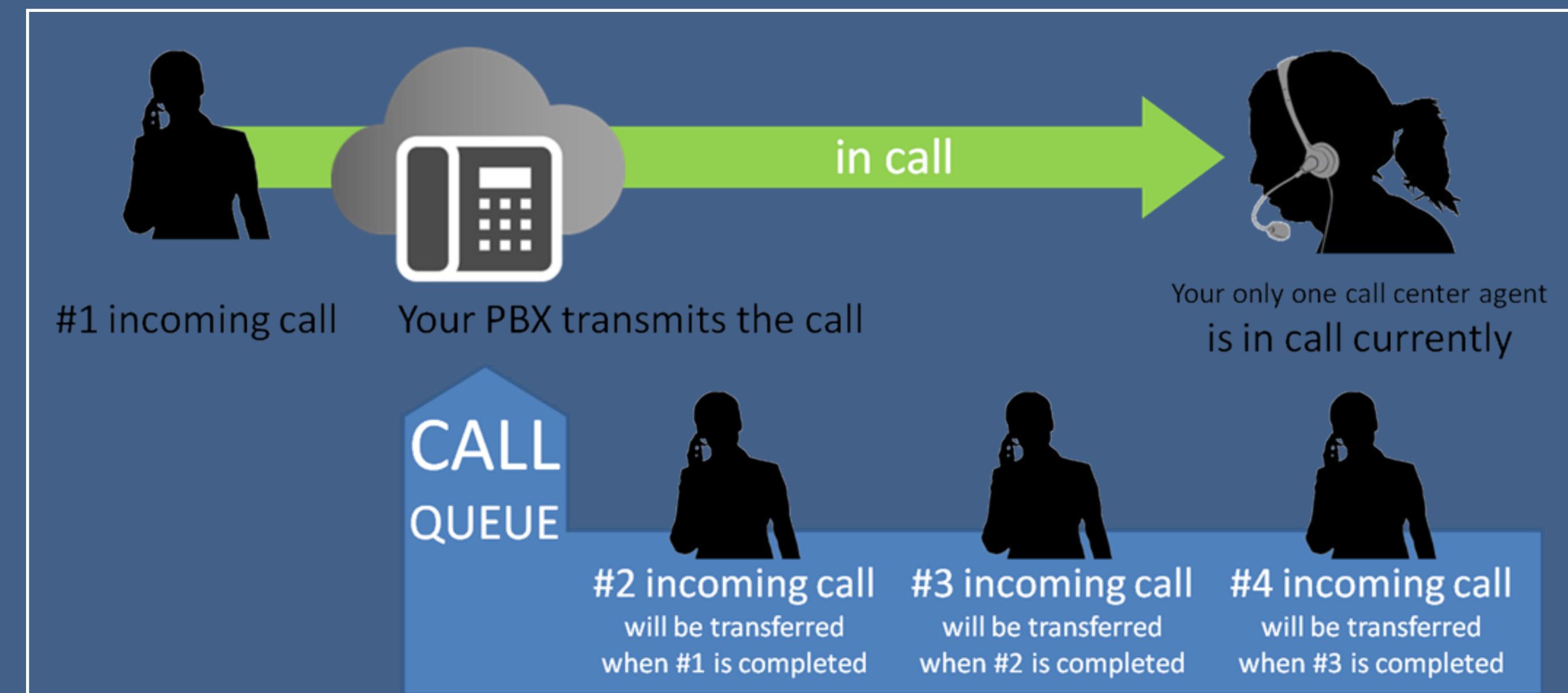
Brother HL-2040 series on PAVILIONWIN7 - Paused					
Document Name	Status	Owner	Pages	Size	Submitted
Microsoft Word - Renovation email.doc	HomeGroupUser\$	4	1.08 MB	9:34:27 PM 4/27/2009	
Microsoft Word - Test2.docx	HomeGroupUser\$	1	1.63 KB	9:32:55 PM 4/27/2009	
Microsoft Word - Mutual of America inquiry.doc	Woody	1	13.7 KB	9:32:32 PM 4/27/2009	
Microsoft Word - Allow Bongkod to pick up Ad...	Woody	2	5.05 MB	9:31:24 PM 4/27/2009	

4 document(s) in queue



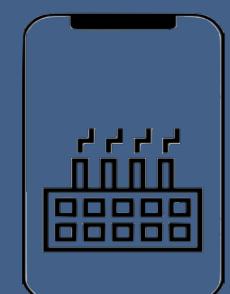
# Why do we need Queue?

Call center phone systems



# Queue Operations

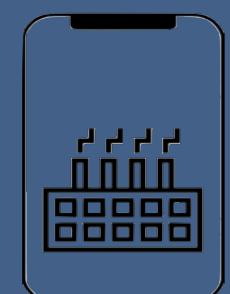
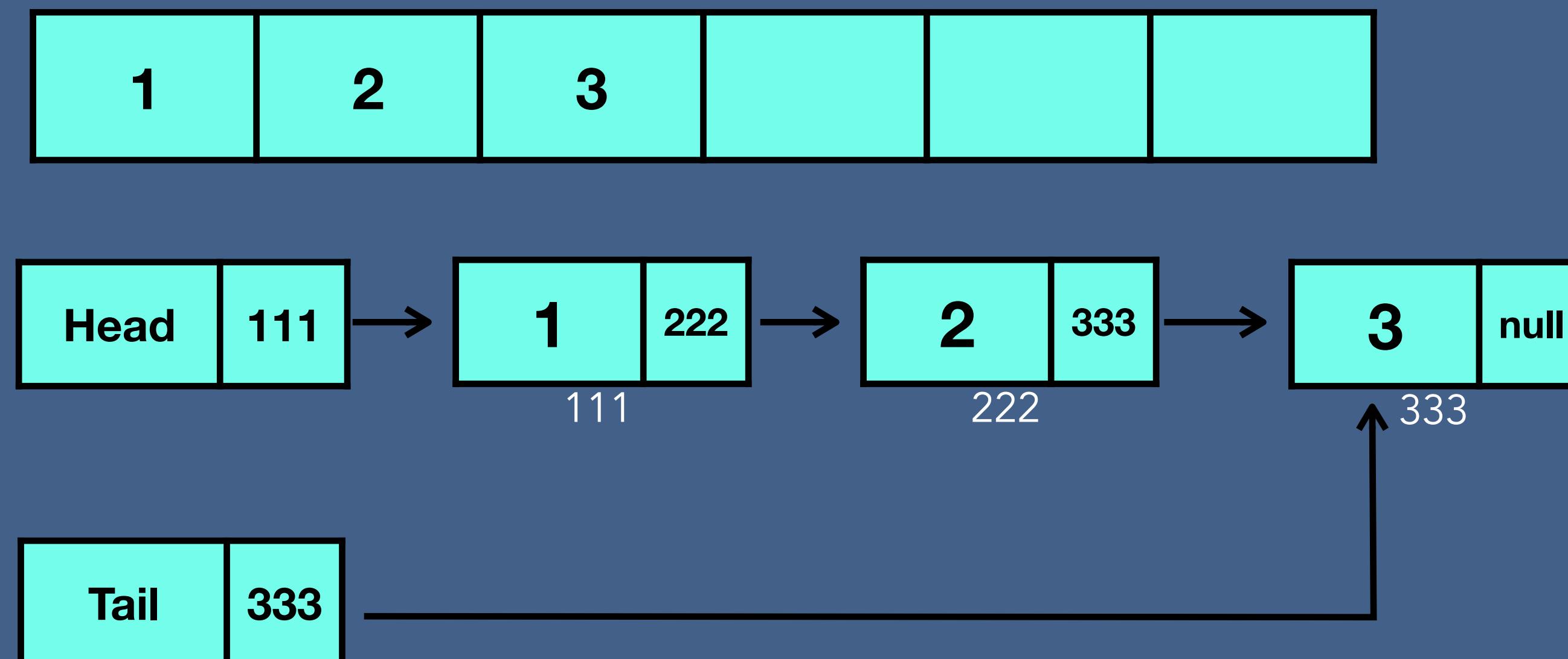
- Create Queue
- Enqueue
- Dequeue
- Peek
- isEmpty
- isFull
- deleteQueue



# Queue Operations

## Implementation

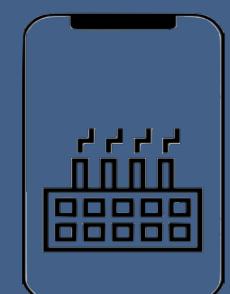
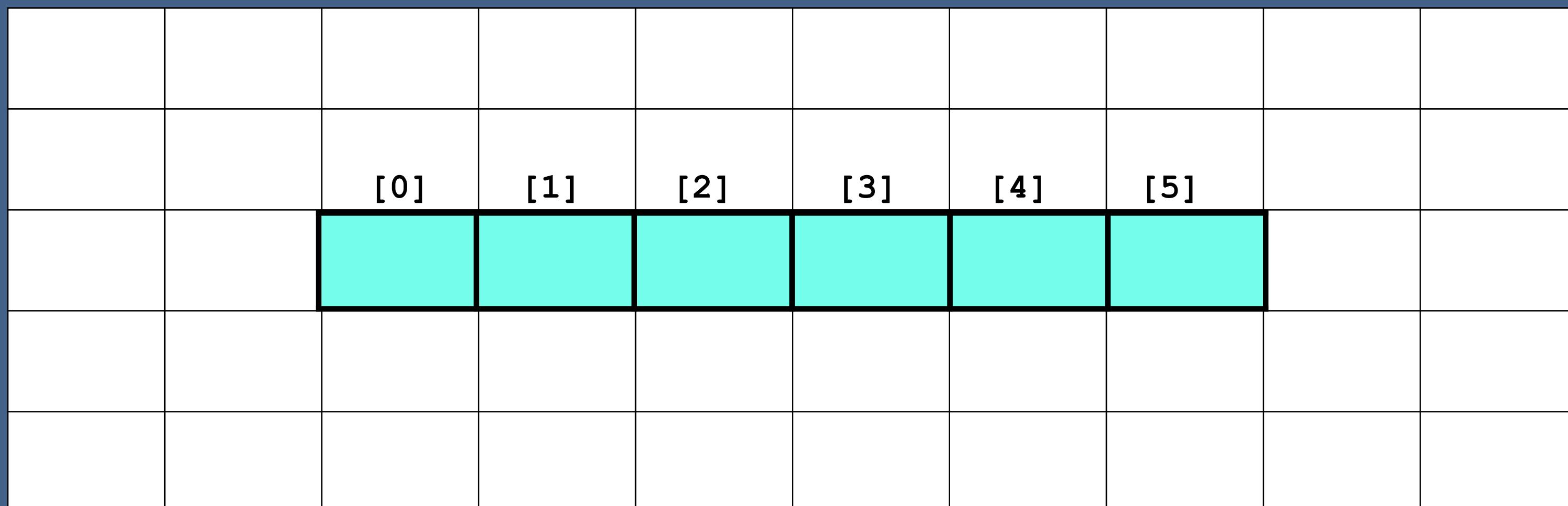
1. Array
  - Linear Queue
  - Circular Queue
2. Linked List



# Linear Queue using Array

## Create a Queue

```
newQueue = Queue(6)  
beginningOfQueue = -1  
topOfQueue = -1
```



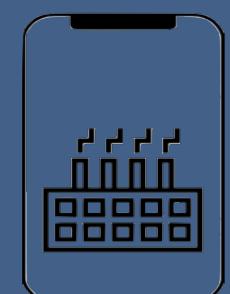
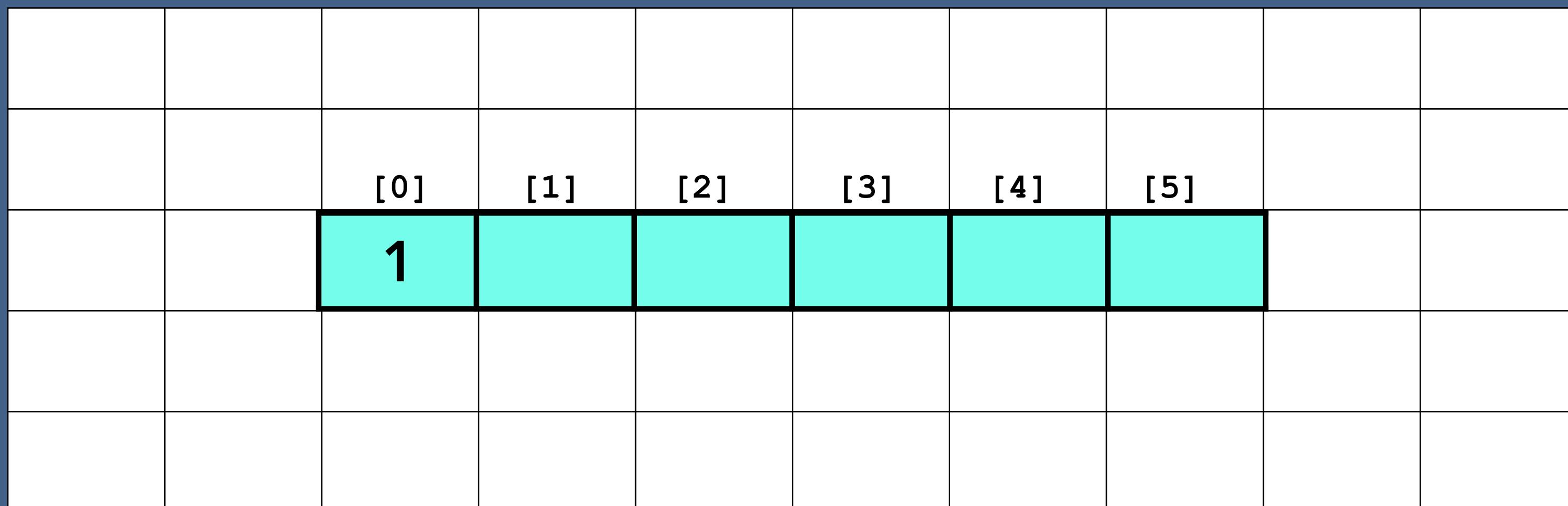
# Linear Queue using Array

## enQueue Method

```
newQueue.enqueue(1)
```

```
beginningOfQueue = 0
```

```
topOfQueue = 0
```



# Linear Queue using Array

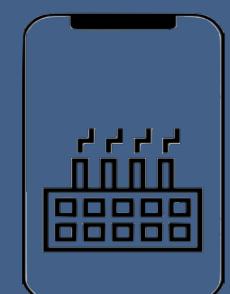
## enQueue Method

```
newQueue.enqueue(2)
```

```
beginningOfQueue = 0
```

```
topOfQueue = 1
```

		[0]	[1]	[2]	[3]	[4]	[5]		
		1	2						



# Linear Queue using Array

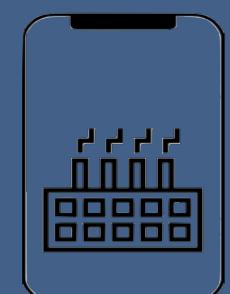
## enQueue Method

```
newQueue.enqueue(3)
```

```
beginningOfQueue = 0
```

```
topOfQueue = 2
```

		[0]	[1]	[2]	[3]	[4]	[5]		
		1	2	3					



# Linear Queue using Array

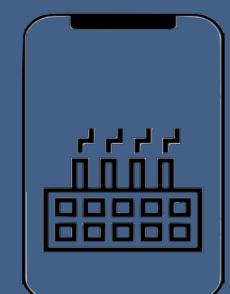
## deQueue Method

newQueue.dequeue() → 1

beginningOfQueue = 0

topOfQueue = 2

		[0]	[1]	[2]	[3]	[4]	[5]		
		1	2	3					



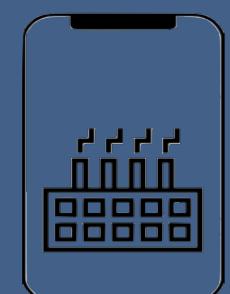
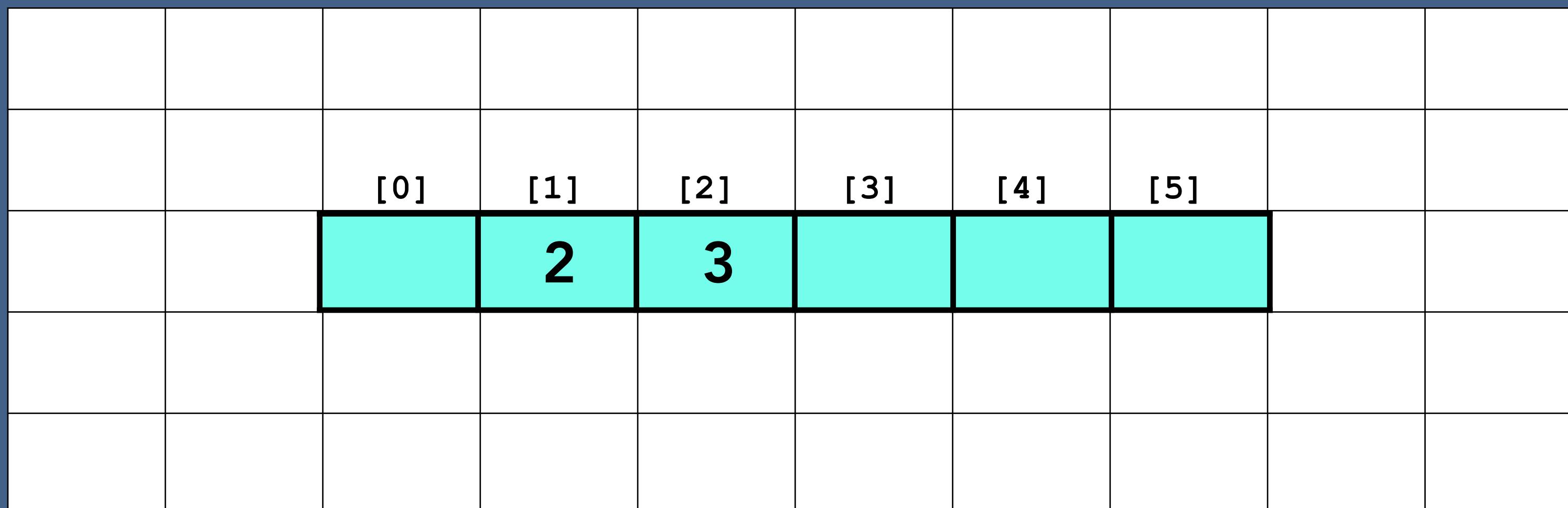
# Linear Queue using Array

## deQueue Method

newQueue.dequeue() → 2

beginningOfQueue = 2

topOfQueue = 2



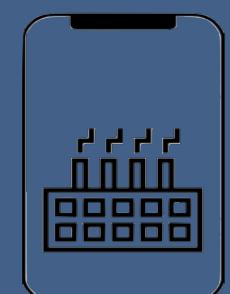
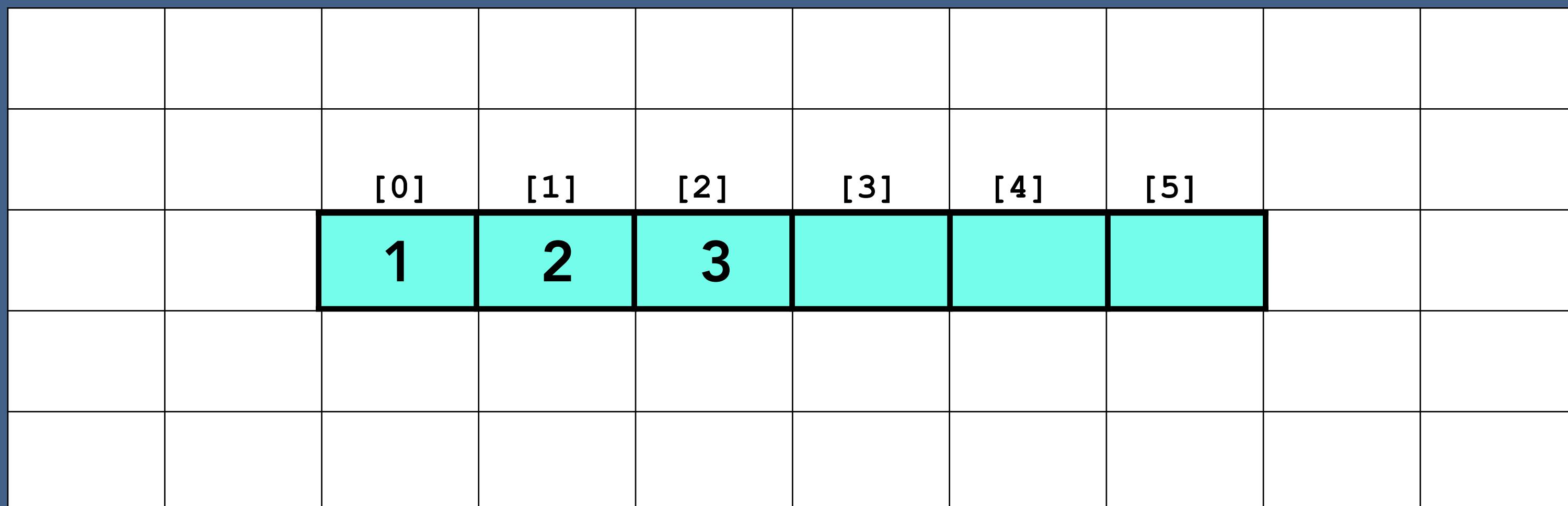
# Linear Queue using Array

## peek Method

newQueue.peek () → 1

beginningOfQueue = 0

topOfQueue = 2



# Linear Queue using Array

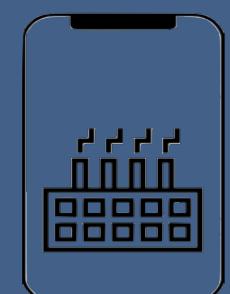
## isEmpty Method

newQueue.isEmpty() → False

beginningOfQueue = 0

topOfQueue = 2

		[0]	[1]	[2]	[3]	[4]	[5]		
		1	2	3					



# Linear Queue using Array

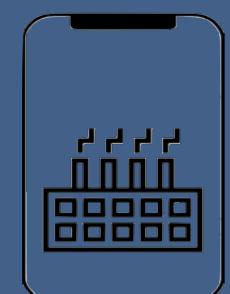
## isFull Method

newQueue.isFull() → False

beginningOfQueue = 0

topOfQueue = 2

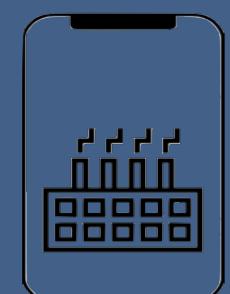
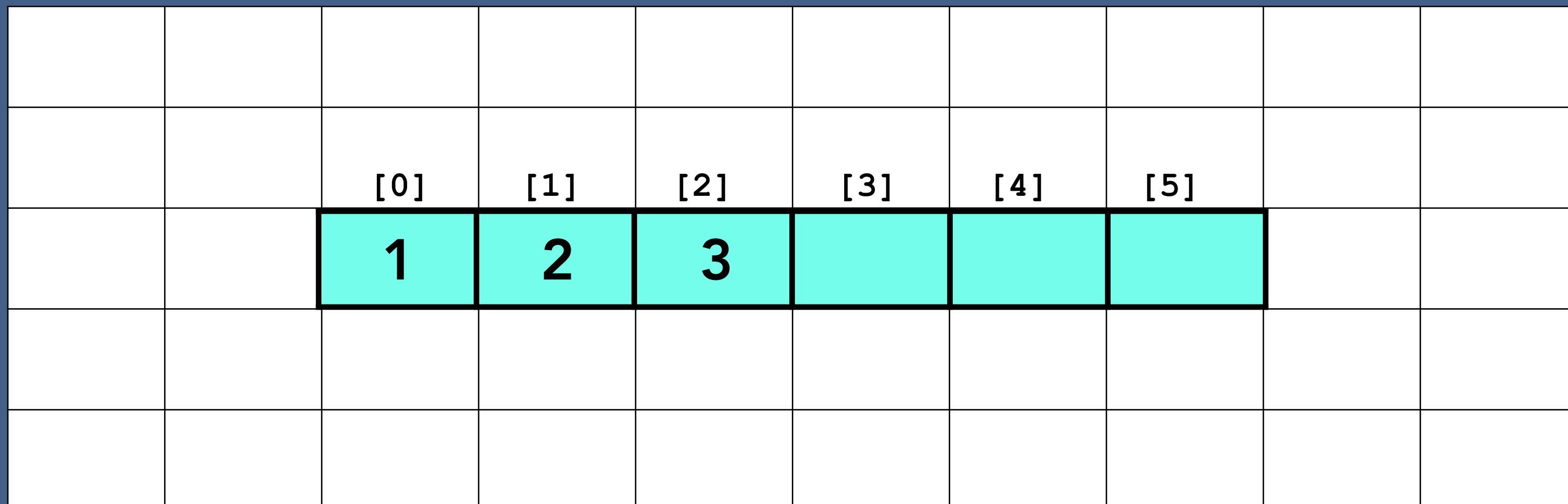
		[0]	[1]	[2]	[3]	[4]	[5]		
		1	2	3					



# Linear Queue using Array

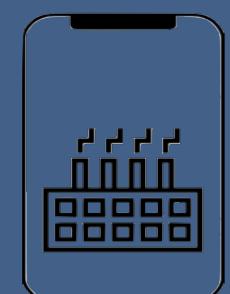
## delete Method

```
newQueue.delete()
```



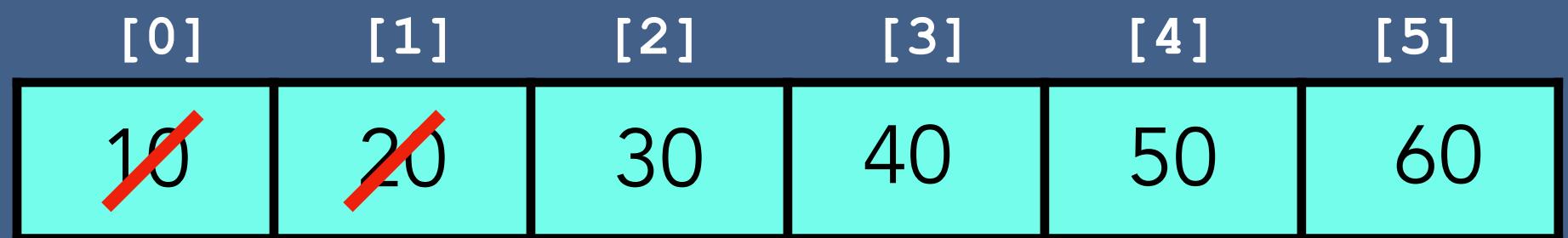
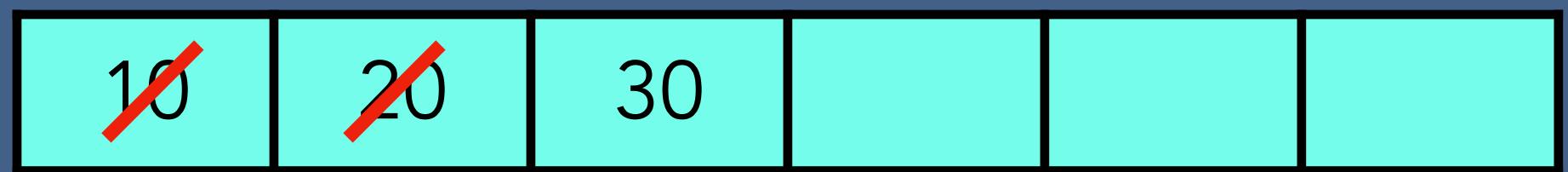
# Time and Space Complexity of Linear Queue using Array

Linear Queue - Array	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	$O(1)$	$O(1)$

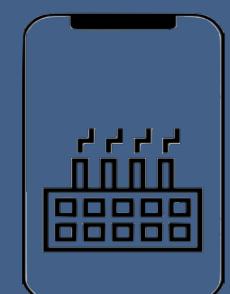


# Why do we need Circular Queue?

deQueue method causes blank cells.



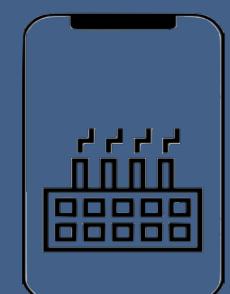
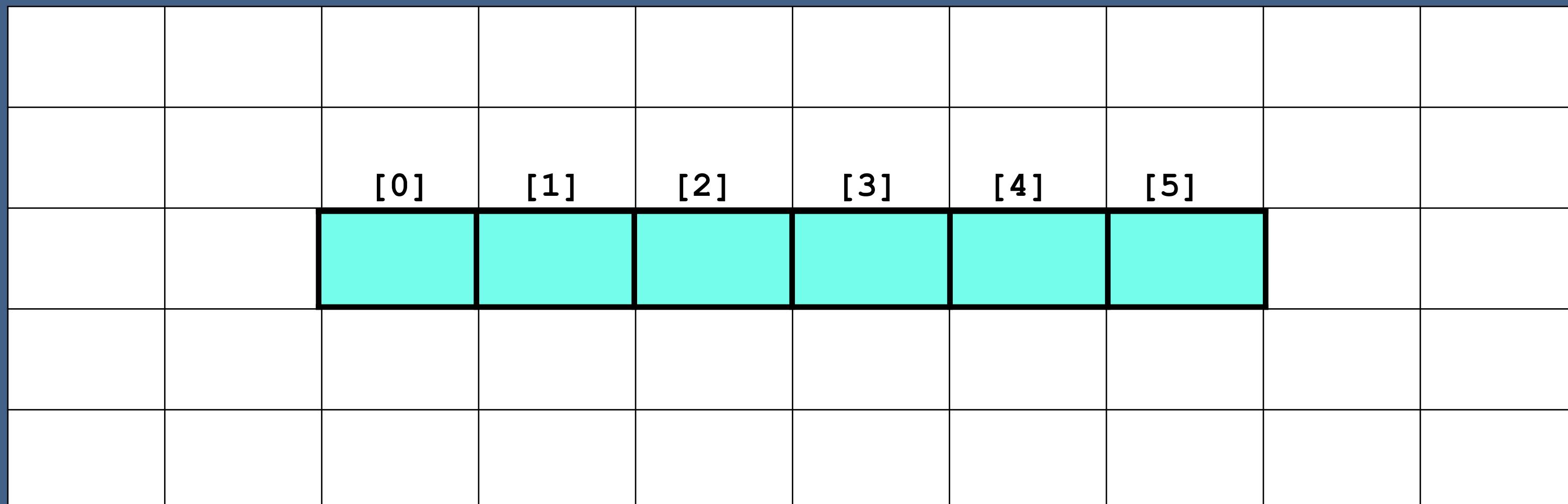
$$\text{topOfQueue} = 2 + 1 = 3 + 1 = 4 + 1 = 5$$



# Circular Queue using Array

## Create a Queue

```
newQueue = Queue(6)  
beginningOfQueue = -1  
topOfQueue = -1
```



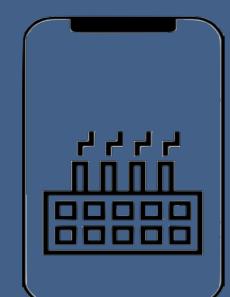
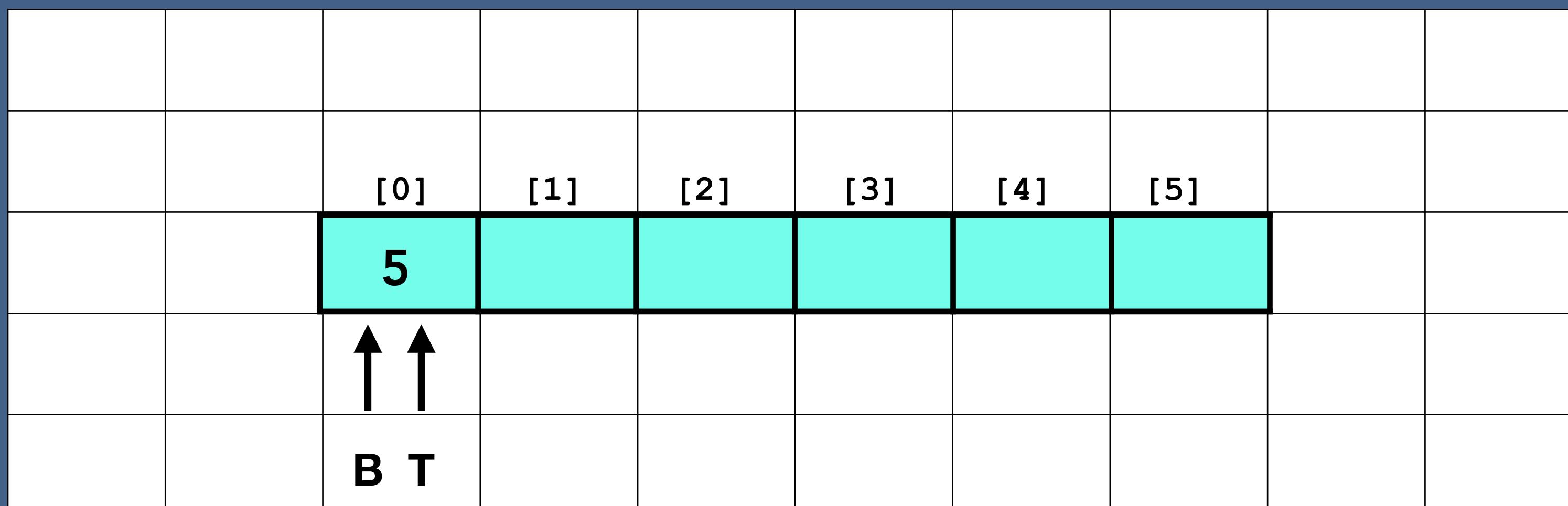
# Circular Queue using Array

## enQueue Method

```
newQueue.enQueue(5)
```

```
beginningOfQueue = 0
```

```
topOfQueue = 0
```

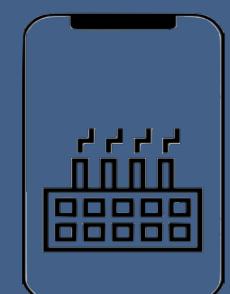
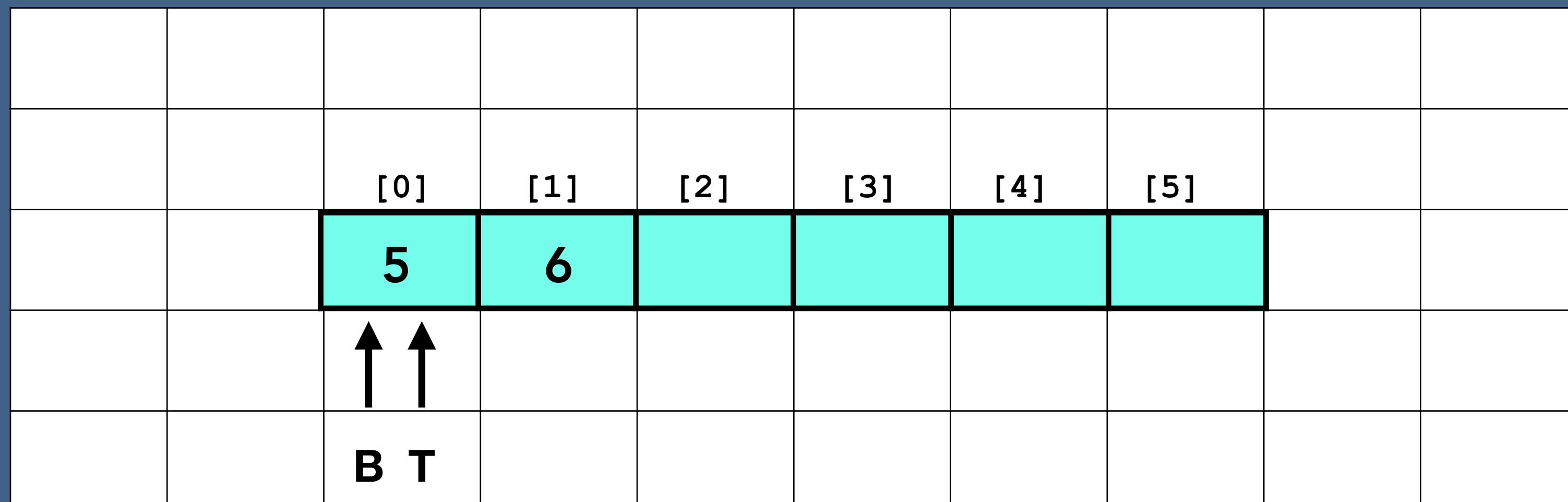


# Circular Queue using Array

## enQueue Method

```
newQueue.enQueue(6)
```

```
beginningOfQueue = 0  
topOfQueue = 0
```

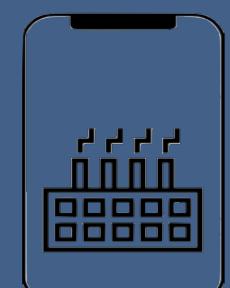
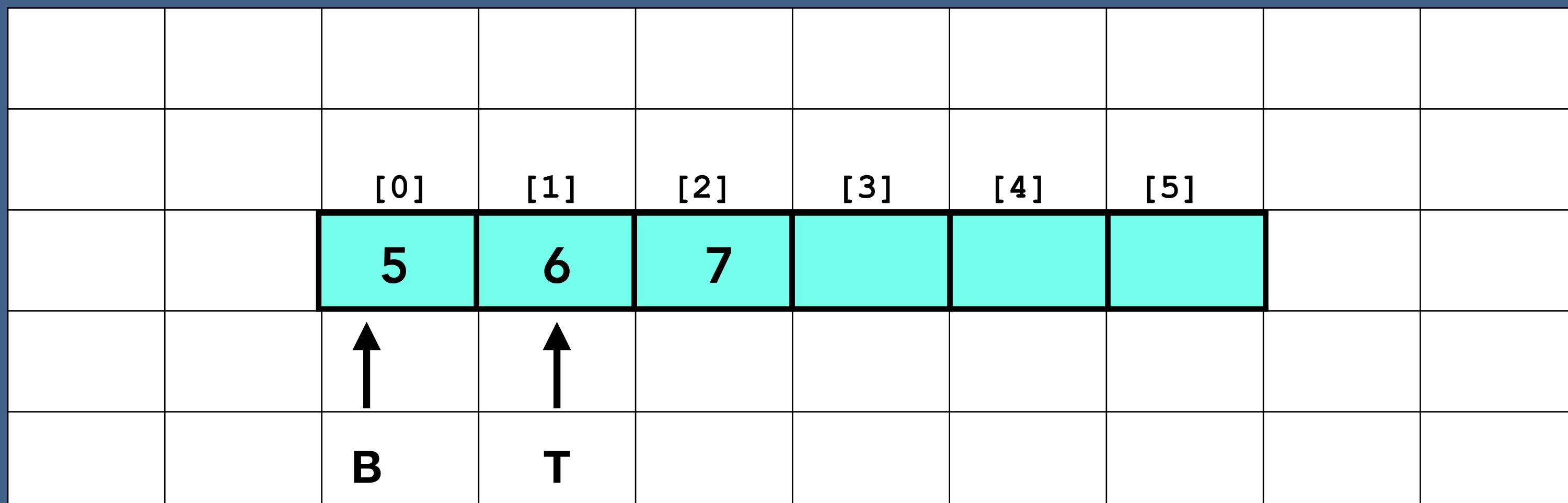


# Circular Queue using Array

## enQueue Method

```
newQueue.enQueue(7)
```

```
beginningOfQueue = 0  
topOfQueue = 2
```



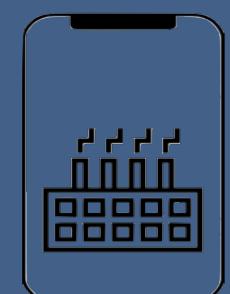
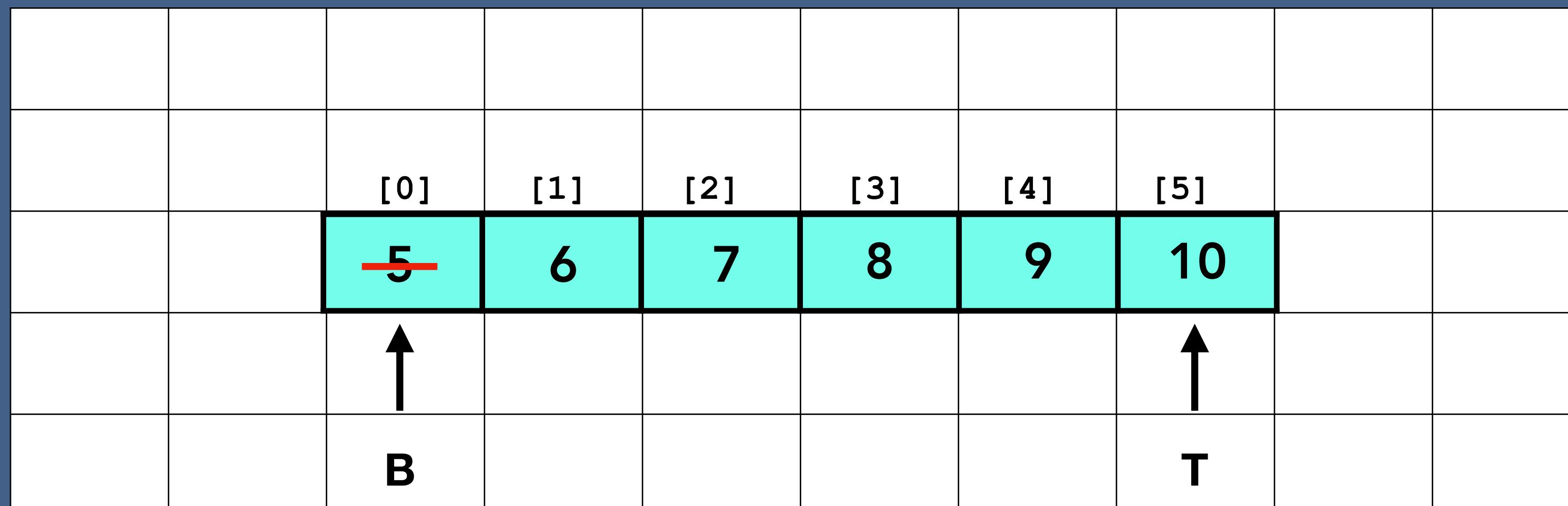
# Circular Queue using Array

## deQueue Method

newQueue.deQueue () → 5

beginningOfQueue = 0

topOfQueue = 5



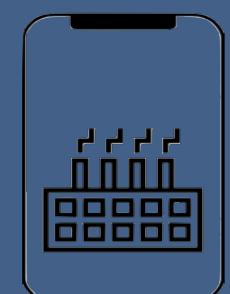
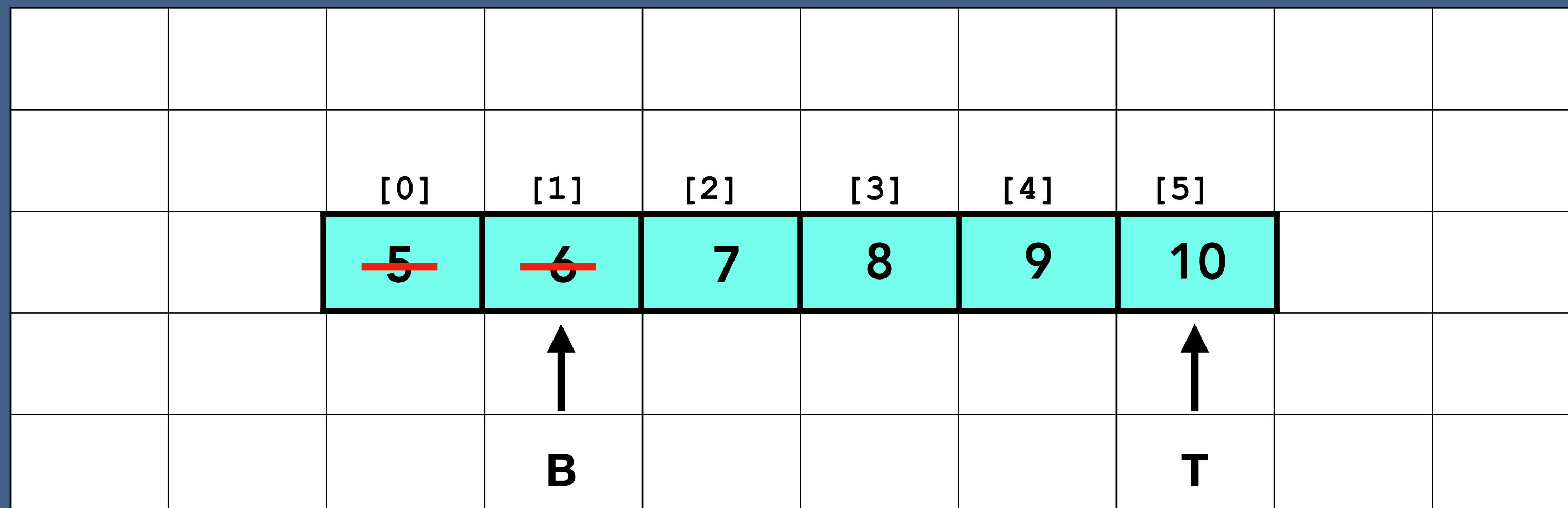
# Circular Queue using Array

## deQueue Method

newQueue.deQueue () → 6

beginningOfQueue = 2

topOfQueue = 5



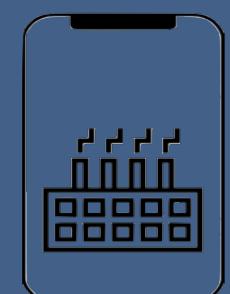
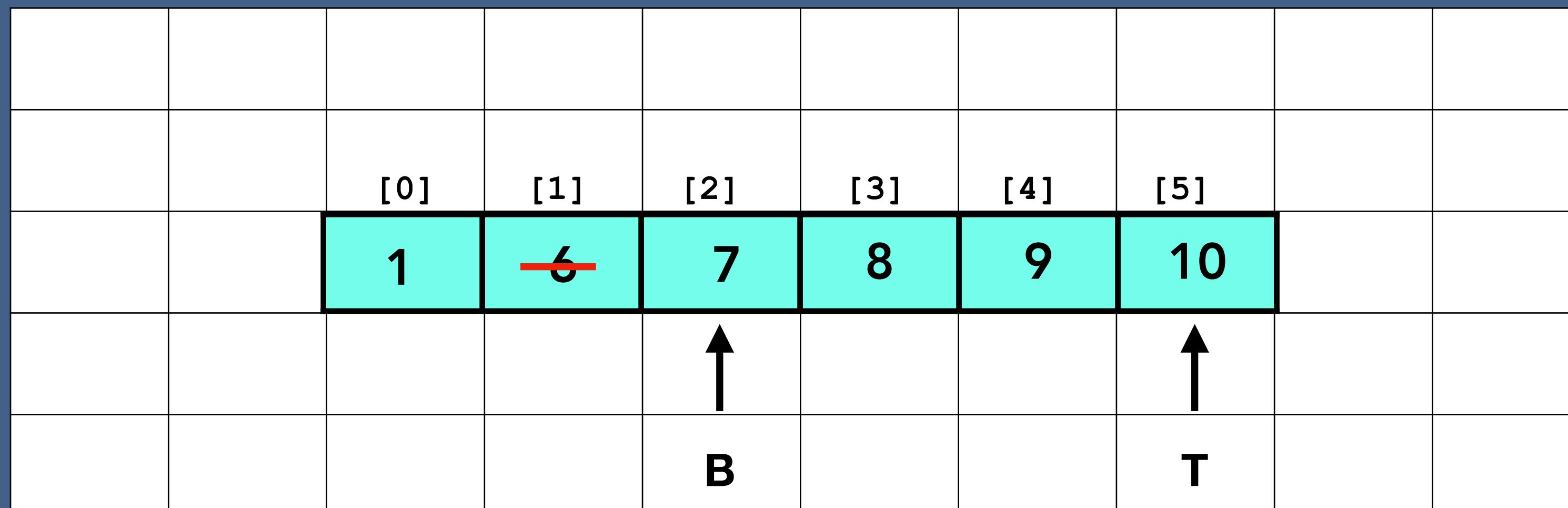
# Circular Queue using Array

## deQueue Method

```
newQueue.enqueue(1)
```

```
beginningOfQueue = 2
```

```
topOfQueue = 6
```



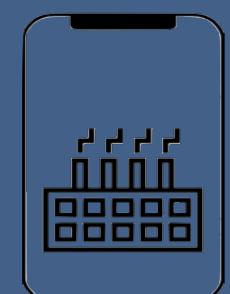
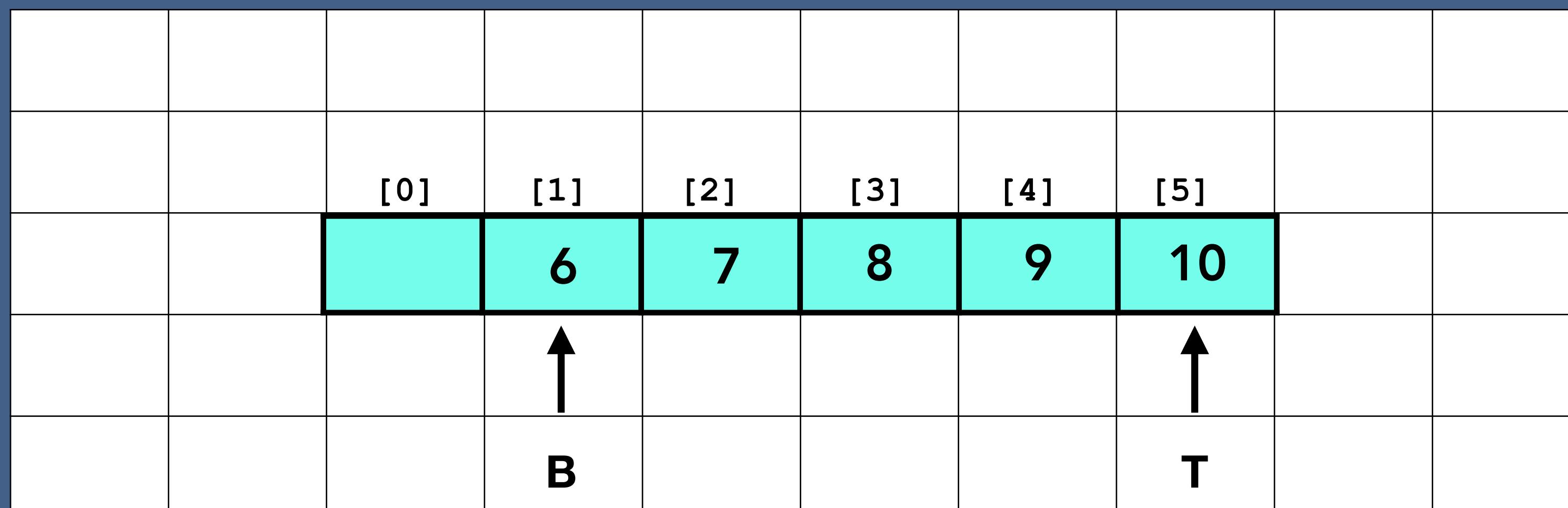
# Circular Queue using Array

## Peek Method

newQueue.peek () → 6

beginningOfQueue = 1

topOfQueue = 5



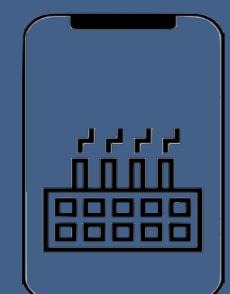
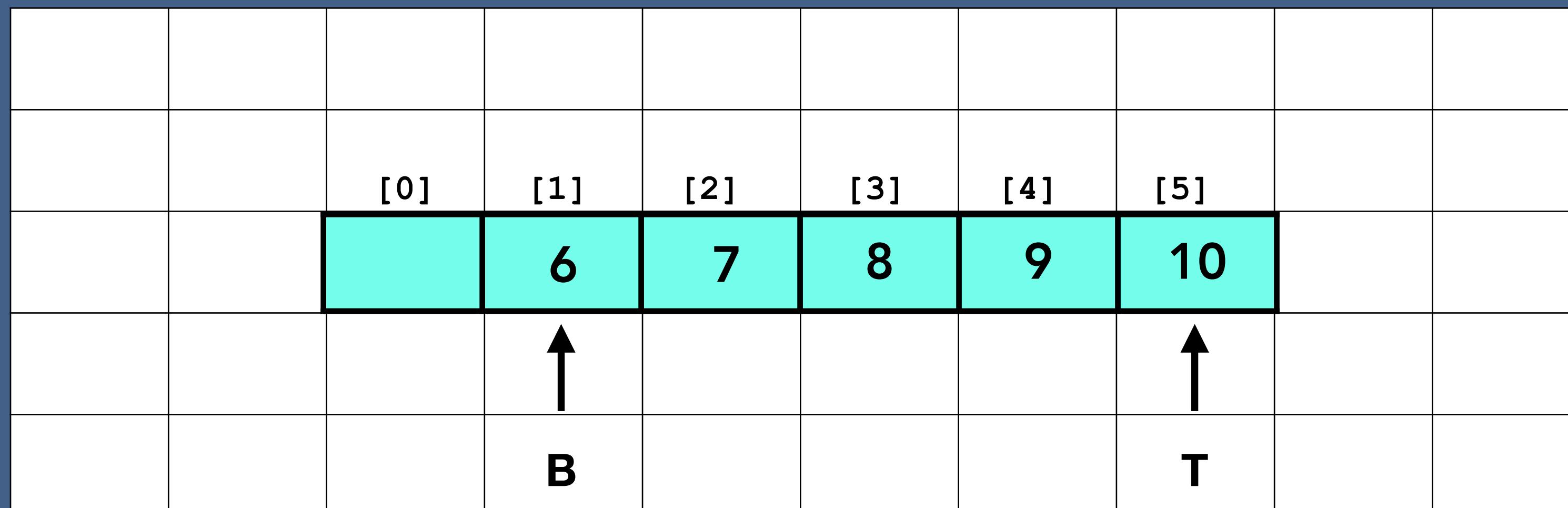
# Circular Queue using Array

## isFull Method

newQueue.isFull() → False

beginningOfQueue = 1

topOfQueue = 5

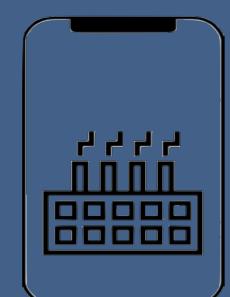
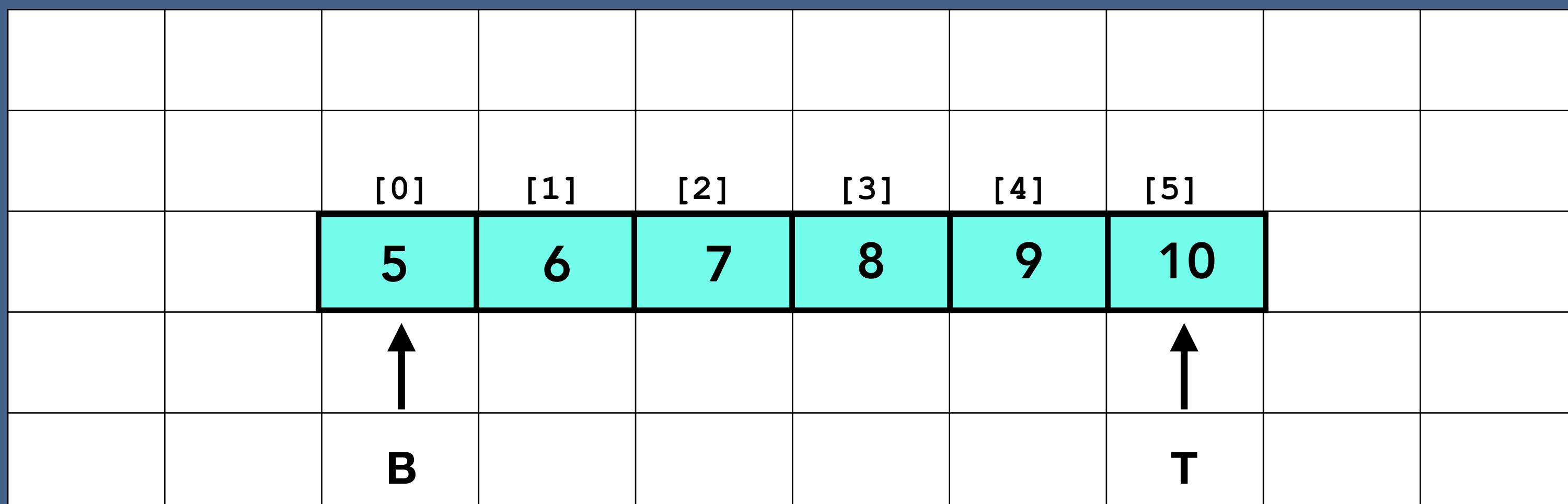


# Circular Queue using Array

## isFull Method

newQueue.isFull() → True

beginningOfQueue = 0  
topOfQueue = 5

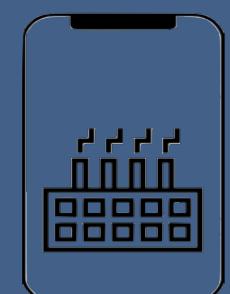
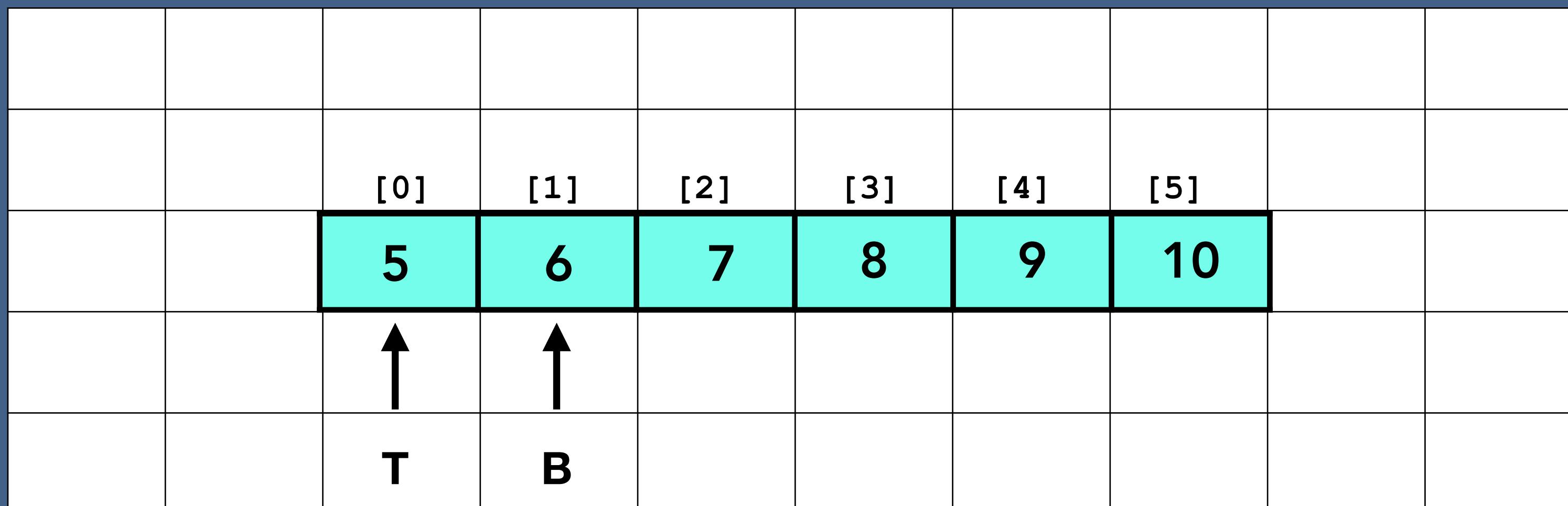


# Circular Queue using Array

## isFull Method

newQueue.isFull() → True

beginningOfQueue = 1  
topOfQueue = 0



# Circular Queue using Array

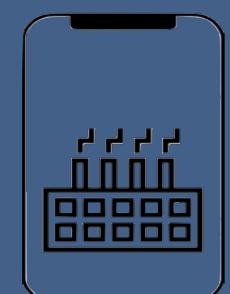
## isEmpty Method

newQueue.isEmpty() → False

beginningOfQueue = 1

topOfQueue = 0

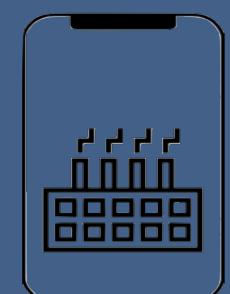
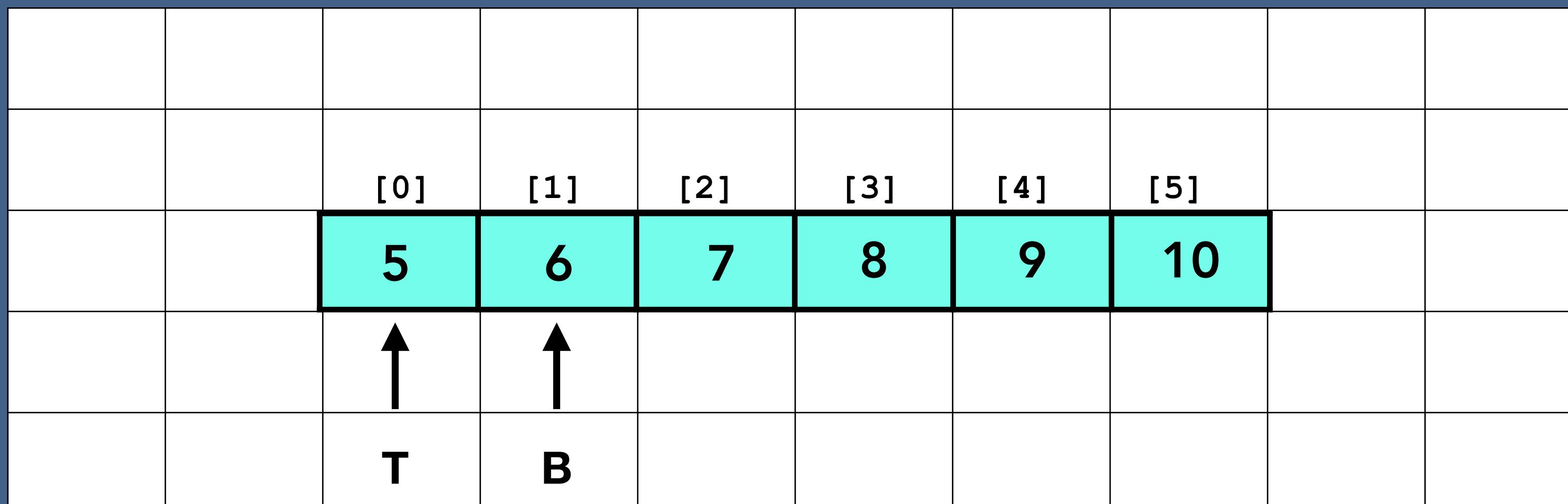
		[0]	[1]	[2]	[3]	[4]	[5]				
		5	6	7	8	9	10				
		↑	↑								
		T	B								



# Circular Queue using Array

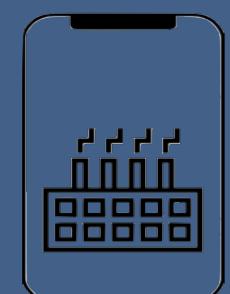
## Delete Method

```
newQueue.delete()  
beginningOfQueue = 1  
topOfQueue = 0
```



# Time and Space Complexity of Circular Queue using Array

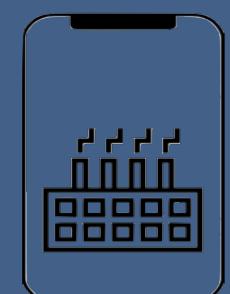
Circular Queue - Array	Time complexity	Space complexity
Create Queue	$O(1)$	$O(N)$
enQueue	$O(1)$	$O(1)$
deQueue	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
isFull	$O(1)$	$O(1)$
Delete Entire Queue	$O(1)$	$O(1)$



# Queue using Linked List

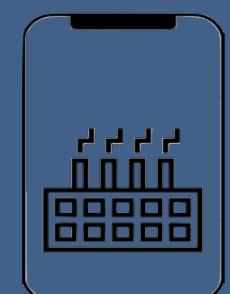
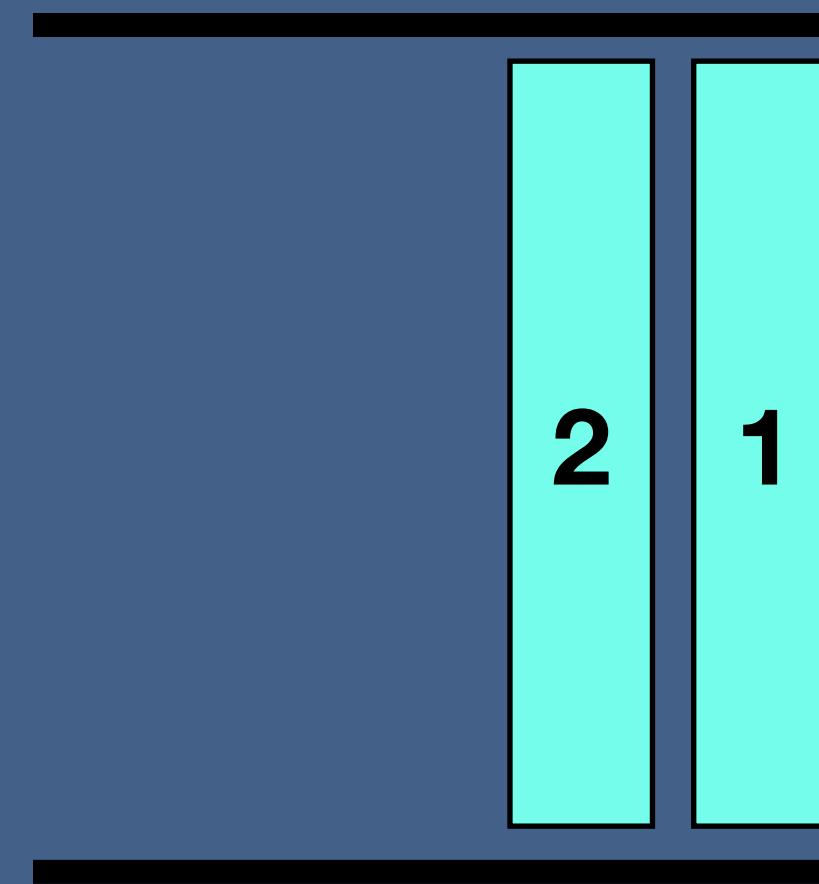
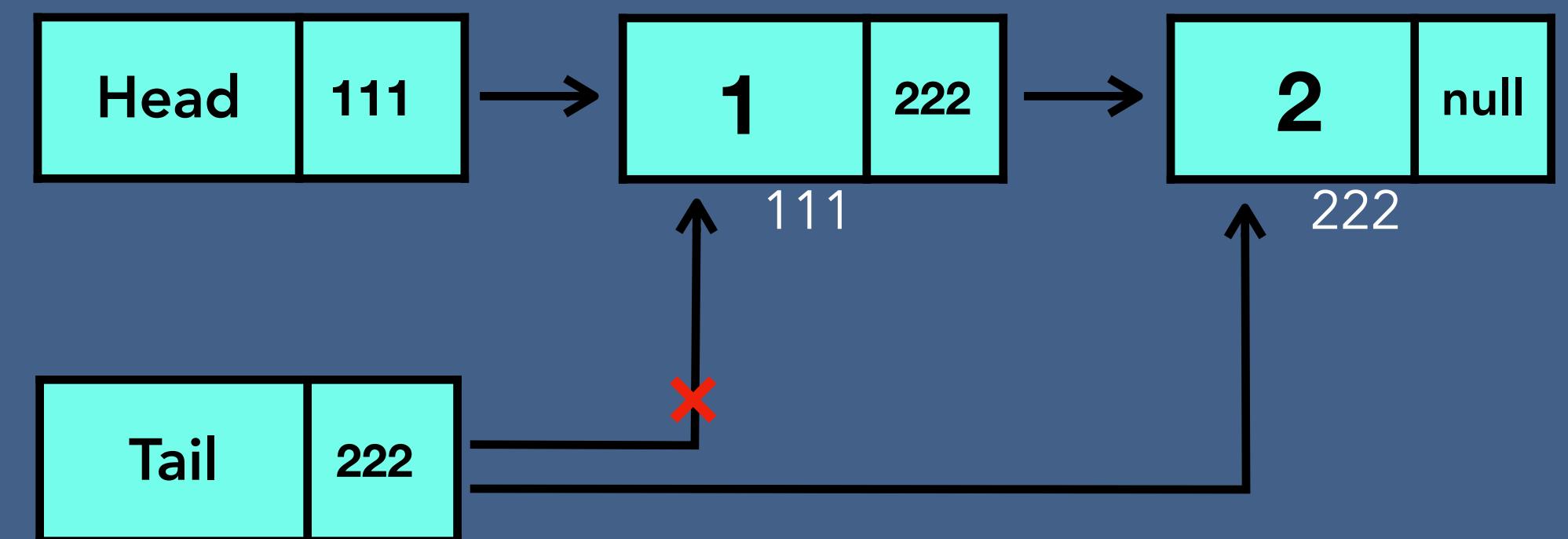
## Create a Queue

Create an object of Linked List class



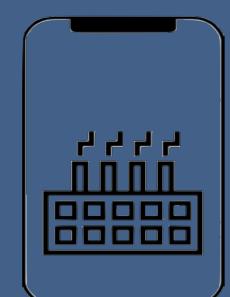
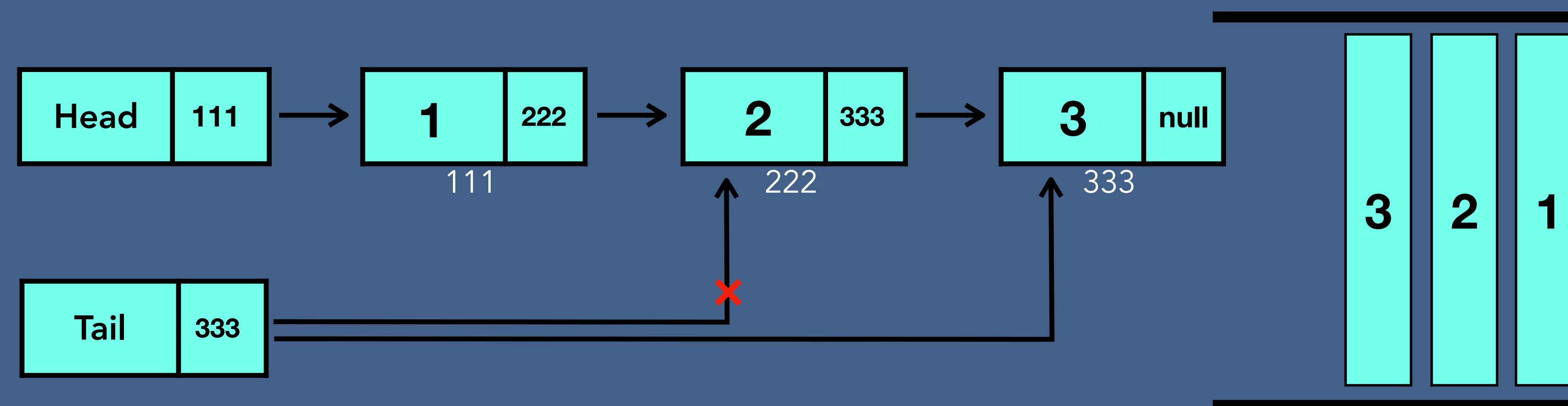
# Queue using Linked List

## enQueue() Method



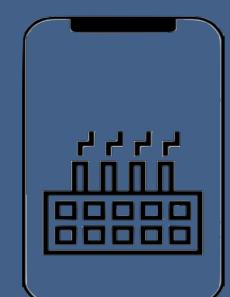
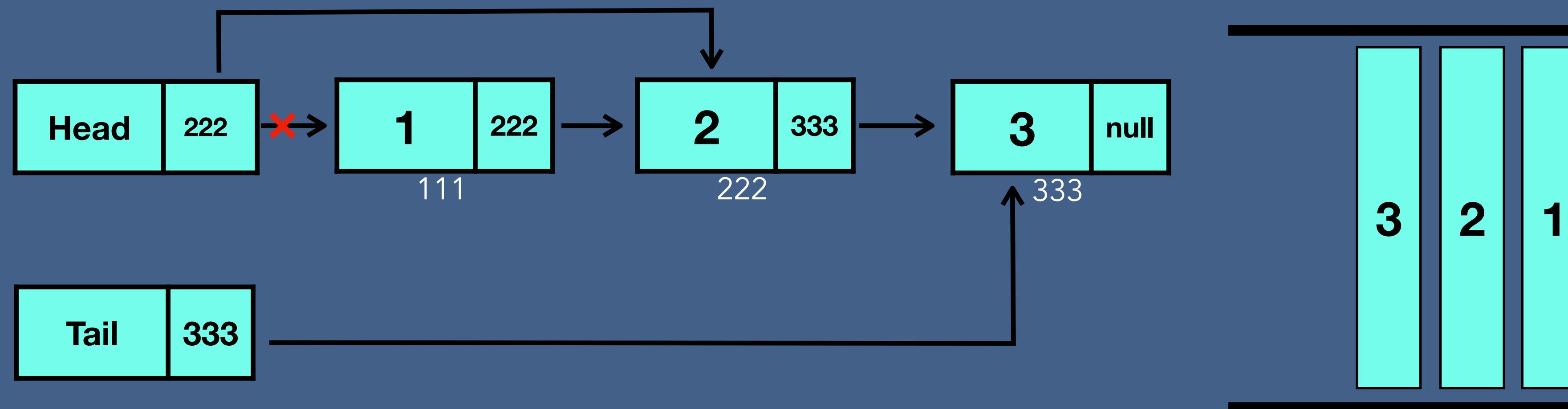
# Queue using Linked List

## enQueue() Method



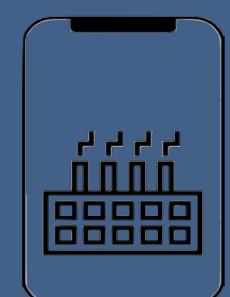
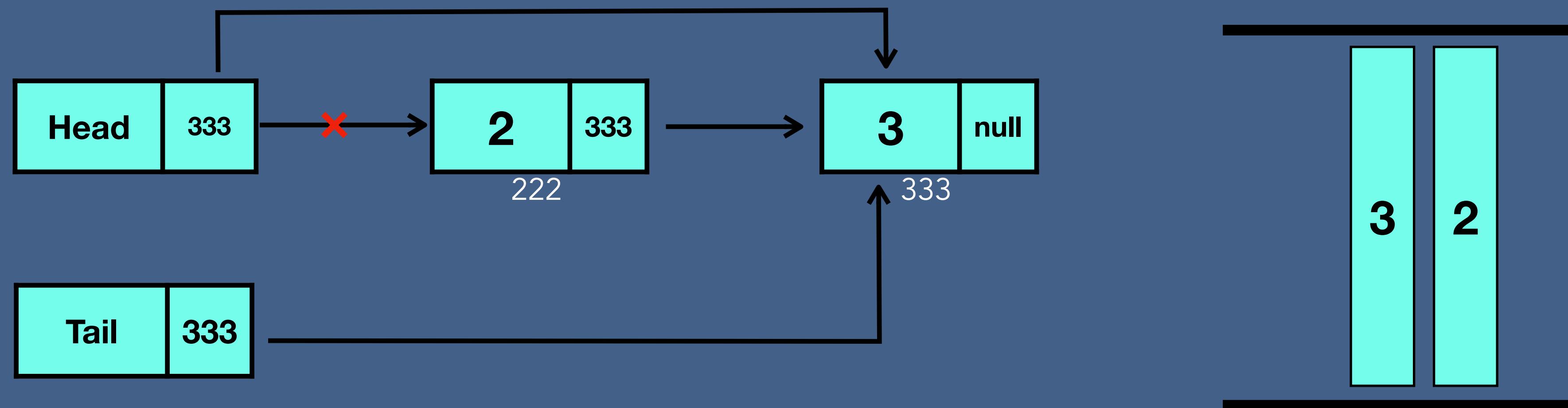
# Queue using Linked List

## deQueue() Method



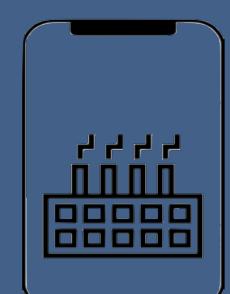
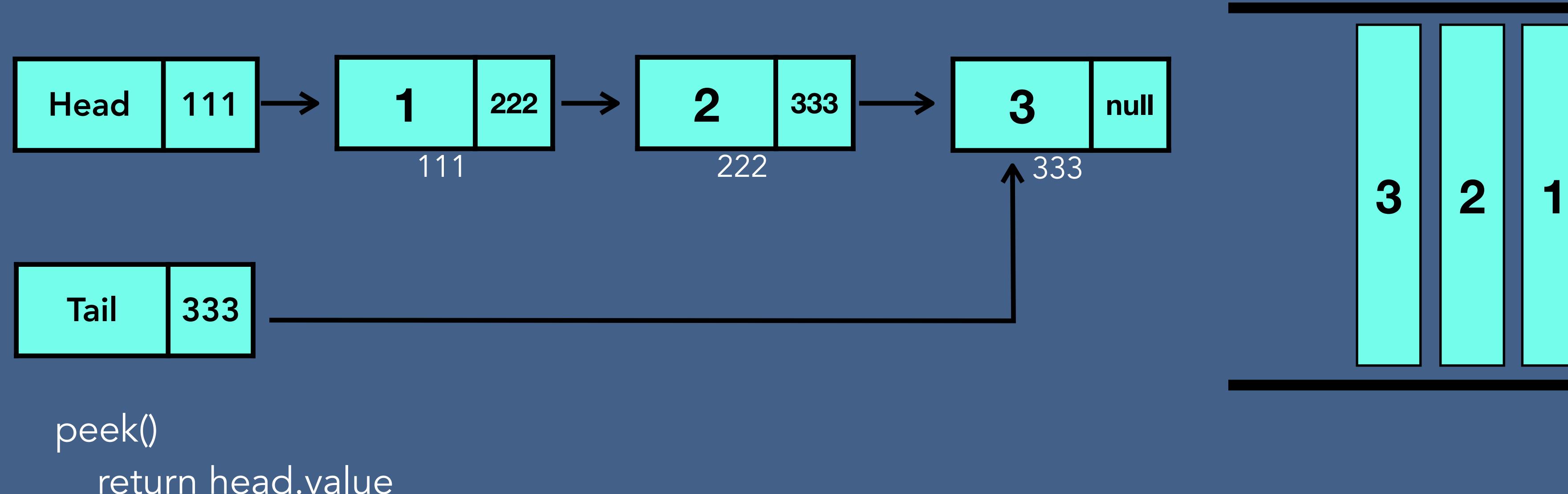
# Queue using Linked List

## deQueue() Method



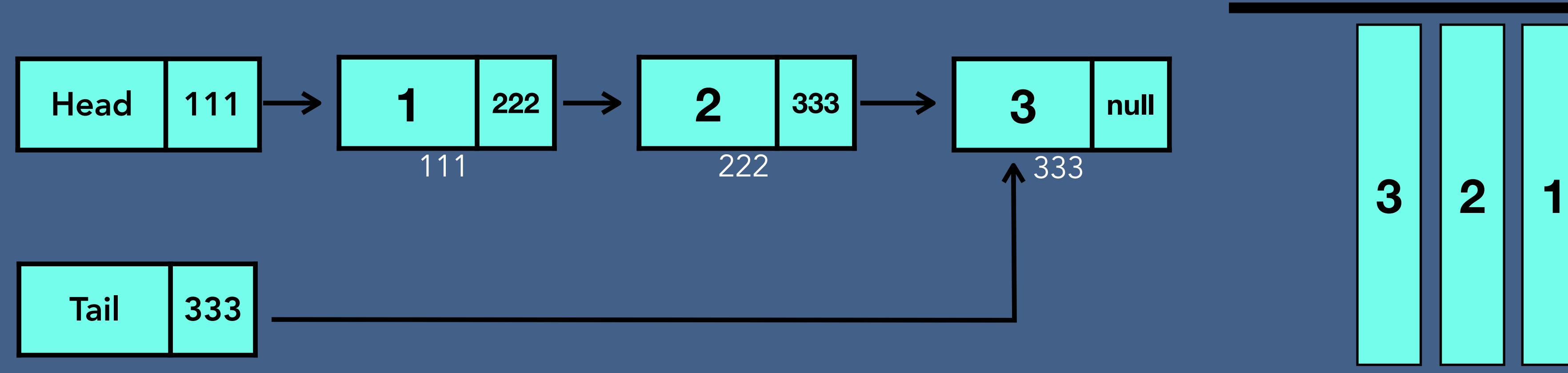
# Queue using Linked List

## peek() Method



# Queue using Linked List

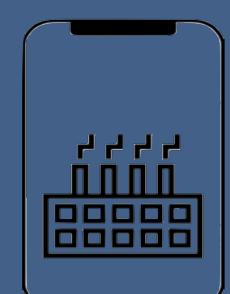
## isEmpty() Method



isEmpty()

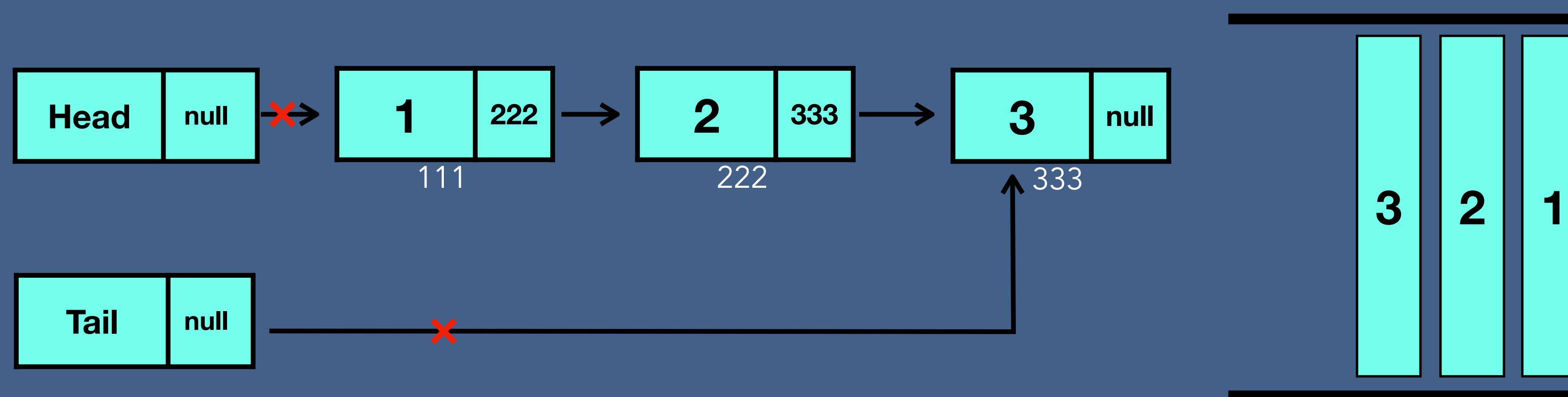
If head is Null:

True

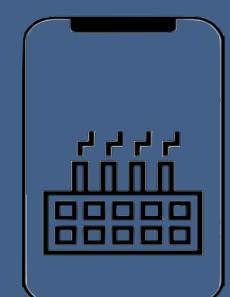


# Queue using Linked List

## delete() Method

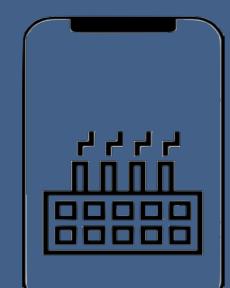


delete()  
head = Null  
tail = Null



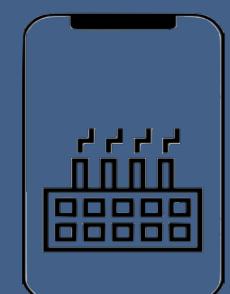
# Time and Space Complexity of Queue using Linked List

Queue - 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	$O(1)$	$O(1)$



# Queue - Array vs Linked List

	Array		Linked List	
	Time complexity	Space complexity	Time complexity	Space complexity
Create Queue	O(1)	O(n)	O(1)	O(1)
Enqueue	O(1)	O(1)	O(1)	O(1)
Dequeue	O(1)	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)	O(1)
isFull	O(1)	O(1)	-	-
Delete Entire Queue	O(1)	O(1)	O(1)	O(1)
<b>Space efficient?</b>		NO		Yes



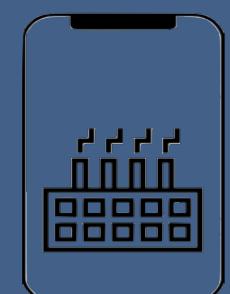
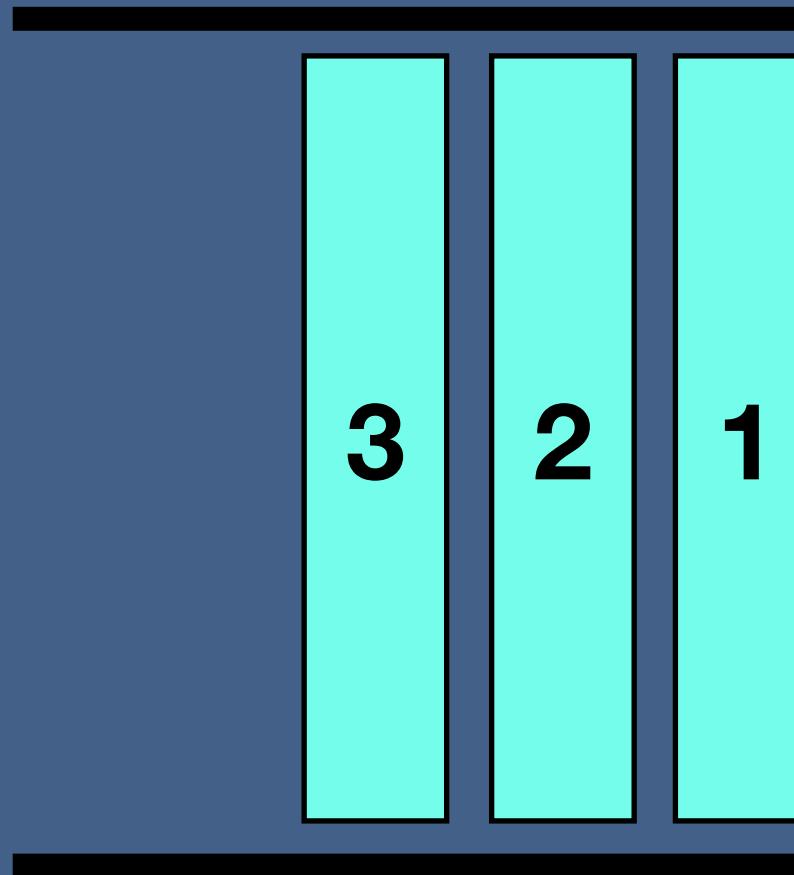
# When to Use/Avoid Queue?

## Use:

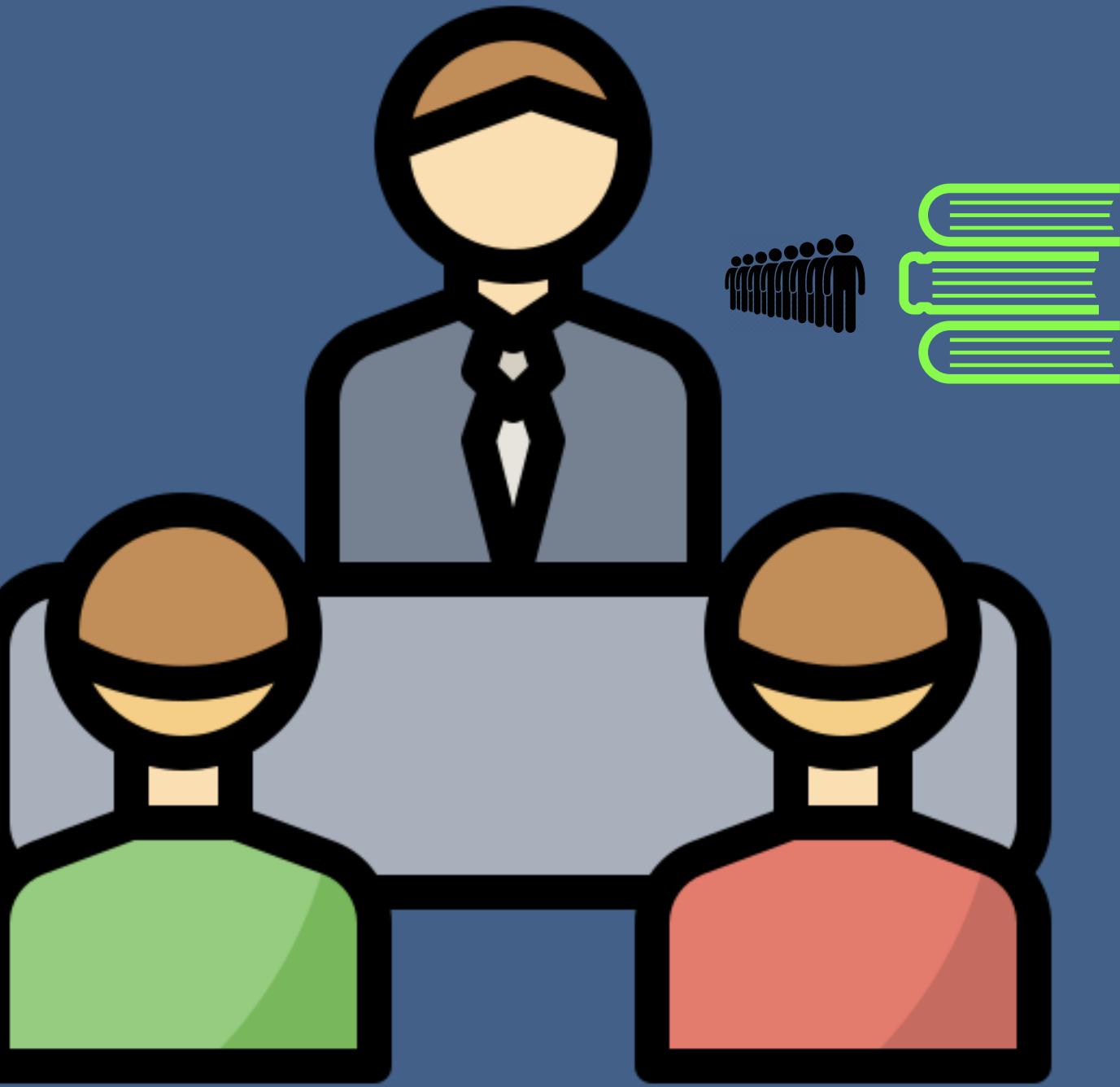
- FIFO functionality
- The chance of data corruption is minimum

## Avoid:

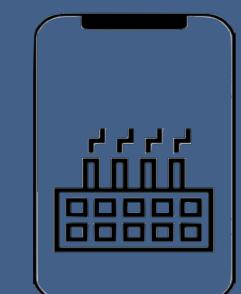
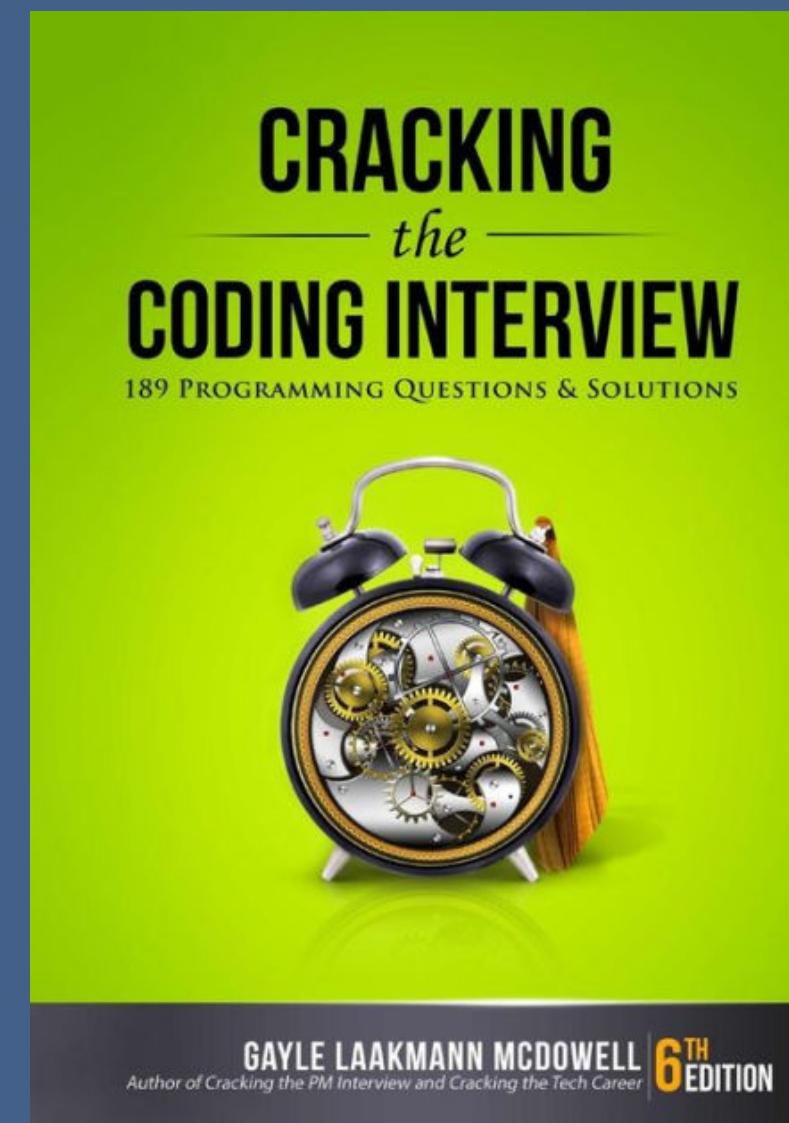
- Random access is not possible



# Stack and Queue Interview Questions

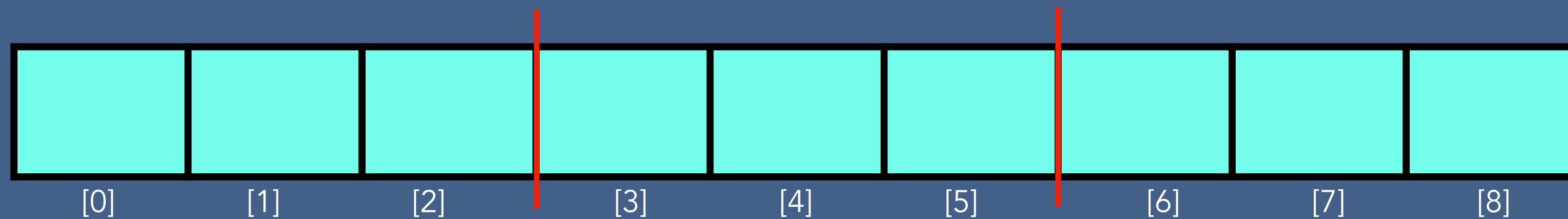


# Stack and Queue Interview Questions

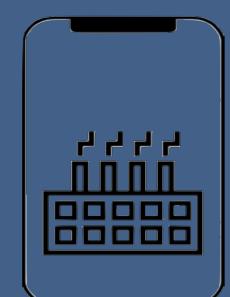


# Three in One

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



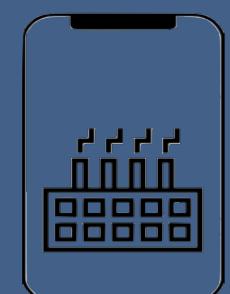
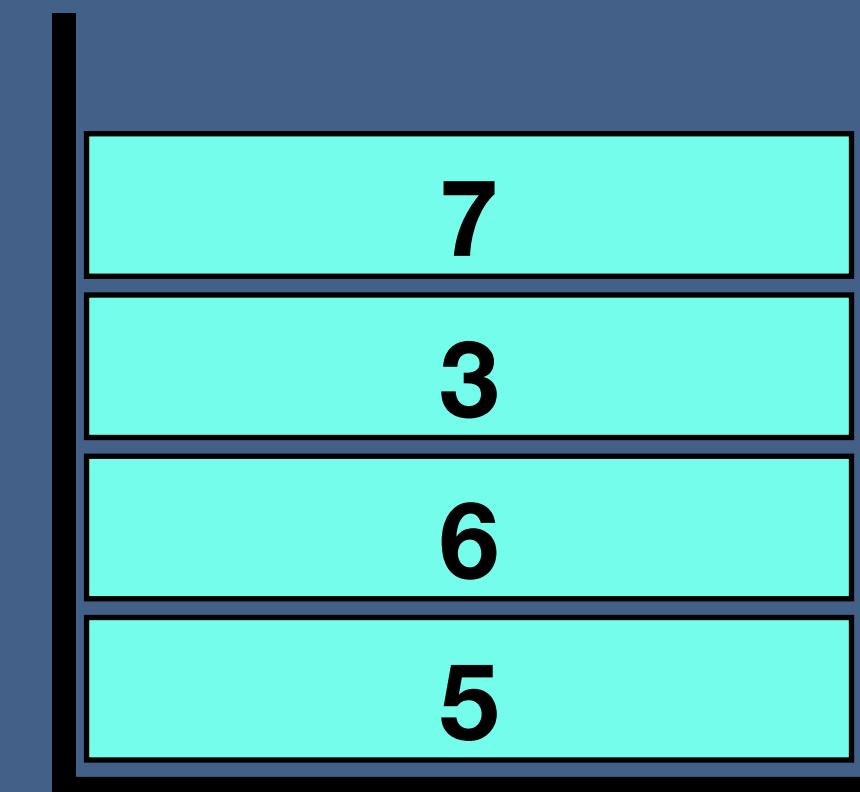
For Stack 1 — [0], [1], [2] → [0,  $n/3$ )  
For Stack 2 — [3], [4], [5] → [ $n/3$ ,  $2n/3$ )  
For Stack 3 — [6], [7], [8] → [ $2n/3$ ,  $n$ )



# 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

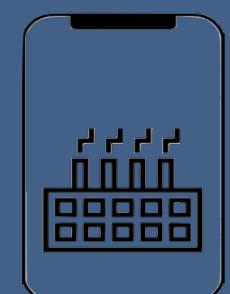
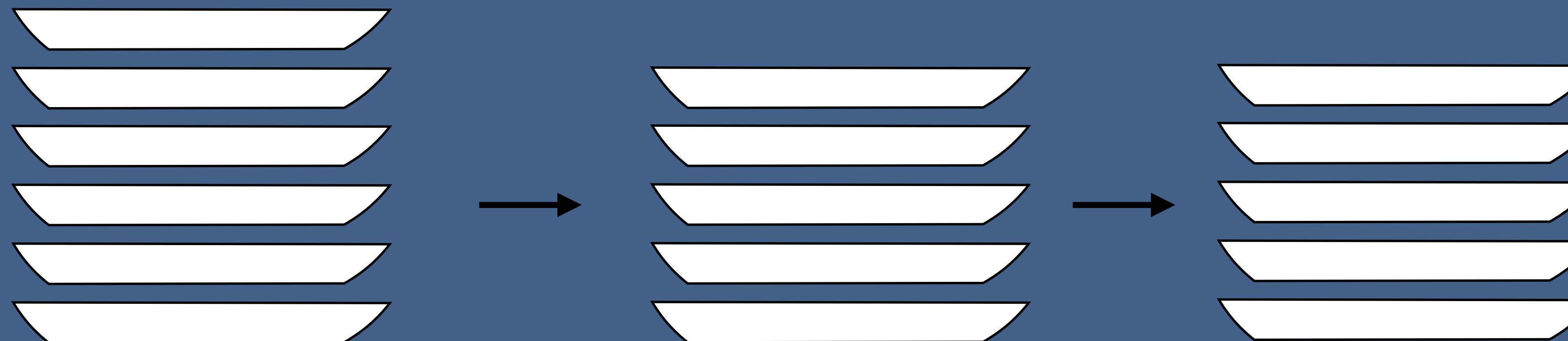


# Stack of Plates

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.



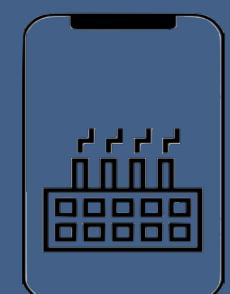
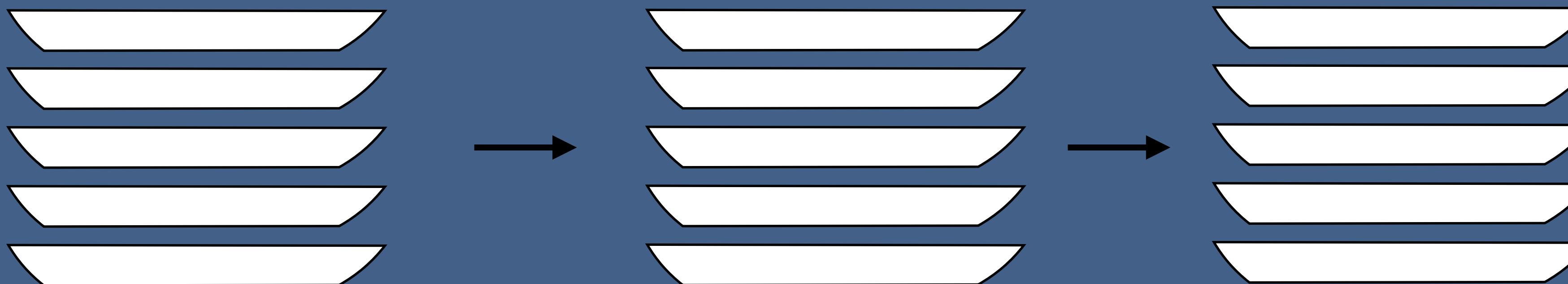
# Stack of Plates

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.

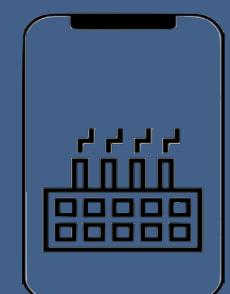
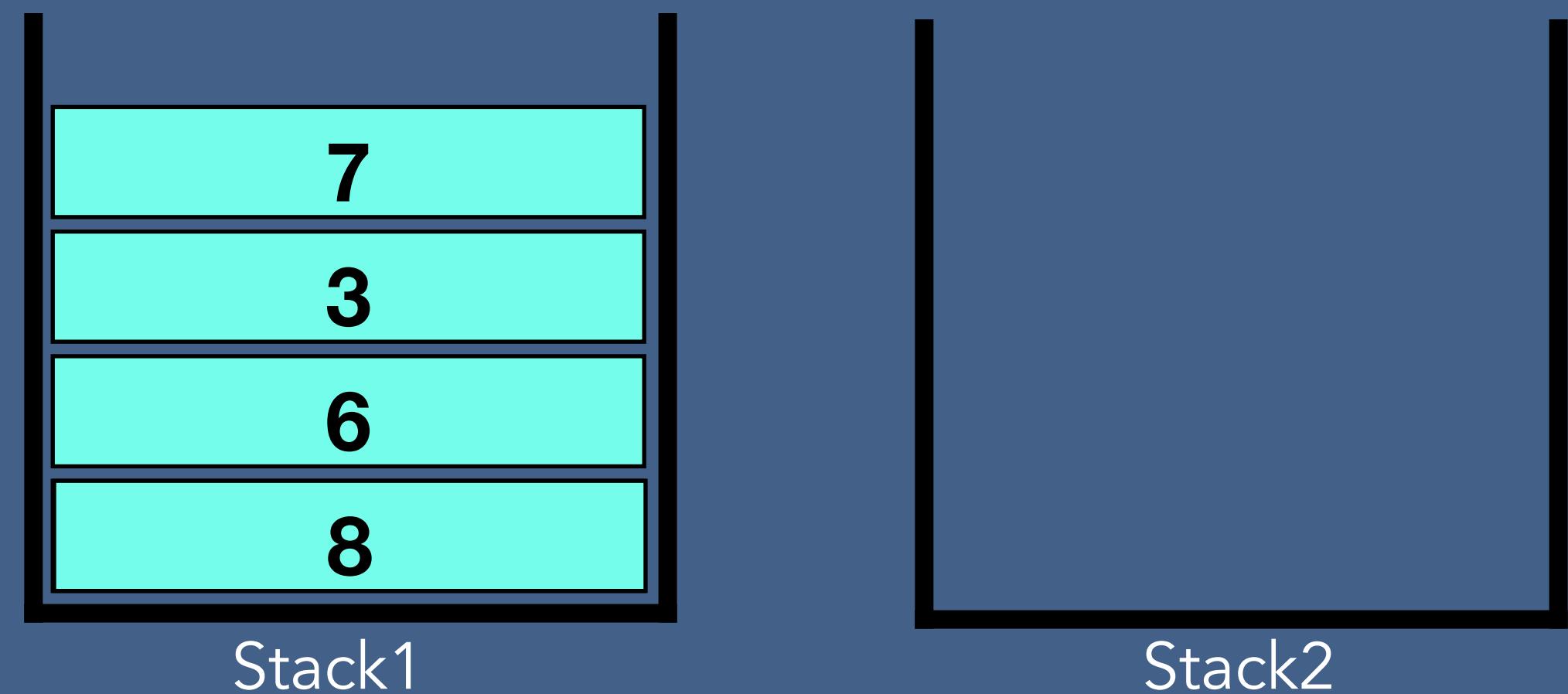
pop( )



# Queue via Stacks

Implement Queue class which implements a queue using two stacks.

`Dequeue()`



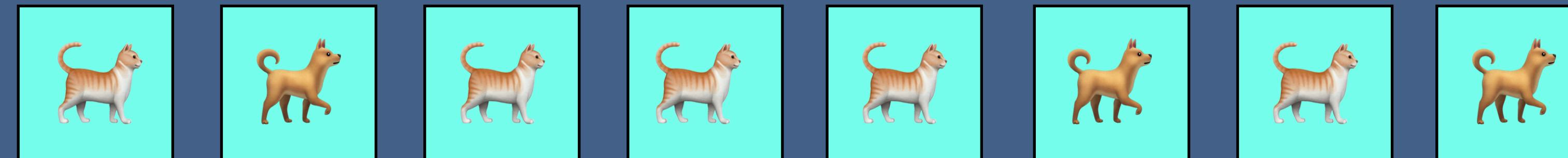
# Animal Shelter

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

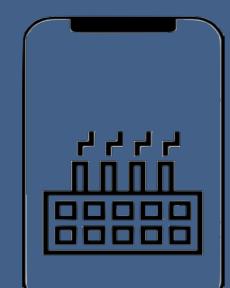
Newest

Oldest



Enqueue(Dog)

Enqueue(Cat)



# 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

Newest

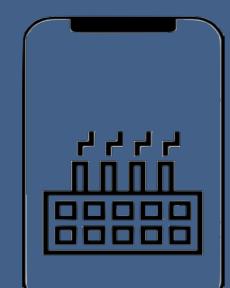
Oldest



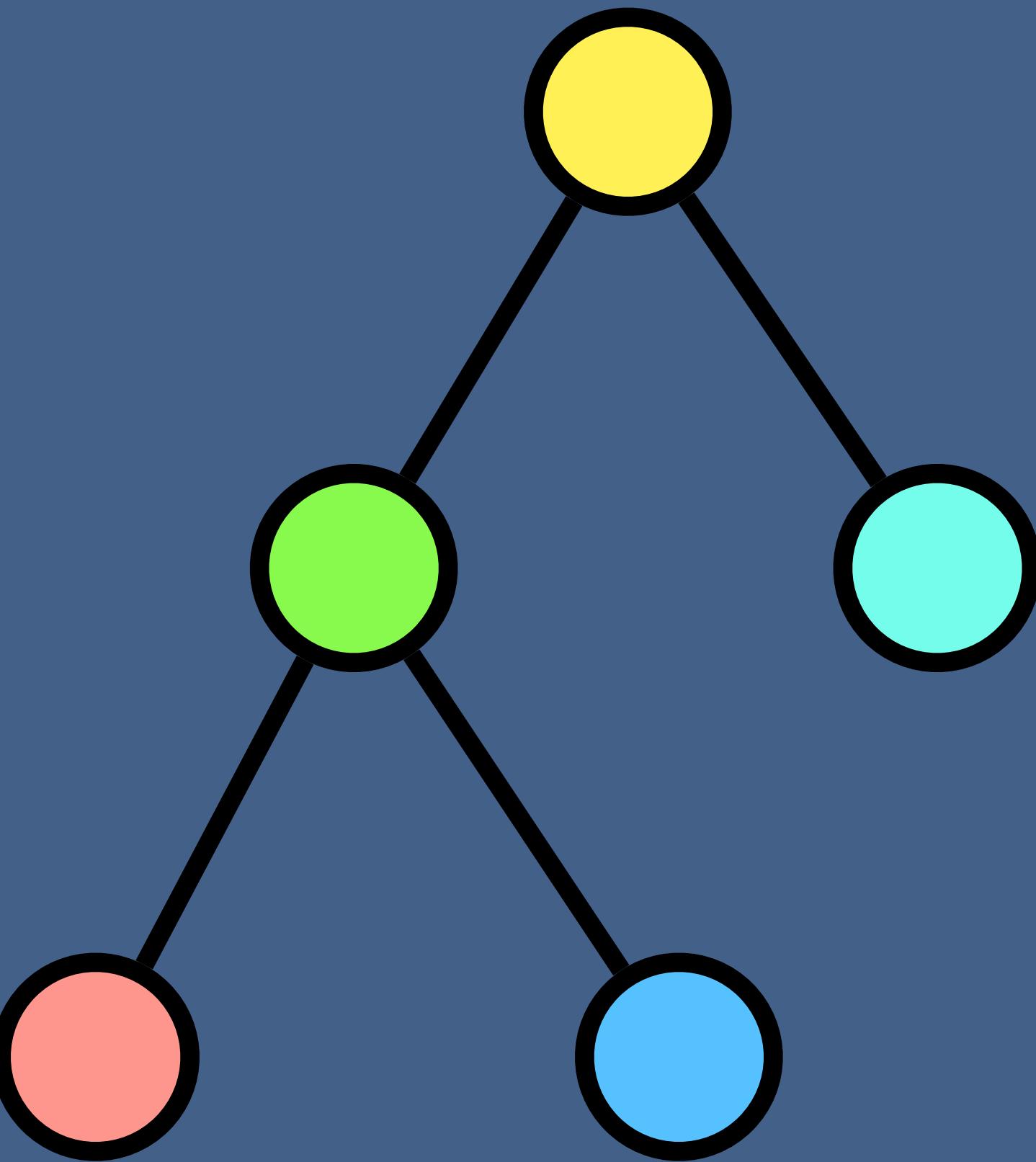
DequeueAny()

DequeueDog()

DequeueCat()

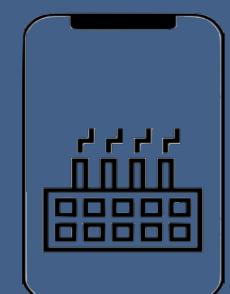
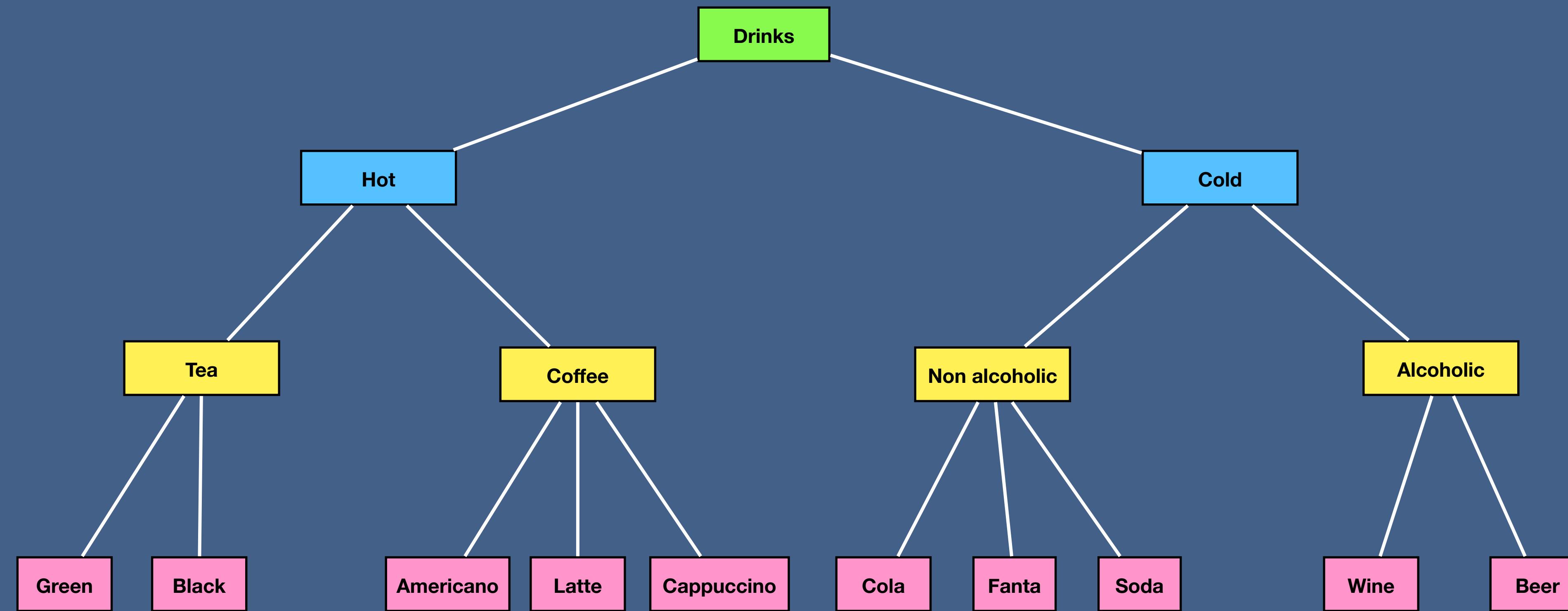


# Tree / Binary Tree



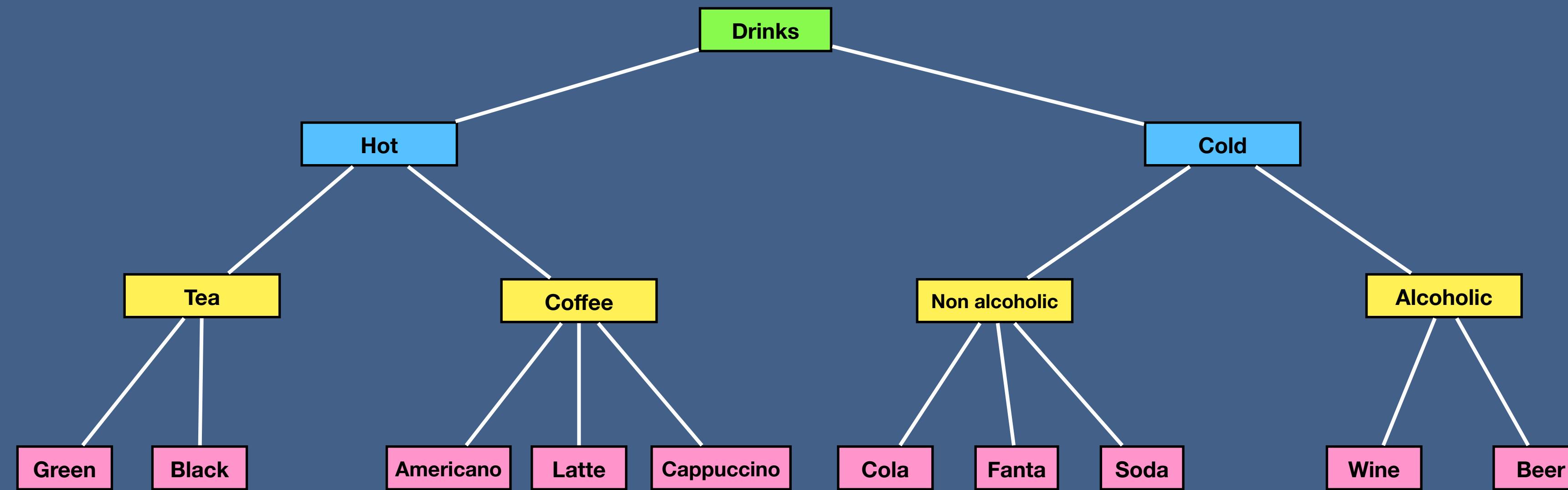
# What is a Tree?

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.



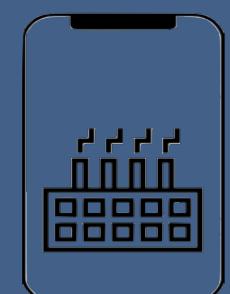
# What is a Tree?

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.



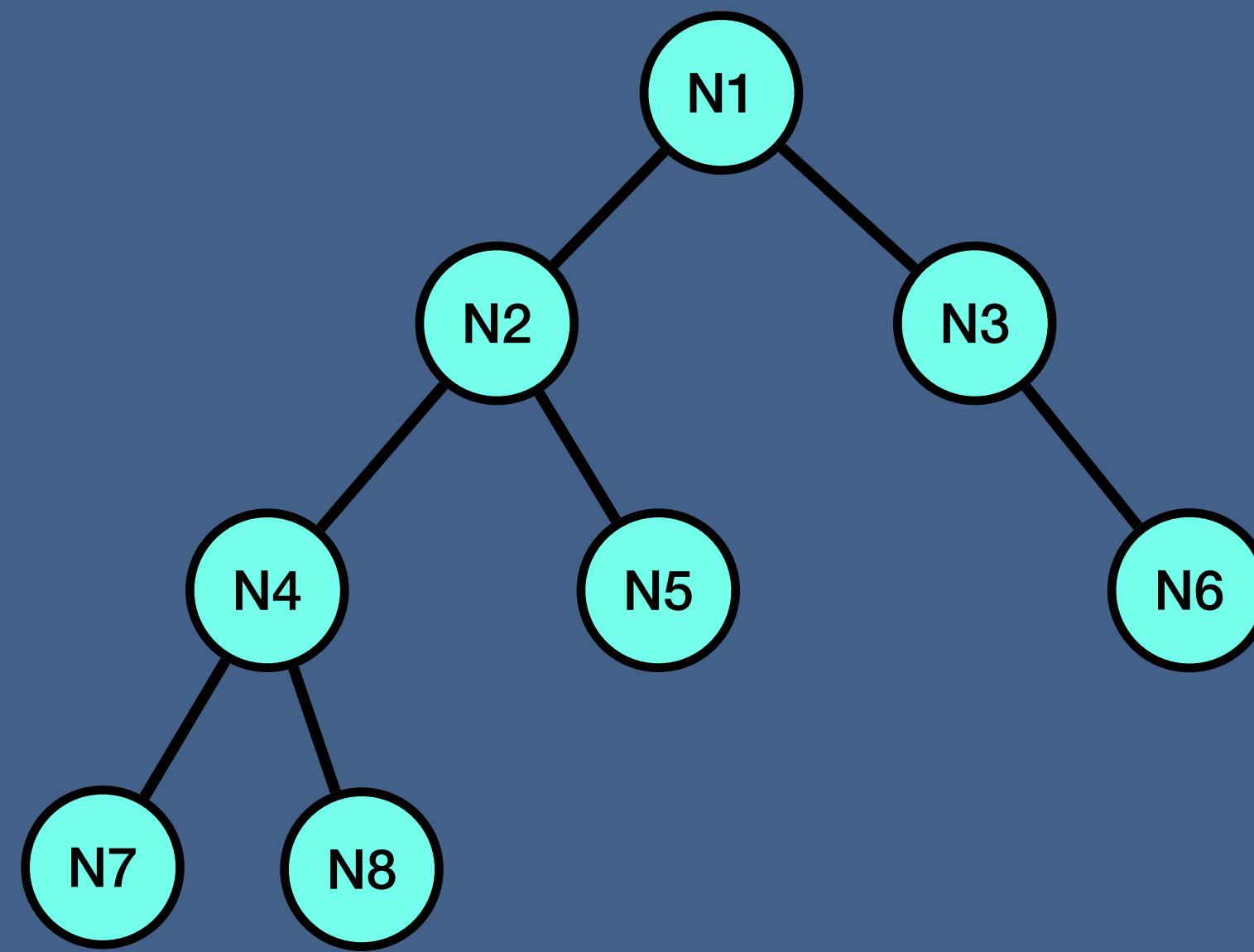
## Properties:

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



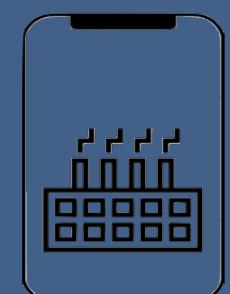
# What is a Tree?

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:

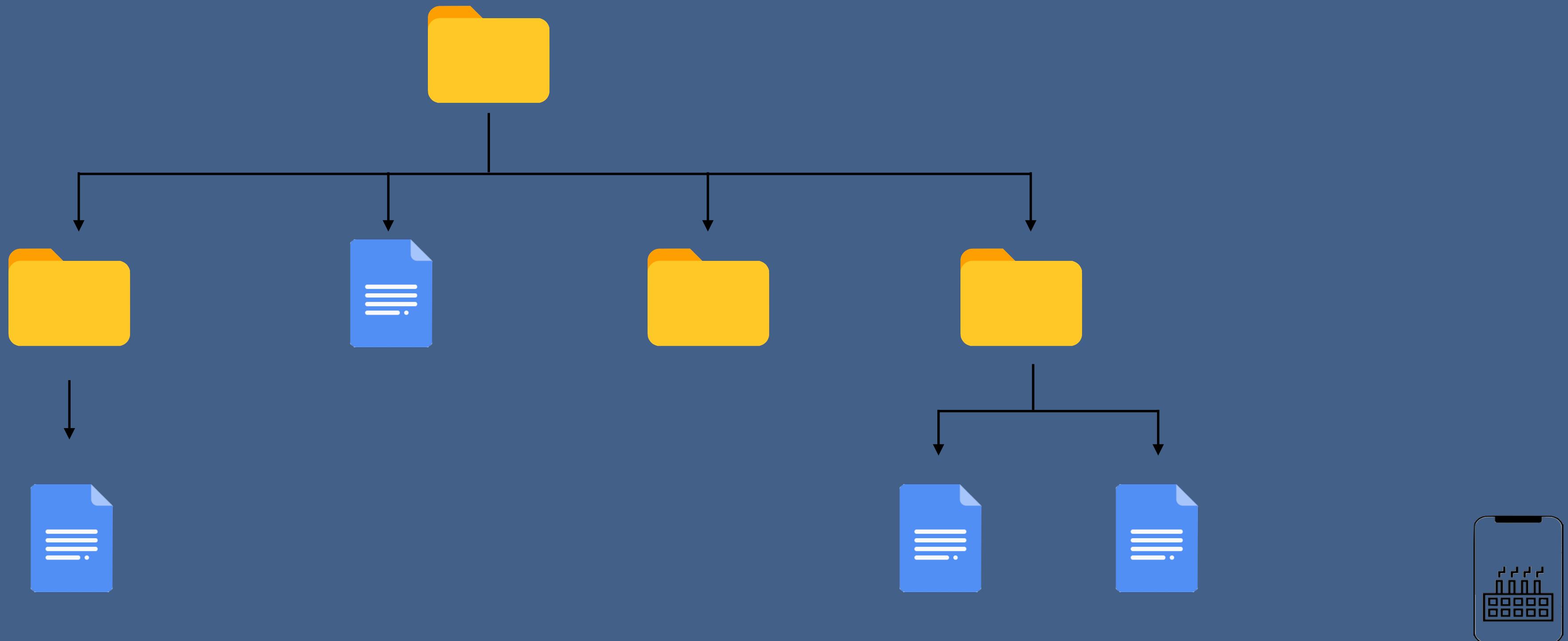
- Represent hierarchical data
- Each node has two components : data and a link to its sub category
- Base category and sub categories under it



# Why Tree?

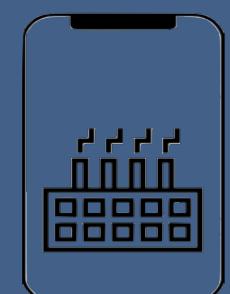
- Quicker and Easier access to the data
- Store hierarchical data, like folder structure, organization structure, XML/HTML data.

The file system on a computer



# Why 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

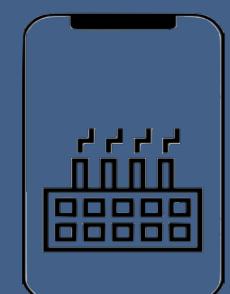
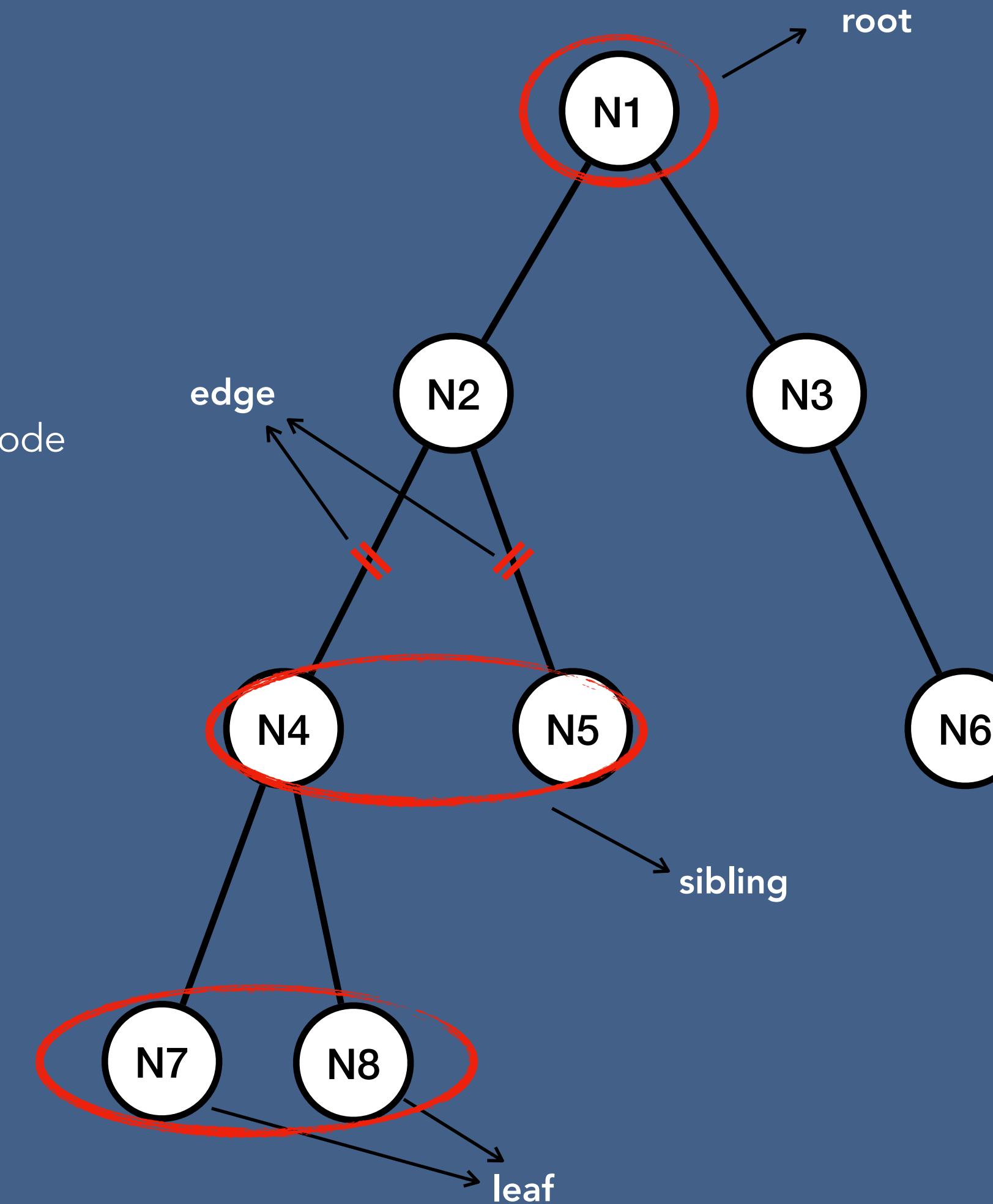
**Root** : top node without parent

**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

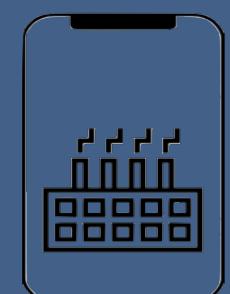
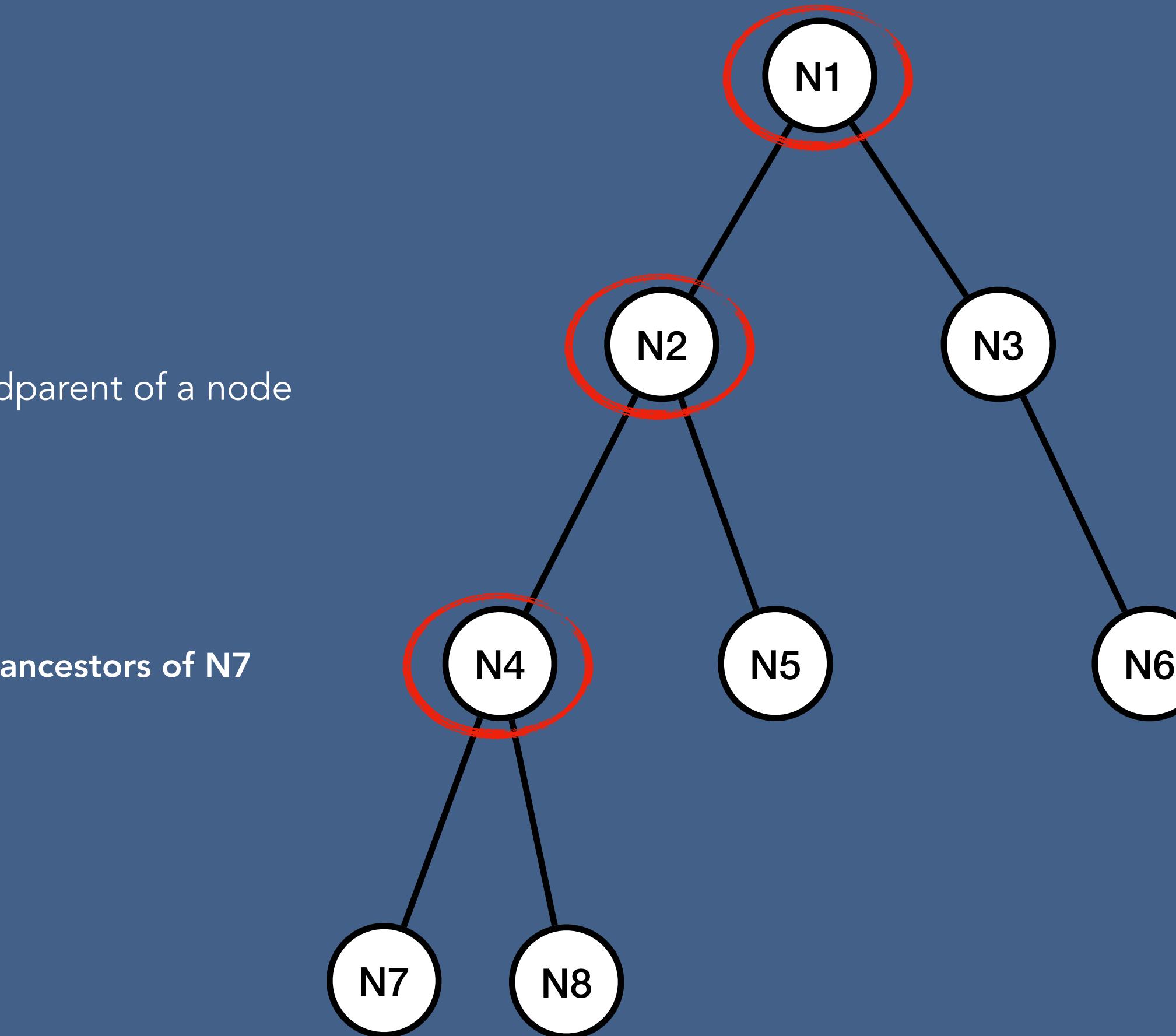
**Root** : top node without parent

**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

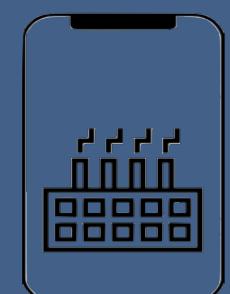
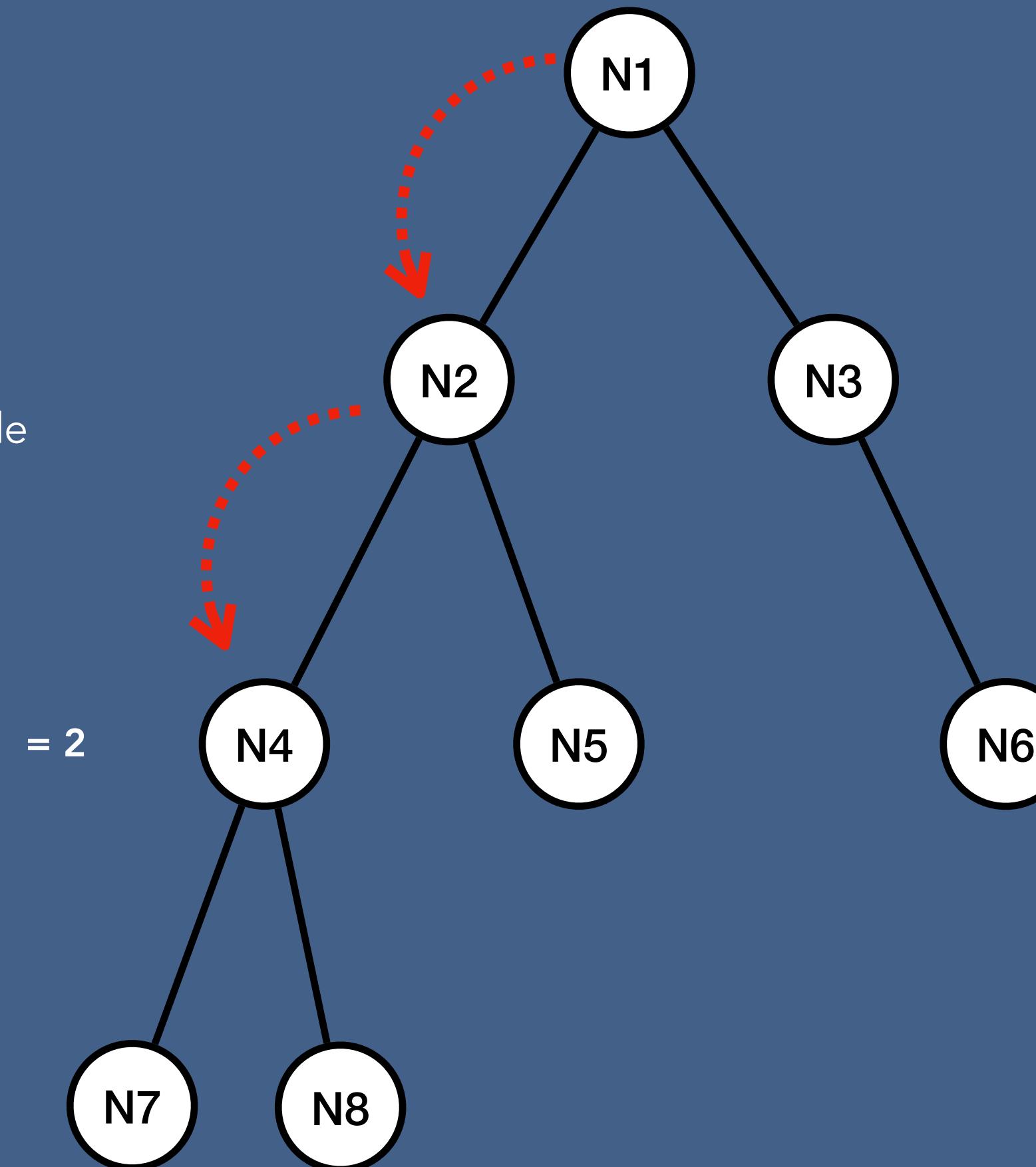
**Edge** : a link between parent and child

**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



# Tree Terminology

**Root** : top node without parent

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

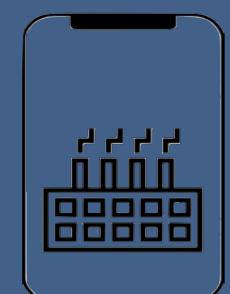
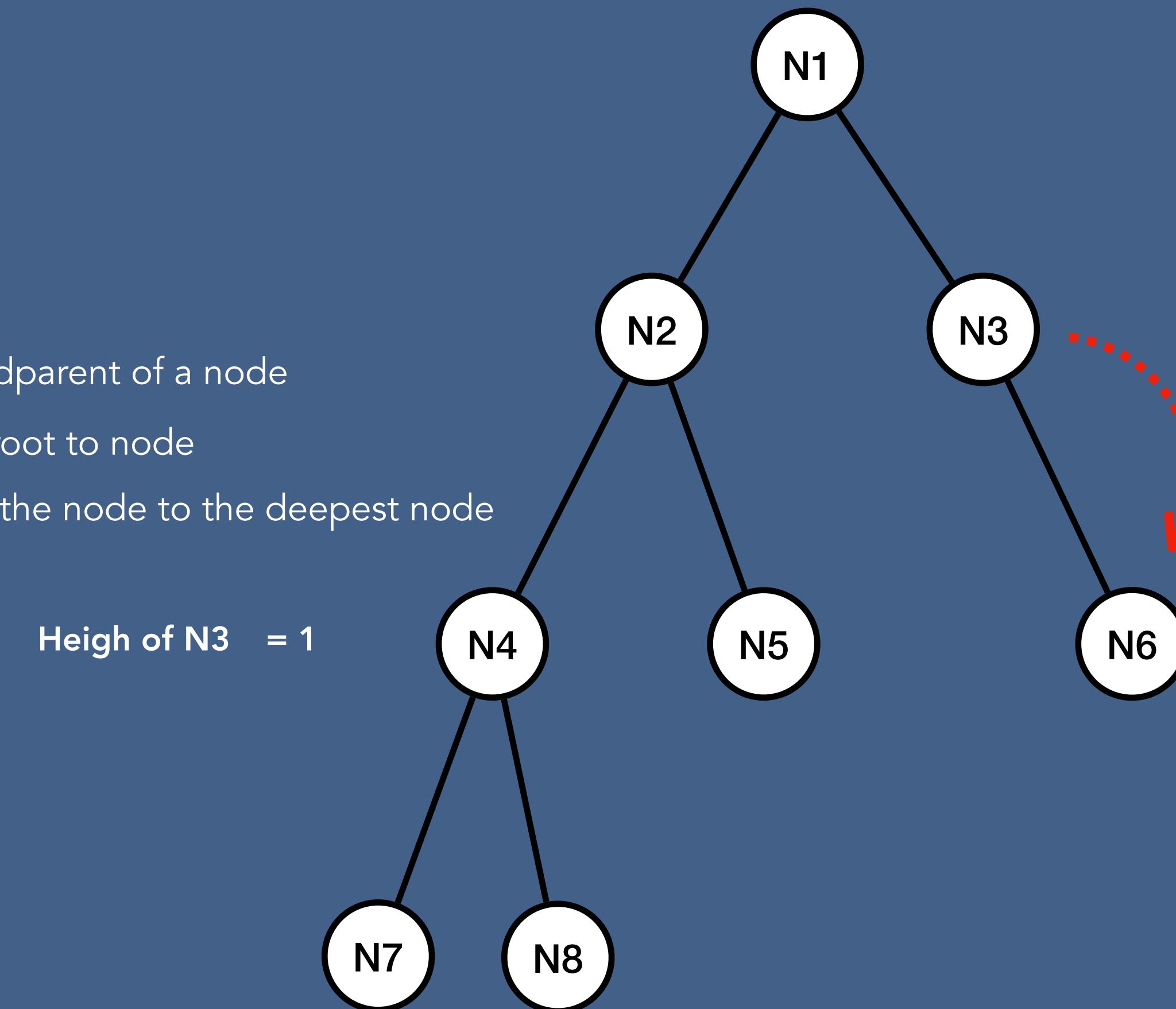
**Leaf** : a node which does not have children

**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



# Tree Terminology

**Root** : top node without parent

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

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

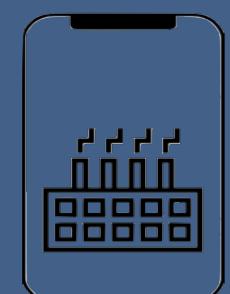
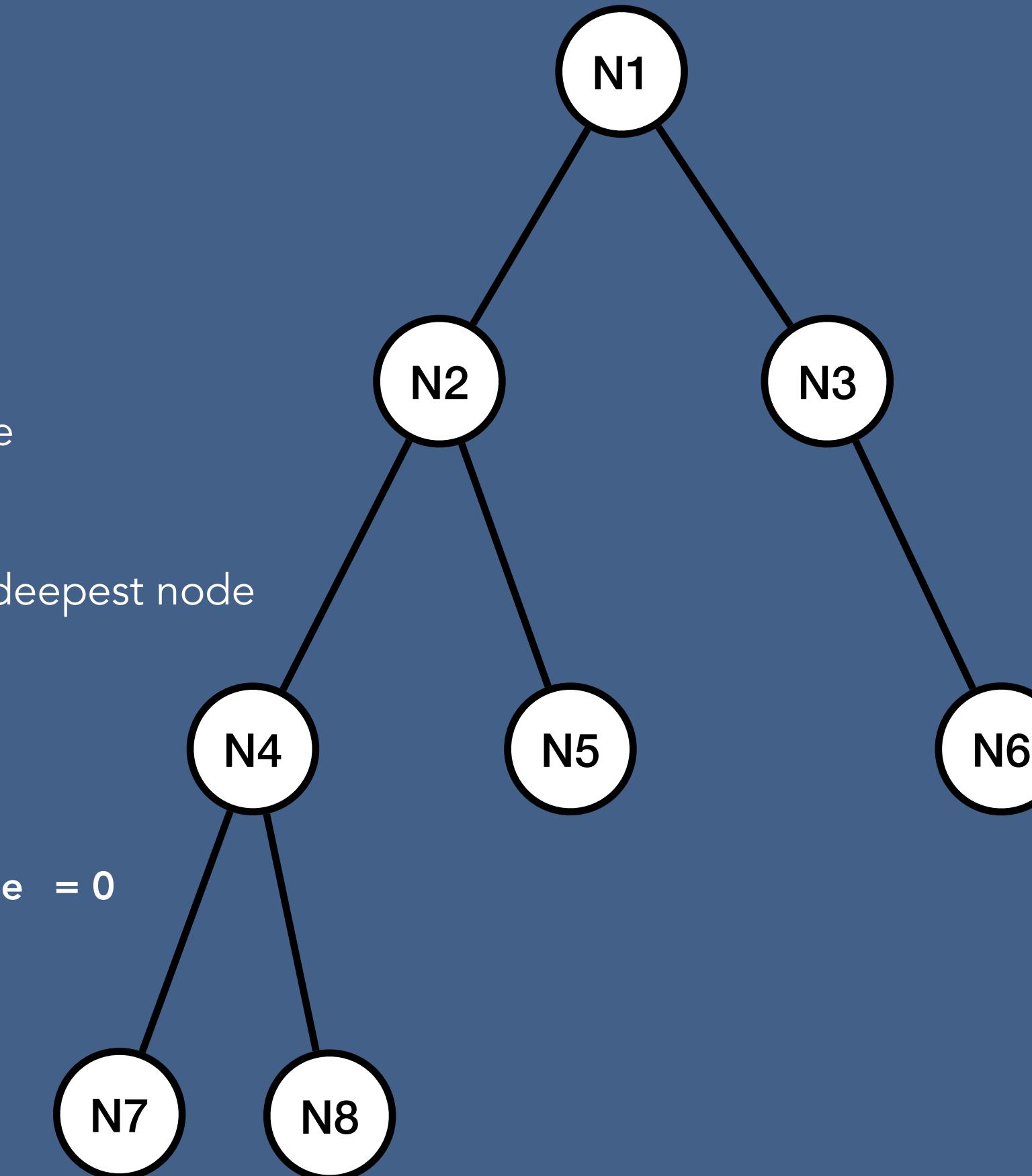
**Sibling** : children of same parent

**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

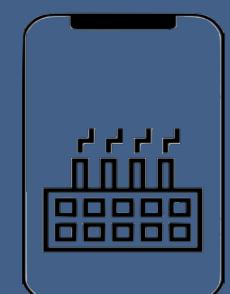
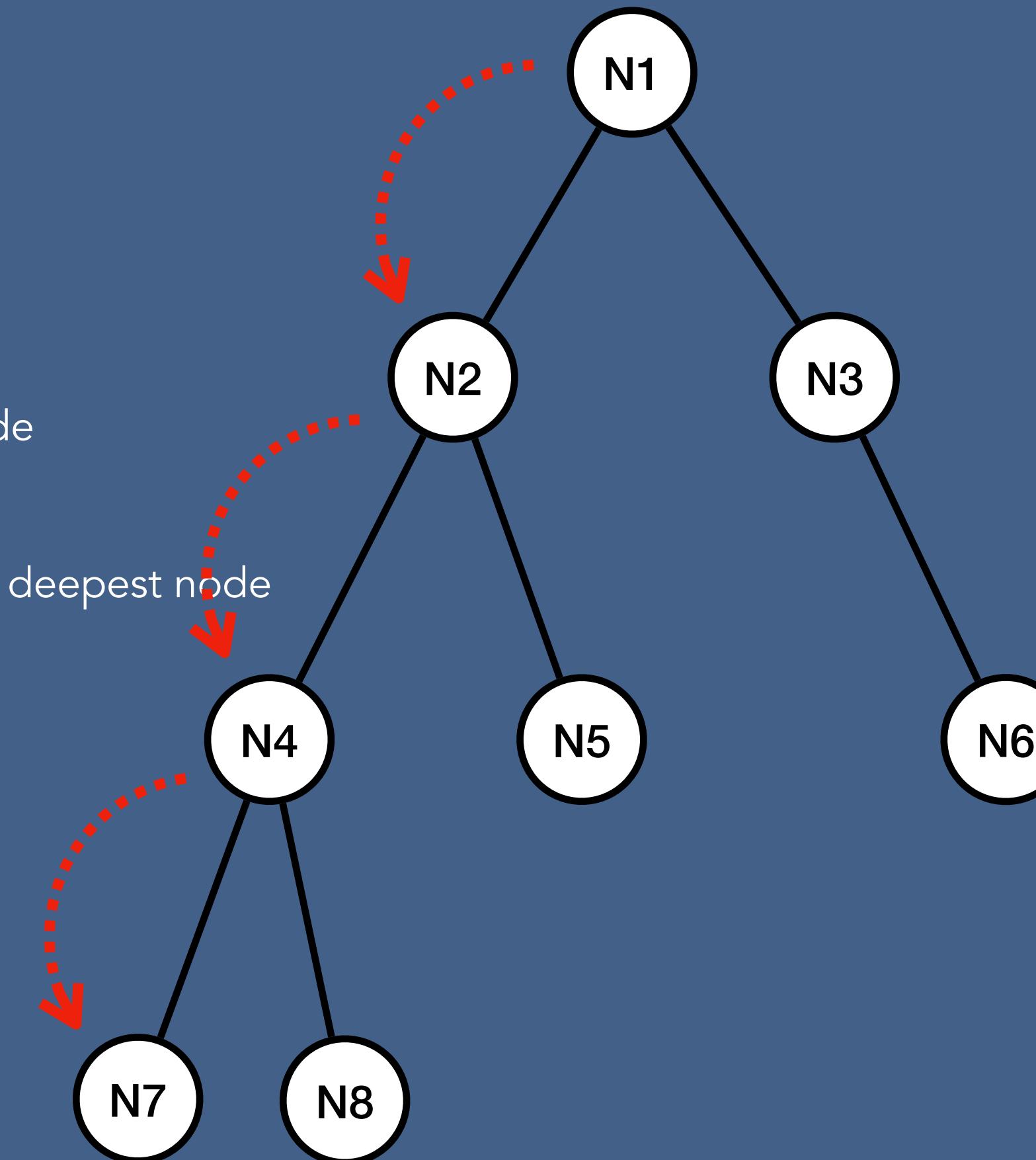
**Ancestor** : parent, grandparent, great grandparent of a node

**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

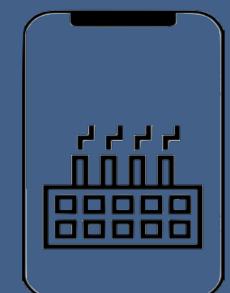
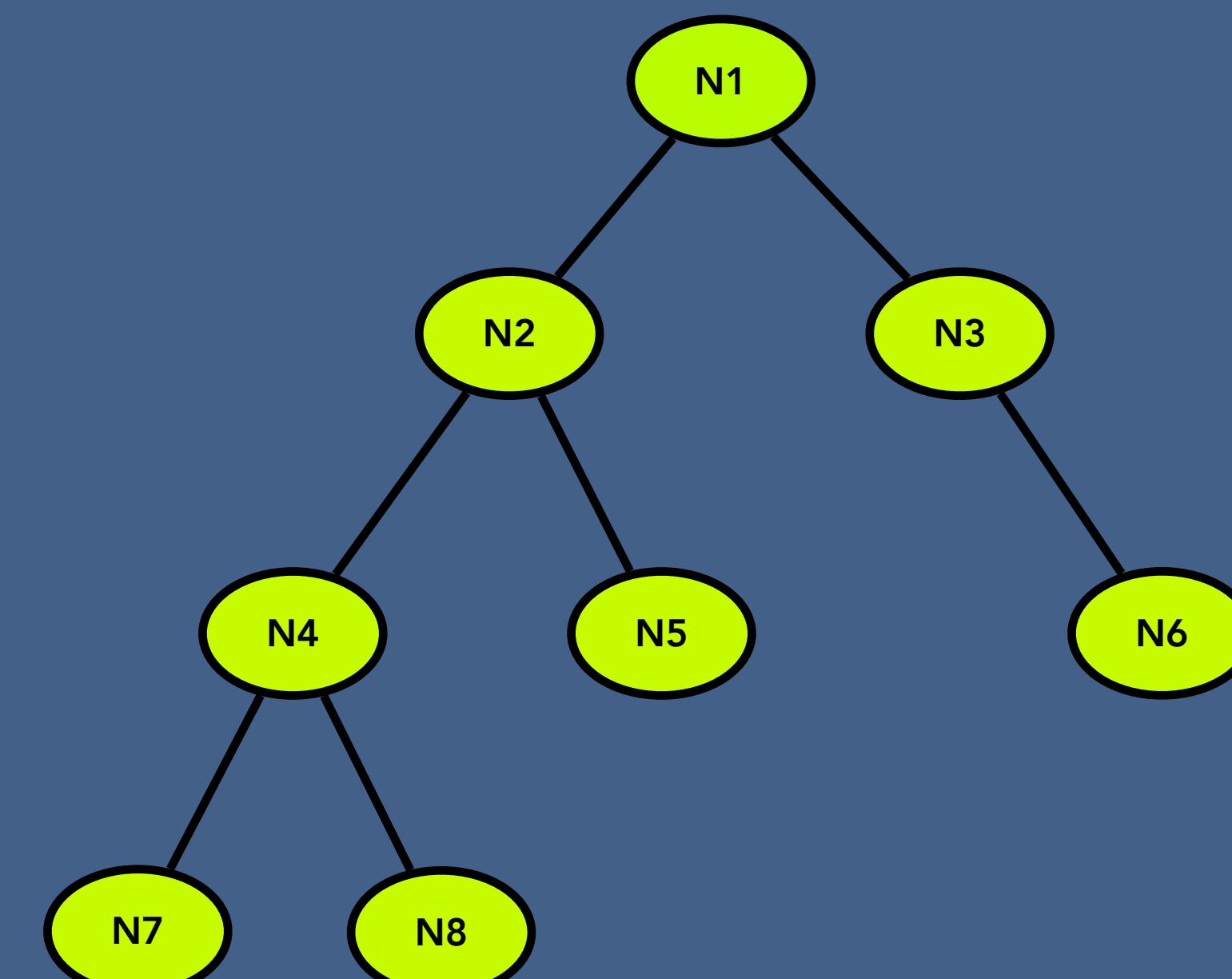
**Depth of tree** : depth of root node

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



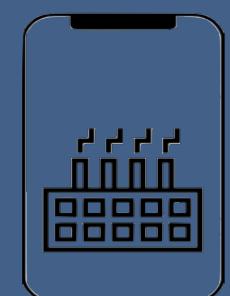
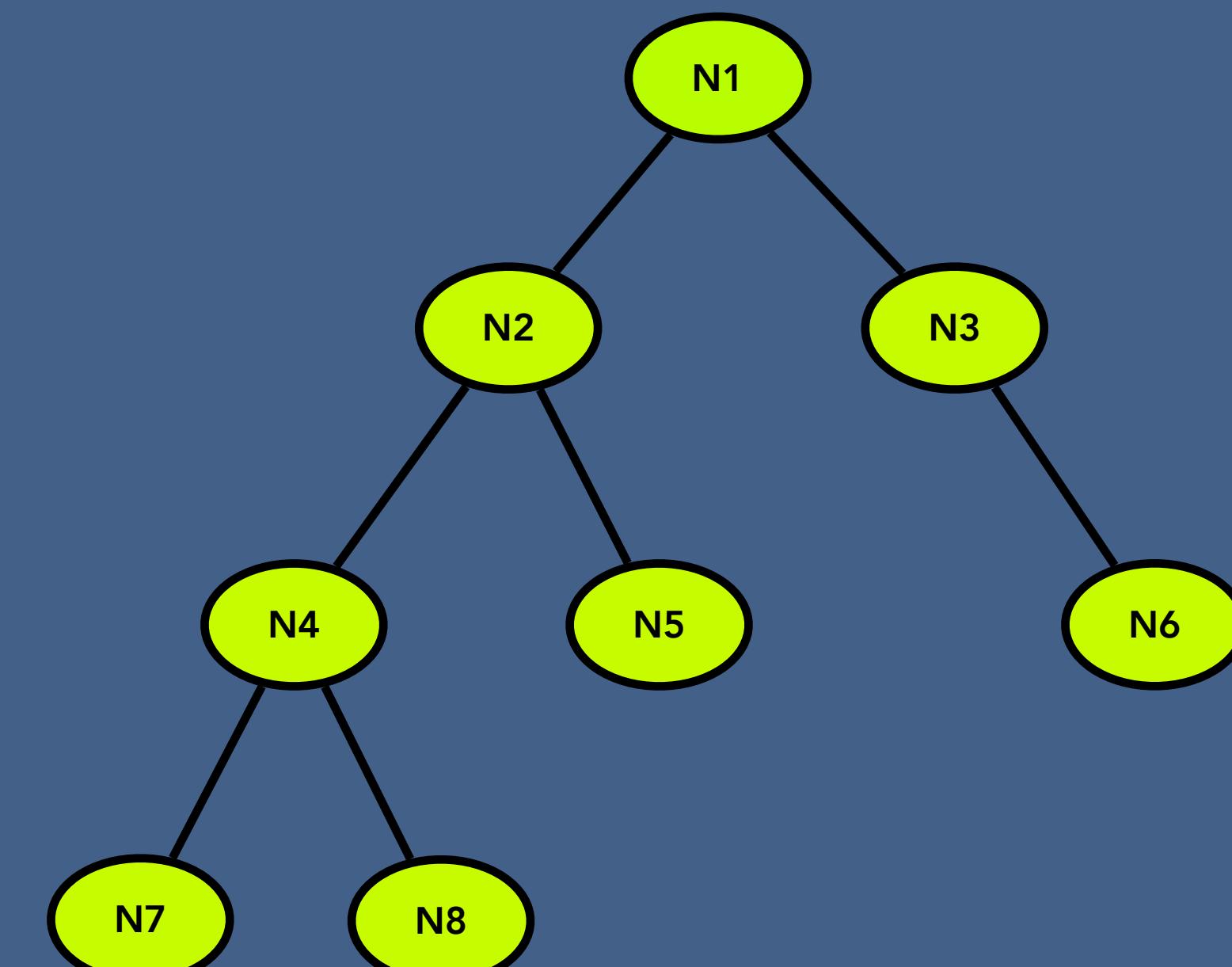
# Binary Tree

- 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)



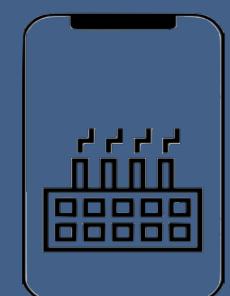
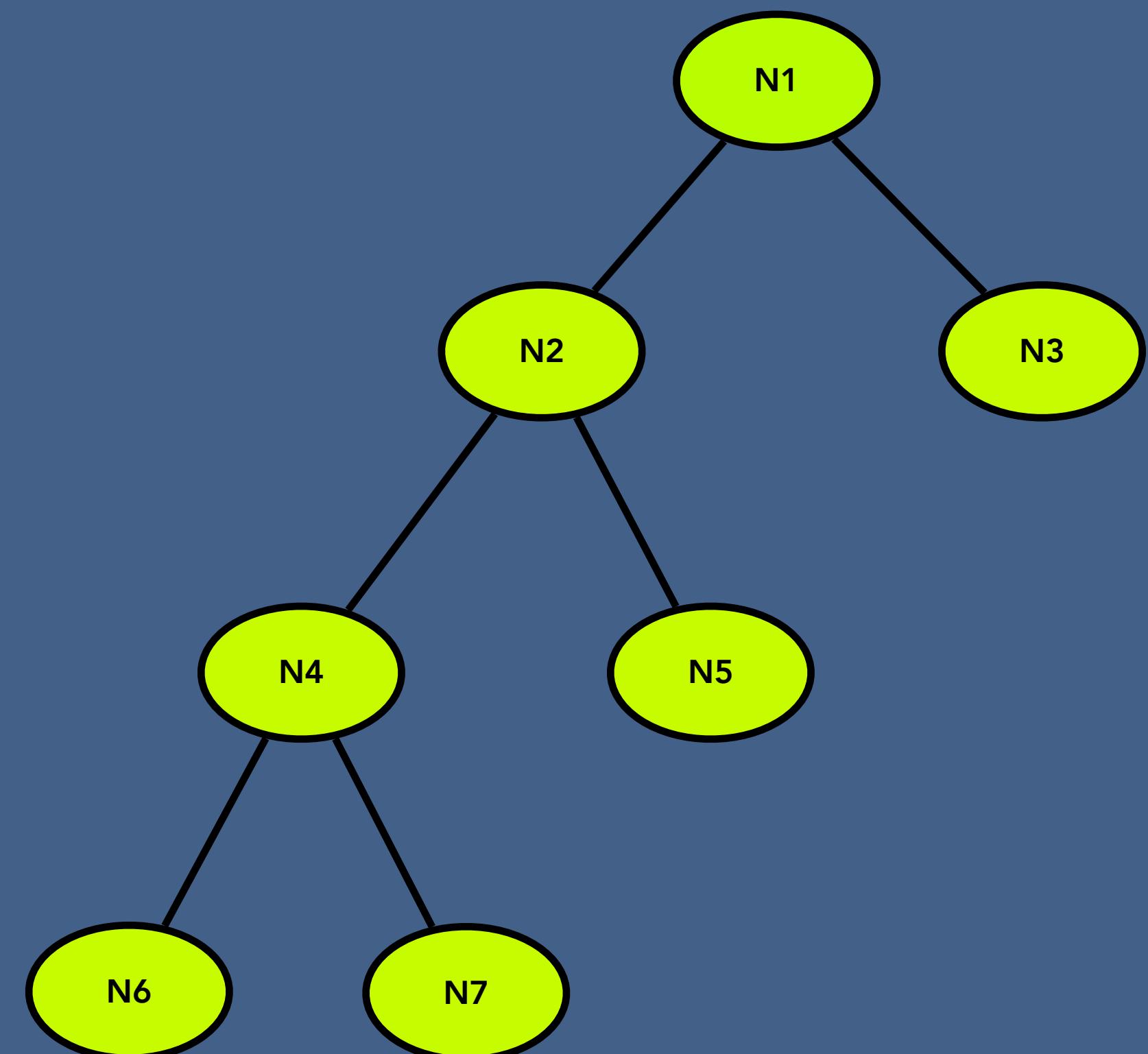
# Why Binary Tree?

- 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,



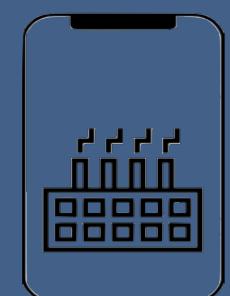
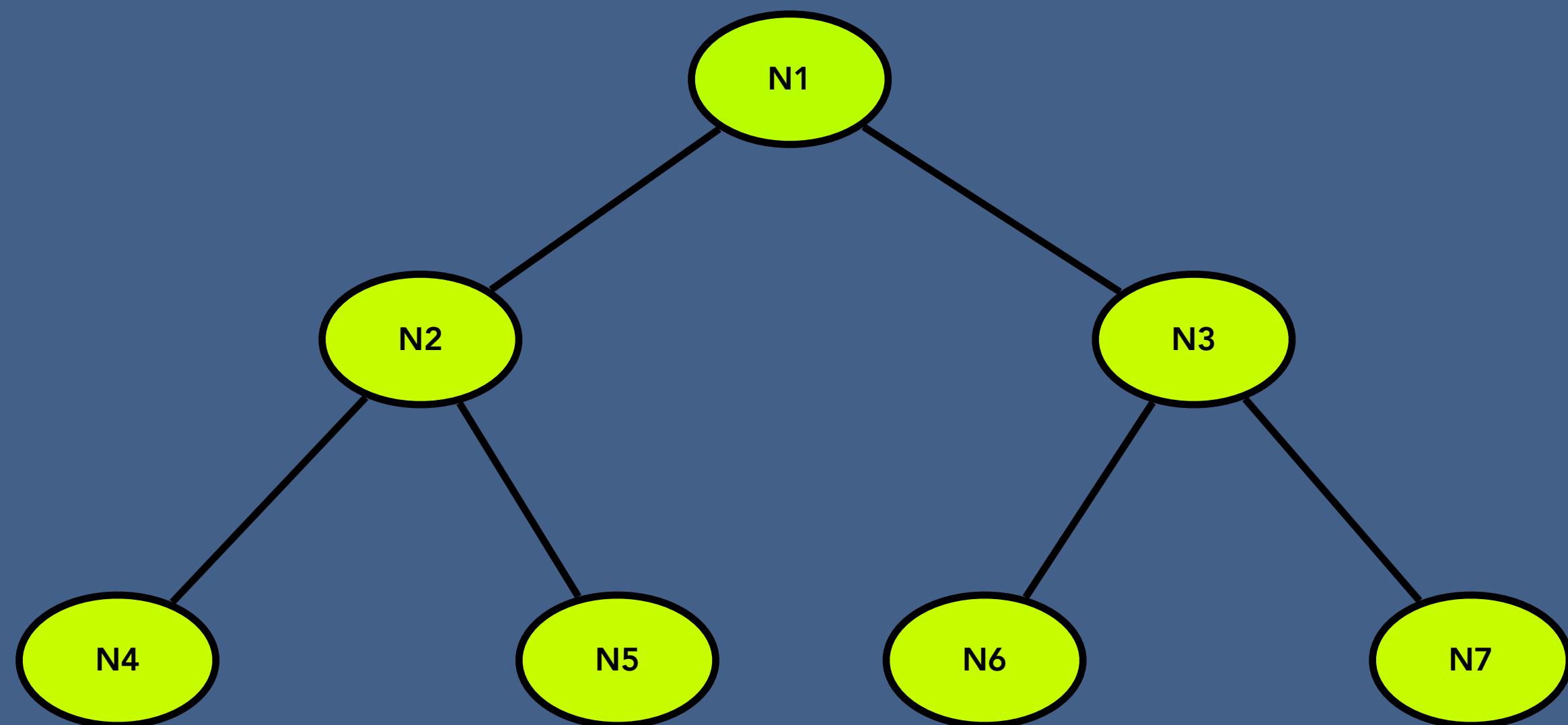
# Types of Binary Tree

## Full Binary Tree



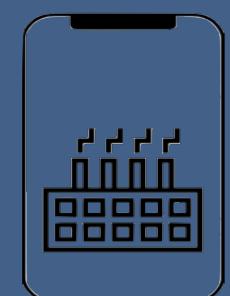
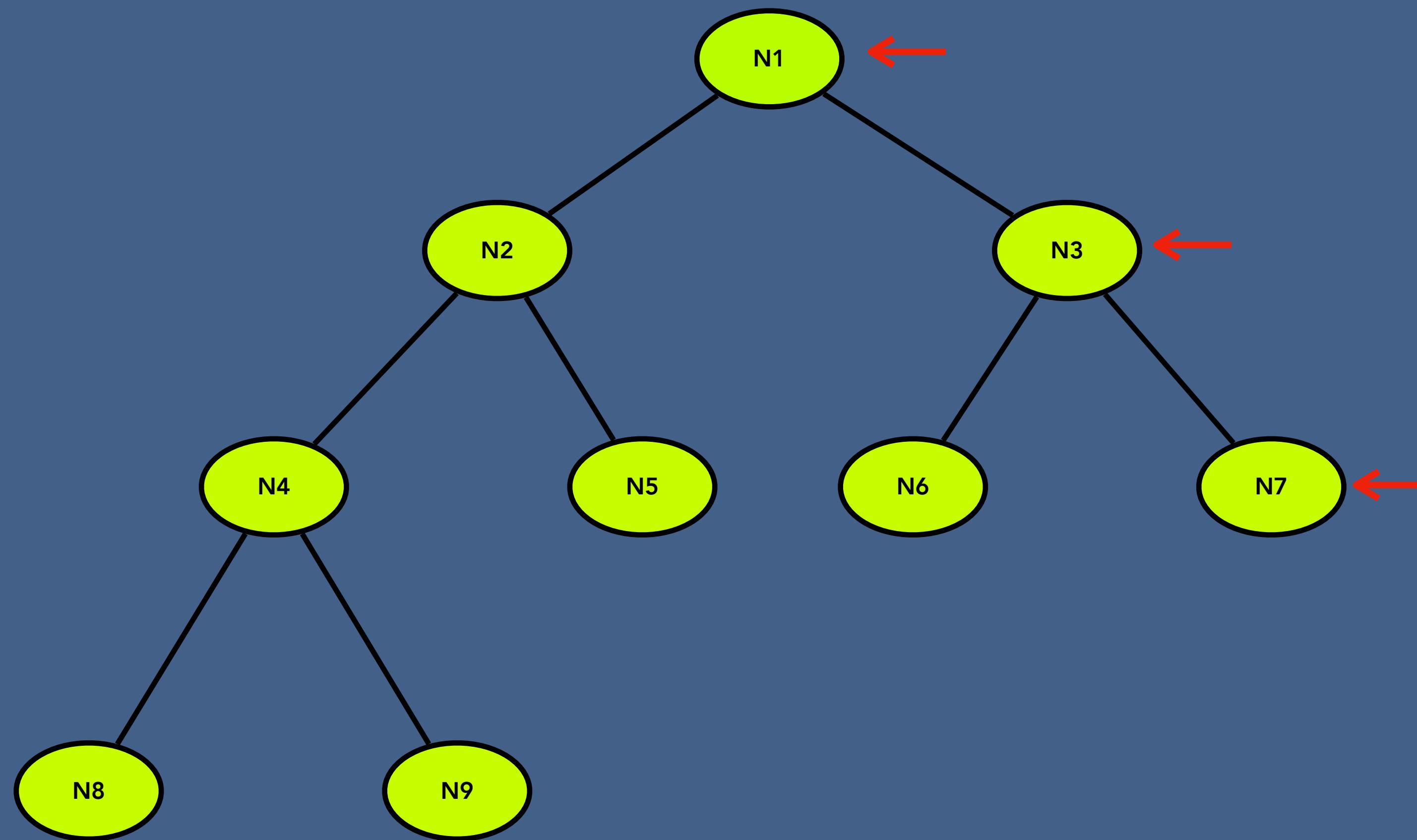
# Types of Binary Tree

## Perfect Binary Tree



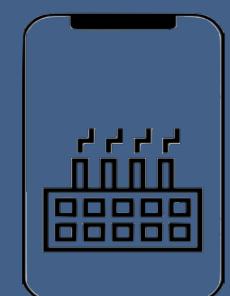
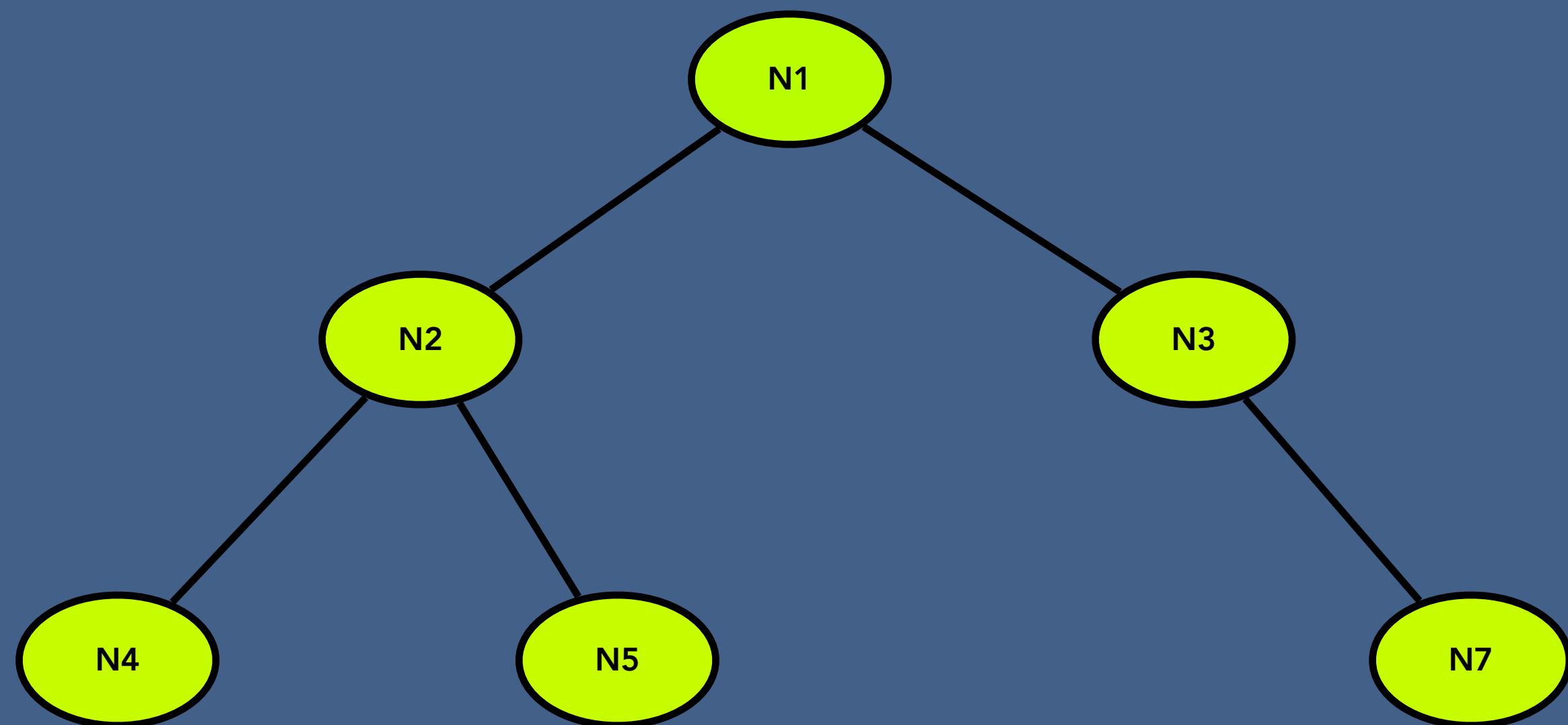
# Types of Binary Tree

## Complete Binary Tree



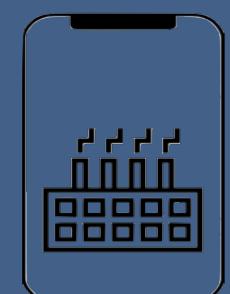
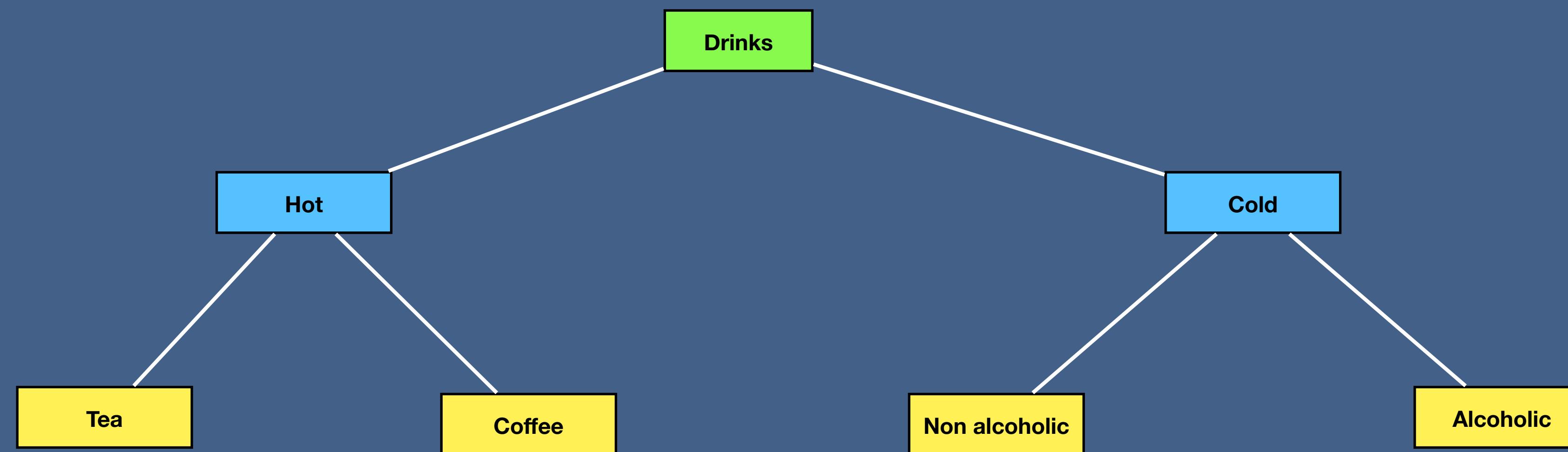
# Types of Binary Tree

## Balanced Binary Tree



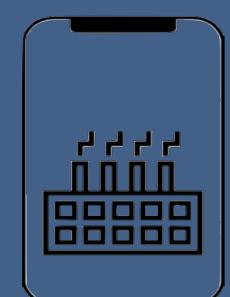
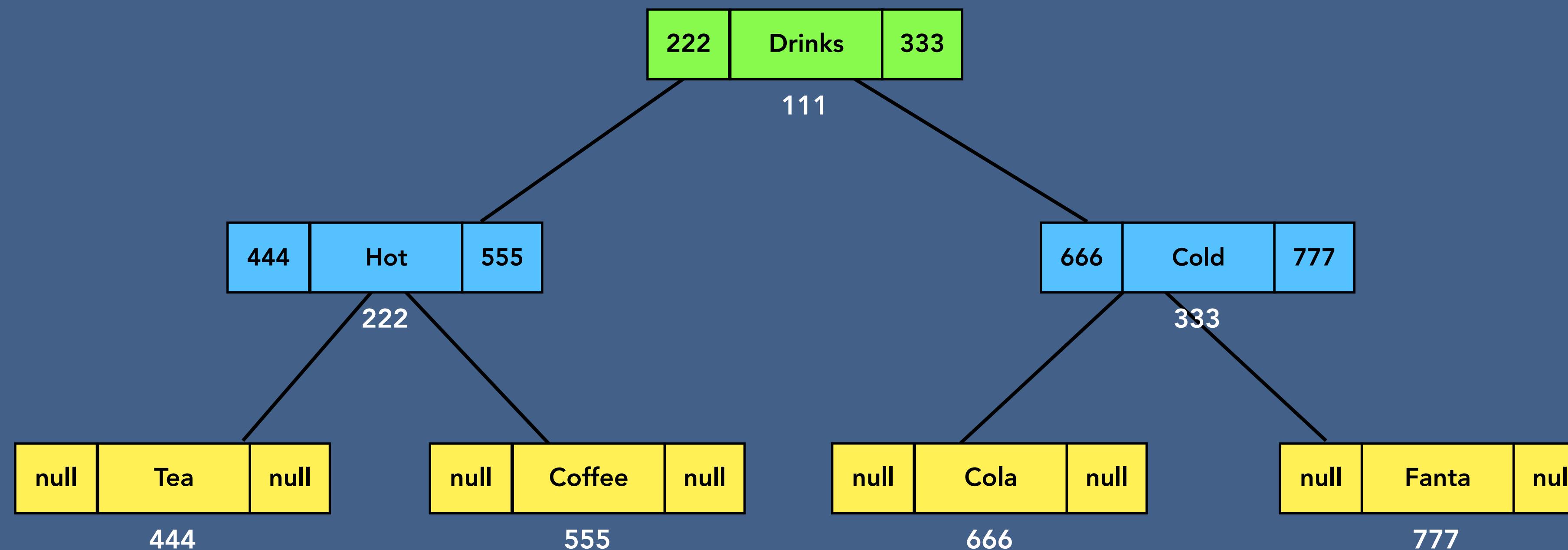
# Binary Tree

- Linked List
- Array



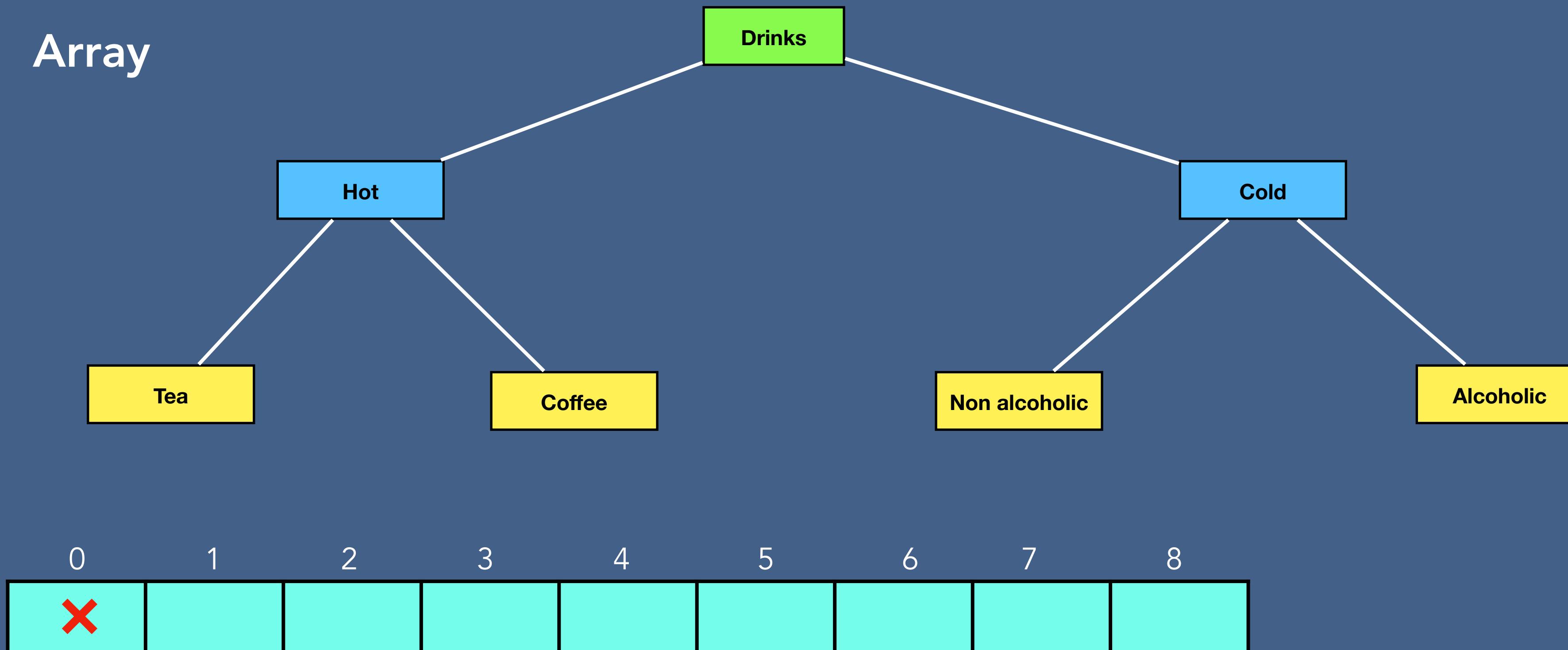
# Binary Tree

## Linked List



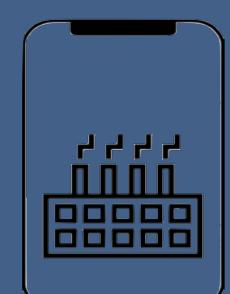
# Binary Tree

Array

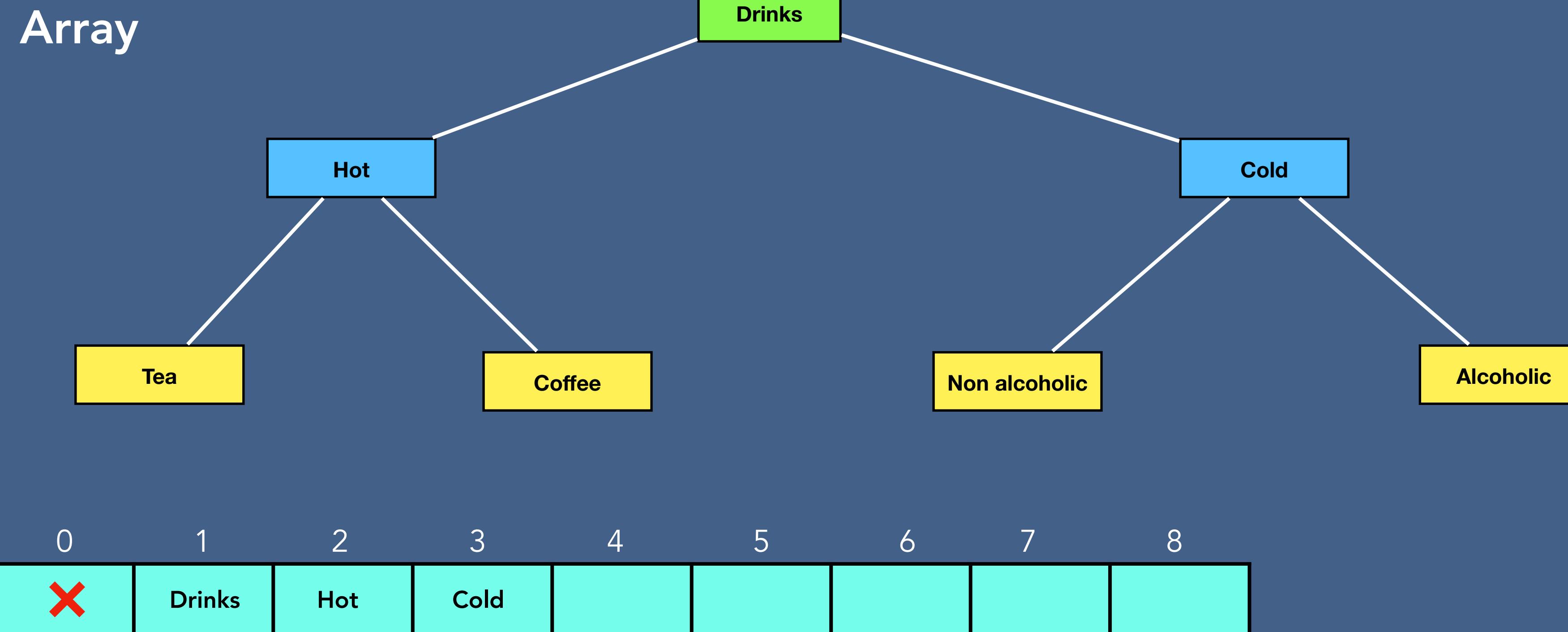


Left child = cell[2x]

Right child = cell[2x+1]

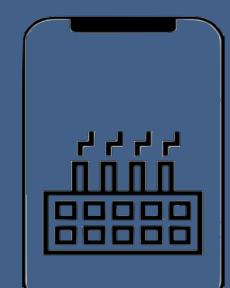


# Binary Tree

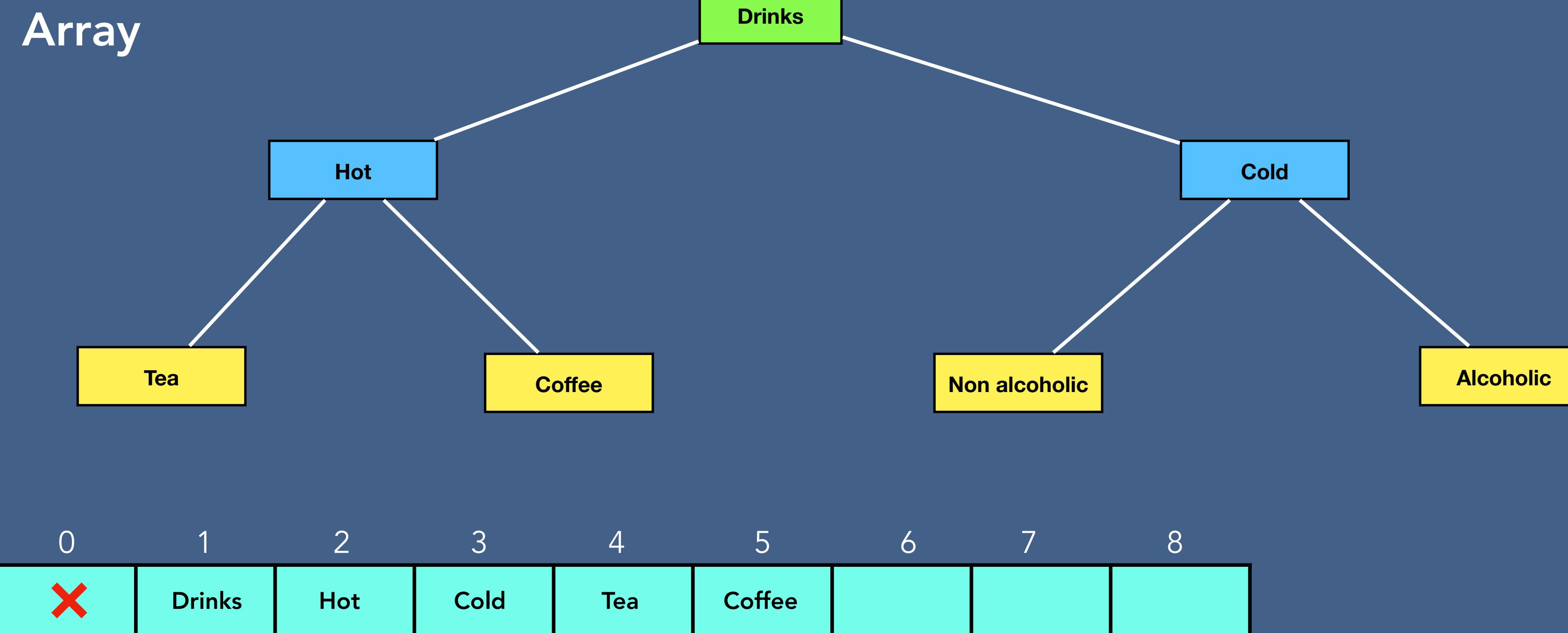


Left child =  $\text{cell}[2x]$   $\longrightarrow$   $x = 1, \text{cell}[2 \times 1 = 2]$

Right child =  $\text{cell}[2x+1]$   $\longrightarrow$   $x = 1, \text{cell}[2 \times 1 + 1 = 3]$

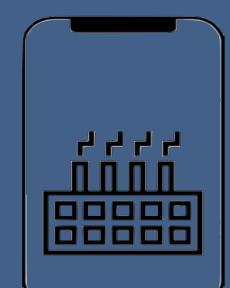


# Binary Tree

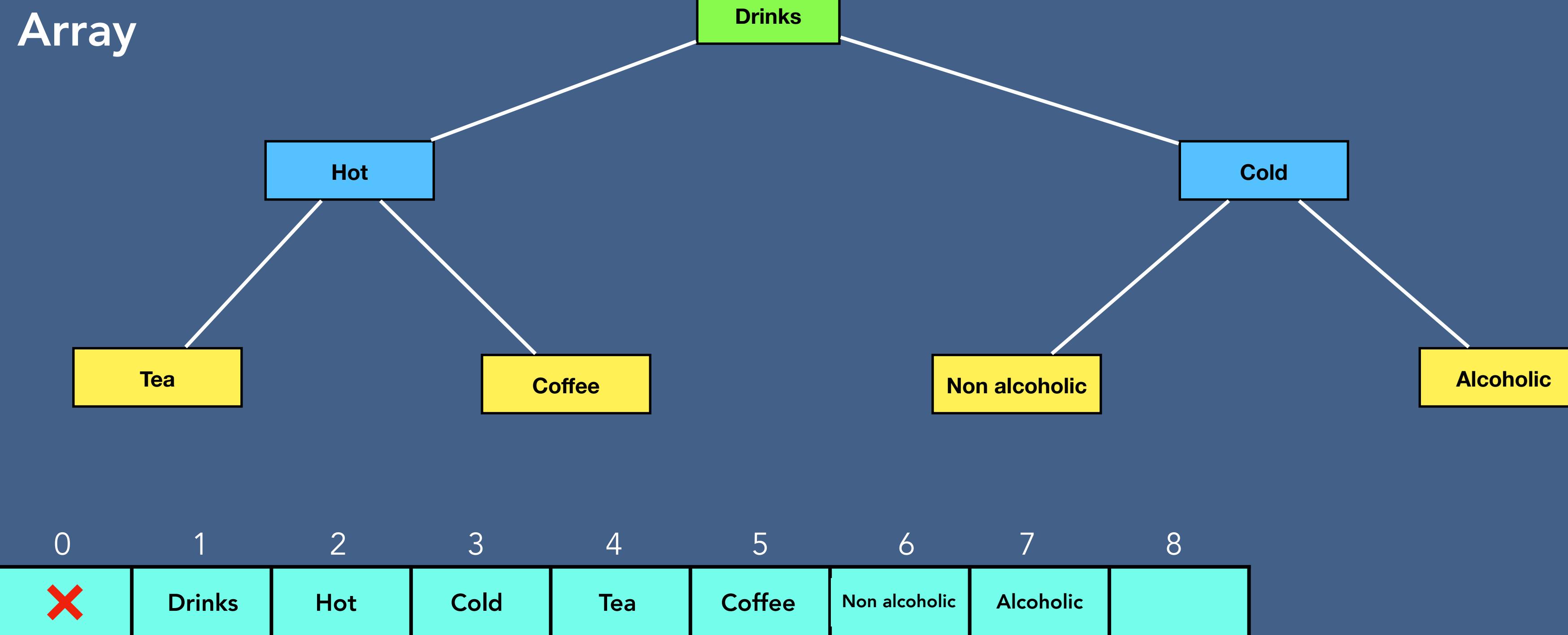


Left child =  $\text{cell}[2x]$   $\longrightarrow$   $x = 2, \text{cell}[2 \times 2 = 4]$

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

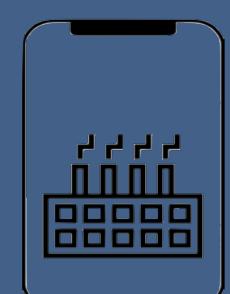


# Binary Tree



Left child =  $\text{cell}[2x]$   $\longrightarrow$   $x = 3, \text{cell}[2 \times 3 = 6]$

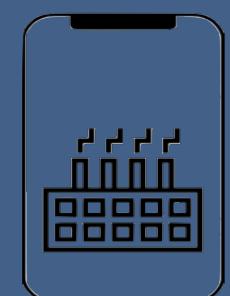
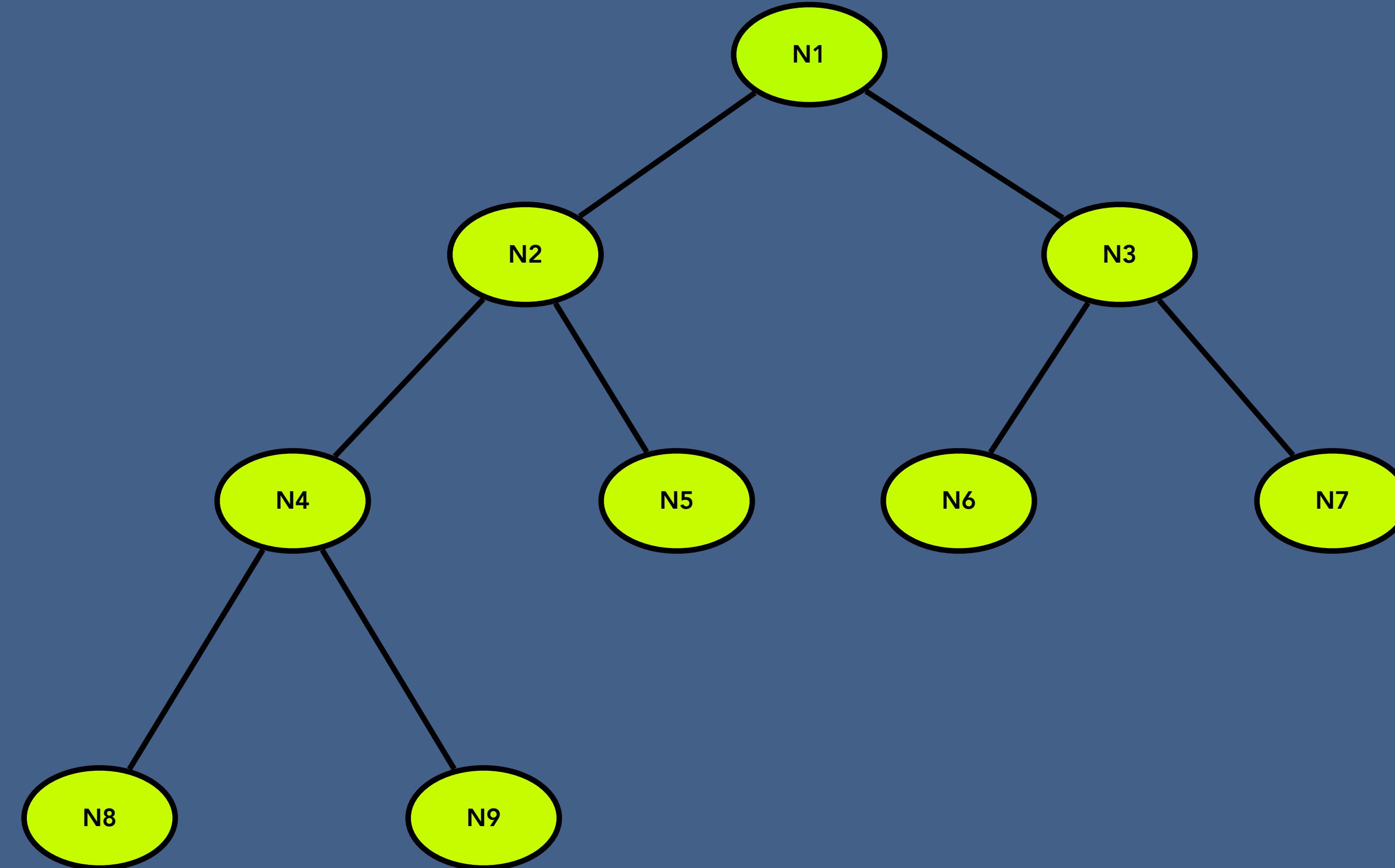
Right child =  $\text{cell}[2x+1]$   $\longrightarrow$   $x = 3, \text{cell}[2 \times 3 + 1 = 7]$



# 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()
```



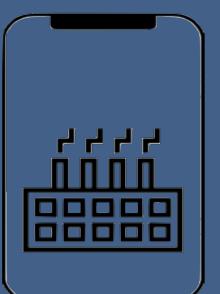
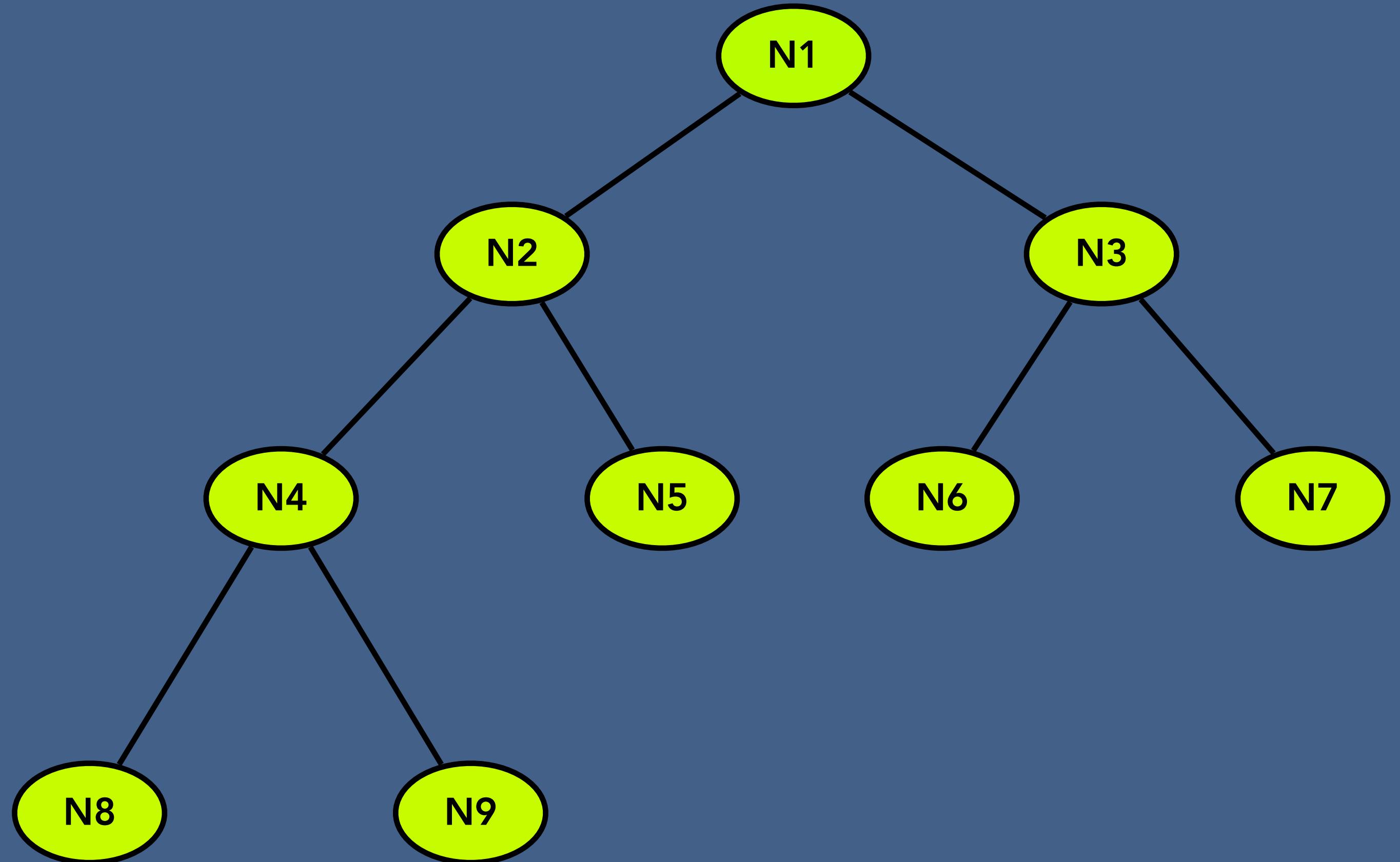
# Binary Tree - Traversal

## Depth first search

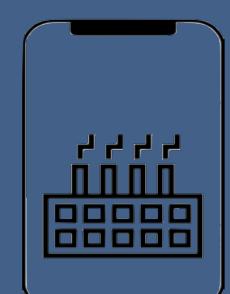
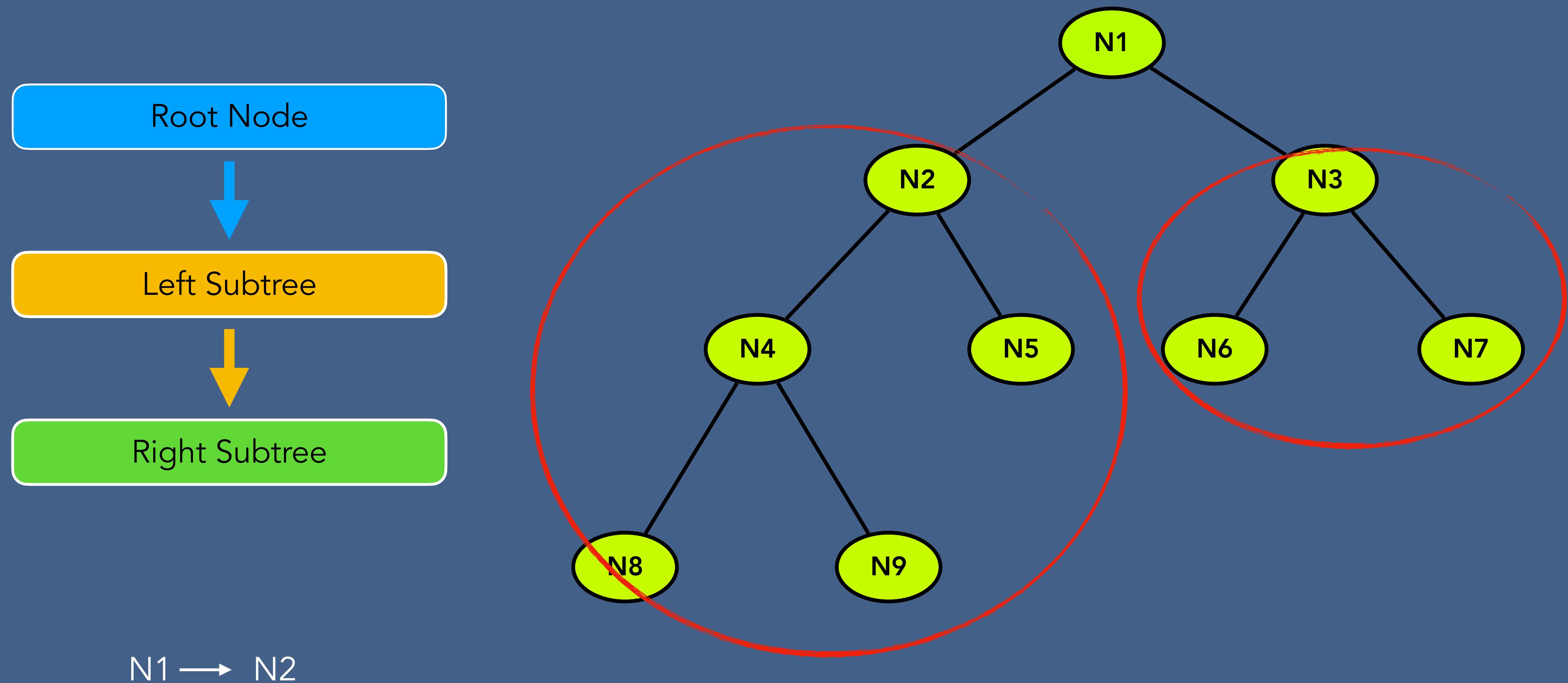
- Preorder traversal
- Inorder traversal
- Post order traversal

## Breadth first search

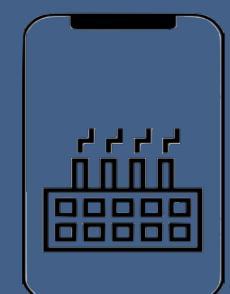
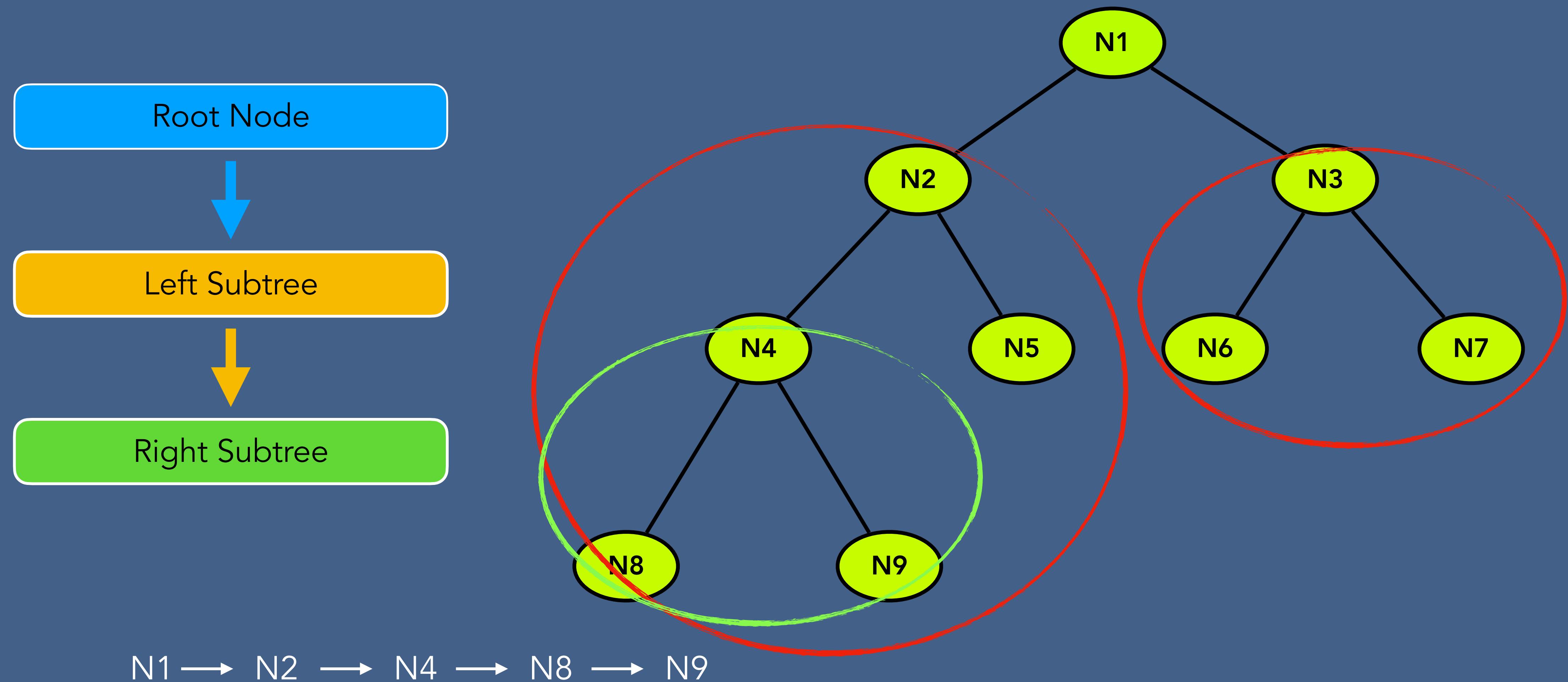
- Level order traversal



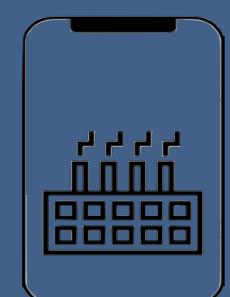
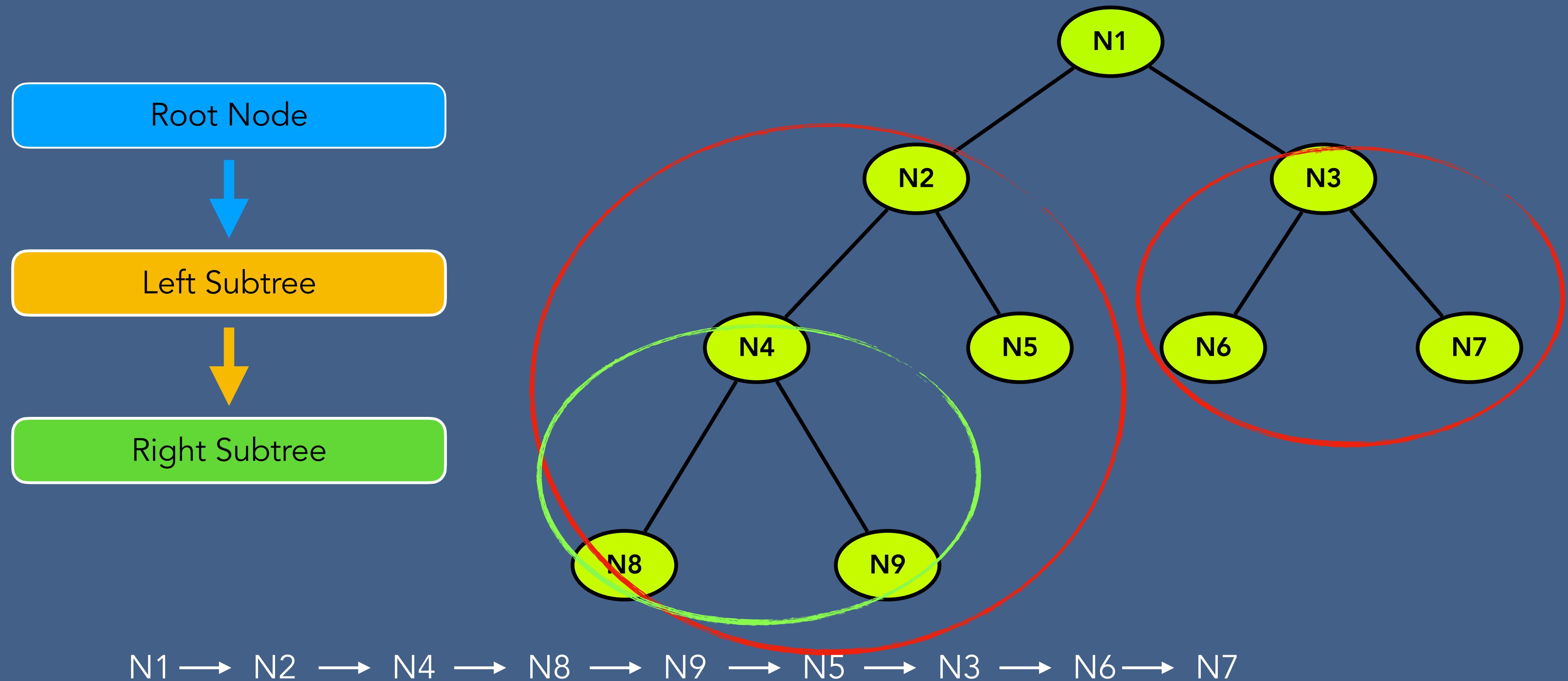
# Binary Tree - PreOrder Traversal



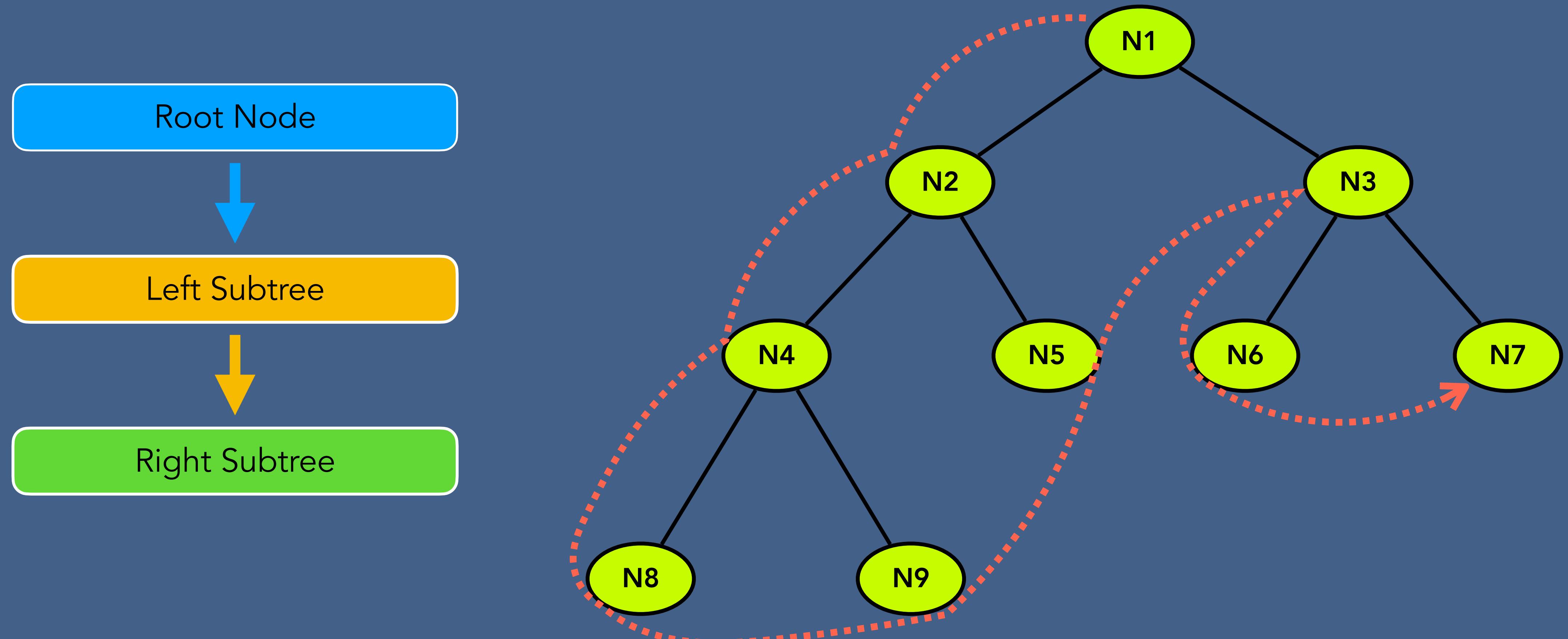
# Binary Tree - PreOrder Traversal



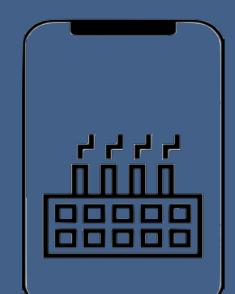
# Binary Tree - PreOrder Traversal



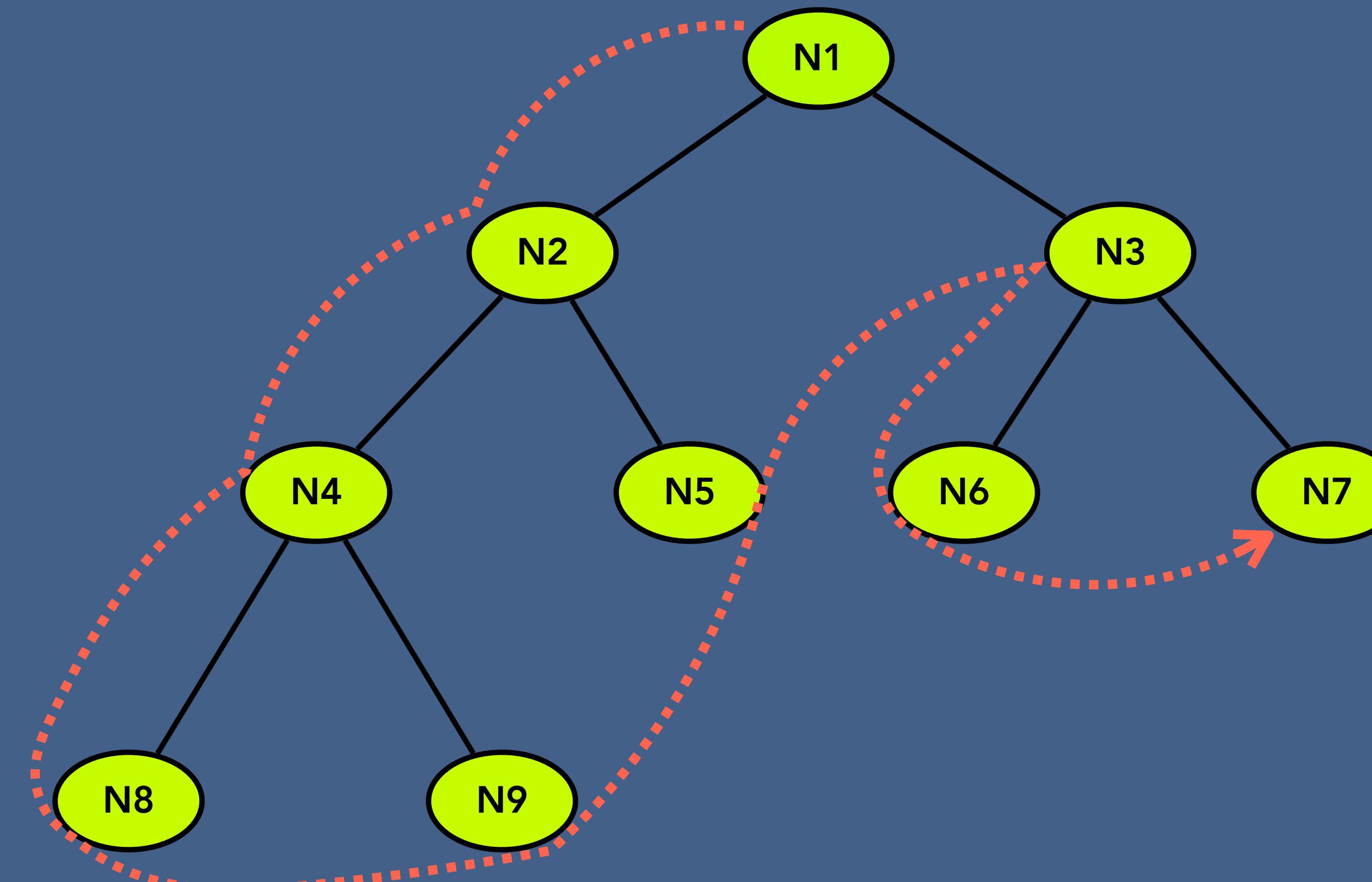
# Binary Tree - PreOrder Traversal



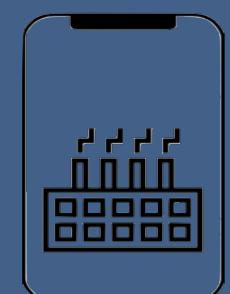
N1 → N2 → N4 → N8 → N9 → N5 → N3 → N6 → N7



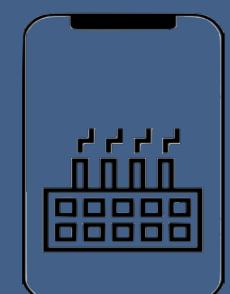
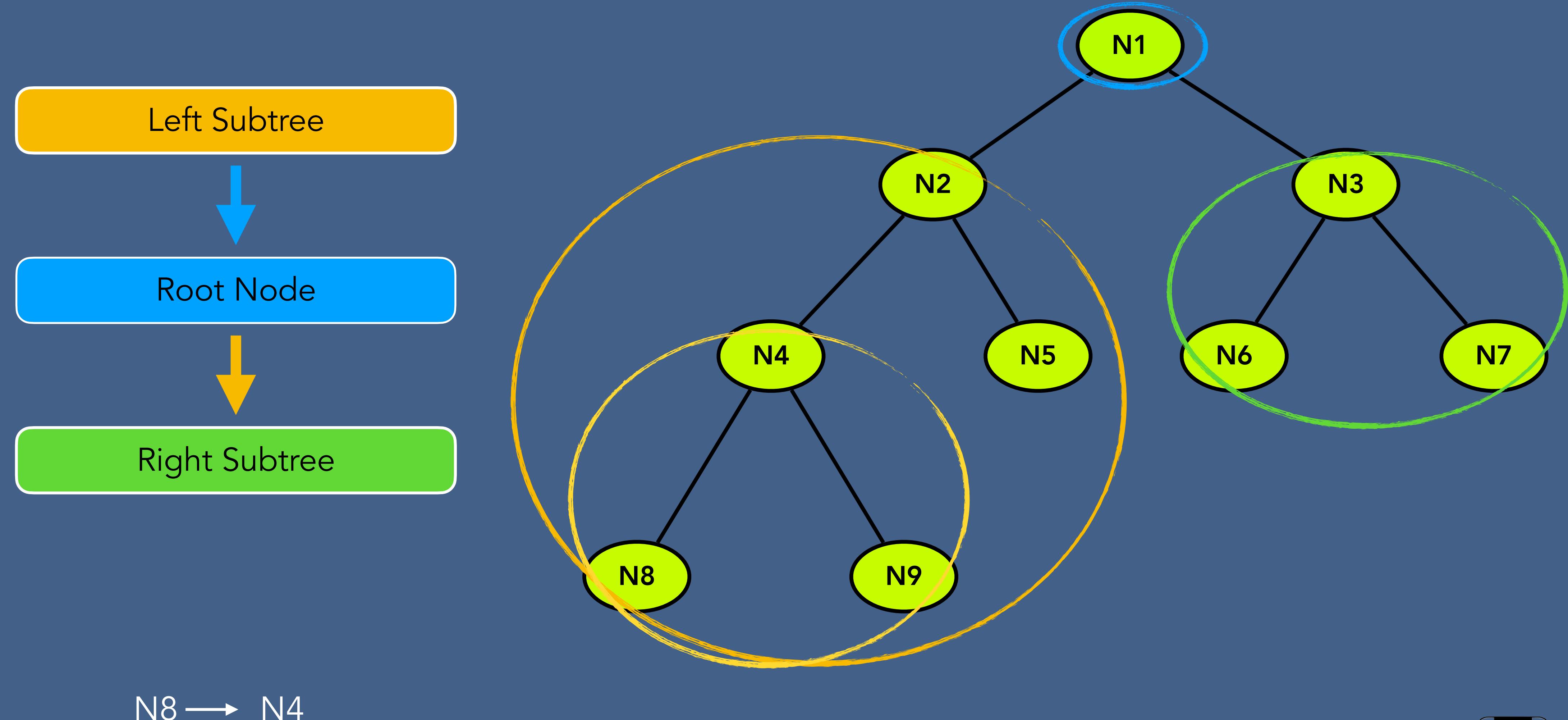
# Binary Tree - PreOrder Traversal



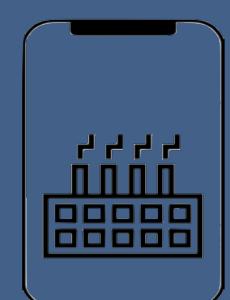
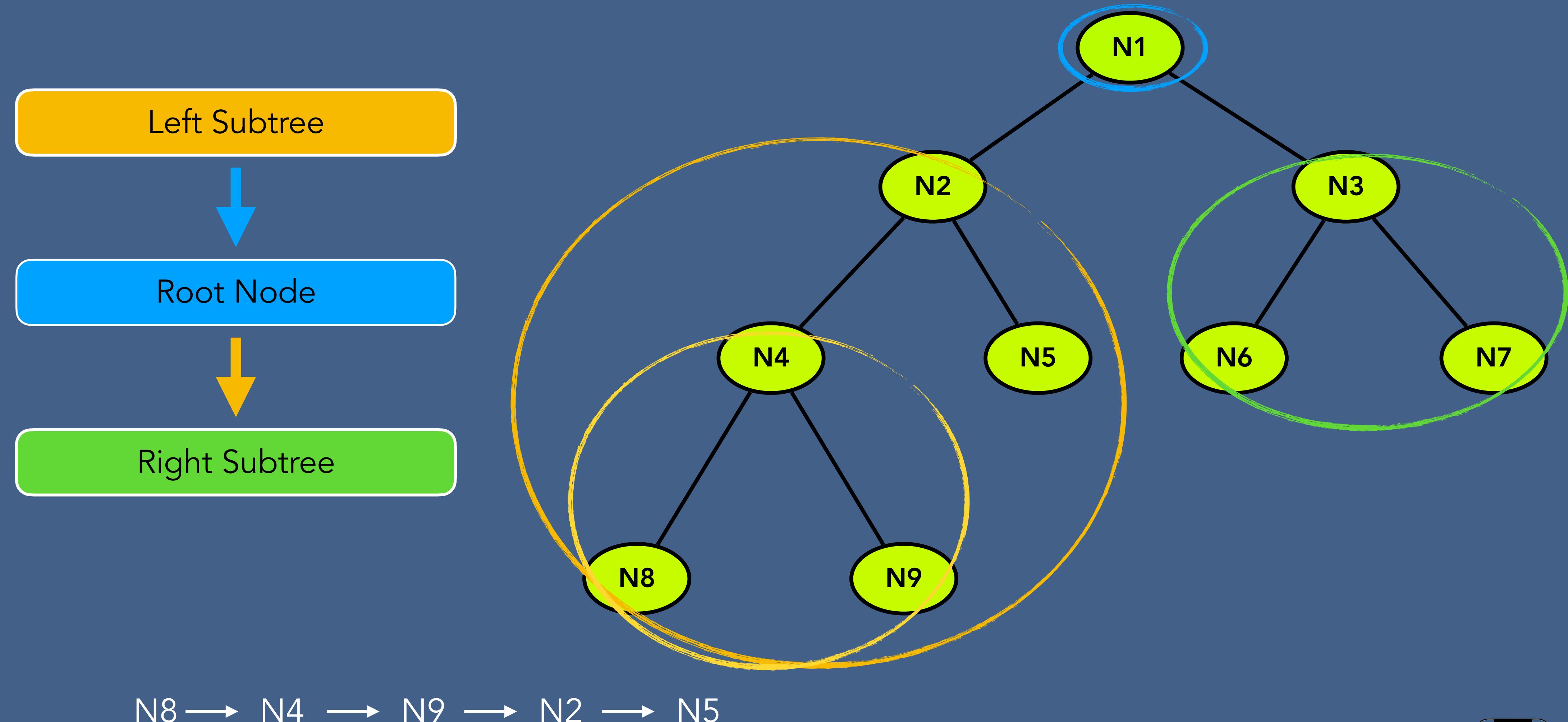
N1 → N2 → N4 → N8 → N9 → N5 → N3 → N6 → N7



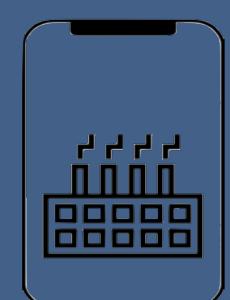
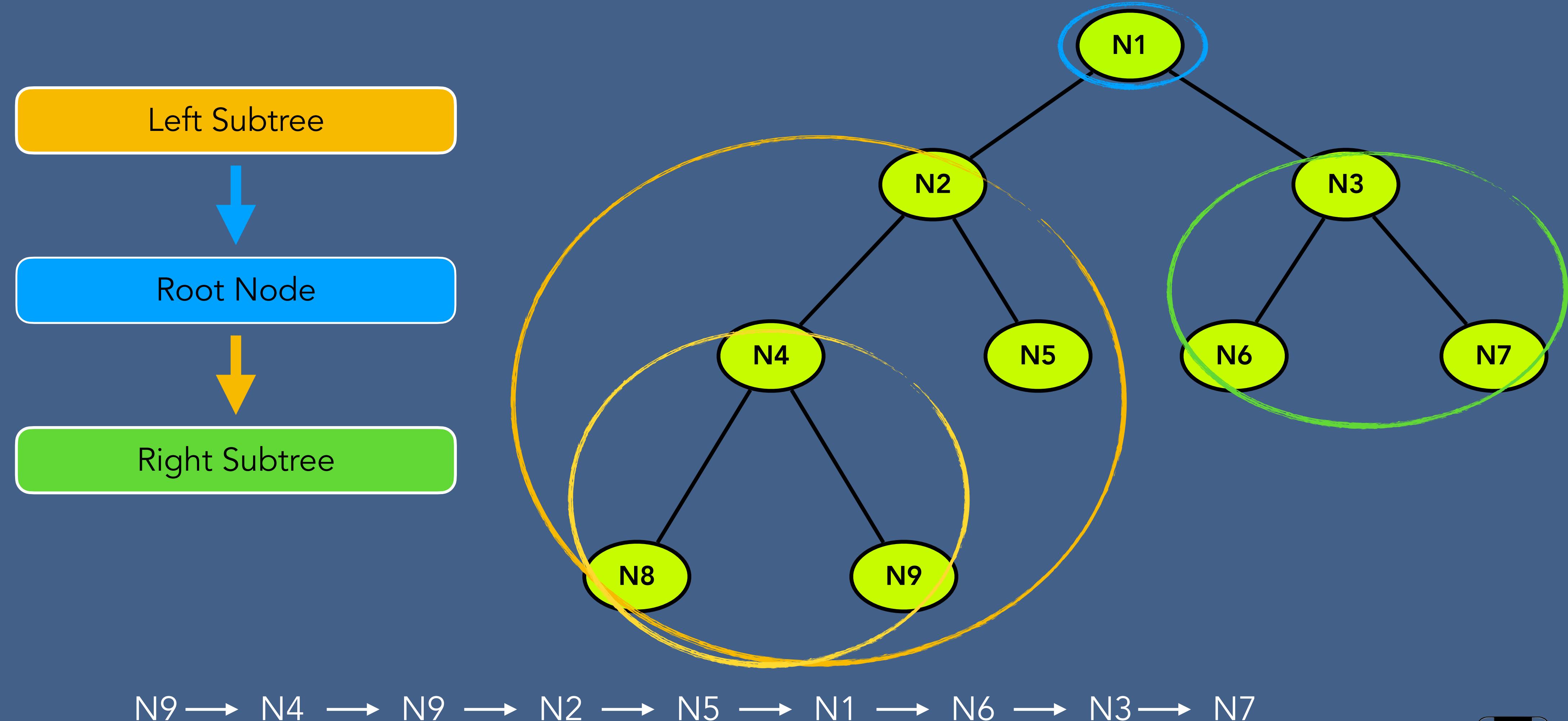
# Binary Tree - InOrder Traversal



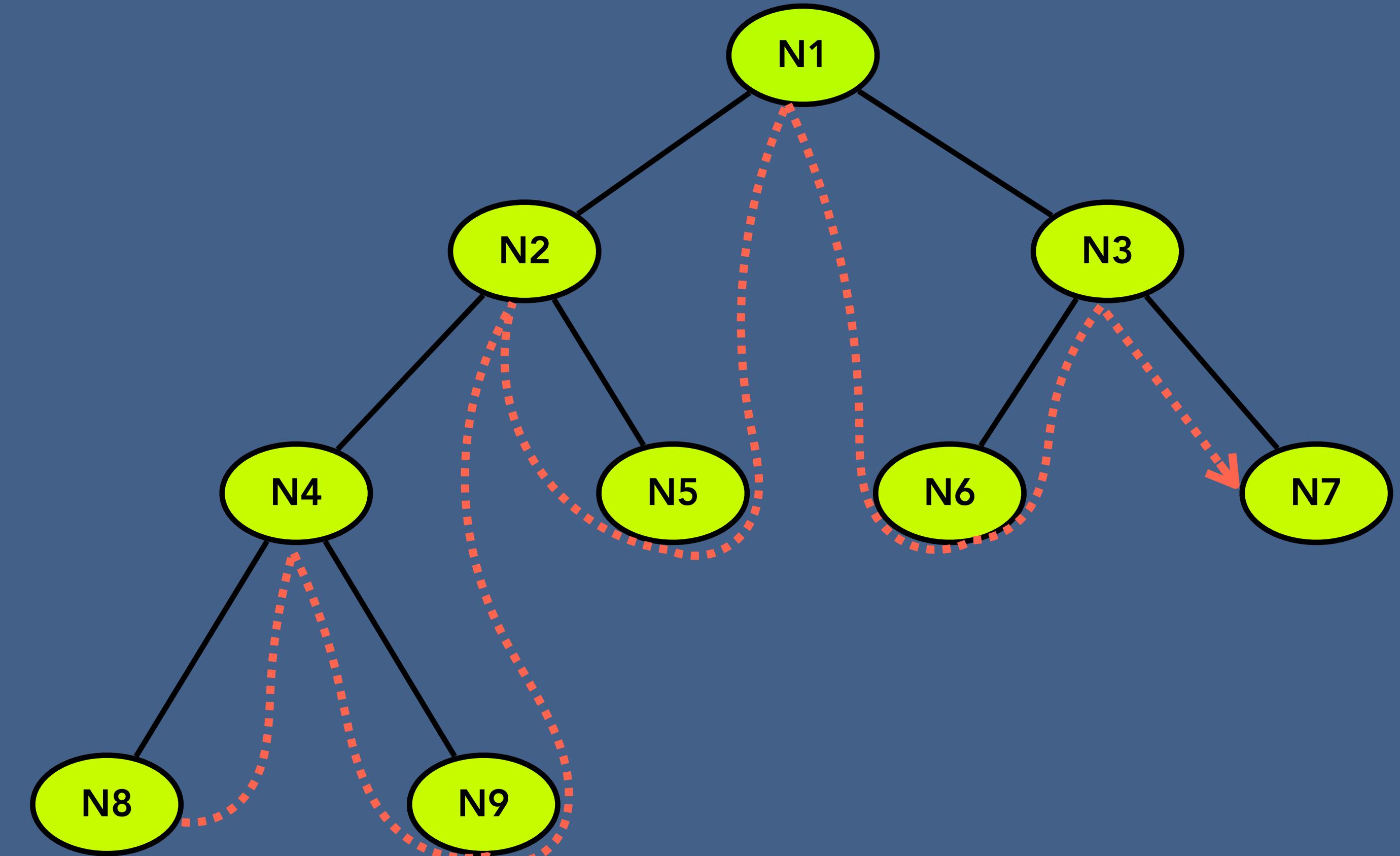
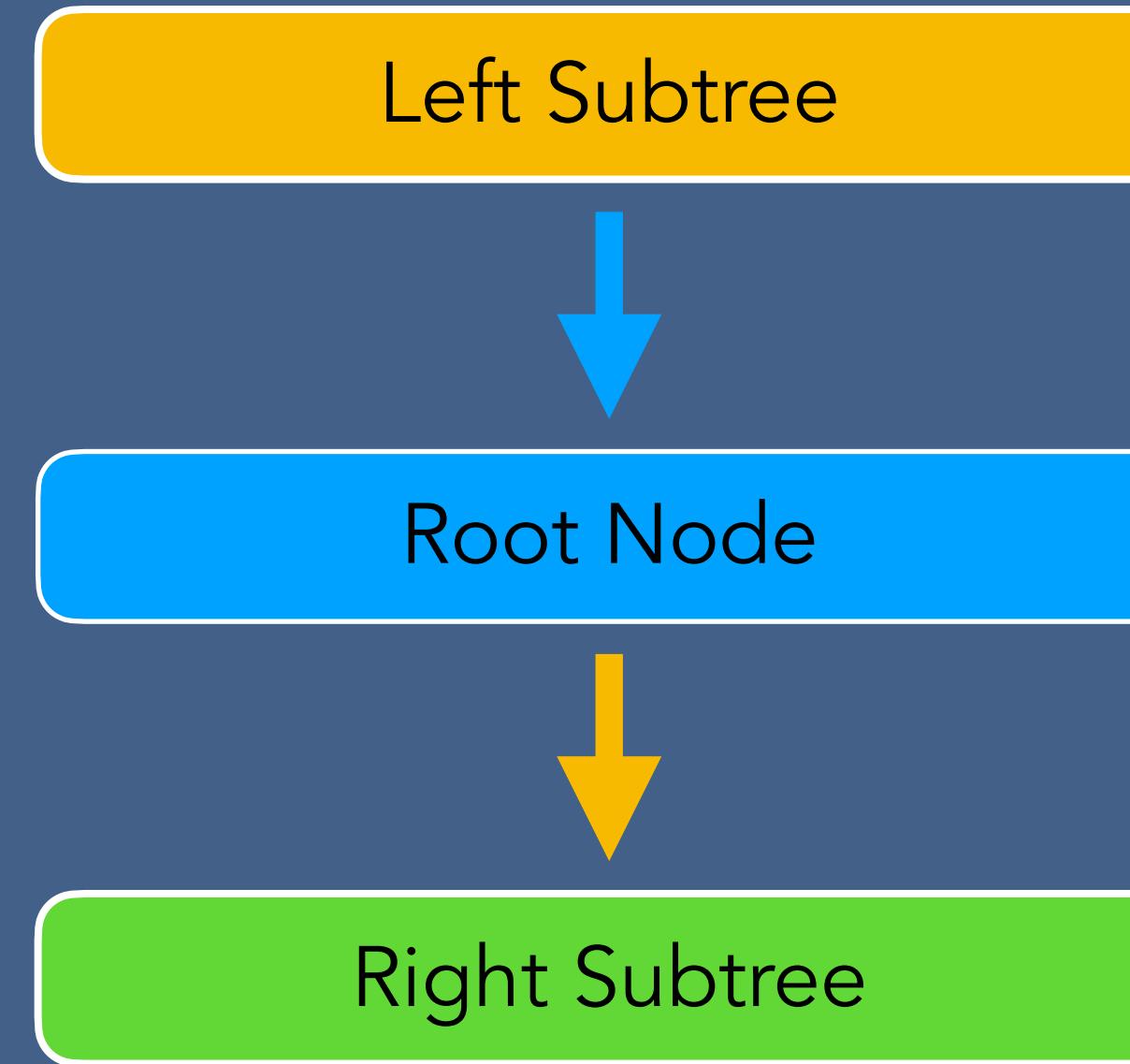
# Binary Tree - InOrder Traversal



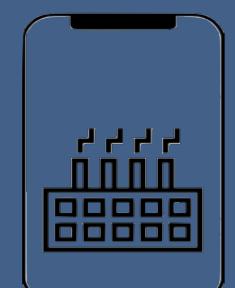
# Binary Tree - InOrder Traversal



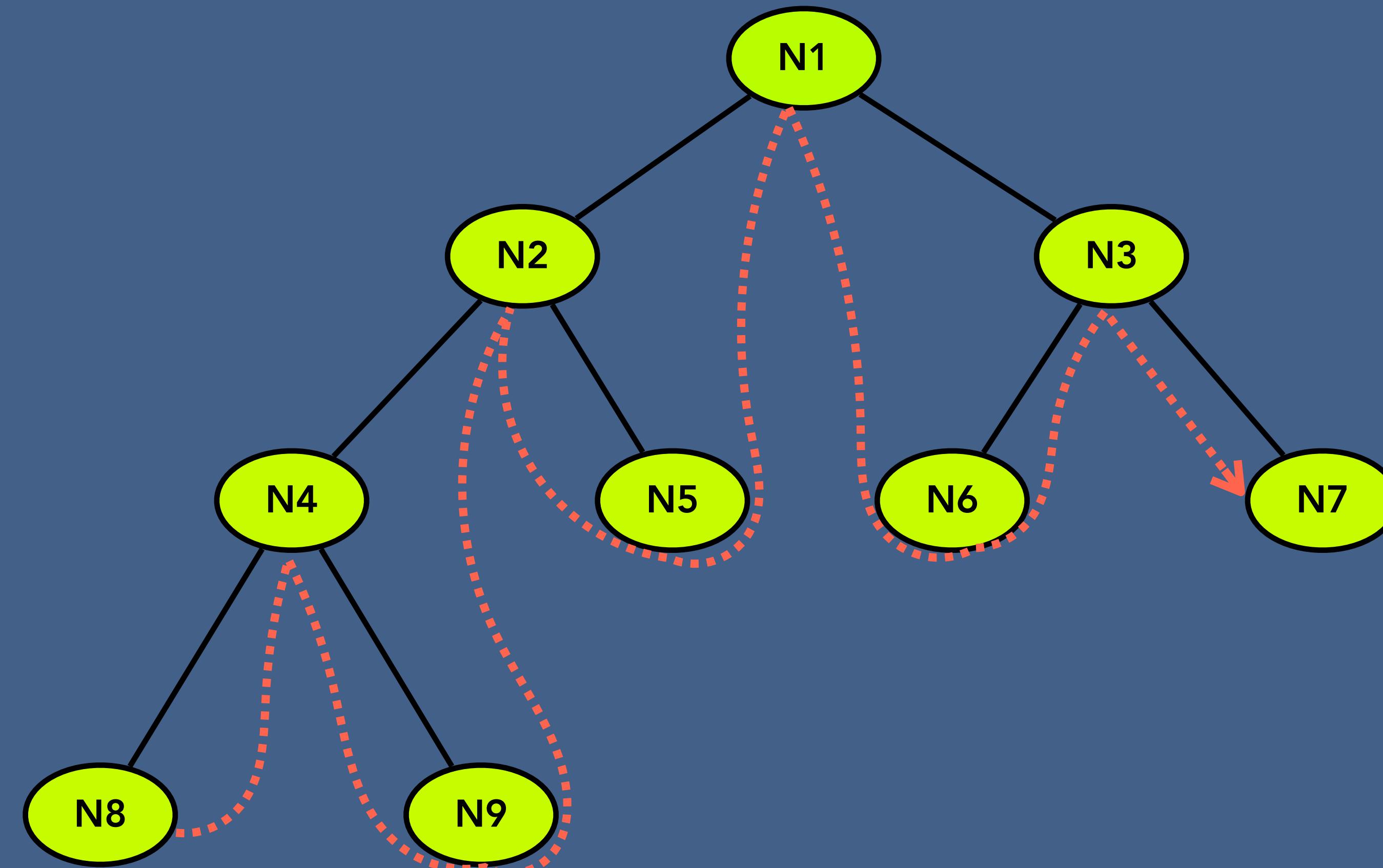
# Binary Tree - InOrder Traversal



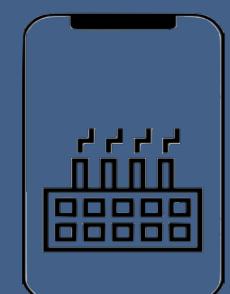
N9 → N4 → N9 → N2 → N5 → N1 → N6 → N3 → N7



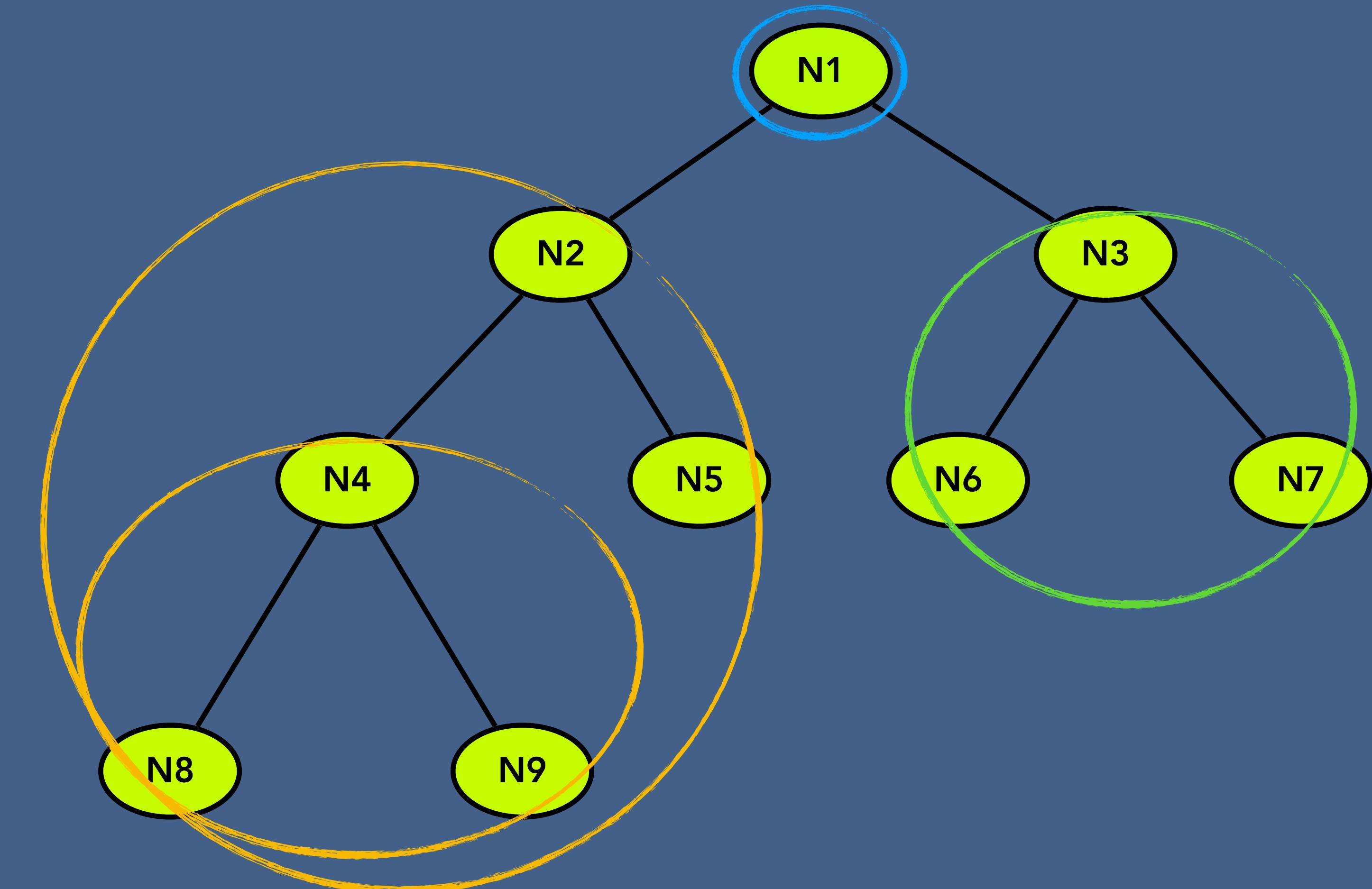
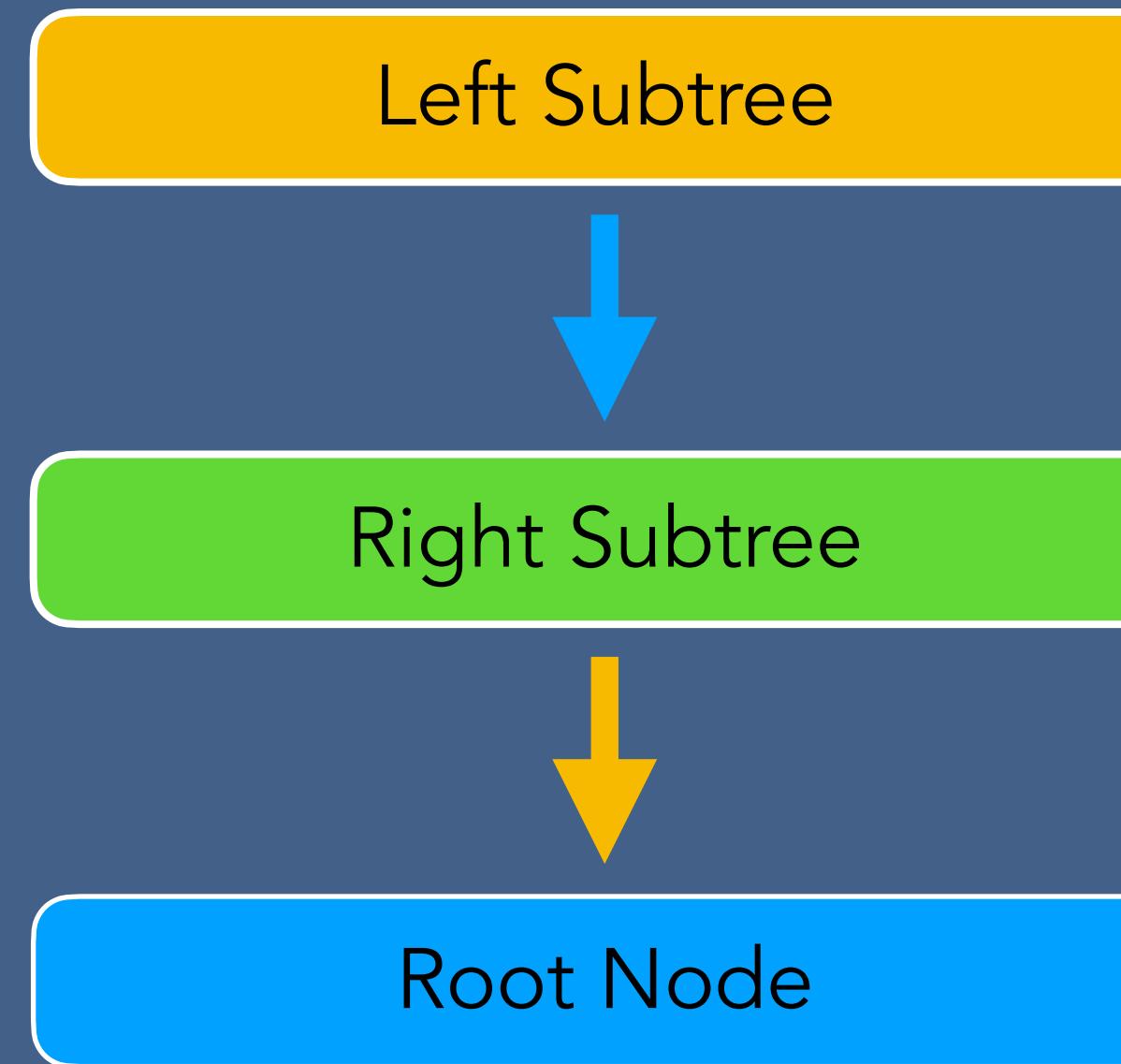
# Binary Tree - InOrder Traversal



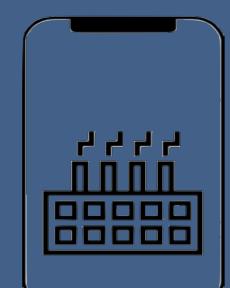
N8 → N4 → N9 → N2 → N5 → N1 → N6 → N3 → N7



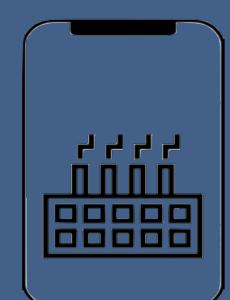
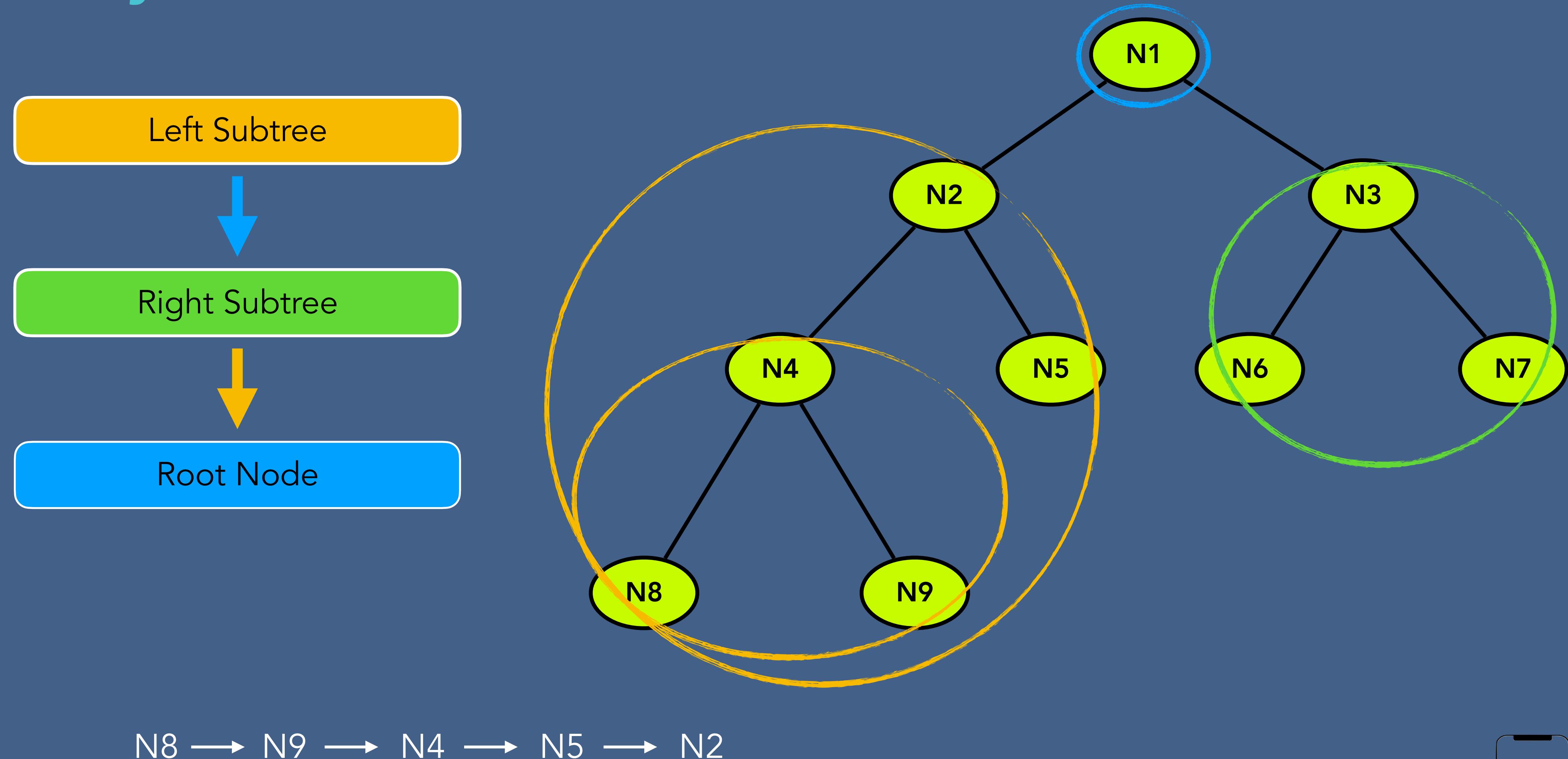
# Binary Tree - Post Traversal



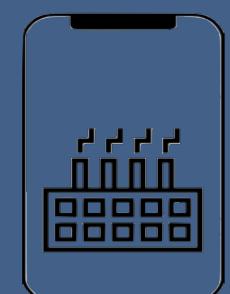
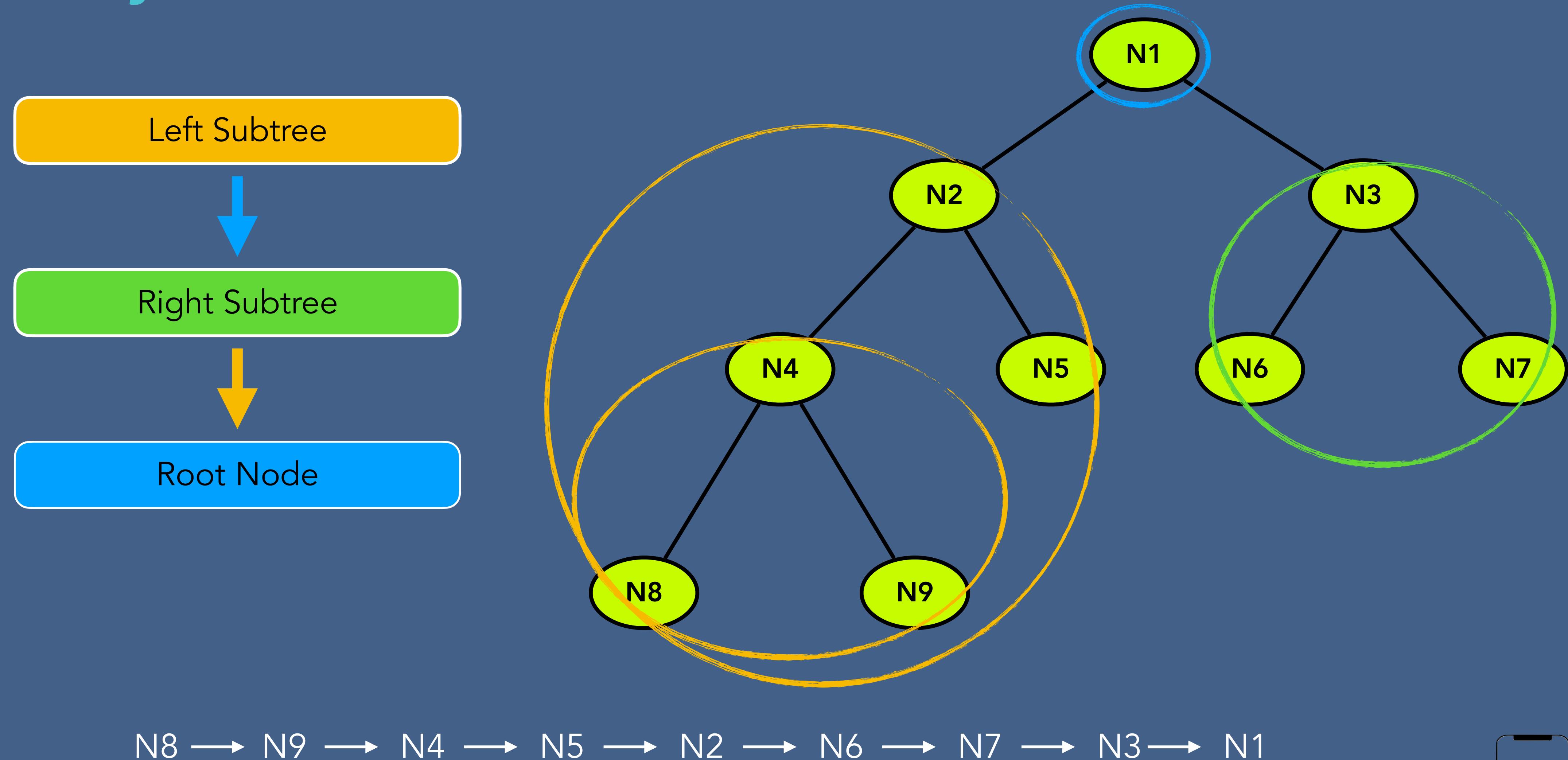
$N8 \rightarrow N9$



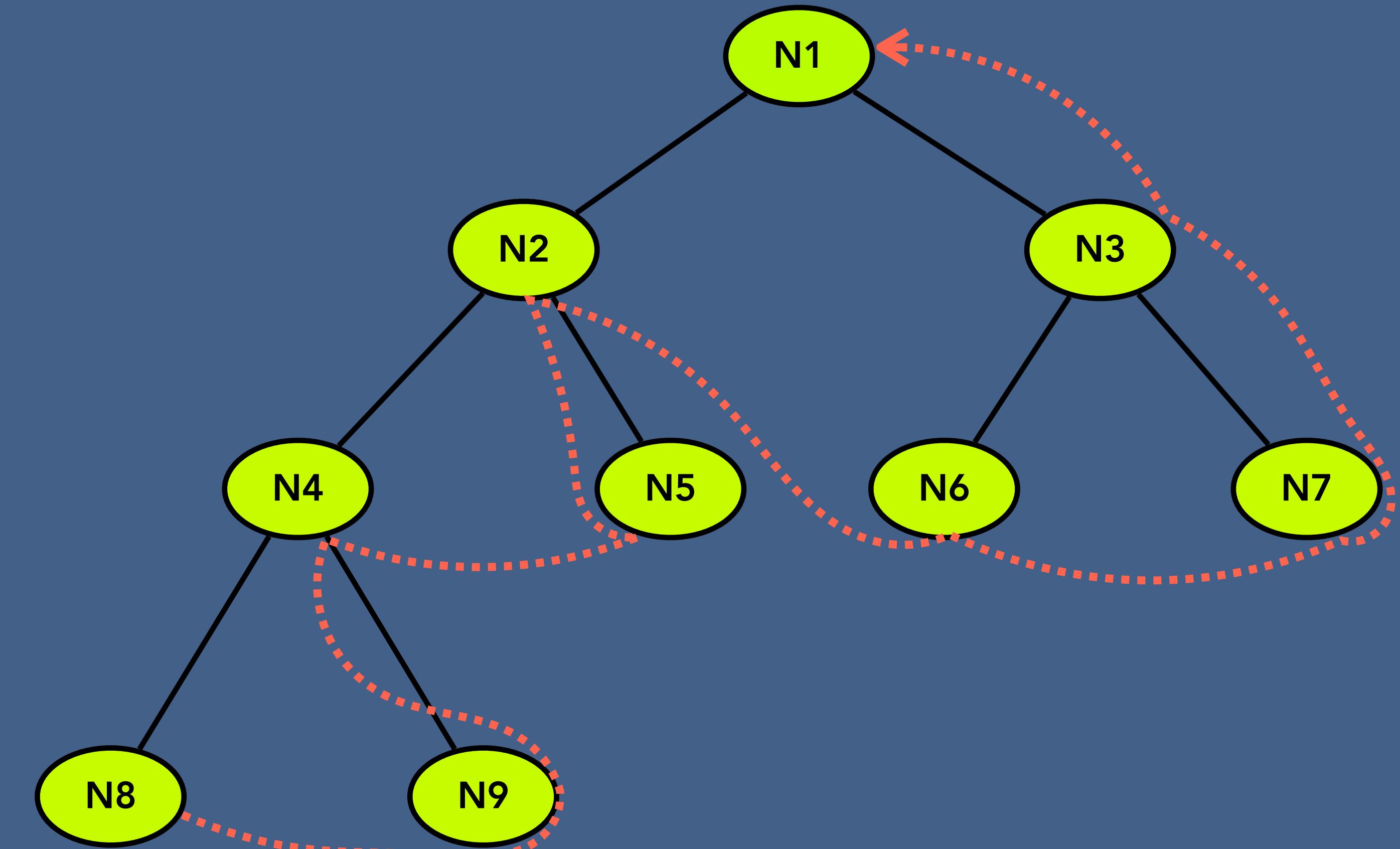
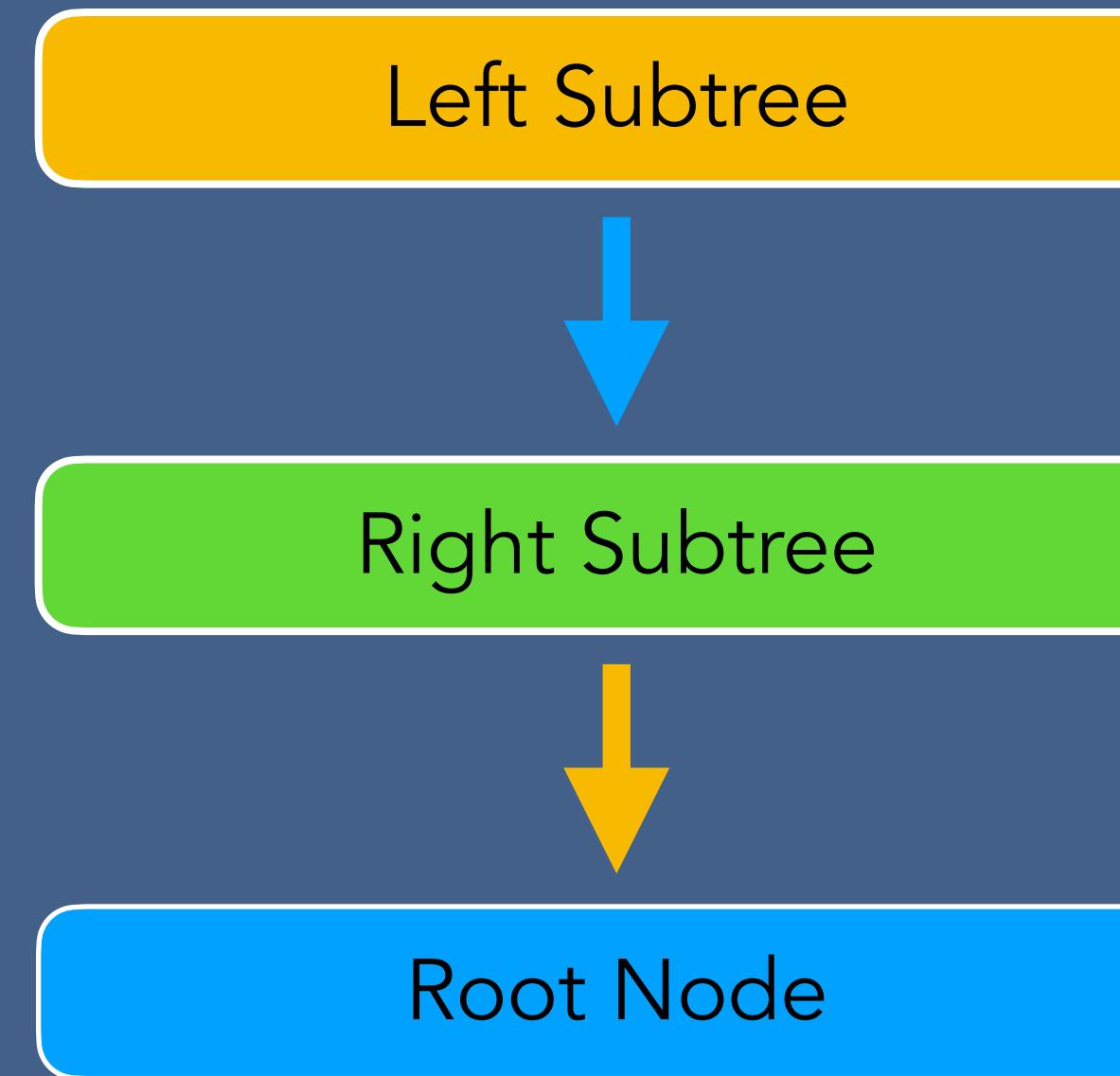
# Binary Tree - Post Traversal



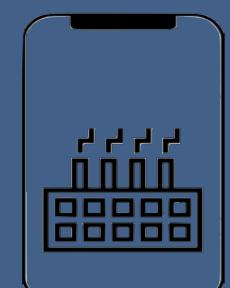
# Binary Tree - Post Traversal



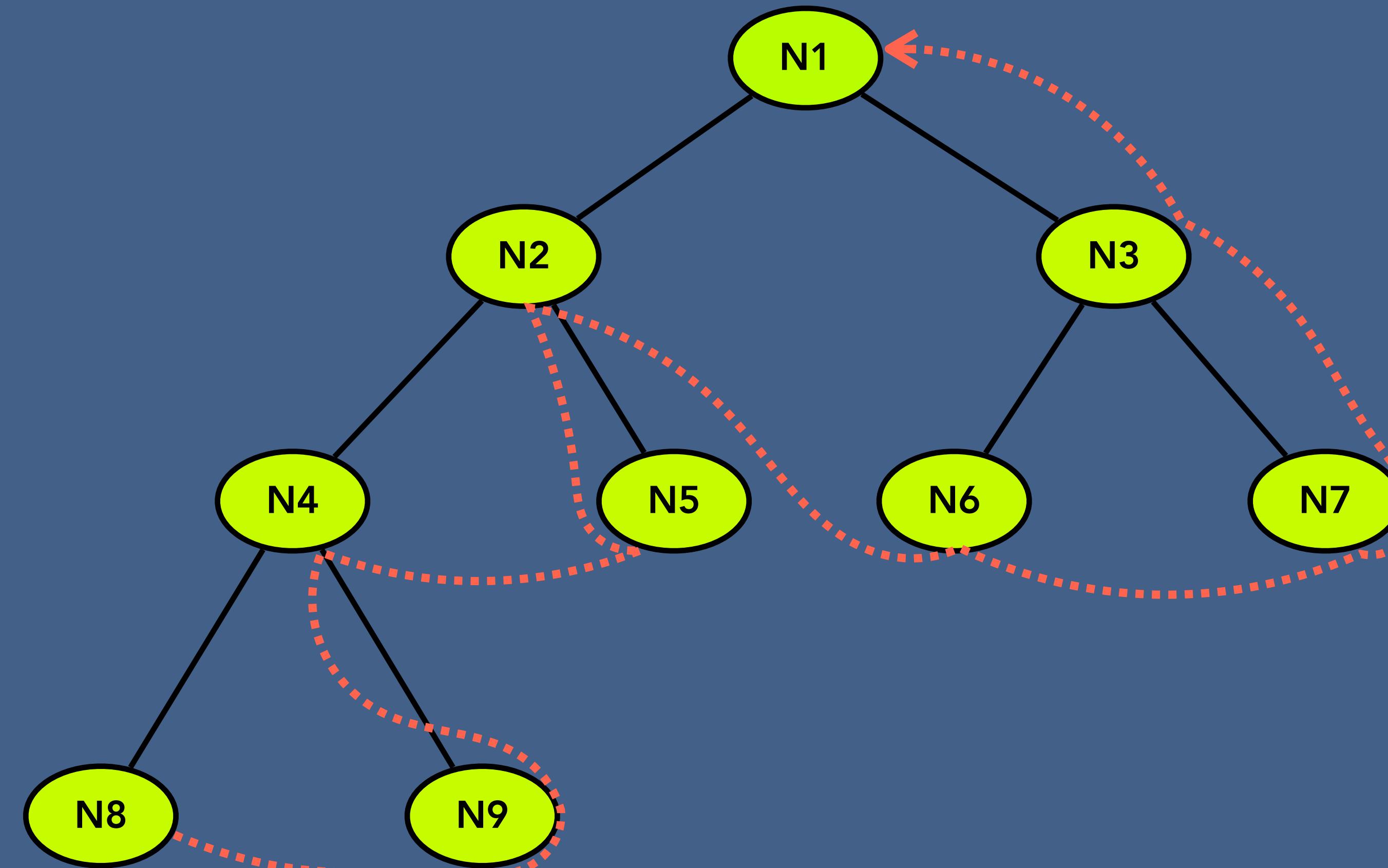
# Binary Tree - Post Traversal



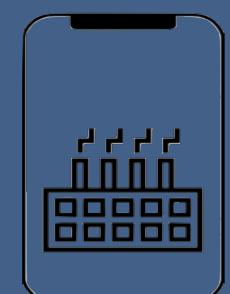
N8 → N9 → N4 → N5 → N2 → N6 → N7 → N3 → N1



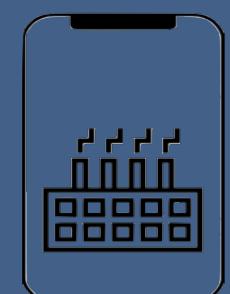
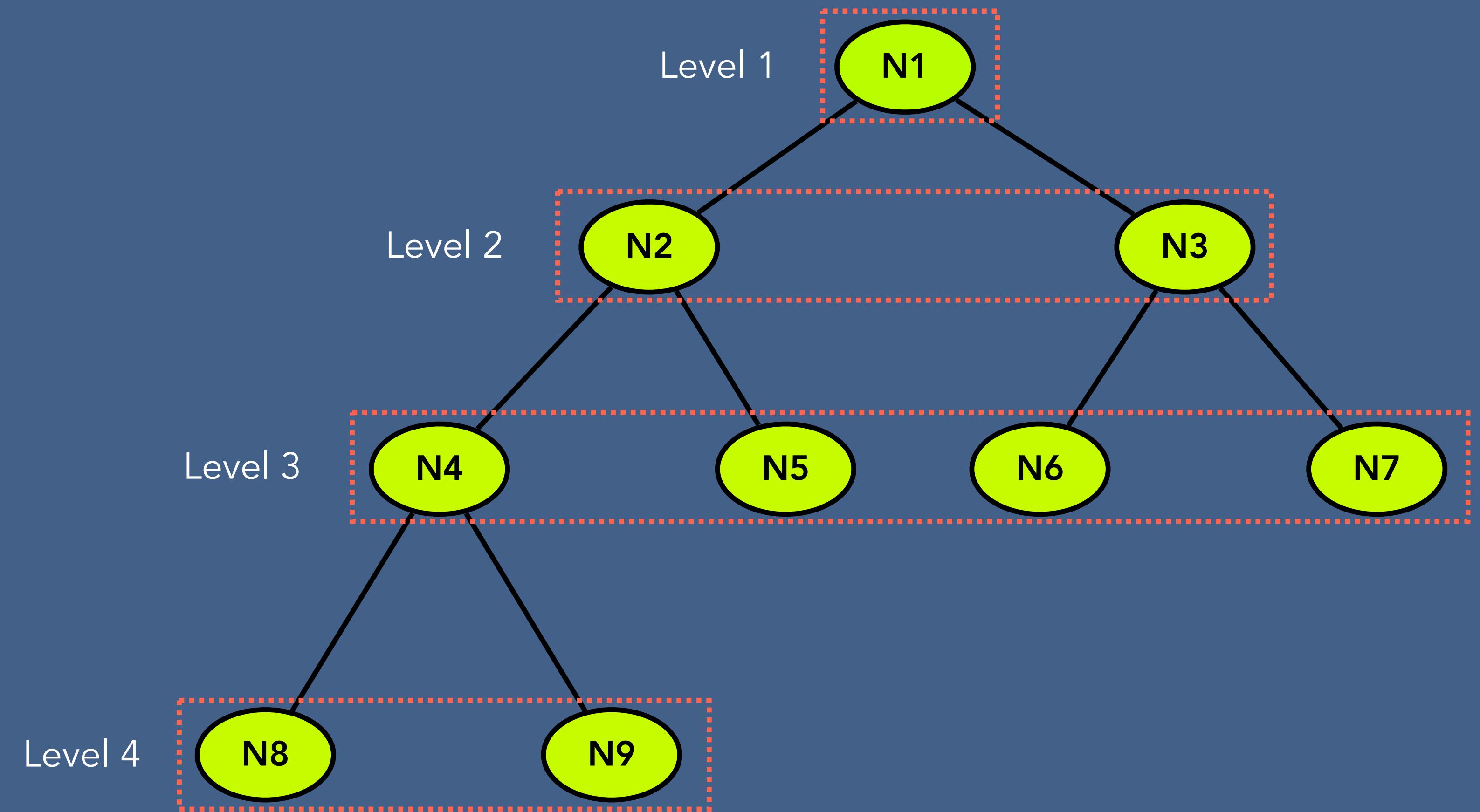
# Binary Tree - Post Traversal



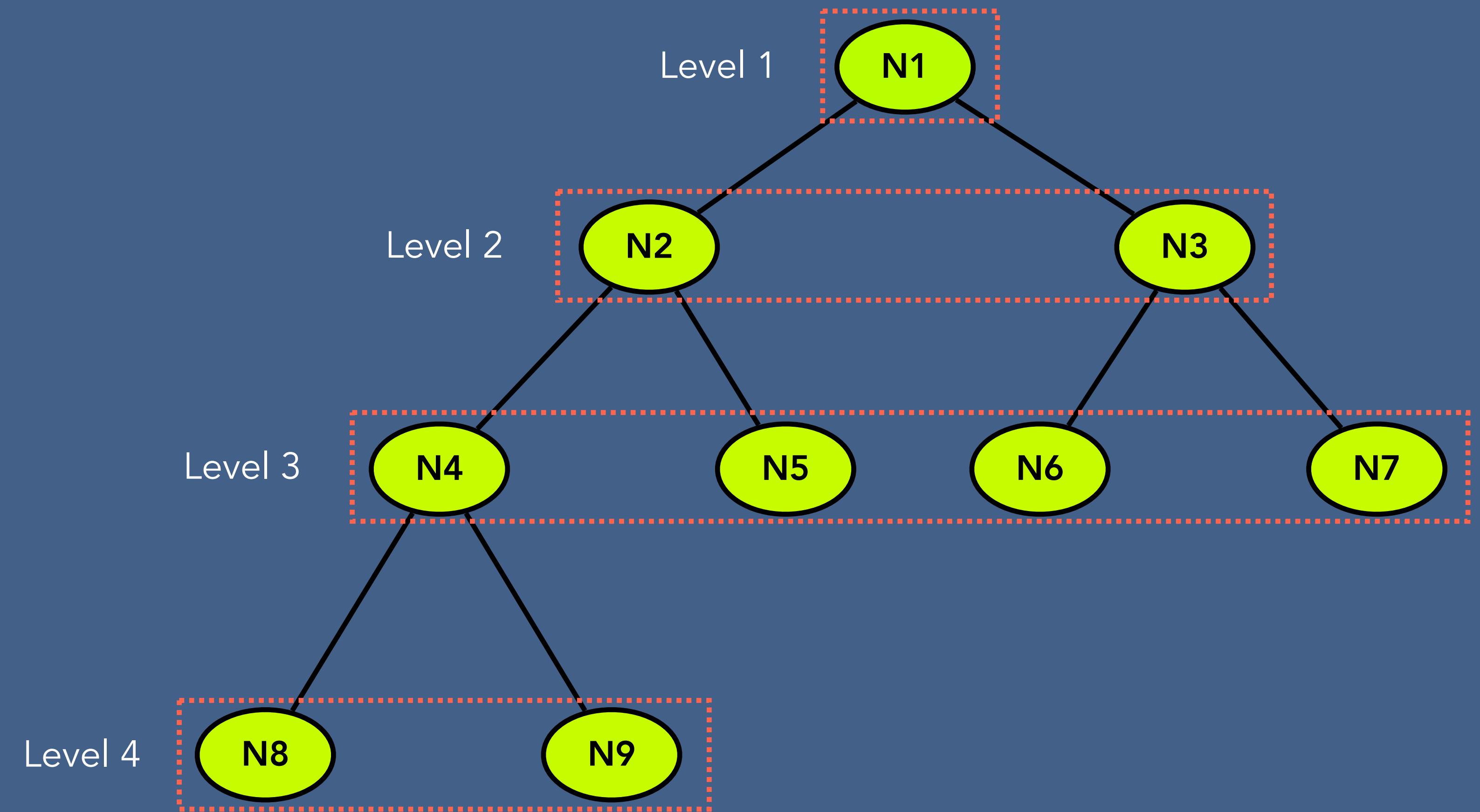
N8 → N9 → N4 → N5 → N2 → N6 → N7 → N3 → N1



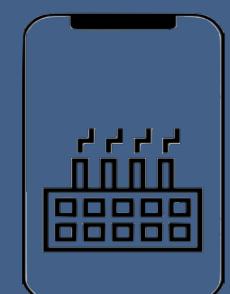
# Binary Tree - LevelOrder Traversal



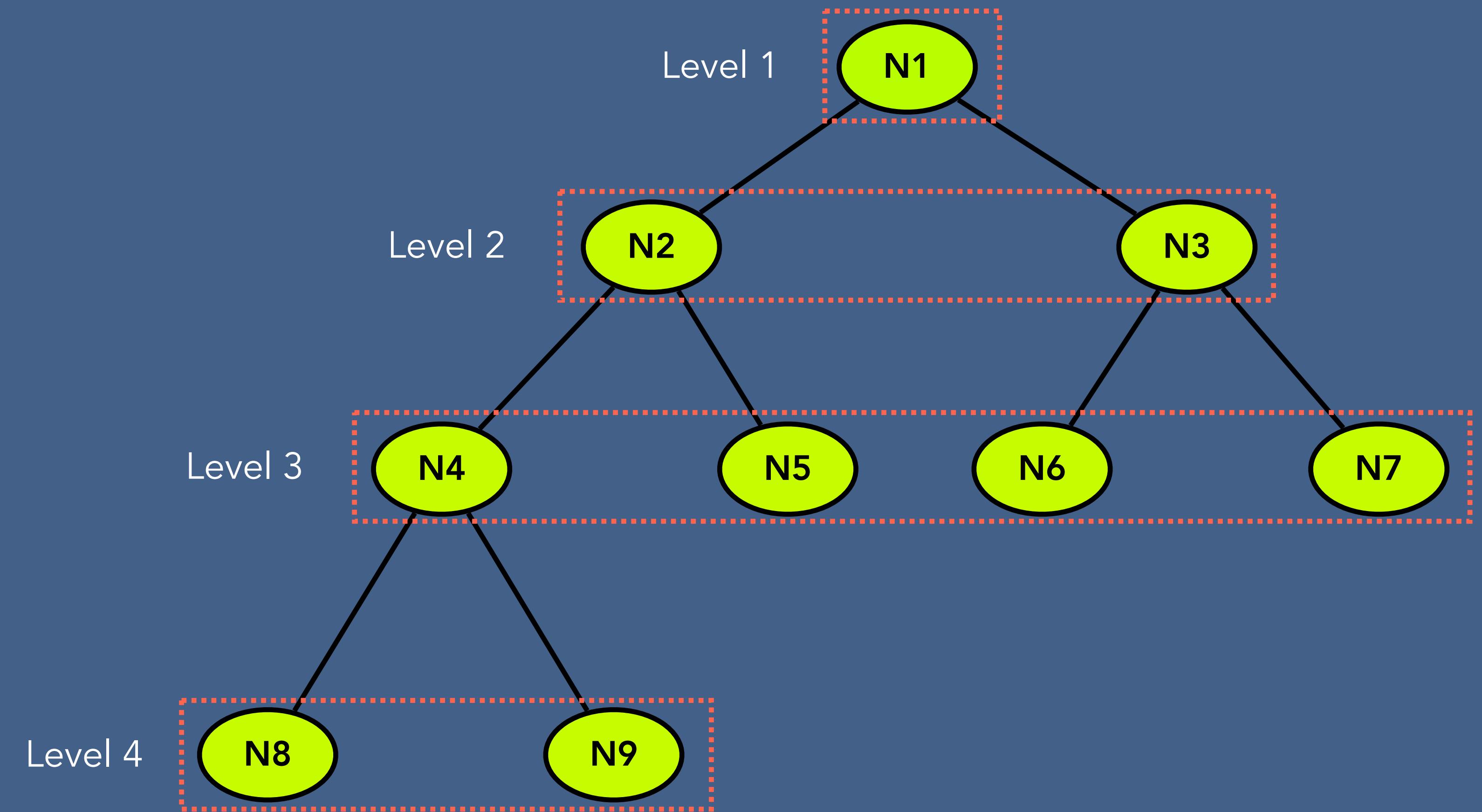
# Binary Tree - LevelOrder Traversal



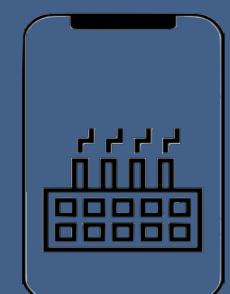
N1 → N2 → N3 → N4 → N5



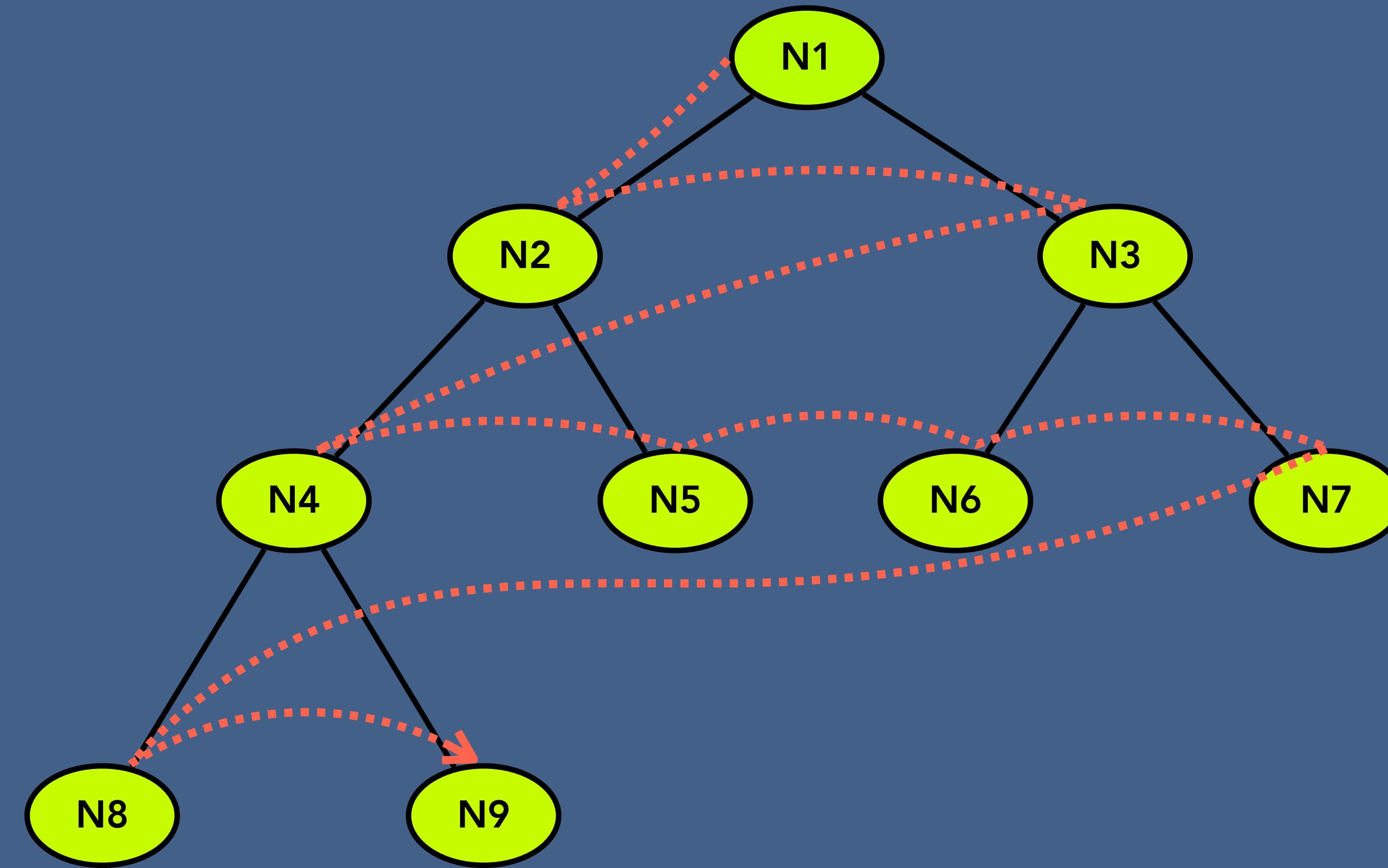
# Binary Tree - LevelOrder Traversal



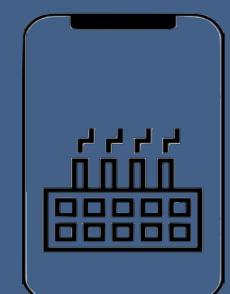
N1 → N2 → N3 → N4 → N5 → N6 → N7 → N8 → N9



# Binary Tree - LevelOrder Traversal



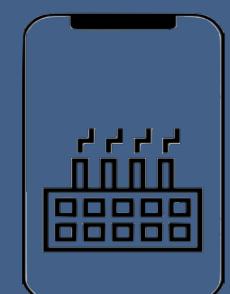
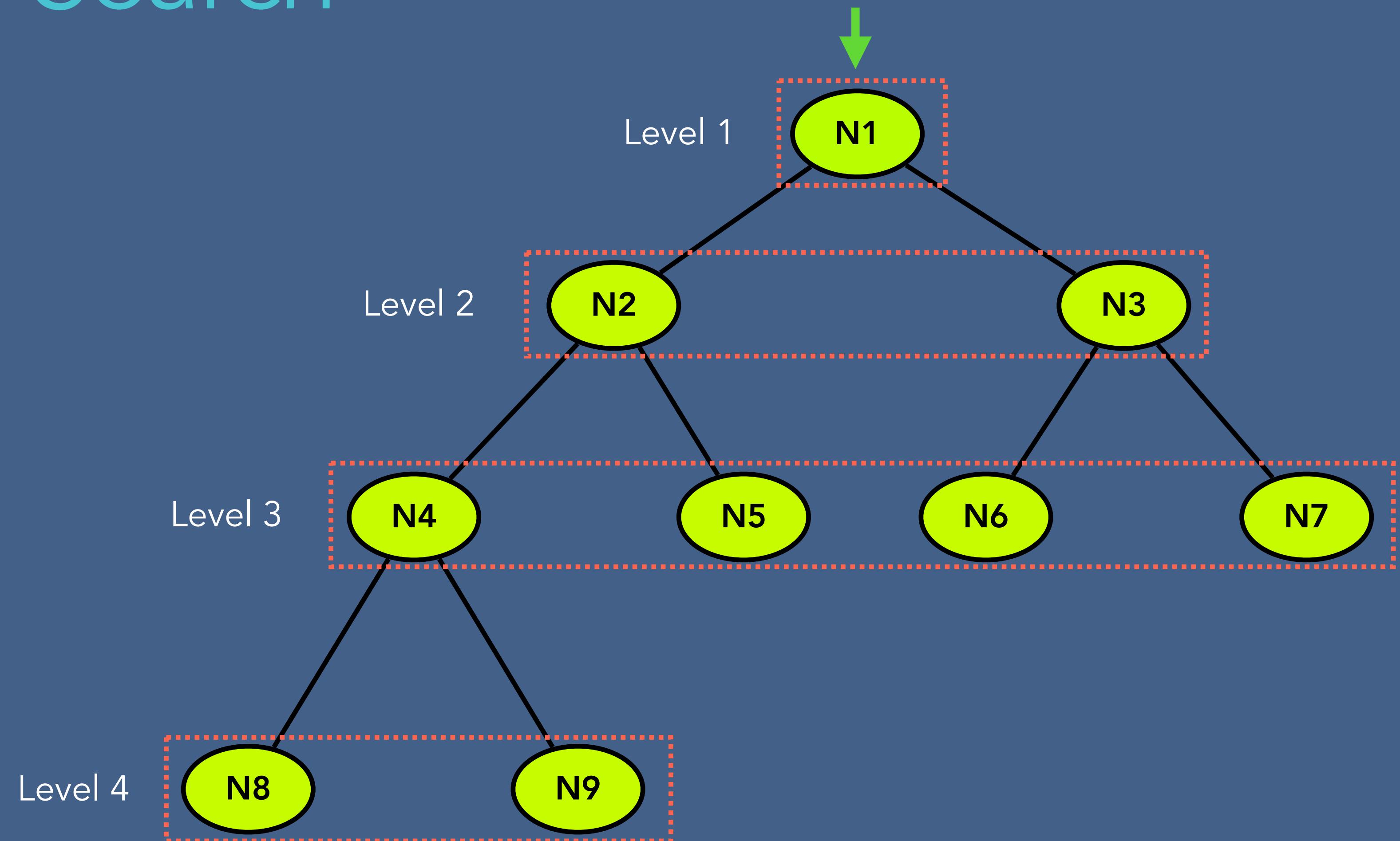
N1 → N2 → N3 → N4 → N5 → N6 → N7 → N8 → N9



# Binary Tree - Search

Level Order Traversal

N5 → Success

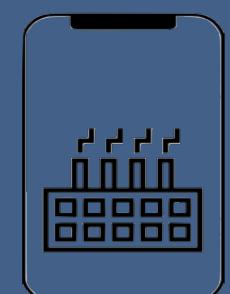
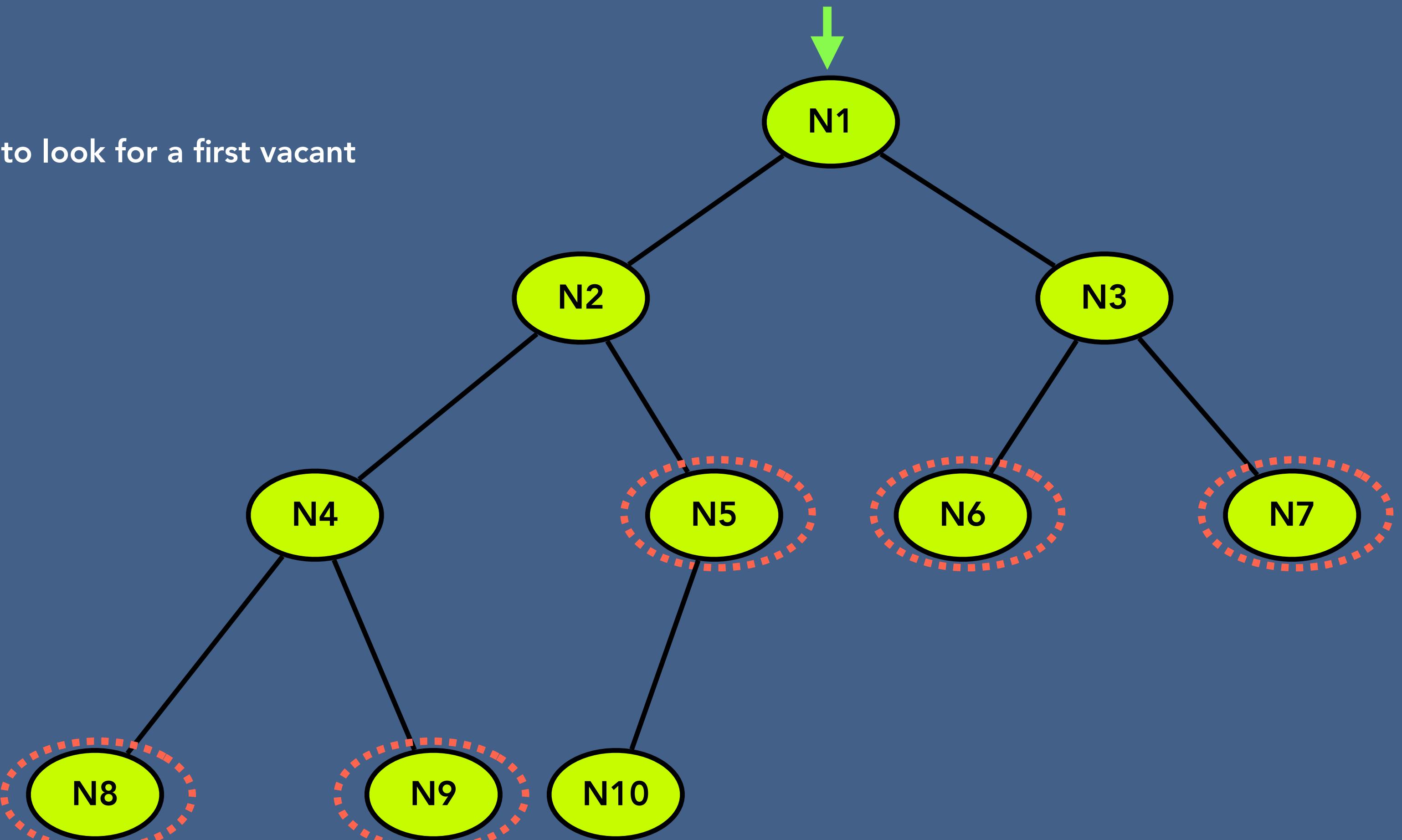


# Binary Tree - Insert a Node

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

Level Order Traversal

newNode



# Binary Tree - Delete a Node

## Level Order Traversal

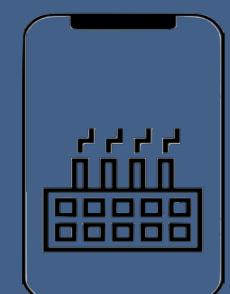
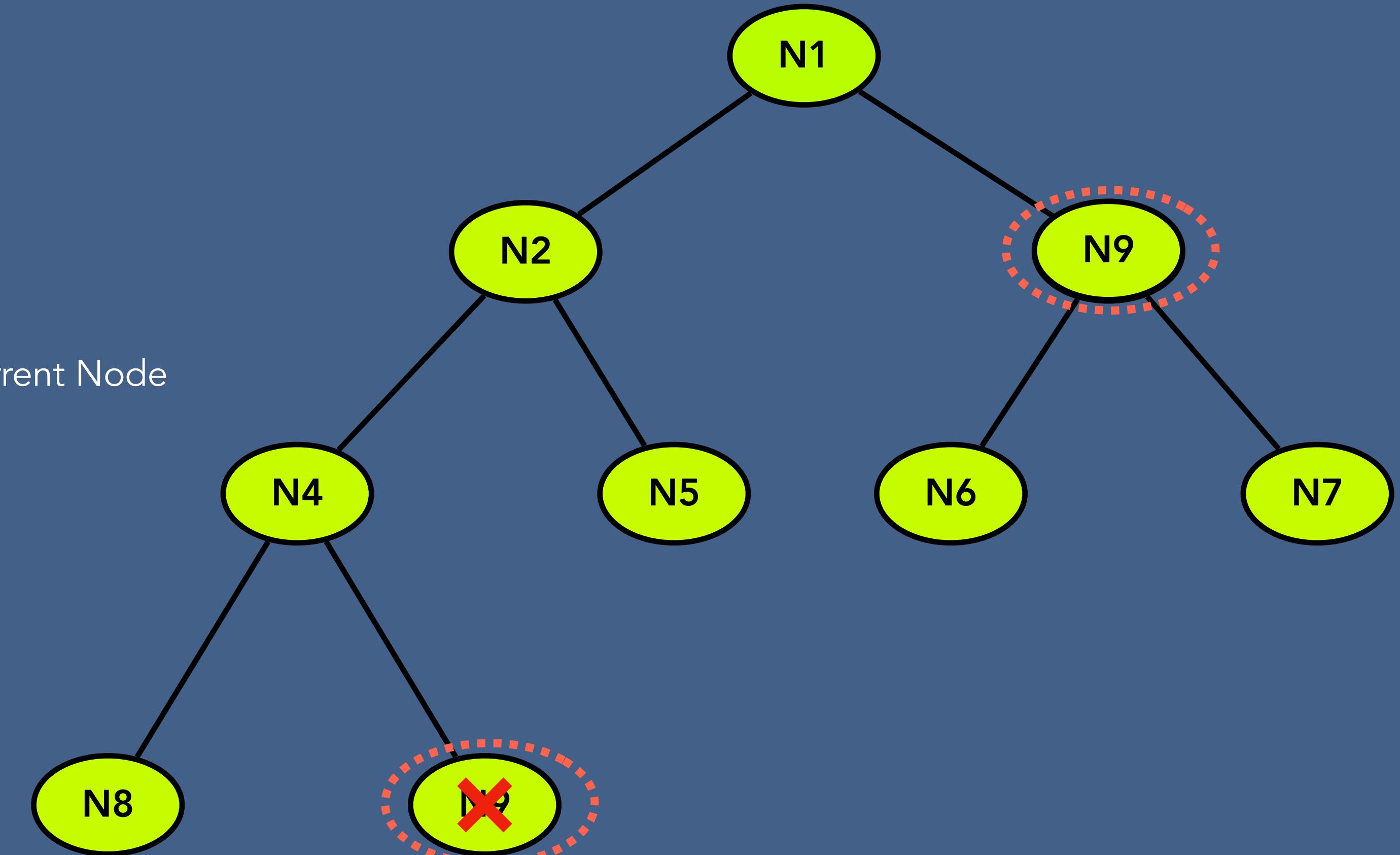
N3

Step 1 - Find the Node

Step 2 - Find Deepest Node

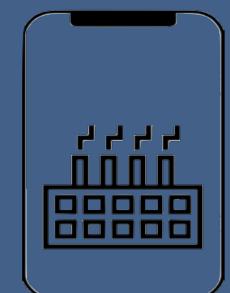
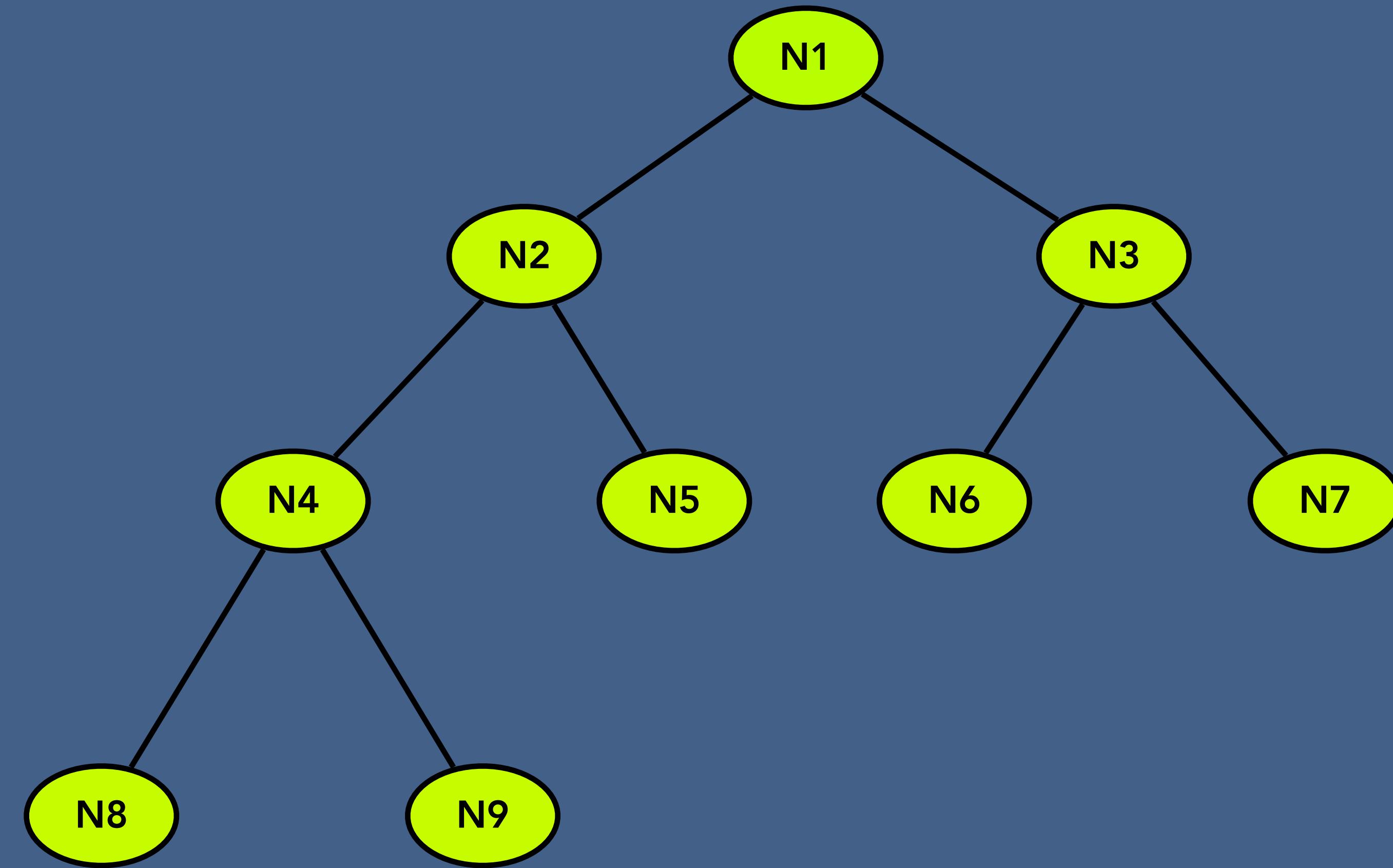
Step 3 - Set Deepest Node's value to Current Node

Step 4 - Delete Deepest Node



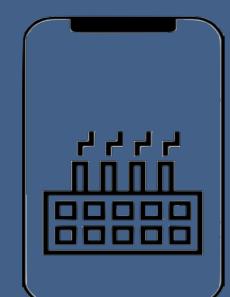
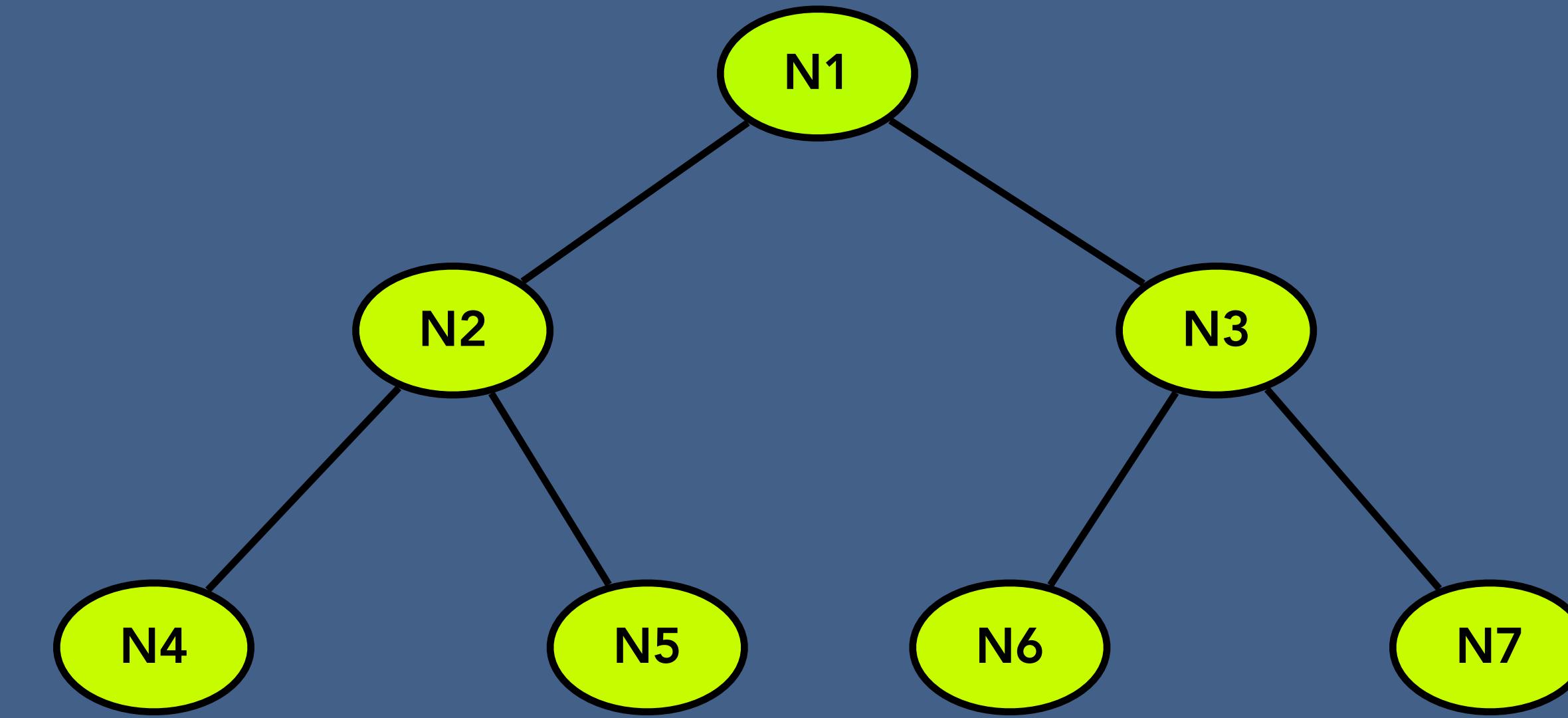
# Binary Tree - Delete Binary Tree

rootNode = Null

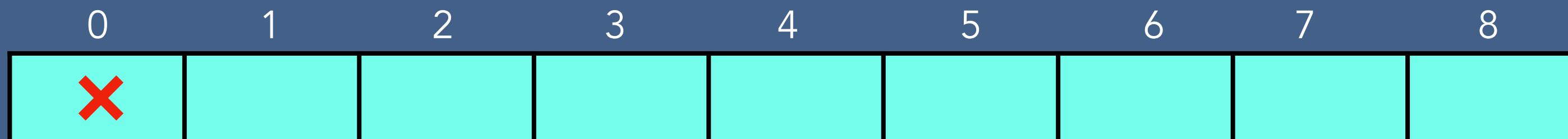
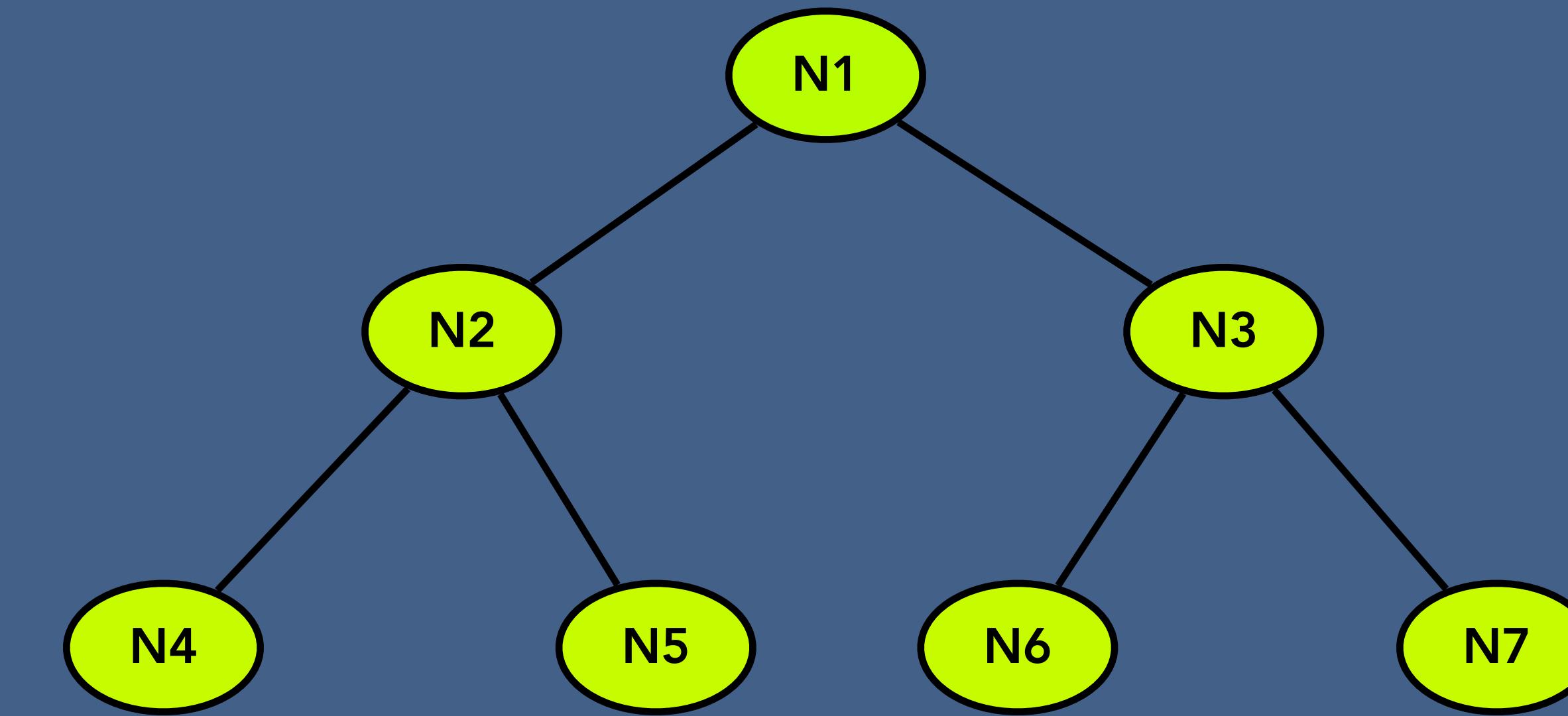


# Binary Tree using Array

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

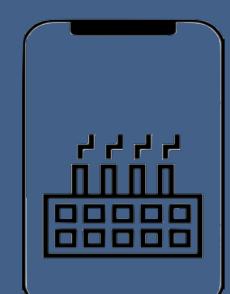


# Binary Tree using Array

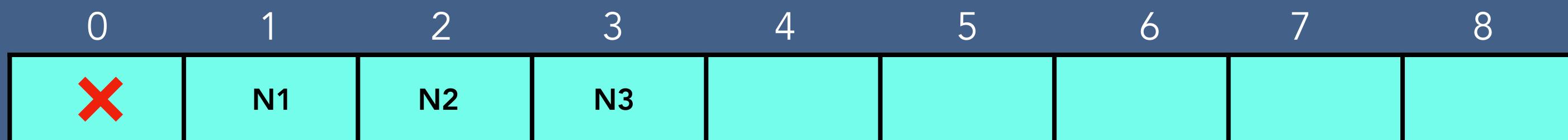
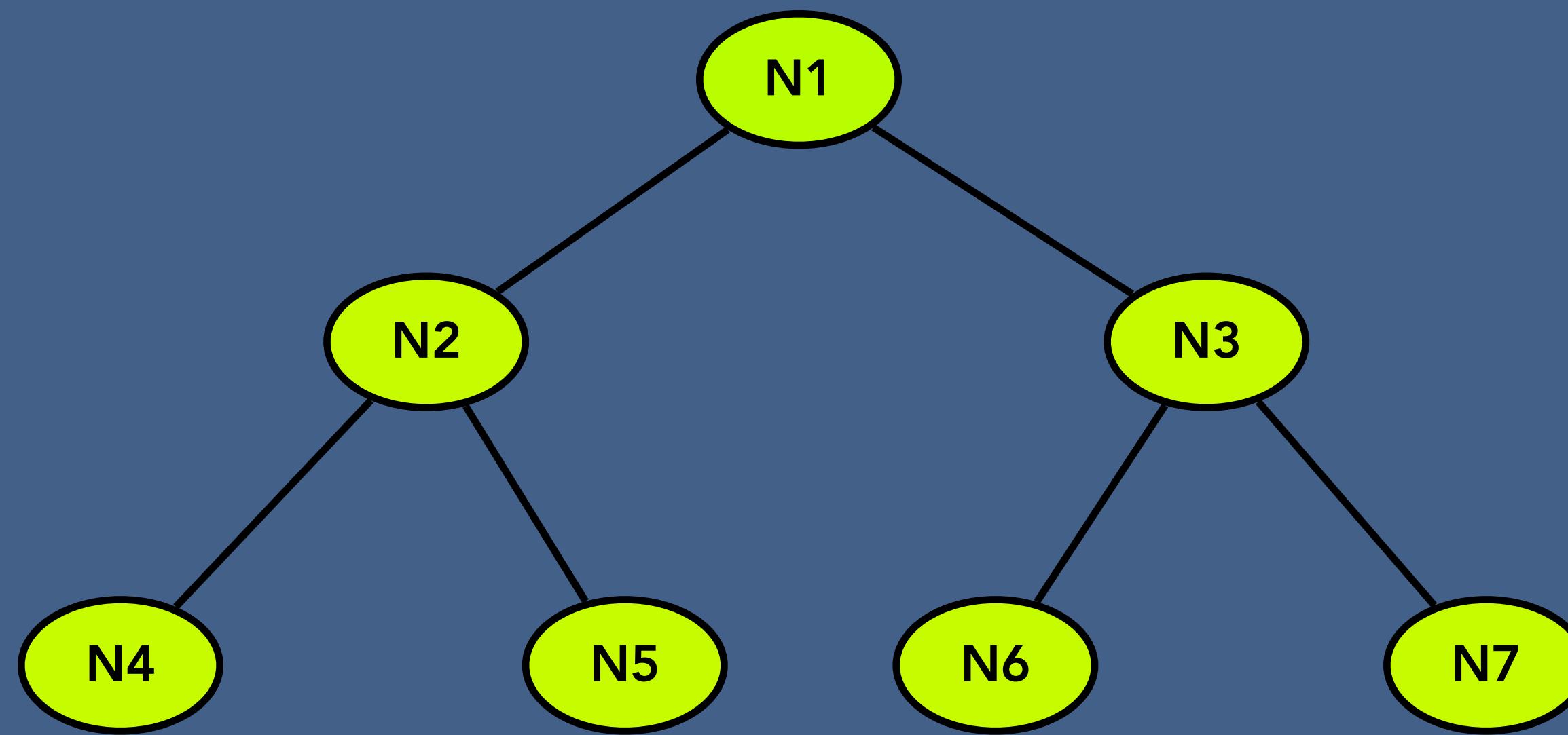


Left child =  $\text{cell}[2x]$

Right child =  $\text{cell}[2x+1]$

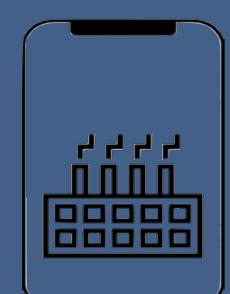


# Binary Tree using Array

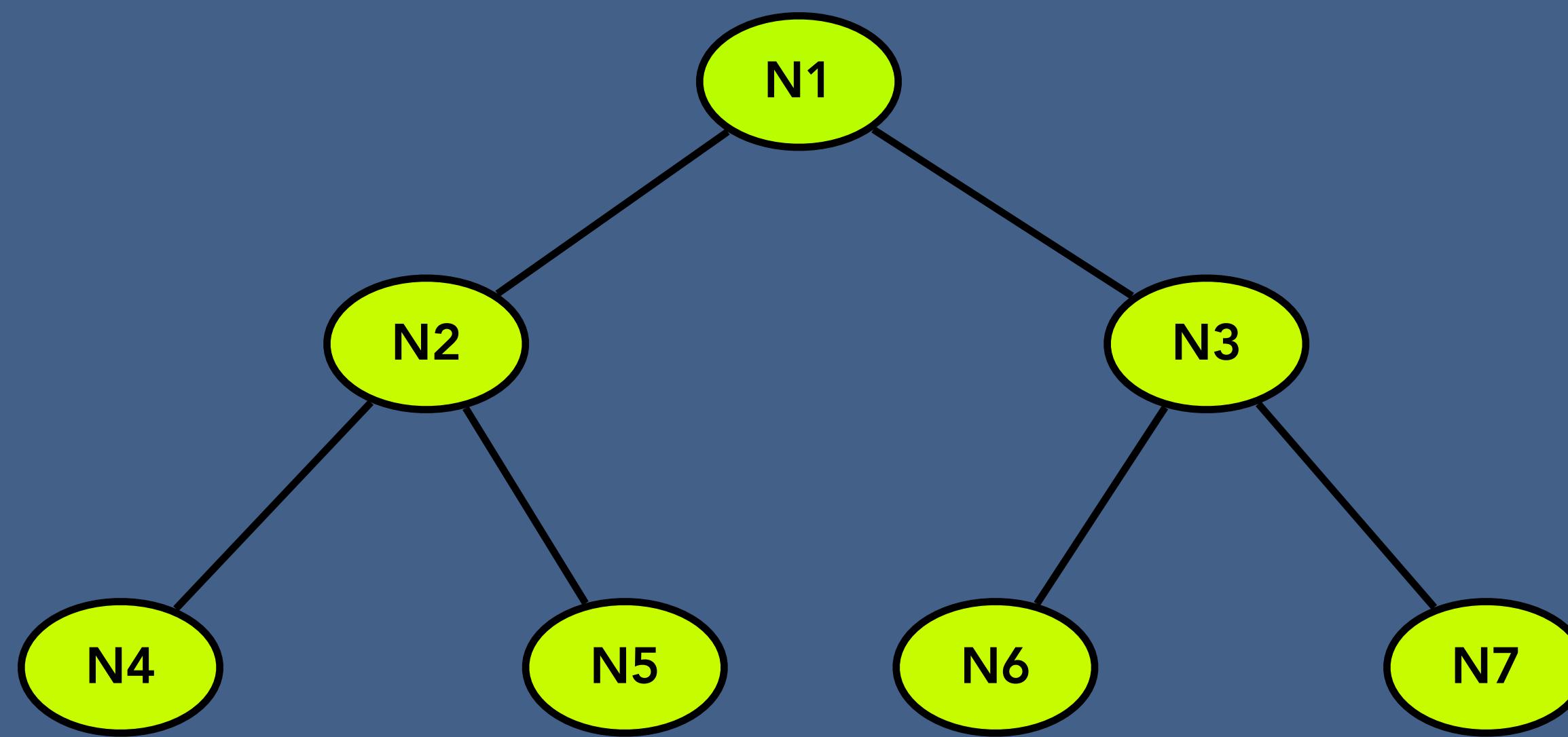


Left child =  $\text{cell}[2x]$   $\longrightarrow$   $x = 1, \text{cell}[2 \times 1 = 2]$

Right child =  $\text{cell}[2x+1]$   $\longrightarrow$   $x = 1, \text{cell}[2 \times 1 + 1 = 3]$

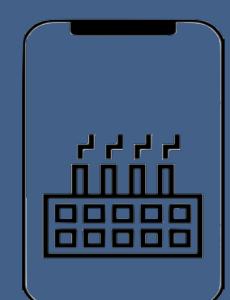


# Binary Tree using Array



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

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

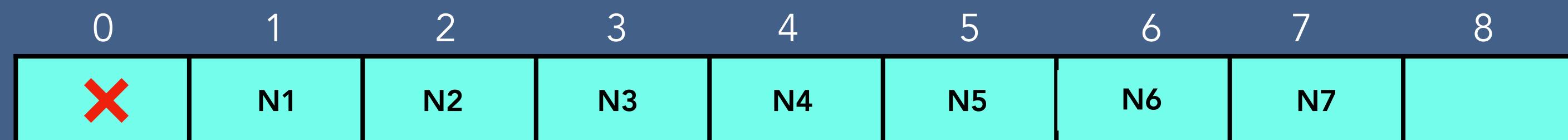
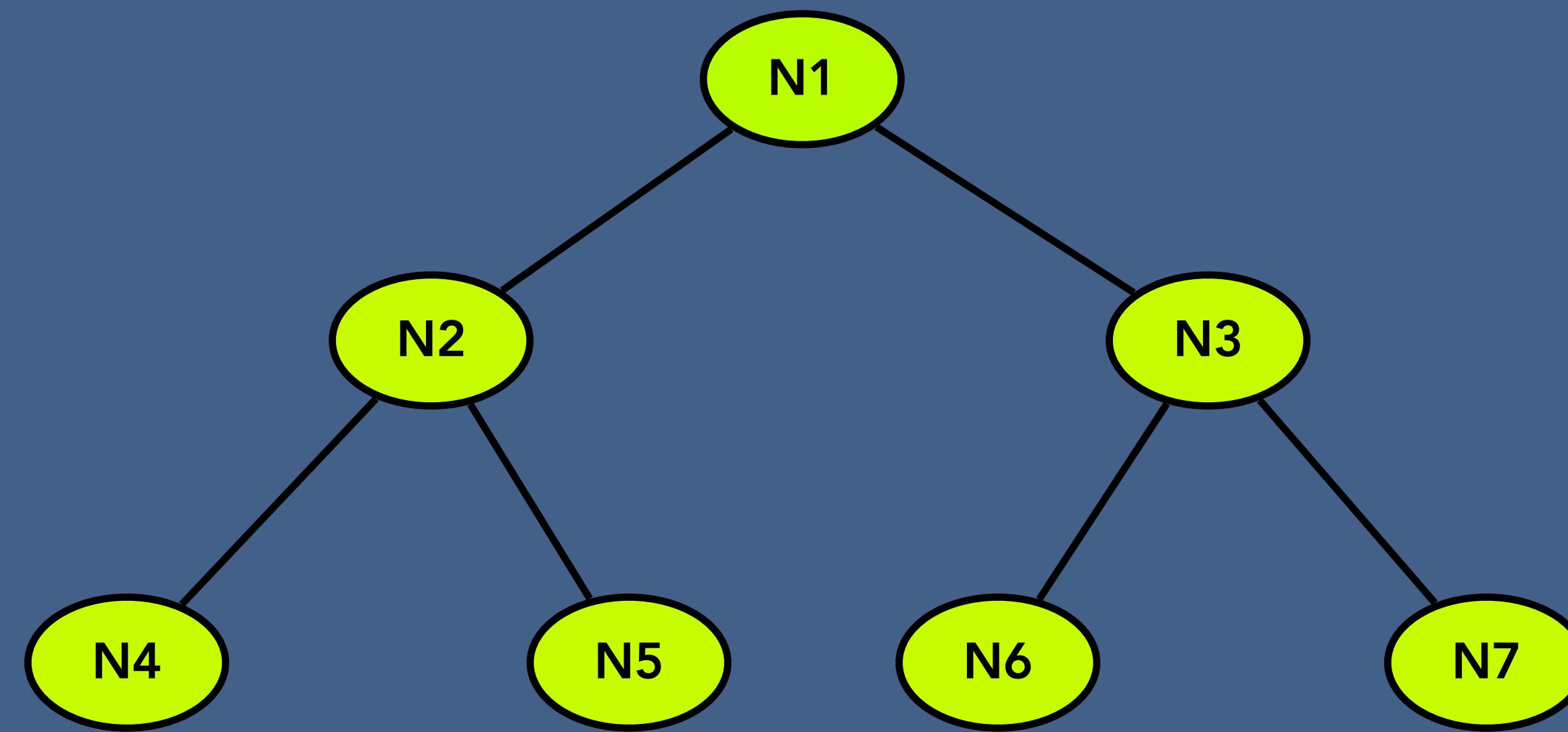


# Binary Tree using Array

**newBT = BTclass()**

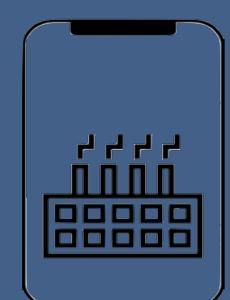
Fixed size Array

lastUsedIndex



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

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



# Binary Tree (Array) - Insert a Node

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

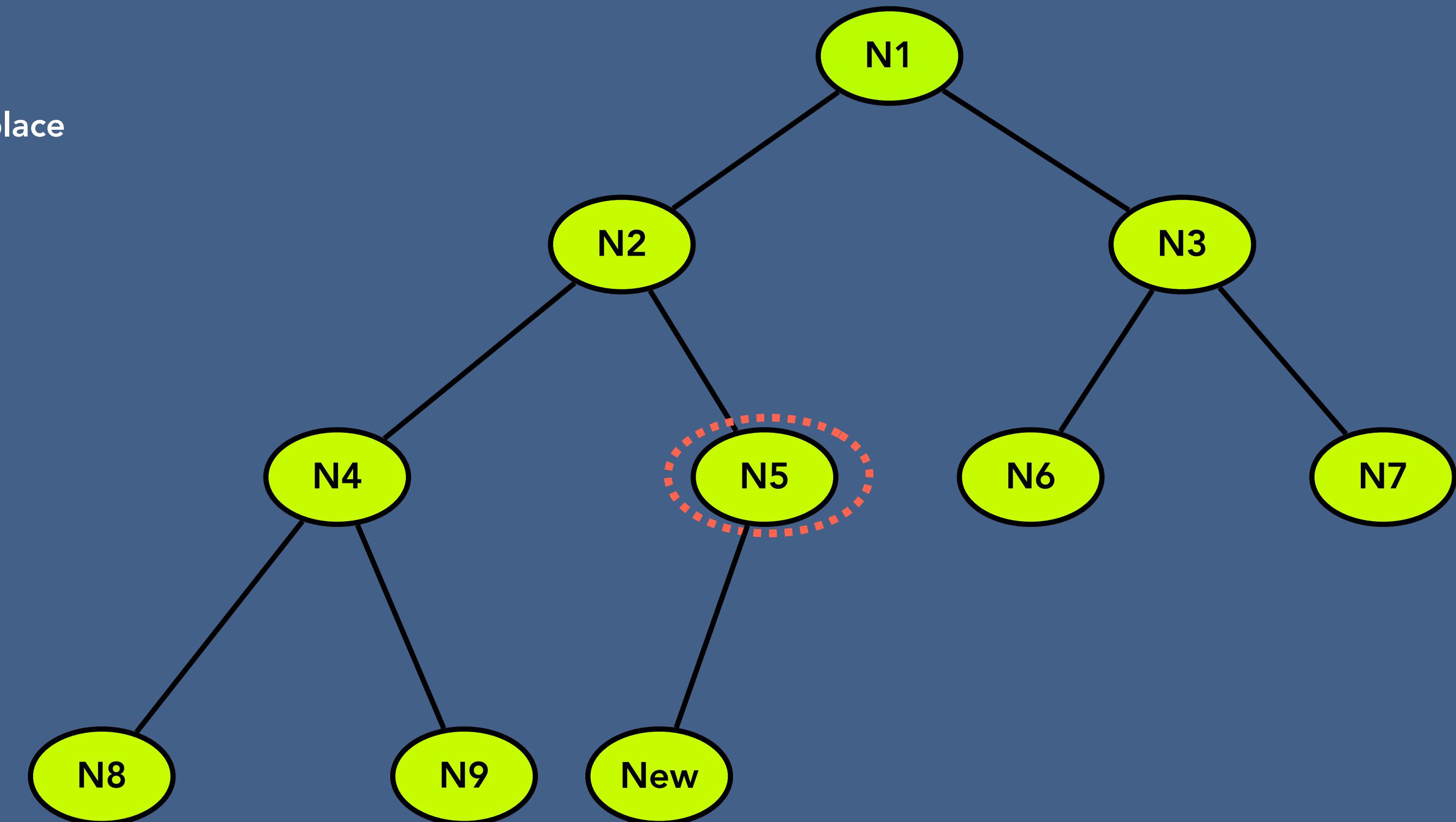
lastUsedIndex = 9

newNode

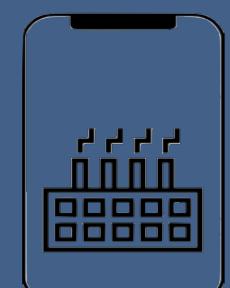
Index = 10

Left child = cell[2x]

indexOfParent =  $10/2 = 5$



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



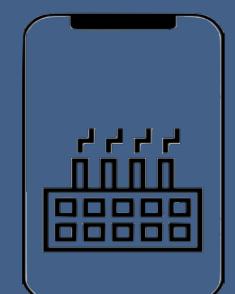
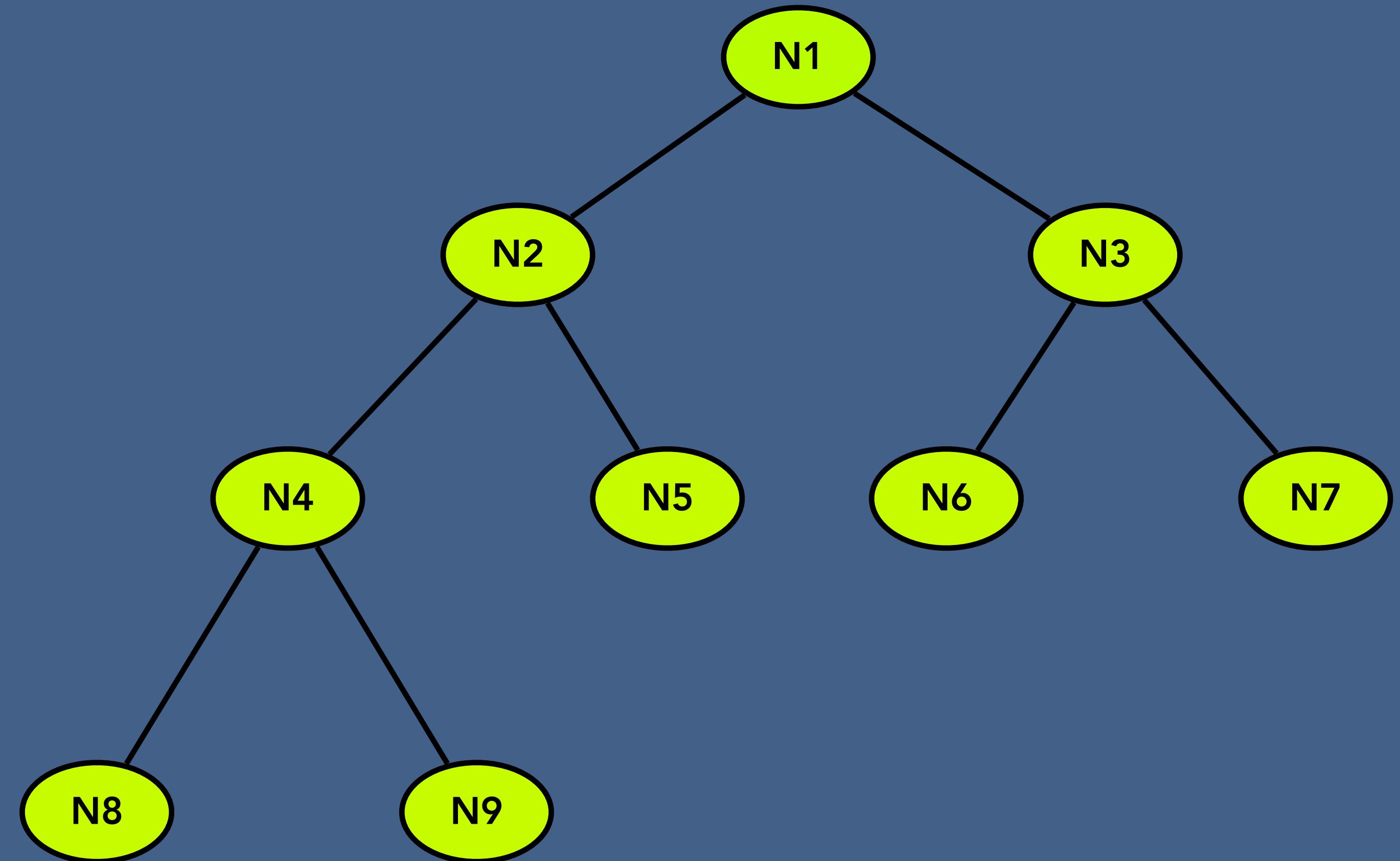
# Binary Tree (Array) - Traversal

## Depth first search

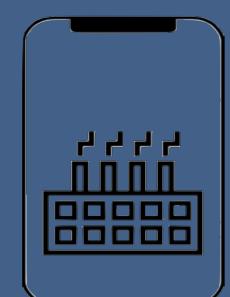
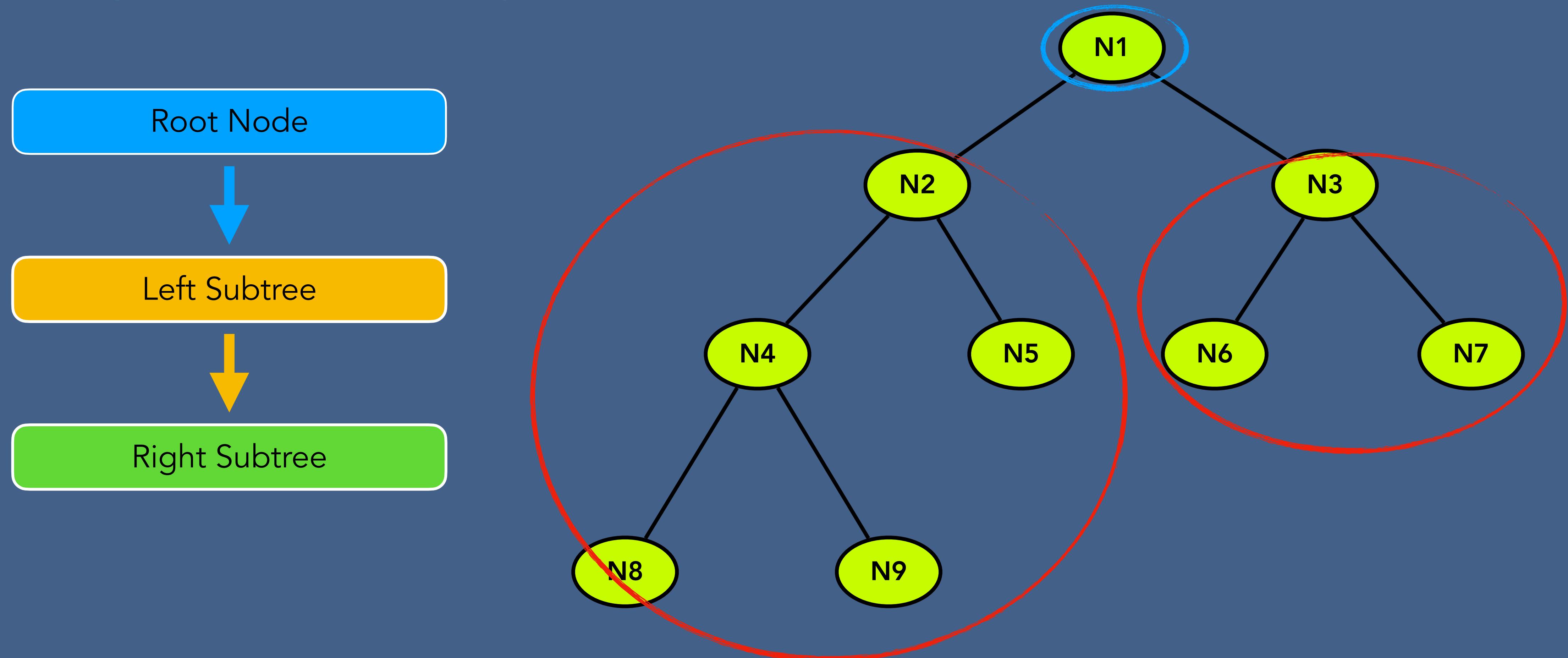
- Preorder traversal
- Inorder traversal
- Post order traversal

## Breadth first search

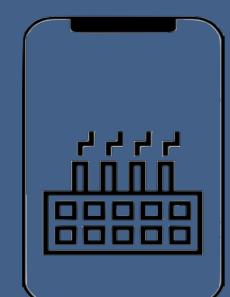
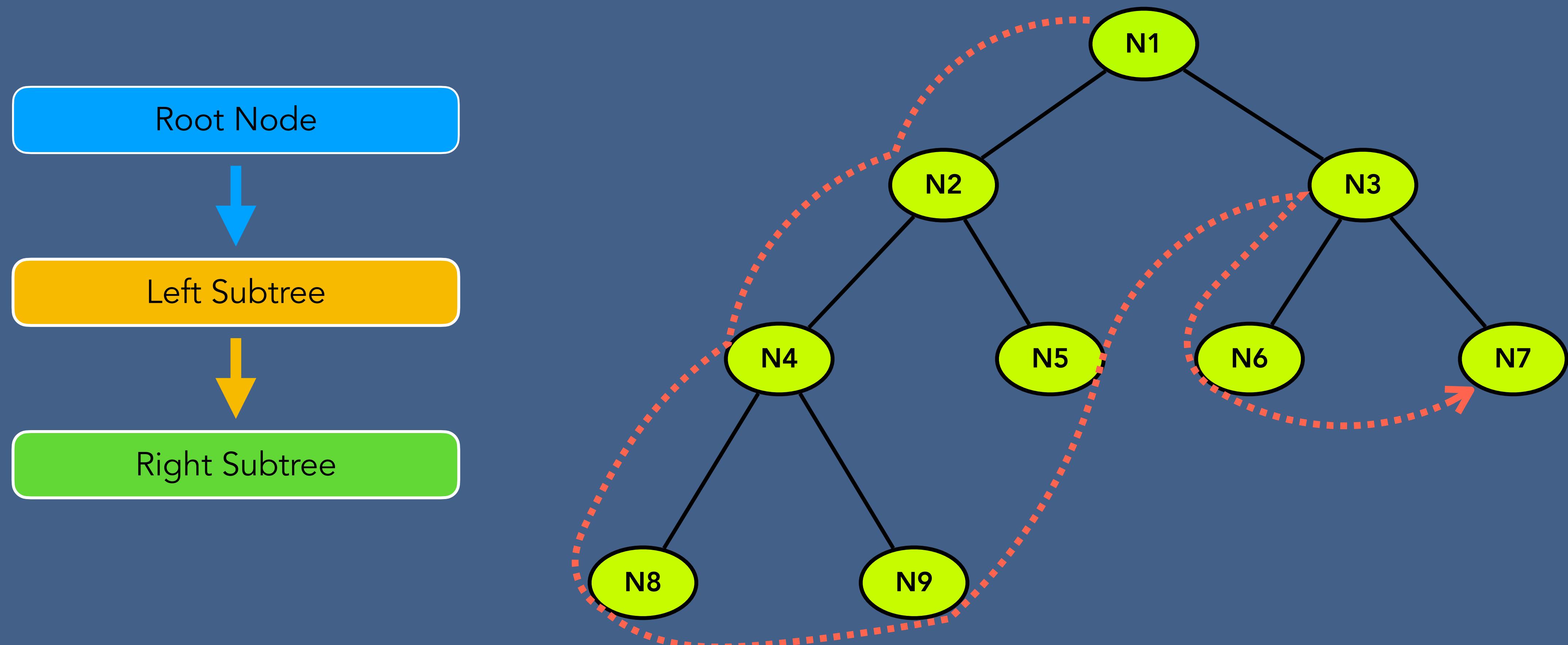
- Level order traversal



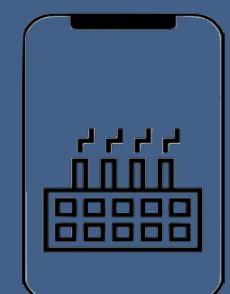
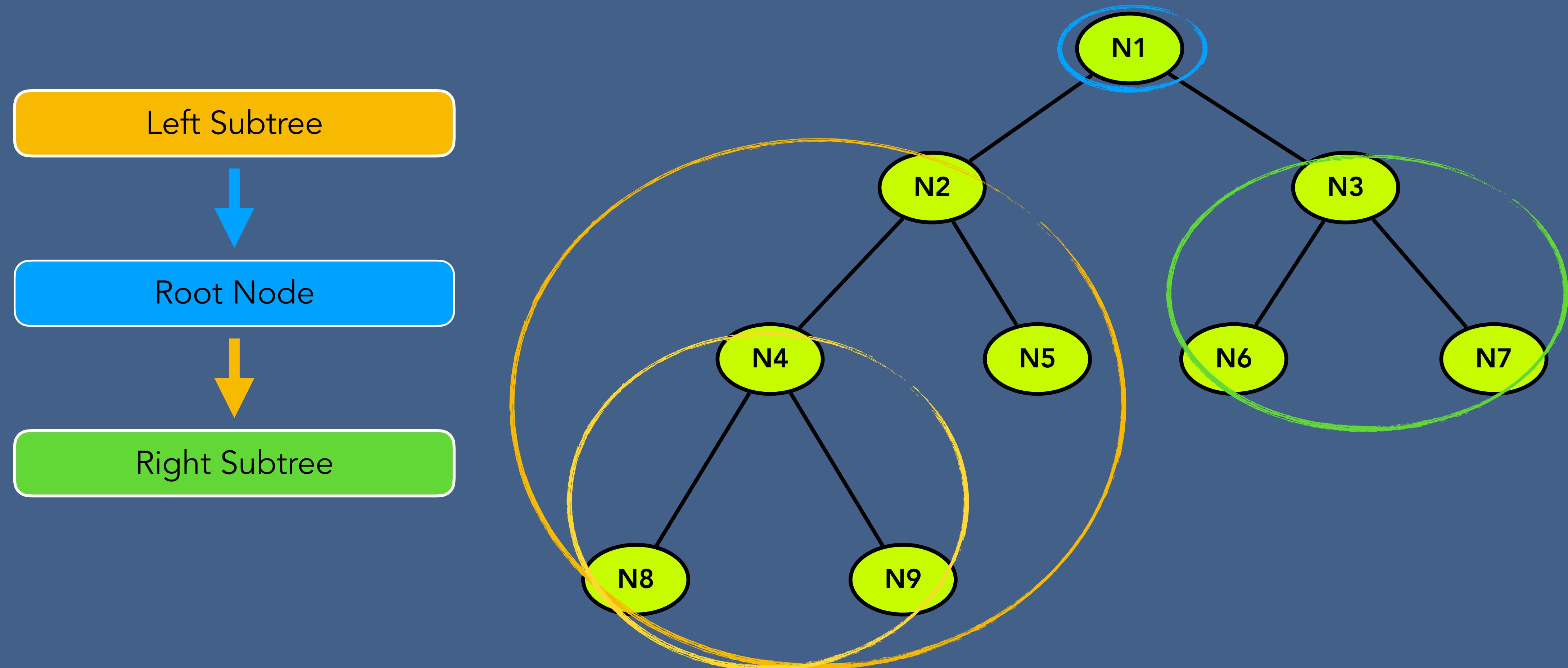
# Binary Tree (Array) - PreOrder Traversal



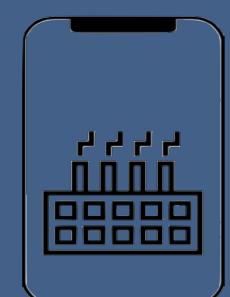
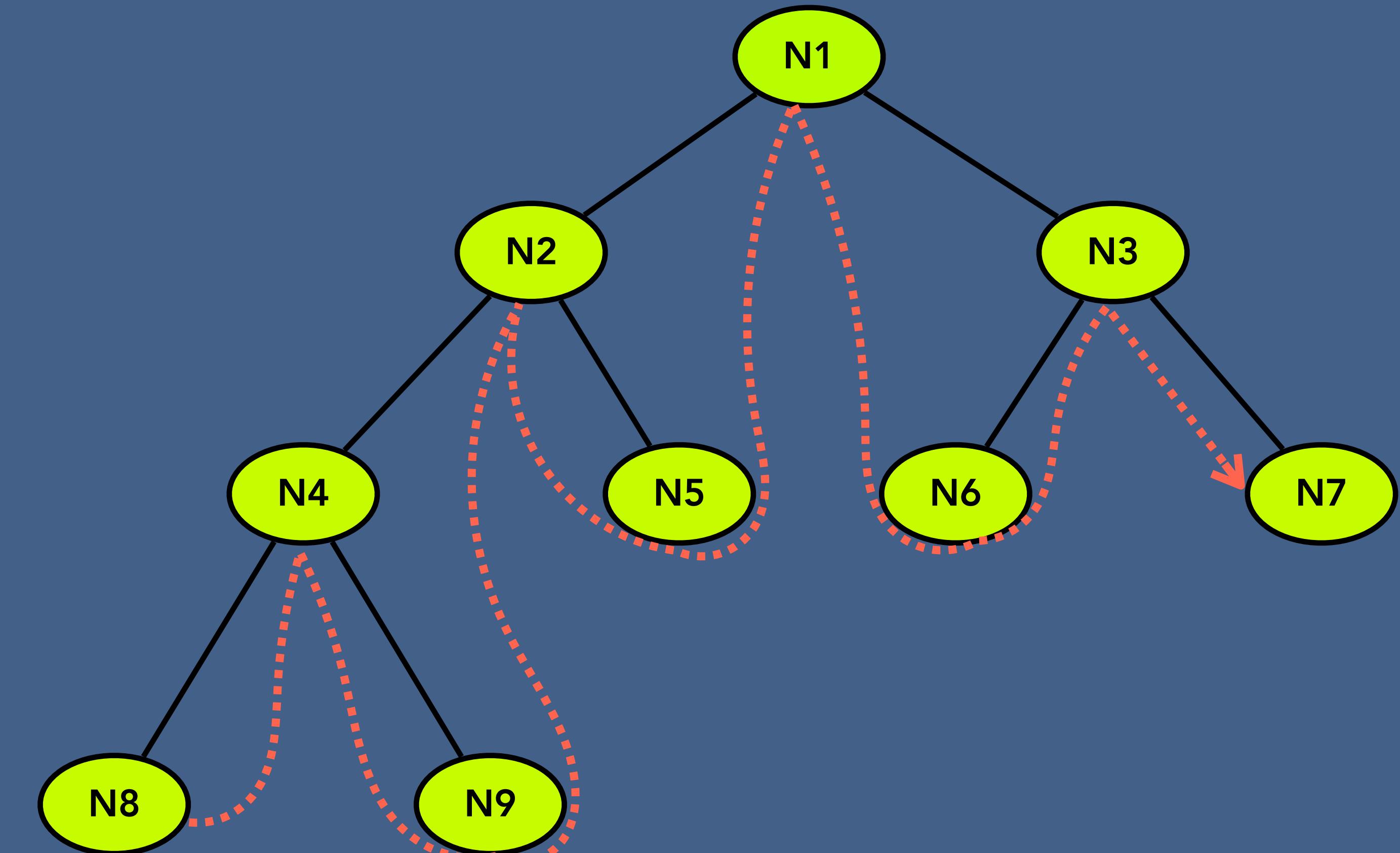
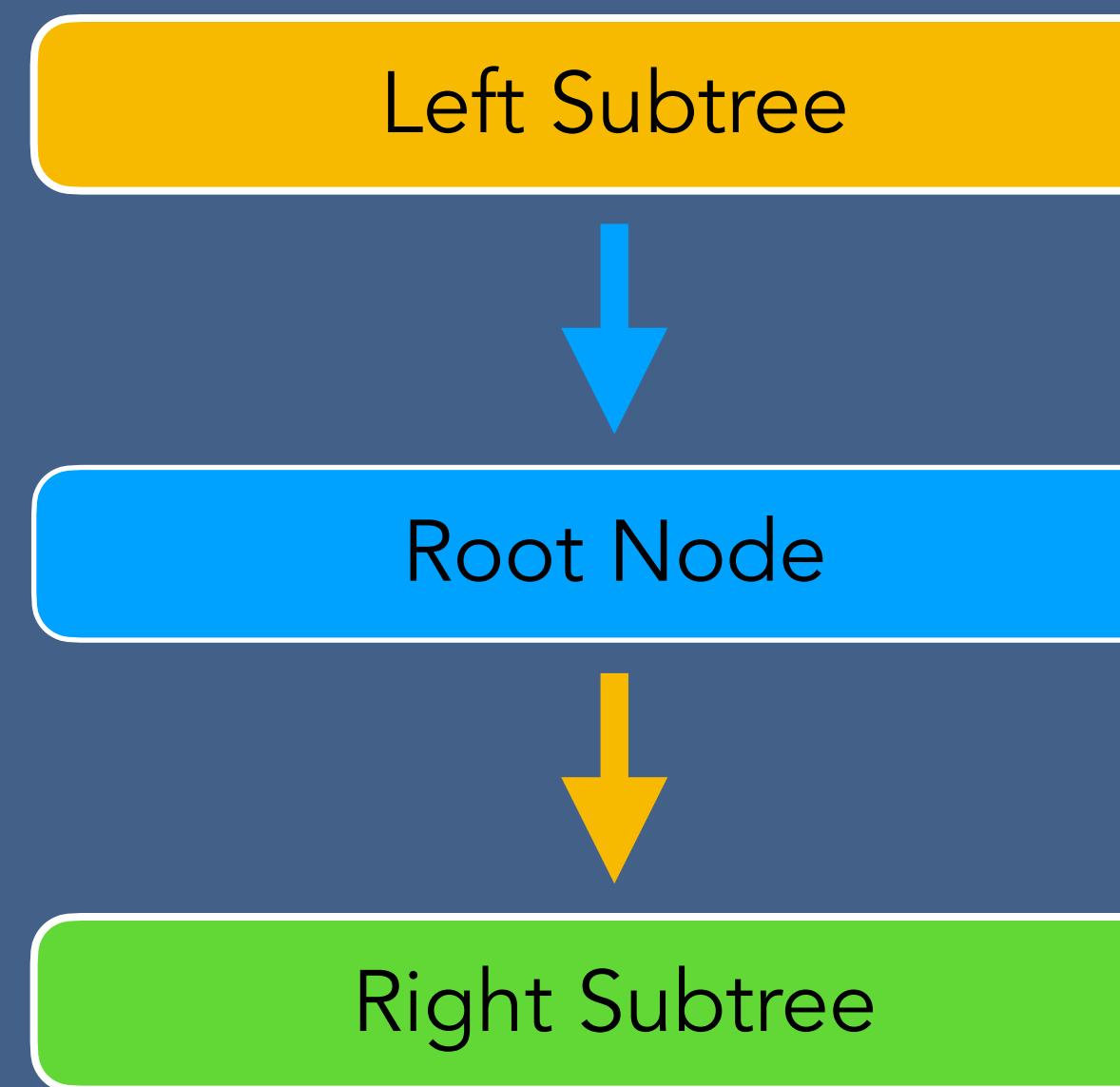
# Binary Tree (Array) - PreOrder Traversal



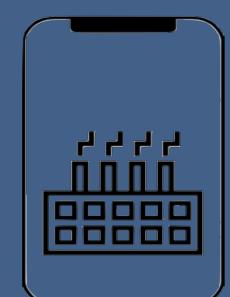
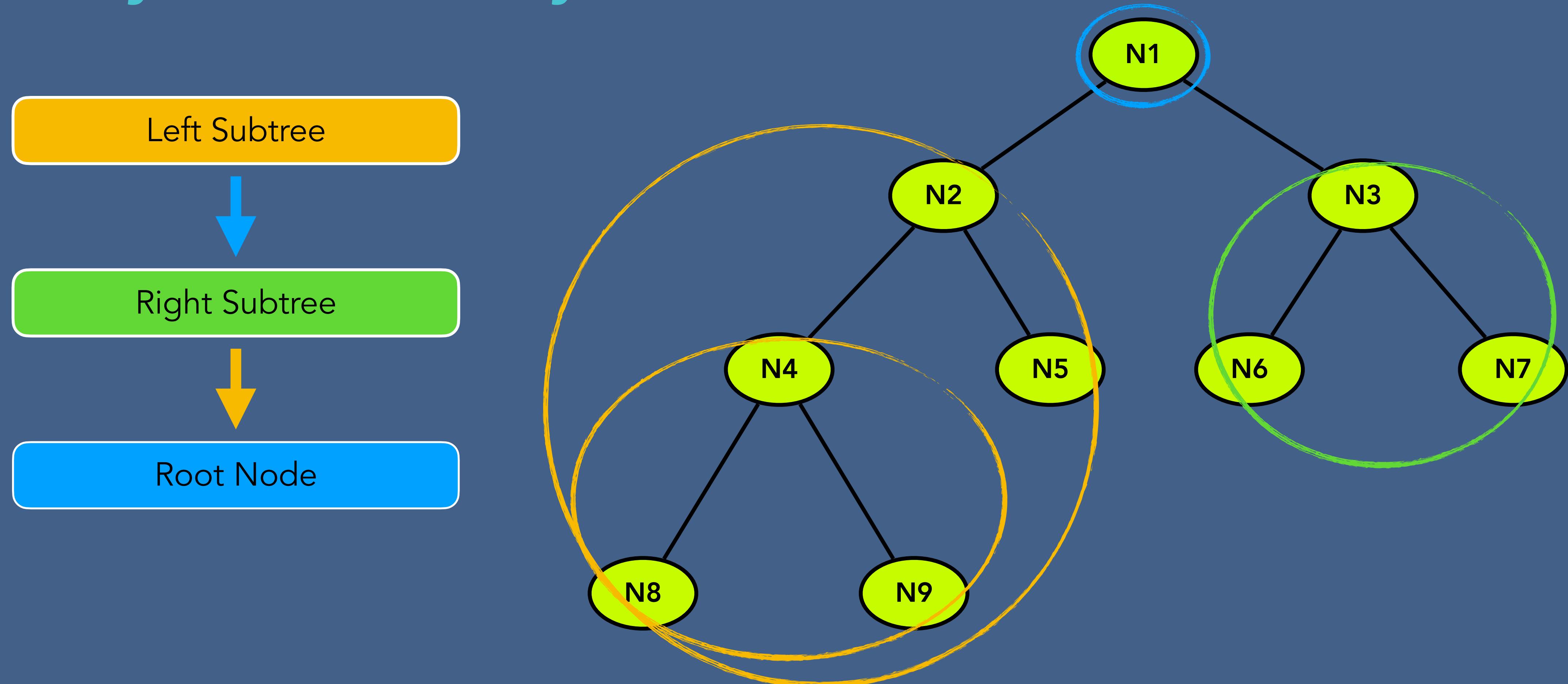
# Binary Tree (Array) - InOrder Traversal



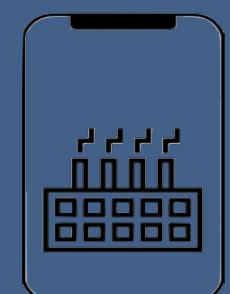
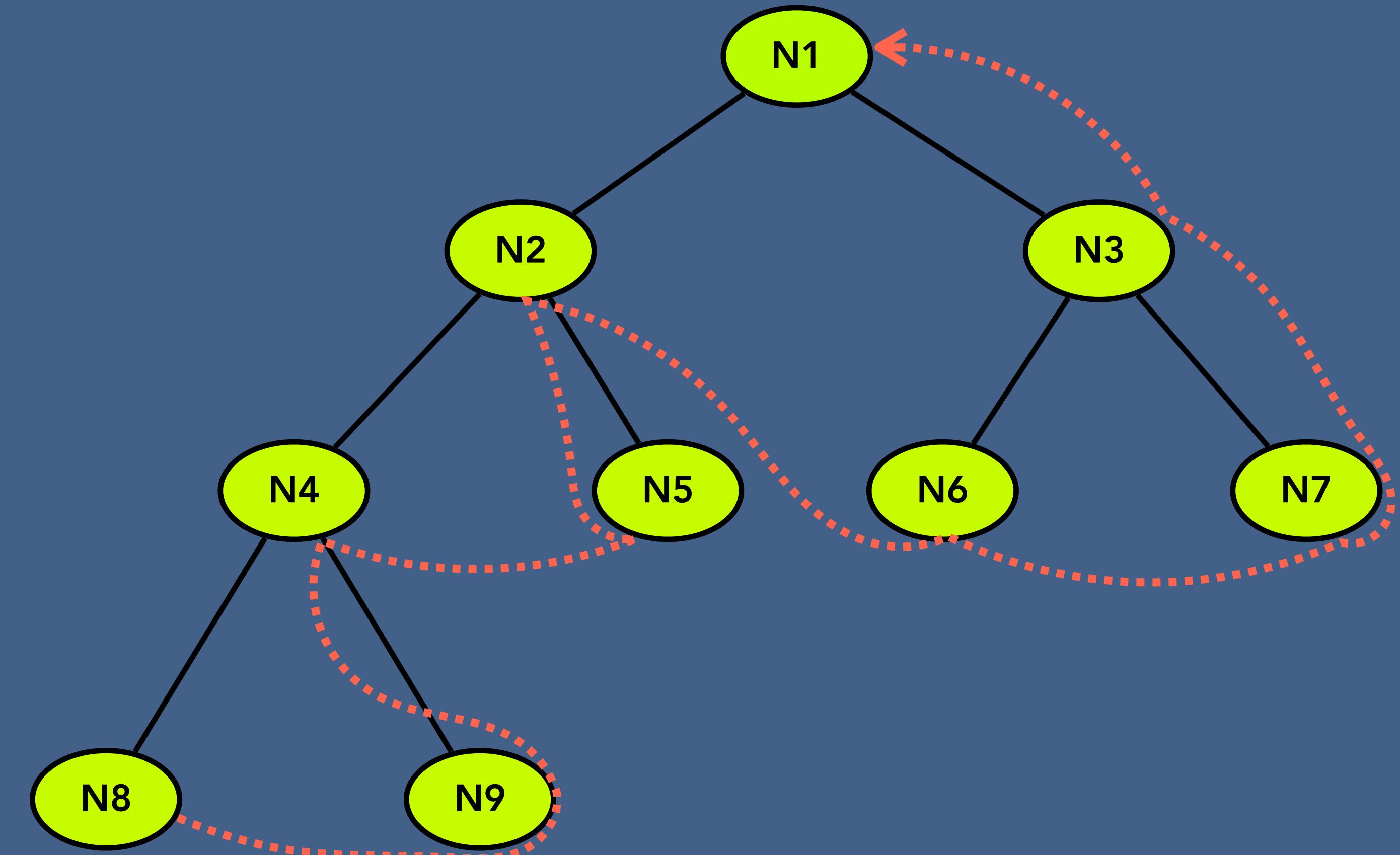
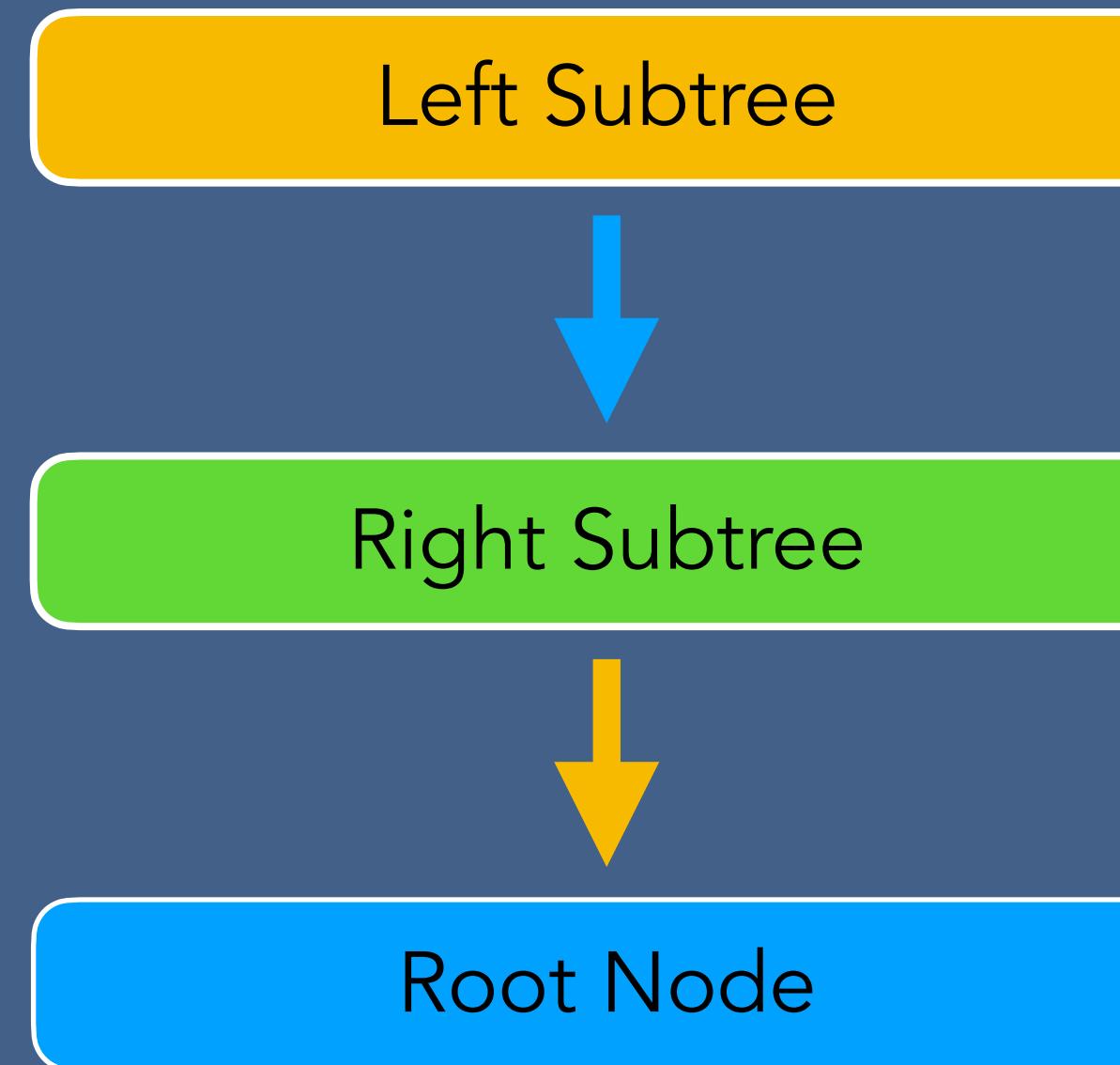
# Binary Tree (Array) - InOrder Traversal



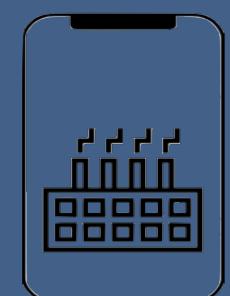
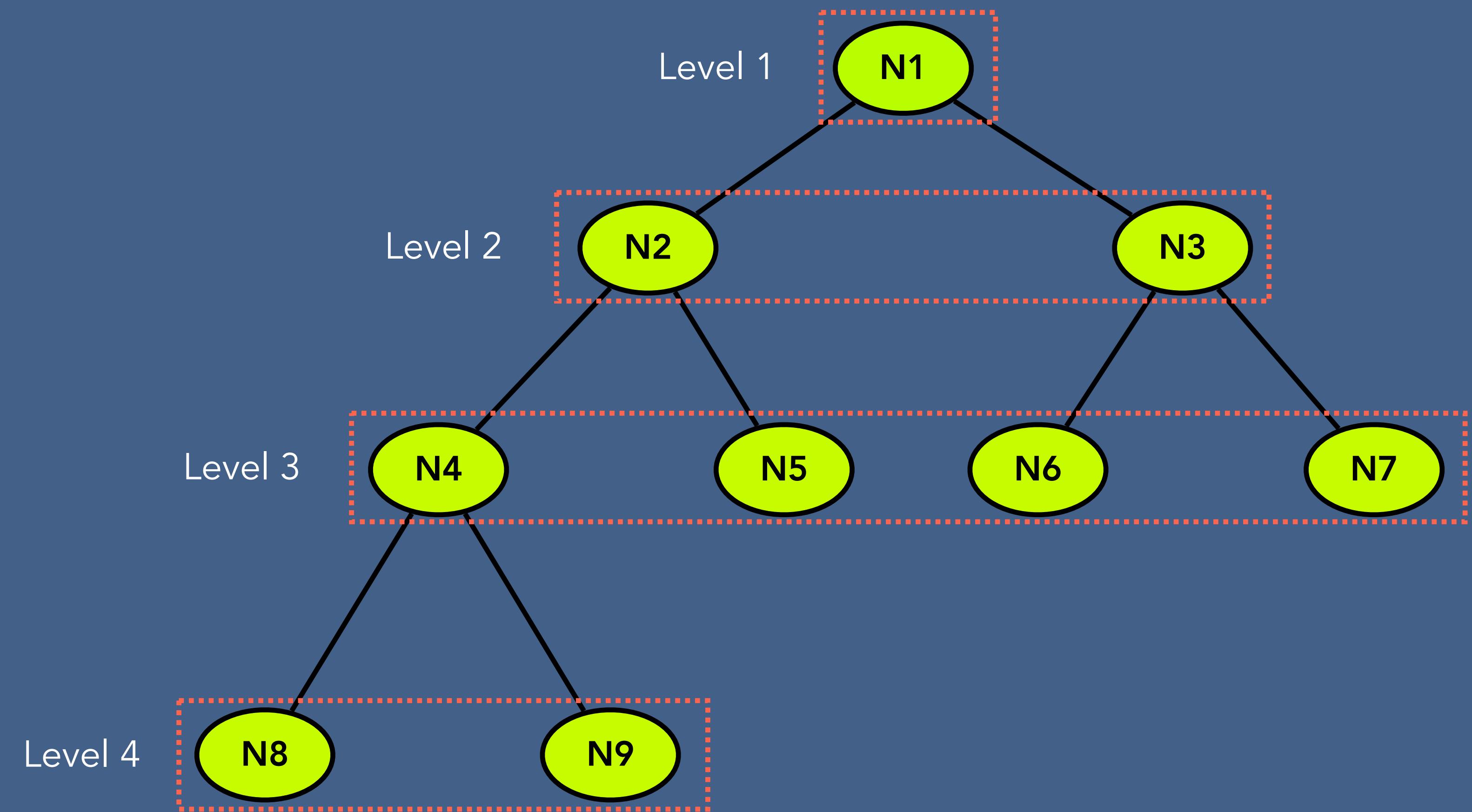
# Binary Tree (Array) - Post Traversal



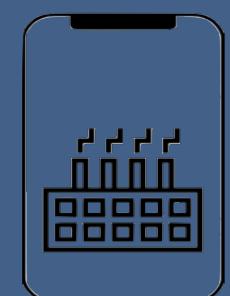
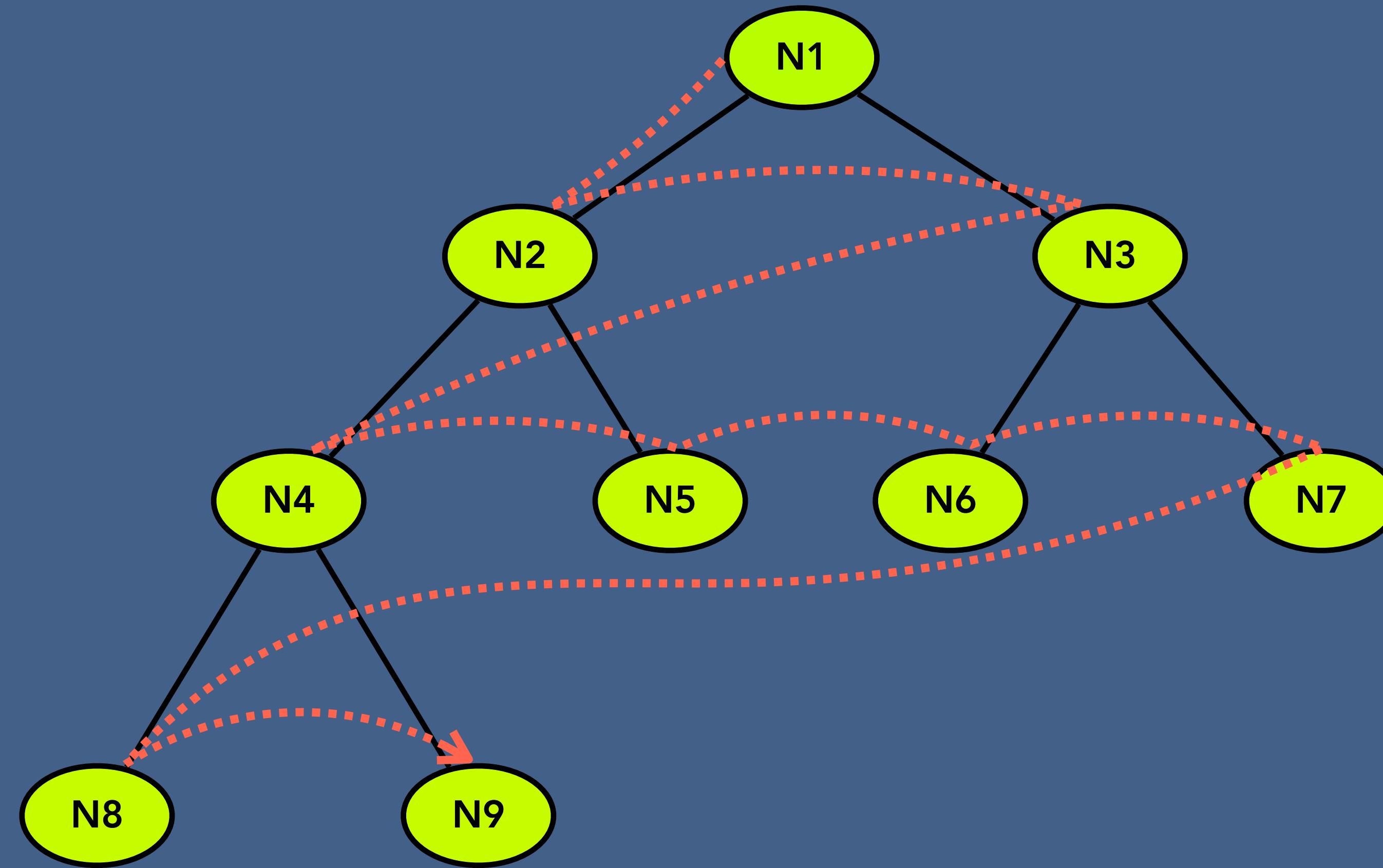
# Binary Tree (Array) - Post Traversal



# Binary Tree (Array) - LevelOrder Traversal



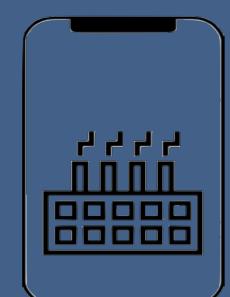
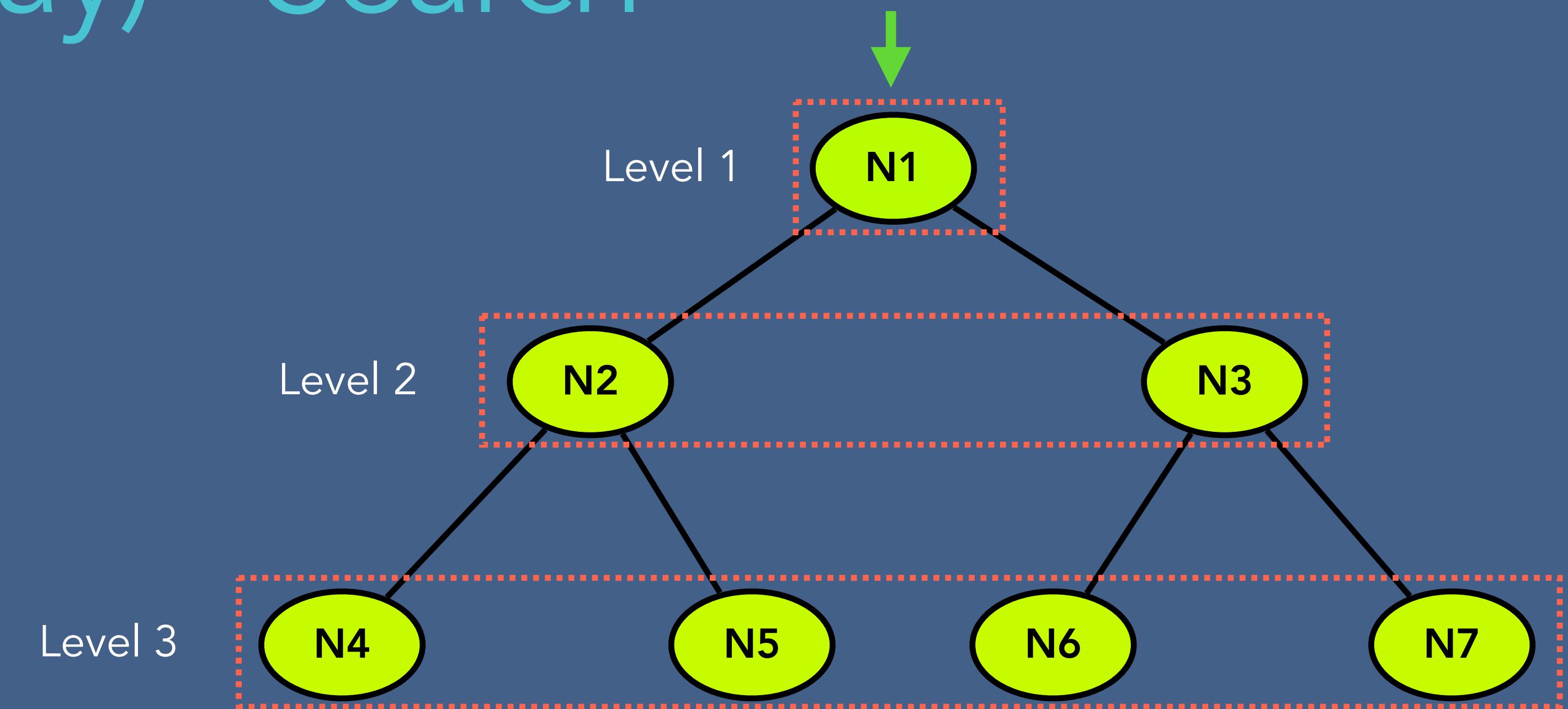
# Binary Tree - LevelOrder Traversal



# Binary Tree (Array) - Search

Level Order Traversal

N5 → 5

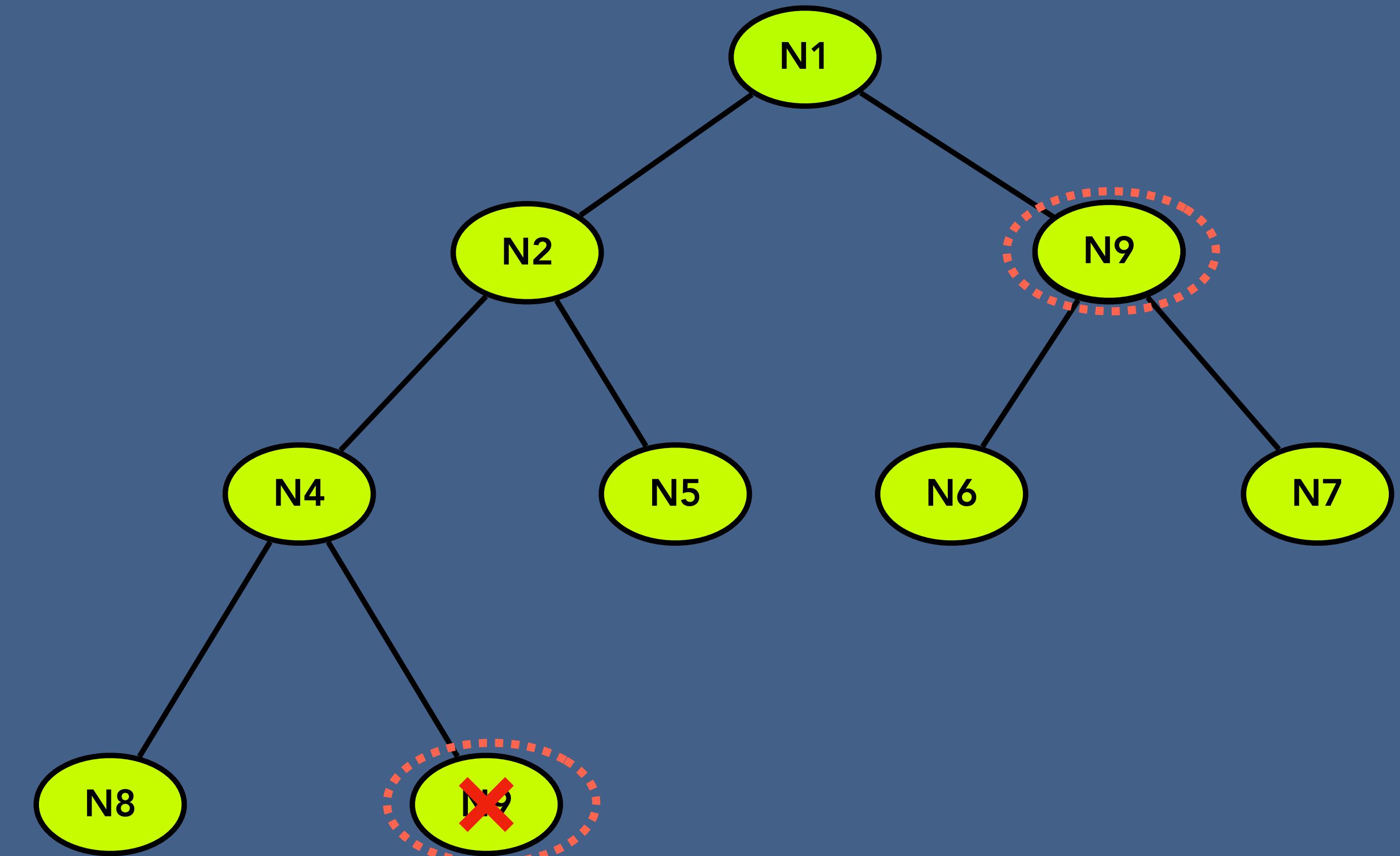


# Binary Tree (Array) - Delete a Node

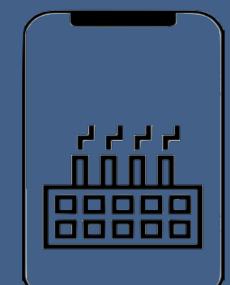
Level Order Traversal

N3

deepestNode = lastUsedIndex

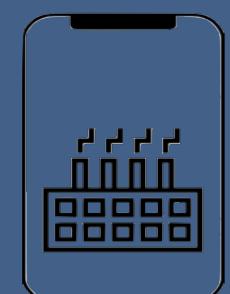
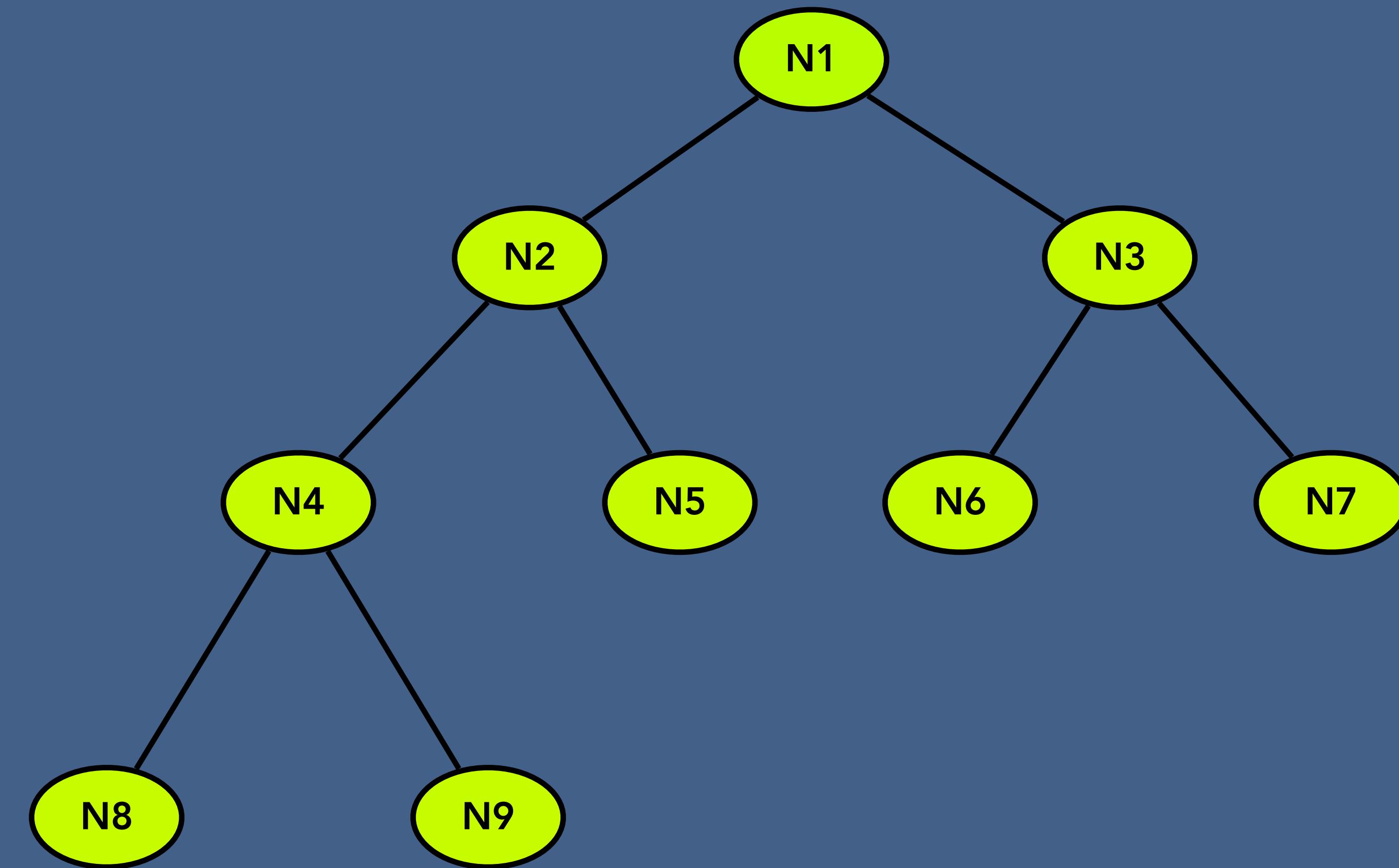


X	N1	N2	N9	N4	N5	N6	N7	N8			
0	1	2	3	4	5	6	7	8	9	10	11



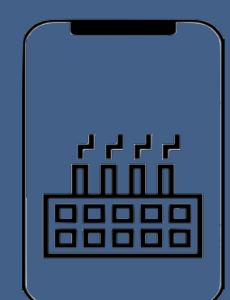
# Binary Tree (Array) - Delete Binary Tree

arr = Null

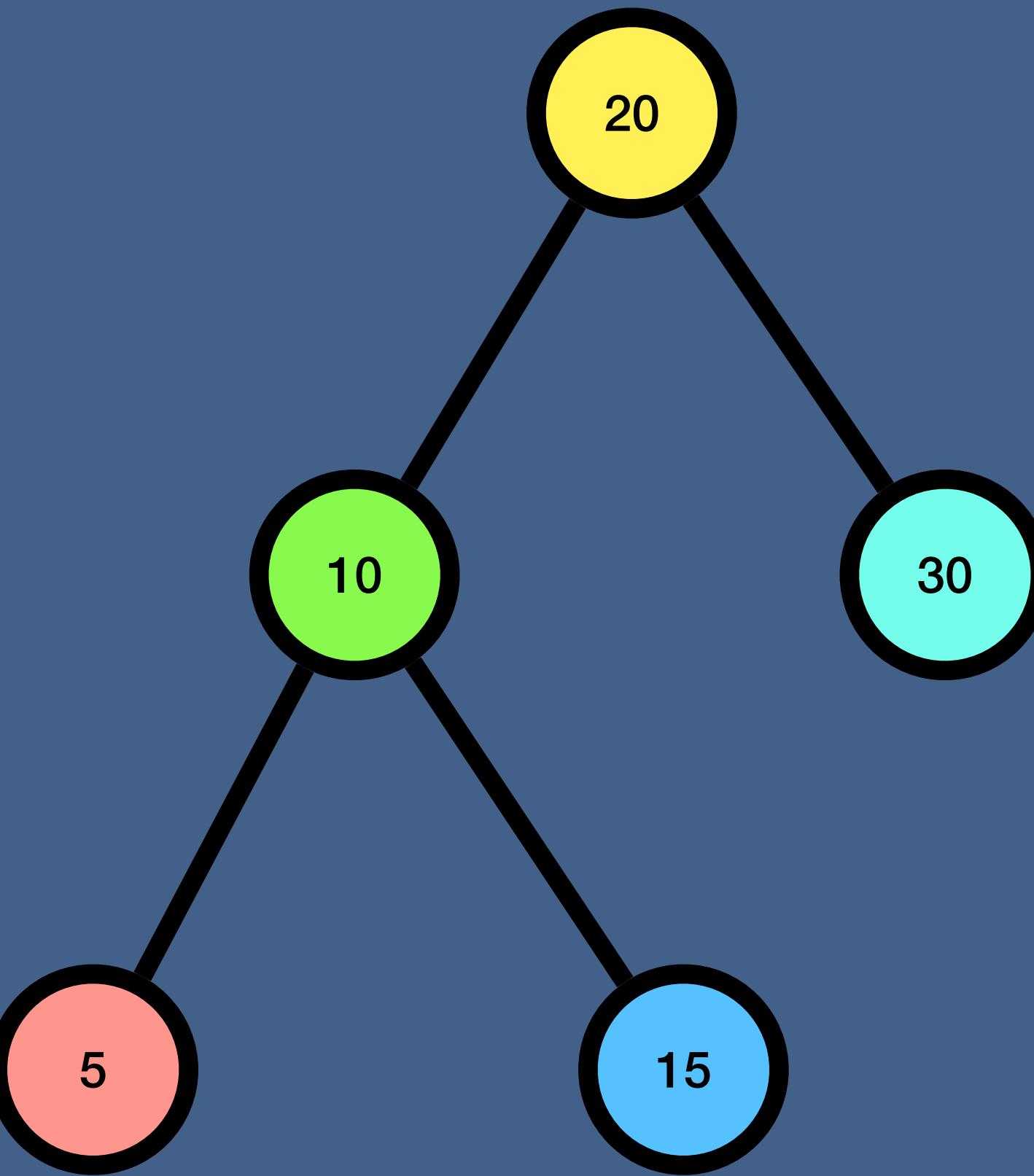


# Binary Tree (Array vs Linked List)

	Array		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)	O(n)
Delete entire Binary Tree	O(1)	O(1)	O(1)	O(1)
Space efficient?		No		Yes

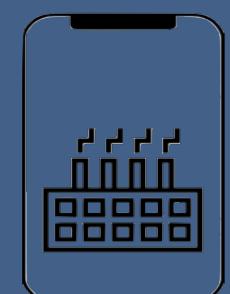
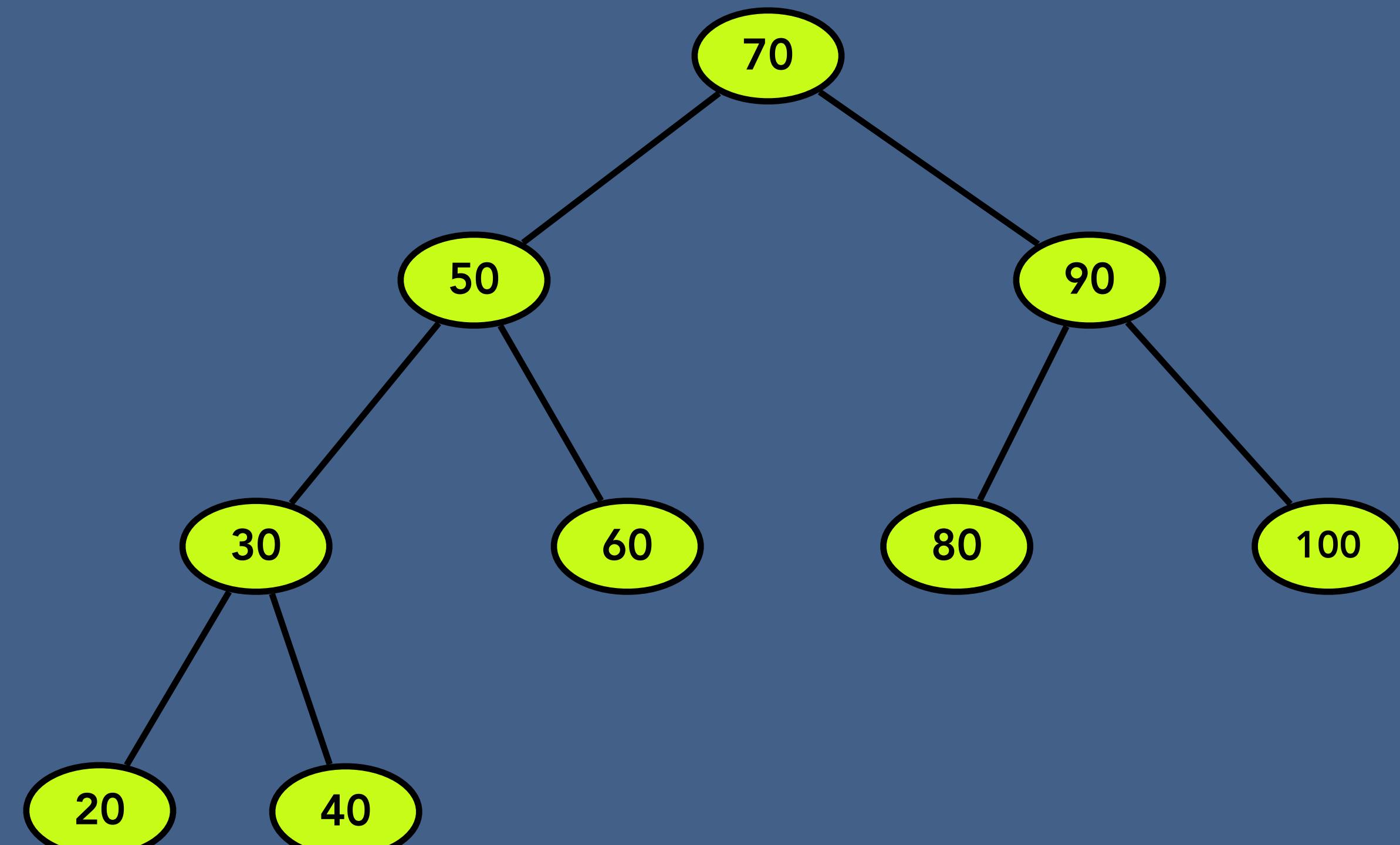


# Binary Search Tree



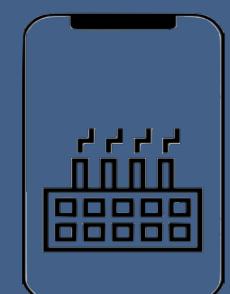
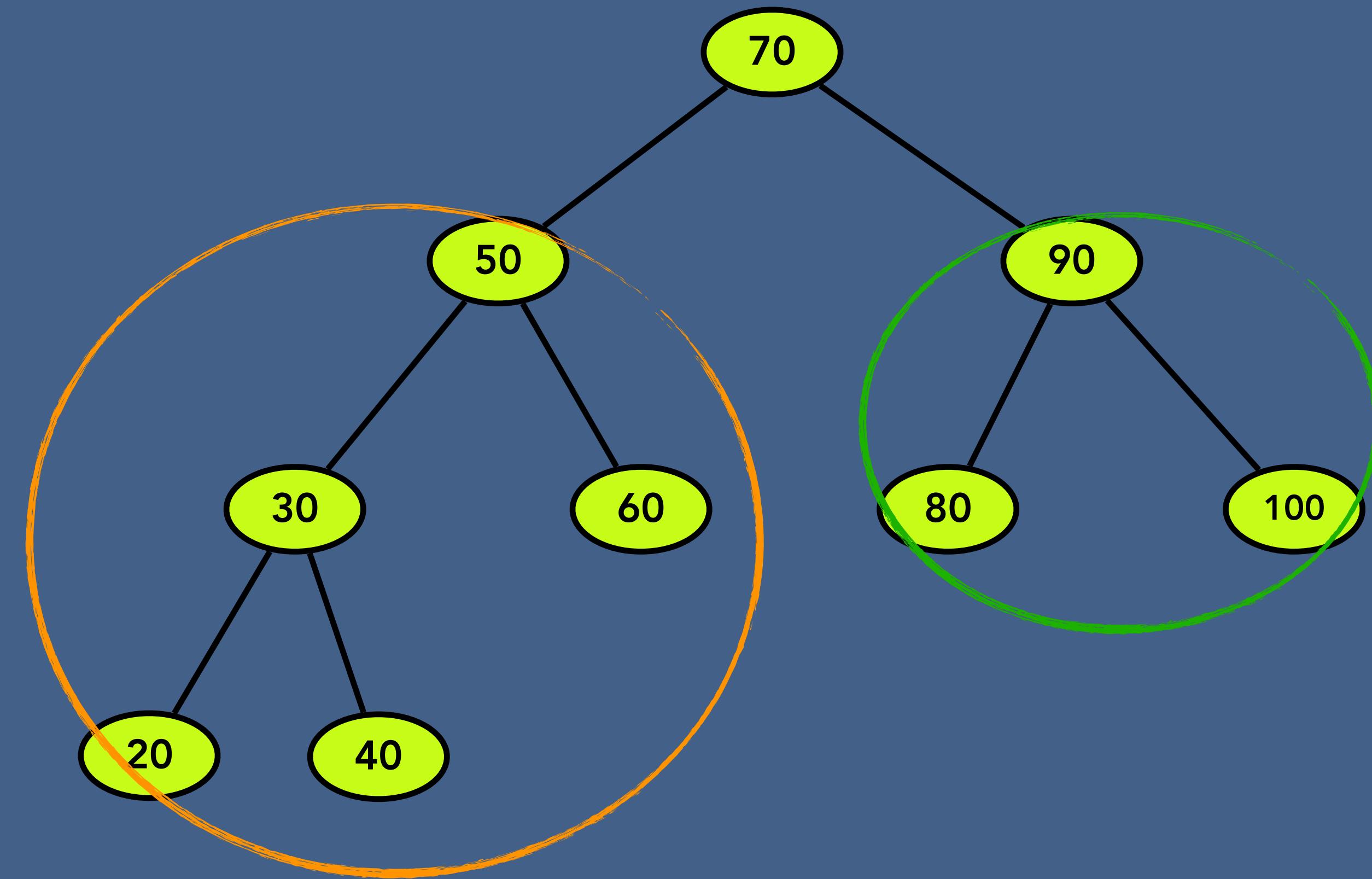
# What is a Binary Search Tree?

- 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



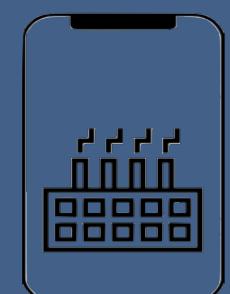
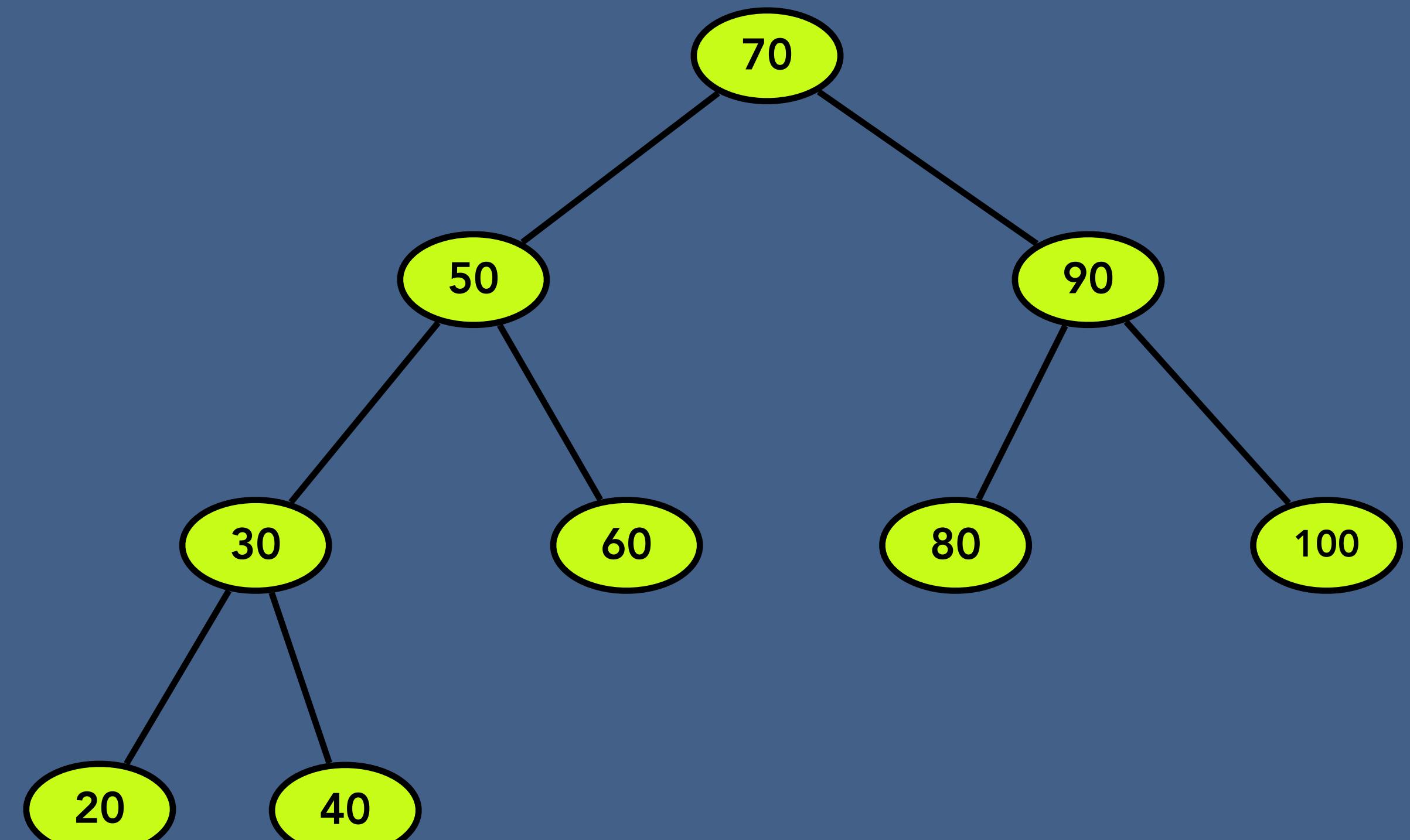
# What is a Binary Search Tree?

- 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



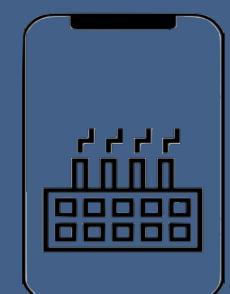
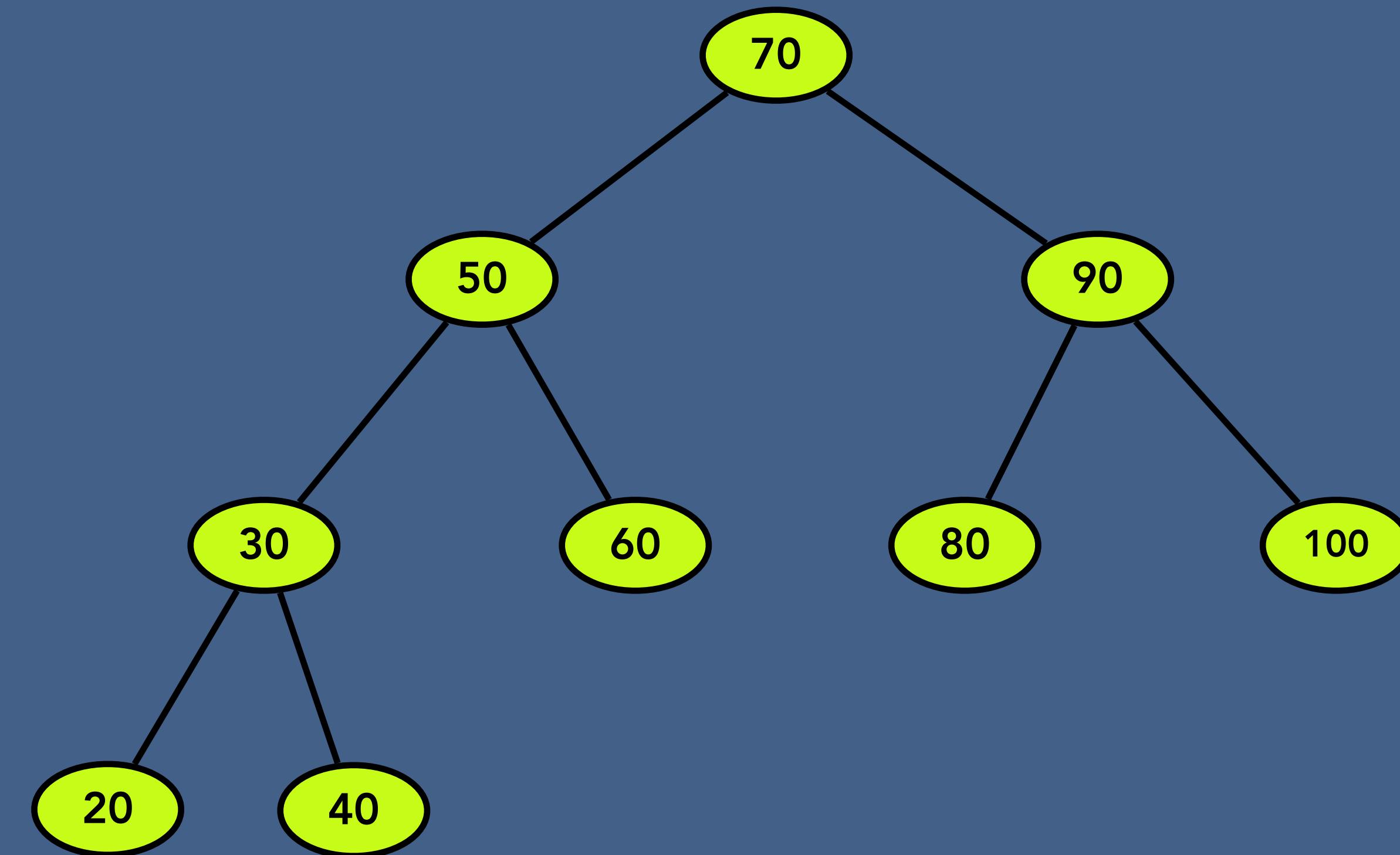
# Why Binary Search Tree?

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



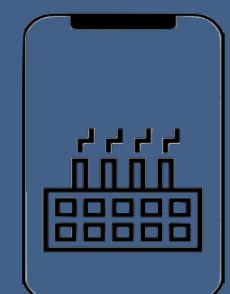
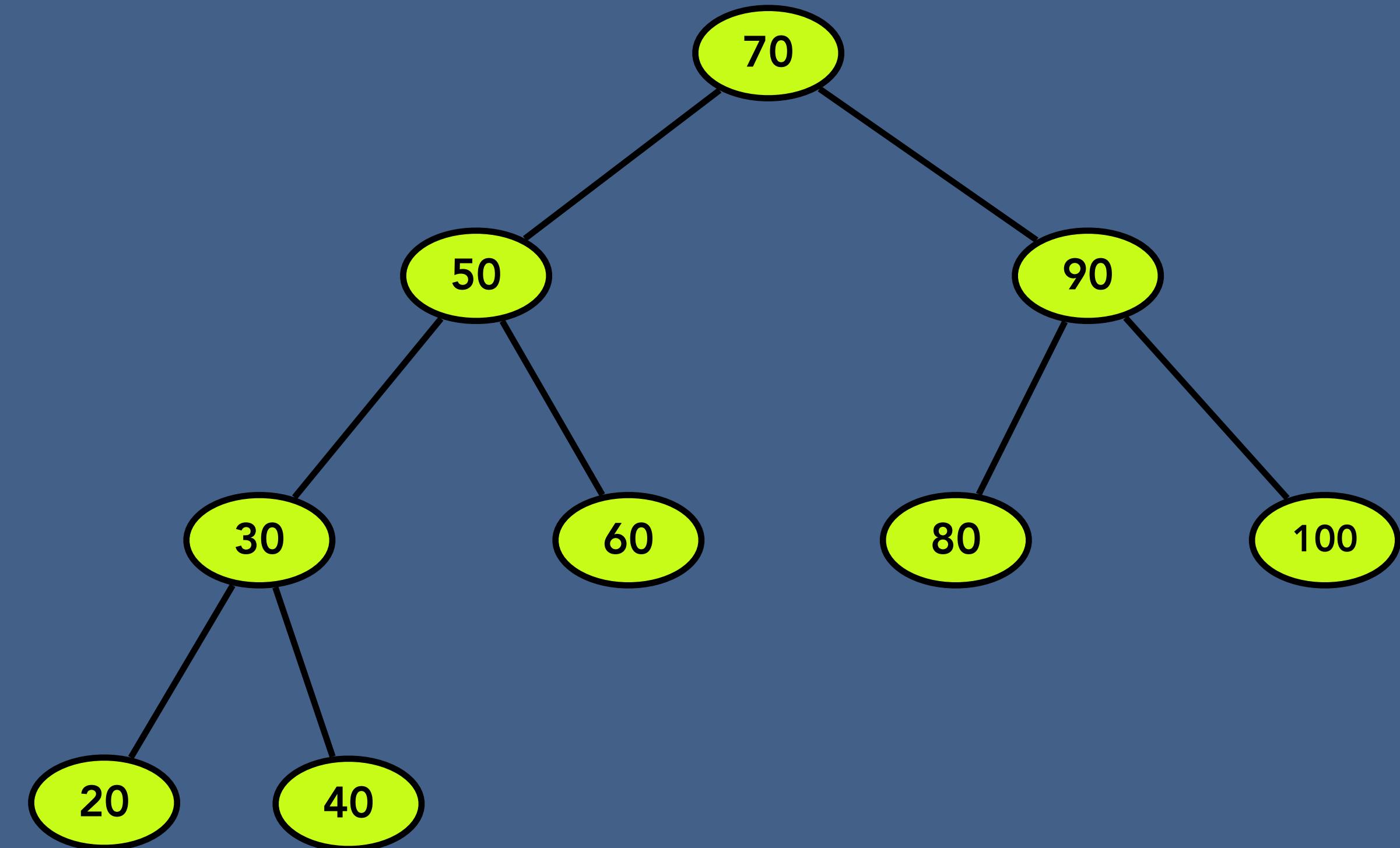
# Common Operations on Binary Search Tree

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

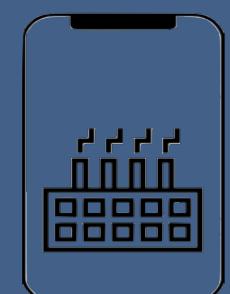
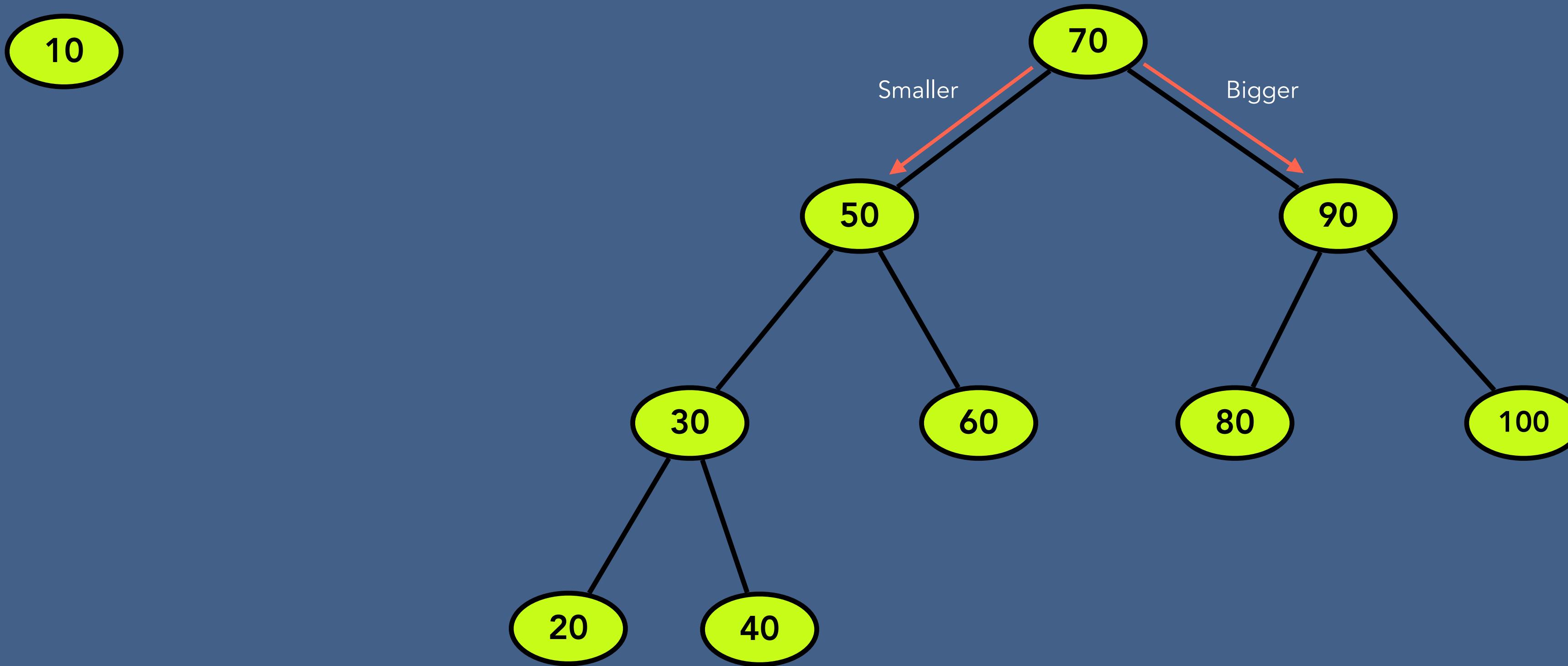


# Create Binary Search Tree

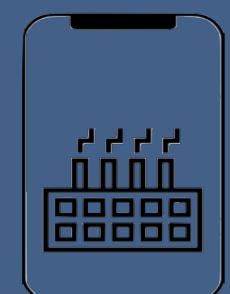
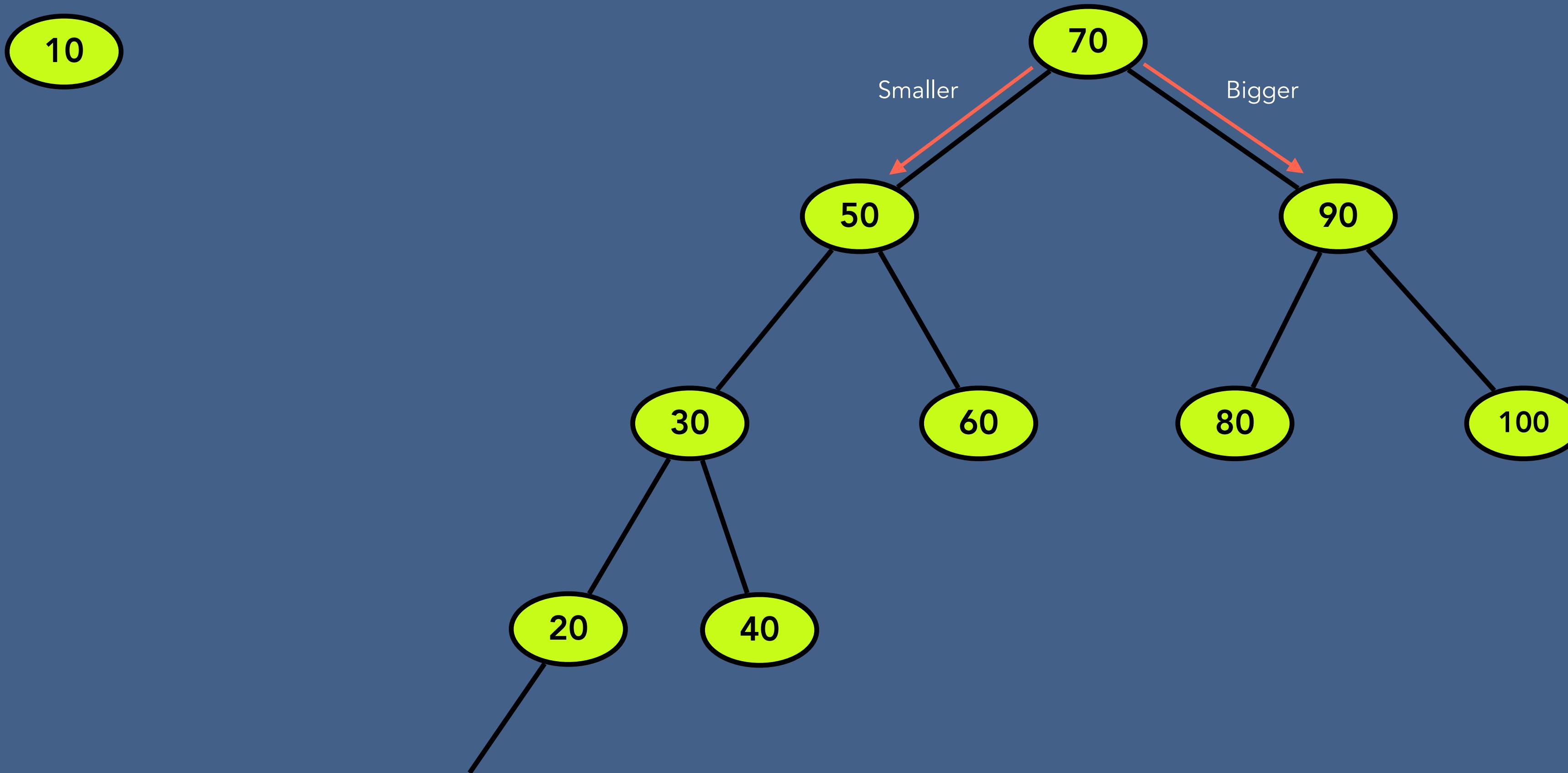
```
newBST = BST()
```



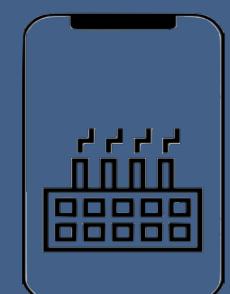
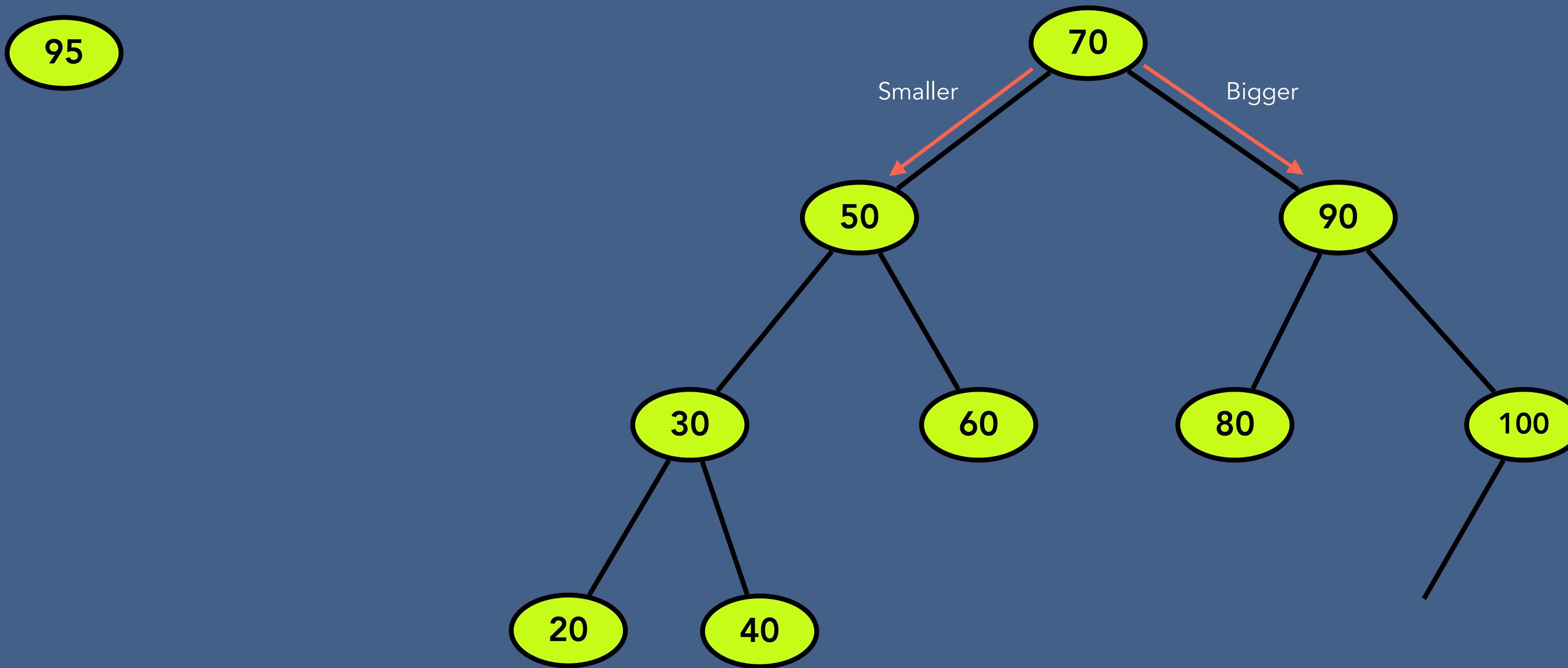
# Binary Search Tree - Insert a Node



# Binary Search Tree - Insert a Node



# Binary Search Tree - Insert a Node



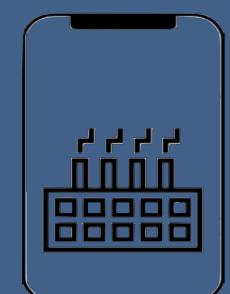
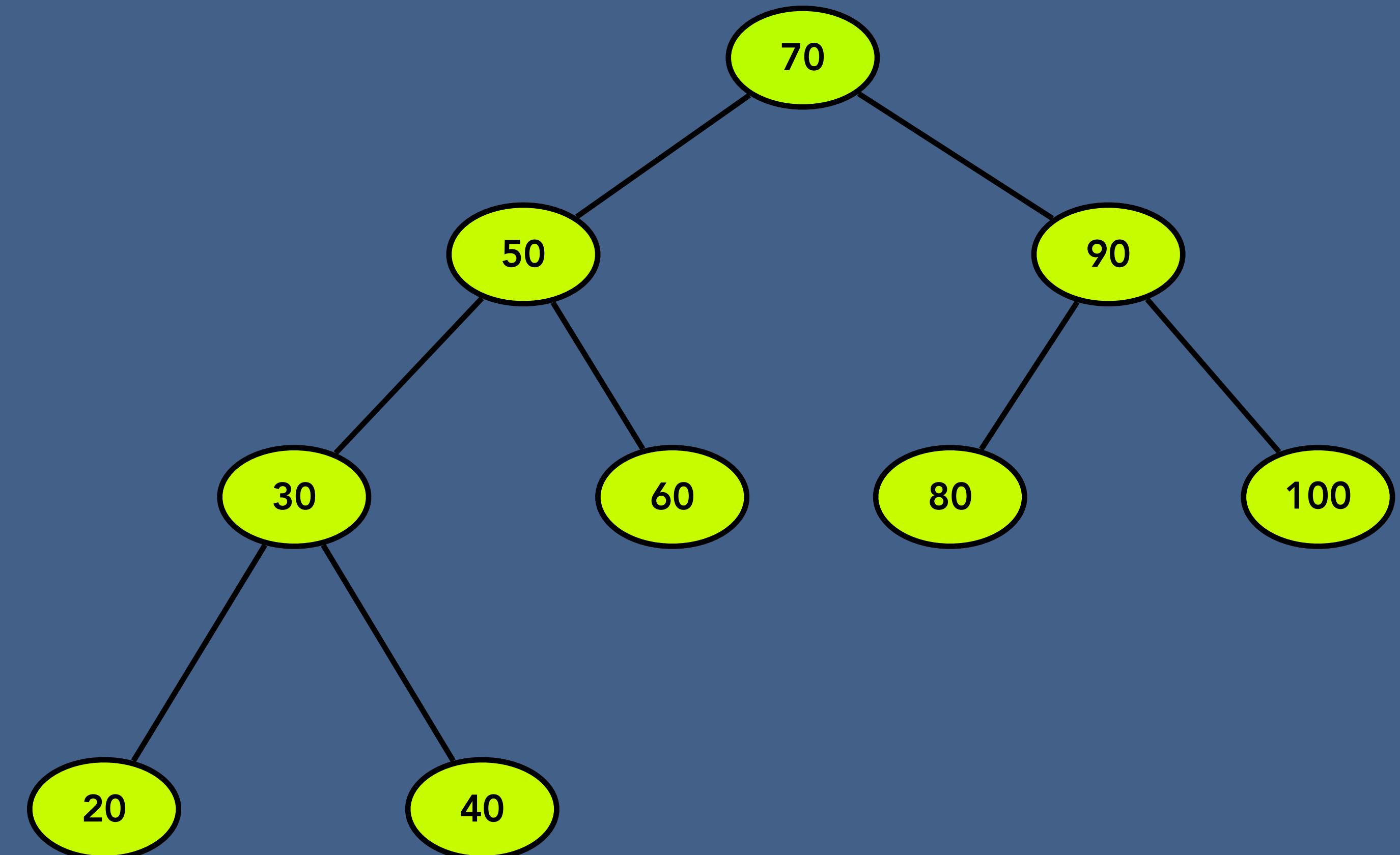
# Binary Search Tree - Traversal

## Depth first search

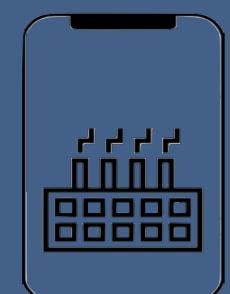
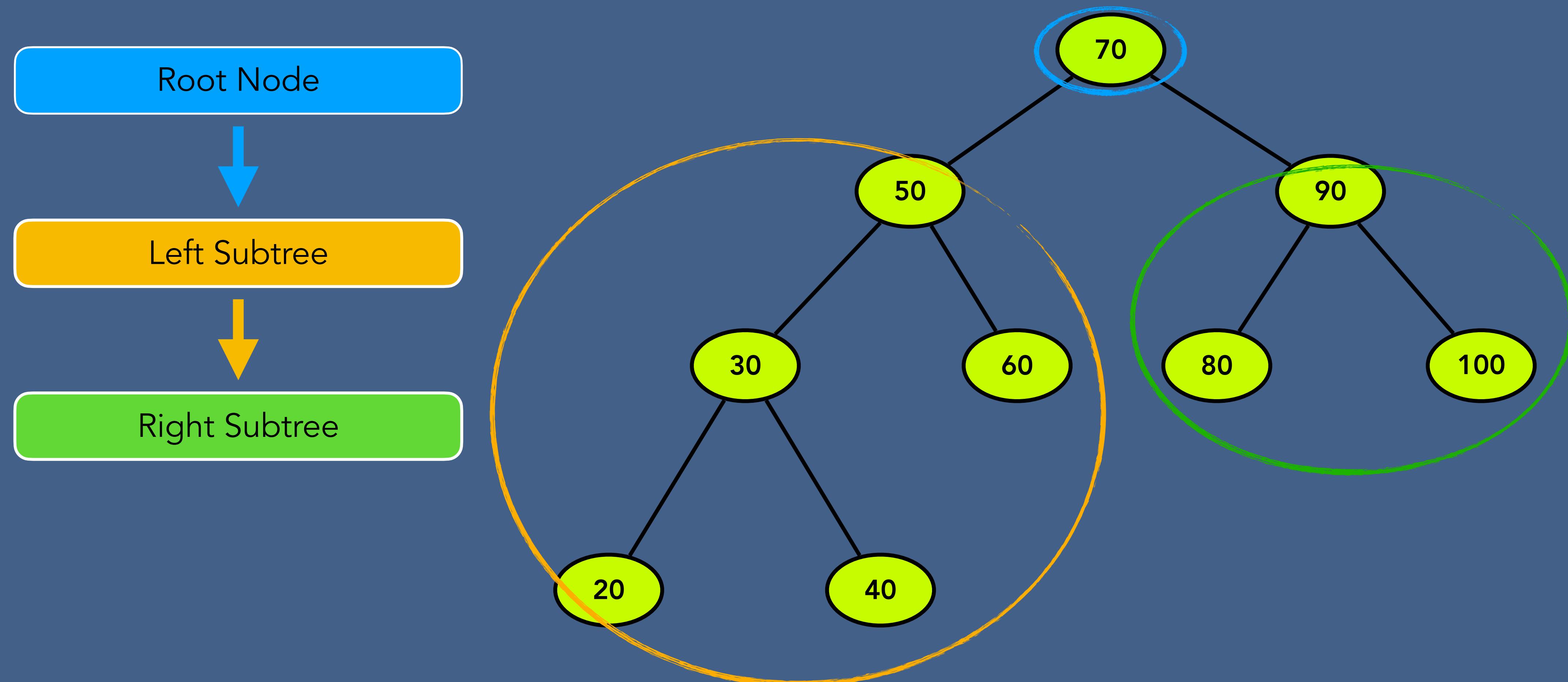
- Preorder traversal
- Inorder traversal
- Post order traversal

## Breadth first search

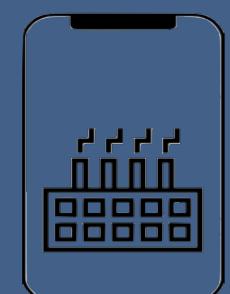
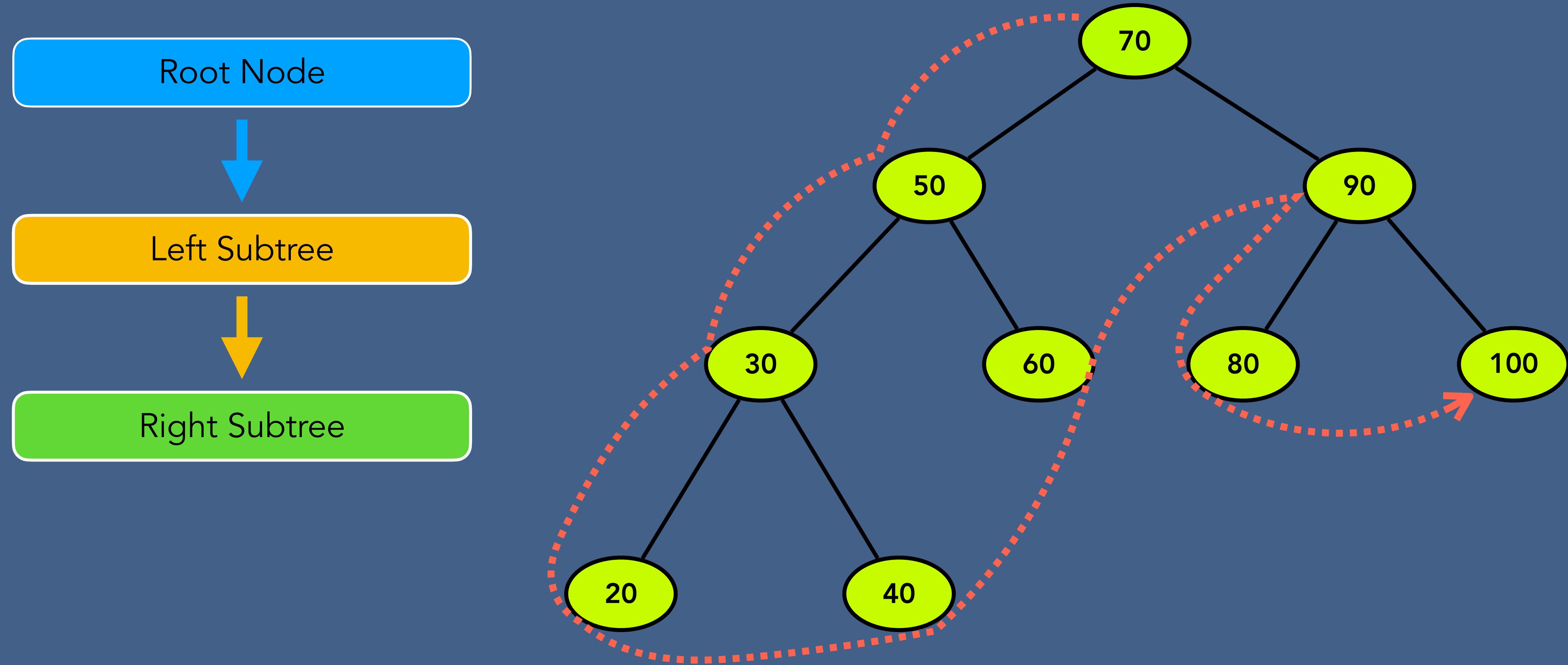
- Level order traversal



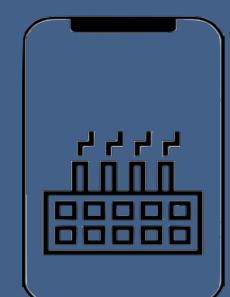
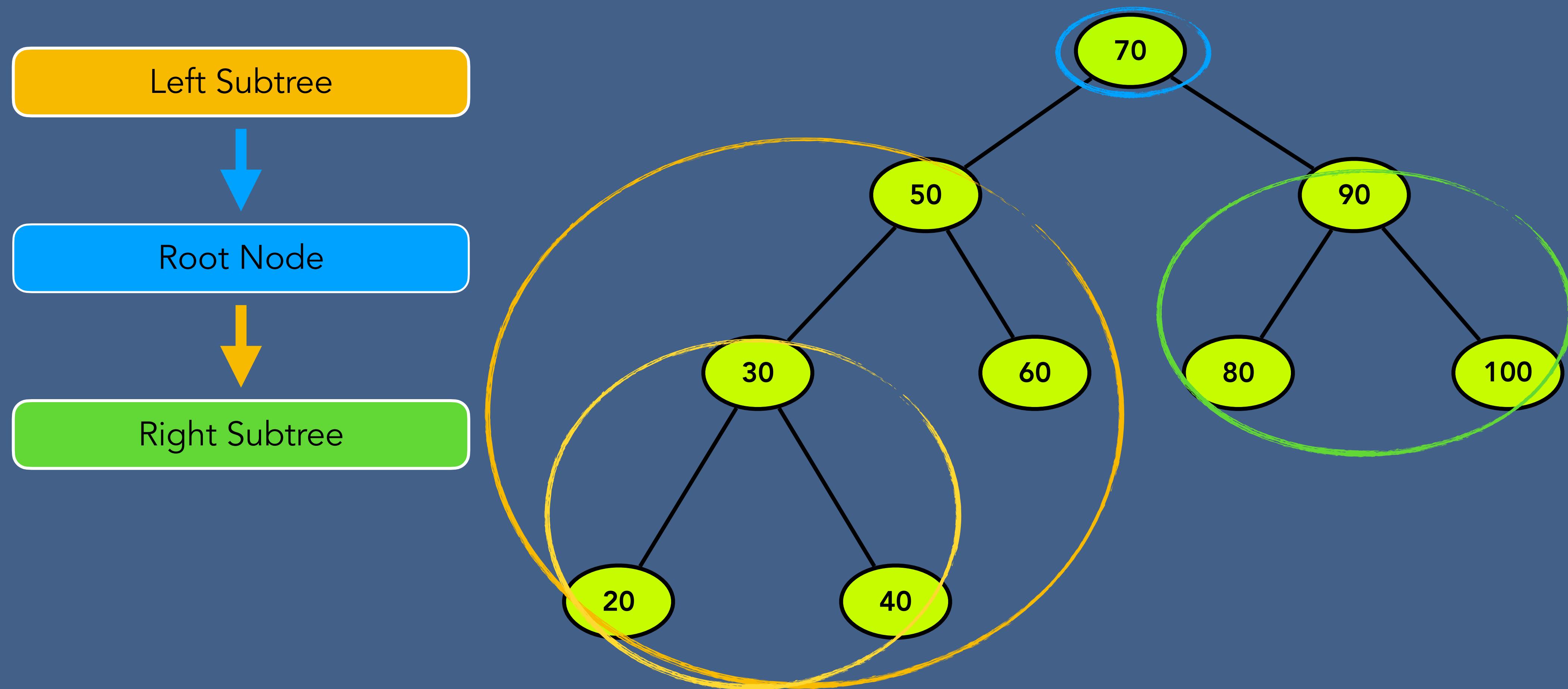
# Binary Search Tree - PreOrder Traversal



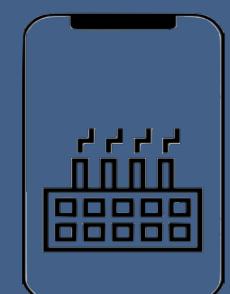
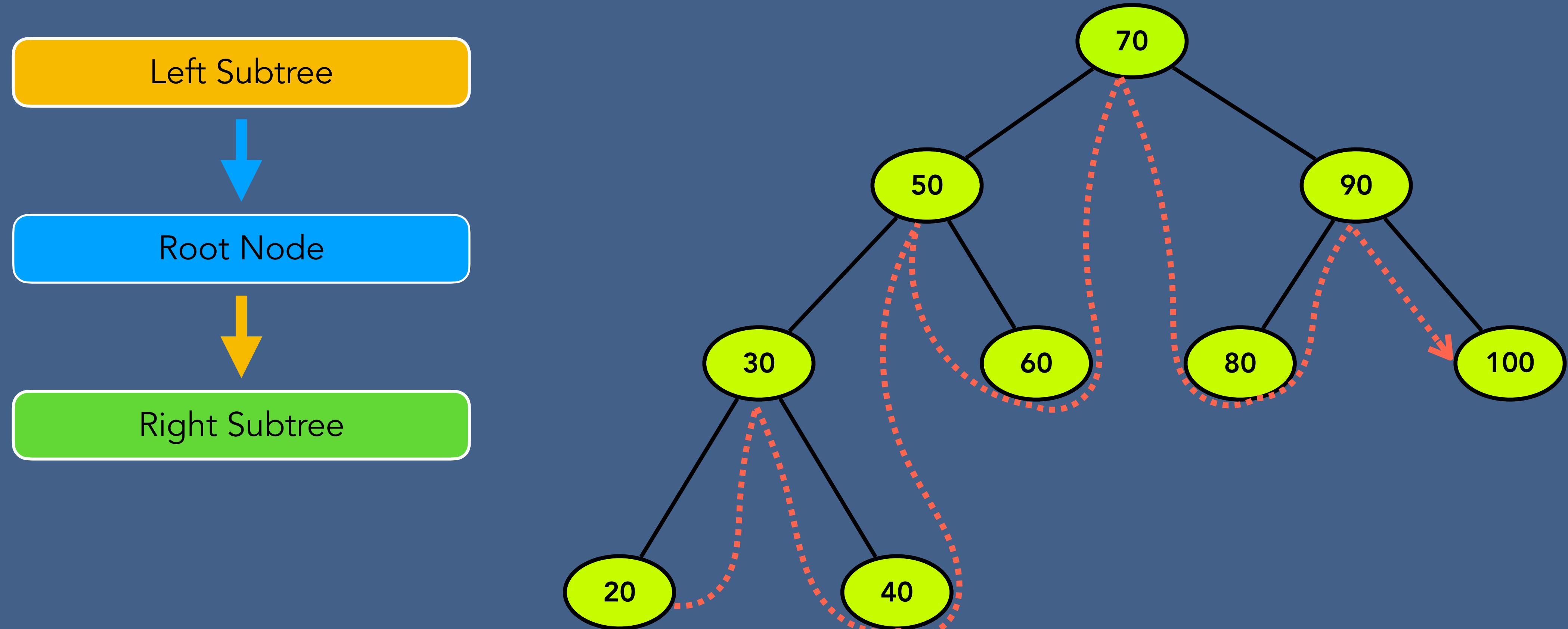
# Binary Search Tree - PreOrder Traversal



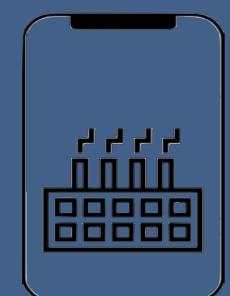
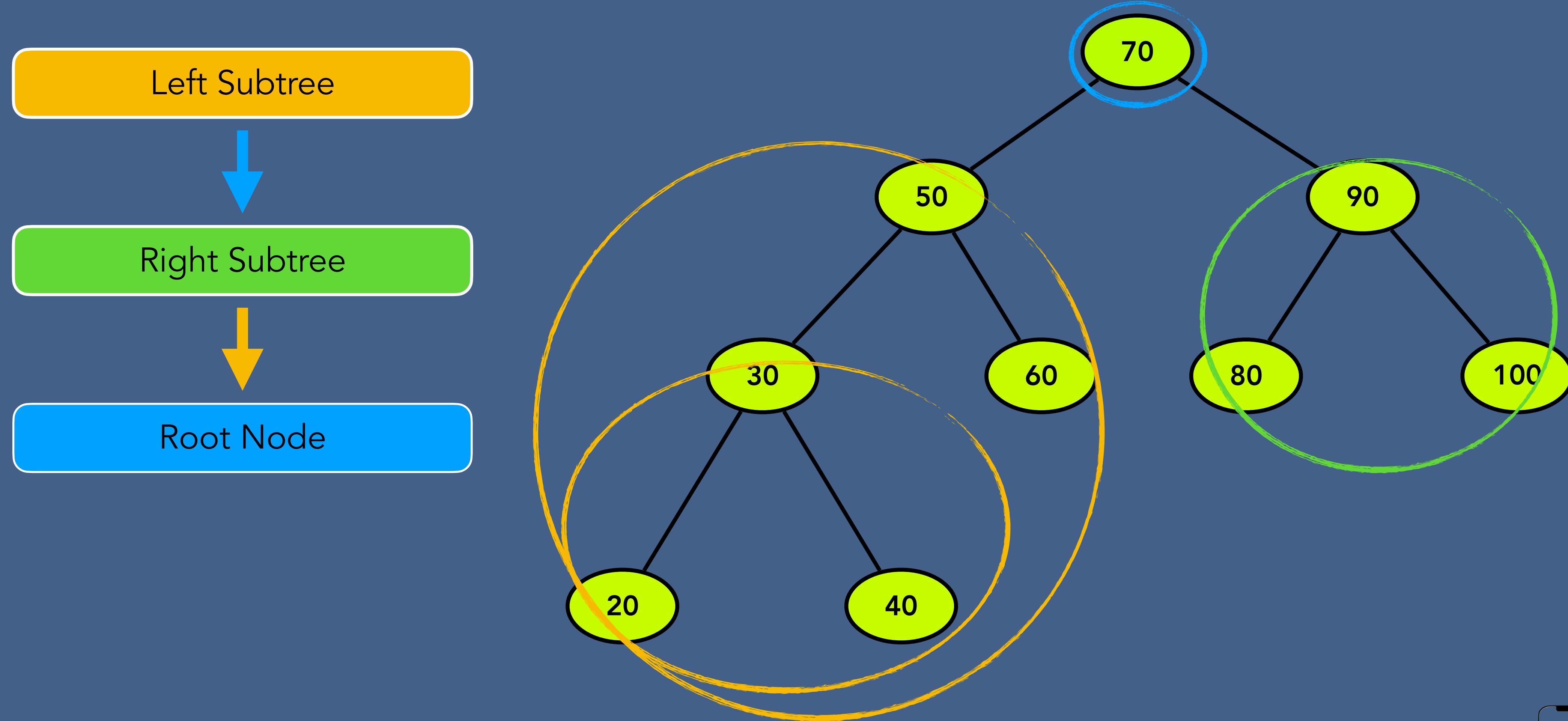
# Binary Search Tree - InOrder Traversal



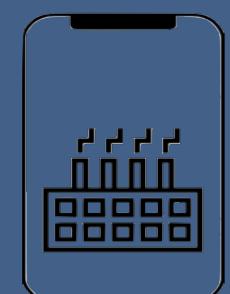
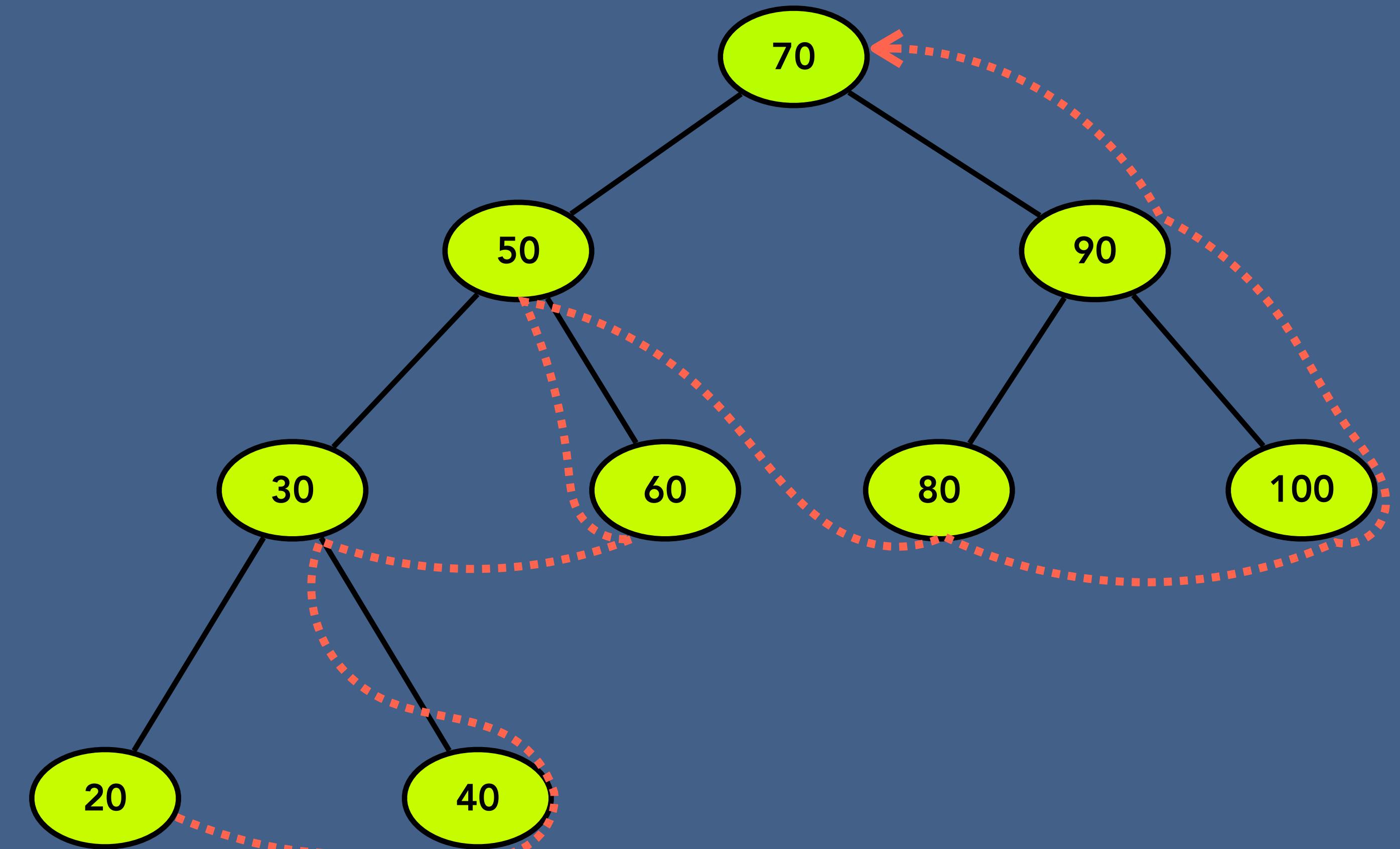
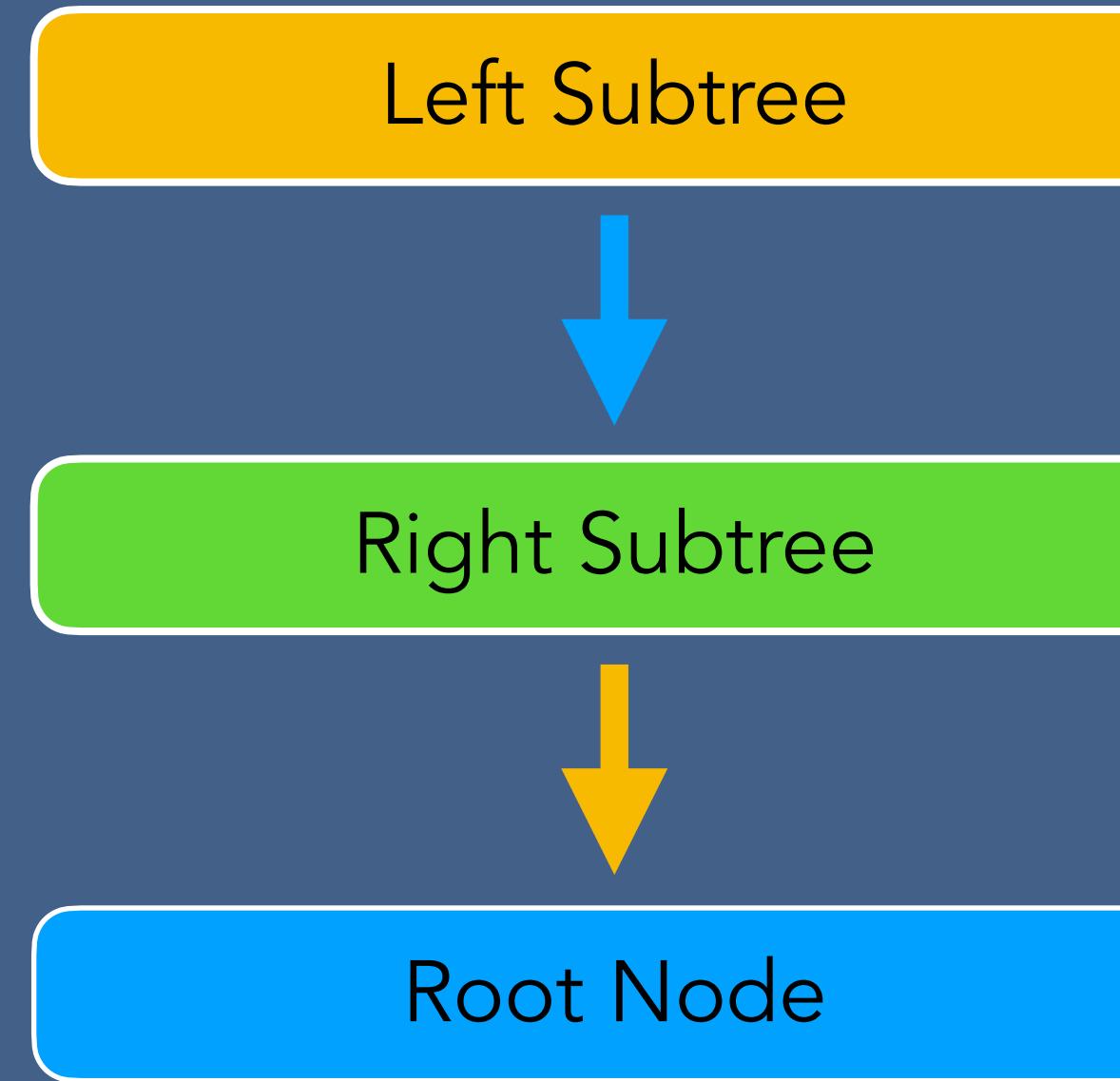
# Binary Search Tree- InOrder Traversal



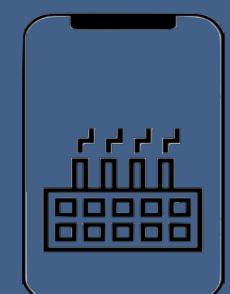
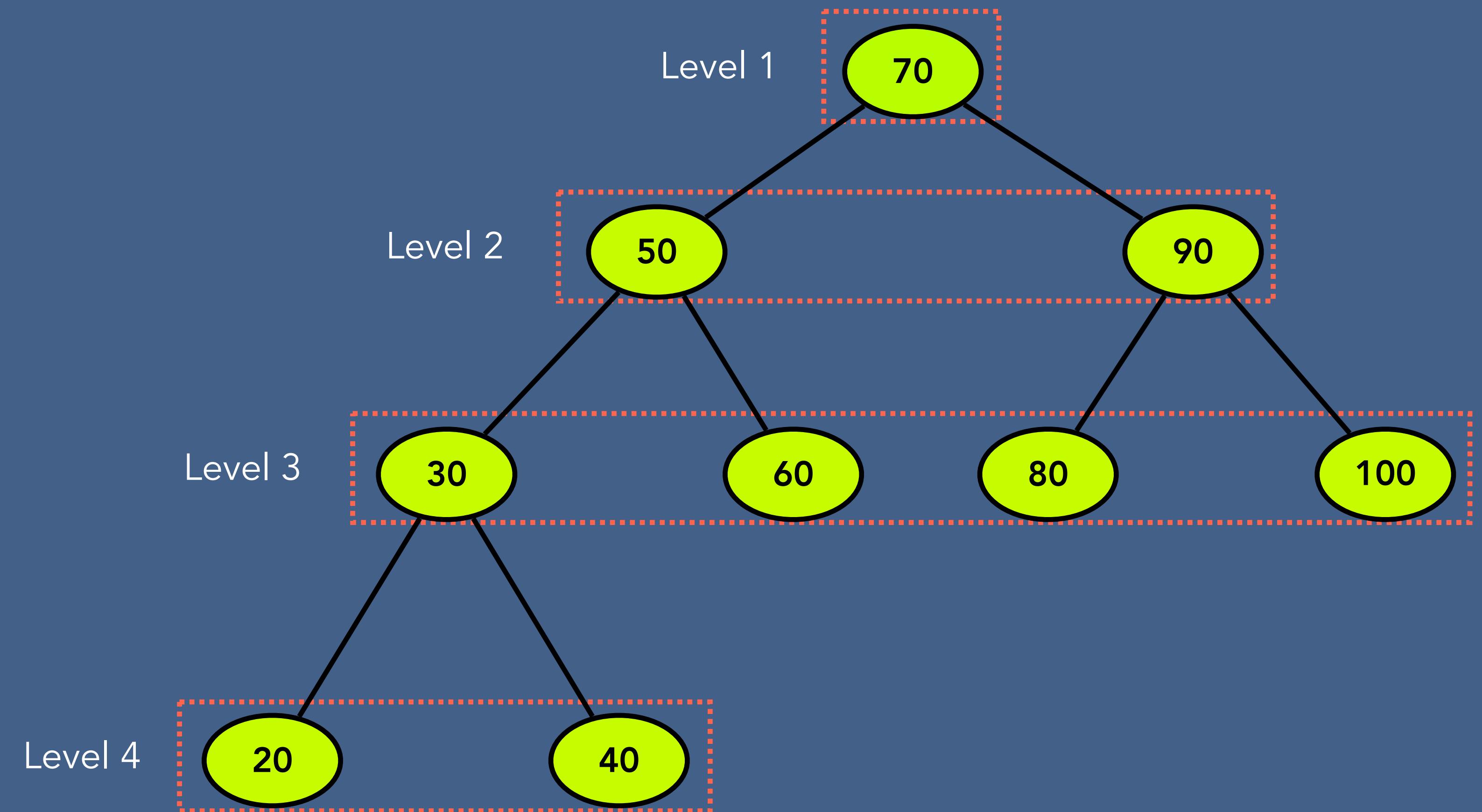
# Binary Search Tree- Post Traversal



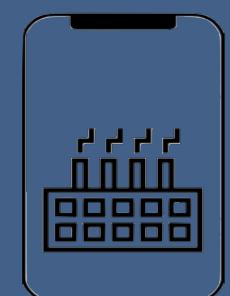
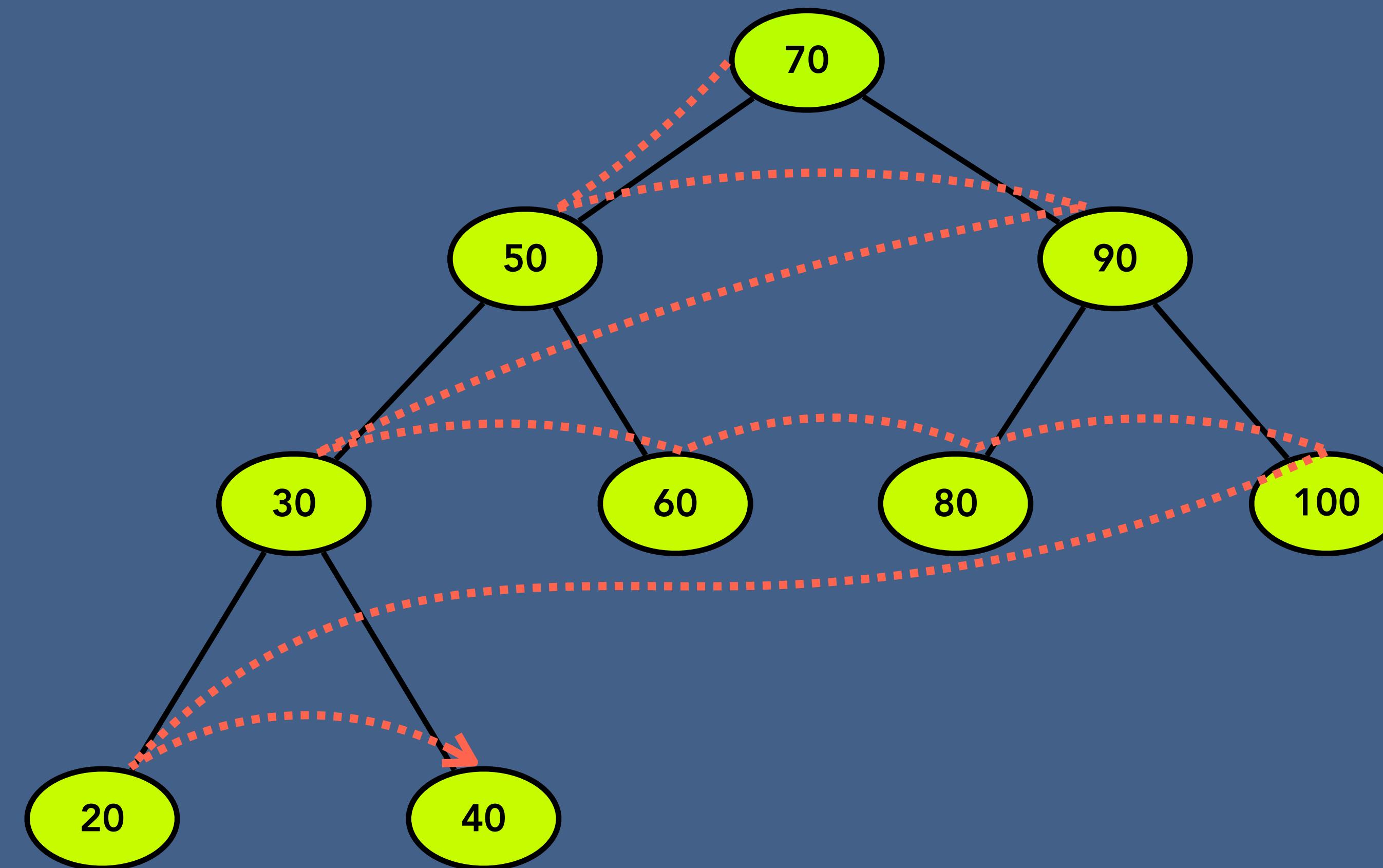
# Binary Search Tree - Post Traversal



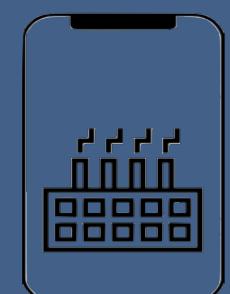
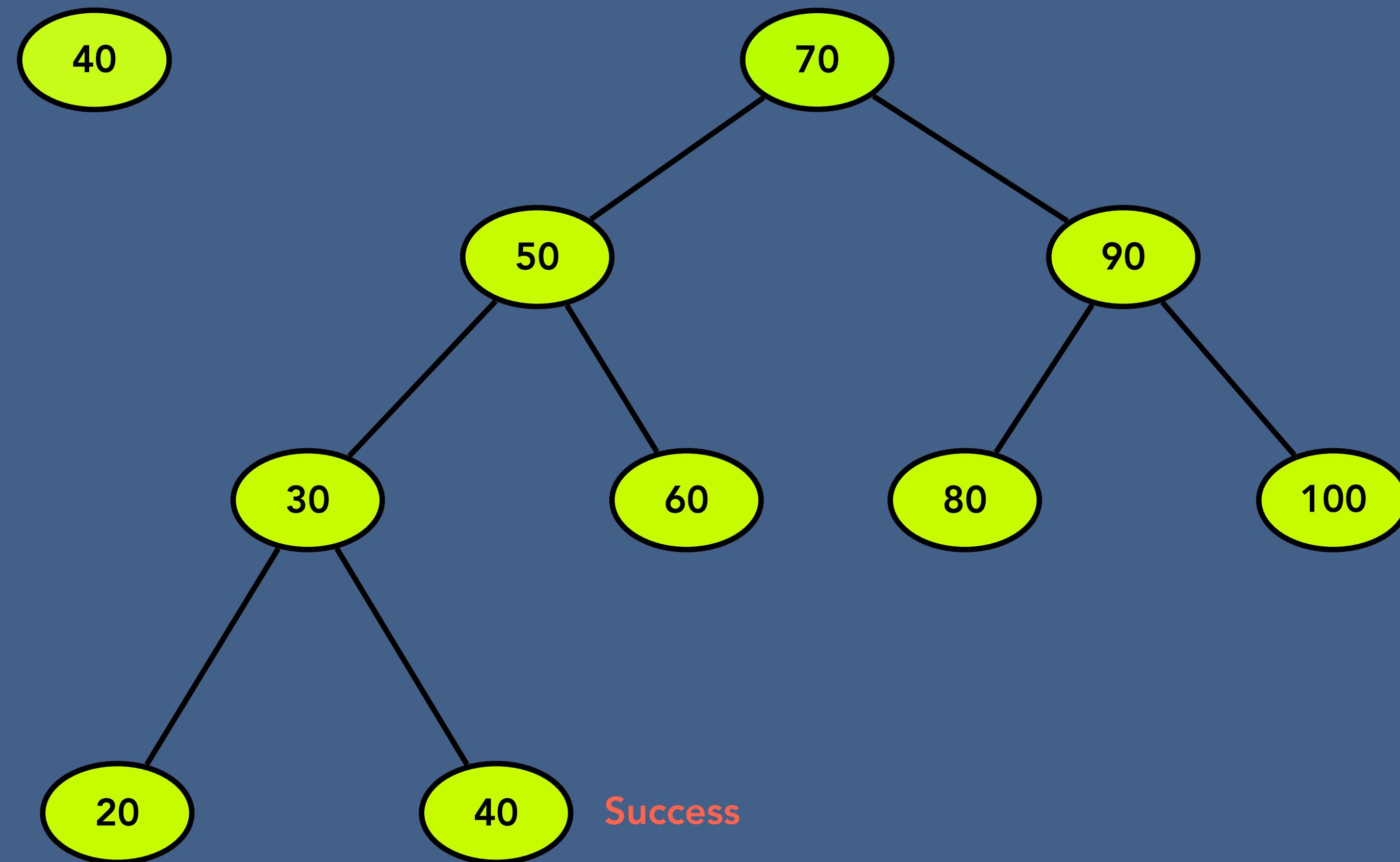
# Binary Search Tree - LevelOrder Traversal



# Binary Search Tree - LevelOrder Traversal



# Binary Search Tree - Search

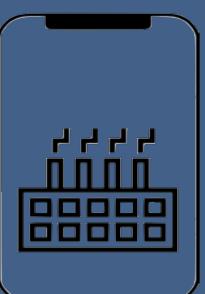
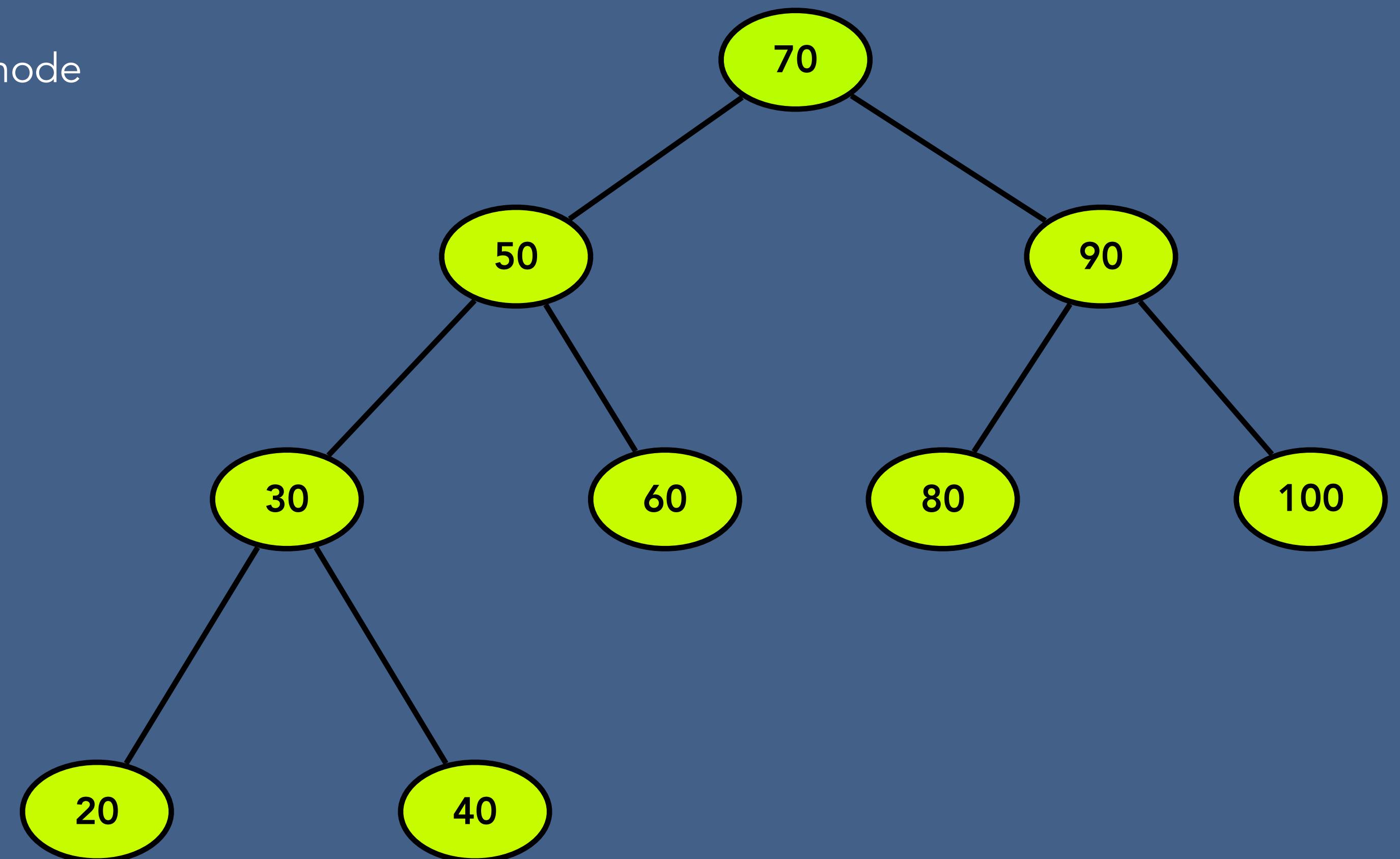


# Binary Search Tree - Delete a Node

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

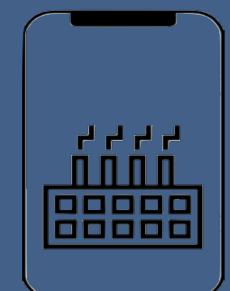
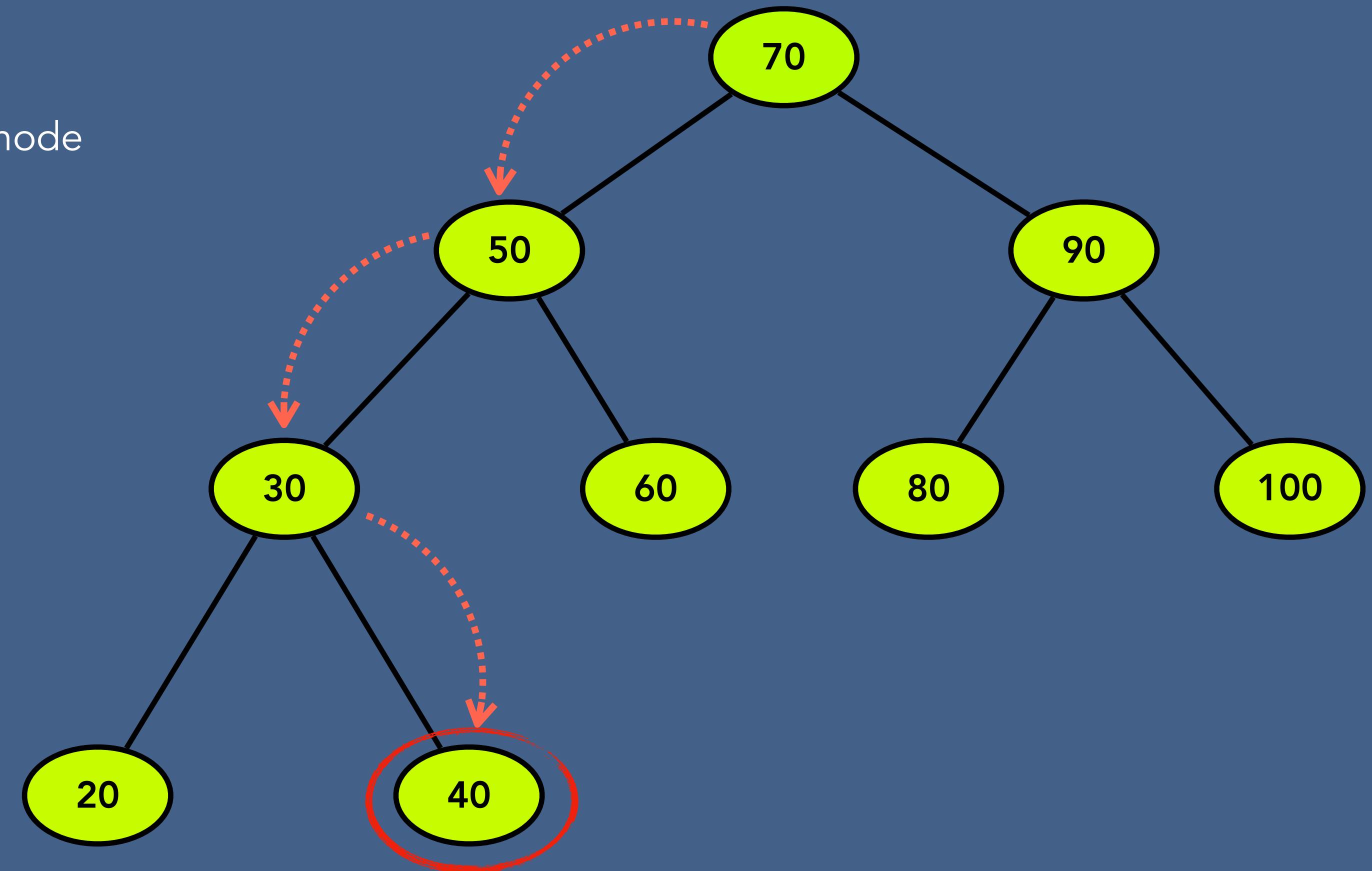
Case 2: The node has one child

Case 3: The node has two children



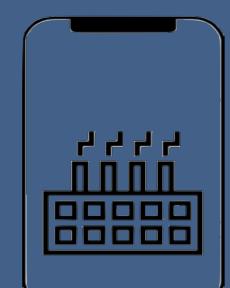
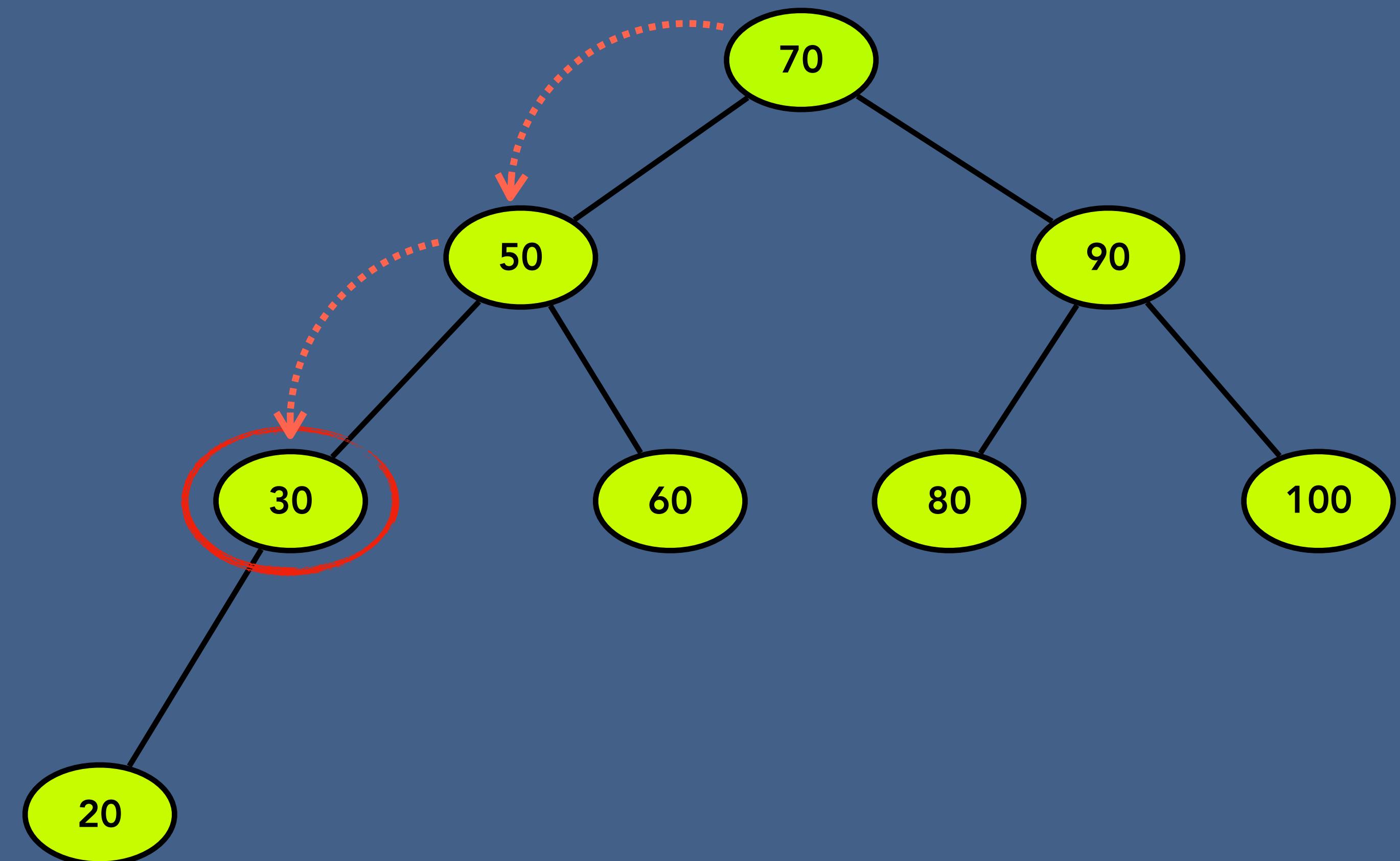
# Binary Search Tree - Delete a Node

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



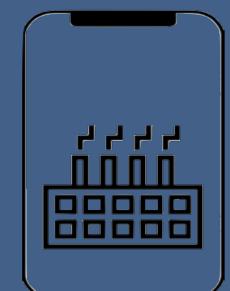
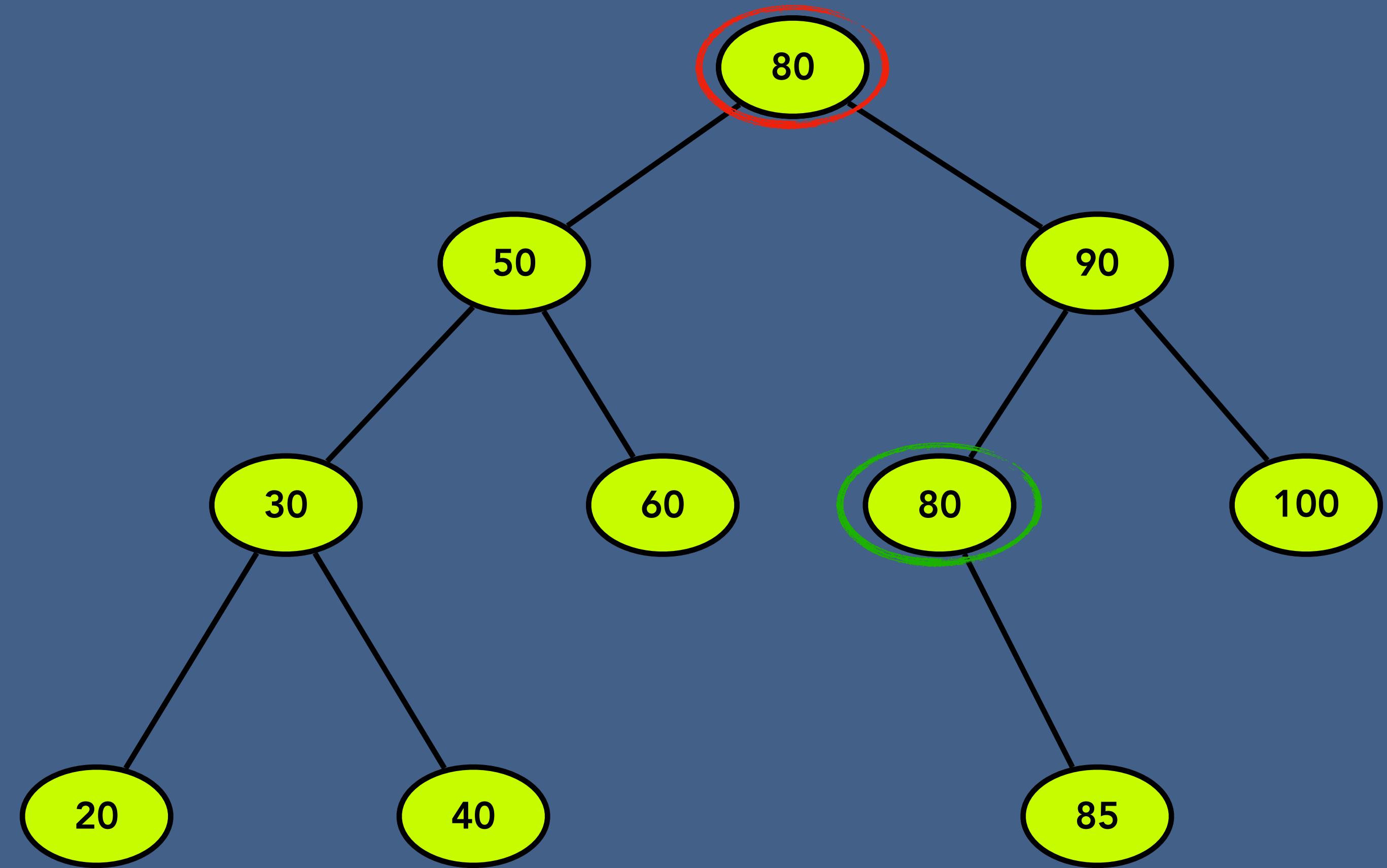
# Binary Search Tree - Delete a Node

Case 2: The node has one child



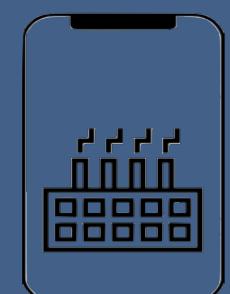
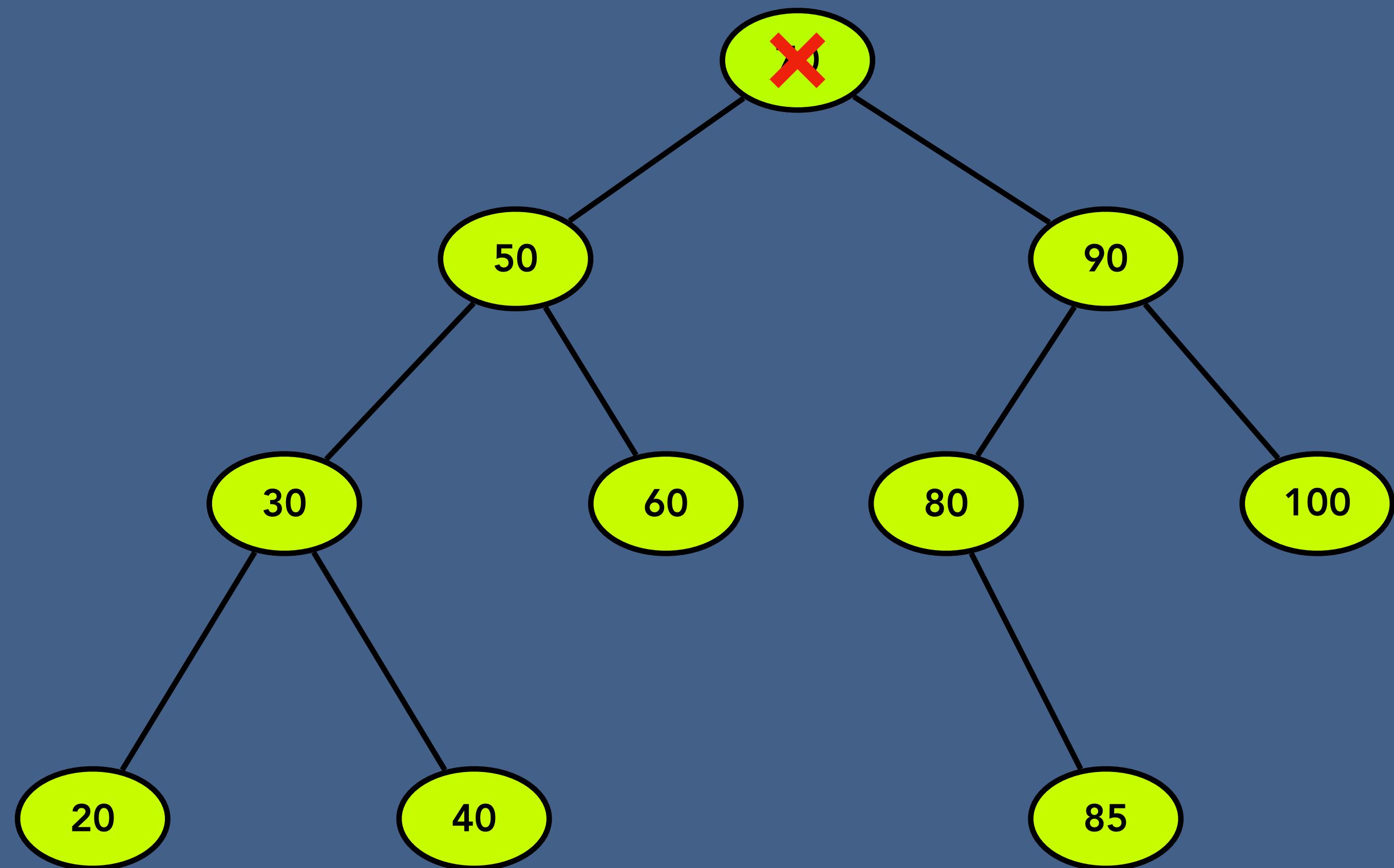
# Binary Search Tree - Delete a Node

Case 3: The node has two children



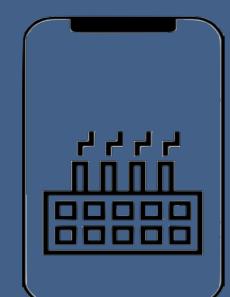
# Binary Search Tree - Delete

root = null

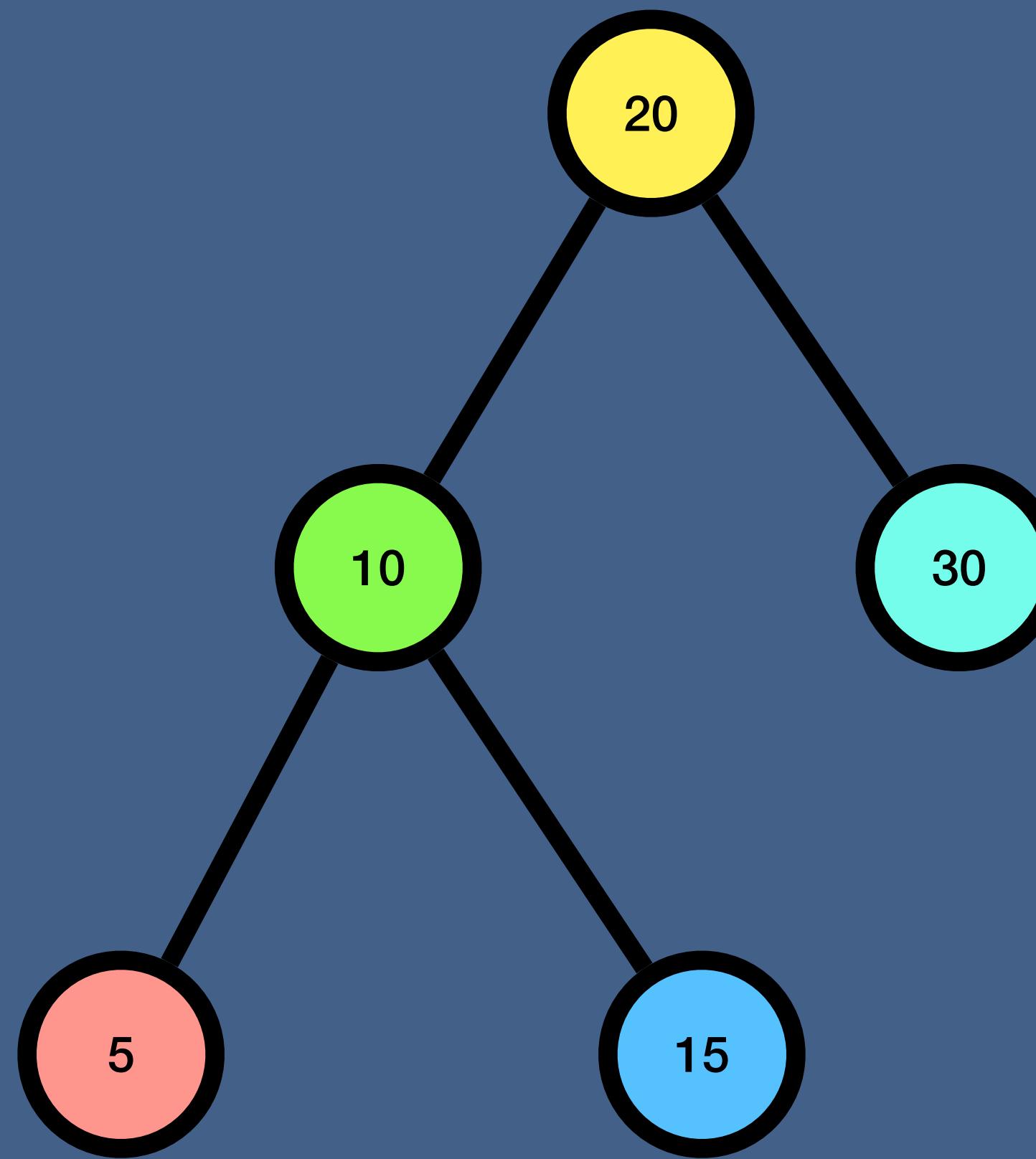


# Binary Search Tree - Time and Space Complexity

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

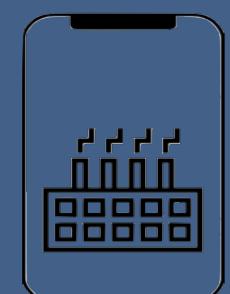
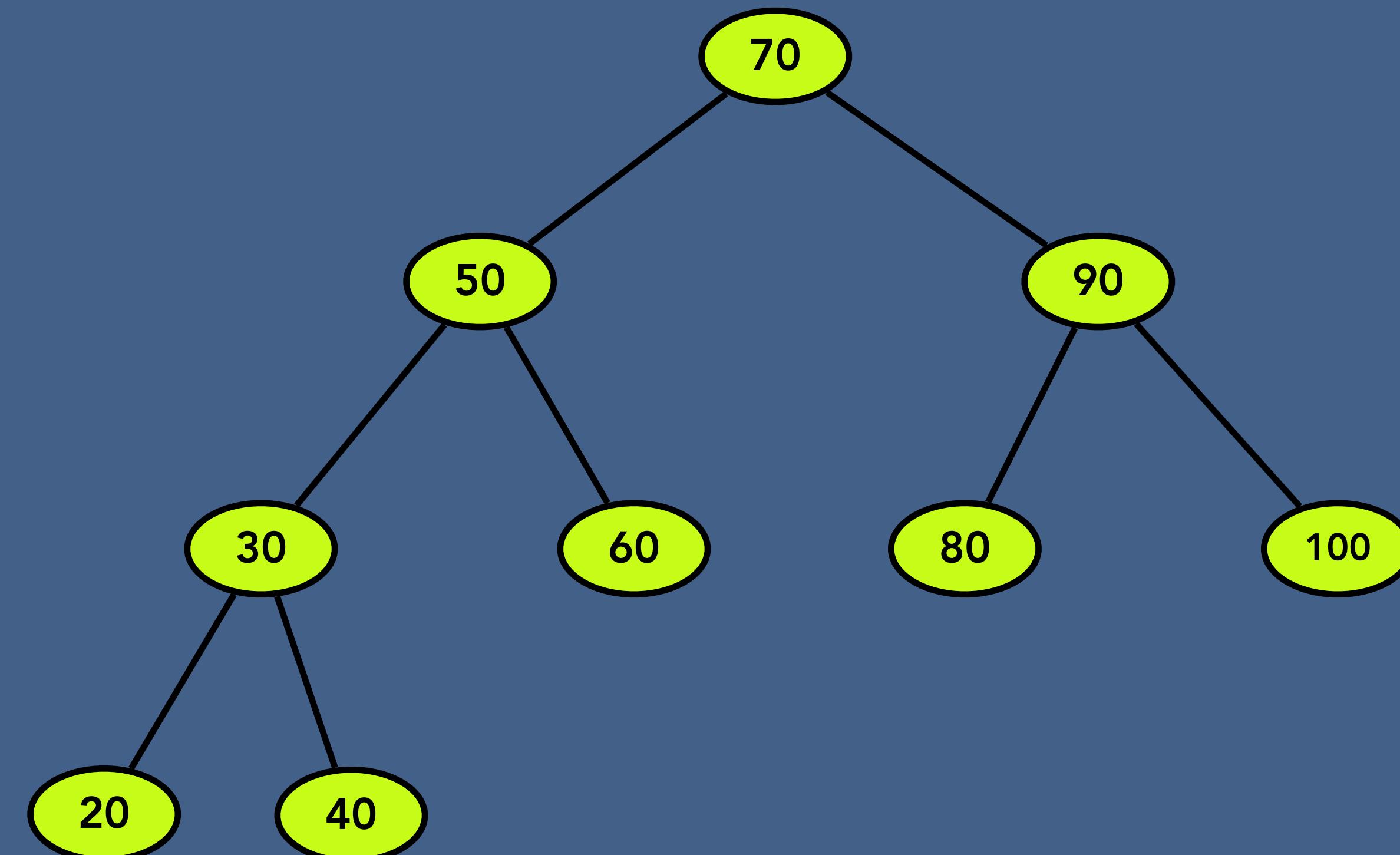


# AVL Tree



# 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

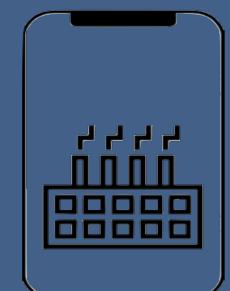
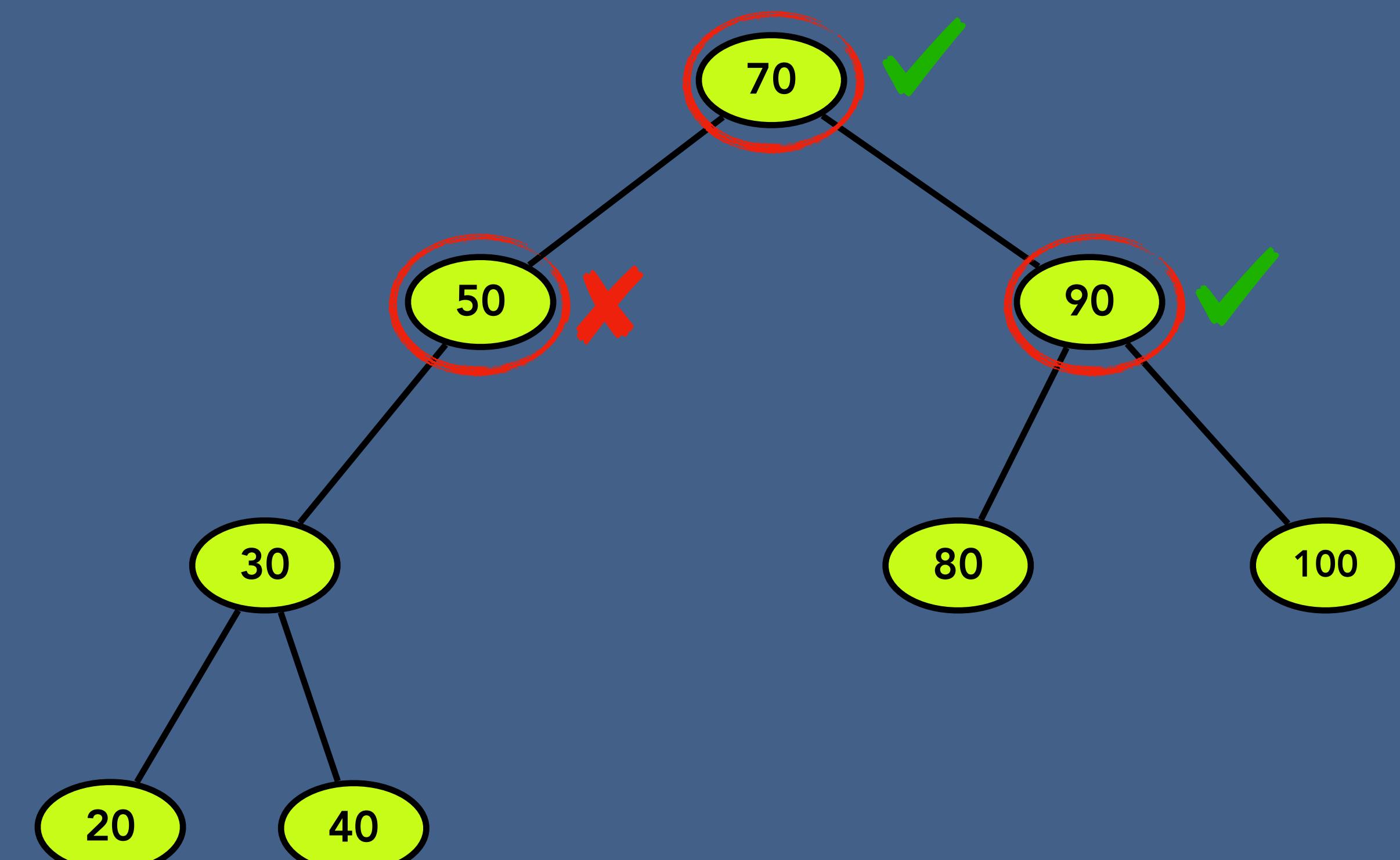
Height of leftSubtree = 2

Height of rightSubtree = 0

difference = 1

difference = 0

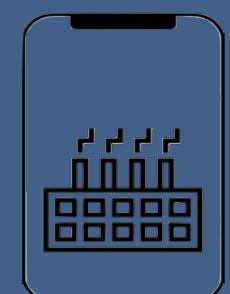
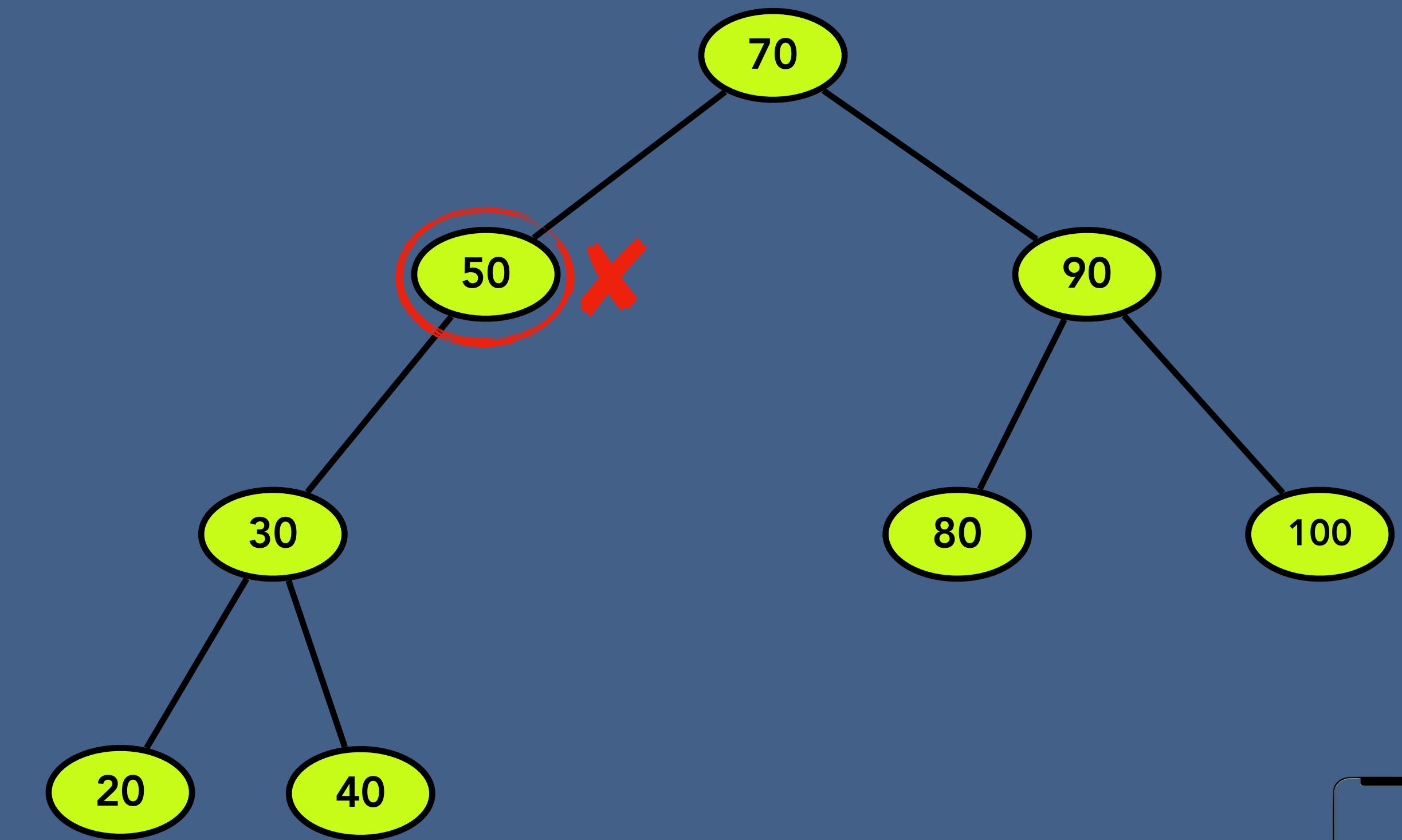
difference = 2



# 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**



# What is an AVL Tree?

## Examples

Height of leftSubtree = 2

Height of rightSubtree = 2

Height of leftSubtree = 1

Height of rightSubtree = 1

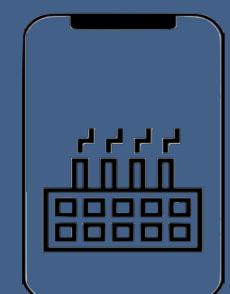
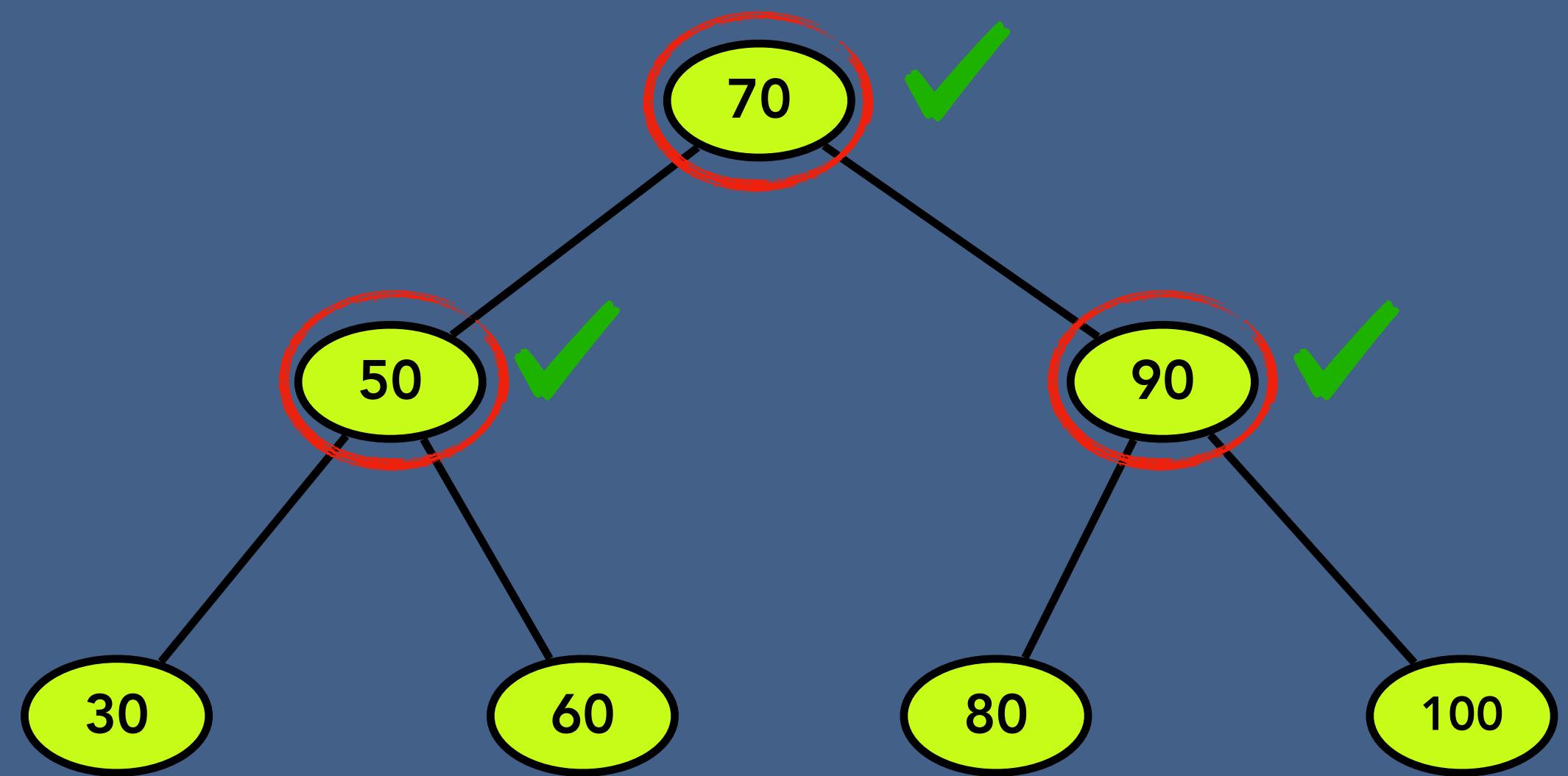
Height of leftSubtree = 1

Height of rightSubtree = 1

difference = 0

difference = 0

difference = 0



# What is an AVL Tree?

## 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

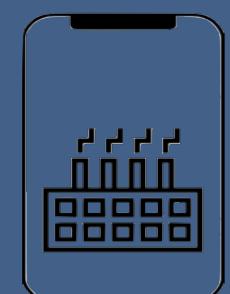
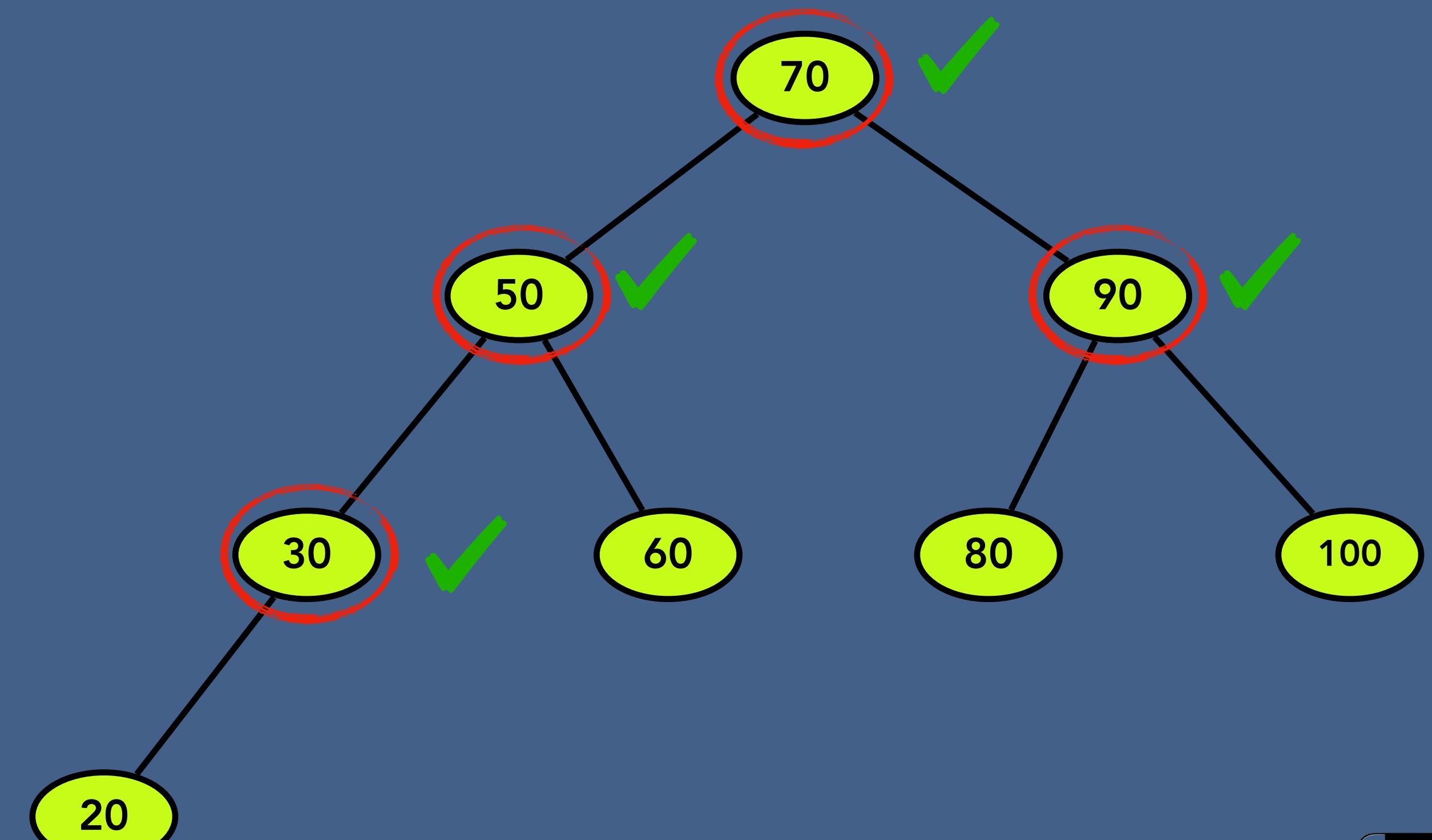
Height of rightSubtree = 0

difference = 1

difference = 1

difference = 0

difference = 1



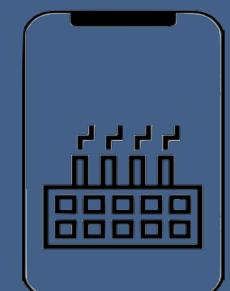
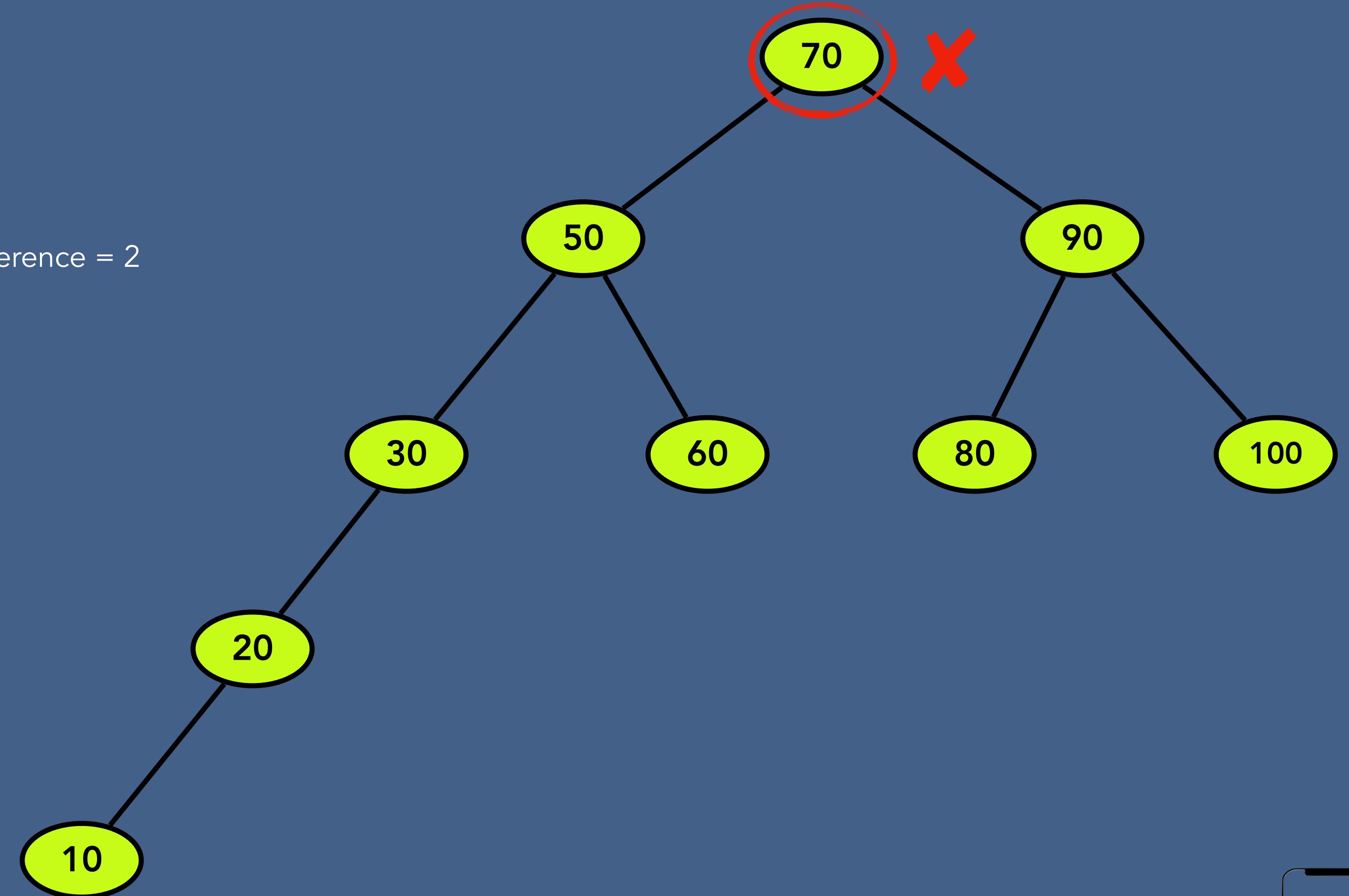
# What is an AVL Tree?

## Examples

Height of leftSubtree = 4

Height of rightSubtree = 2

difference = 2

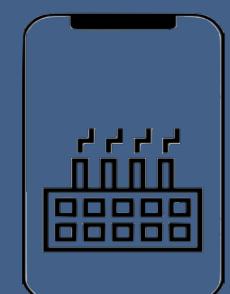
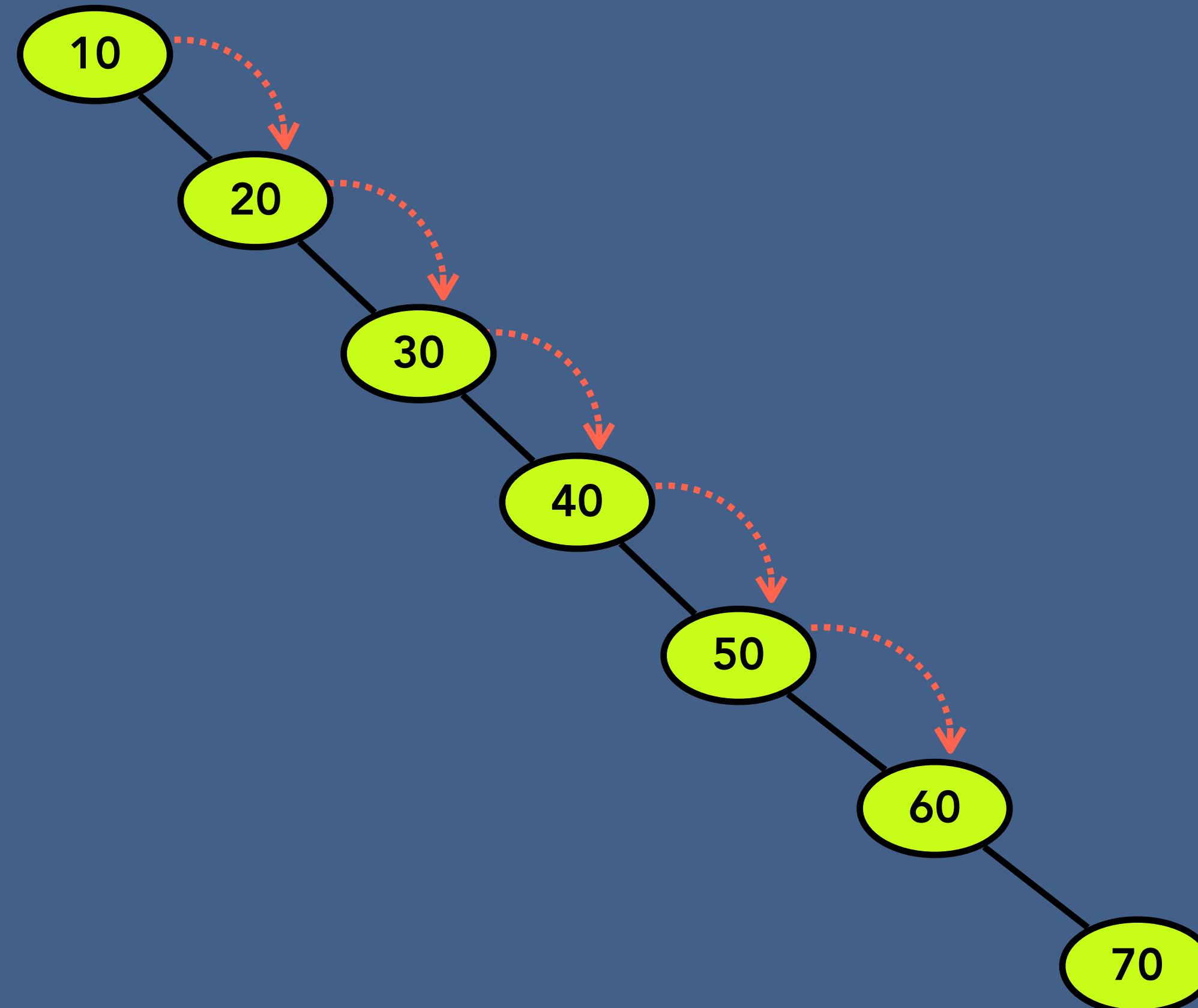


# What do we need AVL Tree?

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

Search for 60

Time complexity is  $O(N)$

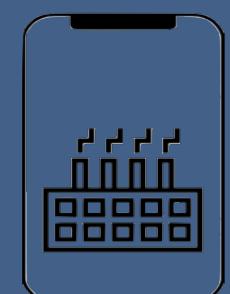
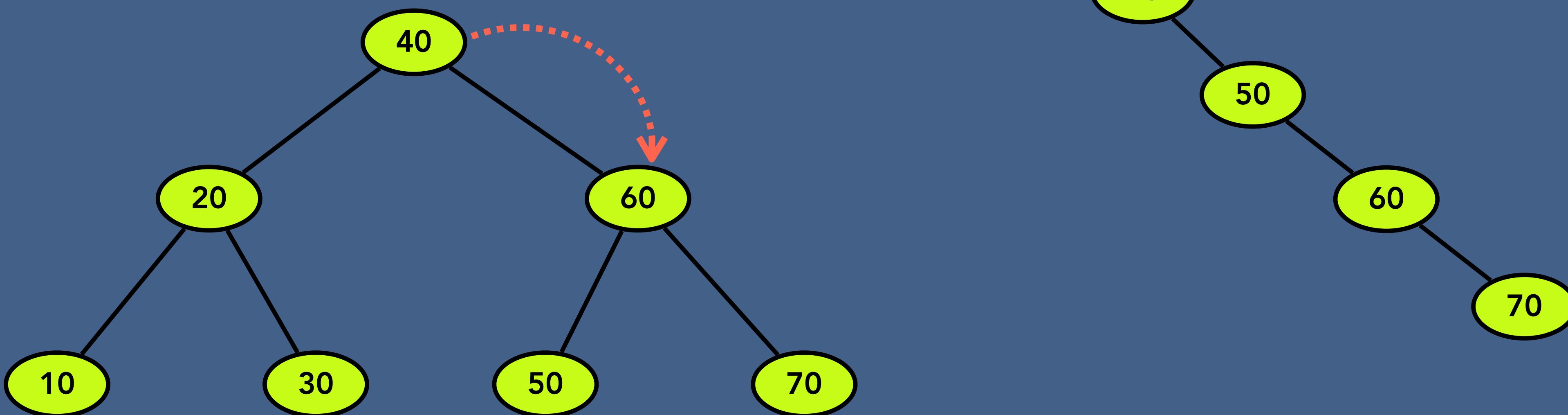


# What do we need AVL Tree?

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

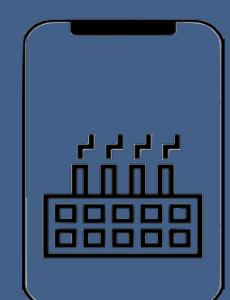
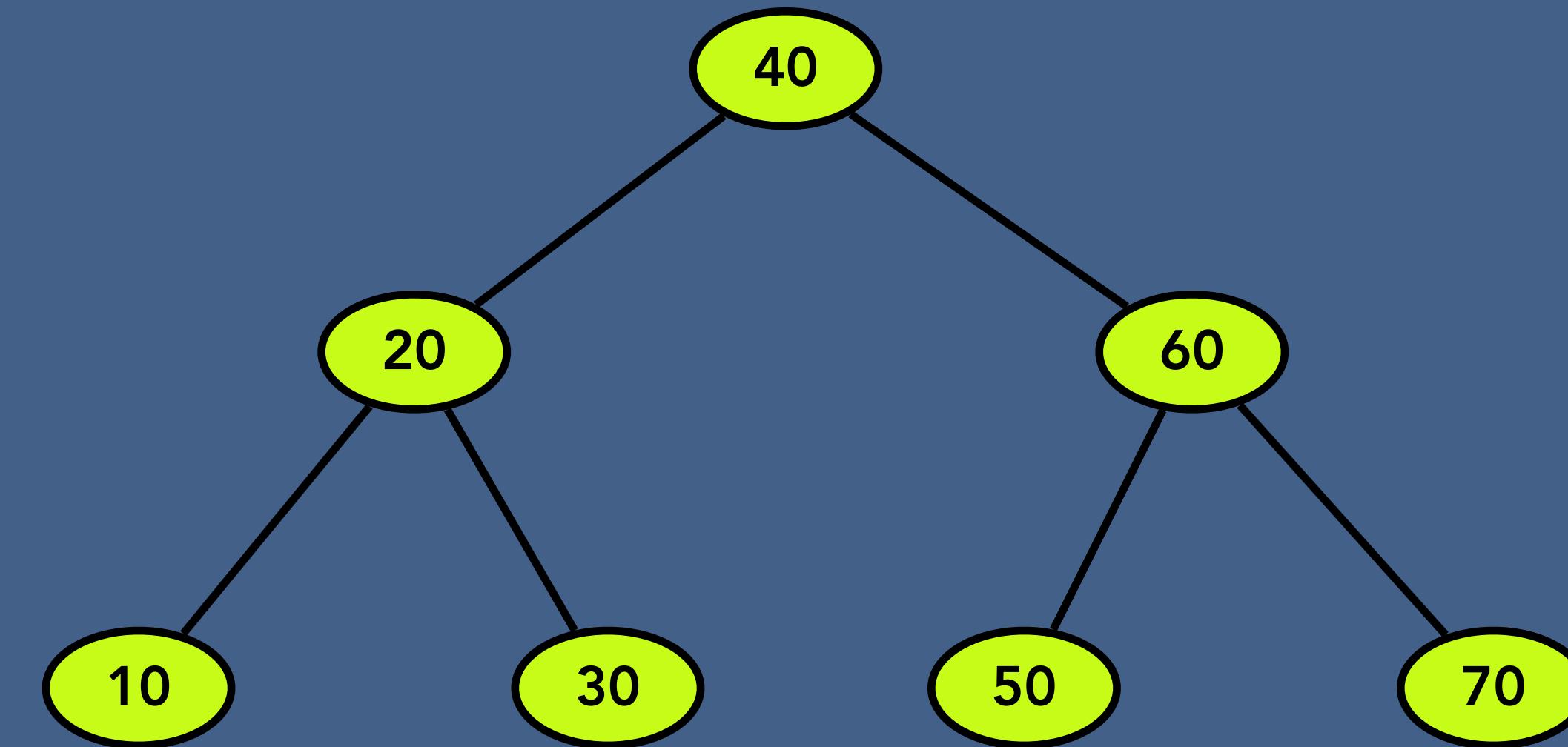
Search for 60

Time complexity is  $O(\log N)$



# Common Operations on AVL Tree

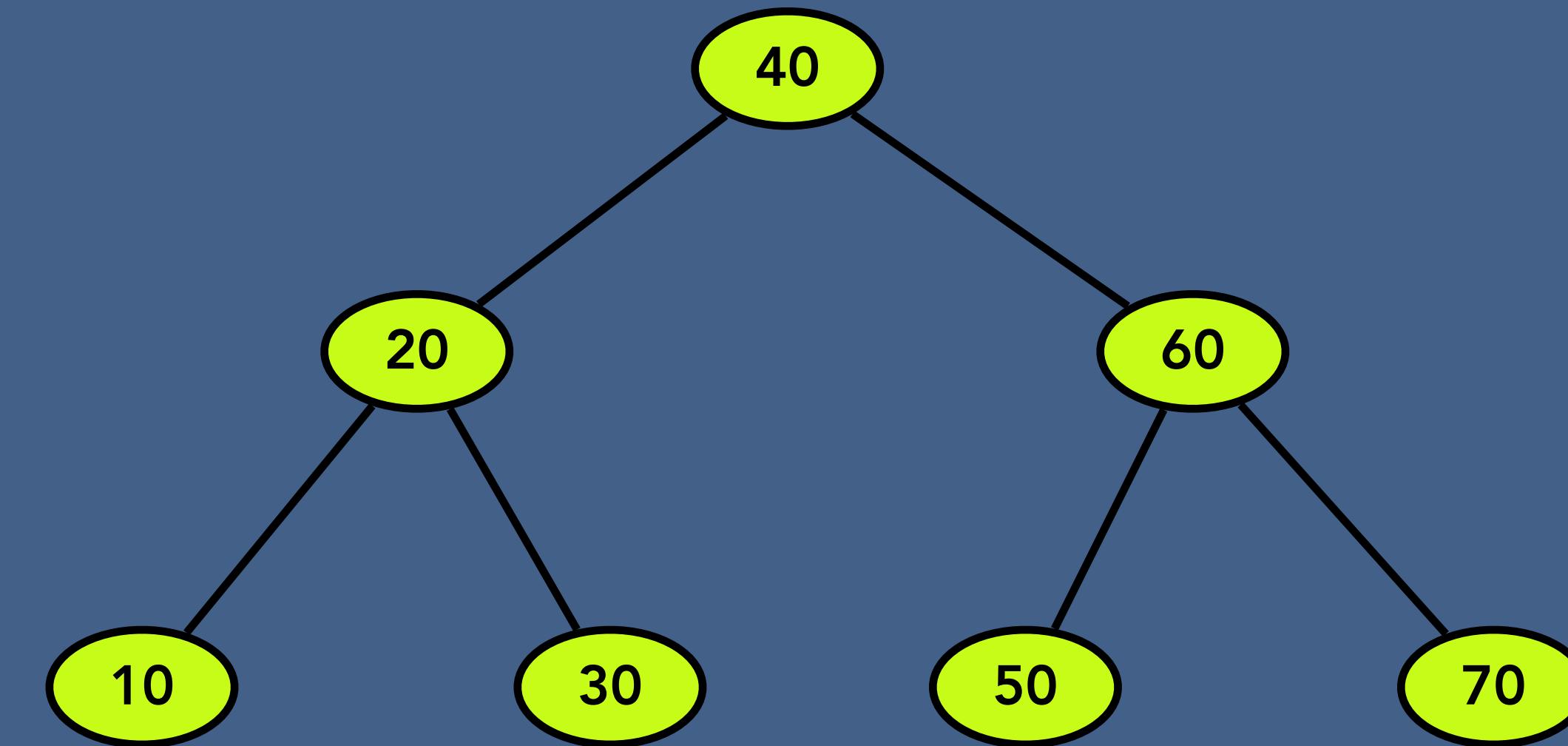
- Creation of AVL trees,
- Search for a node in AVL trees
- Traverse all nodes in AVL trees
- Insert a node in AVL trees
- Delete a node from AVL trees
- Delete the entire AVL trees



# Common Operations on AVL Tree

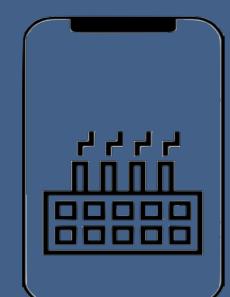
```
newAVL = AVL()
```

```
rootNode = Null
```

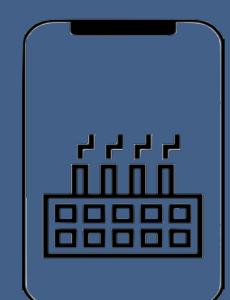
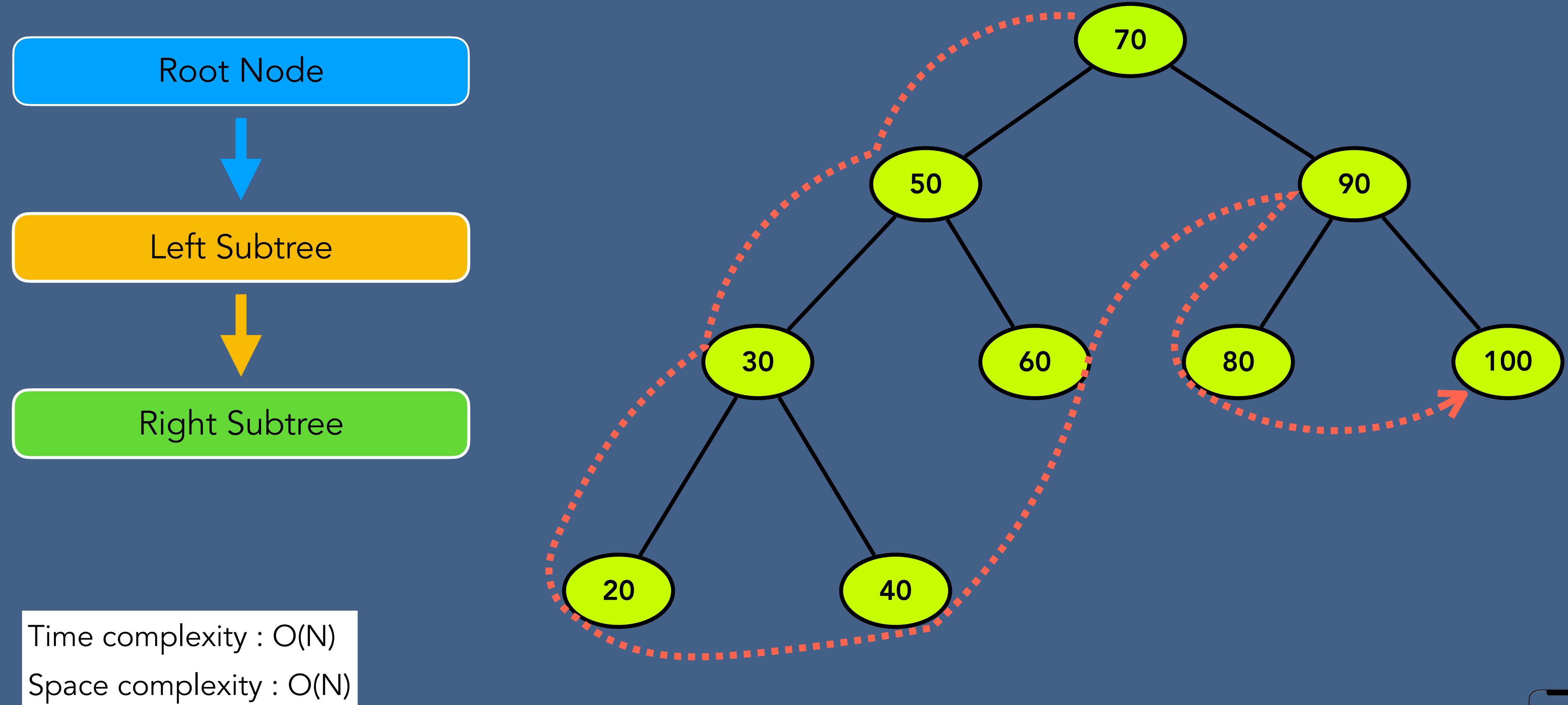


Time complexity : O(1)

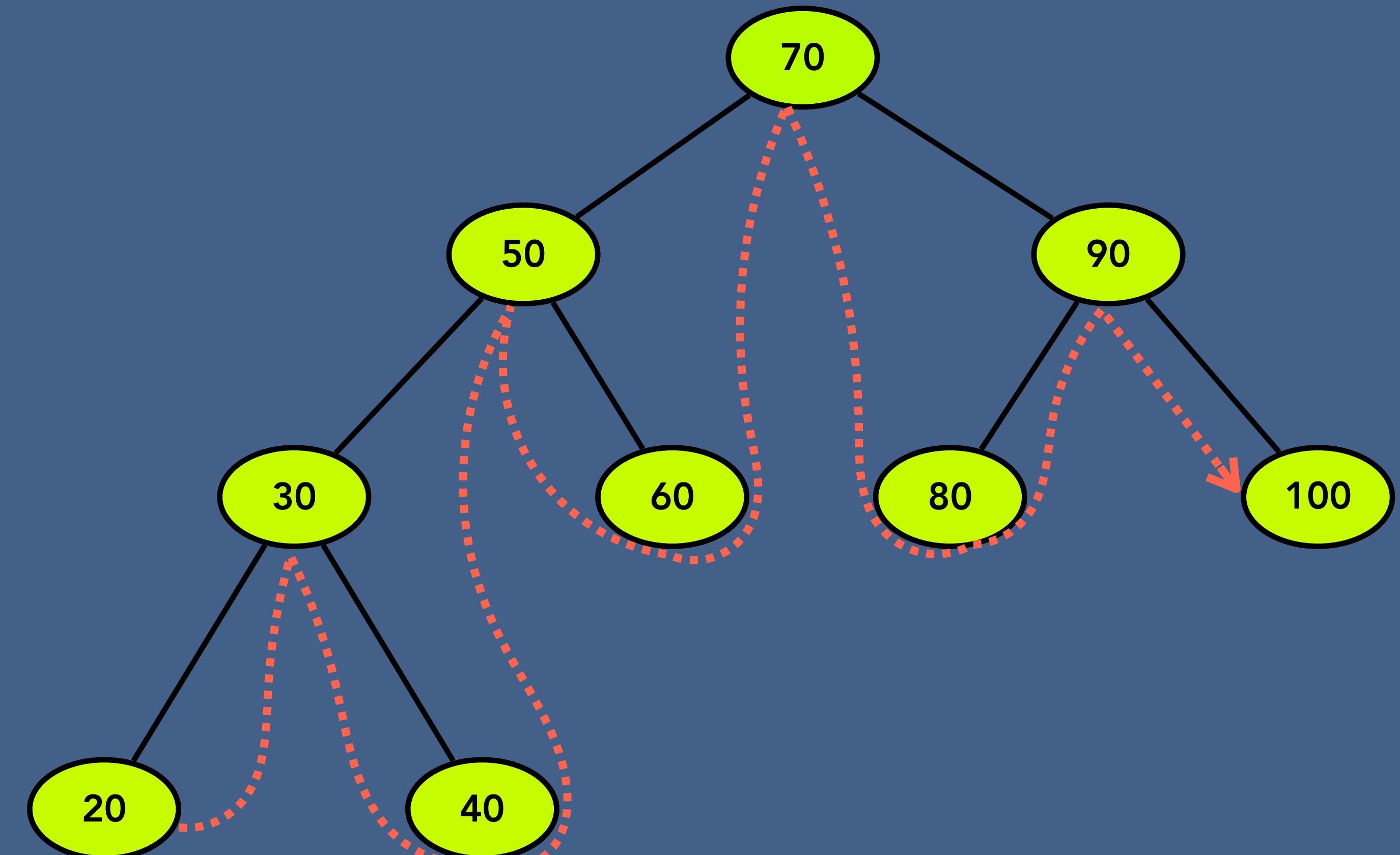
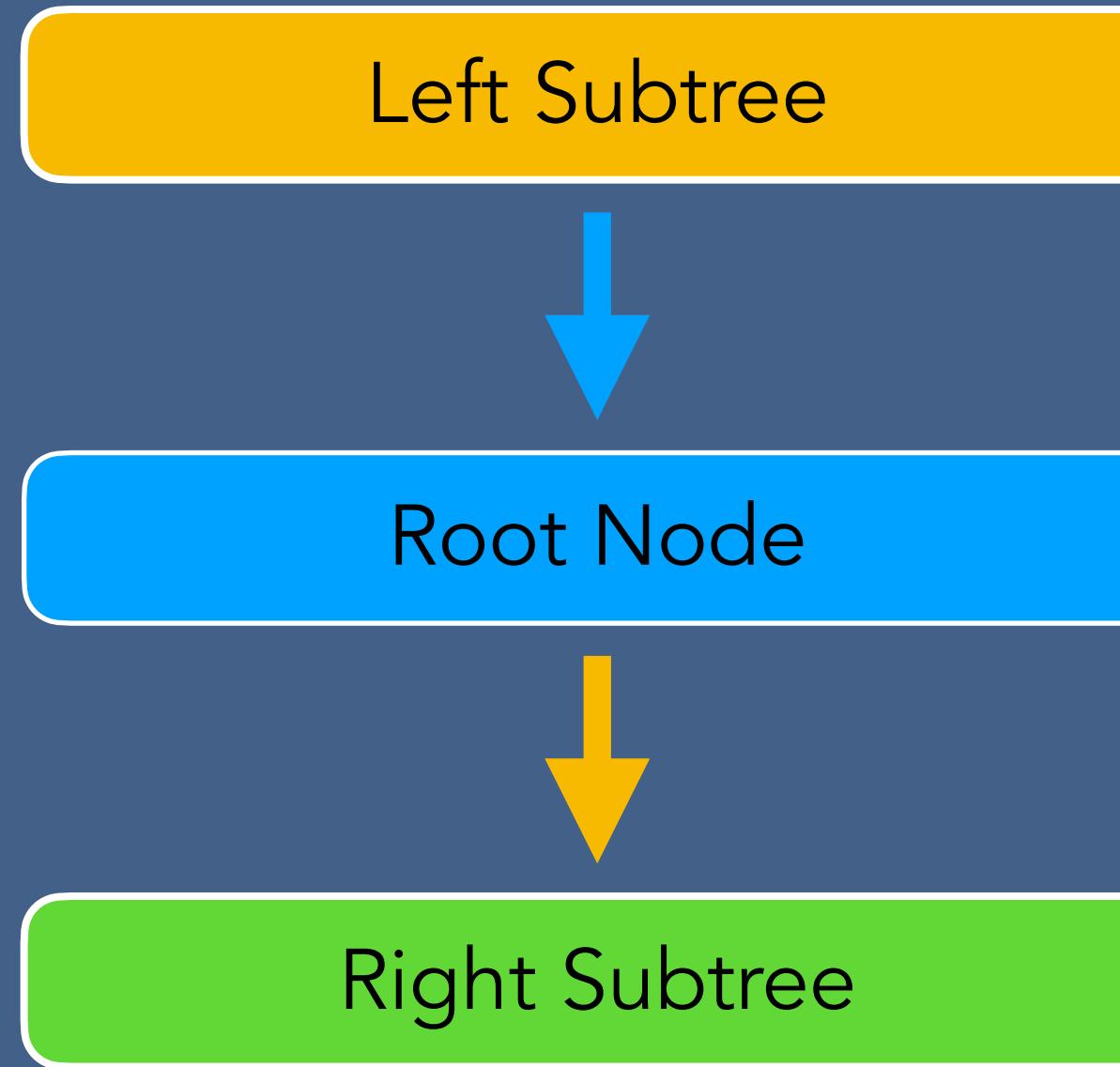
Space complexity : O(1)



# AVL Tree - PreOrder Traversal

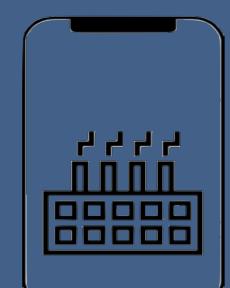


# AVL Tree- InOrder Traversal

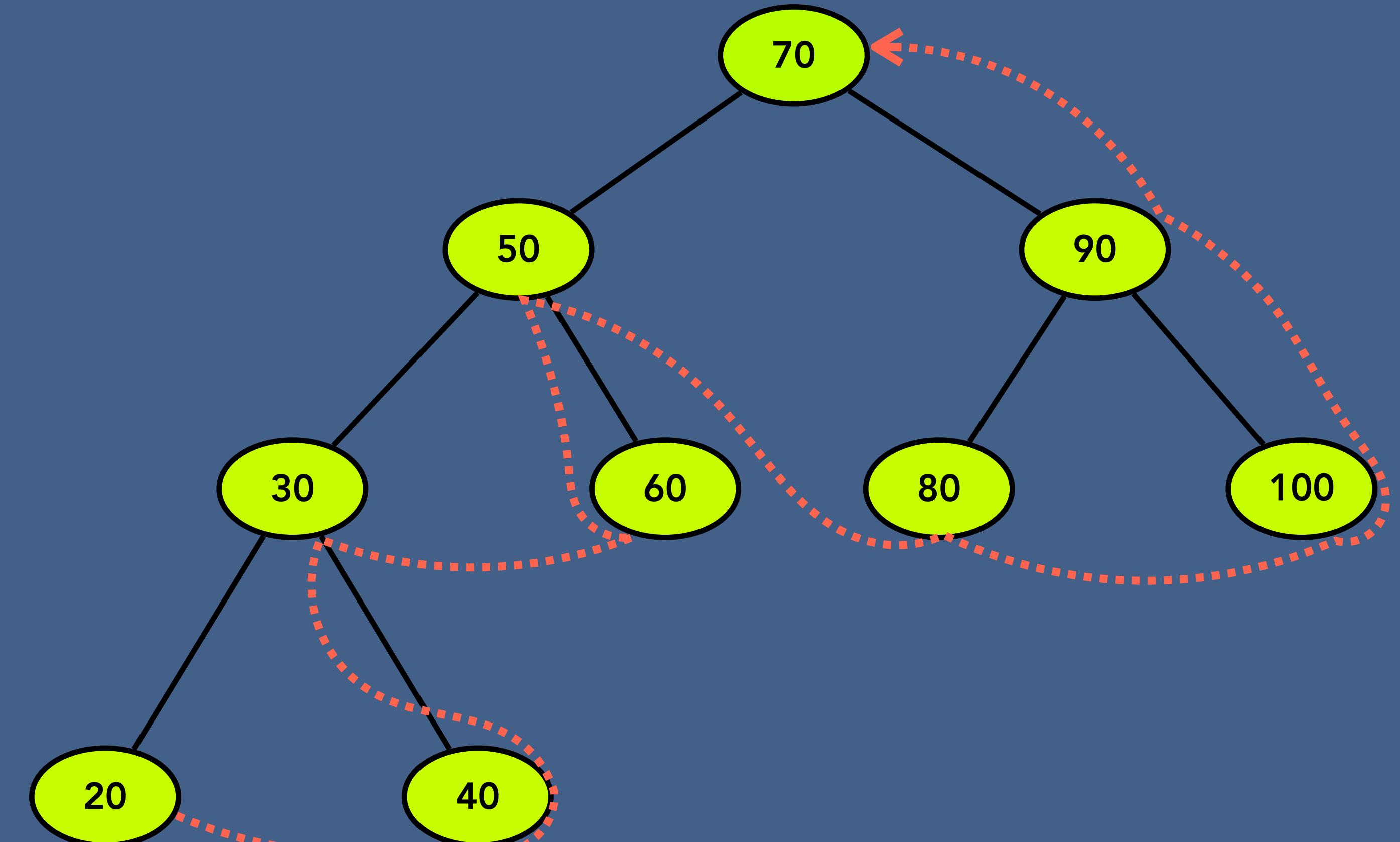
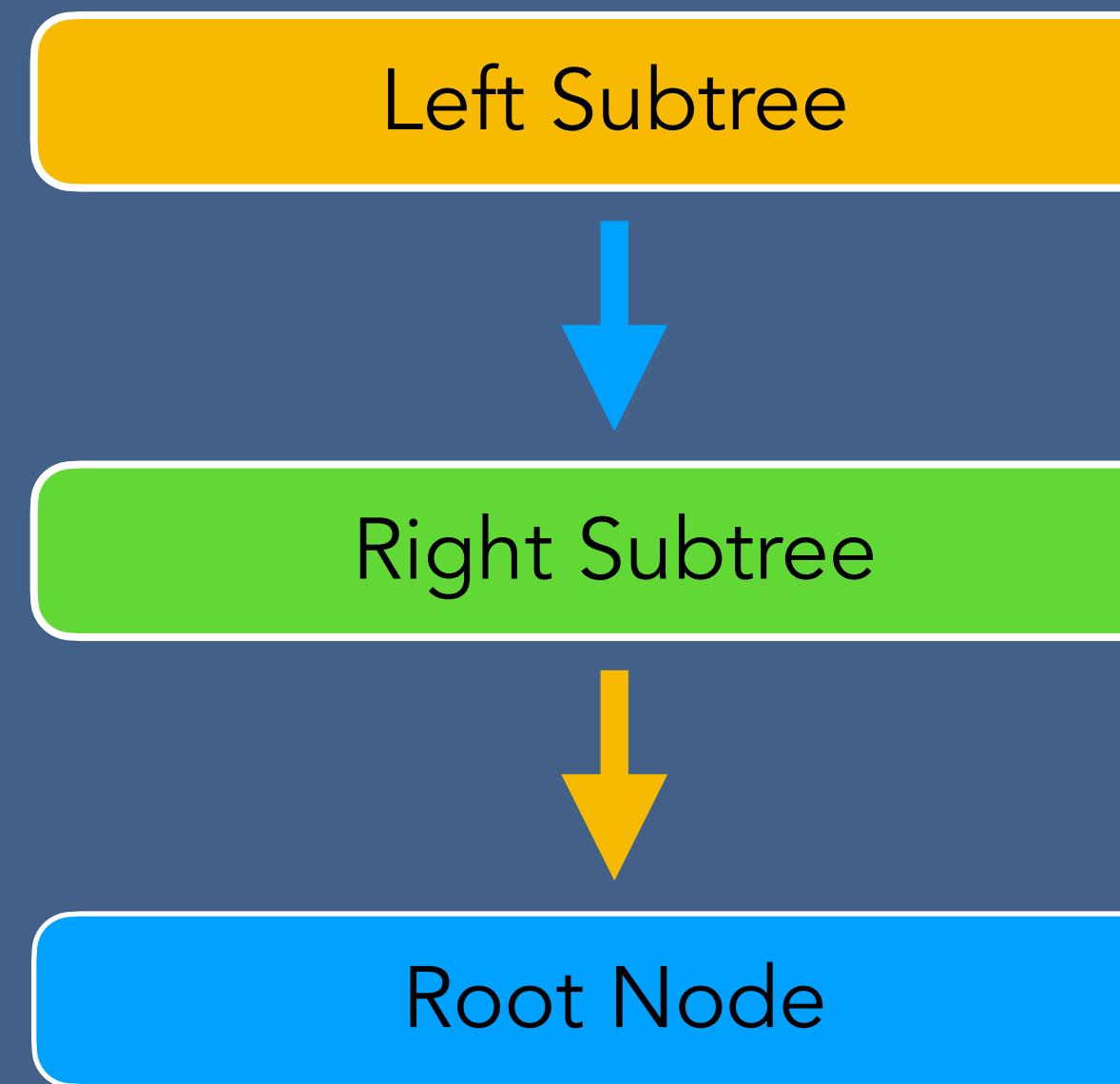


Time complexity :  $O(N)$

Space complexity :  $O(N)$

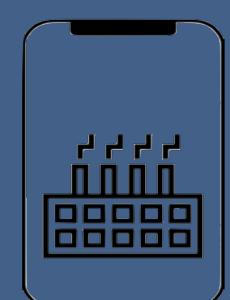


# AVL Tree - Post Traversal

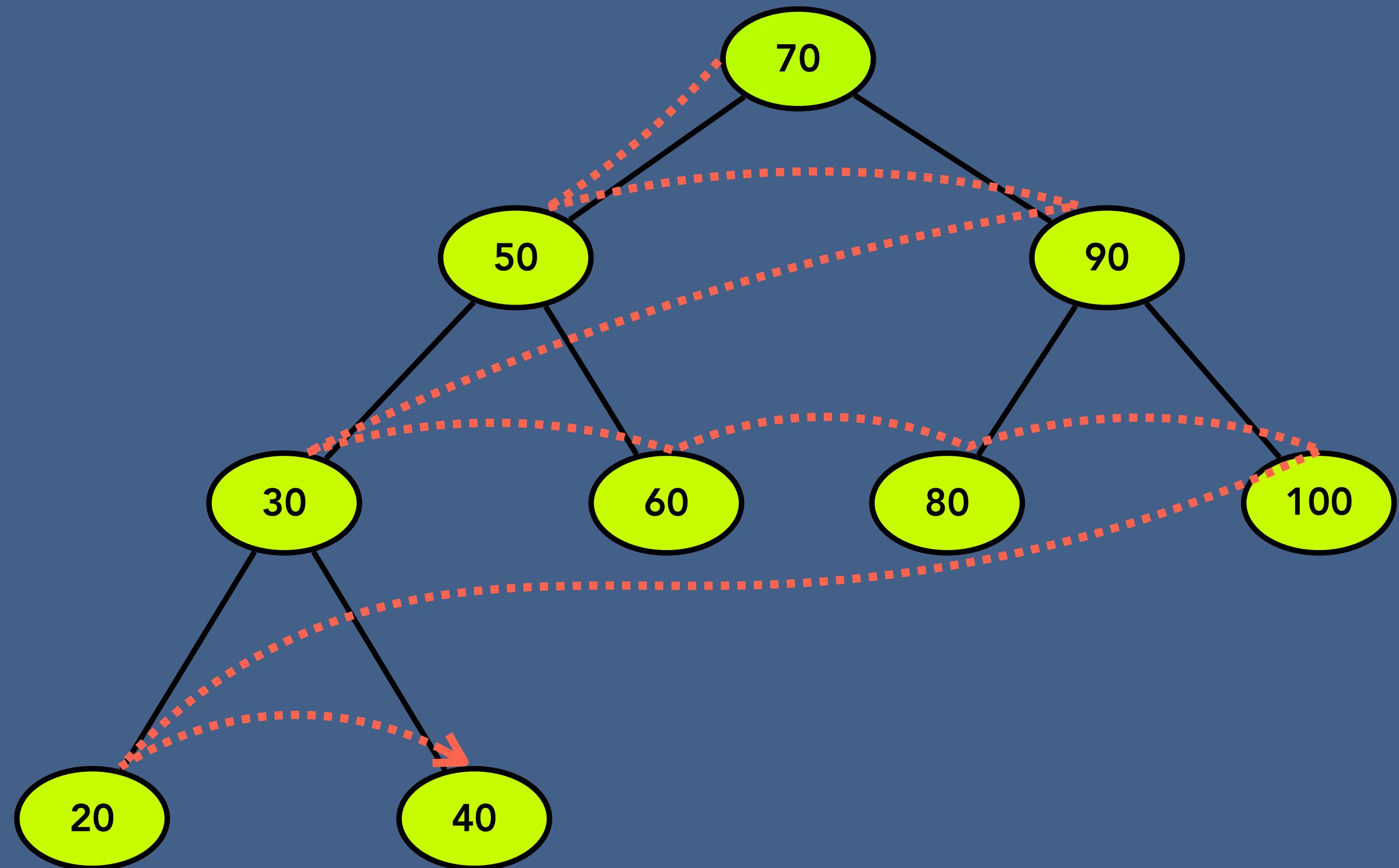


Time complexity :  $O(N)$

Space complexity :  $O(N)$

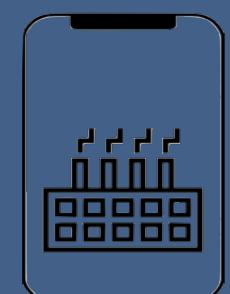


# AVL Tree - LevelOrder Traversal

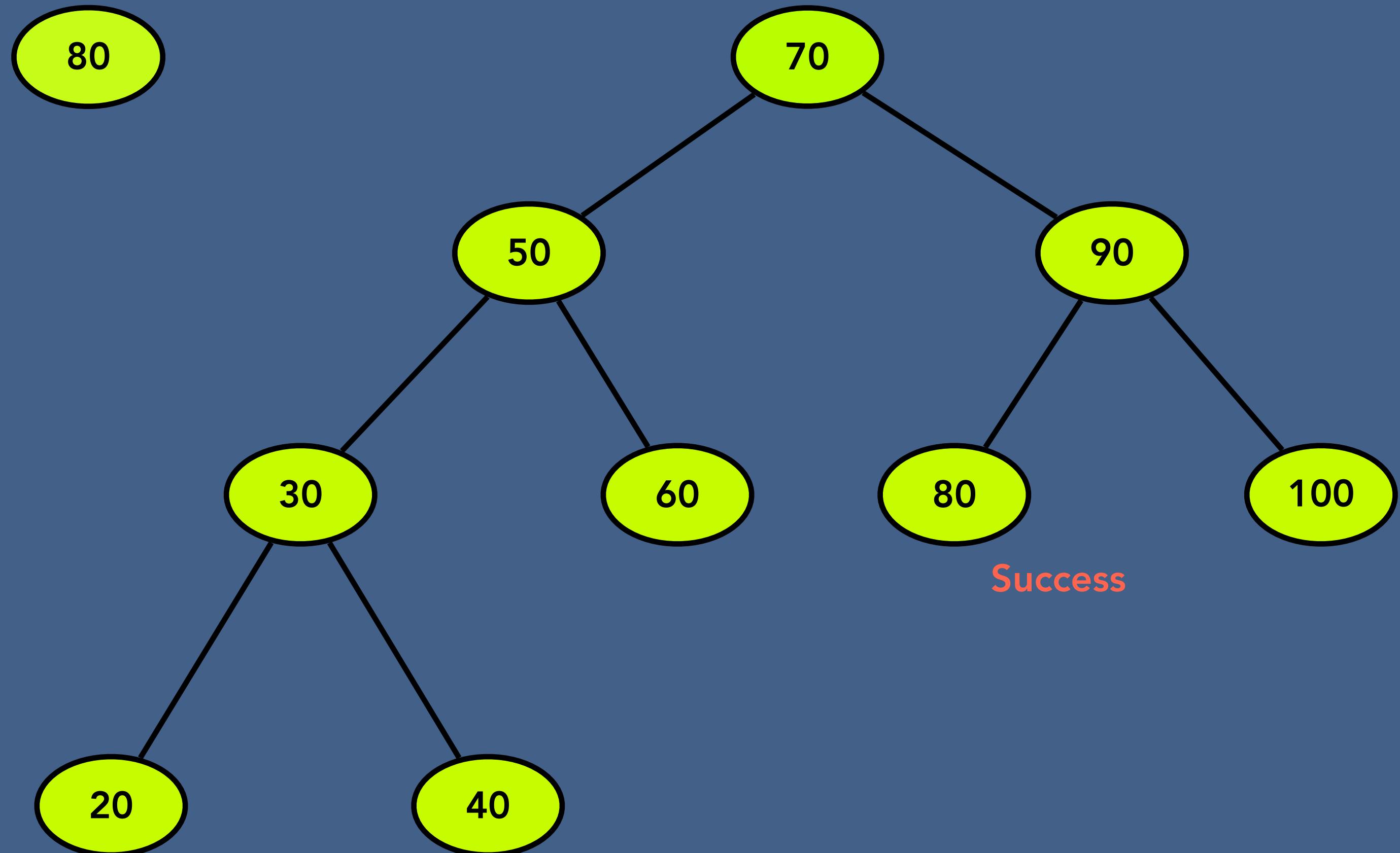


Time complexity :  $O(N)$

Space complexity :  $O(N)$

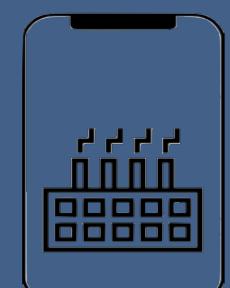


# AVL Tree - Search

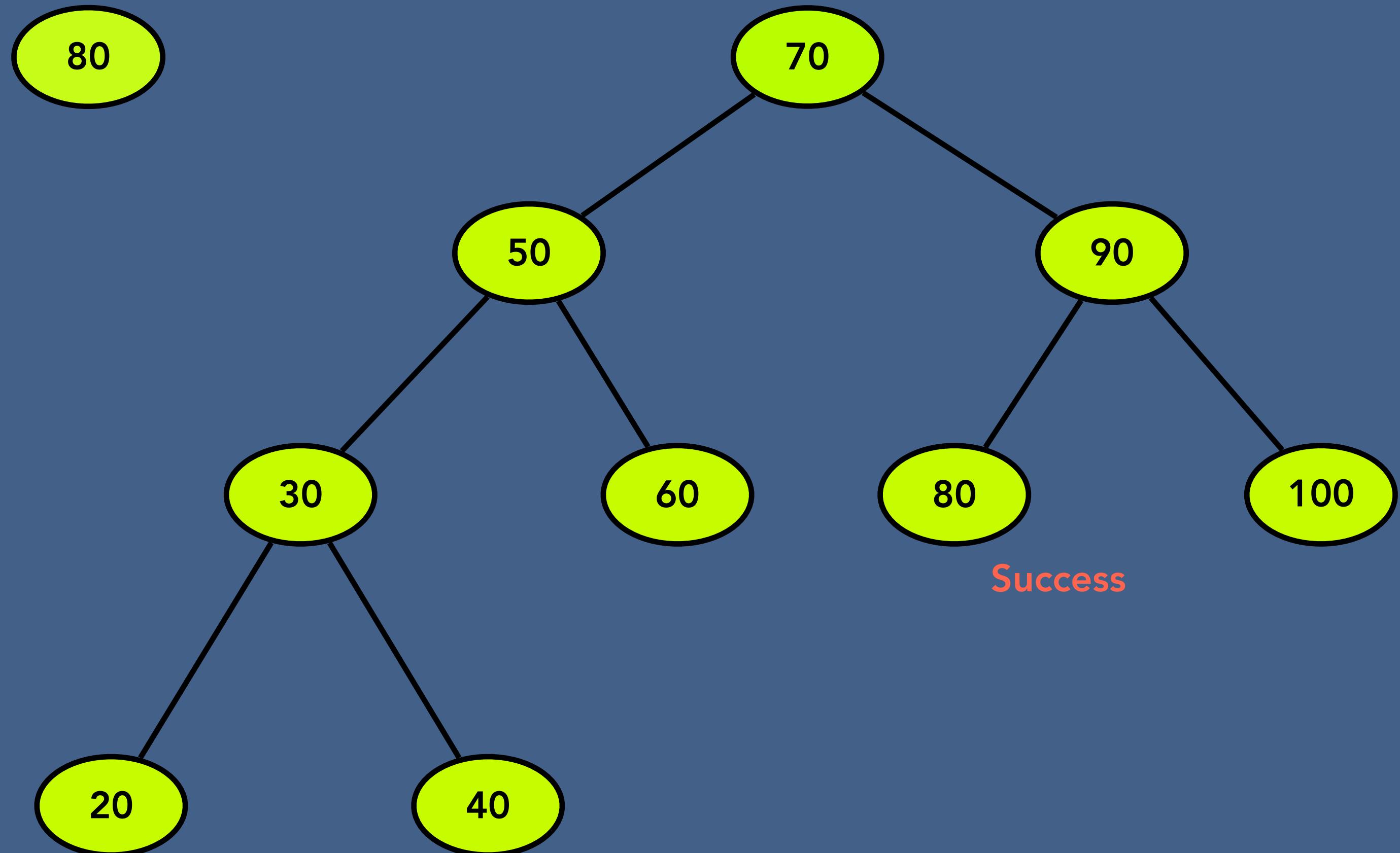


Time complexity :  $O(\log N)$

Space complexity :  $O(\log N)$

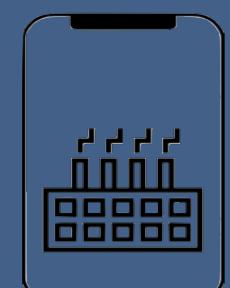


# AVL Tree - Search



Time complexity :  $O(\log N)$

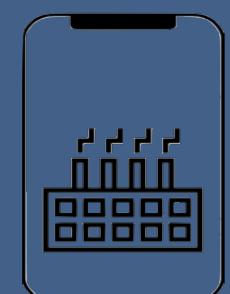
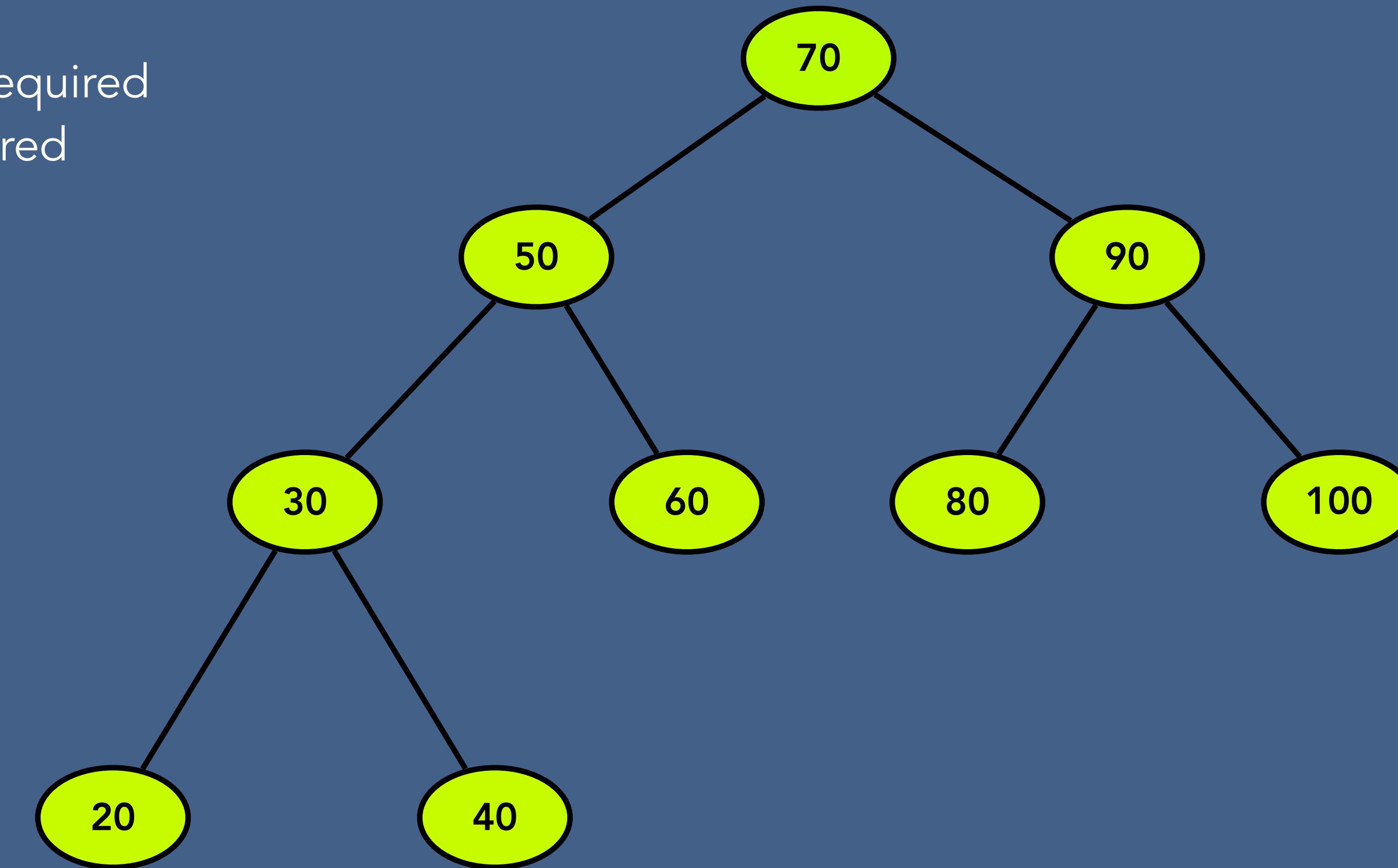
Space complexity :  $O(\log N)$



# AVL Tree - Insert a Node

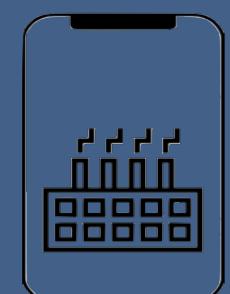
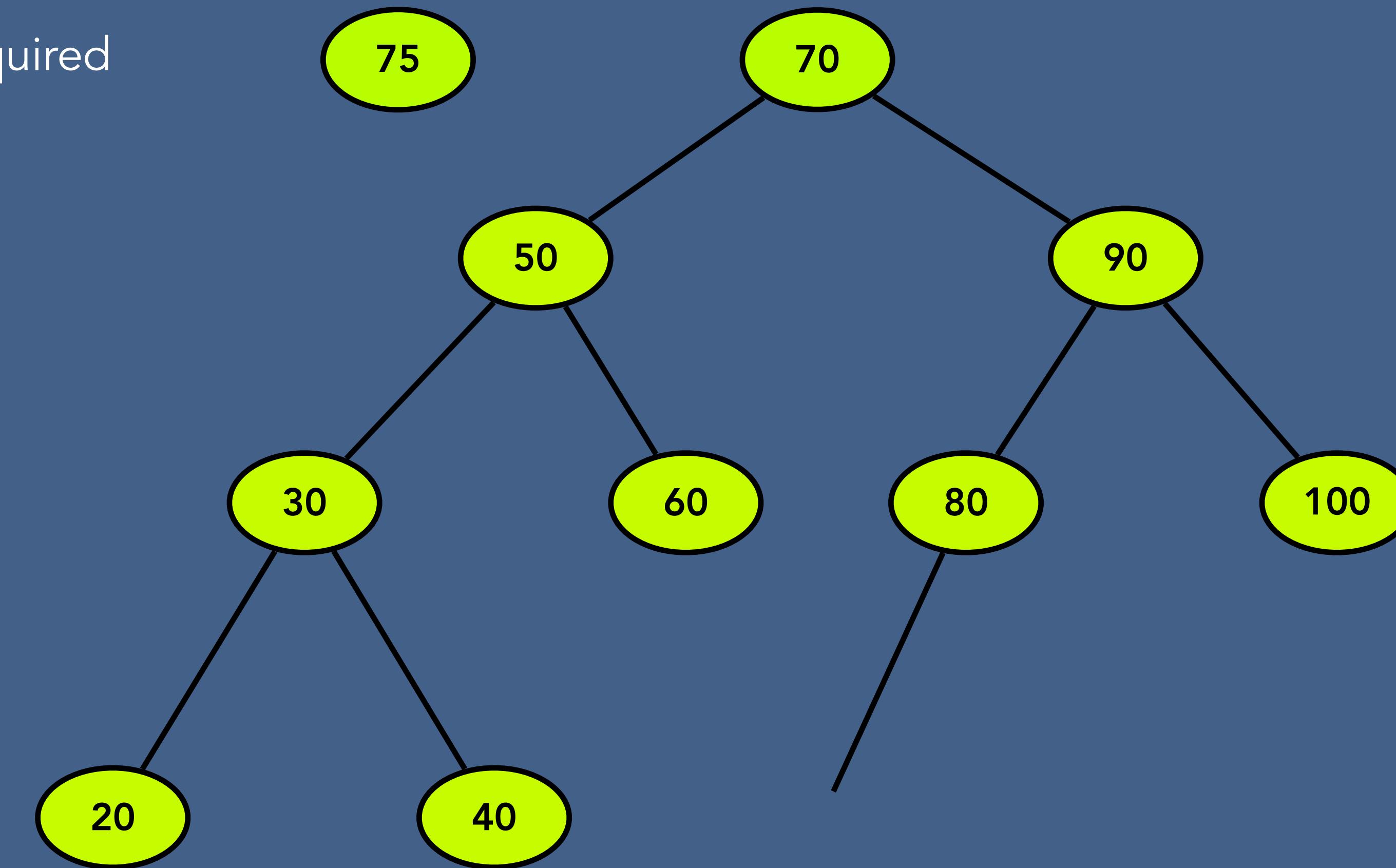
Case 1 : Rotation is not required

Case 2 : Rotation is required



# AVL Tree - Insert a Node

Case 1 : Rotation is not required



# AVL Tree - Insert a Node

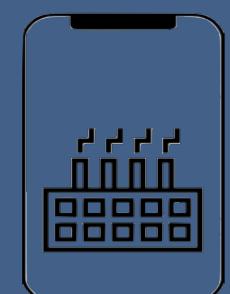
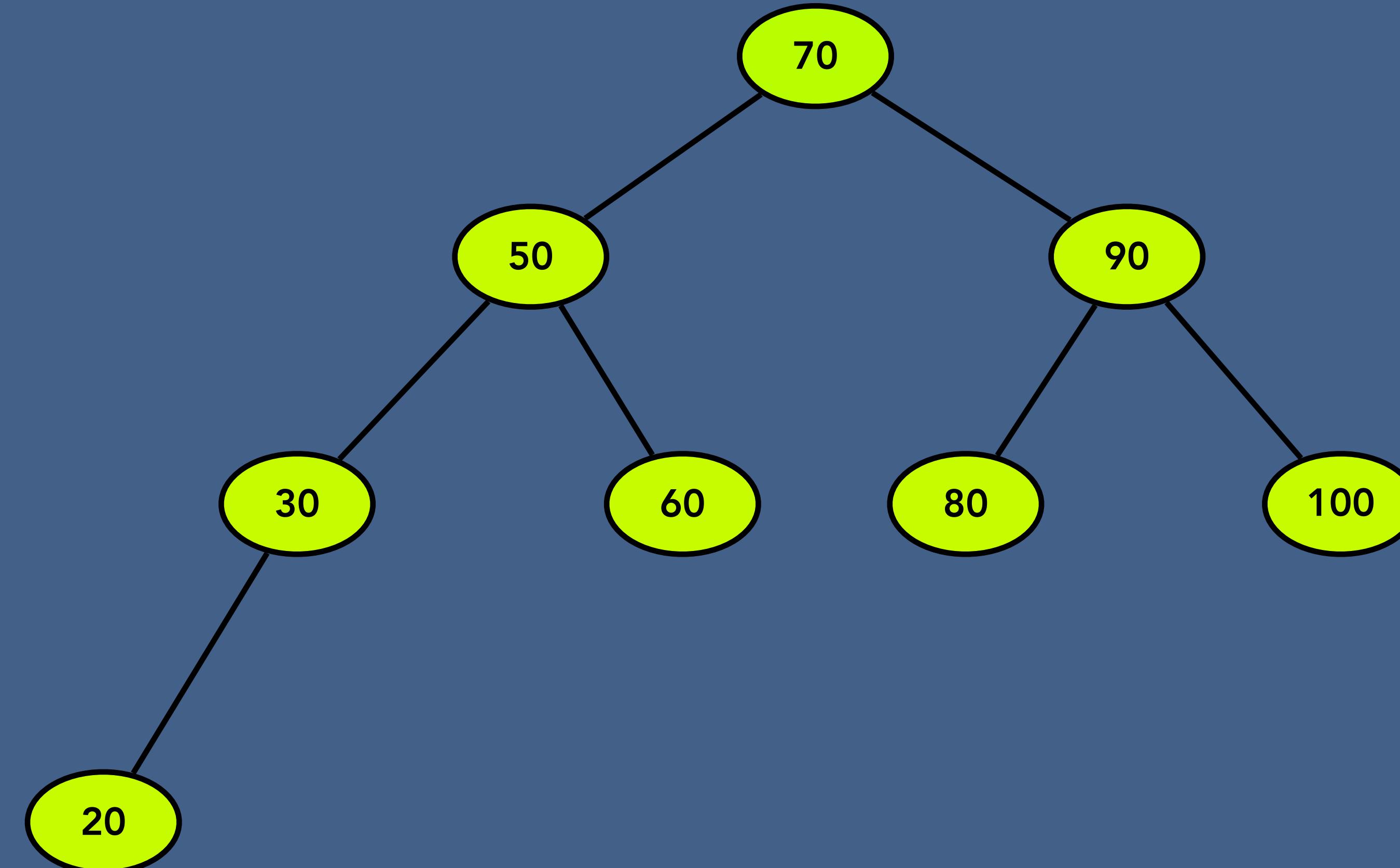
Case 2 : Rotation is required

LL - left left condition

LR - left right condition

RR - right right condition

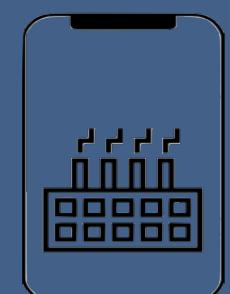
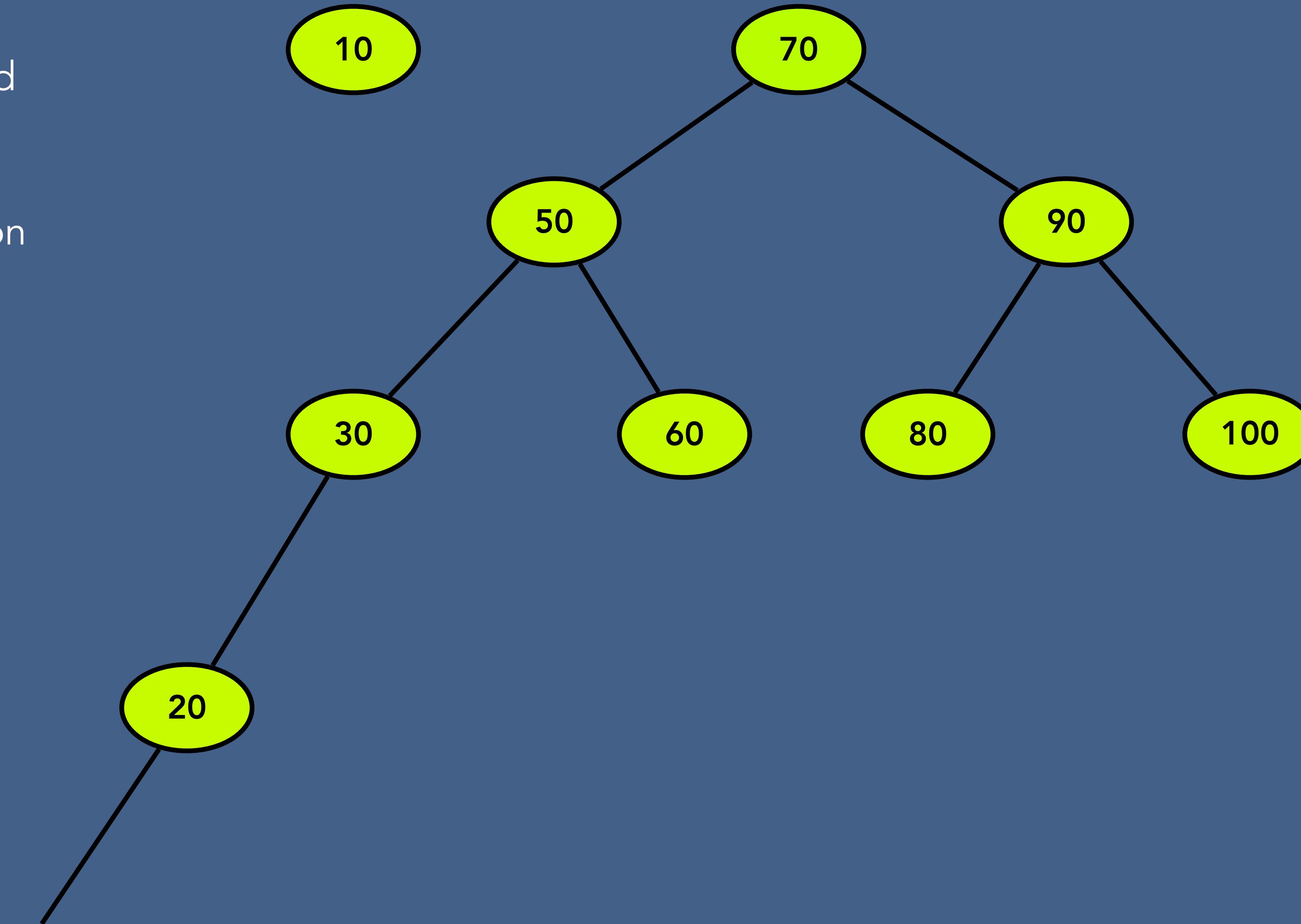
RL - right left condition



# AVL Tree - Insert a Node

Case 2 : Rotation is required

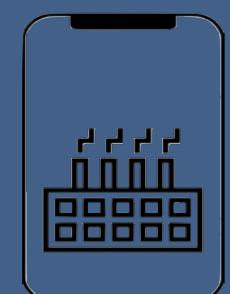
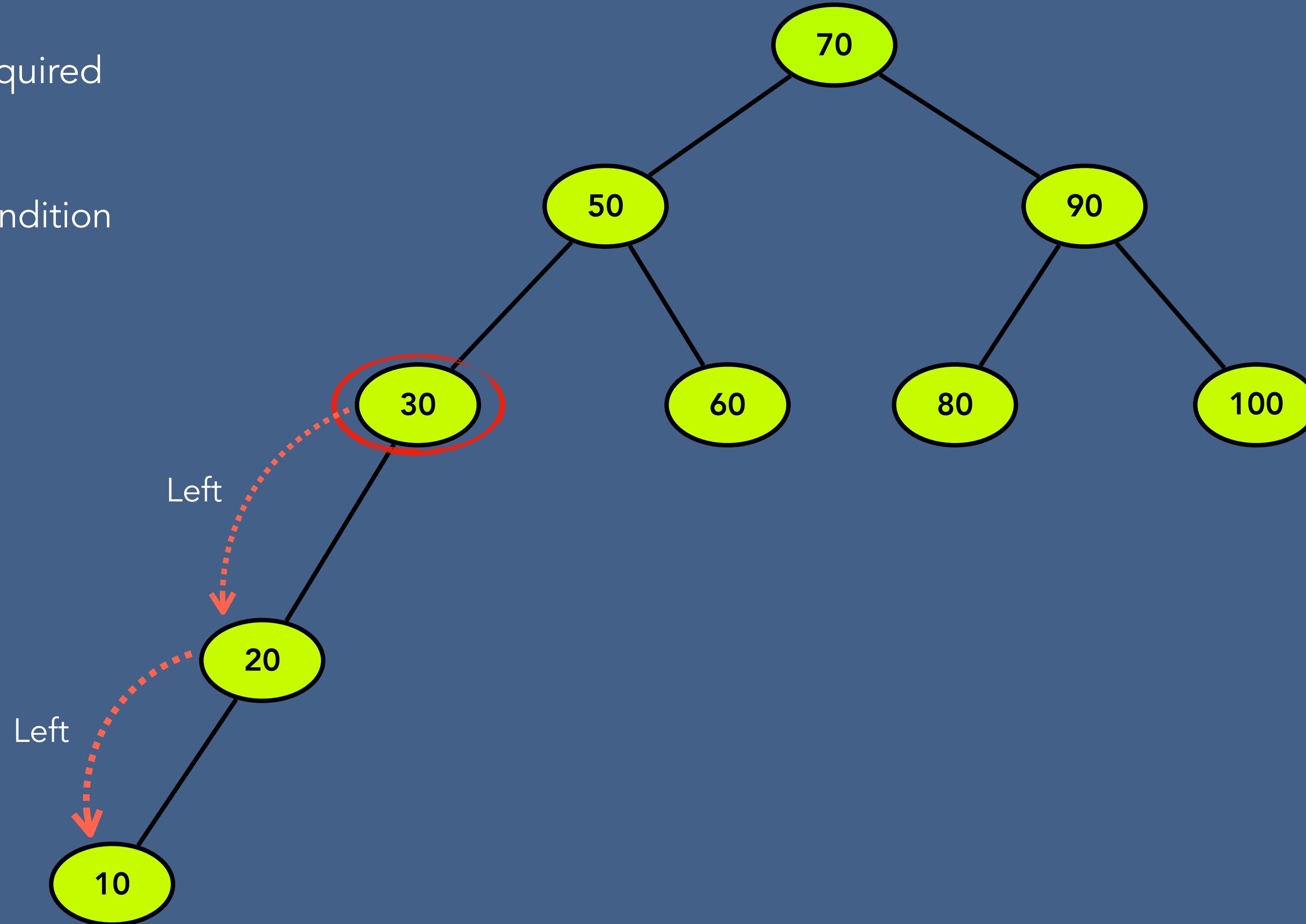
LL - left left condition



# AVL Tree - Insert a Node

Case 2 : Rotation is required

LL - left left condition

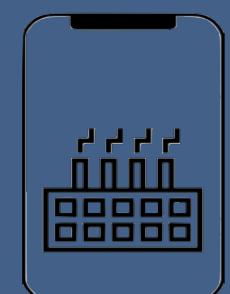
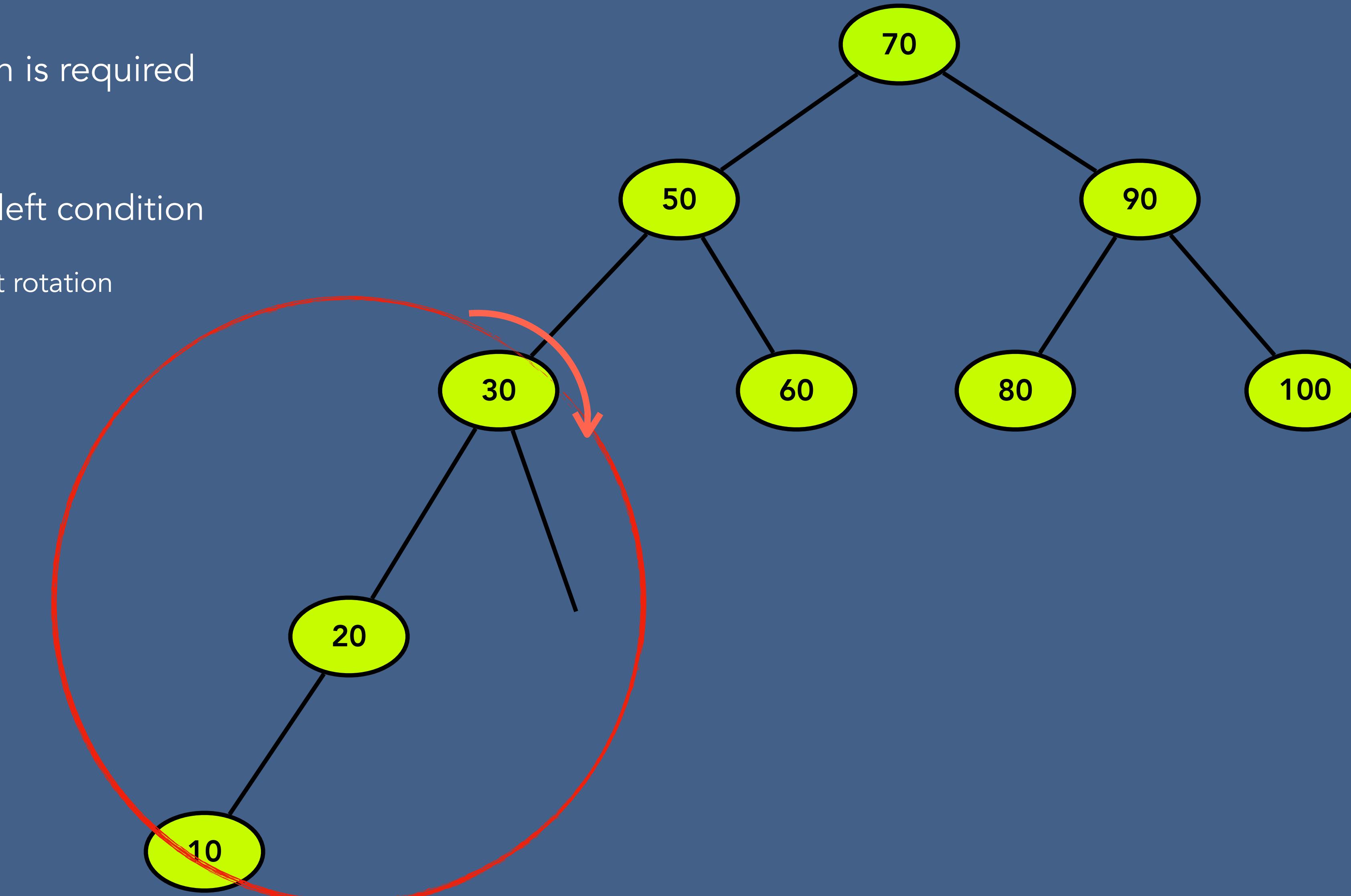


# AVL Tree - Insert a Node

Case 2 : Rotation is required

LL - left left condition

Right rotation

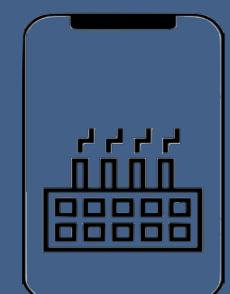
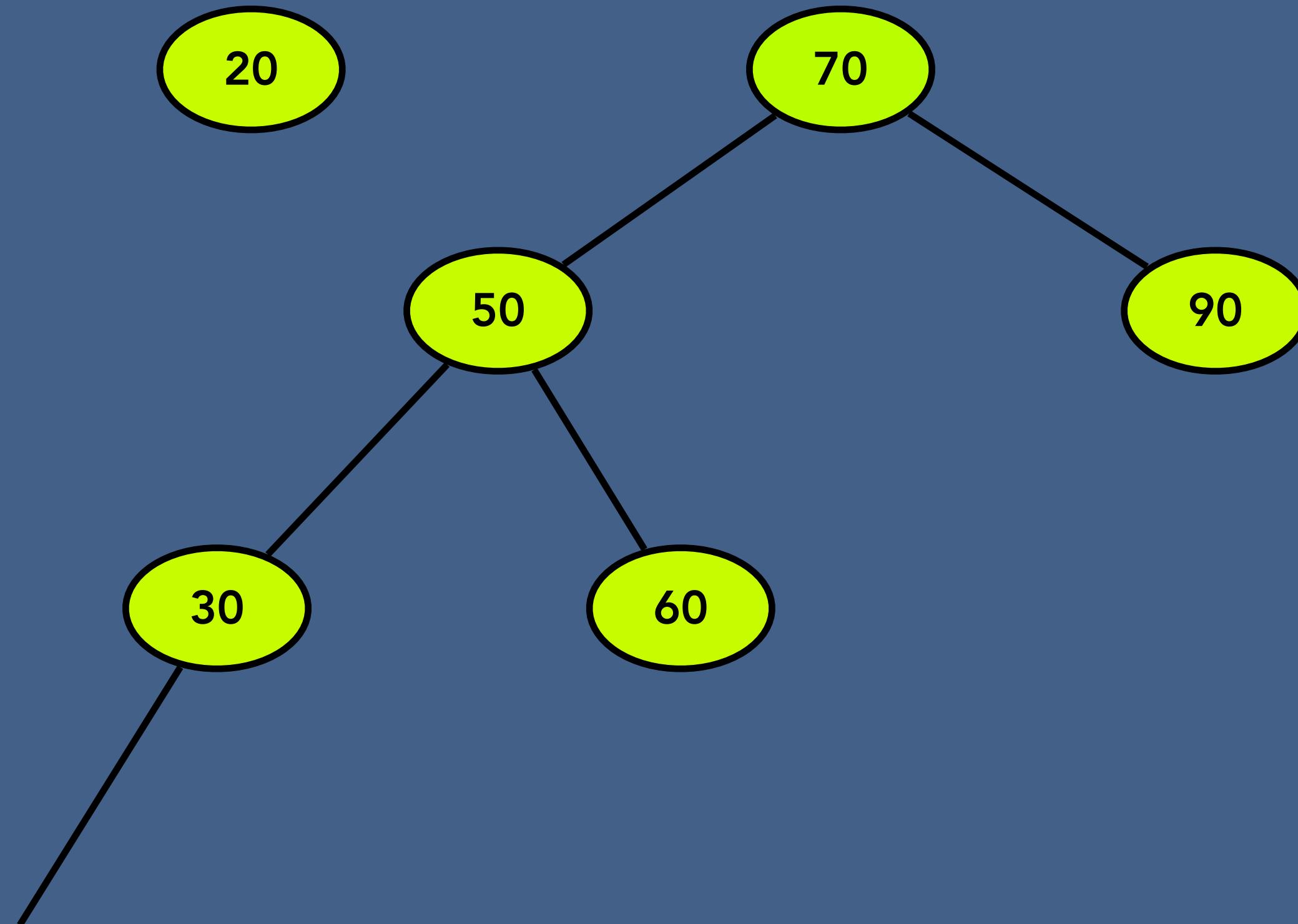


# AVL Tree - Insert a Node

Case 2 : Rotation is required

LL - left left condition

Right rotation - example 2

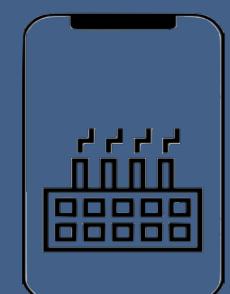
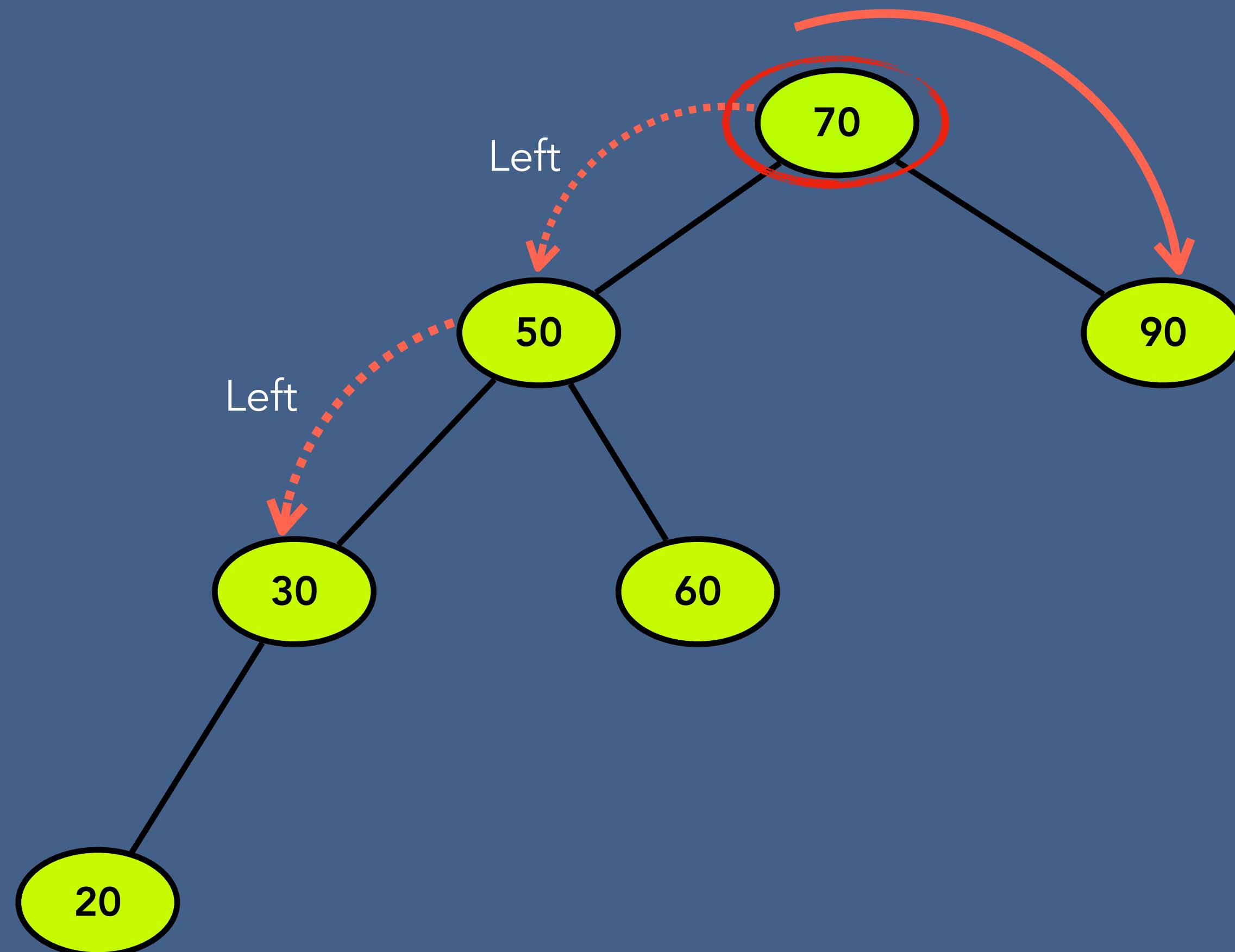


# AVL Tree - Insert a Node

Case 2 : Rotation is required

LL - left left condition

Right rotation - example 2

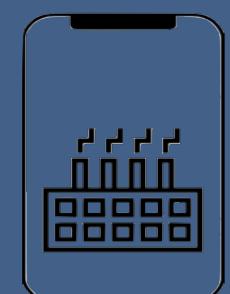
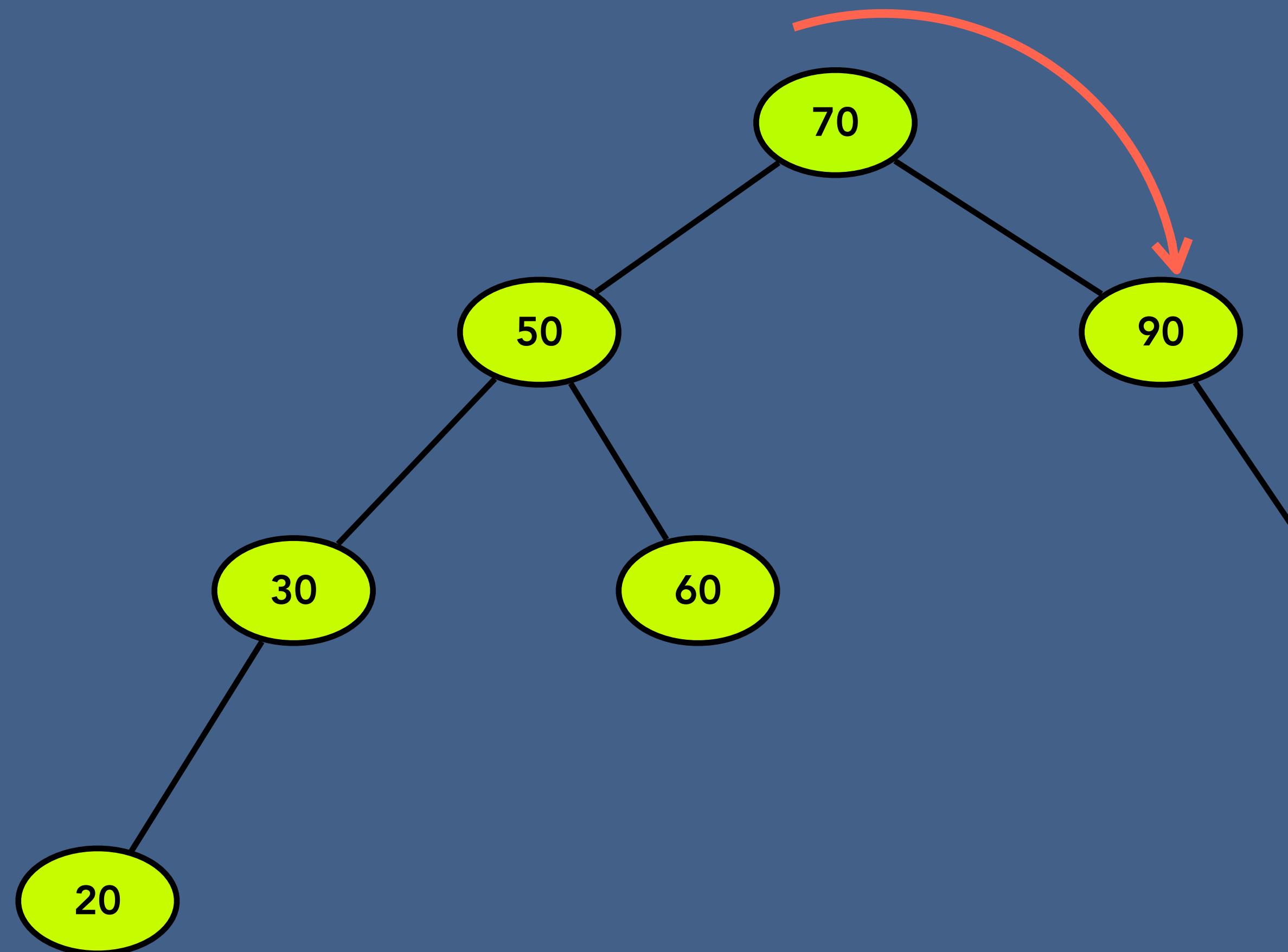


# AVL Tree - Insert a Node

Case 2 : Rotation is required

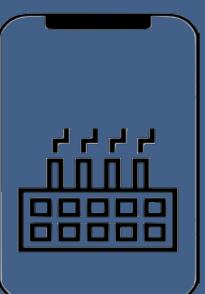
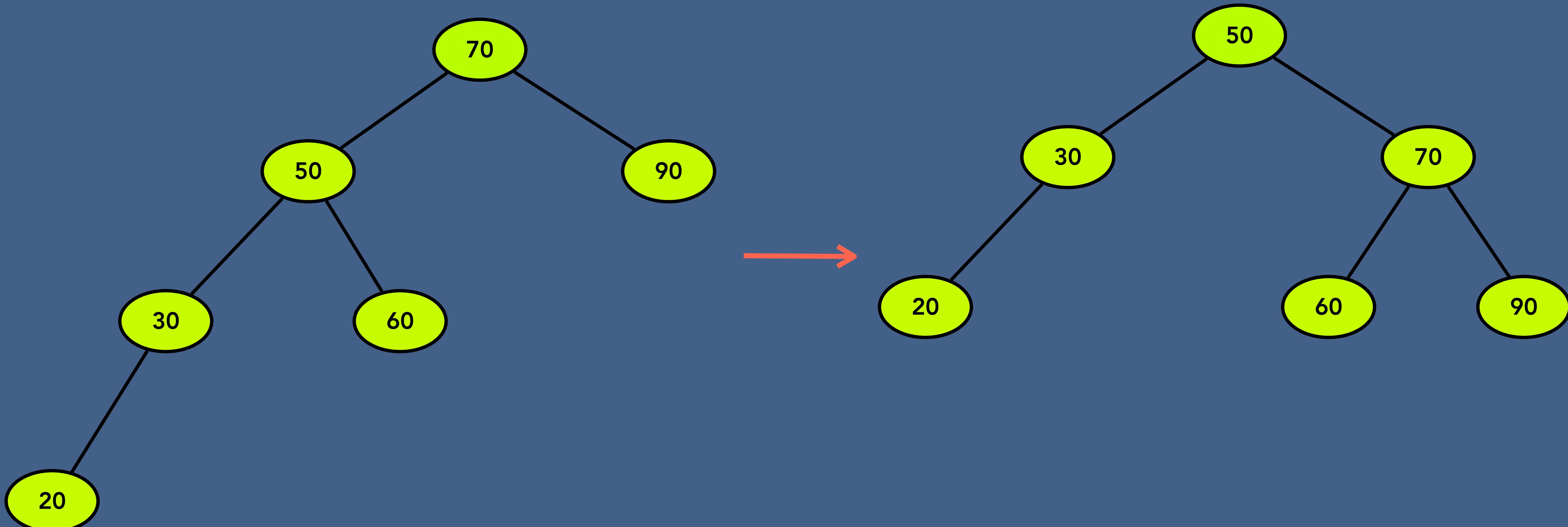
LL - left left condition

Right rotation - example 2



# AVL Tree - Insert a Node

Right Rotation - example 2

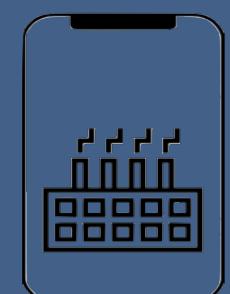
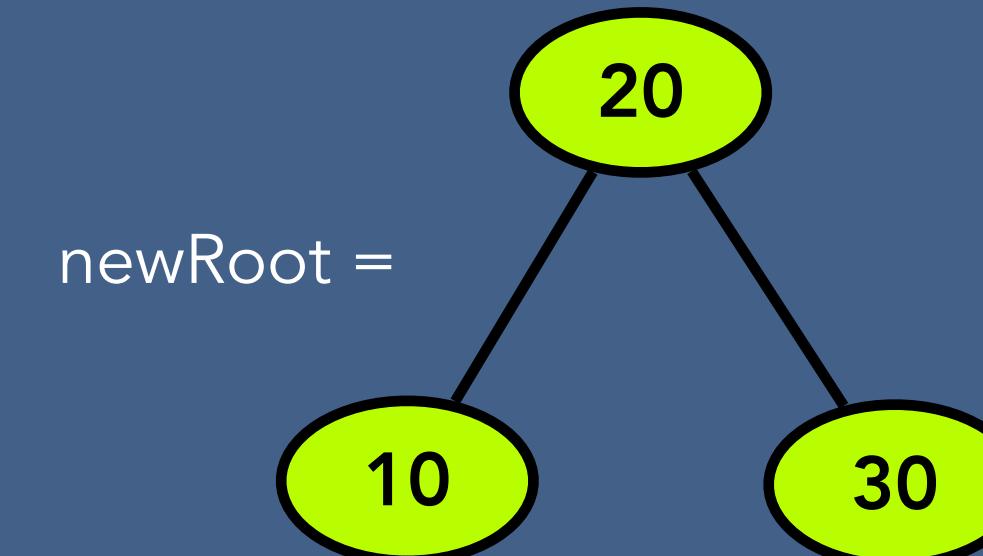
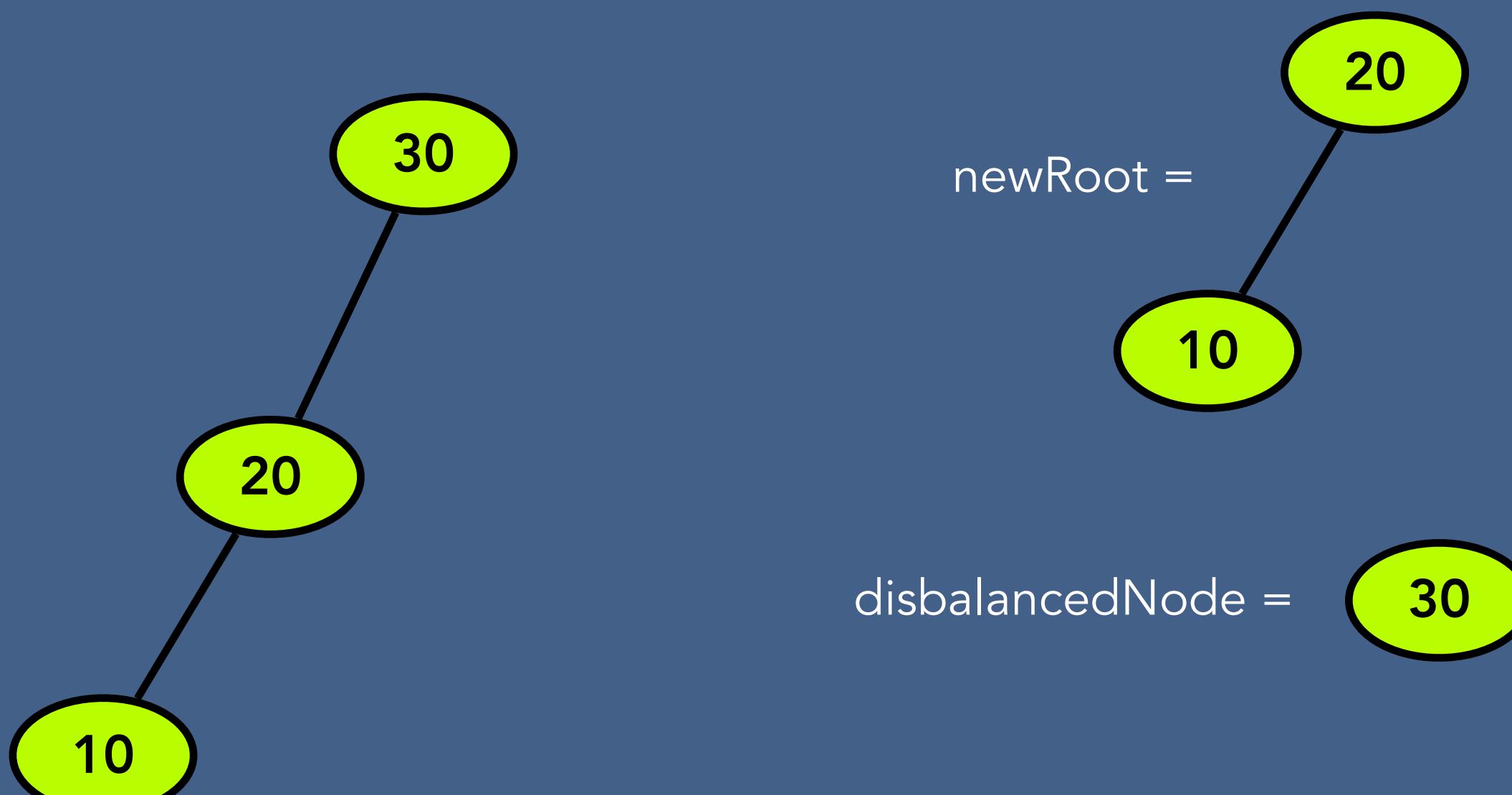


# AVL Tree - Insert a Node

## Algorithm of Left Left (LL) Condition

```
rotateRight(disbalancedNode) {  
    newRoot = disbalancedNode.leftChild ← Null  
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild  
    newRoot.rightChild = disbalancedNode  
    update height of disbalancedNode and newRoot  
    return newRoot  
}
```

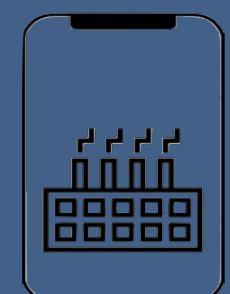
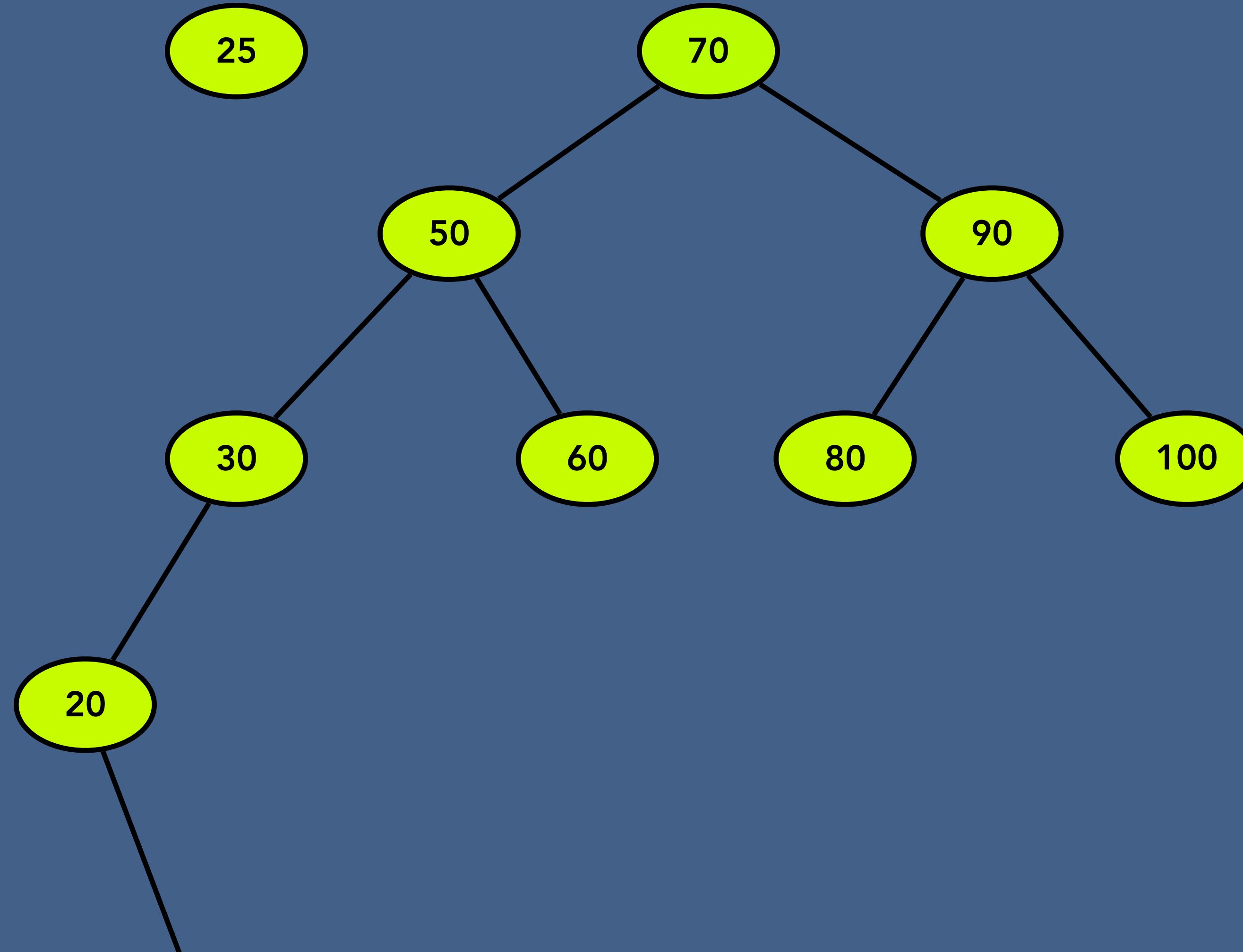
Time complexity : O(1)  
Space complexity : O(1)



# AVL Tree - Insert a Node

Case 2 : Rotation is required

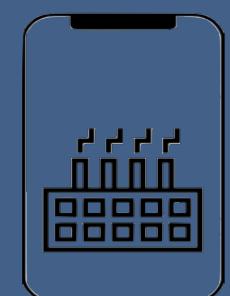
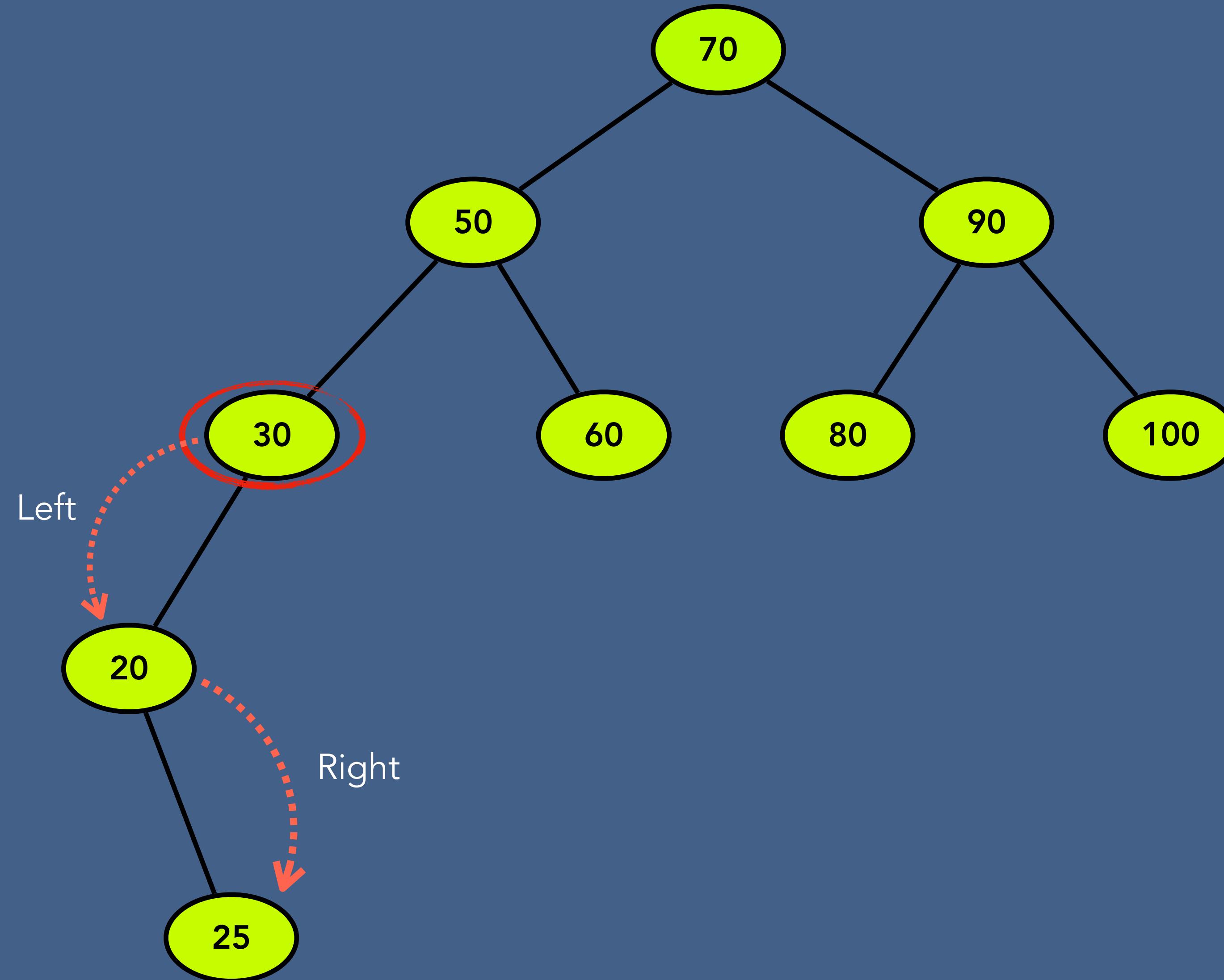
LR - left right condition



# AVL Tree - Insert a Node

Case 2 : Rotation is required

LR - left right condition

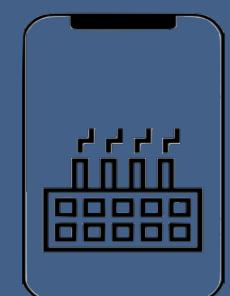
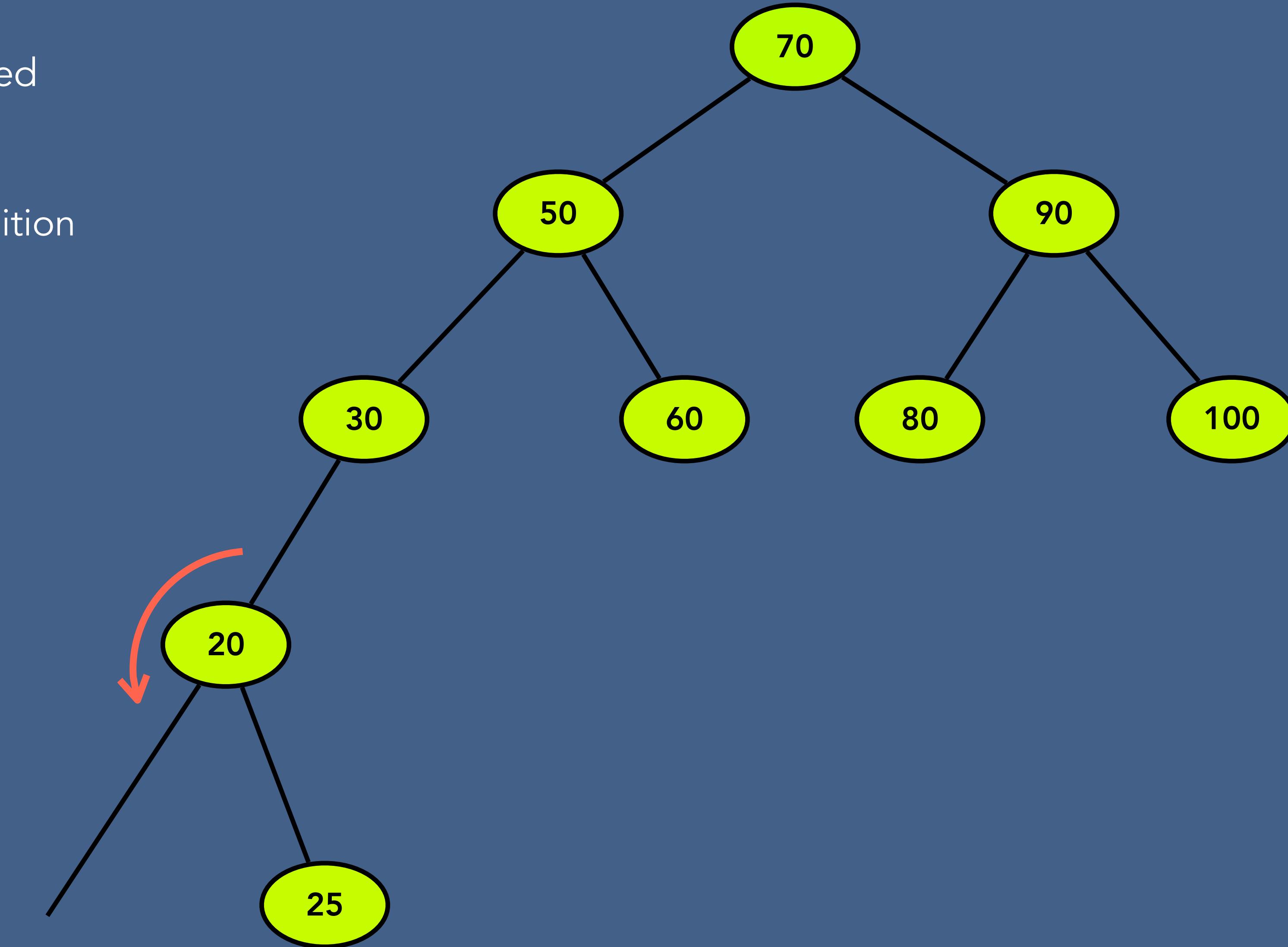


# AVL Tree - Insert a Node

Case 2 : Rotation is required

LR - left right condition

1. Left rotation
2. Right rotation

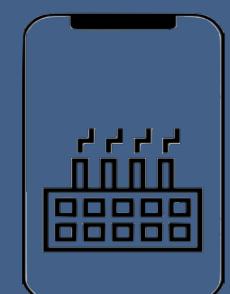
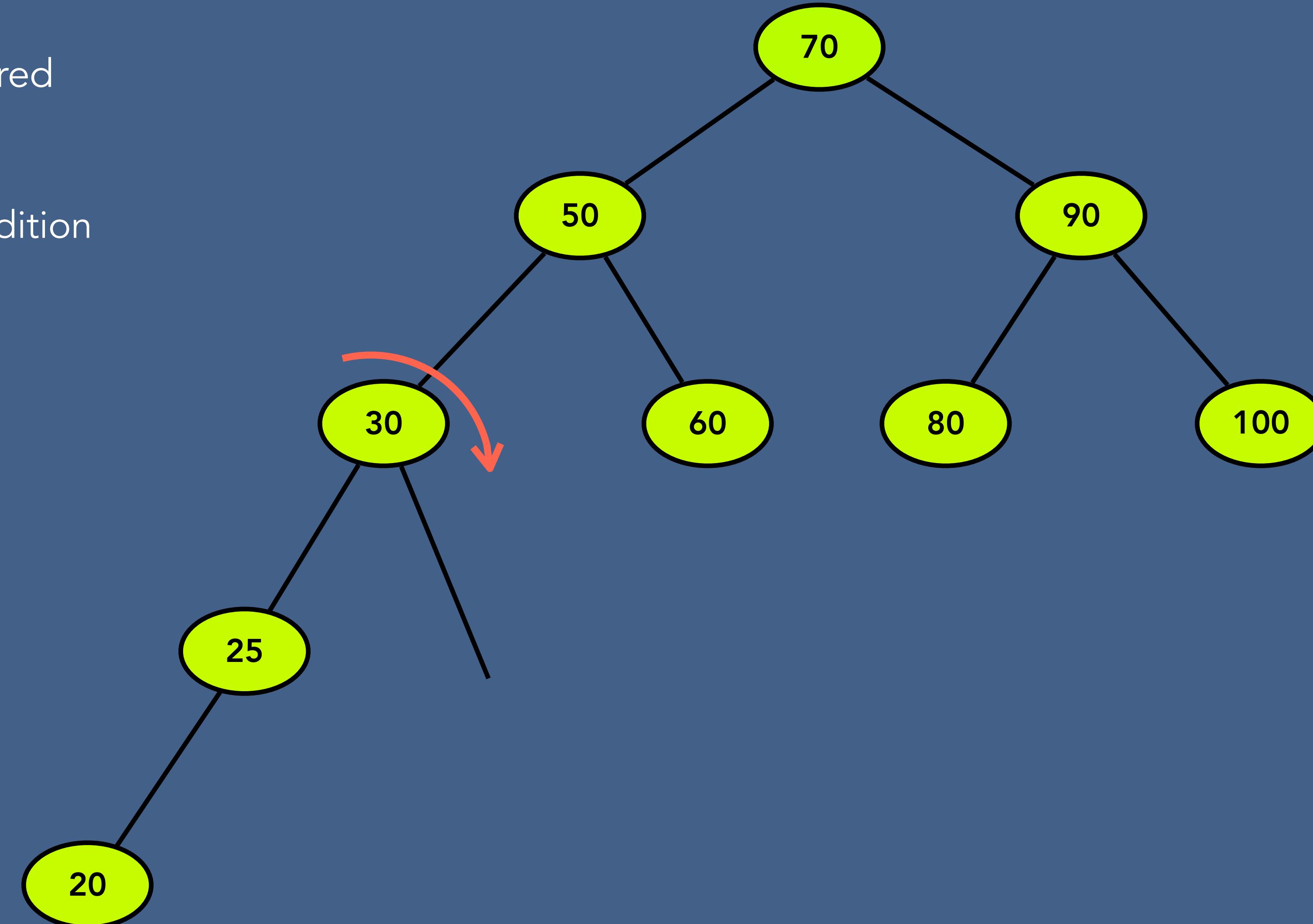


# AVL Tree - Insert a Node

Case 2 : Rotation is required

LR - left right condition

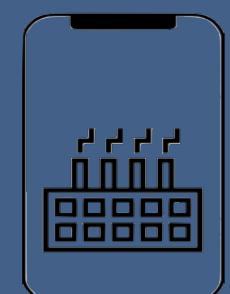
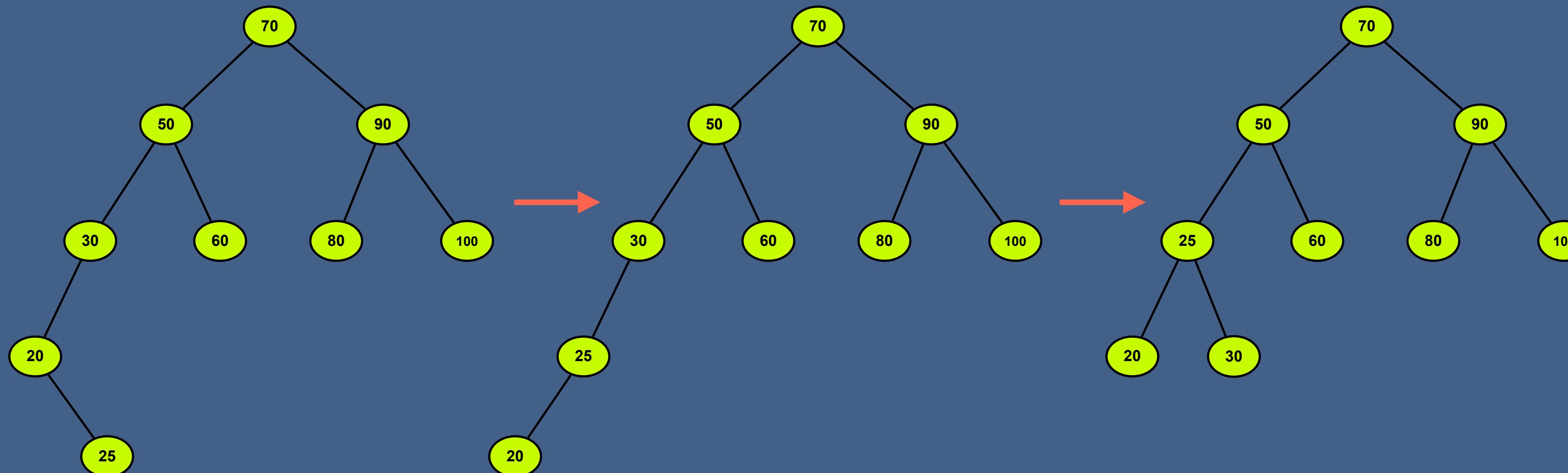
1. Left rotation
2. Right rotation



# AVL Tree - Insert a Node

Case 2 : Left Right Condition

1. Left Rotation
2. Right Rotation



# AVL Tree - Insert a Node

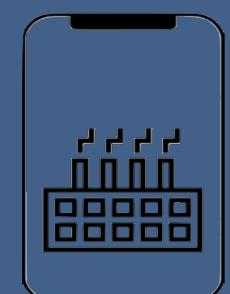
## Case 2 : Left Right Condition

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
```



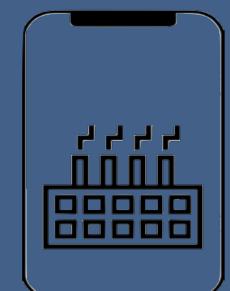
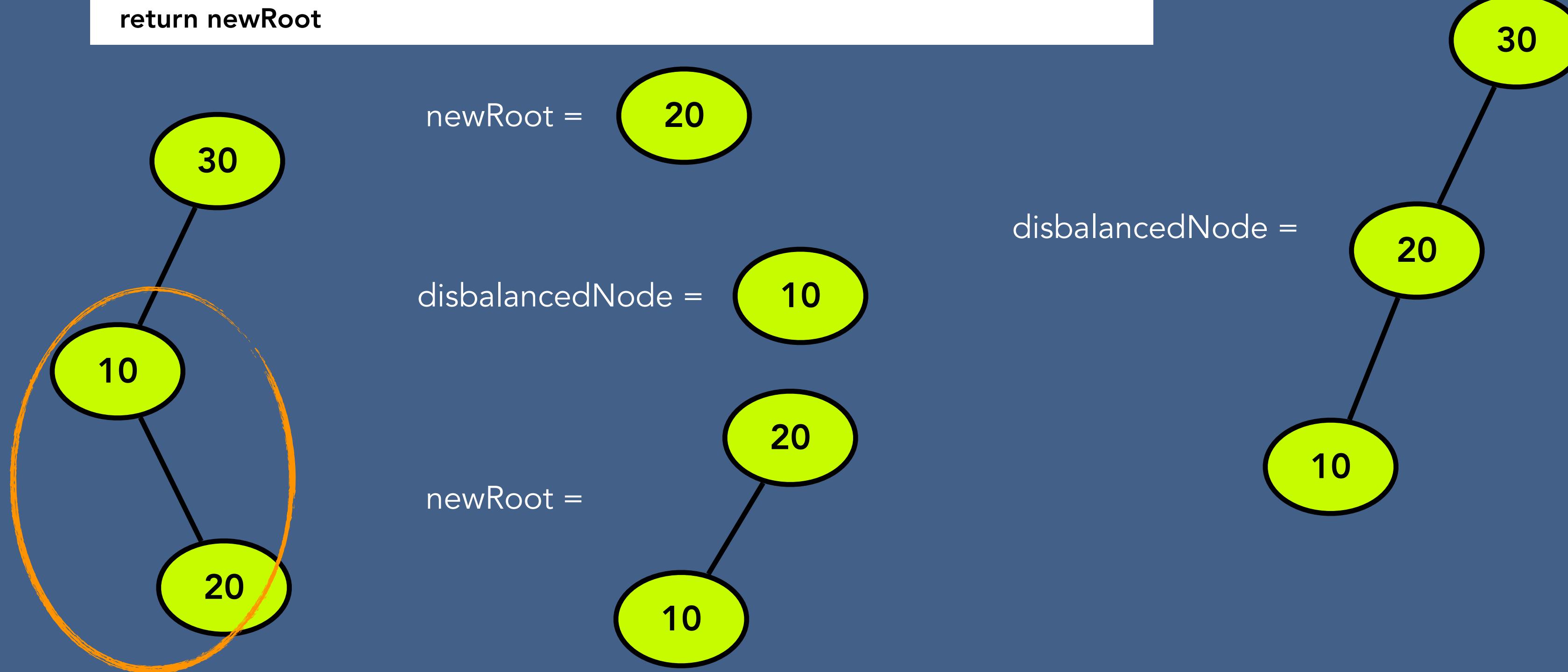
# AVL Tree - Insert a Node

Case 2 : Left Right 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
```



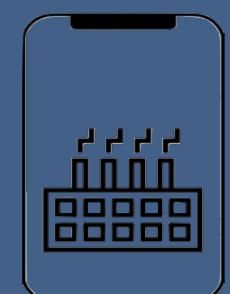
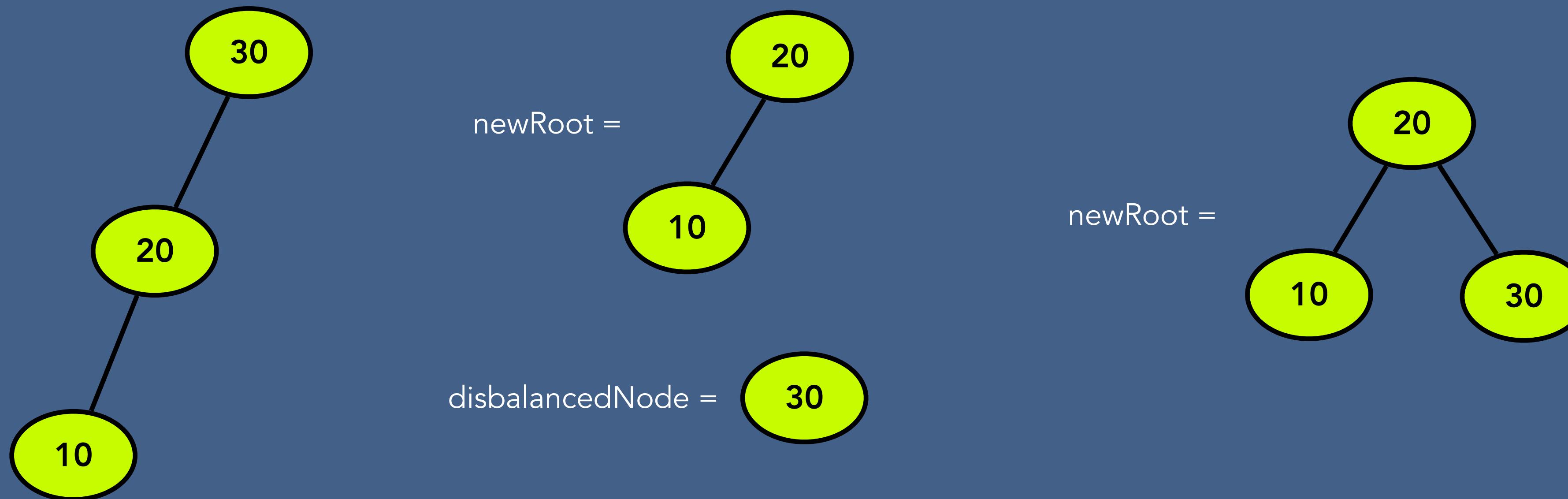
# AVL Tree - Insert a Node

Case 2 : Left Right 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
```



# AVL Tree - Insert a Node

## Case 2 : Left Right Condition

Step 1 : rotate Left disbalancedNode.leftChild

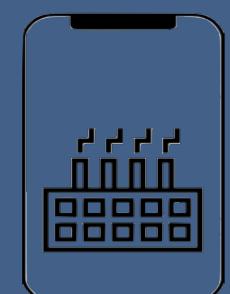
Step 2 : rotate Right disbalancedNode

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

Time complexity : O(1)

Space complexity : O(1)

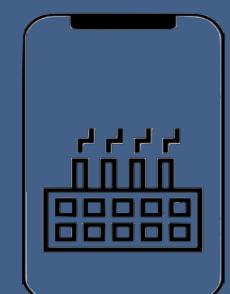
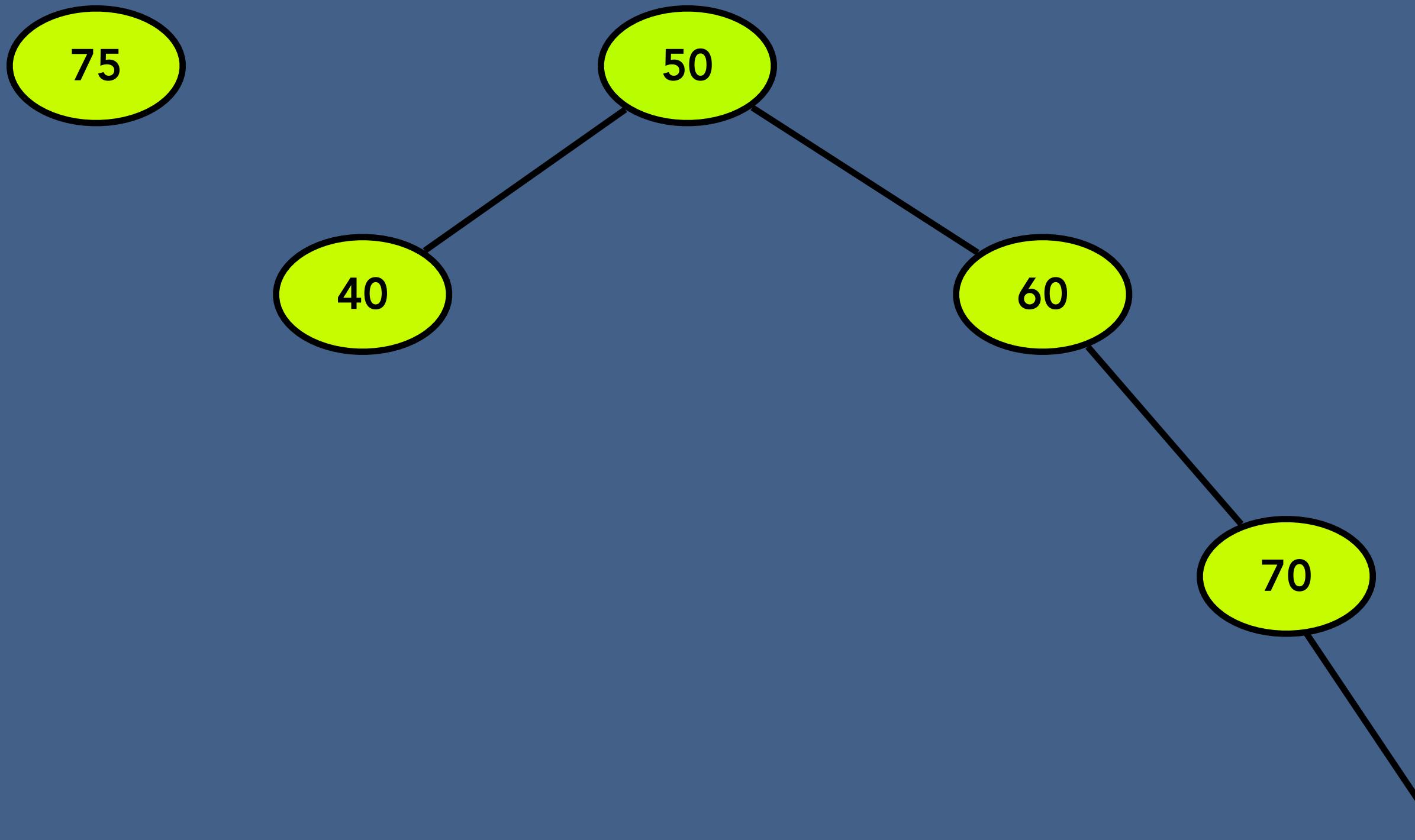
```
rotateRight(disbalancedNode)
    newRoot = disbalancedNode.leftChild
    disbalancedNode.leftChild = disbalancedNode.leftChild.rightChild
    newRoot.rightChild = disbalancedNode
    update height of disbalancedNode and newRoot
    return newRoot
```



# AVL Tree - Insert a Node

Case 2 : Rotation is required

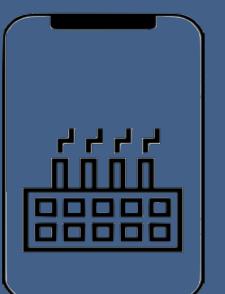
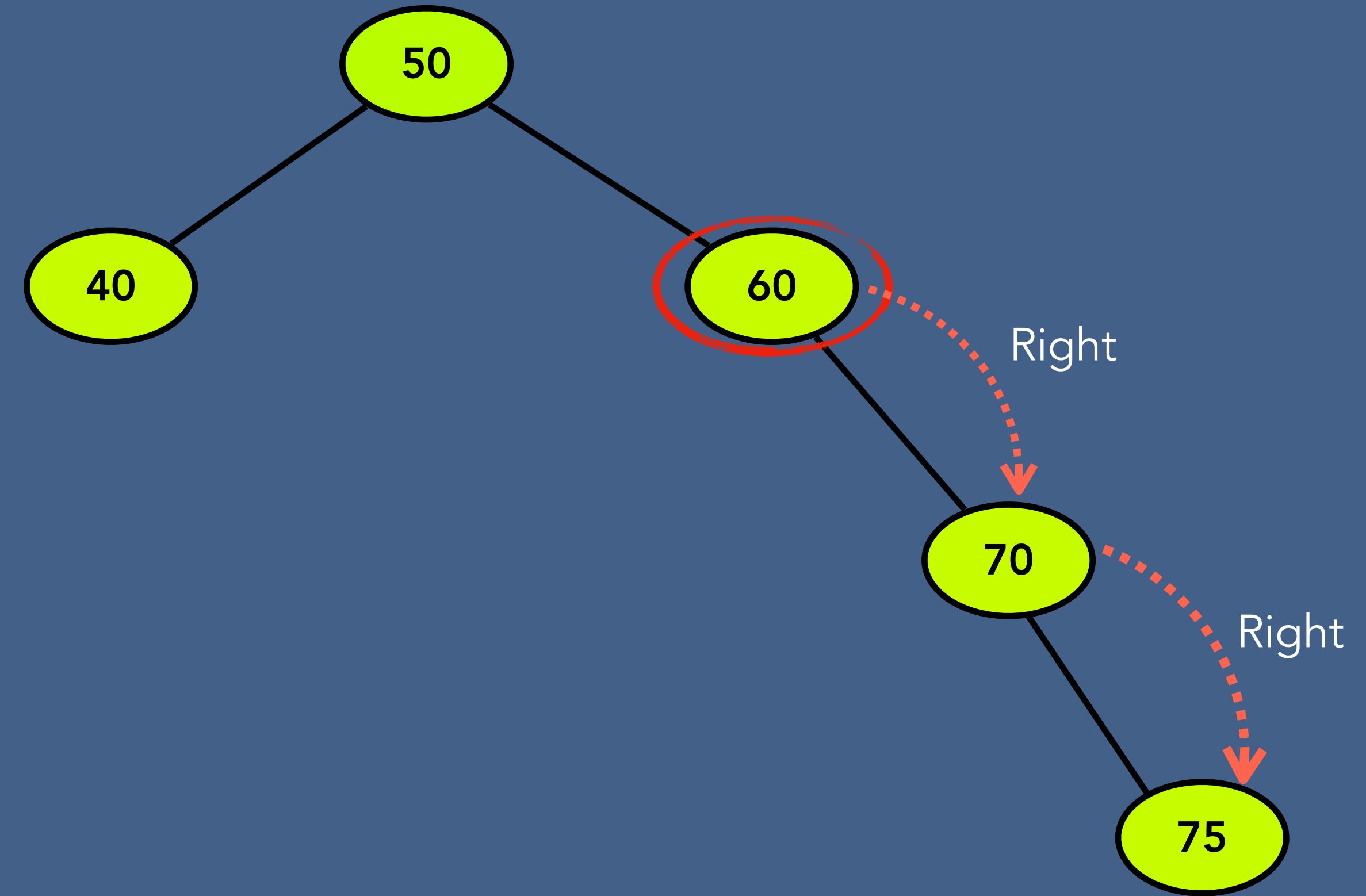
RR - right right condition



# AVL Tree - Insert a Node

Case 2 : Rotation is required

RR - right right condition

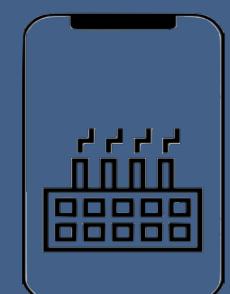
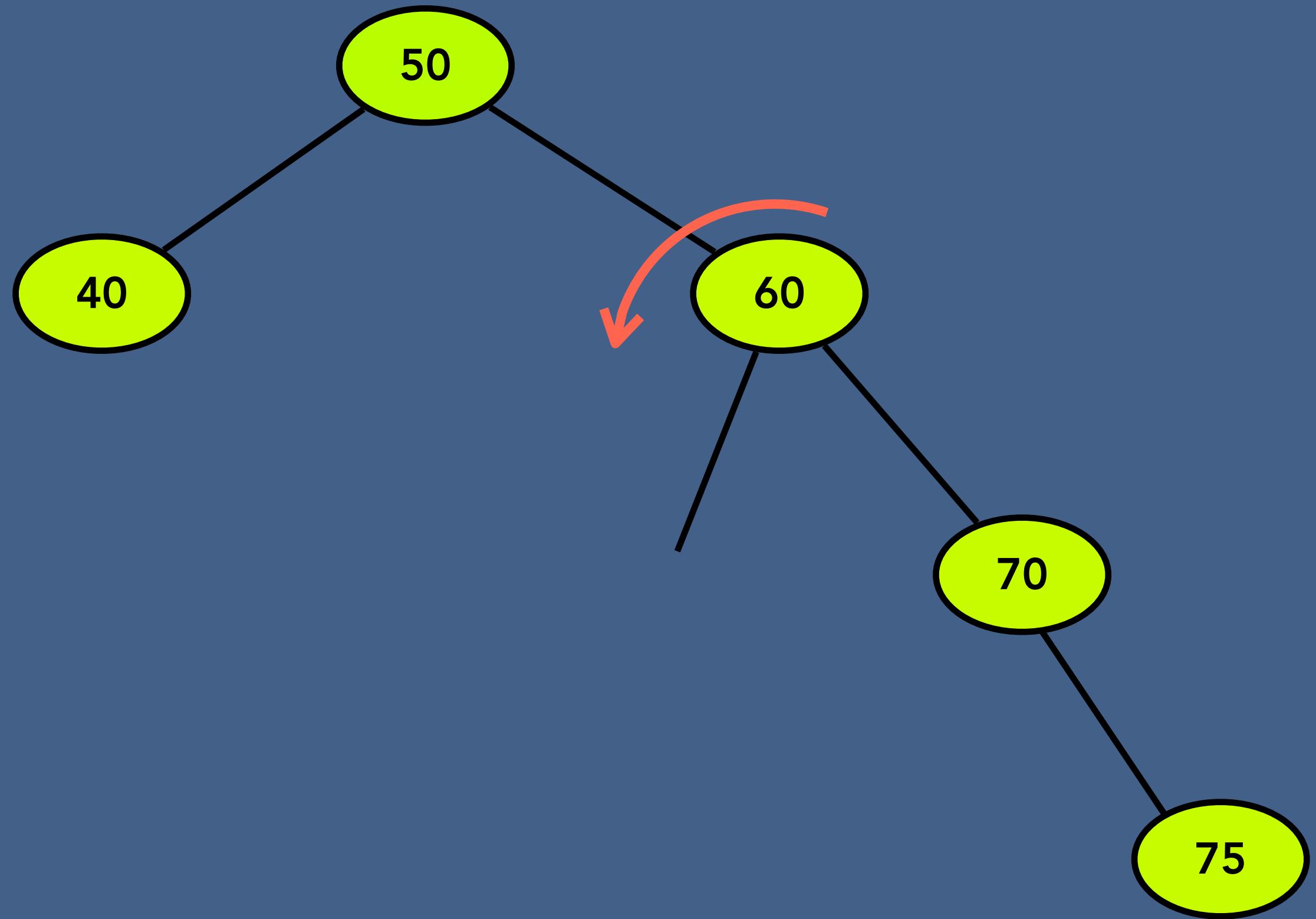


# AVL Tree - Insert a Node

Case 2 : Rotation is required

RR - right right condition

Left Rotation

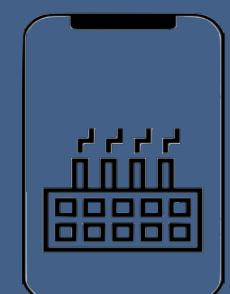
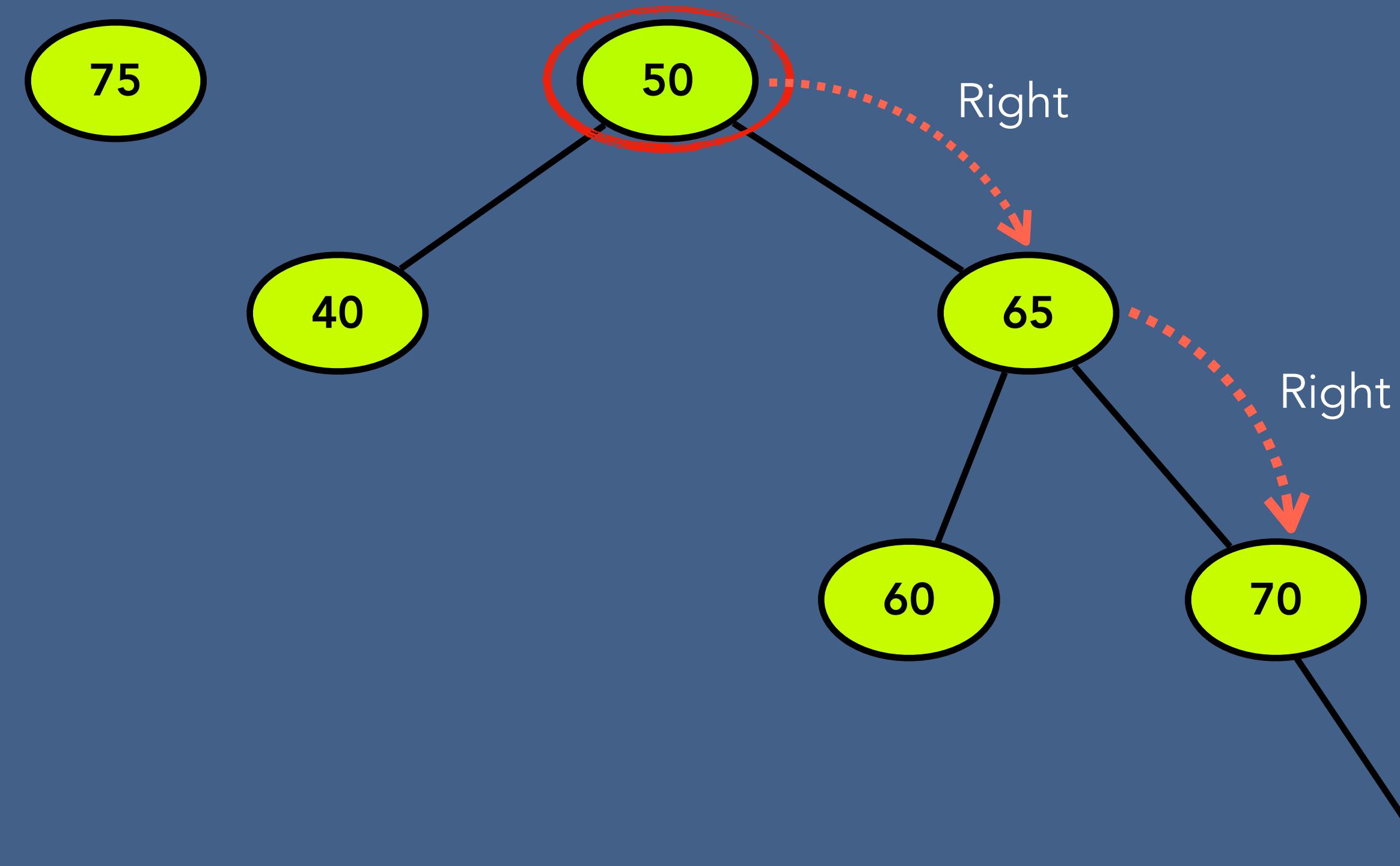


# AVL Tree - Insert a Node

Case 2 : Rotation is required

RR - right right condition

Left Rotation - example 2

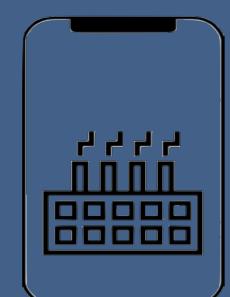
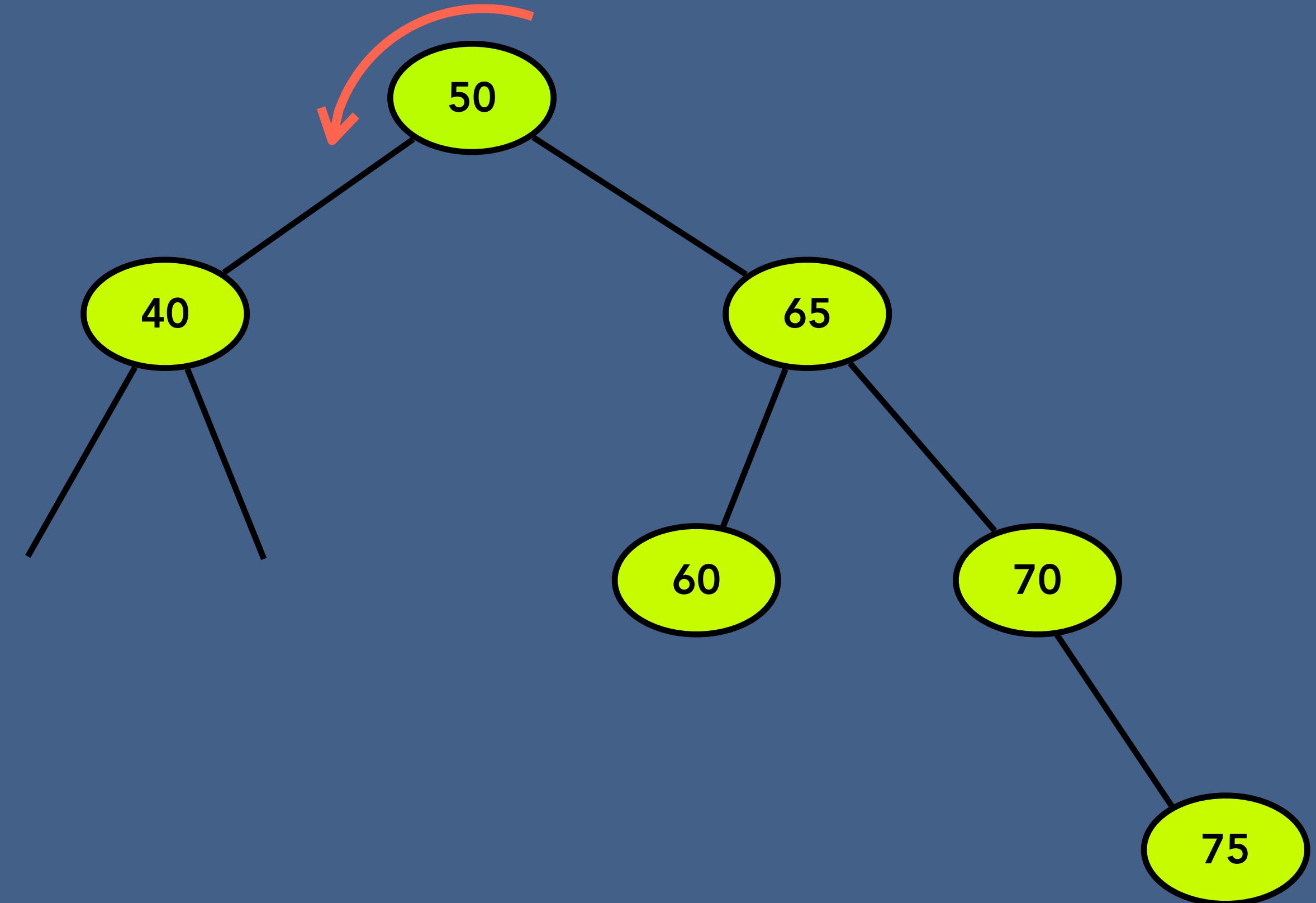


# AVL Tree - Insert a Node

Case 2 : Rotation is required

RR - right right condition

Left Rotation - example 2

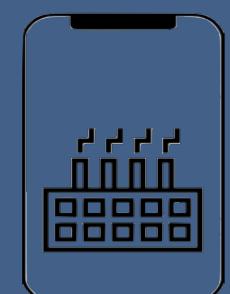
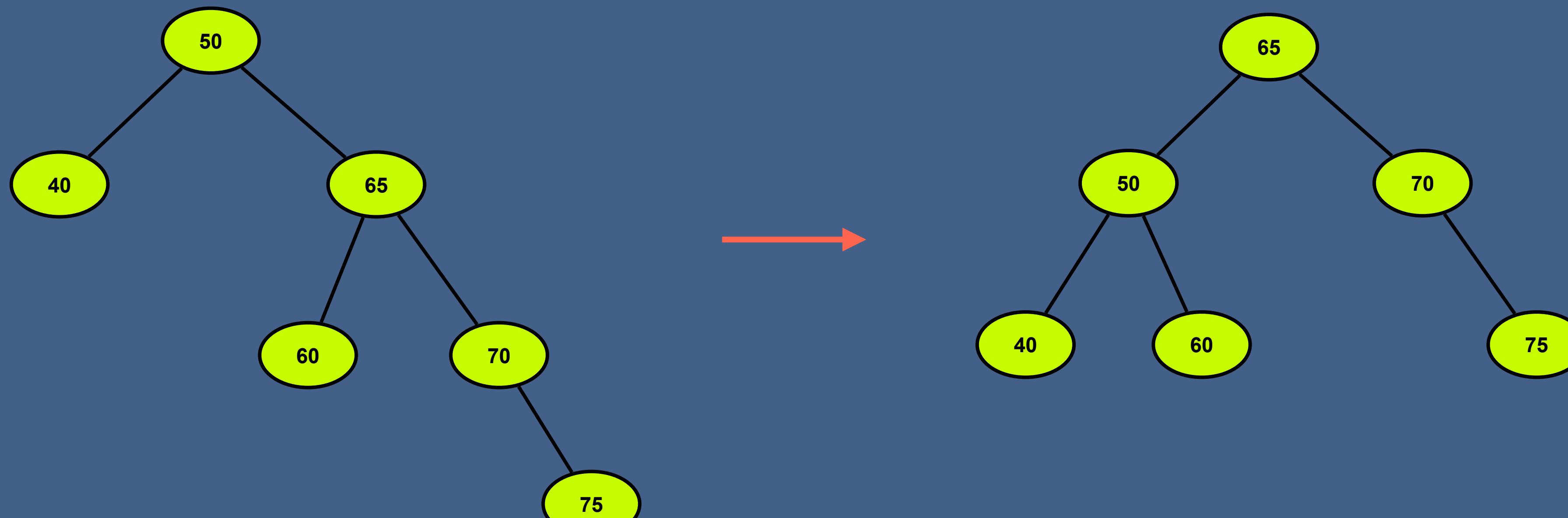


# AVL Tree - Insert a Node

Case 2 : Rotation is required

RR - right right condition

Left Rotation



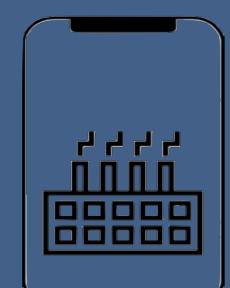
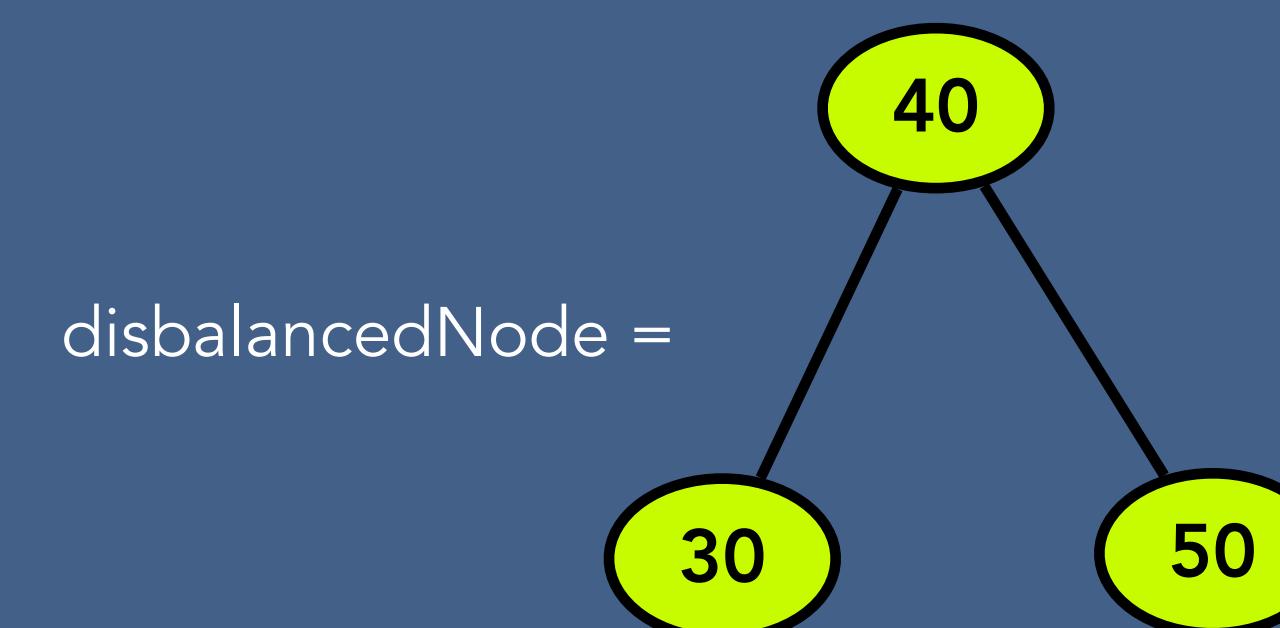
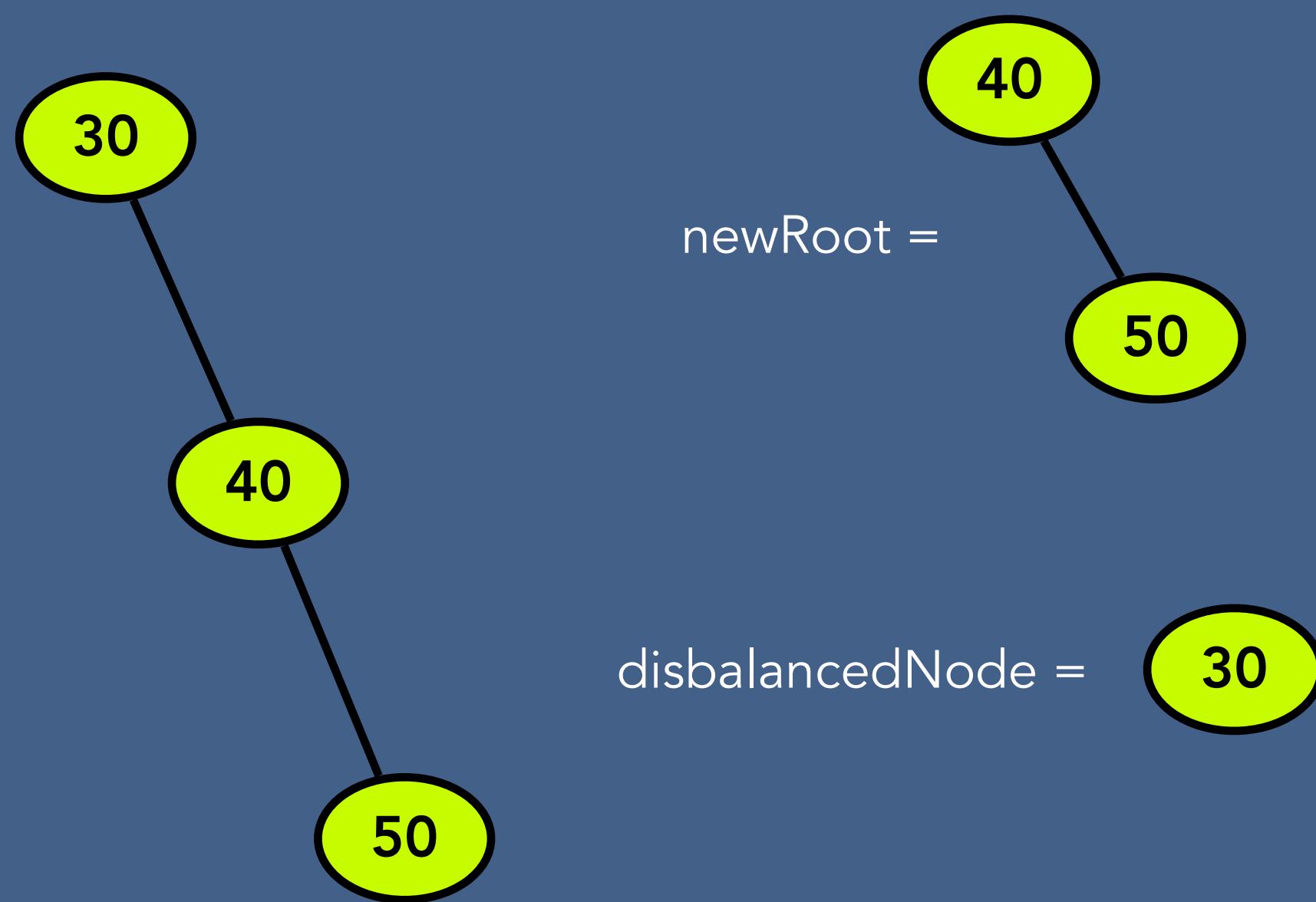
# AVL Tree - Insert a Node

RR - right right condition

- rotate Left disbalancedNode

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

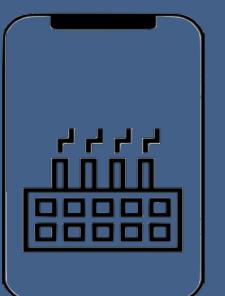
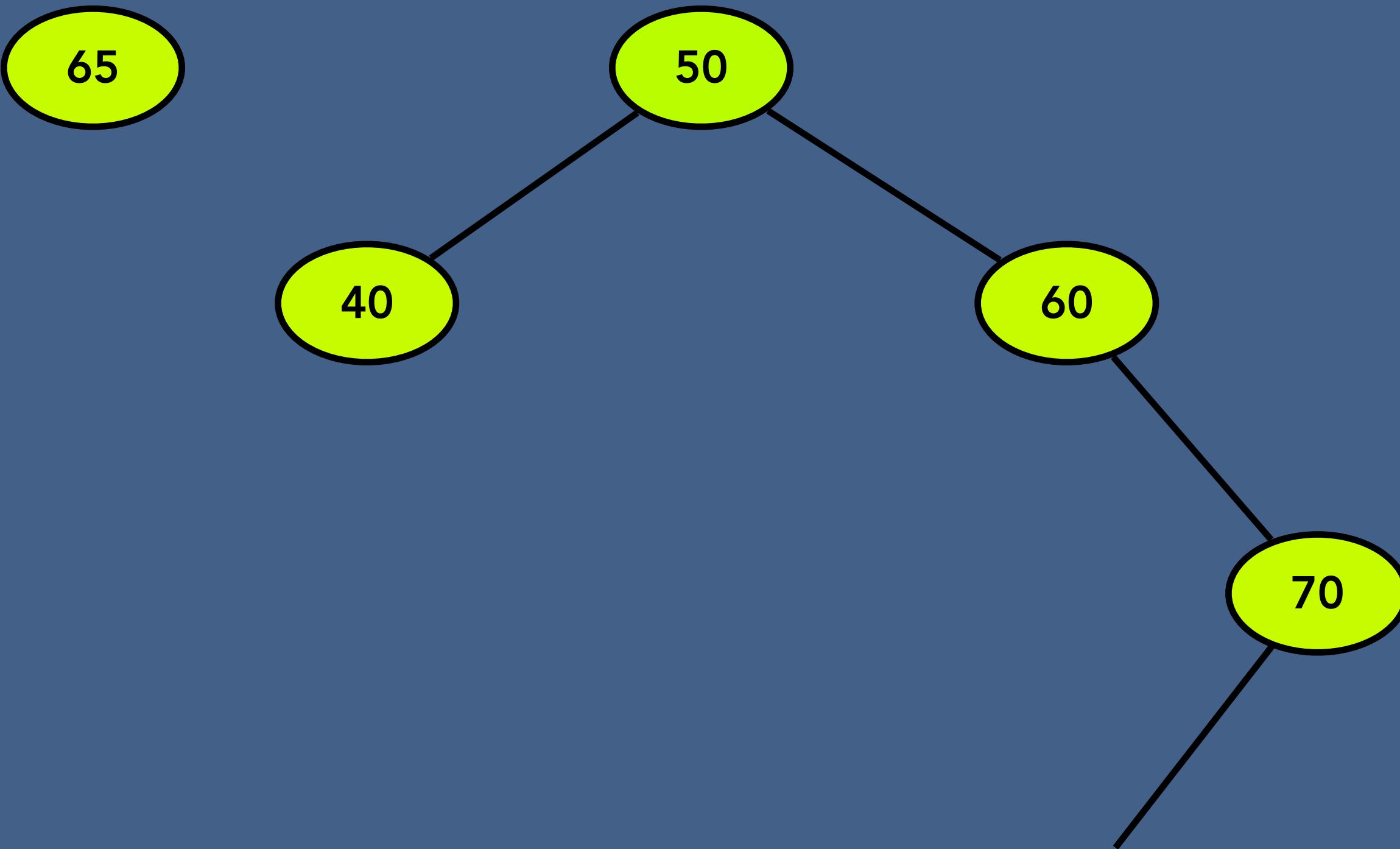
Time complexity : O(1)  
Space complexity : O(1)



# AVL Tree - Insert a Node

Case 2 : Rotation is required

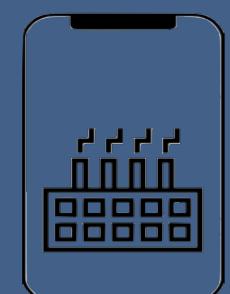
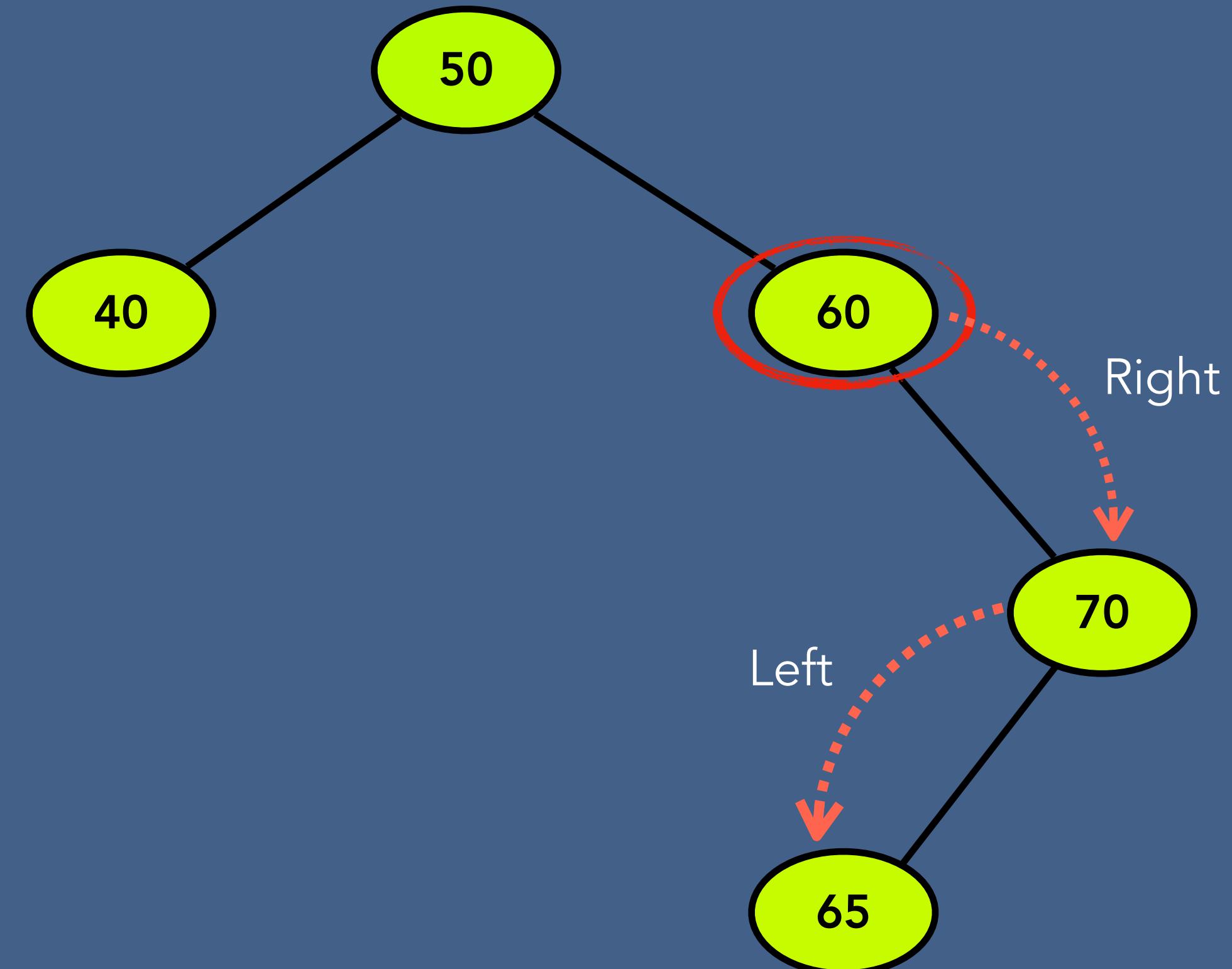
RL - right left condition



# AVL Tree - Insert a Node

Case 2 : Rotation is required

RL - right left condition

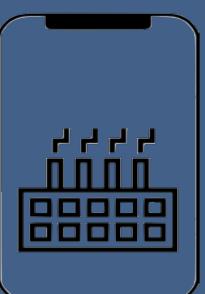
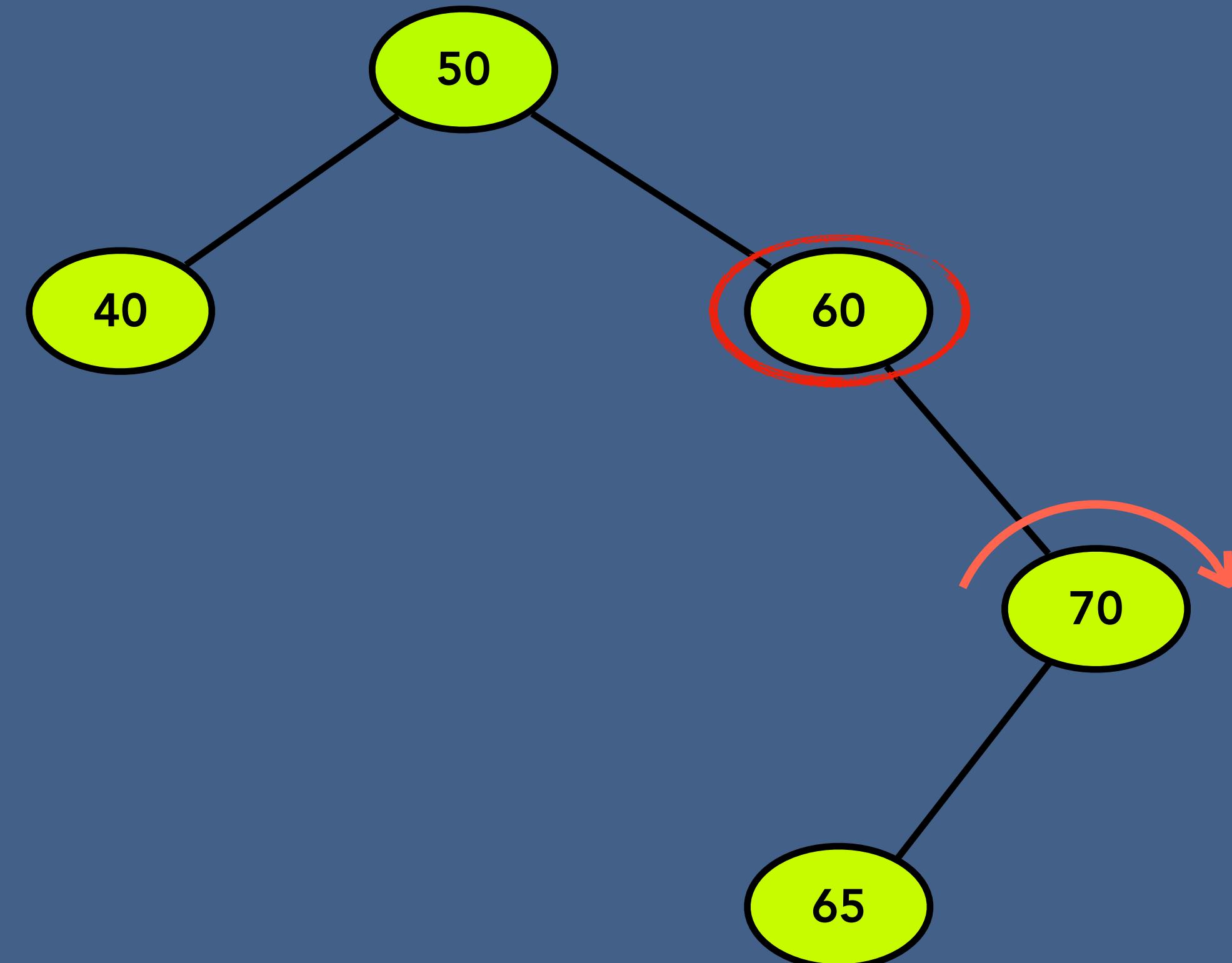


# AVL Tree - Insert a Node

Case 2 : Rotation is required

RL - right left condition

1. Right rotation
2. Left rotation

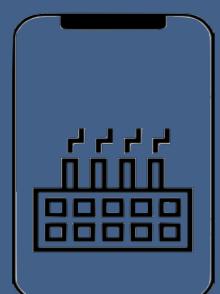
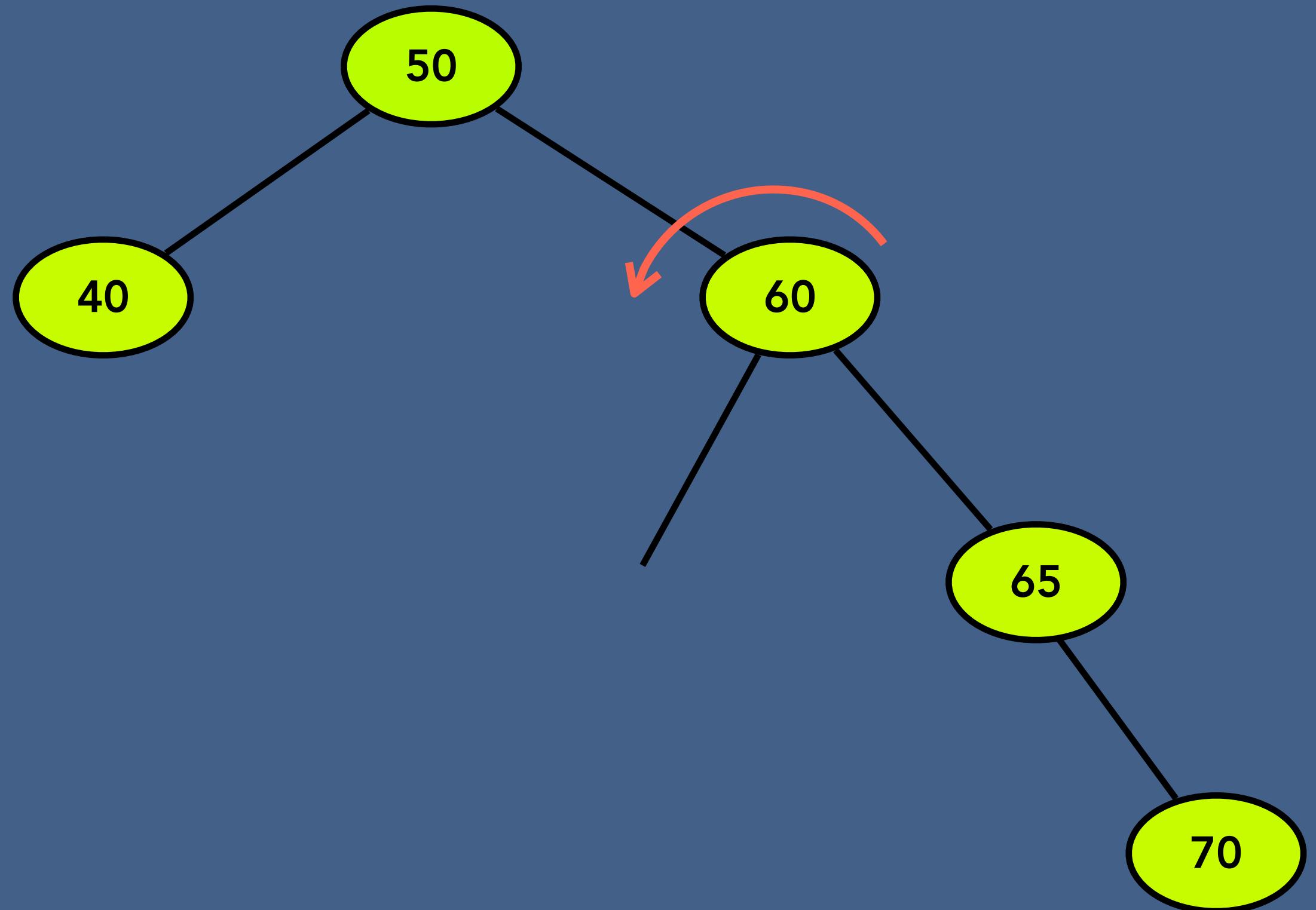


# AVL Tree - Insert a Node

Case 2 : Rotation is required

RL - right left condition

1. Right rotation
2. Left rotation



# AVL Tree - Insert a Node

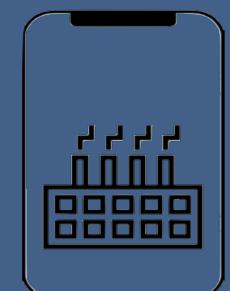
RL - right left condition

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
```



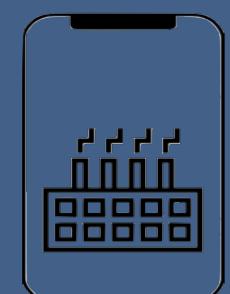
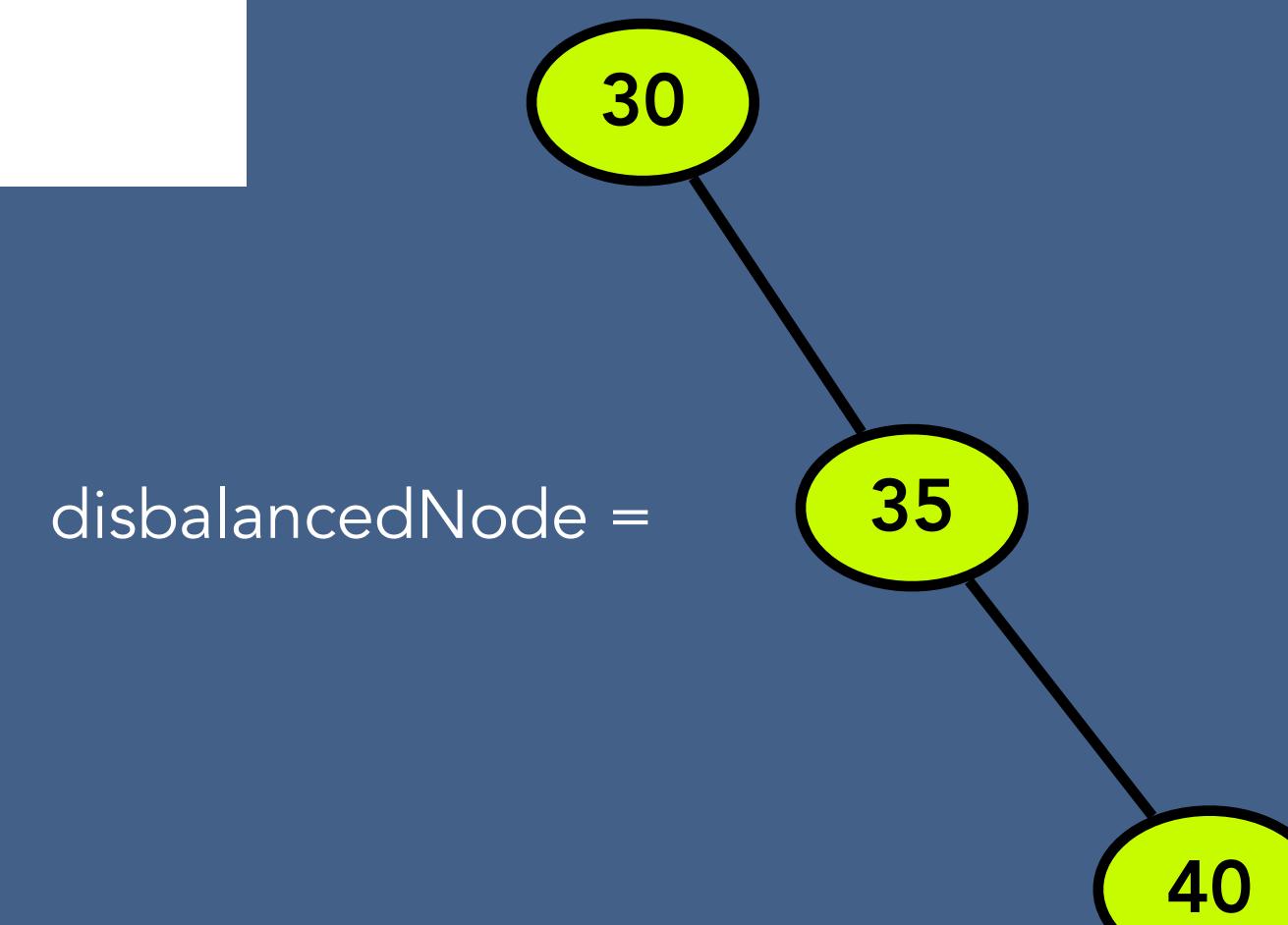
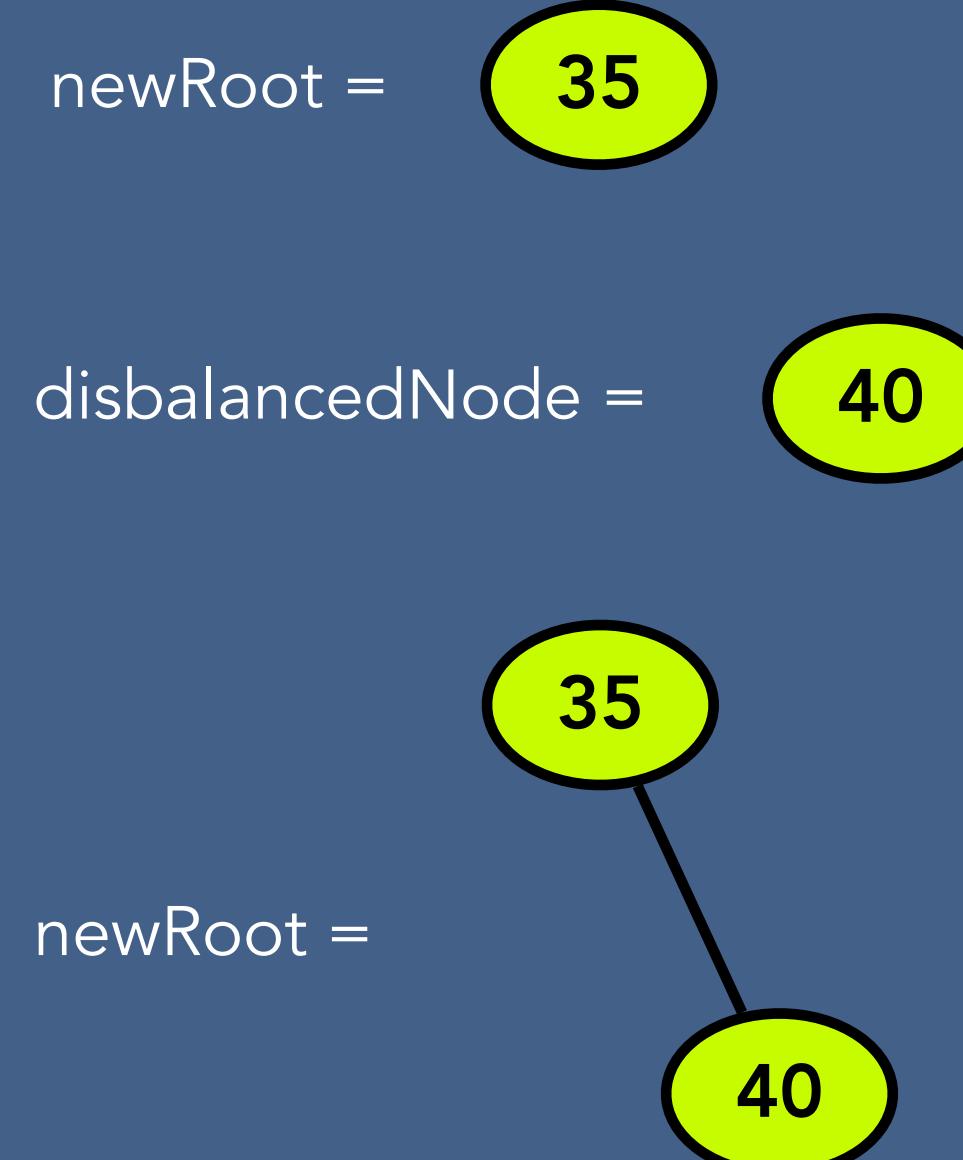
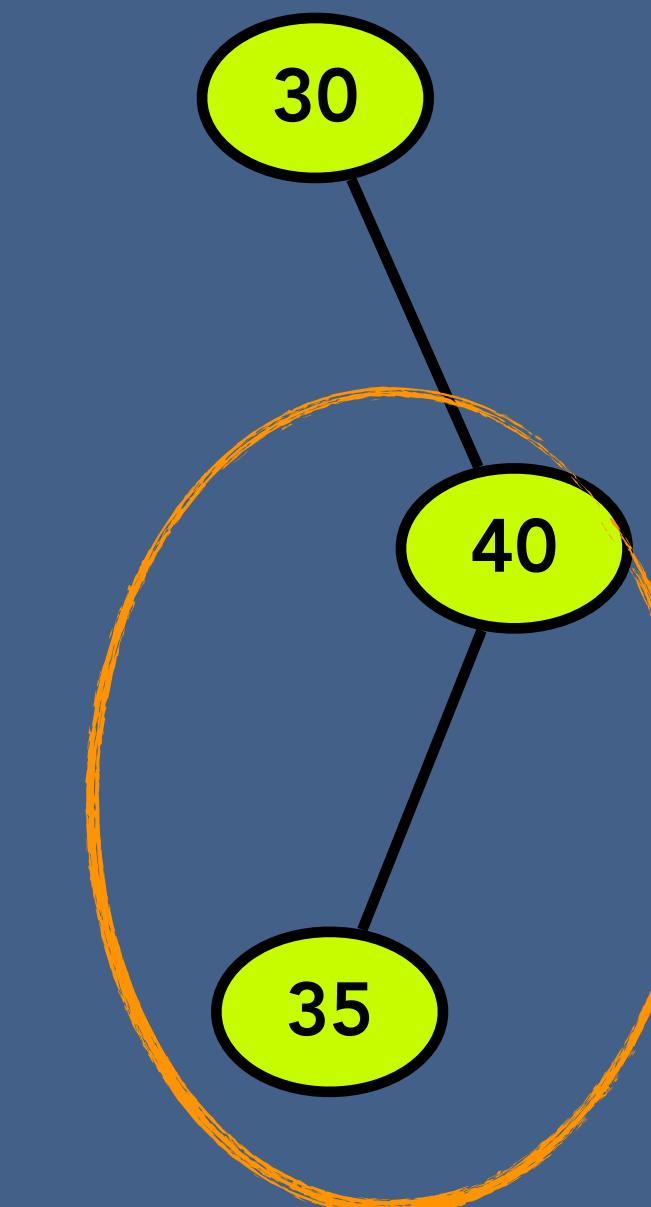
# AVL Tree - Insert a Node

RL - right left 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
```



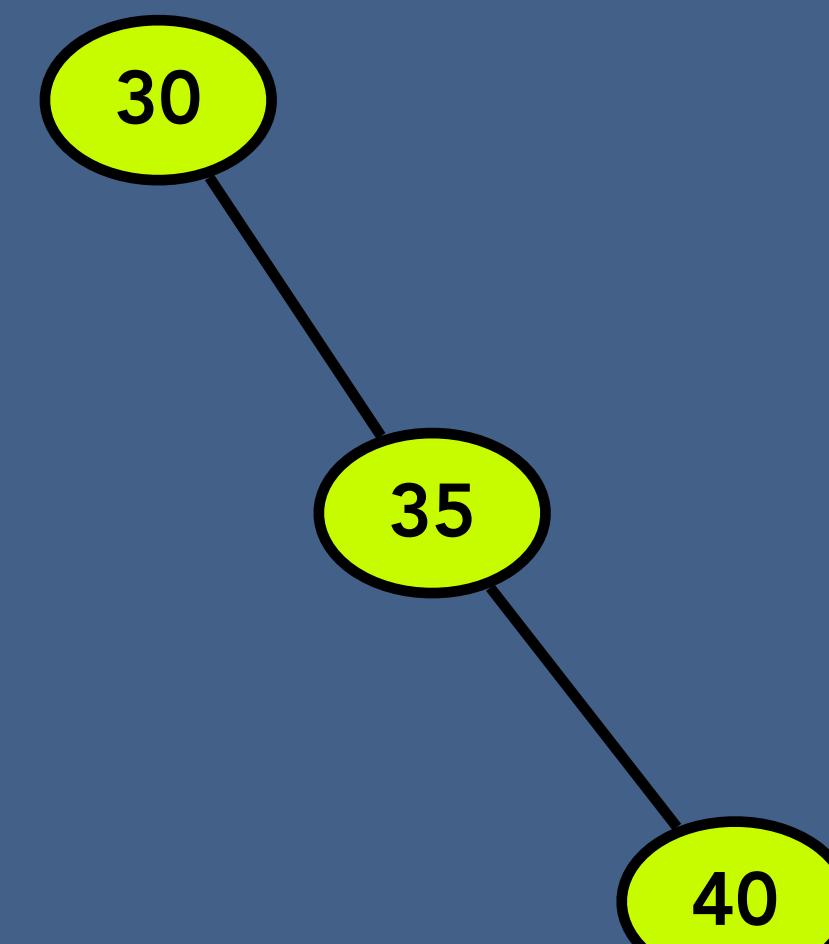
# AVL Tree - Insert a Node

RL - right left condition

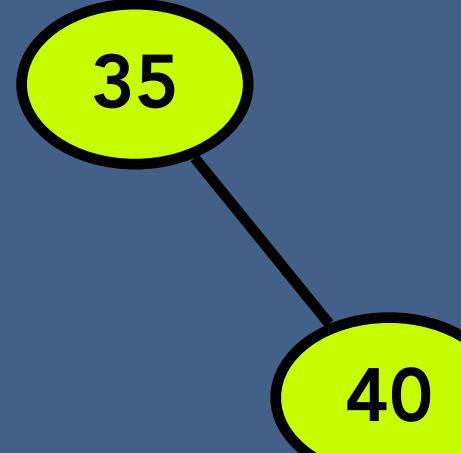
Step 1 : rotate Right disbalancedNode.rightChild

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
```



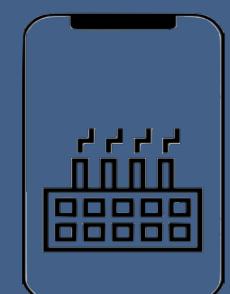
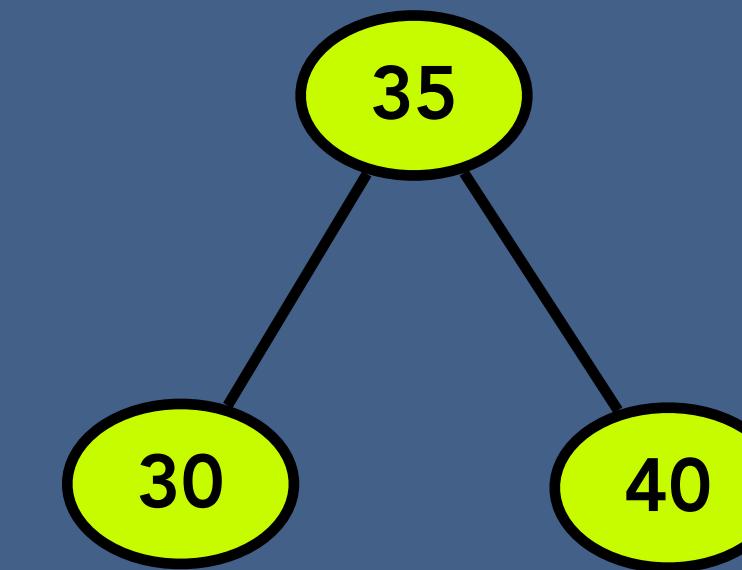
newRoot =



disbalancedNode =



newRoot =



# AVL Tree - Insert a Node

RL - right left condition

Step 1 : rotate Right disbalancedNode.rightChild

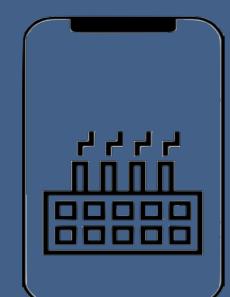
Step 2 : rotate Left disbalancedNode

```
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)

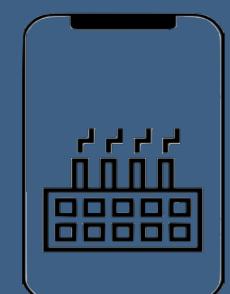


# AVL Tree - Insert a Node (All together)

Case 1 : Rotation is not required

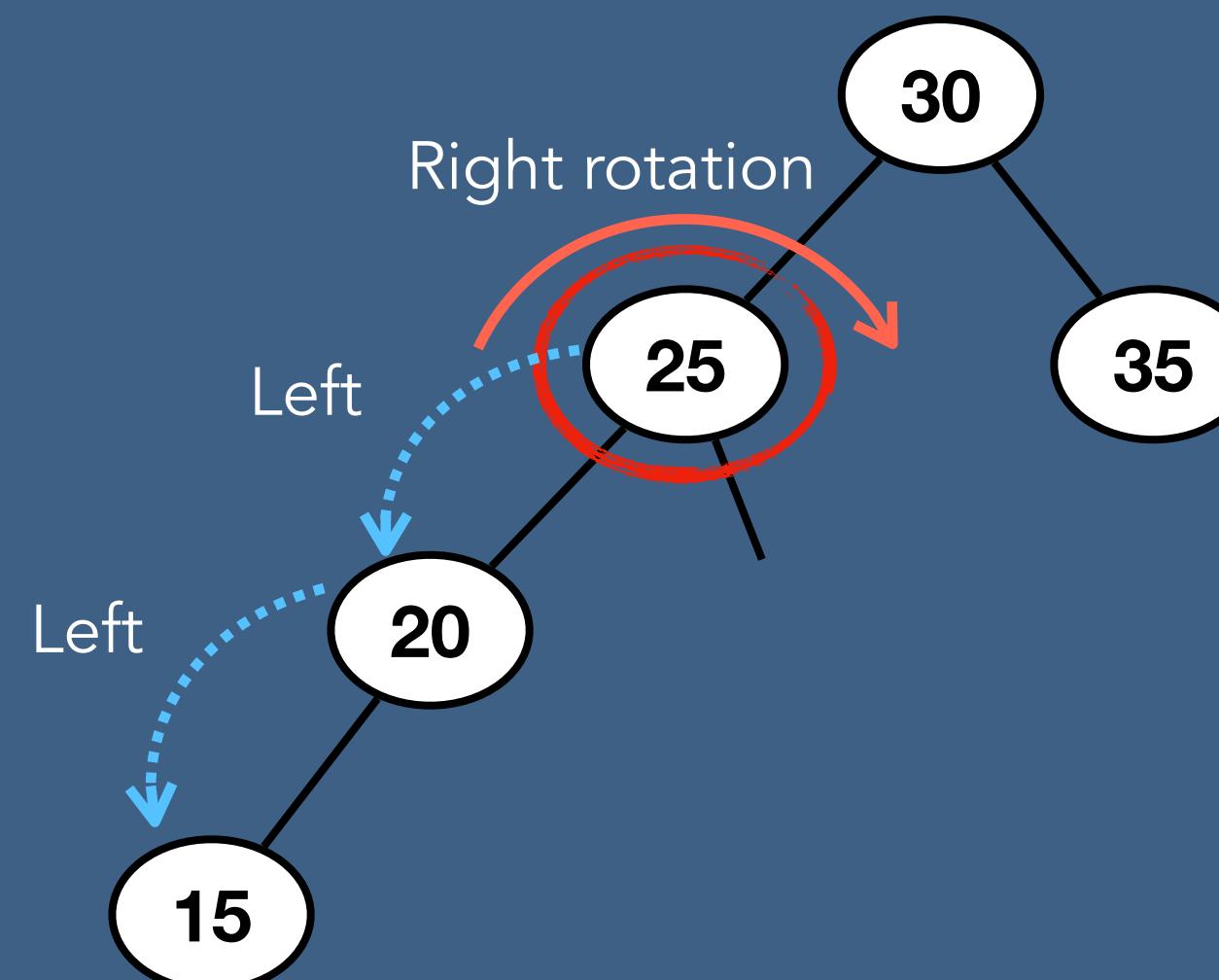
Case 2 : Rotation is required

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

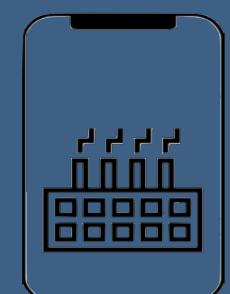


# AVL Tree - Insert a Node (All together)

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



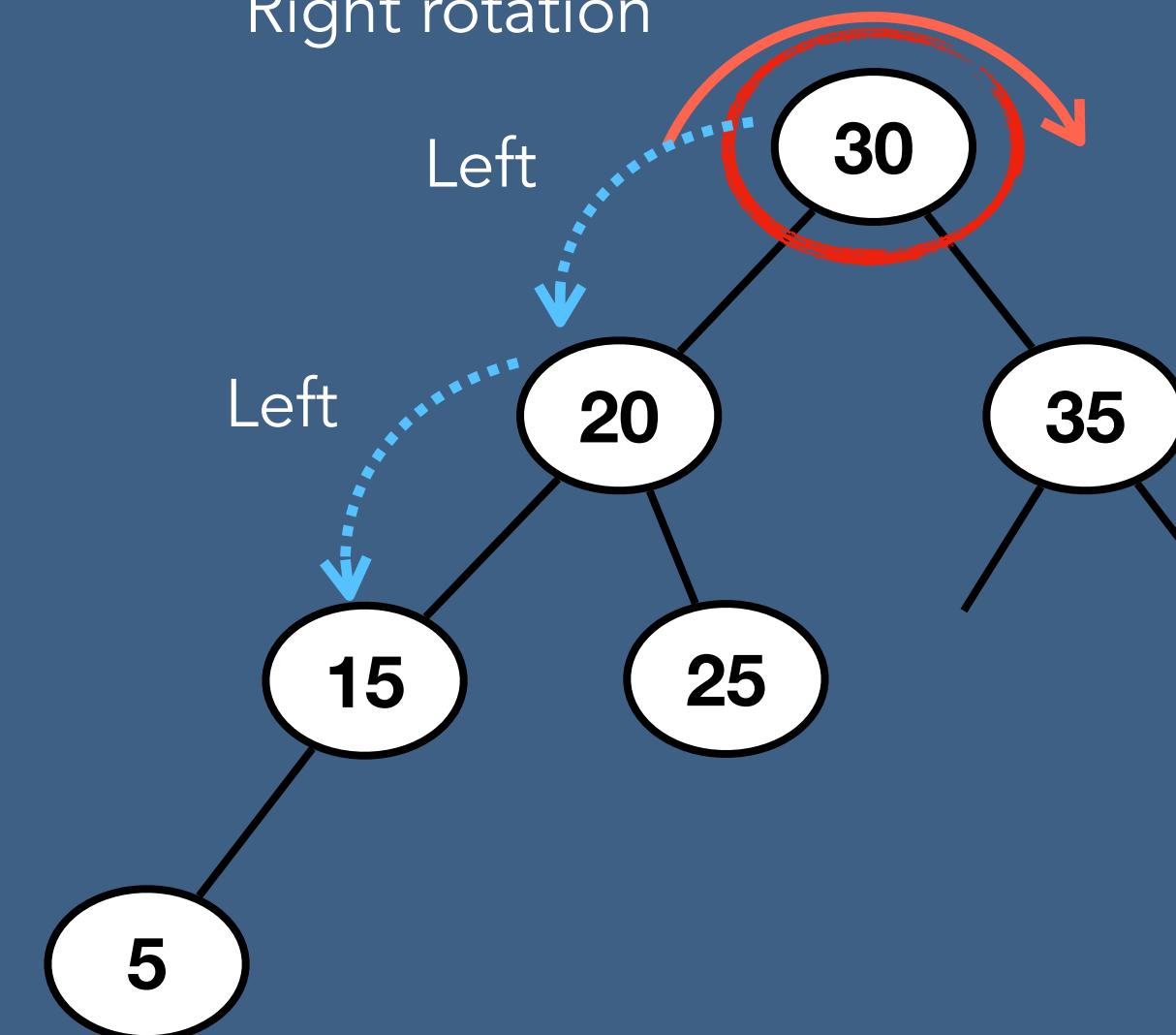
- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)



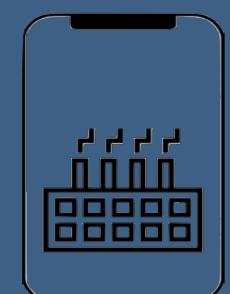
# AVL Tree - Insert a Node (All together)

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

Right rotation

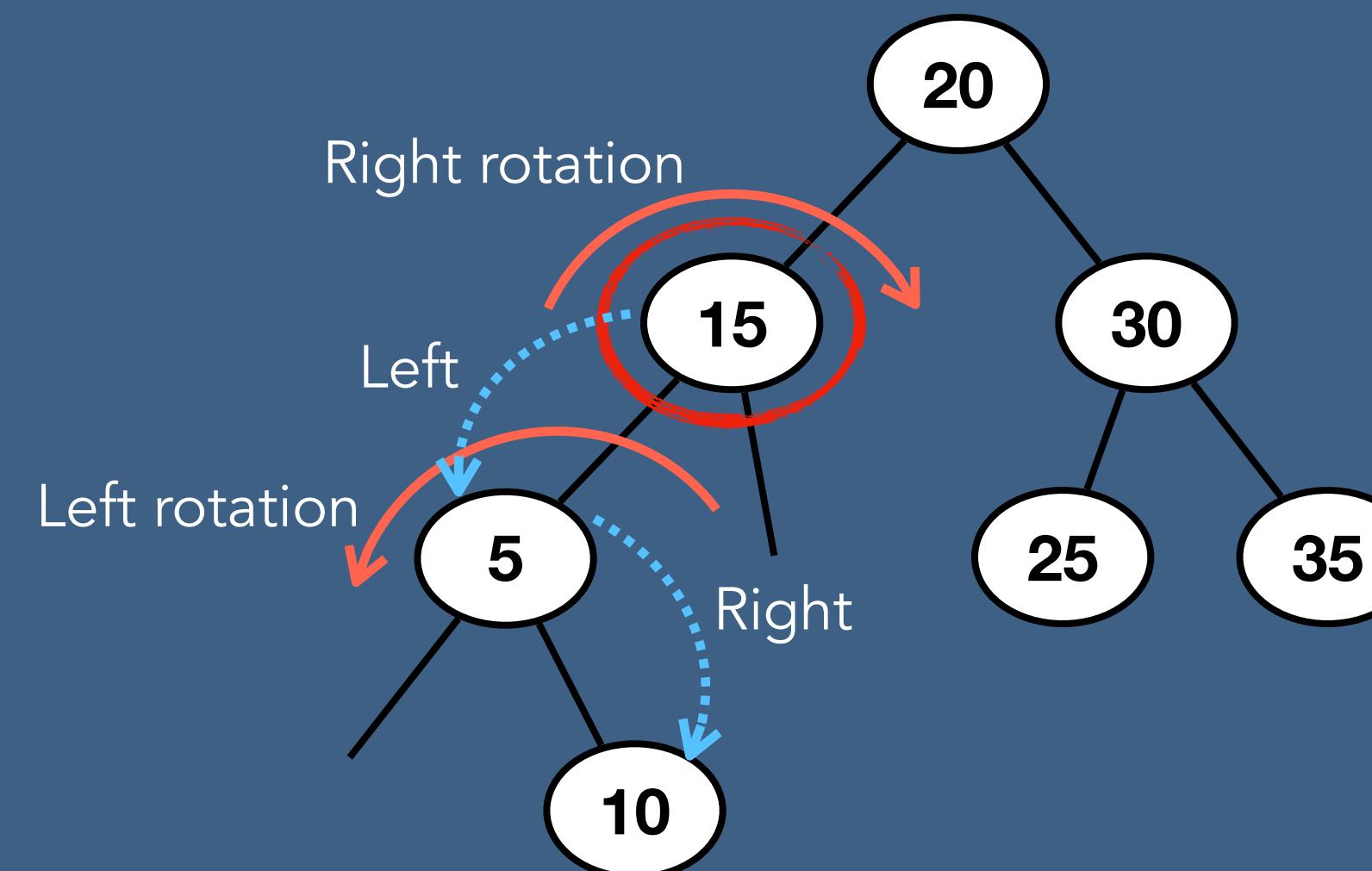


- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)

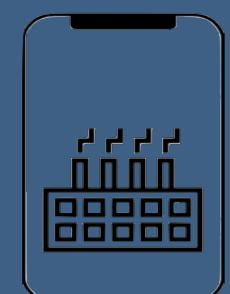


# AVL Tree - Insert a Node (All together)

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

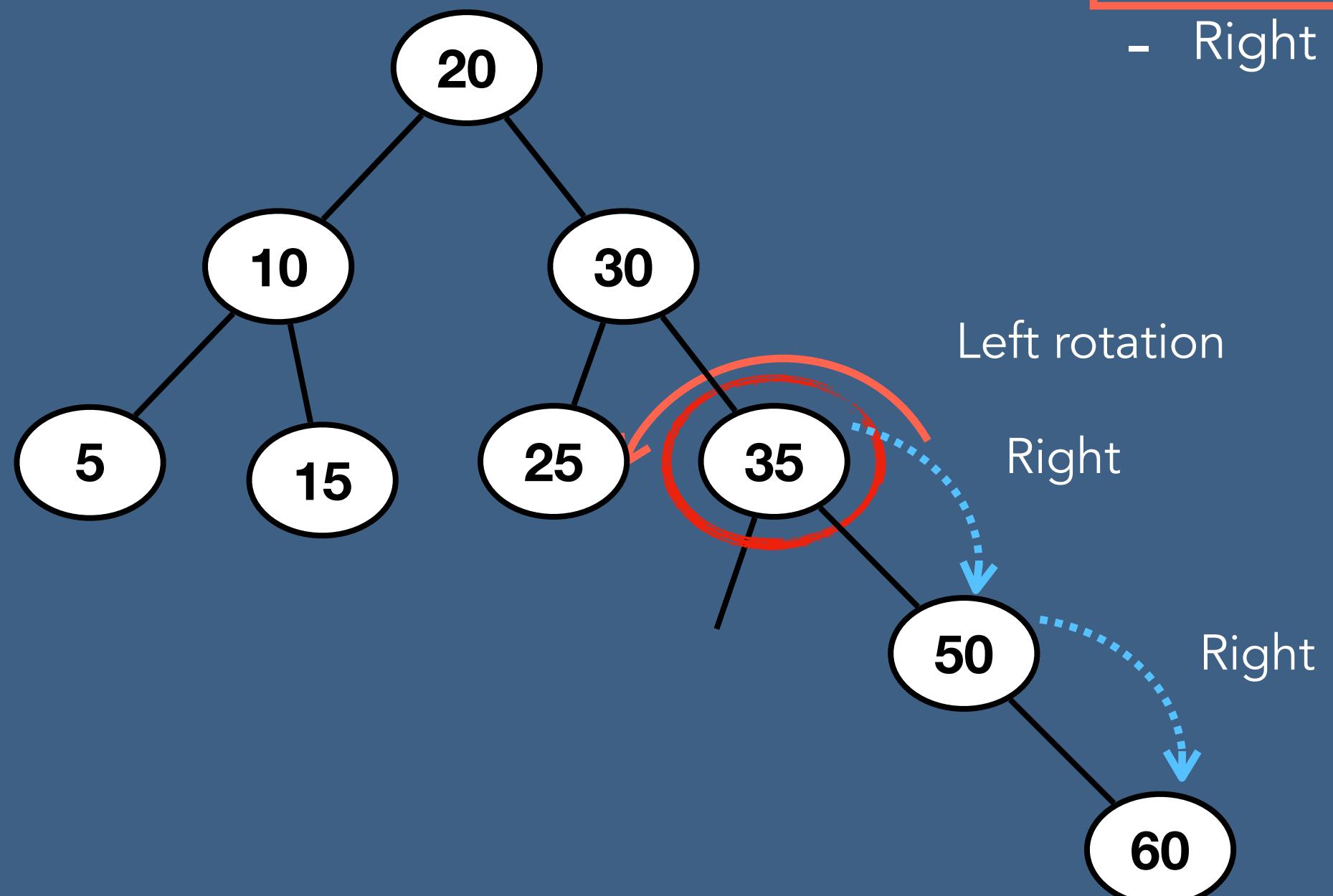


- Left Left condition (LL)
- **Left Right condition (LR)**
- Right Right condition (RR)
- Right Left condition (RL)

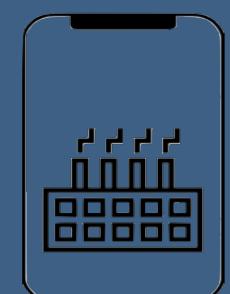


# AVL Tree - Insert a Node (All together)

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

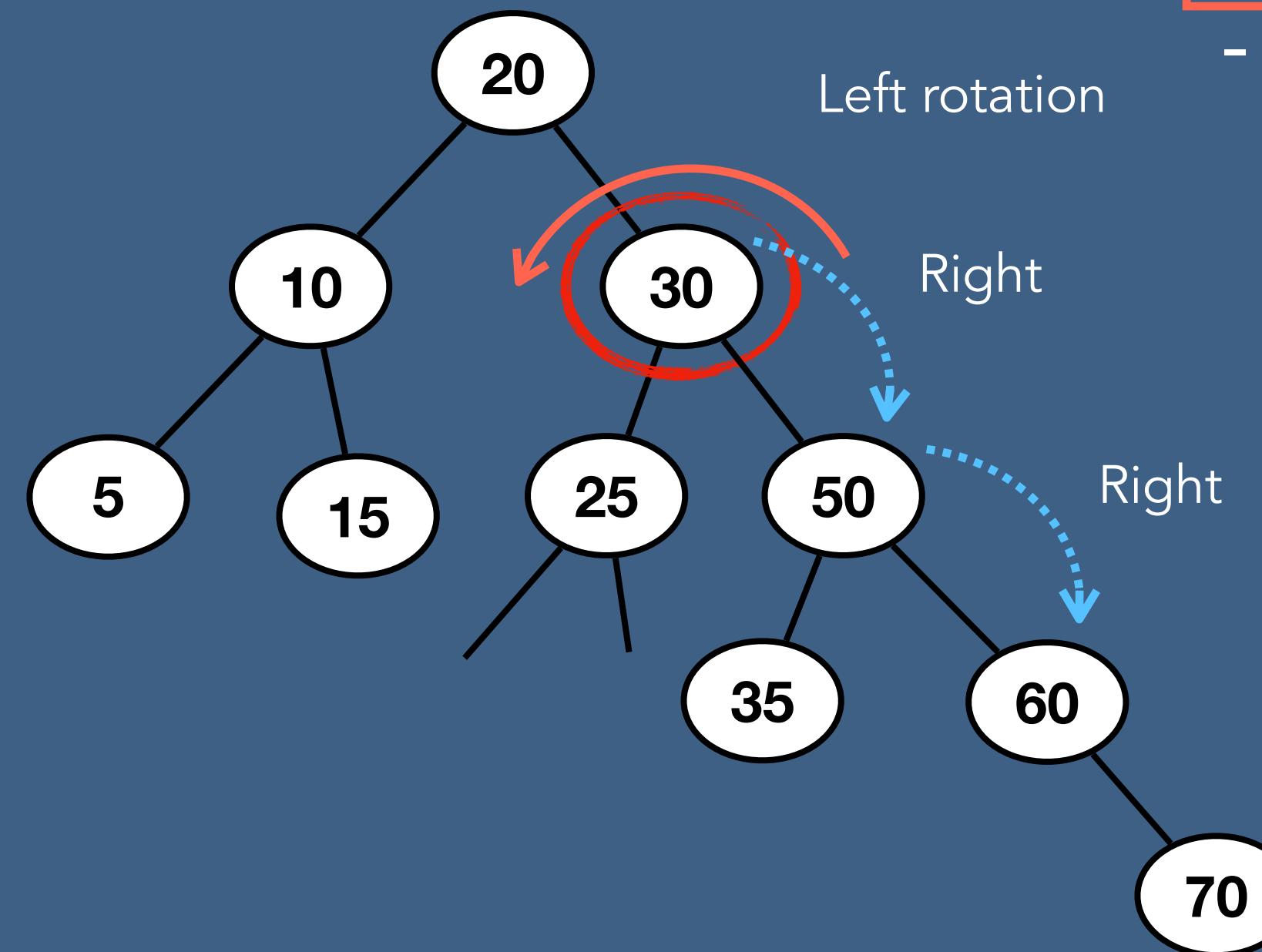


- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR) (highlighted)
- Right Left condition (RL)

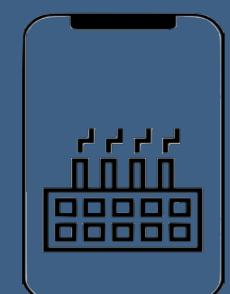


# AVL Tree - Insert a Node (All together)

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



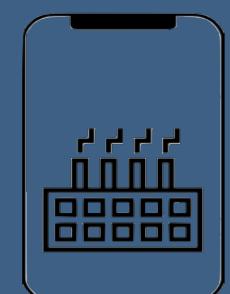
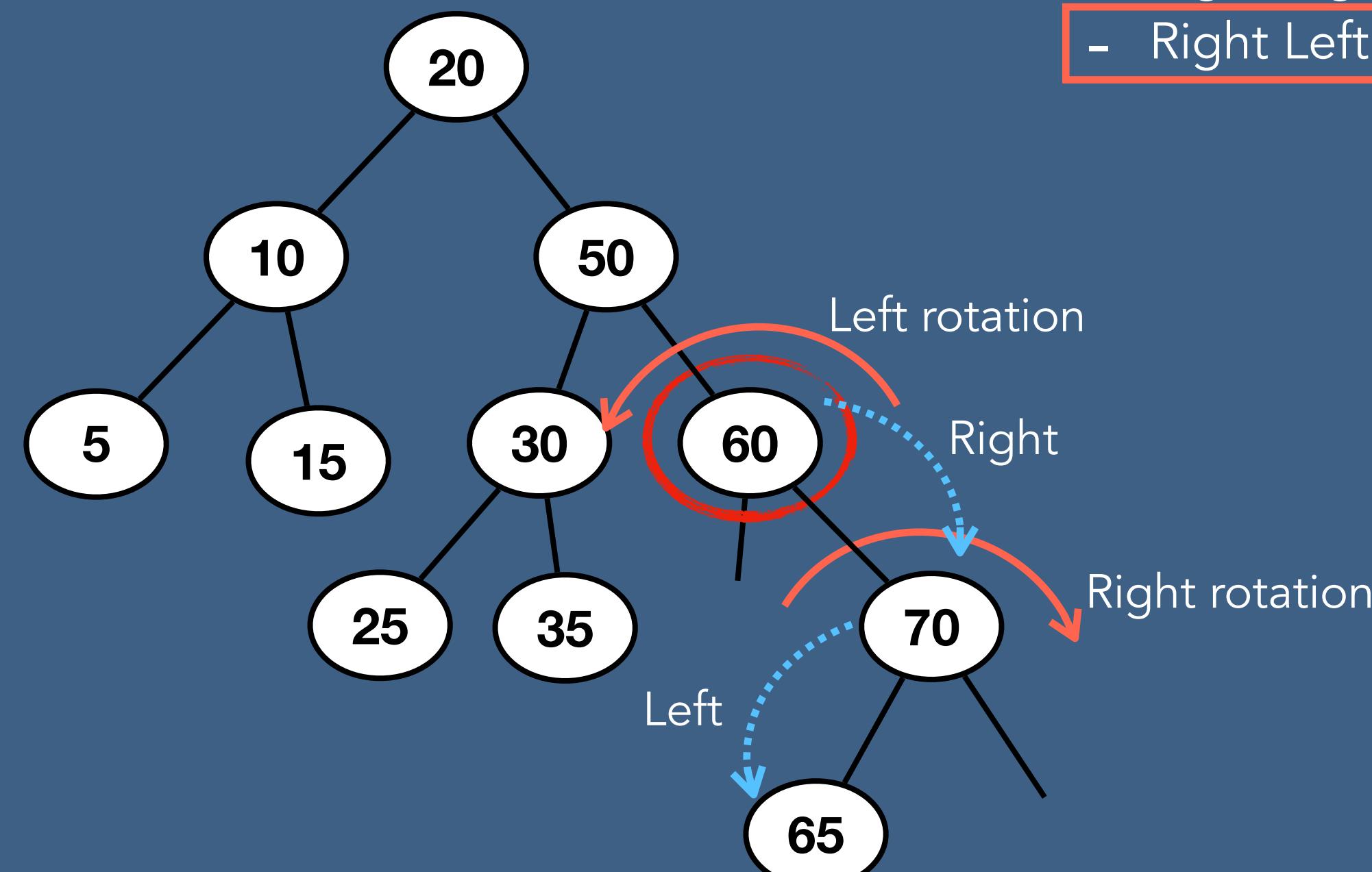
- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR) (highlighted)
- Right Left condition (RL)



# AVL Tree - Insert a Node (All together)

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

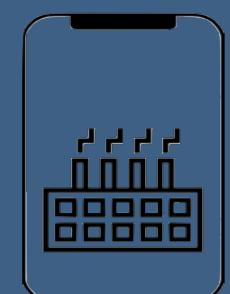
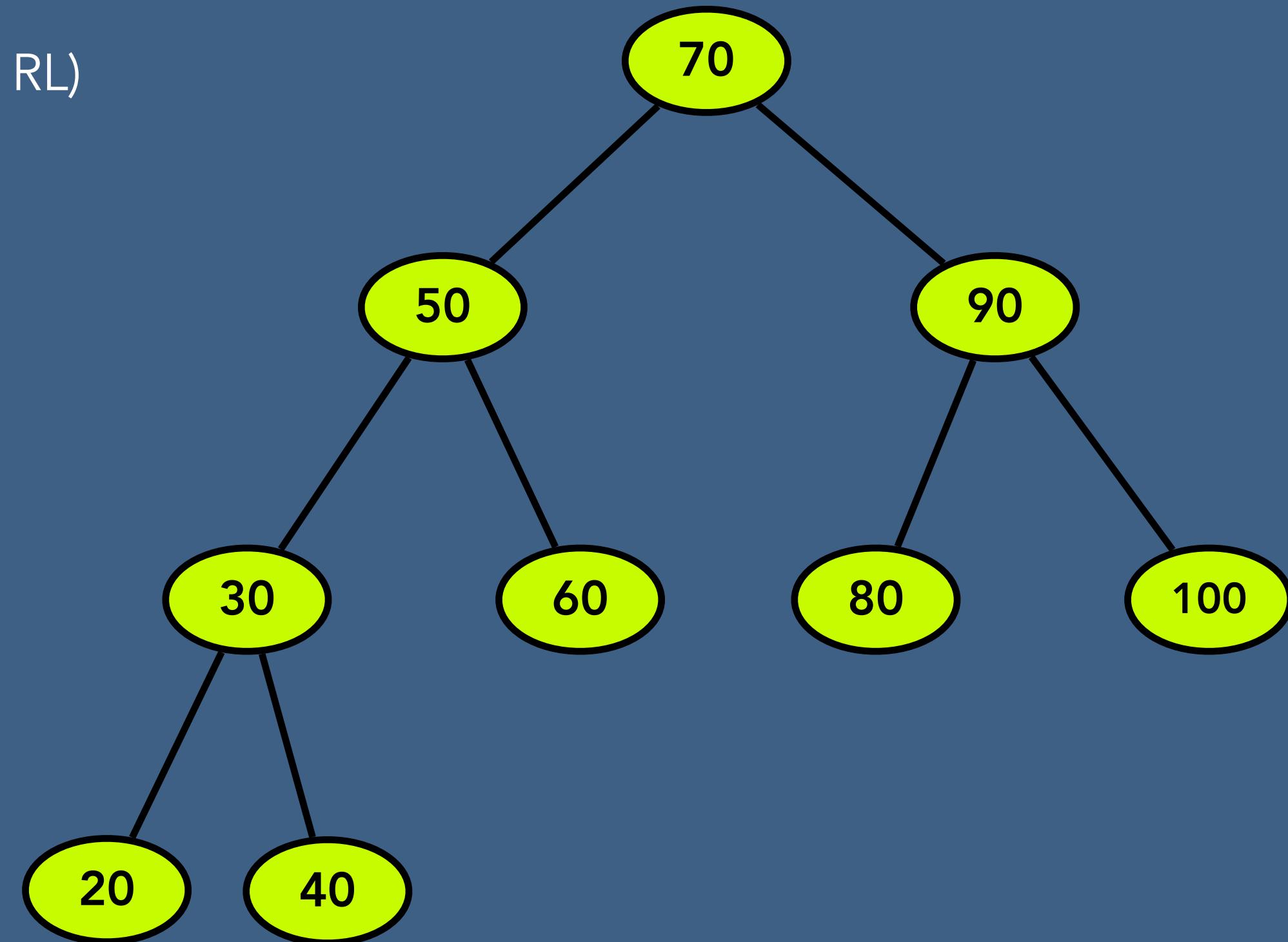
- Left Left condition (LL)
- Left Right condition (LR)
- Right Right condition (RR)
- Right Left condition (RL)



# AVL Tree - Delete a Node

Case 1 - Rotation is not required

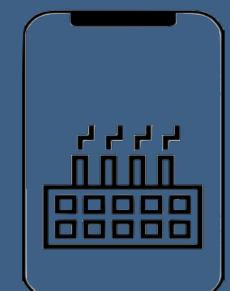
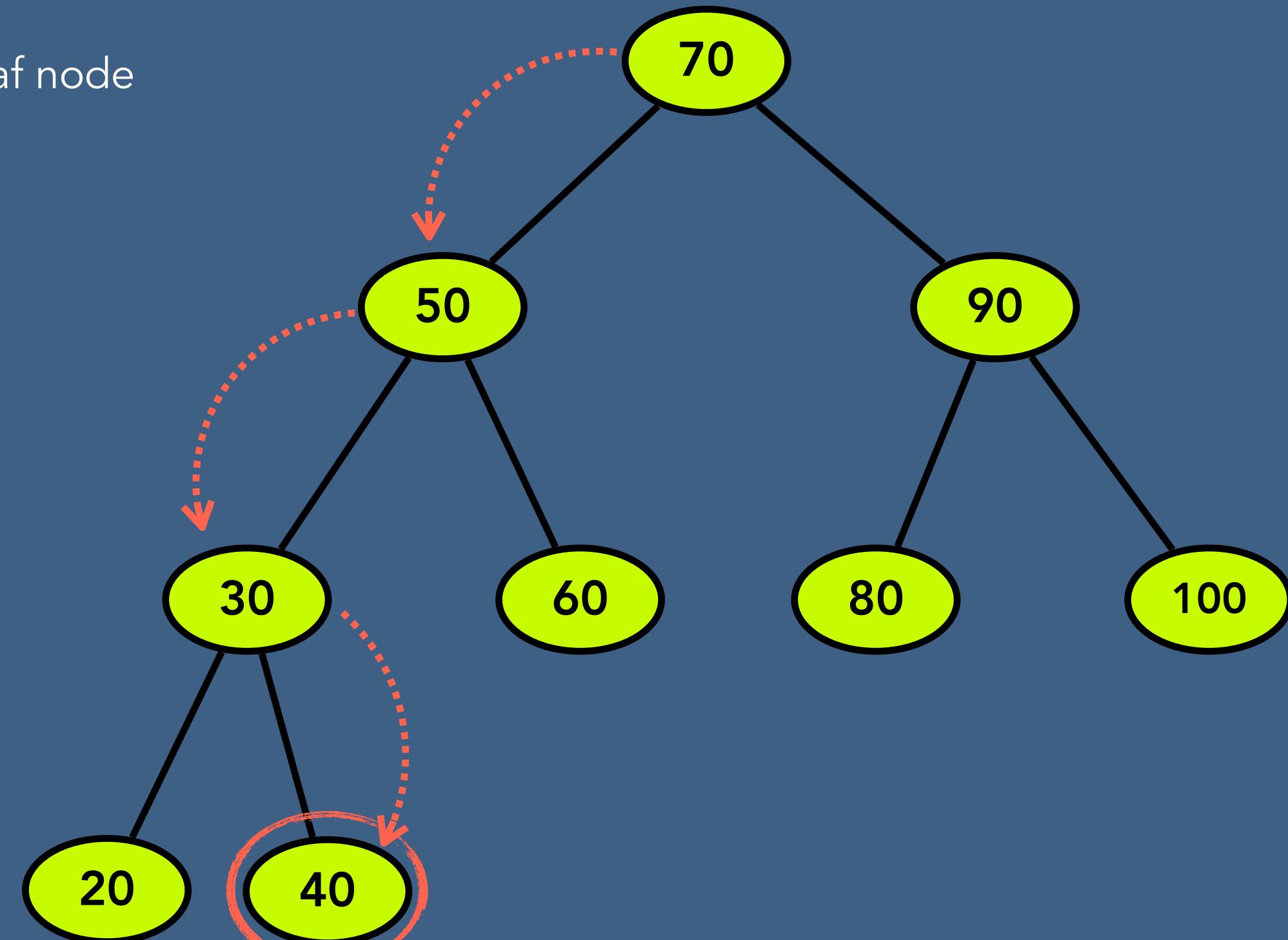
Case 2 - Rotation is required (LL, LR, RR, RL)



# AVL Tree - Delete a Node

## Case 1 - Rotation is not required

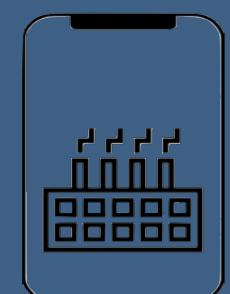
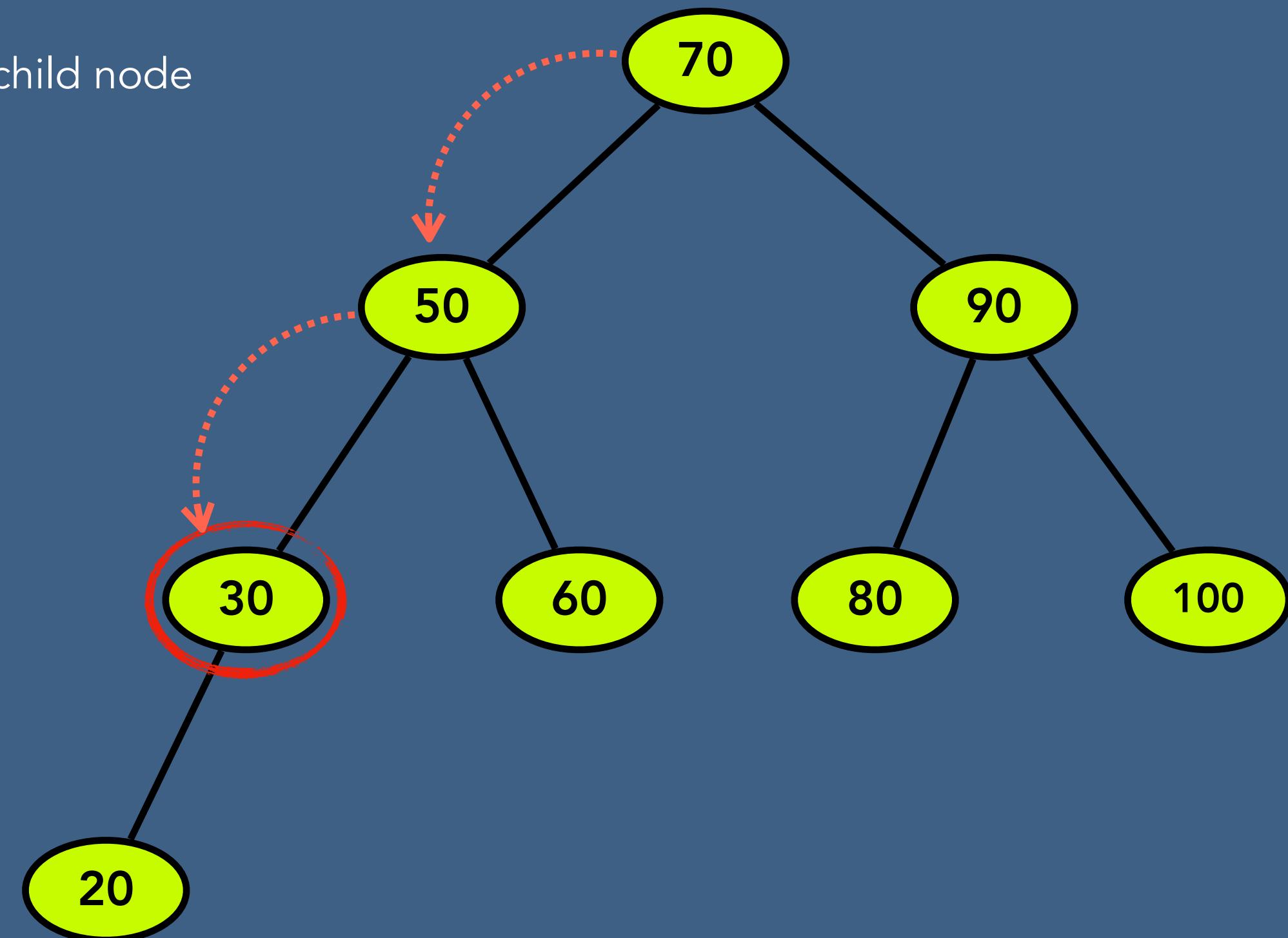
- The node to be deleted is a leaf node



# AVL Tree - Delete a Node

## Case 1 - Rotation is not required

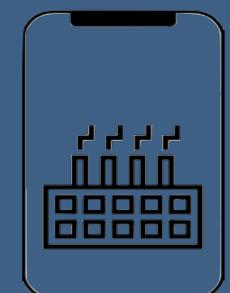
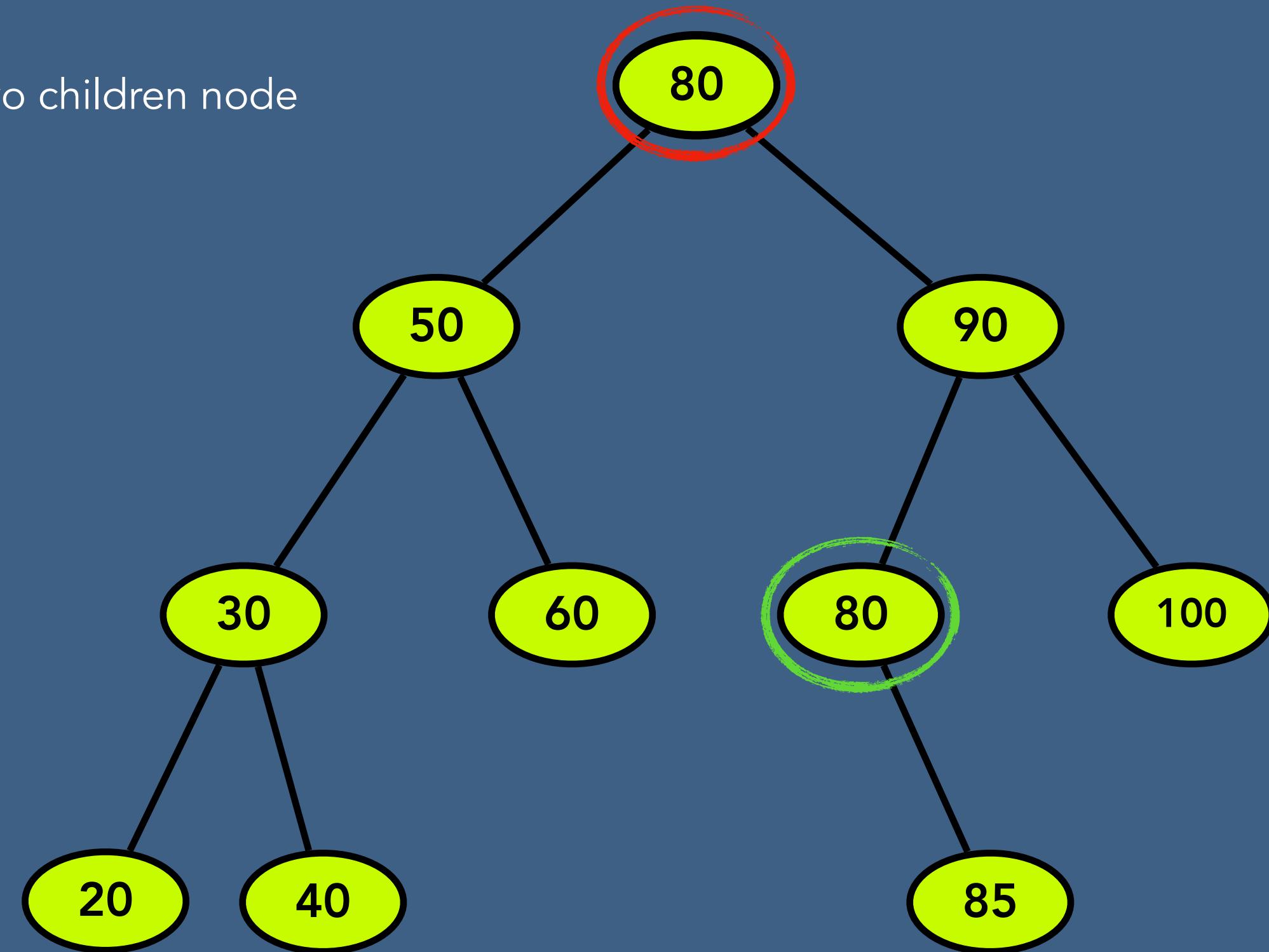
- The node to be deleted has a child node



# AVL Tree - Delete a Node

## Case 1 - Rotation is not required

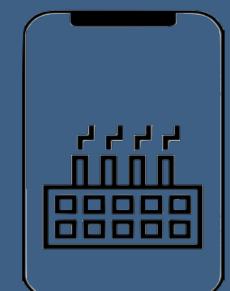
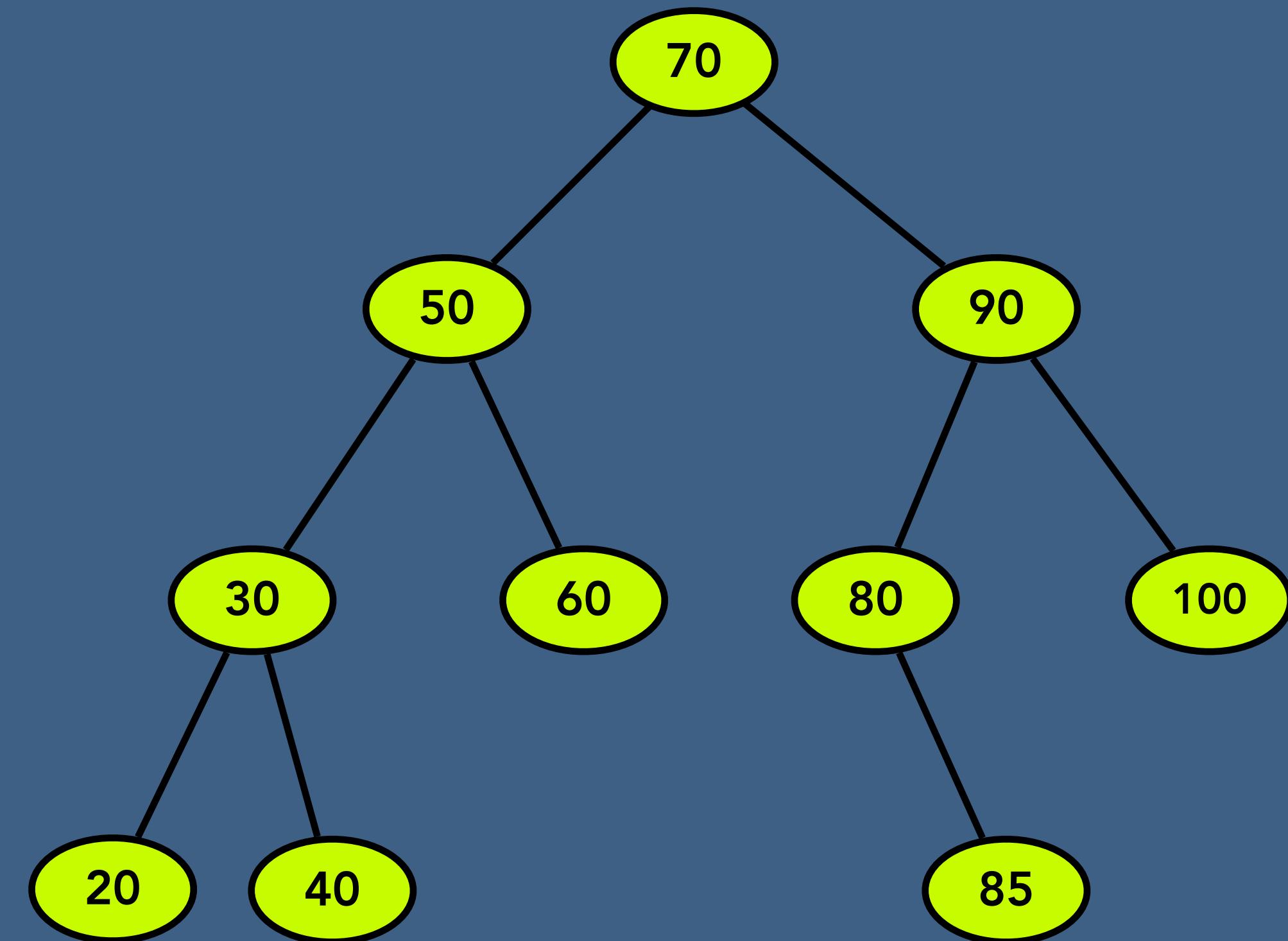
- The node to be deleted has two children node



# AVL Tree - Delete a Node

## Case 2 - Rotation is required

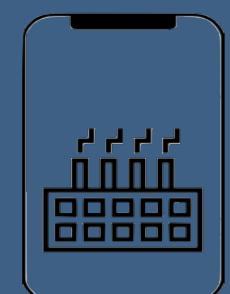
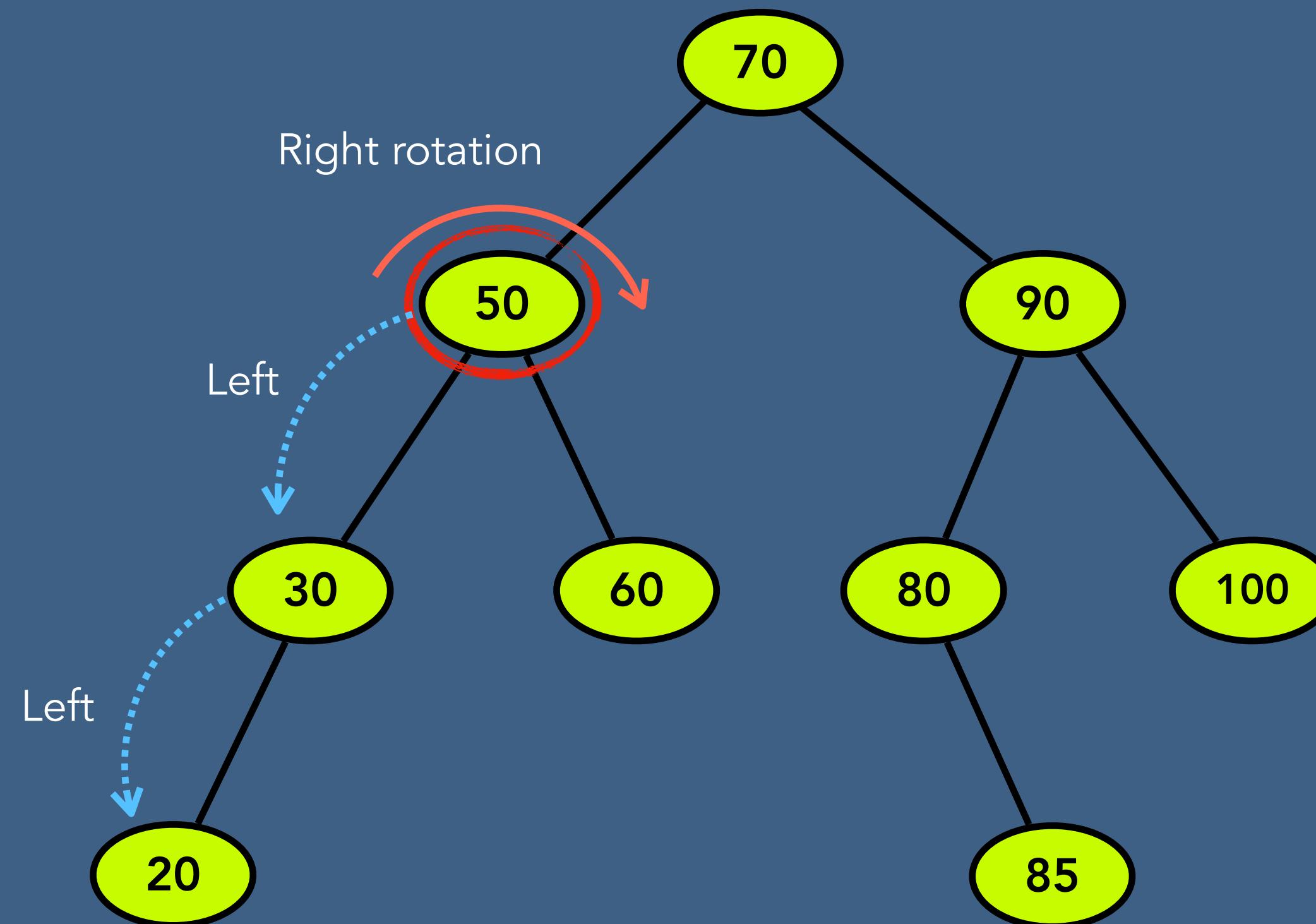
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



# AVL Tree - Delete a Node

## Case 2 - Rotation is required

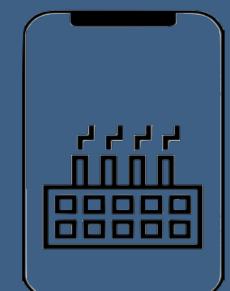
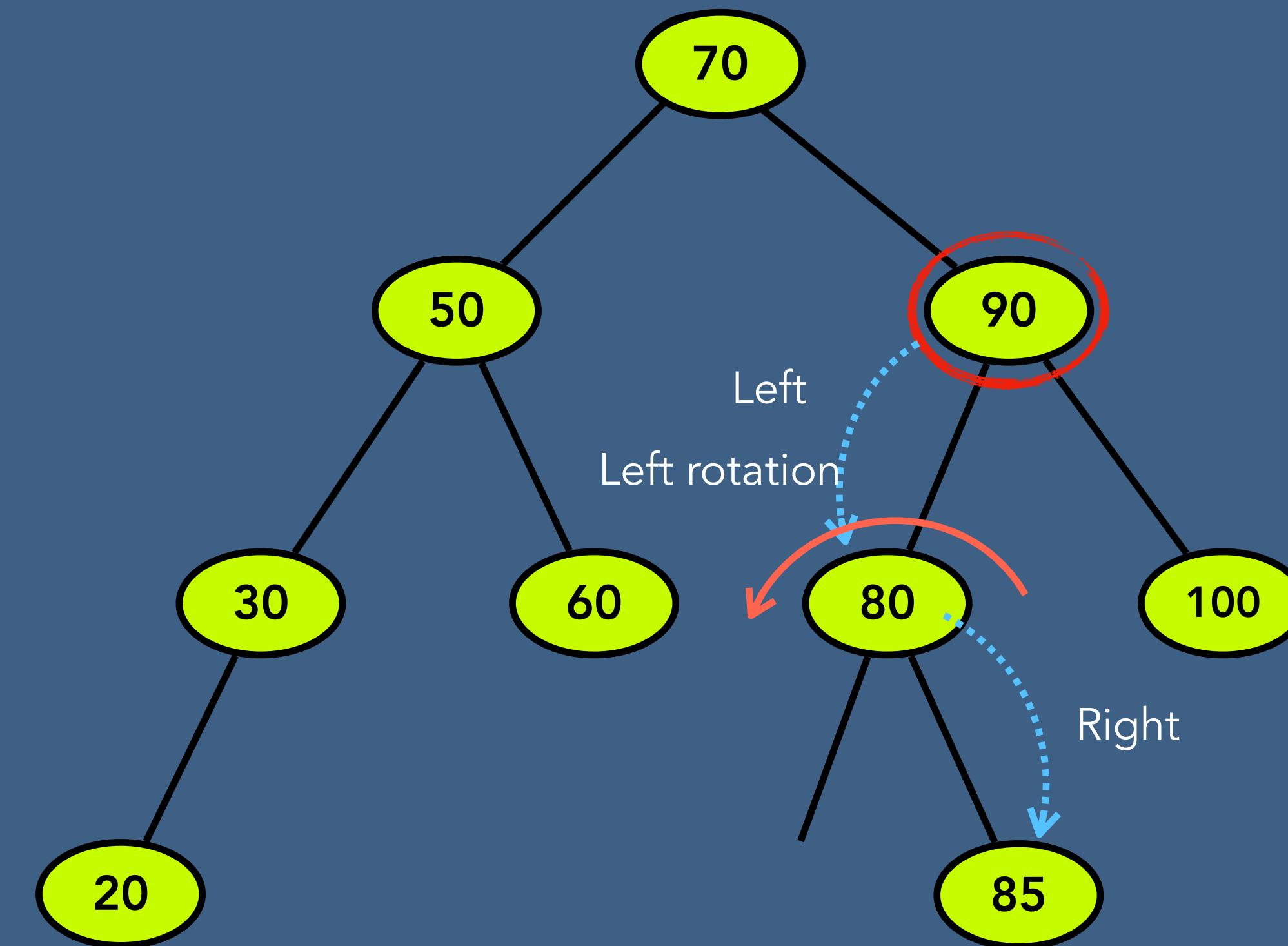
Left Left Condition (LL)



# AVL Tree - Delete a Node

## Case 2 - Rotation is required

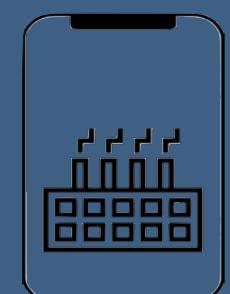
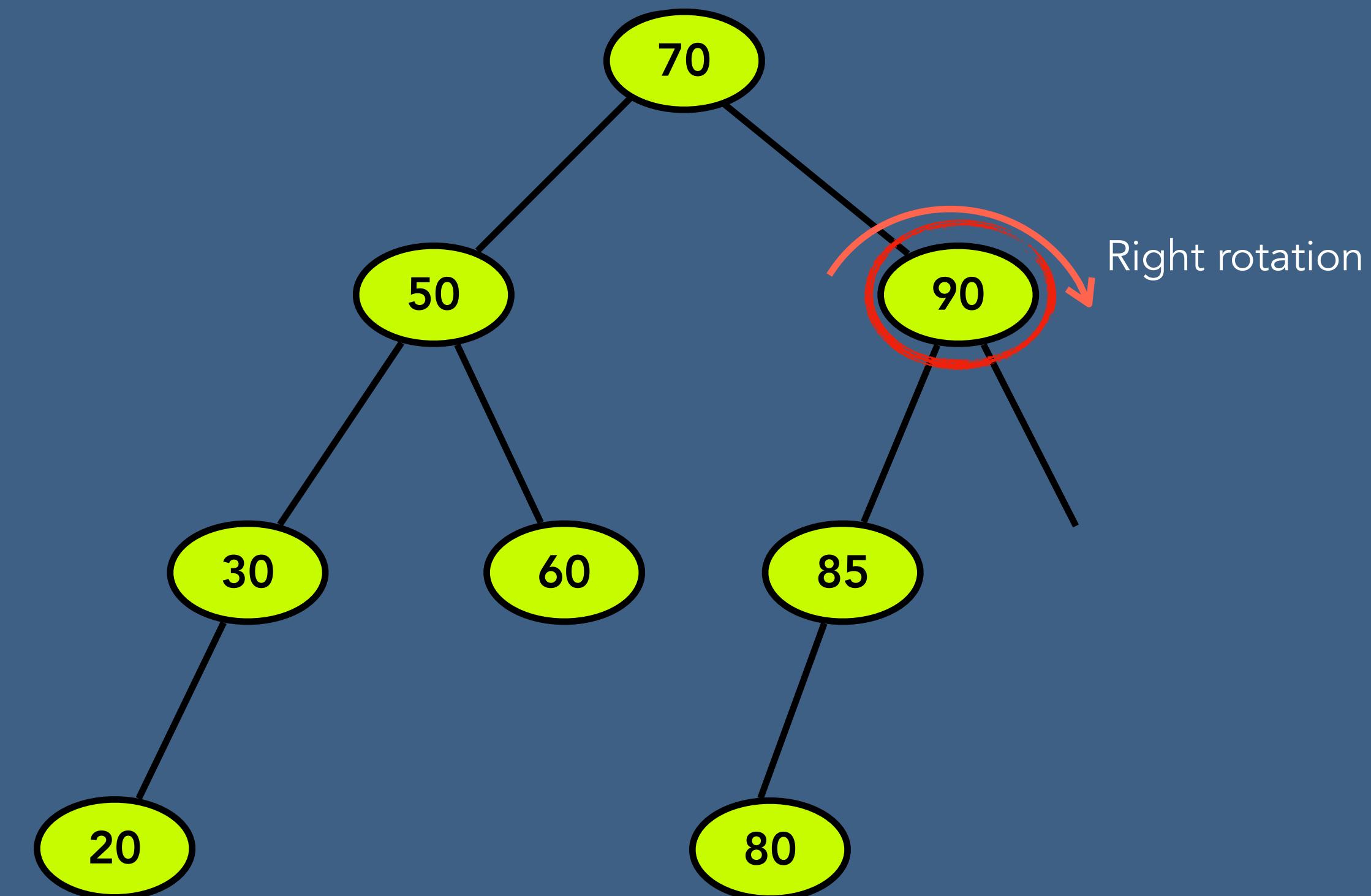
Left Right Condition (LR)



# AVL Tree - Delete a Node

Case 2 - Rotation is required

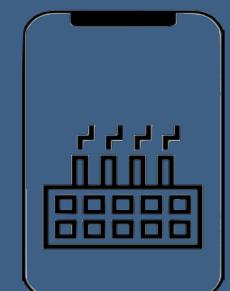
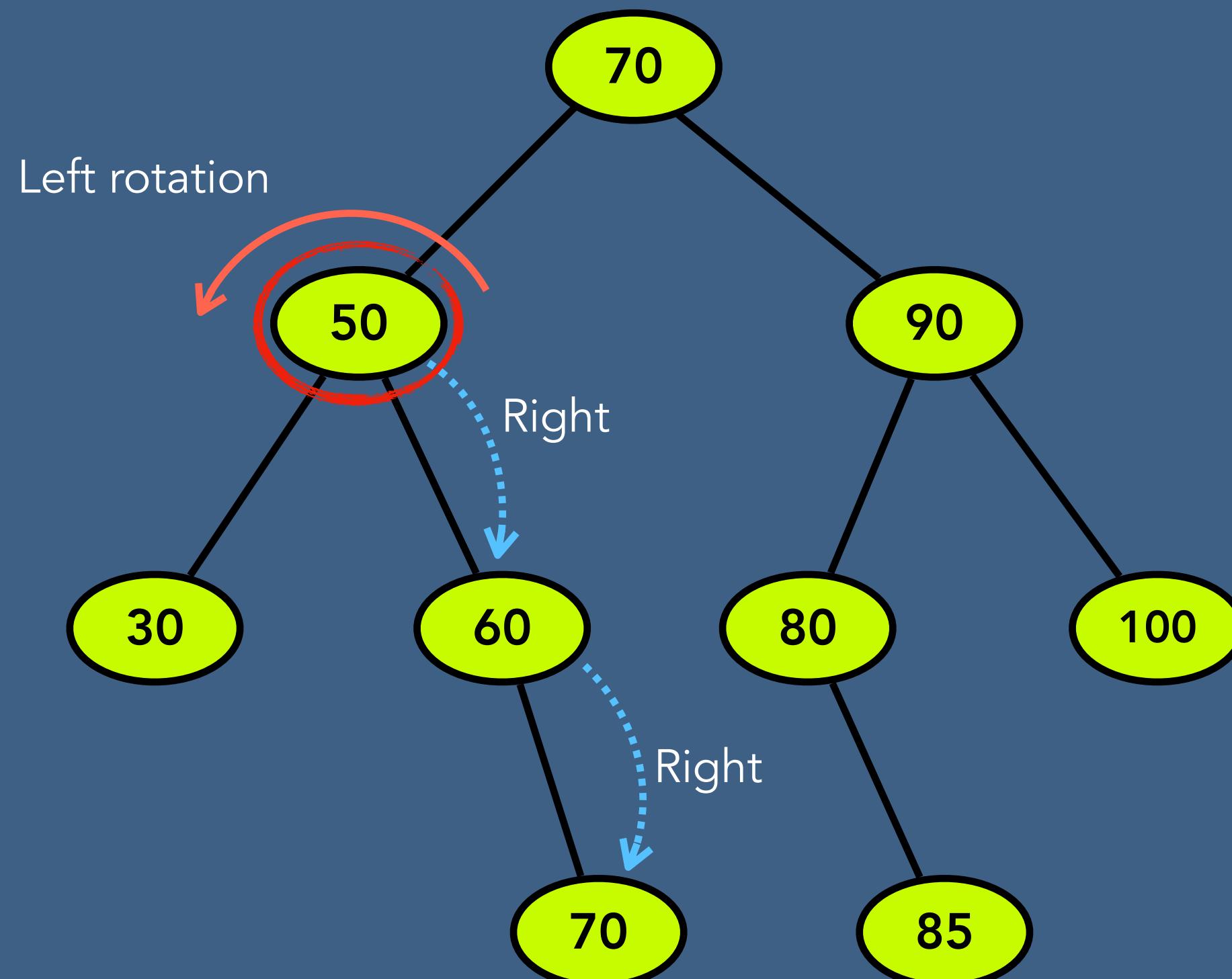
Left Right Condition (LR)



# AVL Tree - Delete a Node

## Case 2 - Rotation is required

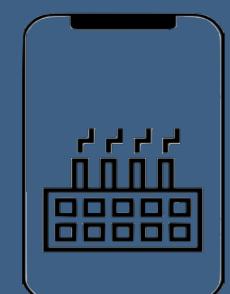
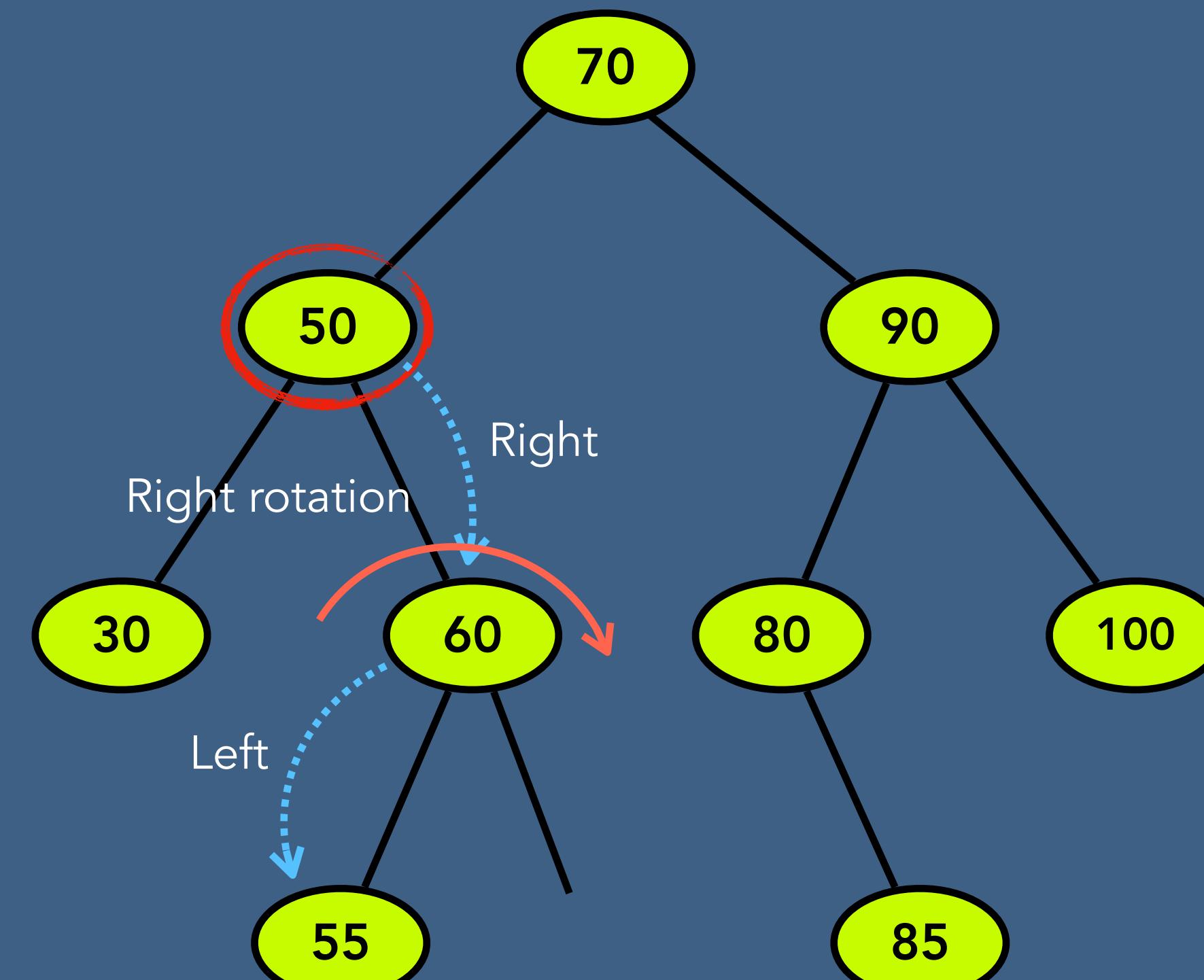
Right Right Condition (RR)



# AVL Tree - Delete a Node

## Case 2 - Rotation is required

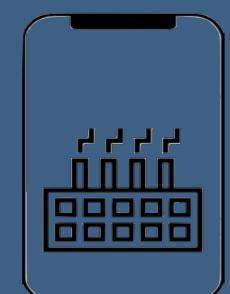
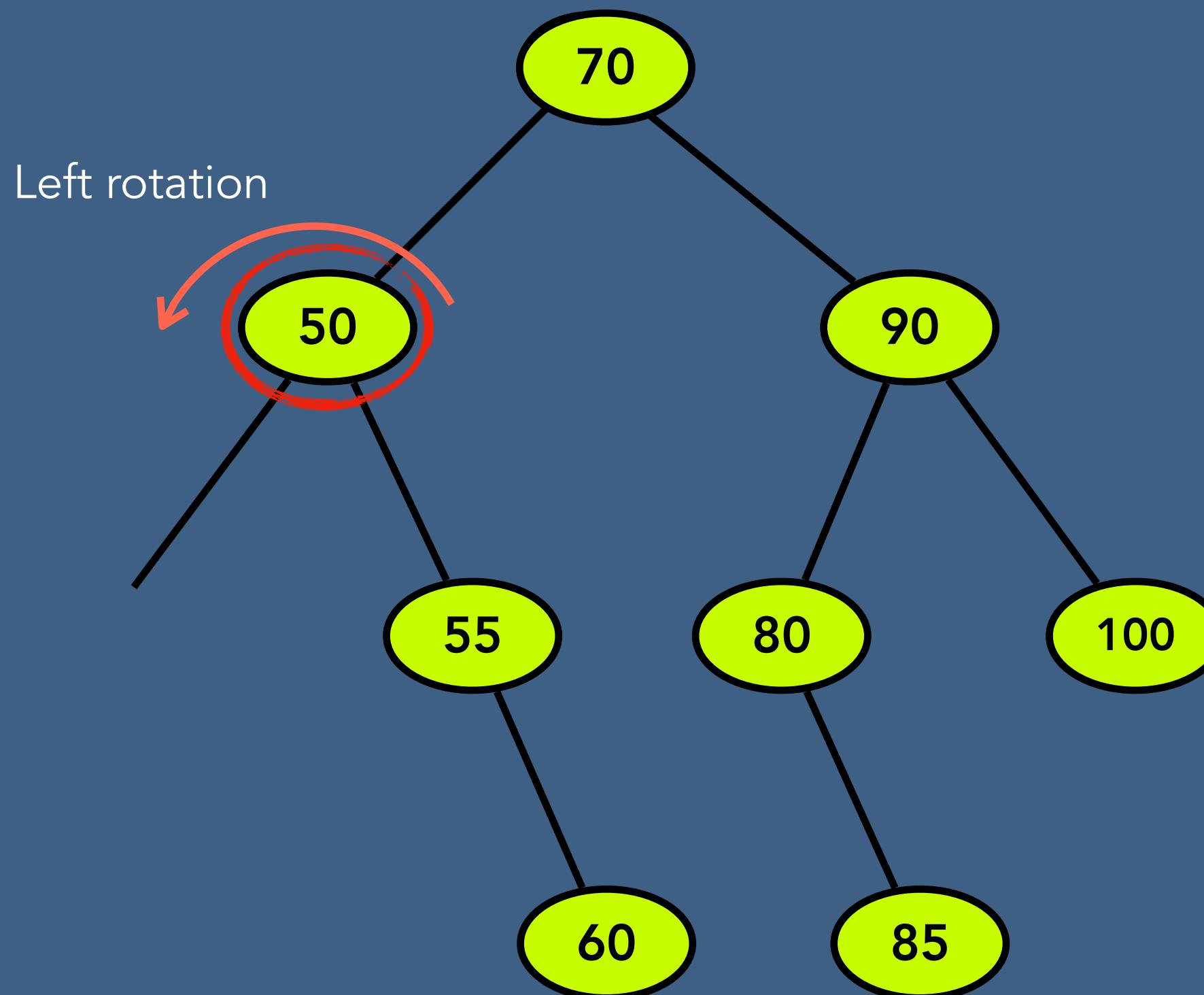
Right Left Condition (RL)



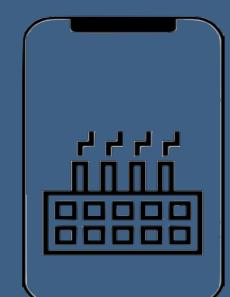
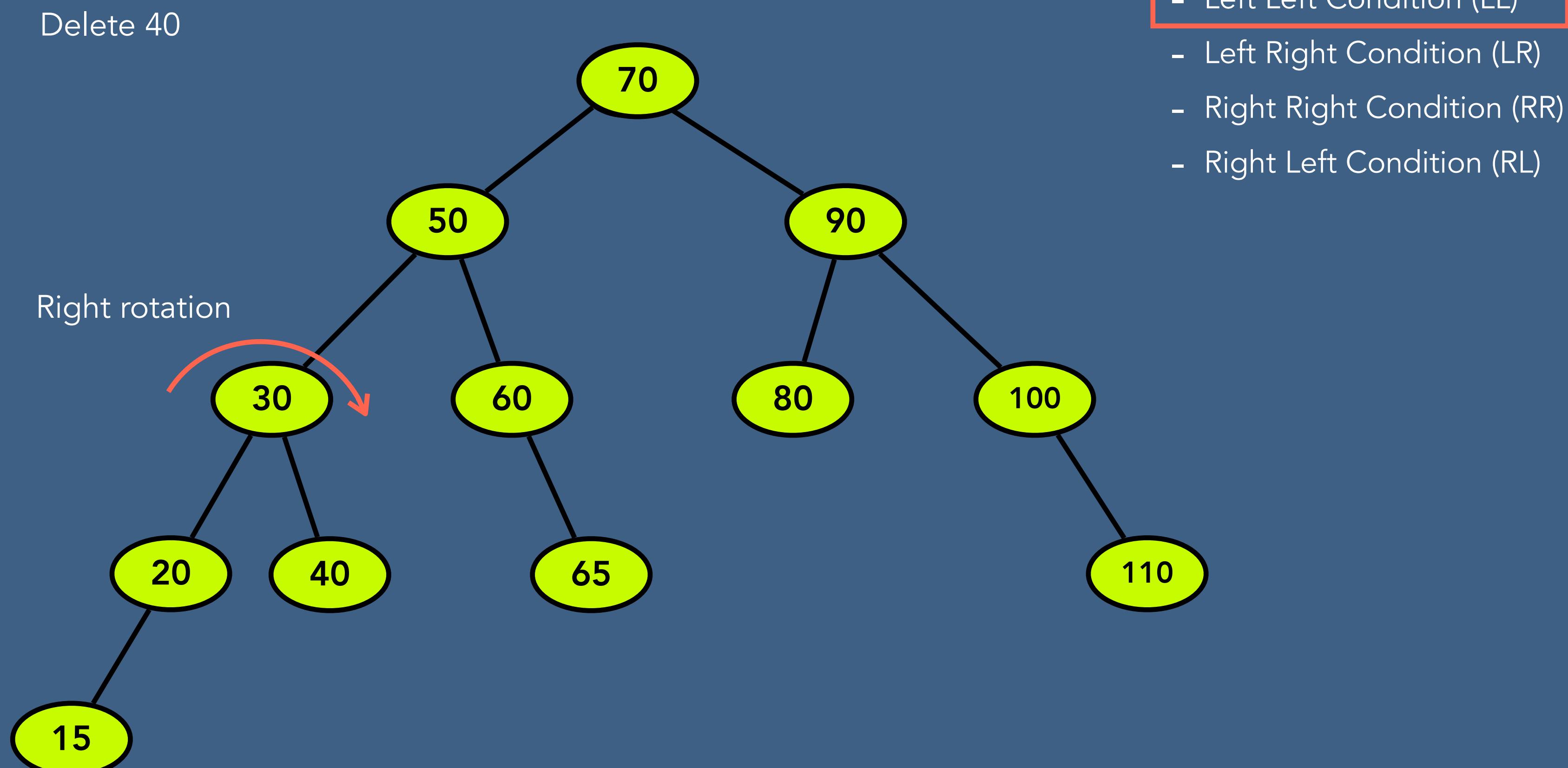
# AVL Tree - Delete a Node

## Case 2 - Rotation is required

Right Left Condition (RL)



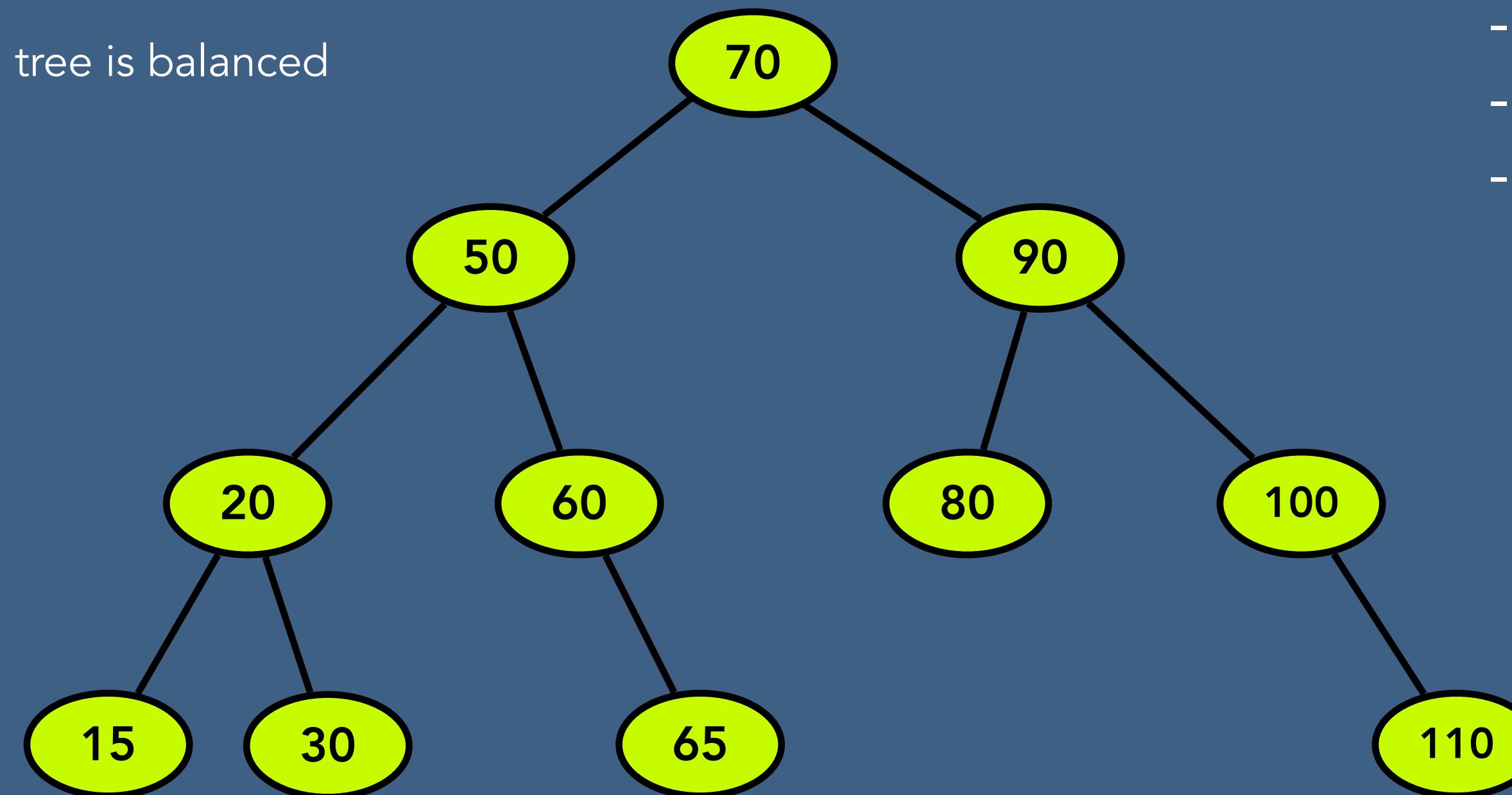
# AVL Tree - Delete a Node (all together)



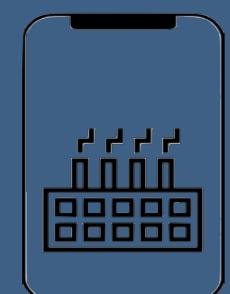
# AVL Tree - Delete a Node (all together)

Delete 15

The tree is balanced



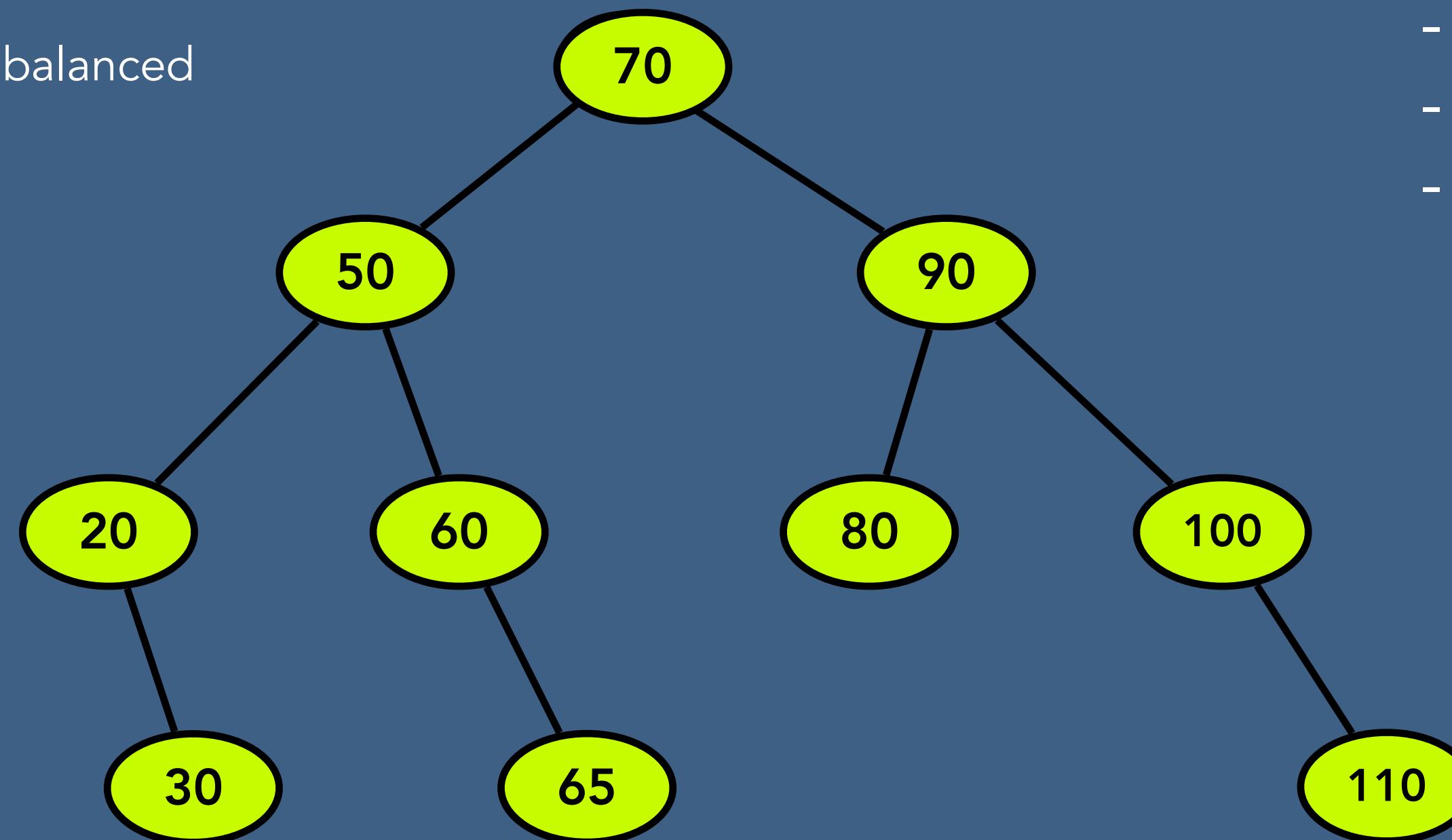
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



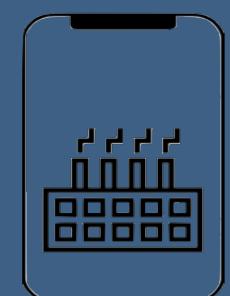
# AVL Tree - Delete a Node (all together)

Delete 65

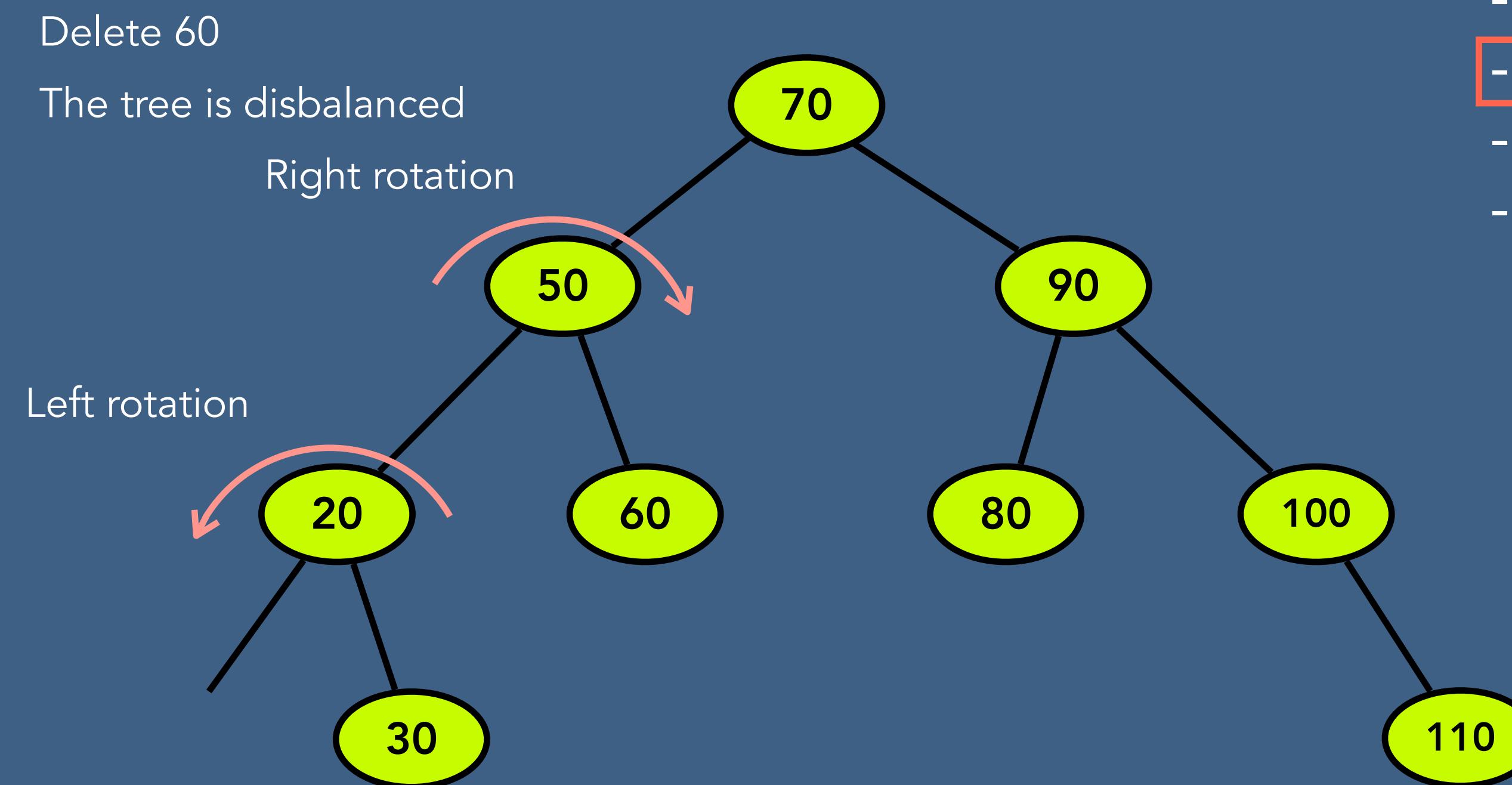
The tree is balanced



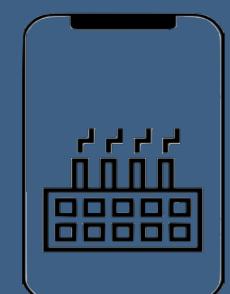
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



# AVL Tree - Delete a Node (all together)



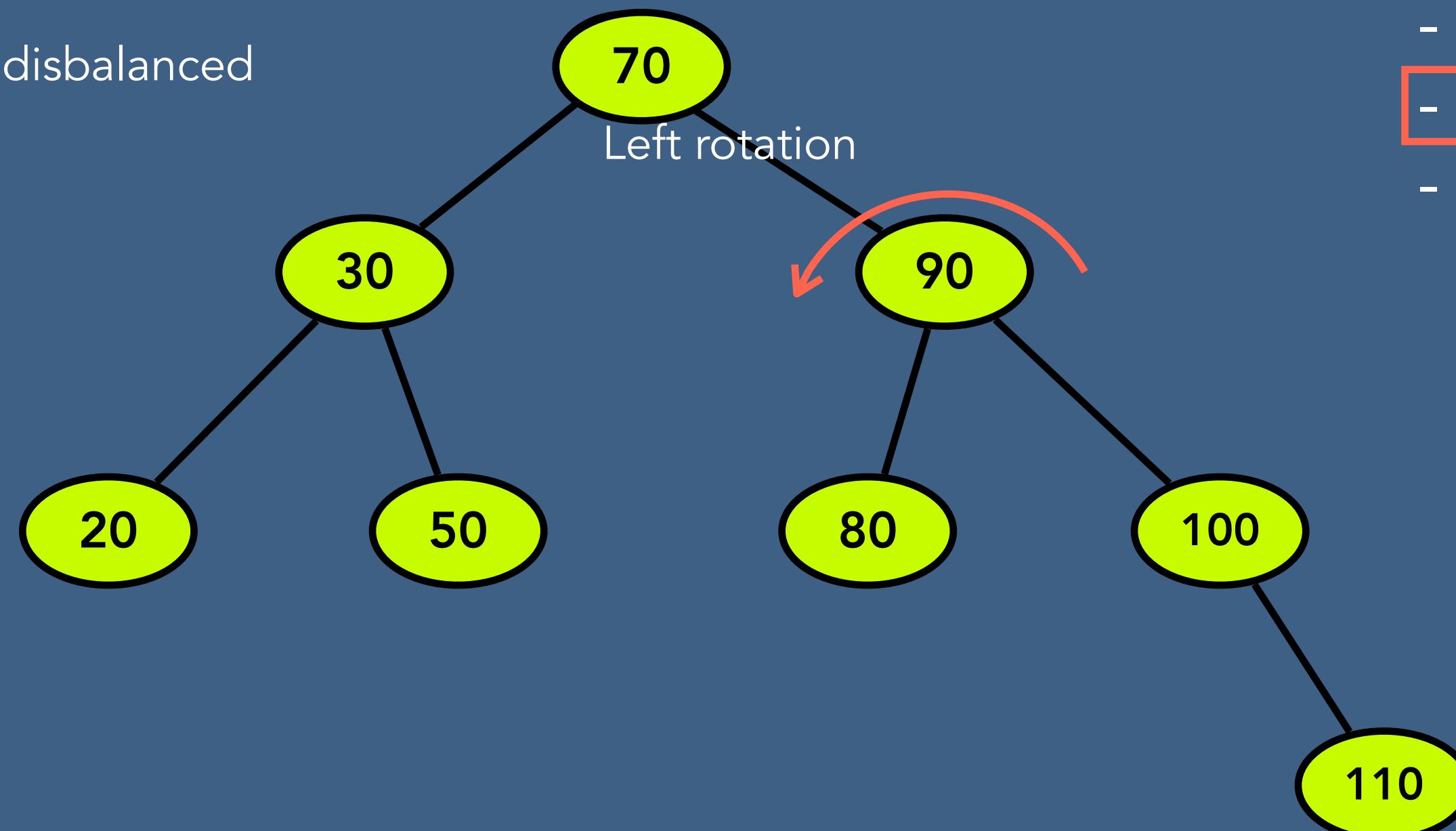
- Left Left Condition (LL)
- **Left Right Condition (LR)**
- Right Right Condition (RR)
- Right Left Condition (RL)



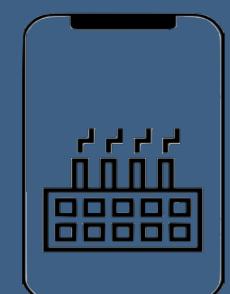
# AVL Tree - Delete a Node (all together)

Delete 80

The tree is disbalanced

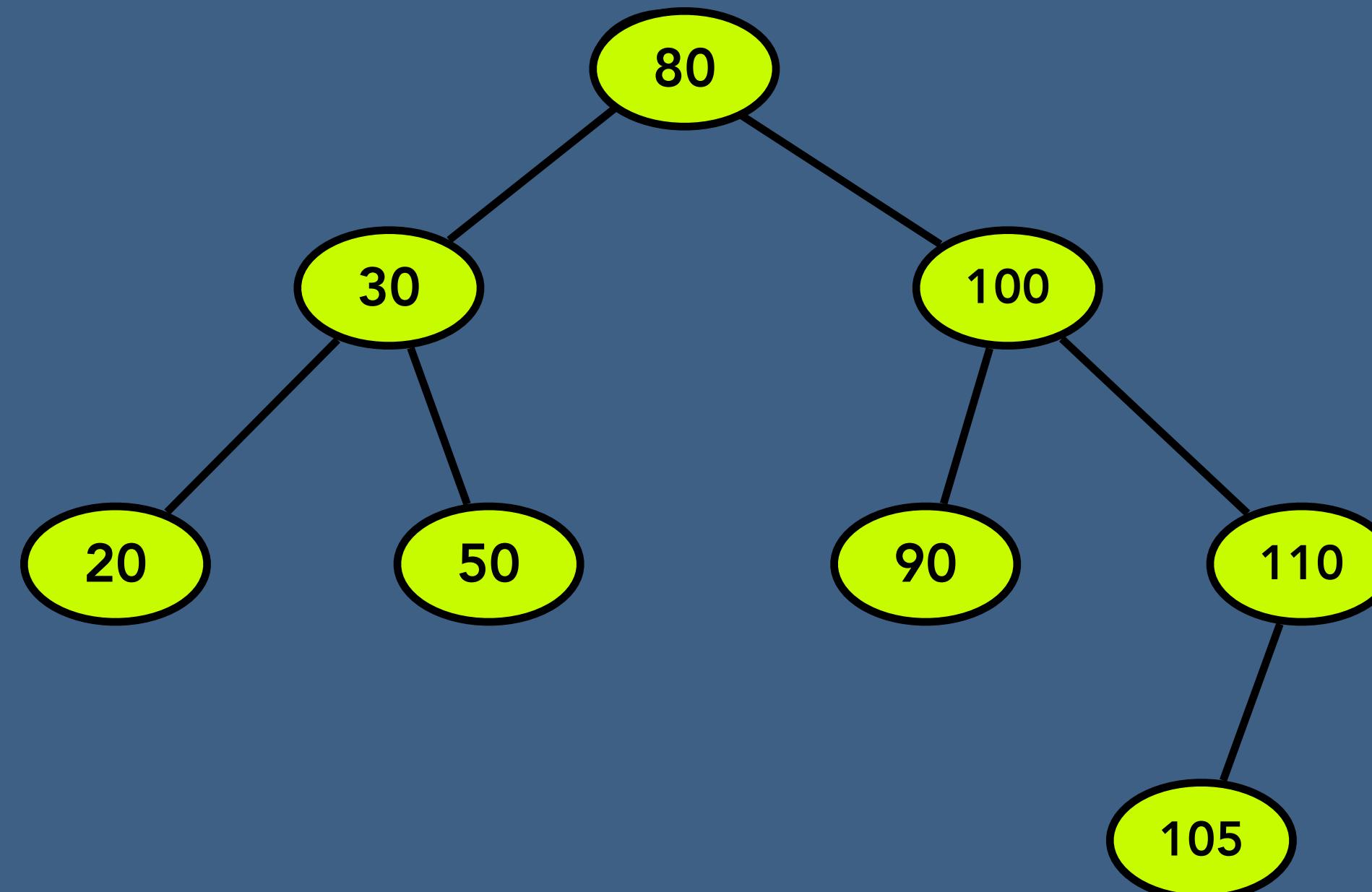


- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR) (highlighted)
- Right Left Condition (RL)

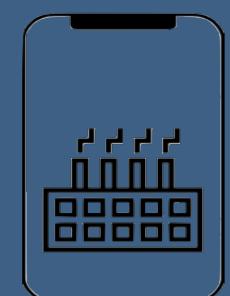


# AVL Tree - Delete a Node (all together)

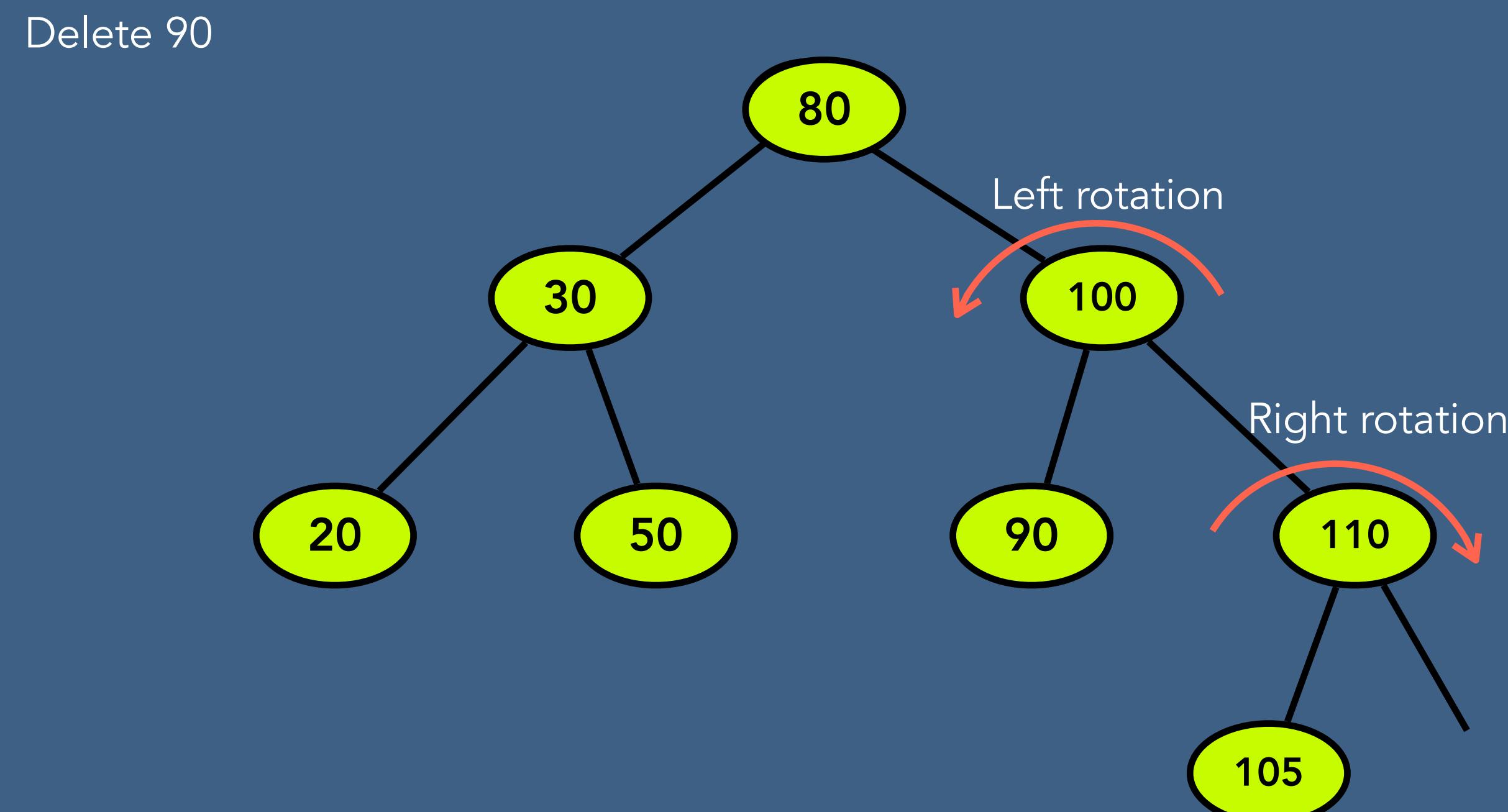
Insert 105



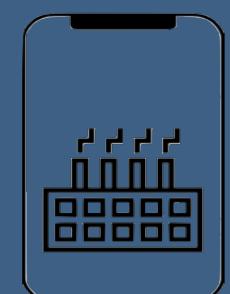
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



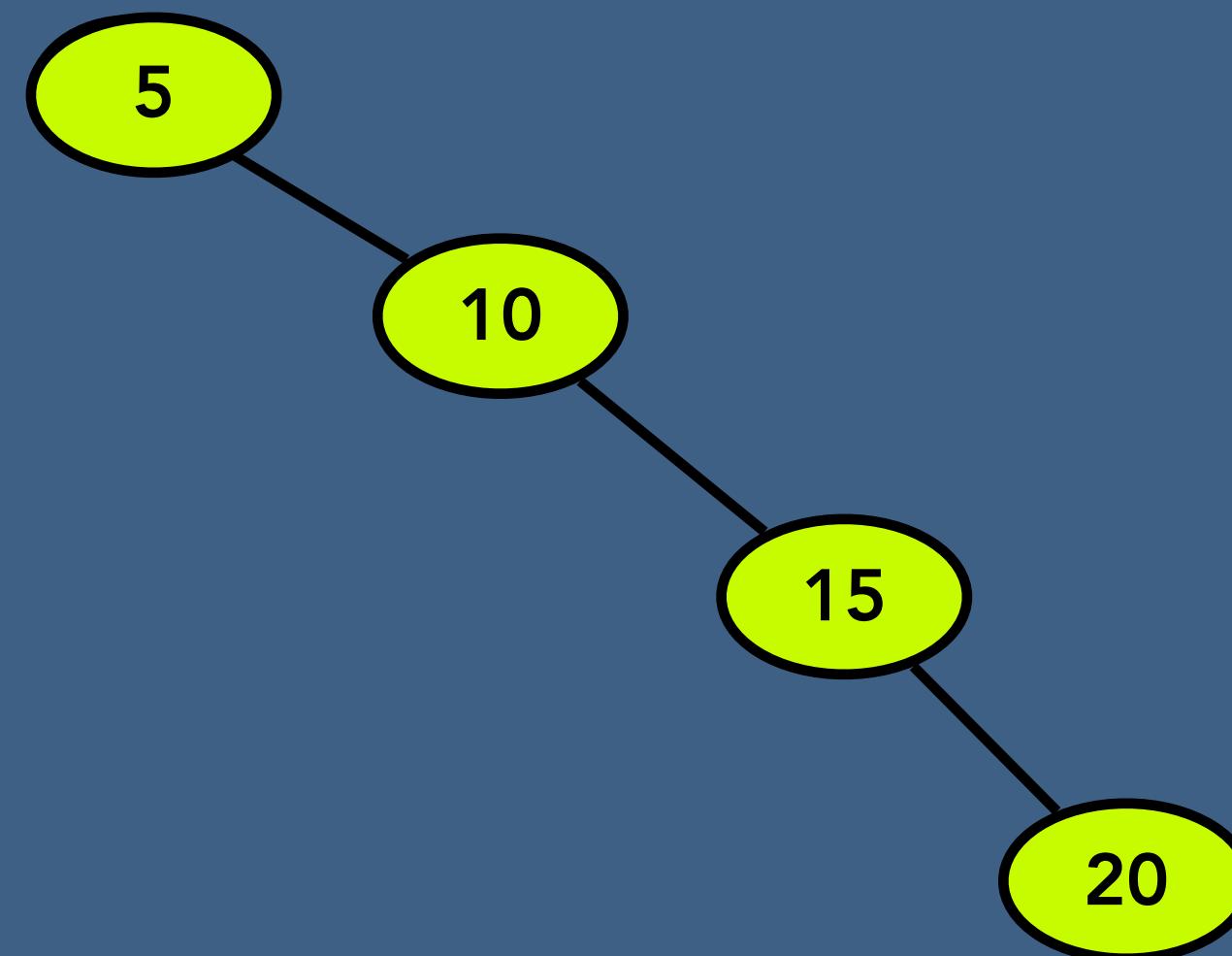
# AVL Tree - Delete a Node (all together)



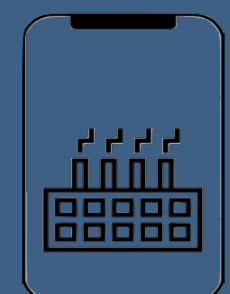
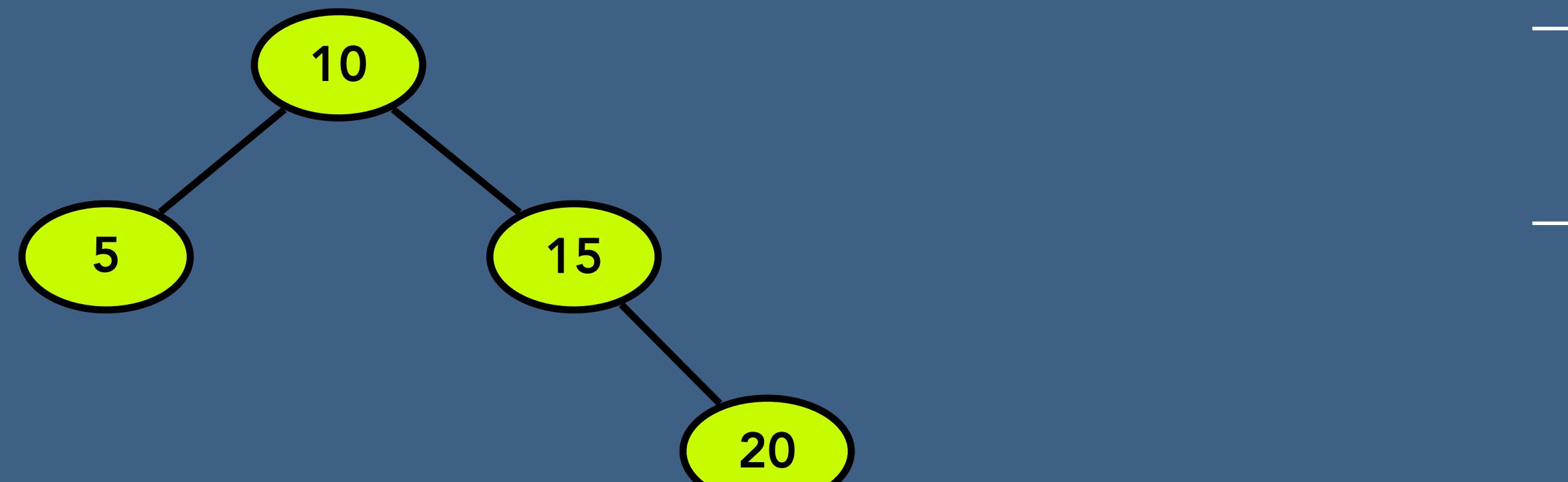
- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



# AVL Tree - Delete a Node (all together)

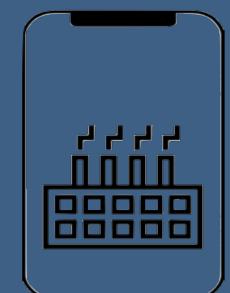
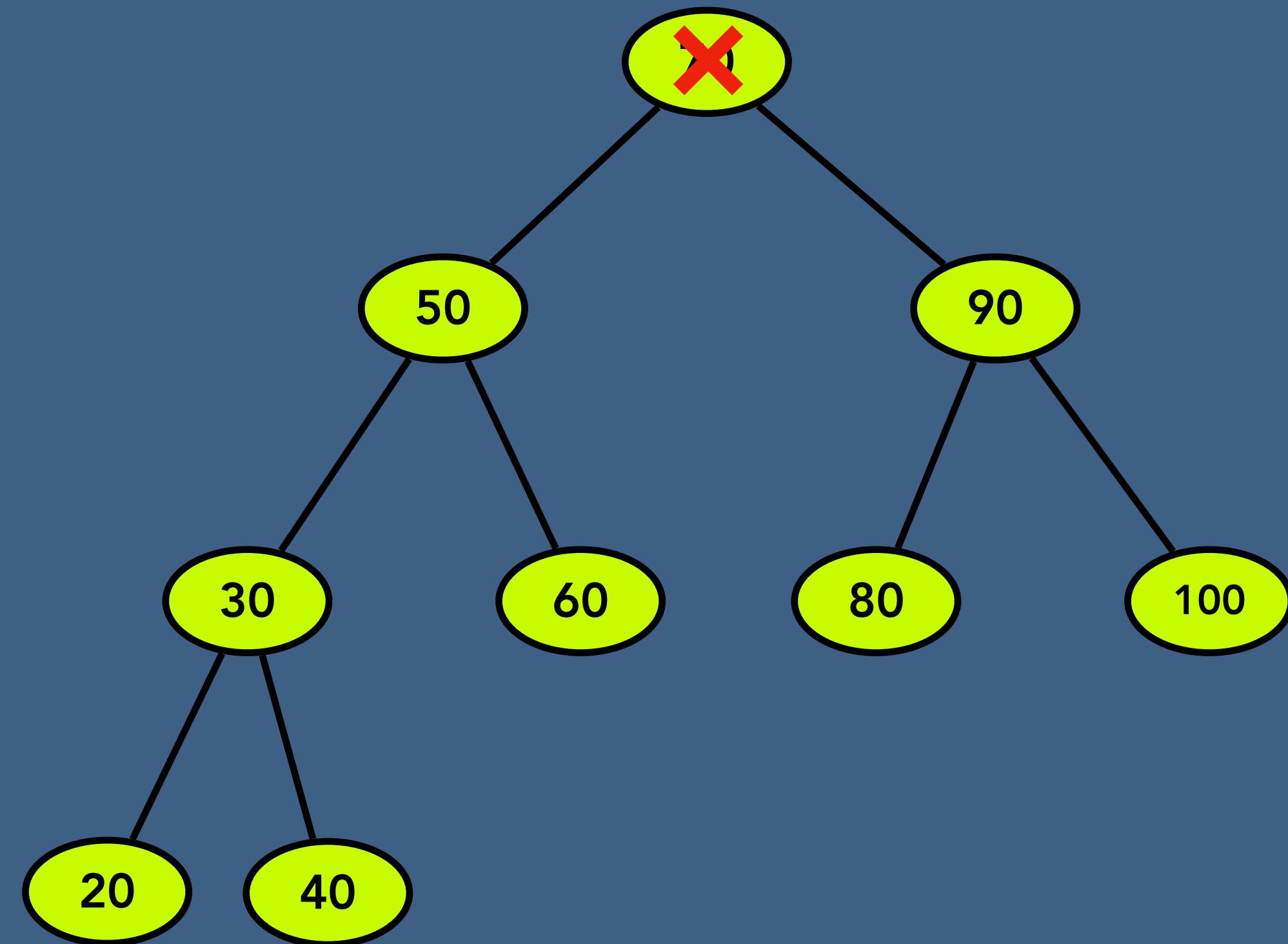


- Left Left Condition (LL)
- Left Right Condition (LR)
- Right Right Condition (RR)
- Right Left Condition (RL)



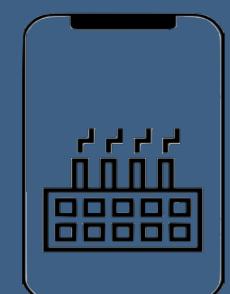
# AVL Tree - Delete

```
rootNode = Null
```



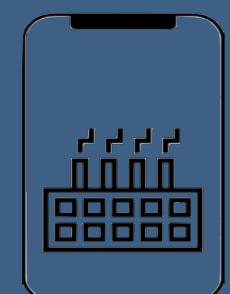
# Time and Space Complexity of AVL

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

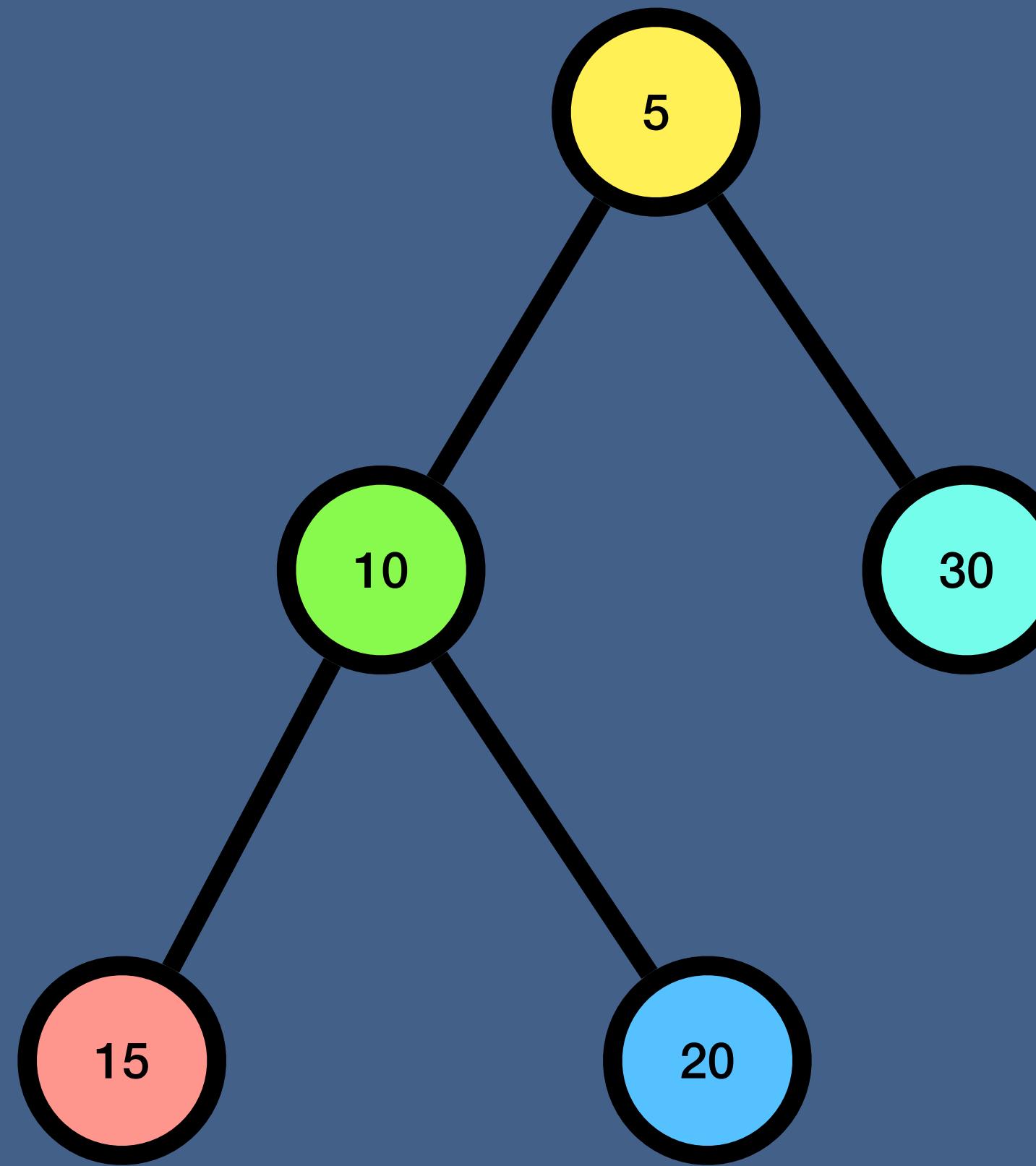


# Binary Search Tree vs AVL Tree

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



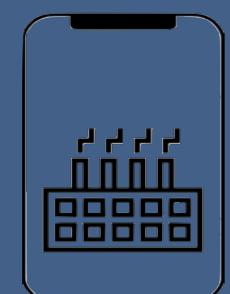
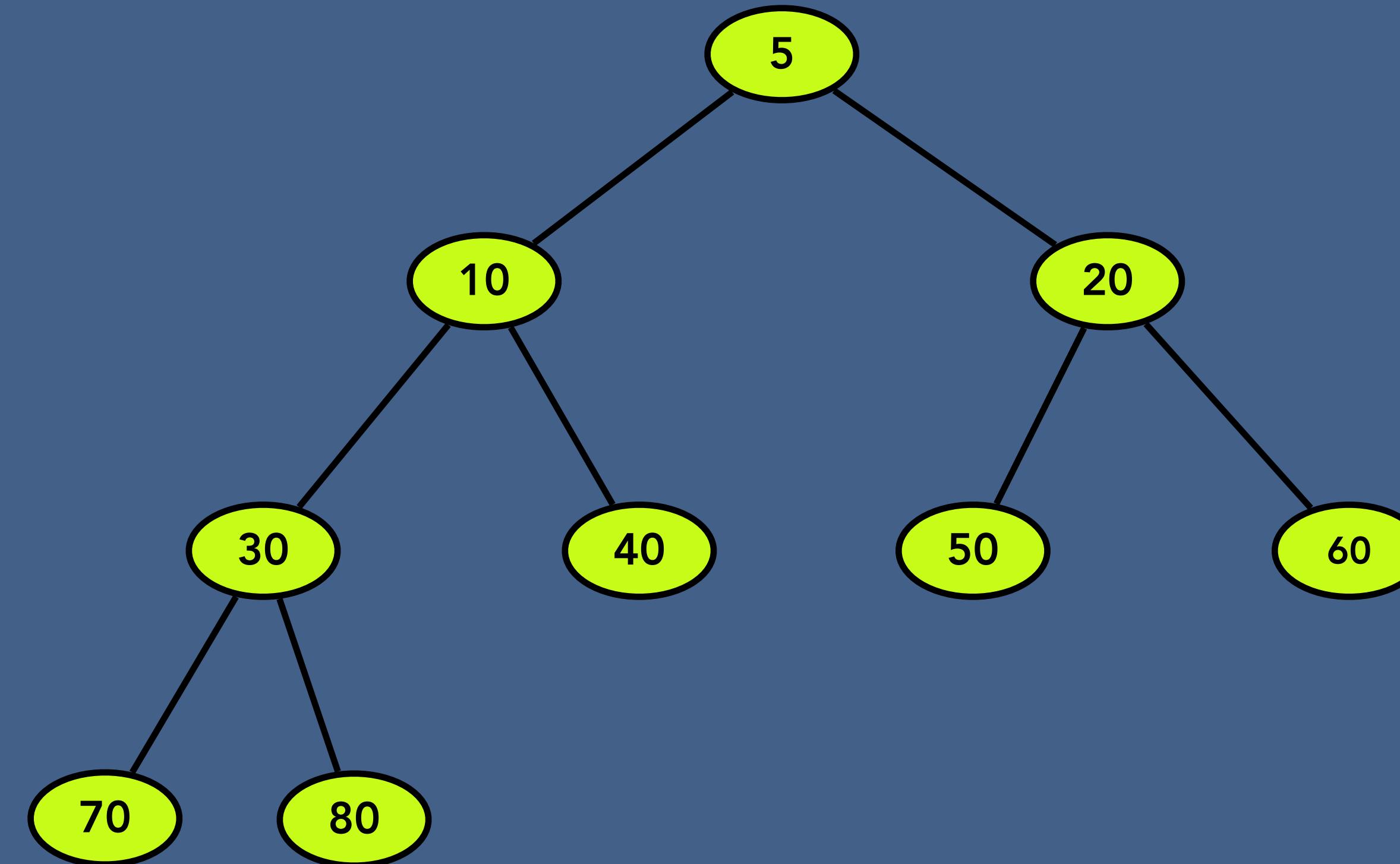
# Binary Heap



# 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.

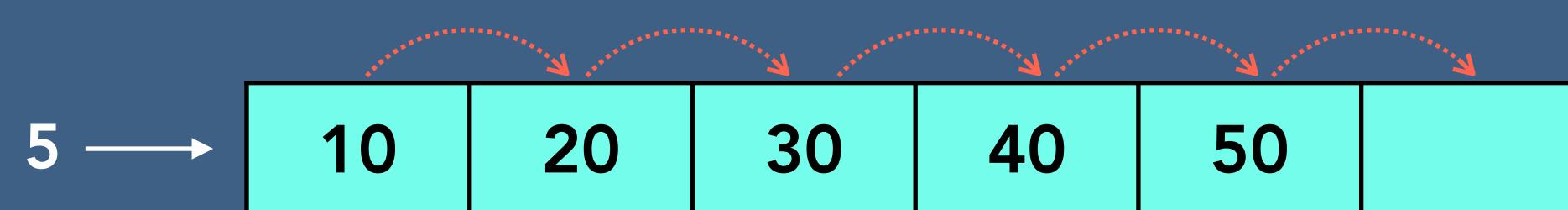


# Why we need a Binary Heap?

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

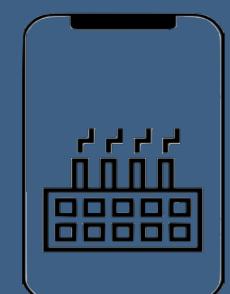
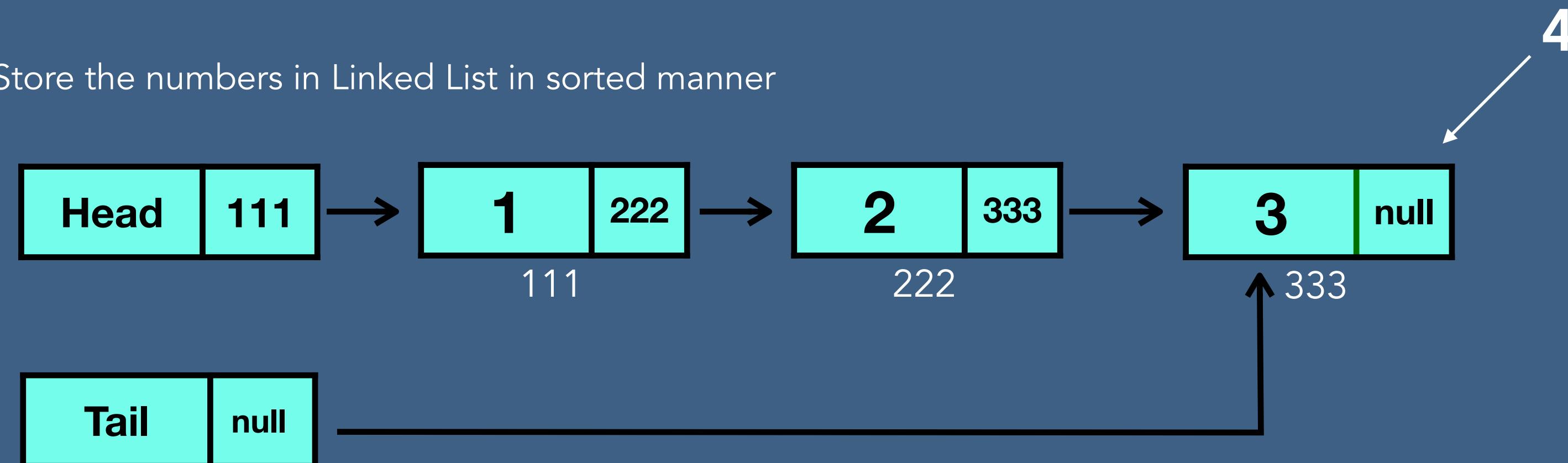
- Store the numbers in sorted Array



Find minimum:  $O(1)$

Insertion:  $O(n)$

- Store the numbers in Linked List in sorted manner

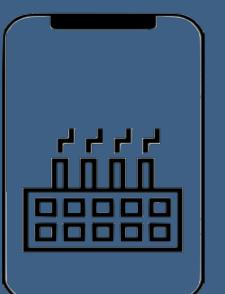


# Why we need a Binary Heap?

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

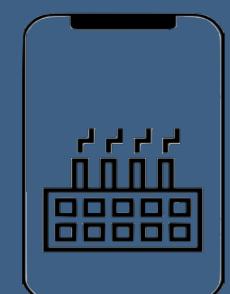
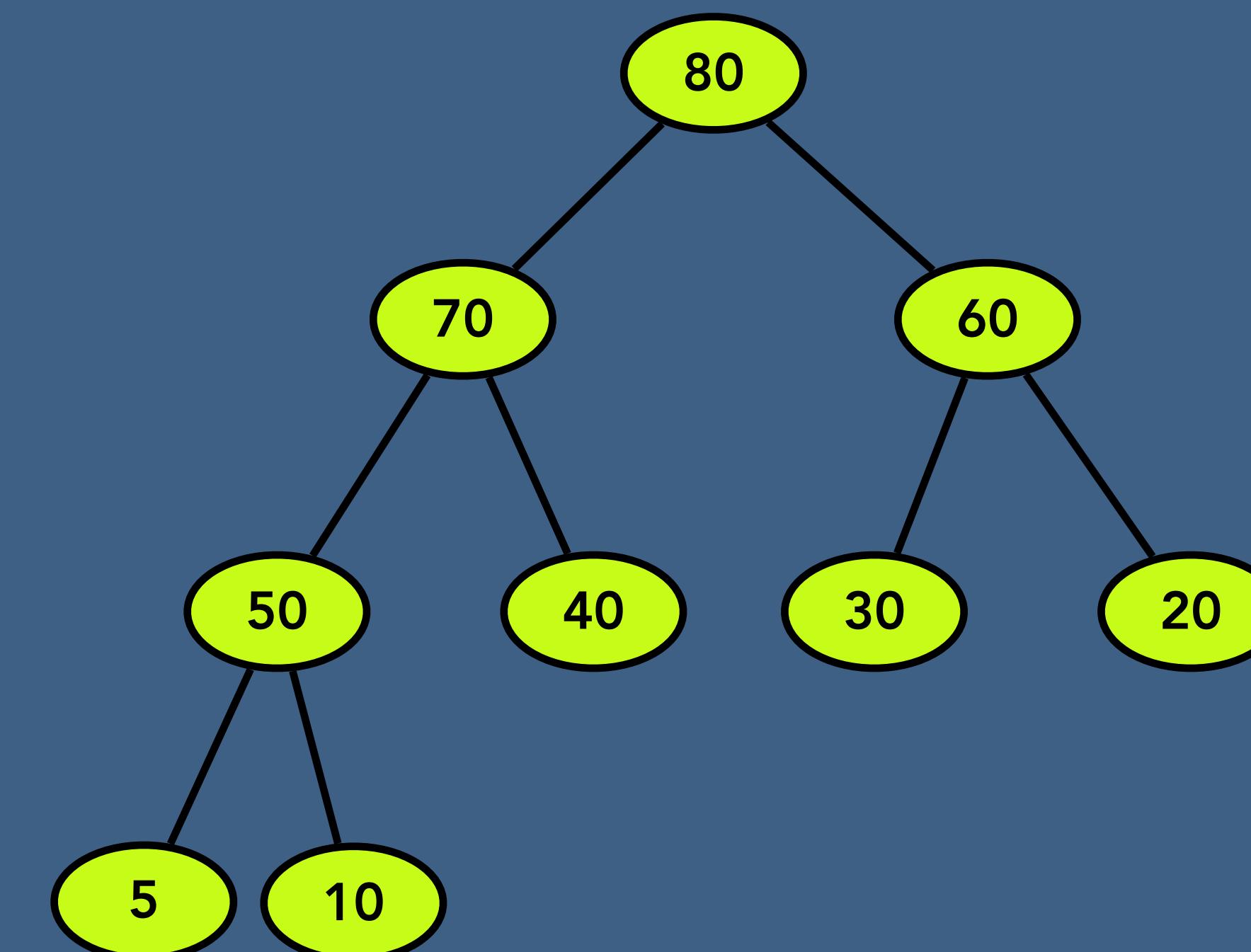
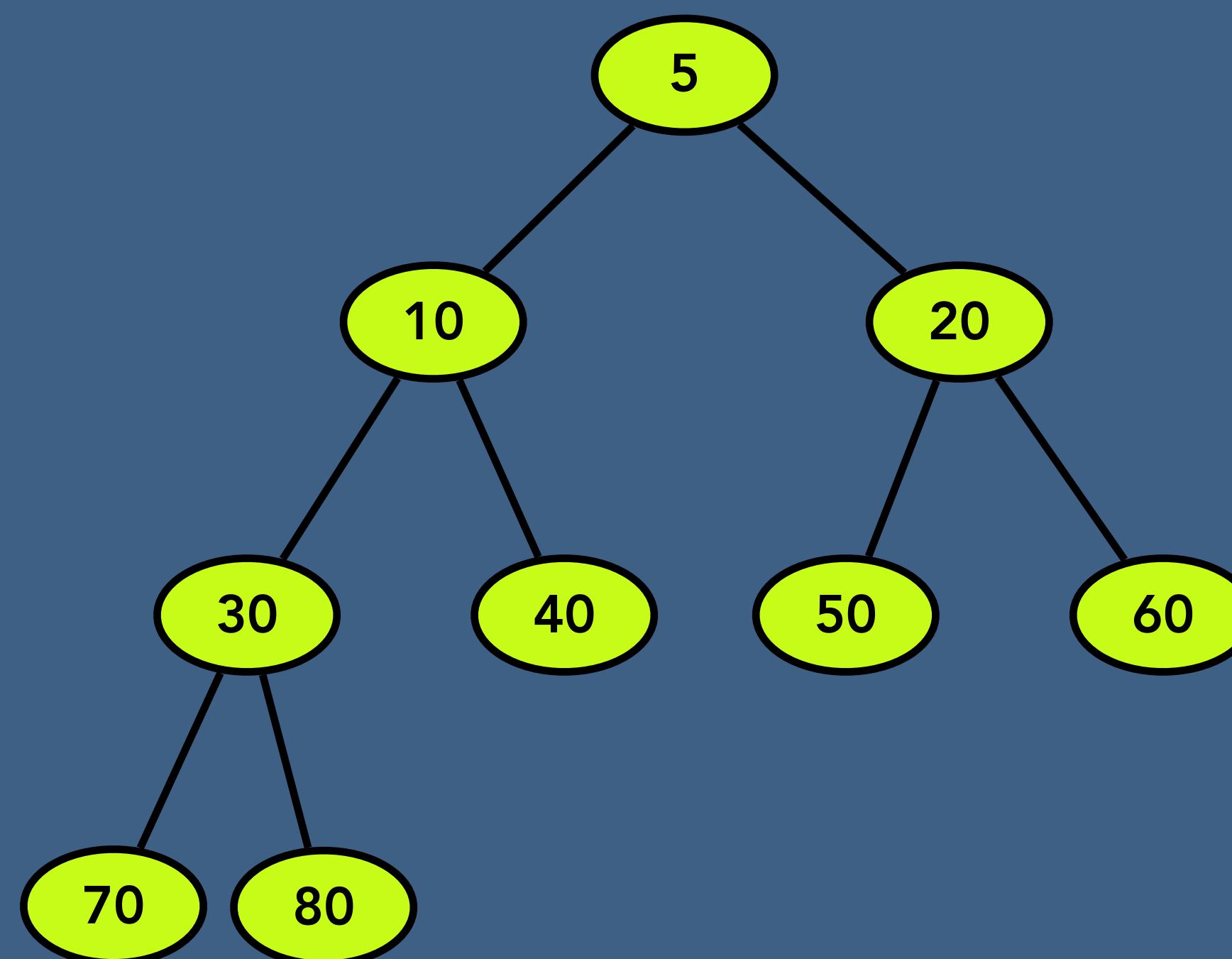
- Prim's Algorithm
- Heap Sort
- Priority Queue



# Types of Binary Heap

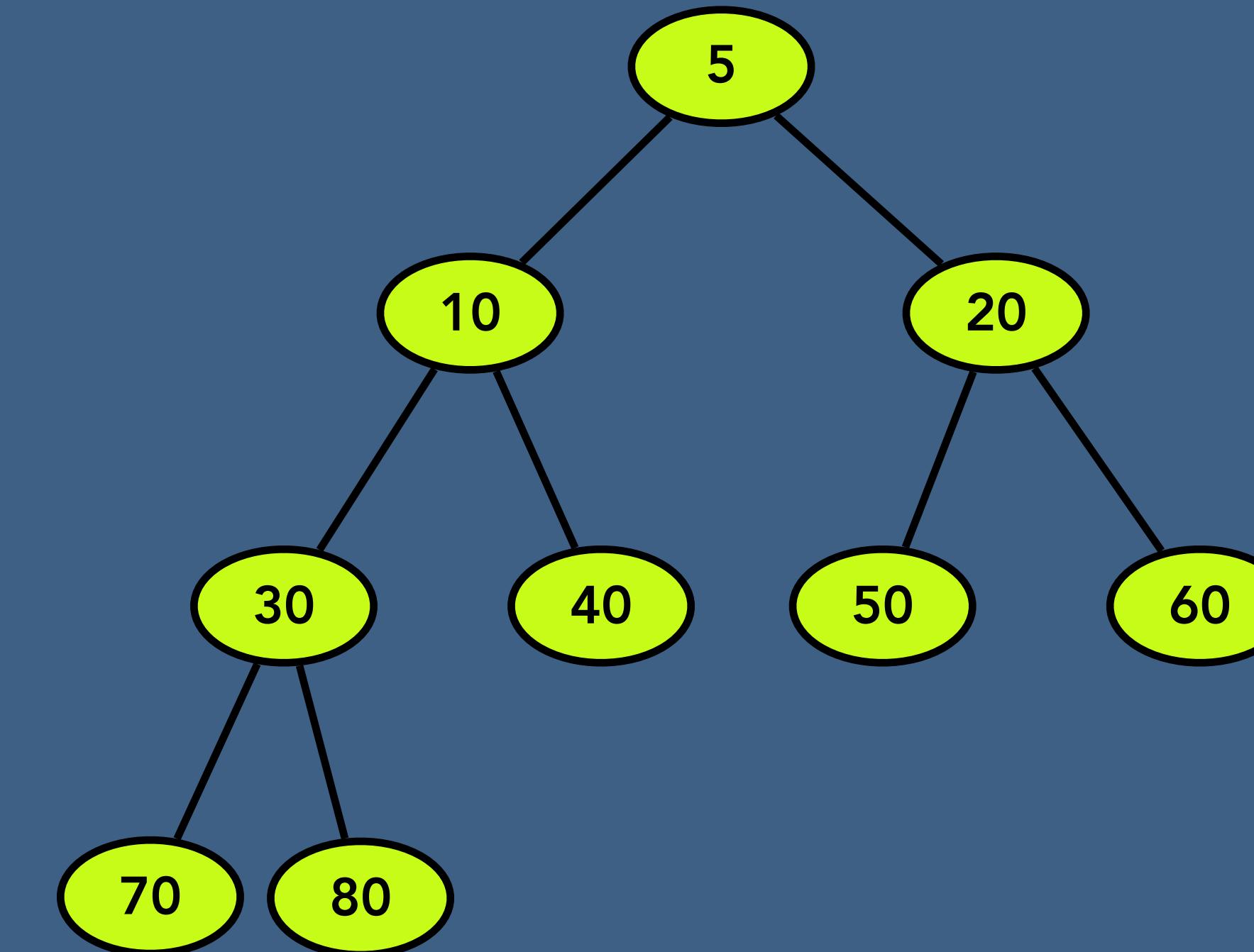
**Min heap** - the value of each node is less than or equal to the value of both its children.

**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.



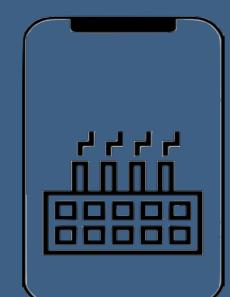
# Common Operations on Binary Heap

- 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

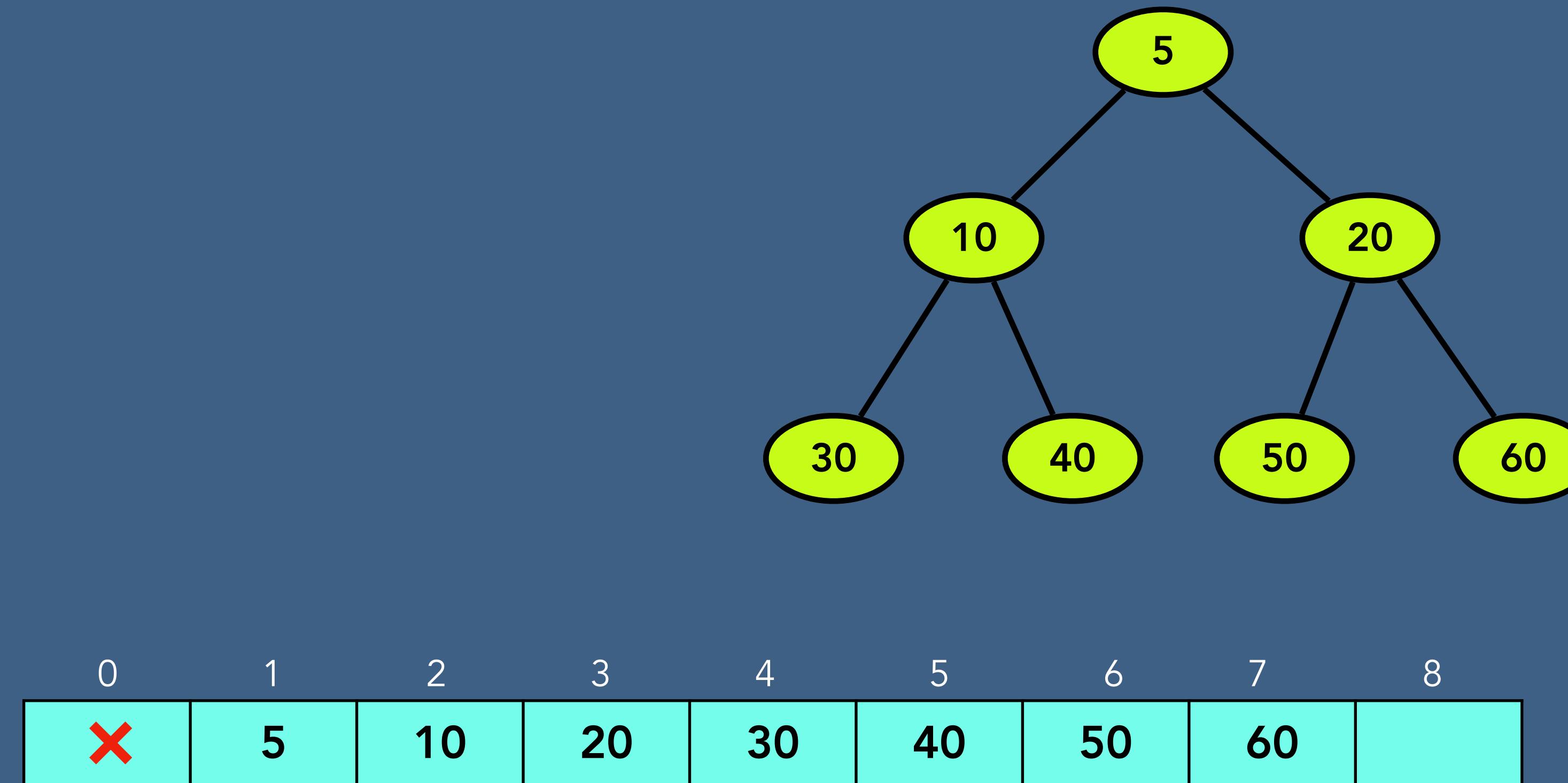


## Implementation Options

- Array Implementation
- Reference /pointer Implementation

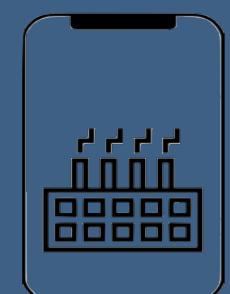


# Common Operations on Binary Heap



Left child = `cell[2x]`

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

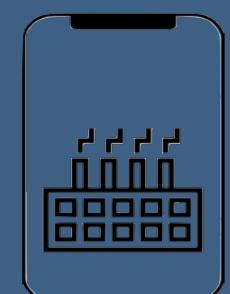
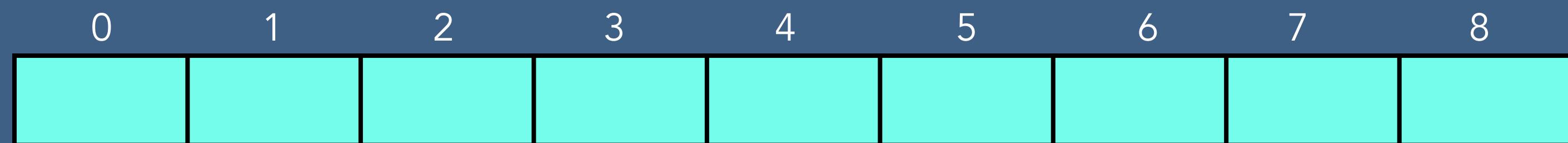


# Common Operations on Binary Heap

- Creation of Binary Heap

Initialize Array

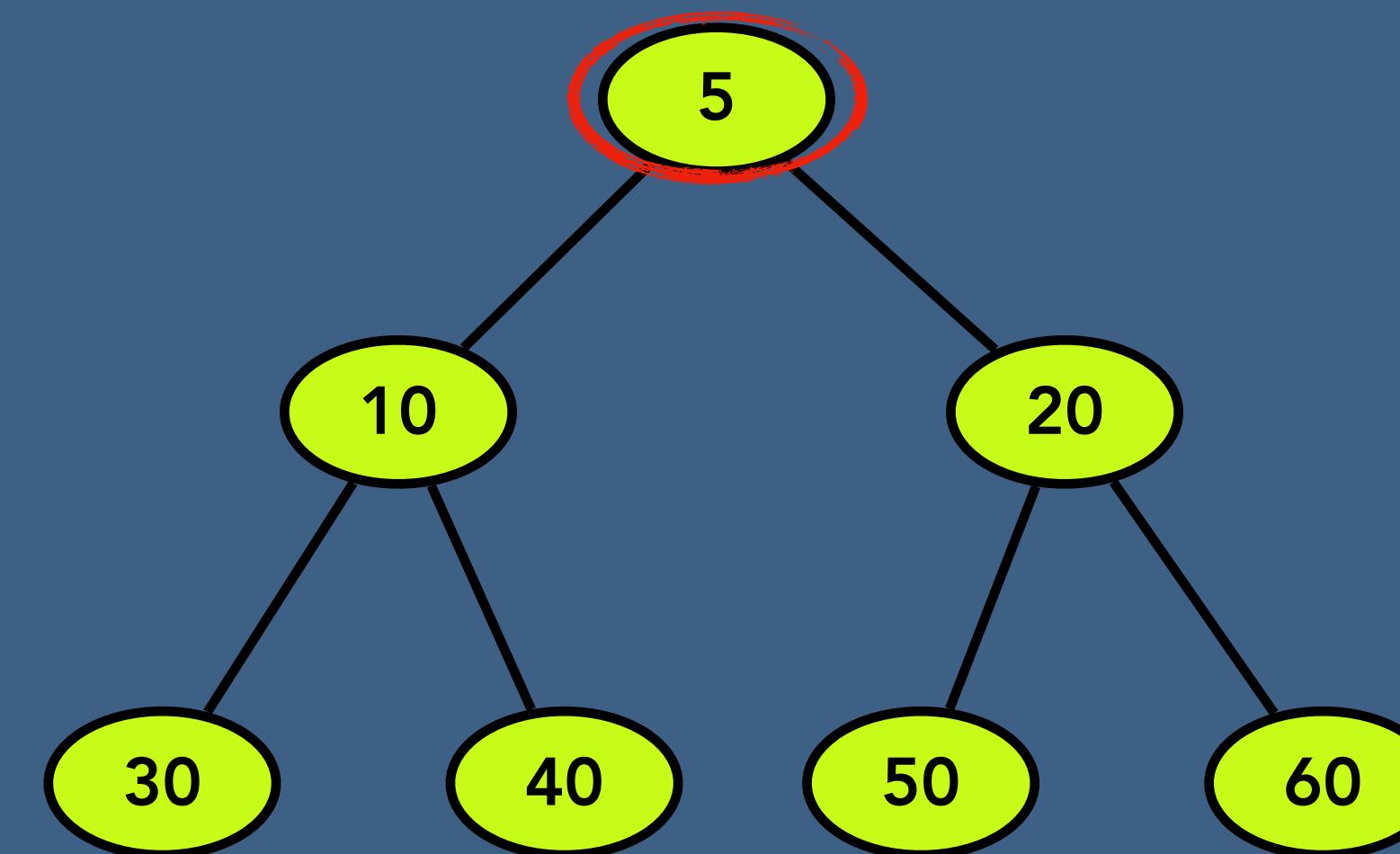
set size of Binary Heap to 0



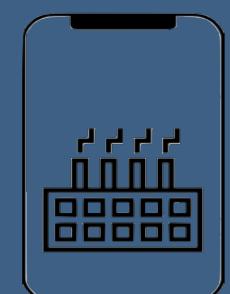
# Common Operations on Binary Heap

- Peek of Binary Heap

Return Array[1]



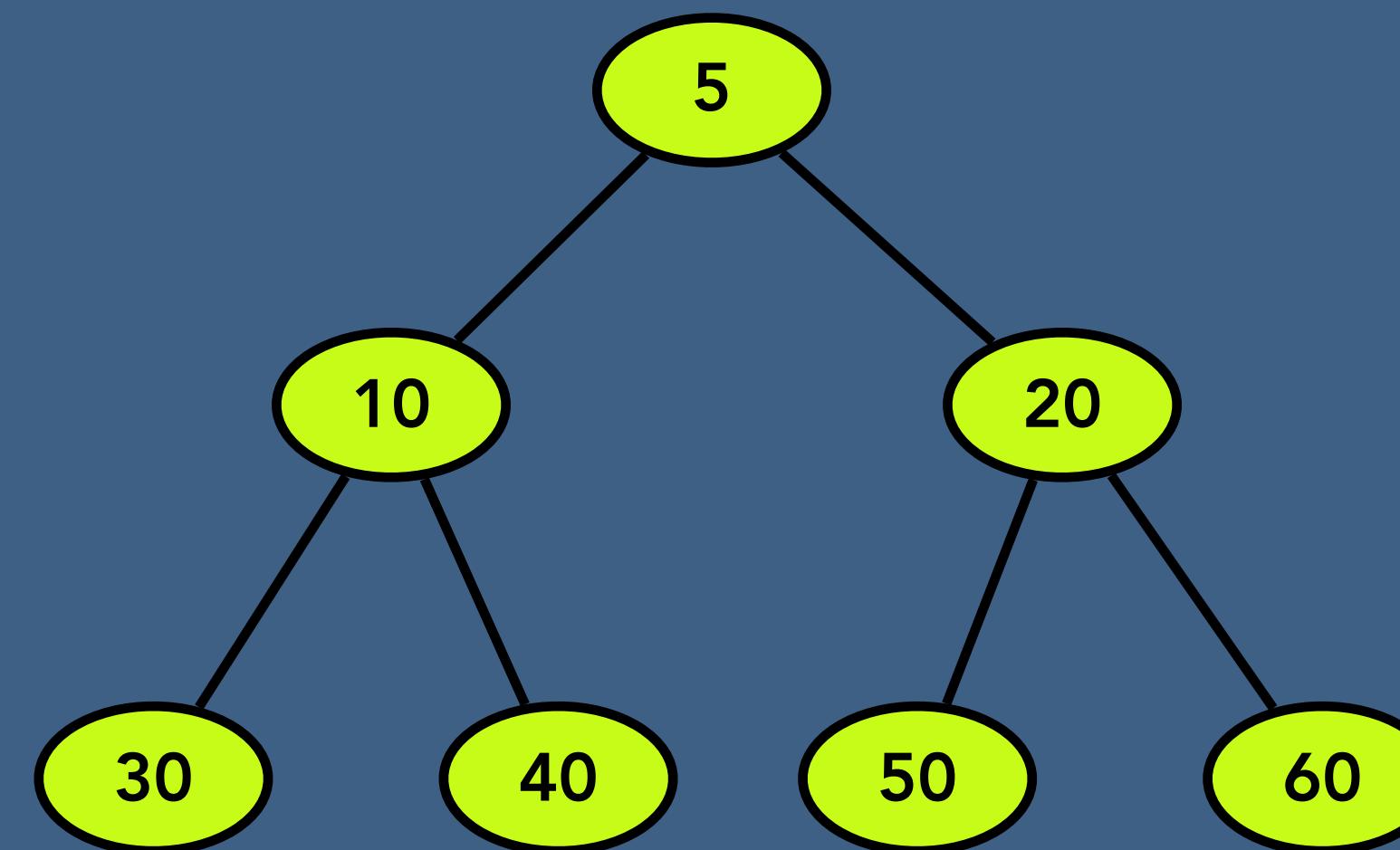
0	1	2	3	4	5	6	7	8
✗	5	10	20	30	40	50	60	



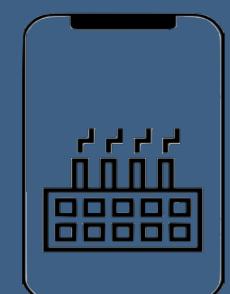
# Common Operations on Binary Heap

- Size Binary Heap

Return number of filled cells

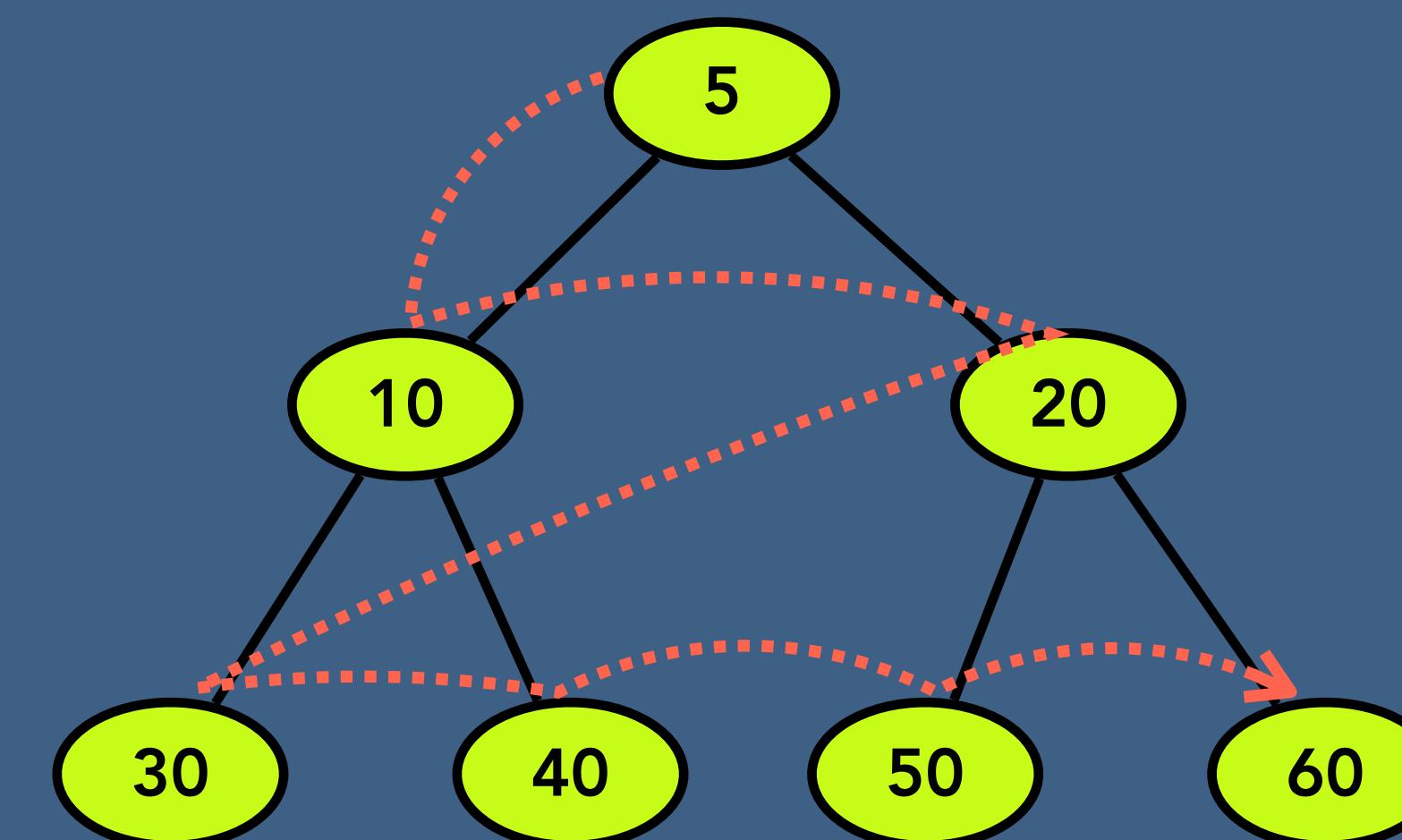


0	1	2	3	4	5	6	7	8
✗	5	10	20	30	40	50	60	

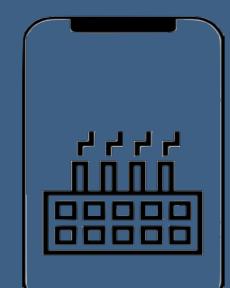


# Common Operations on Binary Heap

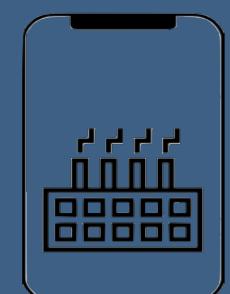
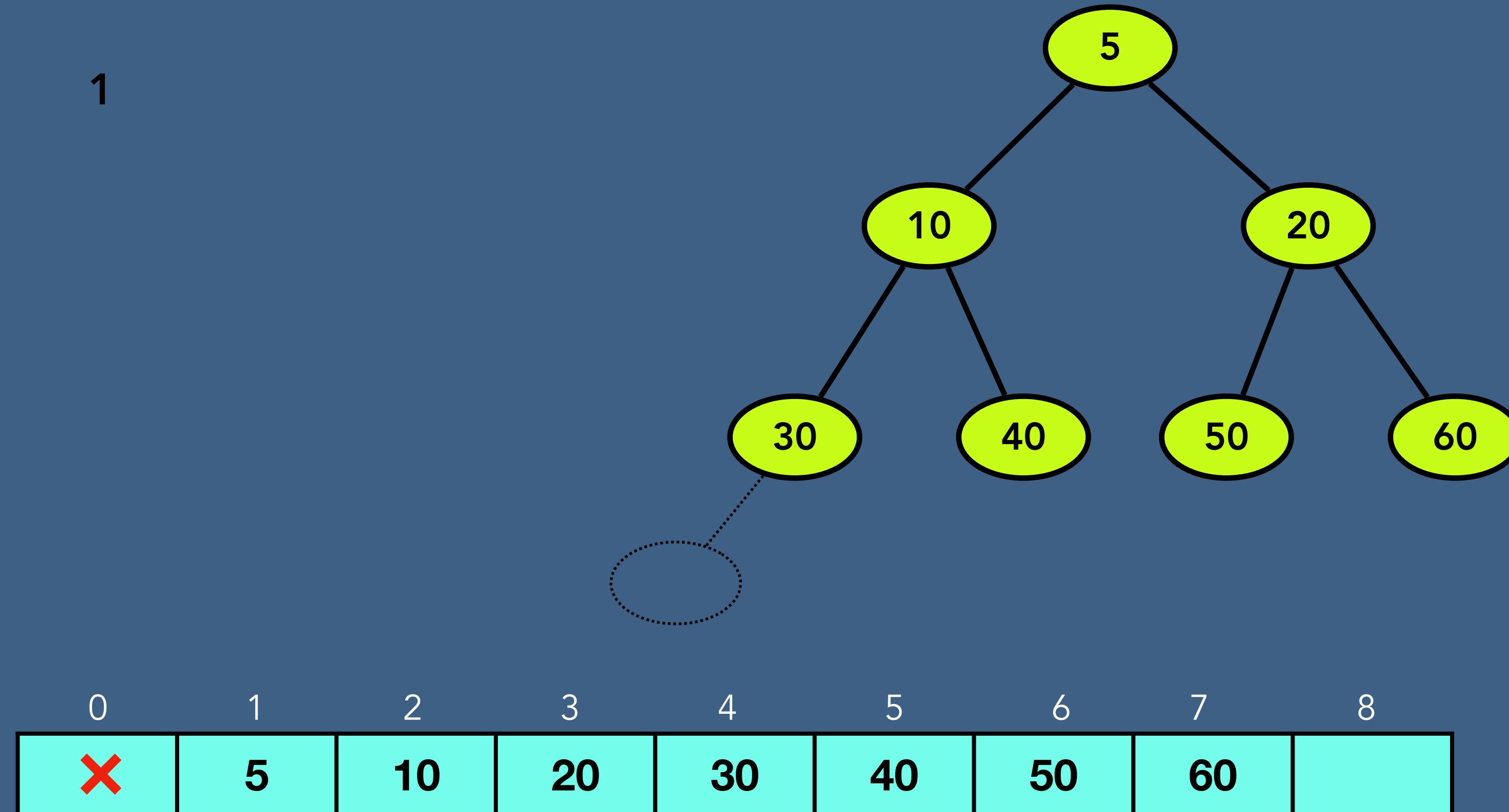
- Level Order Traversal



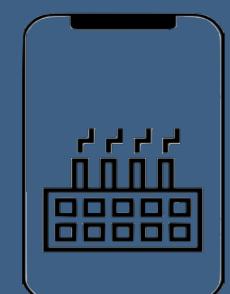
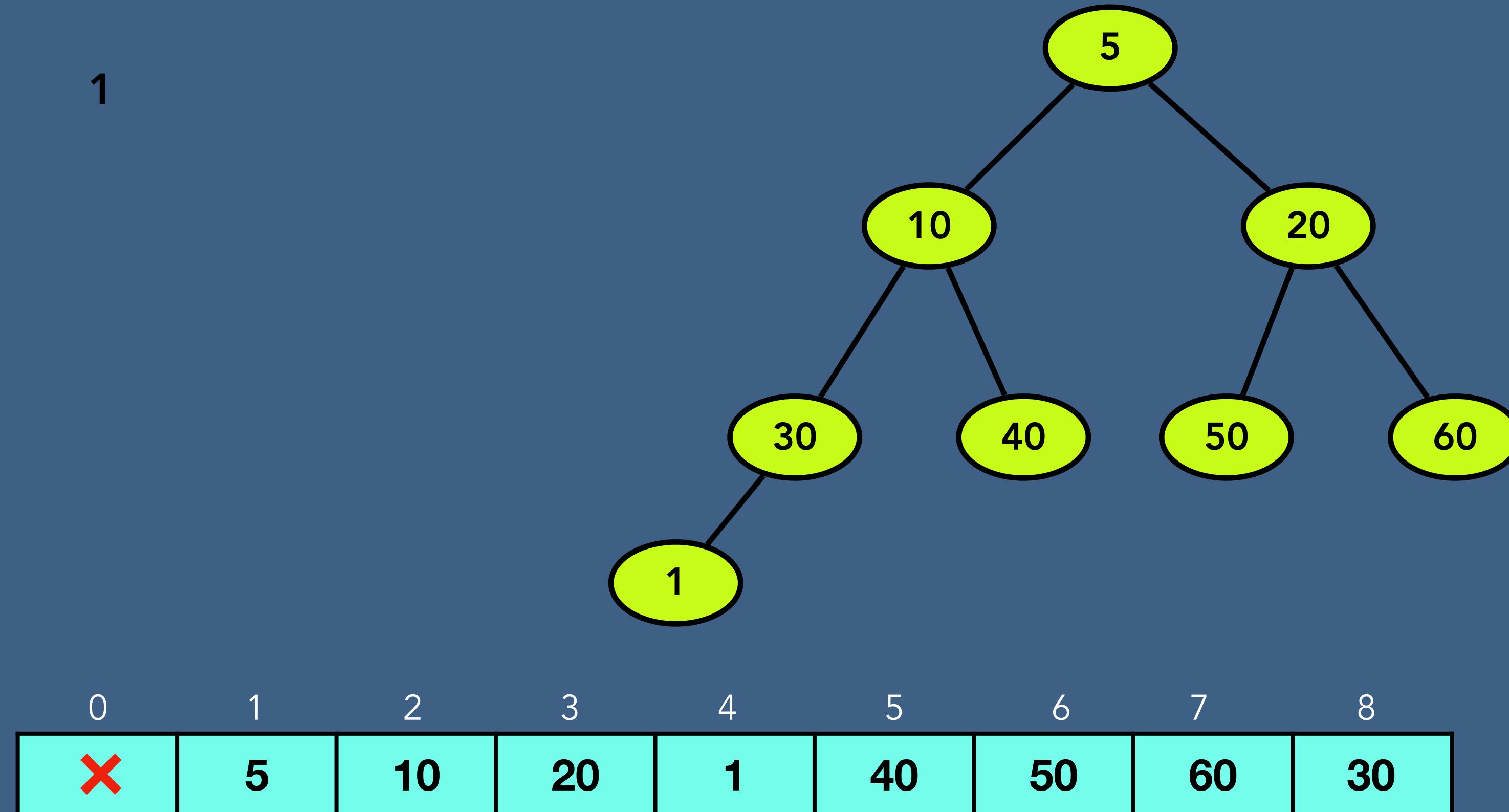
0	1	2	3	4	5	6	7	8
✗	5	10	20	30	40	50	60	



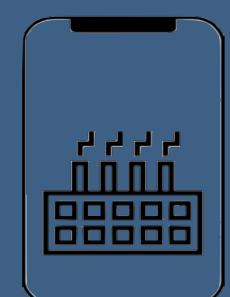
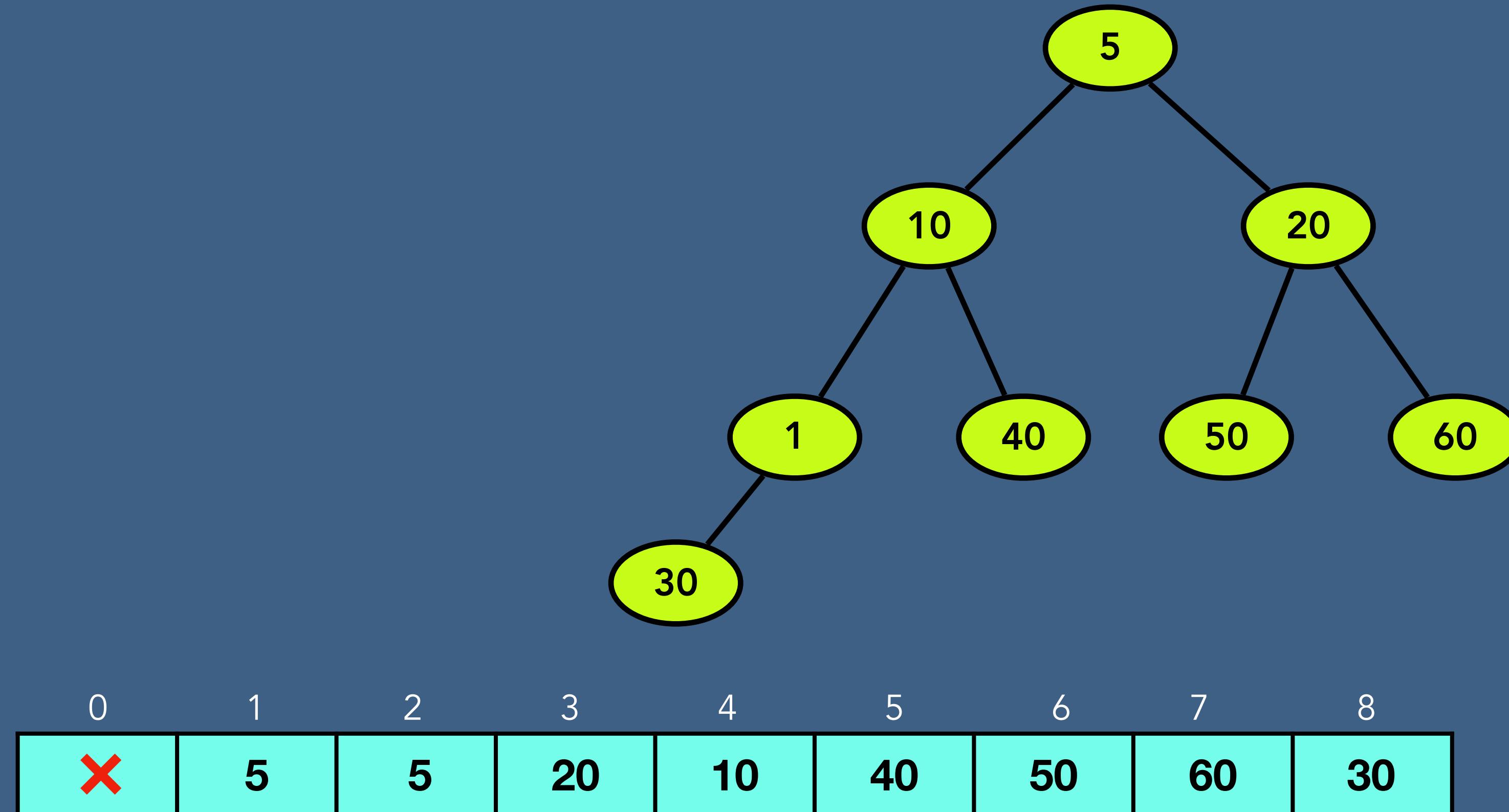
# Binary Heap - Insert a Node



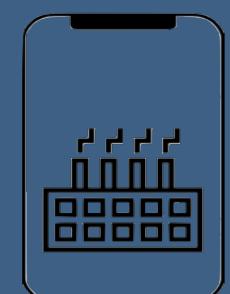
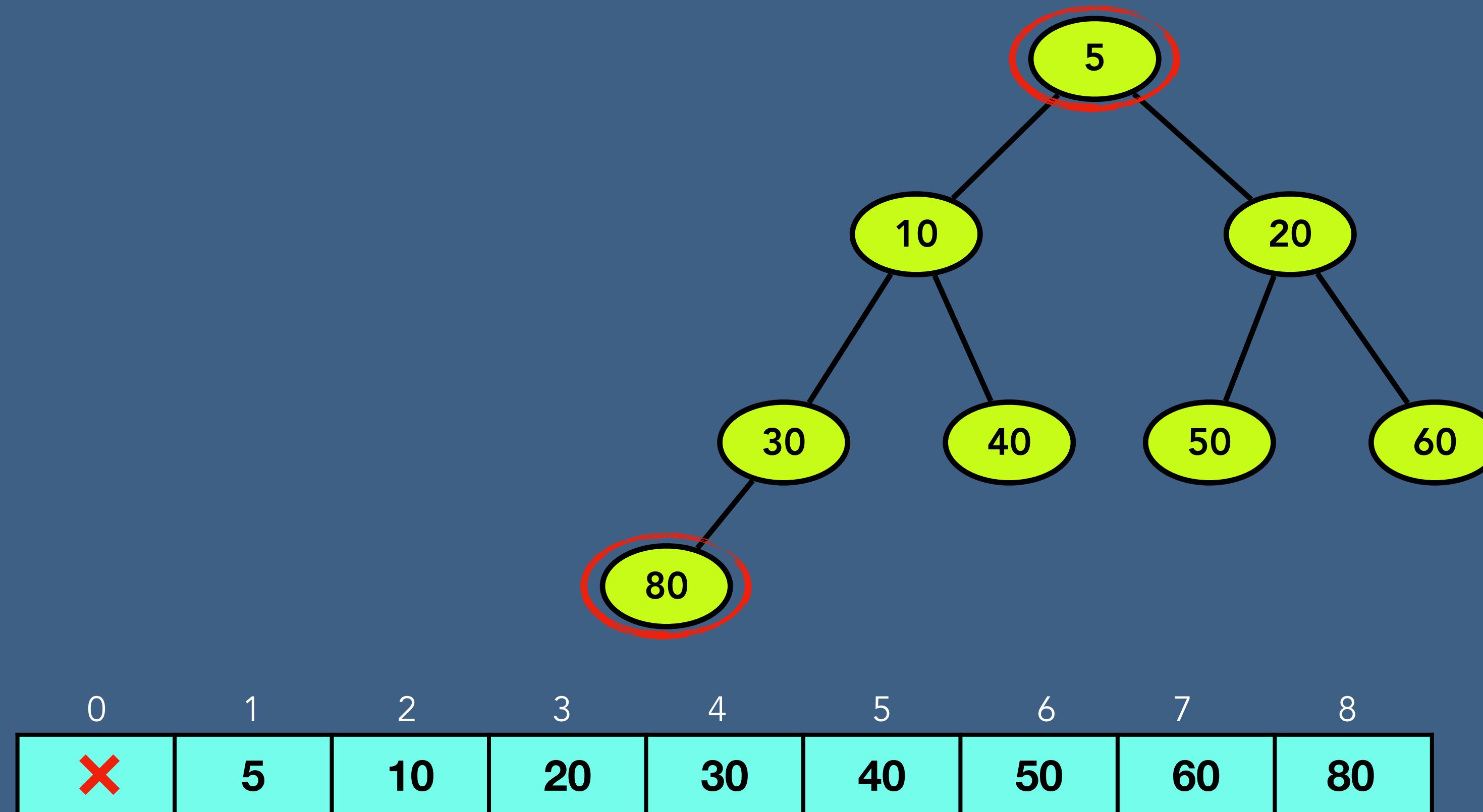
# Binary Heap - Insert a Node



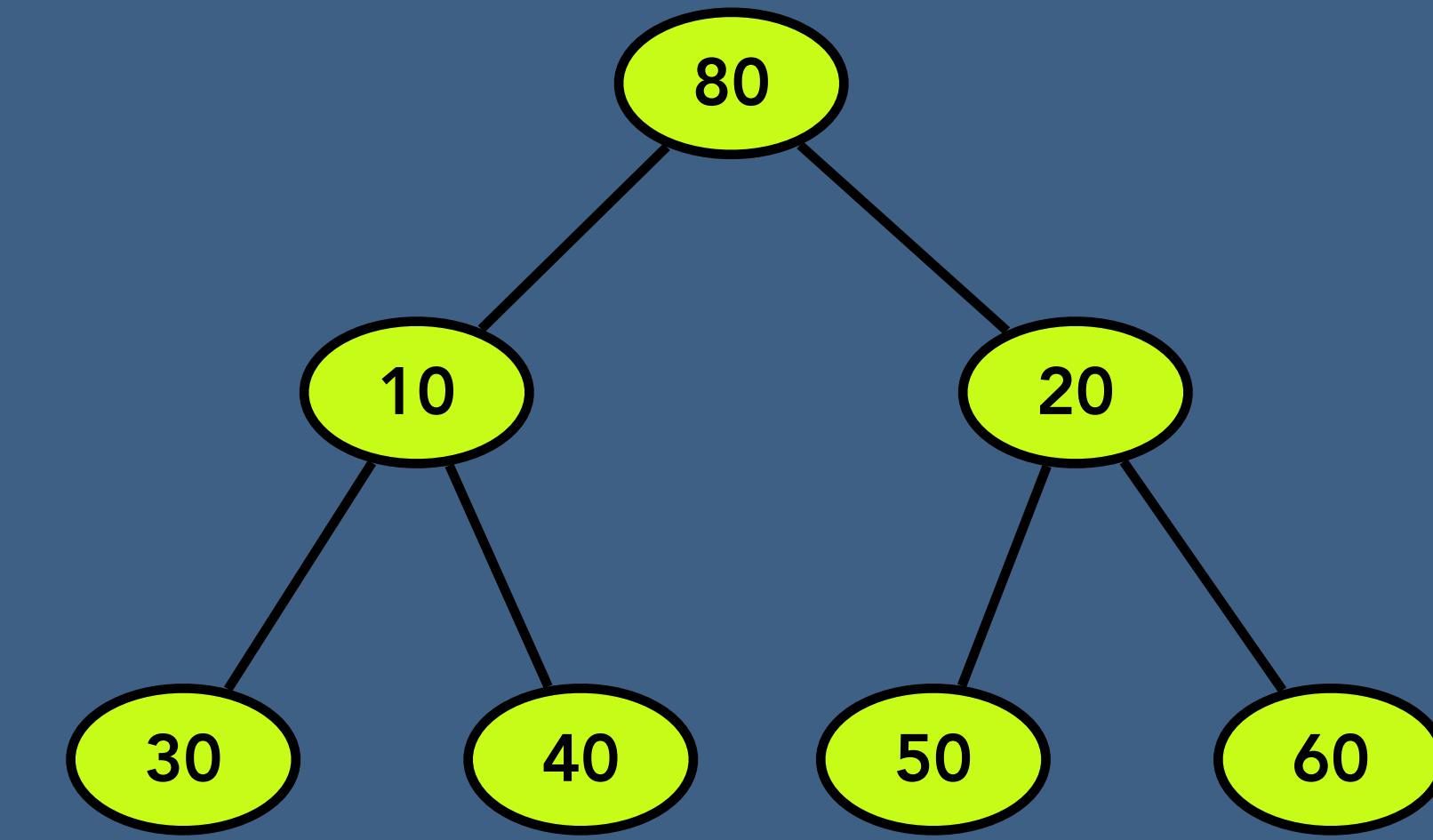
# Binary Heap - Insert a Node



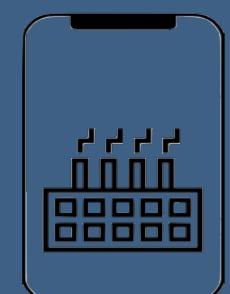
# Binary Heap - Extract a Node



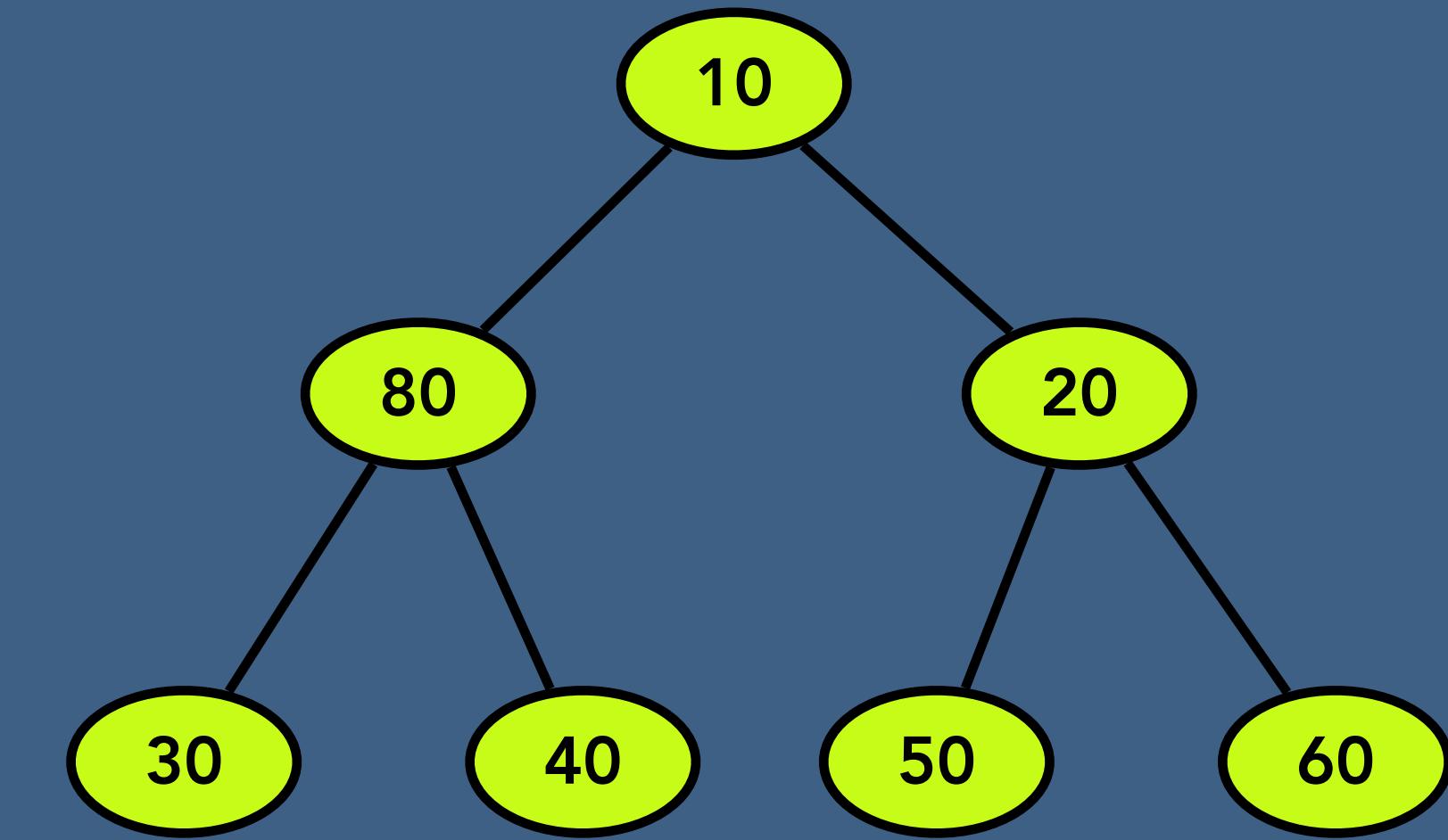
# Binary Heap - Extract a Node



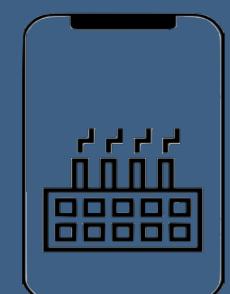
0	1	2	3	4	5	6	7	8
✖	80	10	20	30	40	50	60	



# Binary Heap - Extract a Node

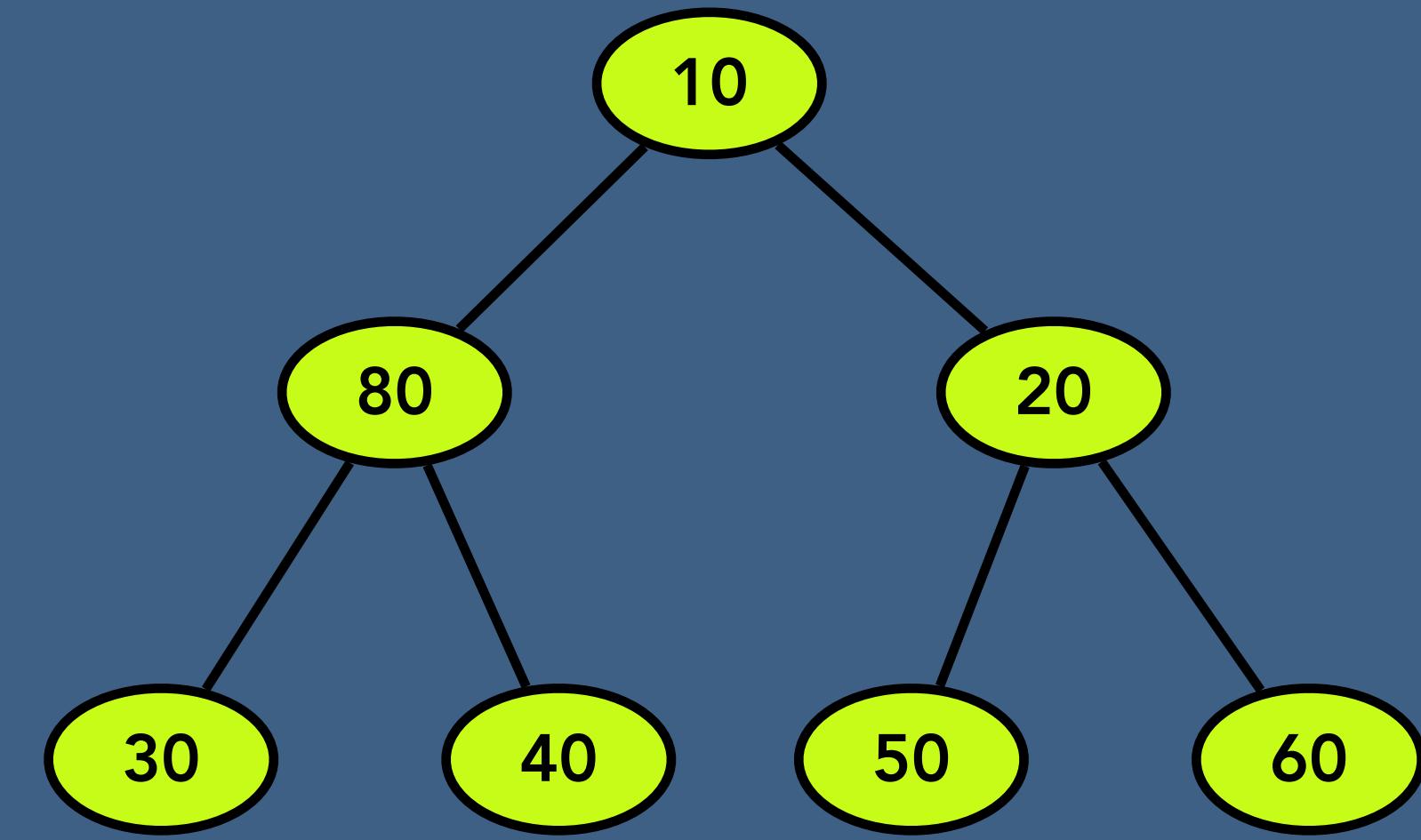


0	1	2	3	4	5	6	7	8
✗	10	80	20	30	40	50	60	

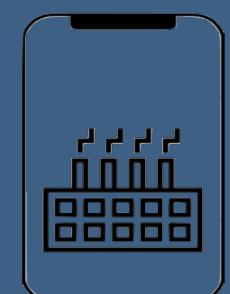


# Binary Heap - Delete

array = Null

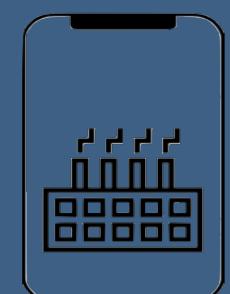


0	1	2	3	4	5	6	7	8
✗	10	80	20	30	40	50	60	

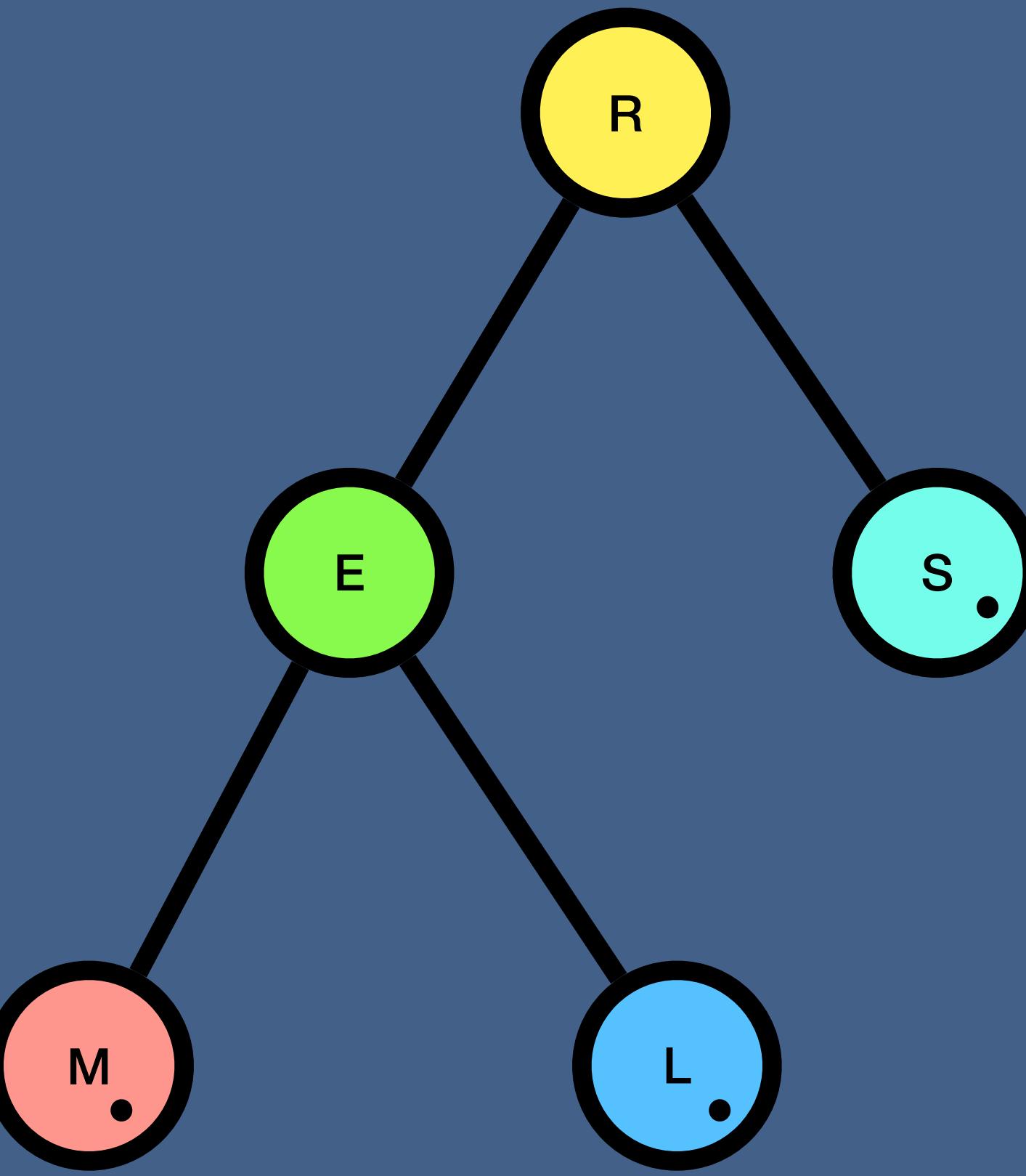


# Time and Space Complexity of Binary Heap

	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	$O(\log N)$	$O(\log N)$
Extract a node from Binary Heap	$O(\log N)$	$O(\log N)$
Delete Entire Binary Heap	$O(1)$	$O(1)$



# Trie



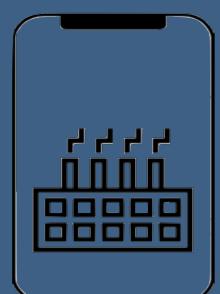
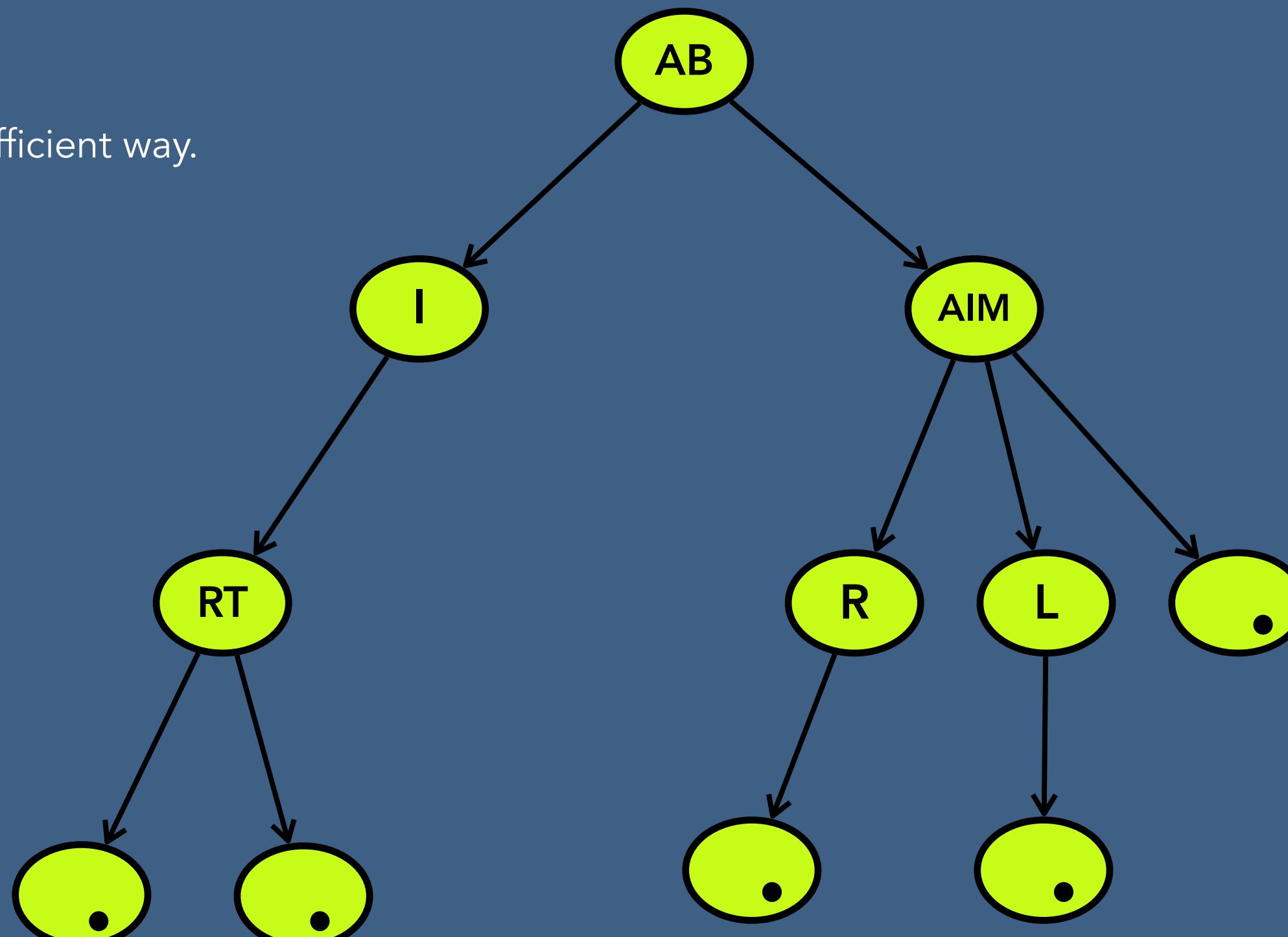
# What is a Trie?

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'

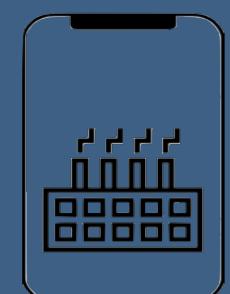
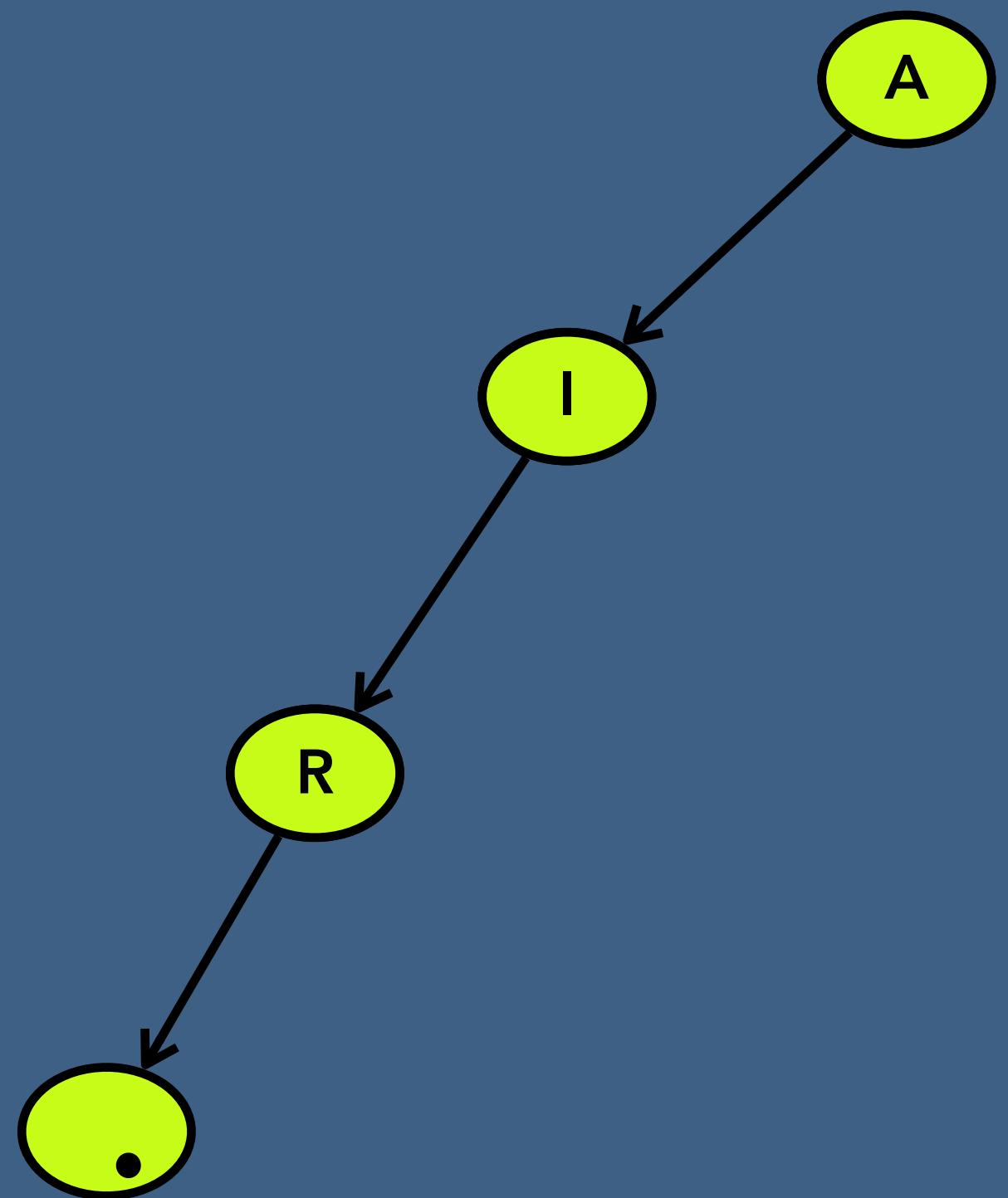
AIR, AIT, BAR, BIL, BM



# What is a Trie?

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

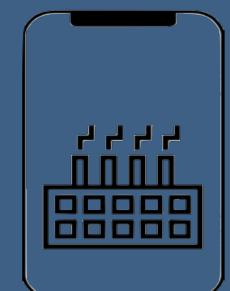
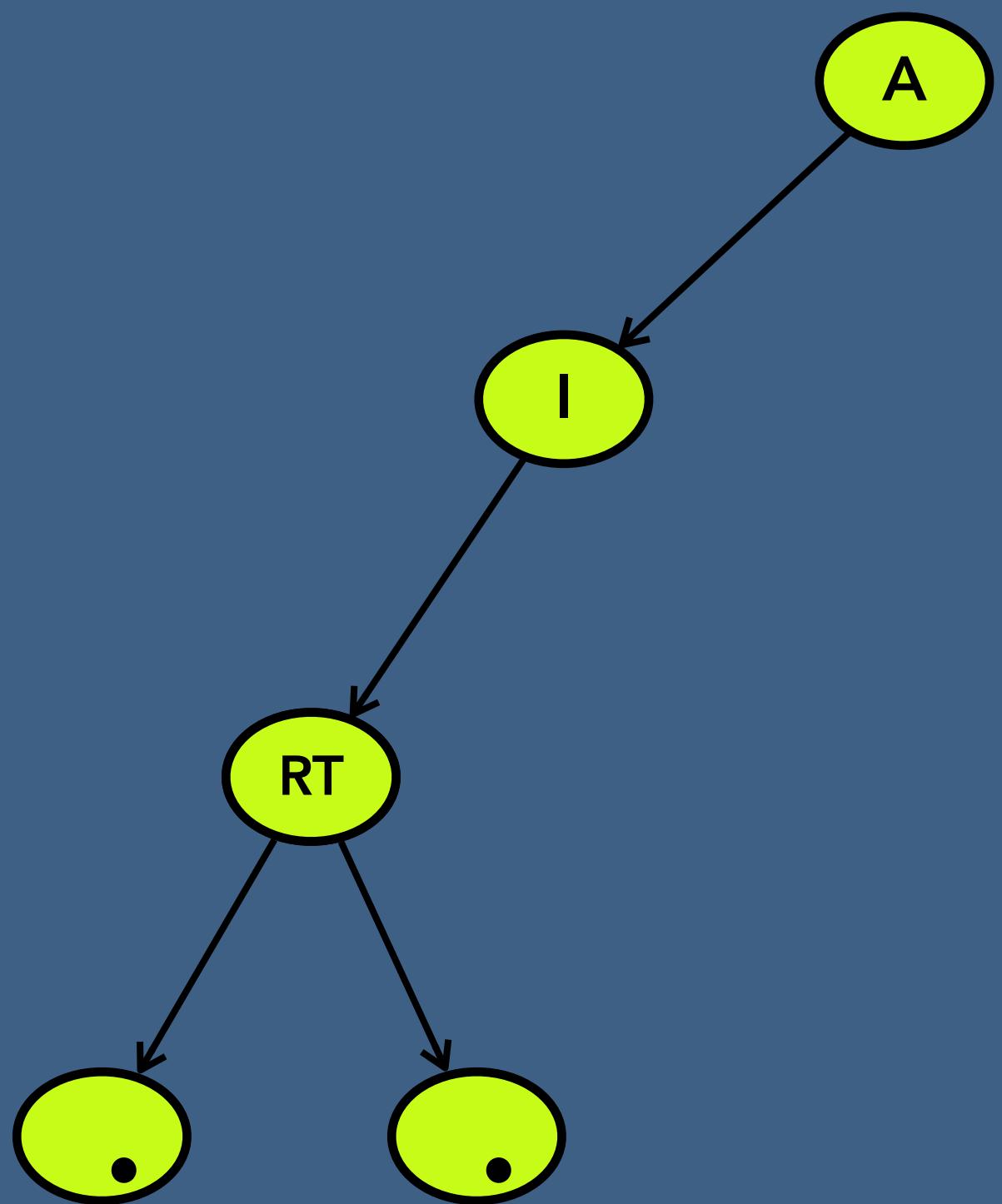
AIR



# What is a Trie?

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

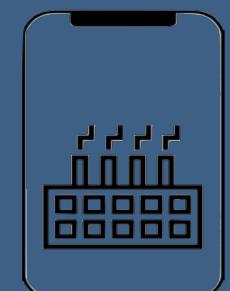
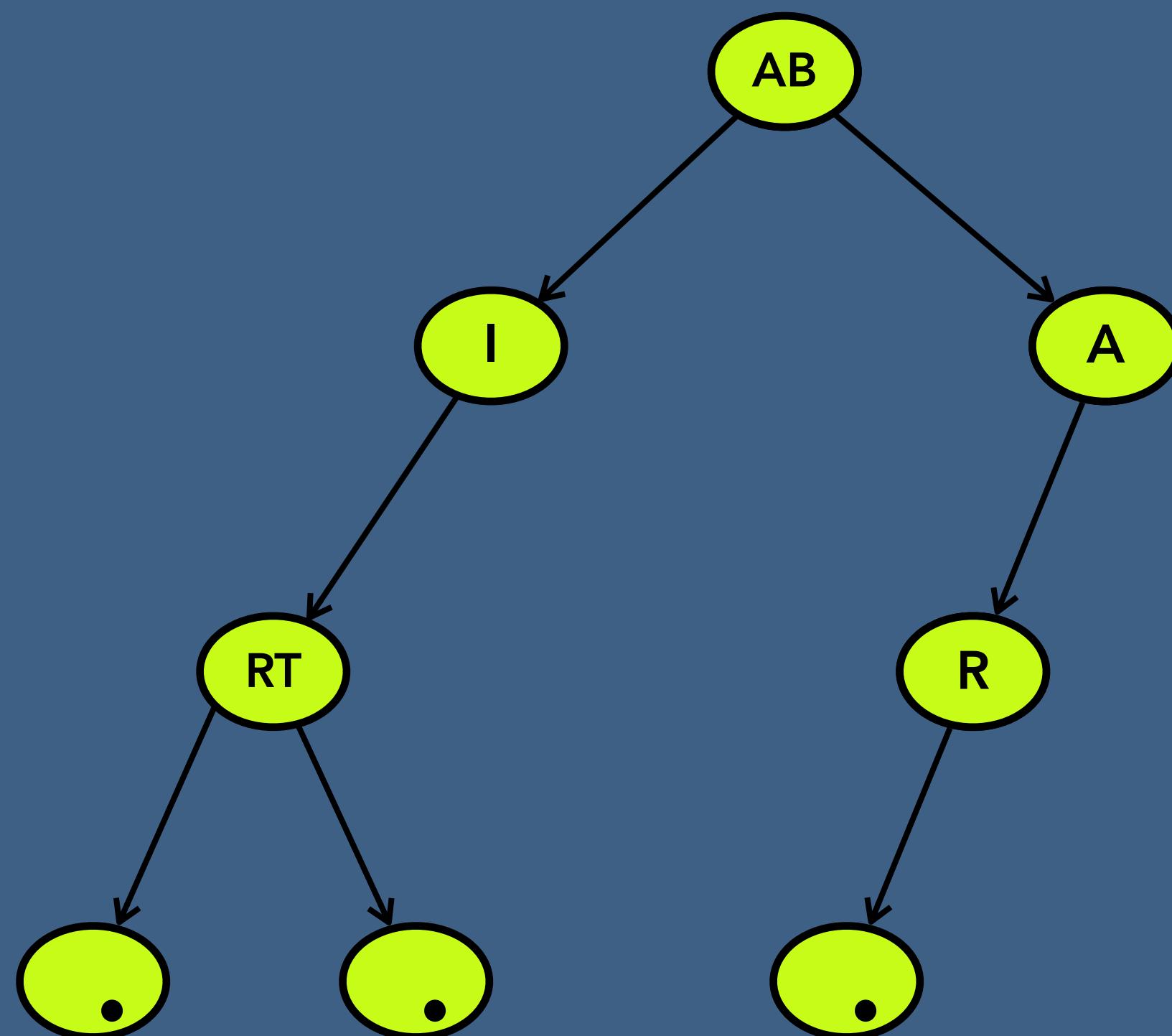
AIT



# What is a Trie?

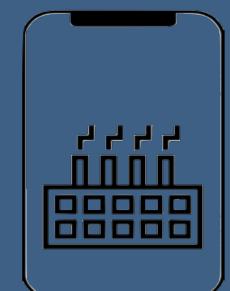
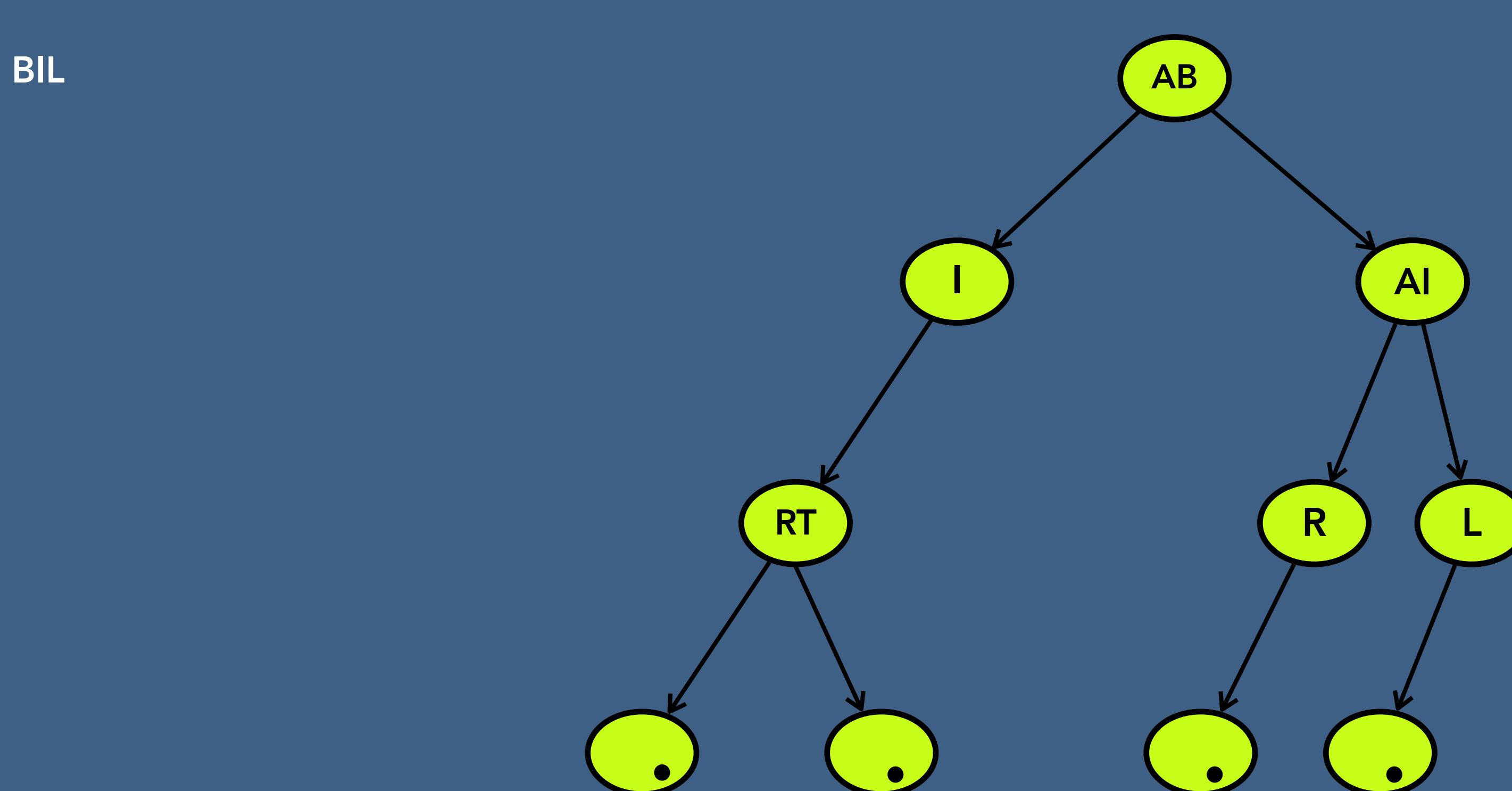
A Trie is a tree-based data structure that organizes information in a hierarchy.

BAR



# What is a Trie?

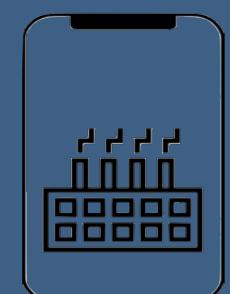
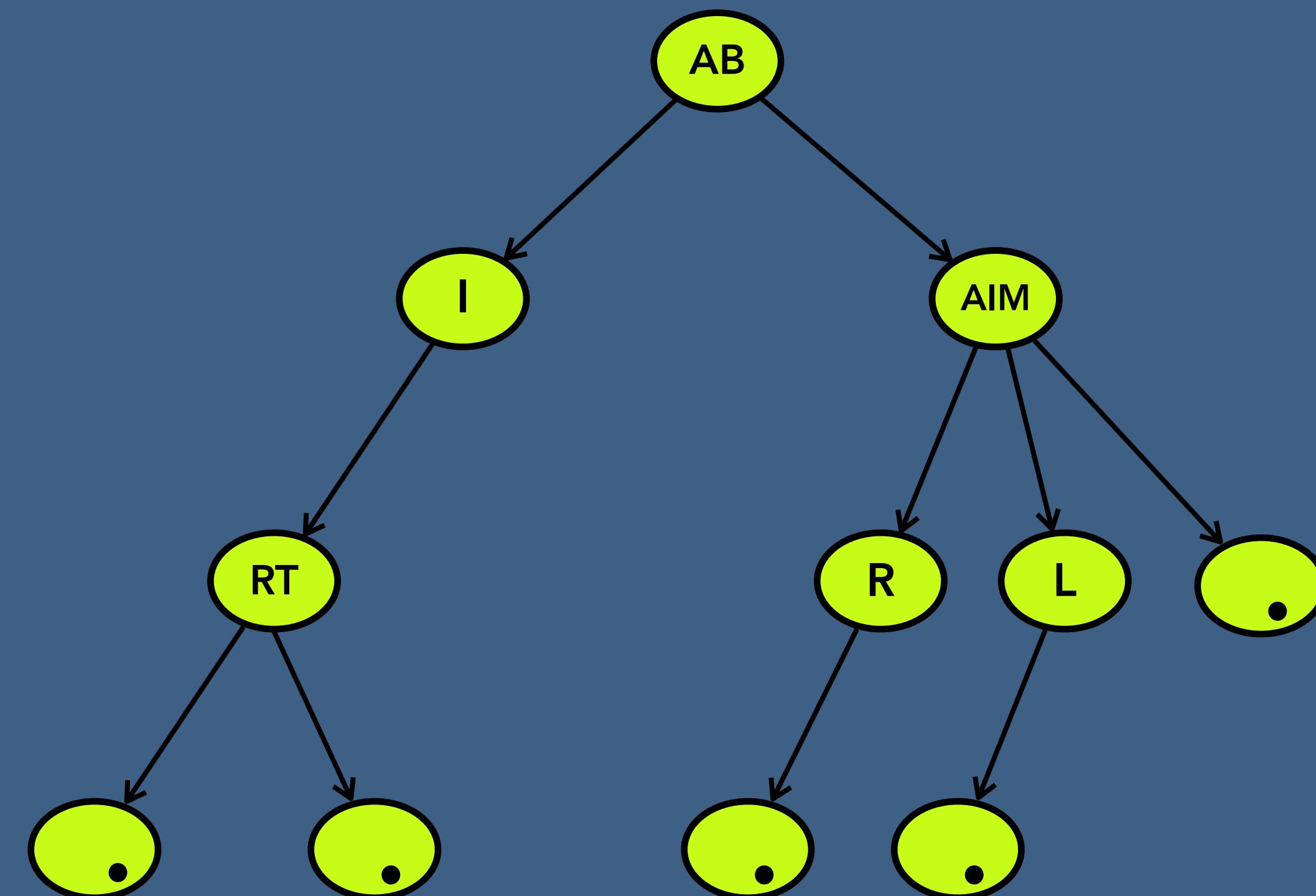
A Trie is a tree-based data structure that organizes information in a hierarchy.



# What is a Trie?

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

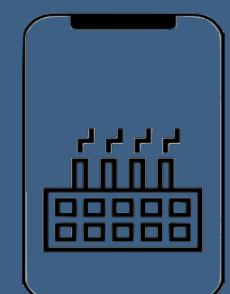
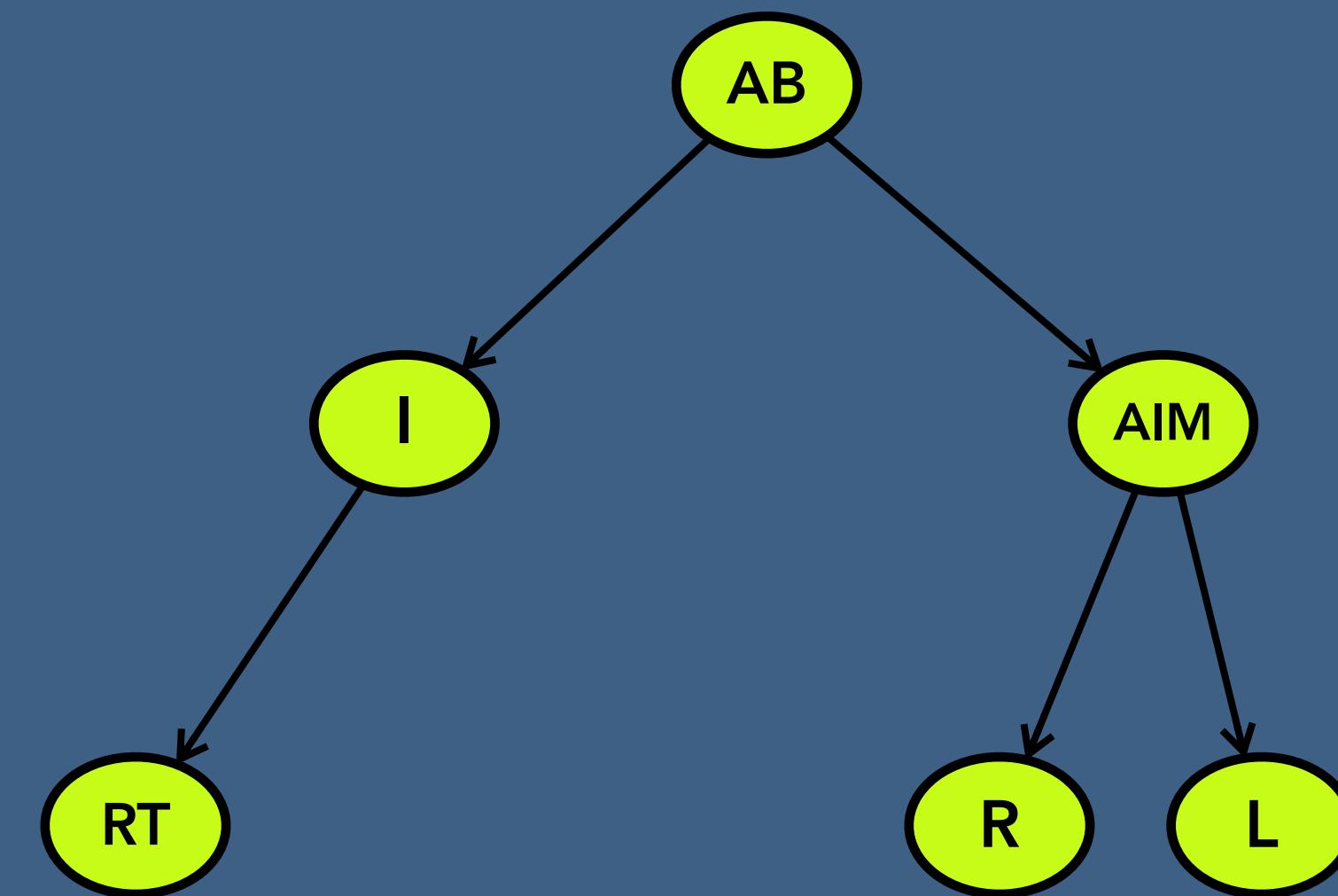
BM



# Why we need Trie?

To solve many standard problems in efficient way

- Spelling checker
- Auto completion

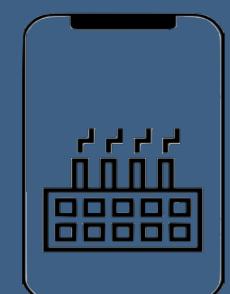
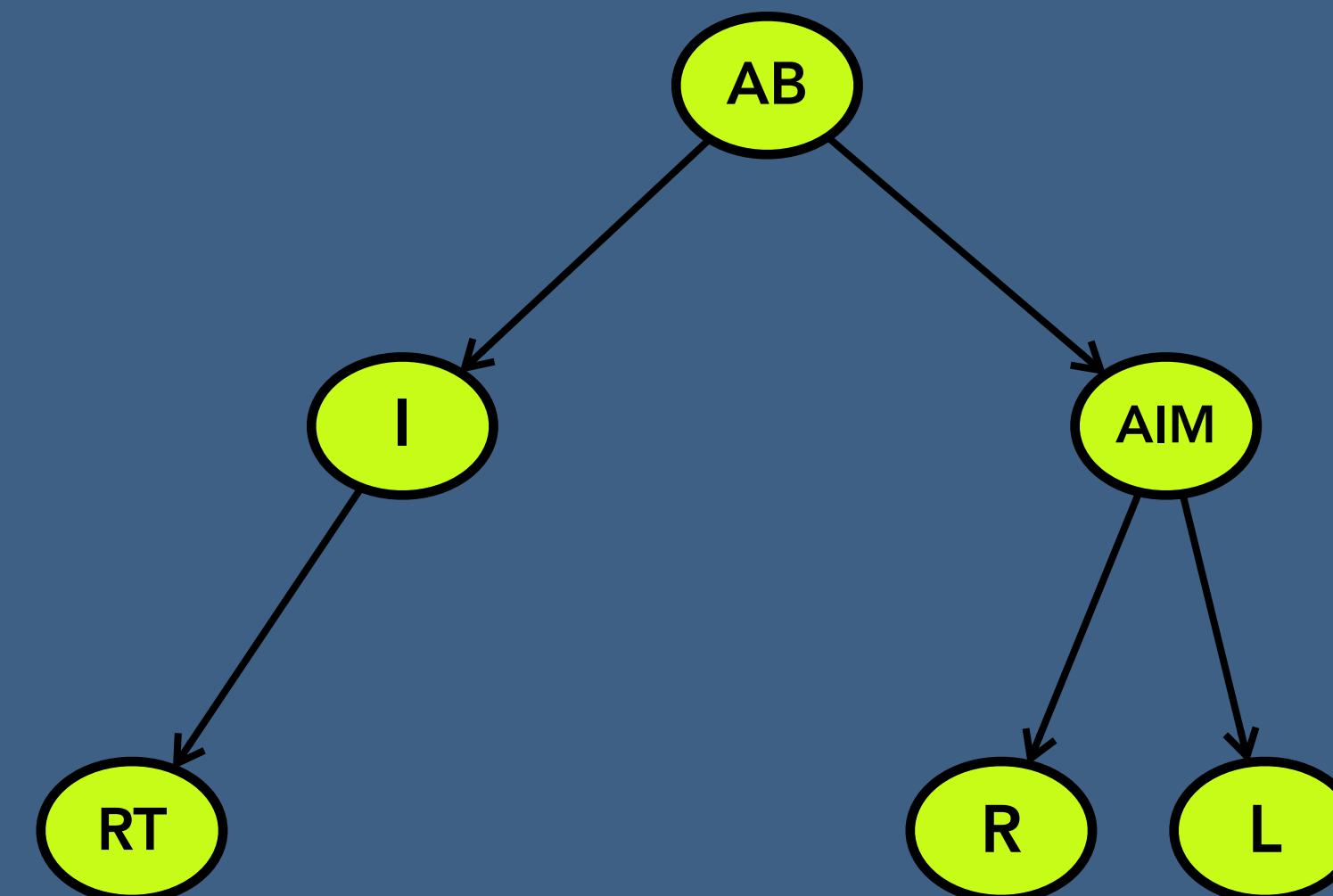


# Why we need Trie?

To solve many standard problems in efficient way

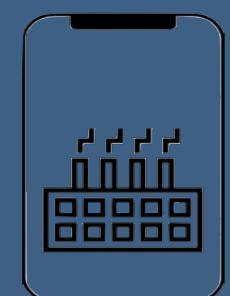
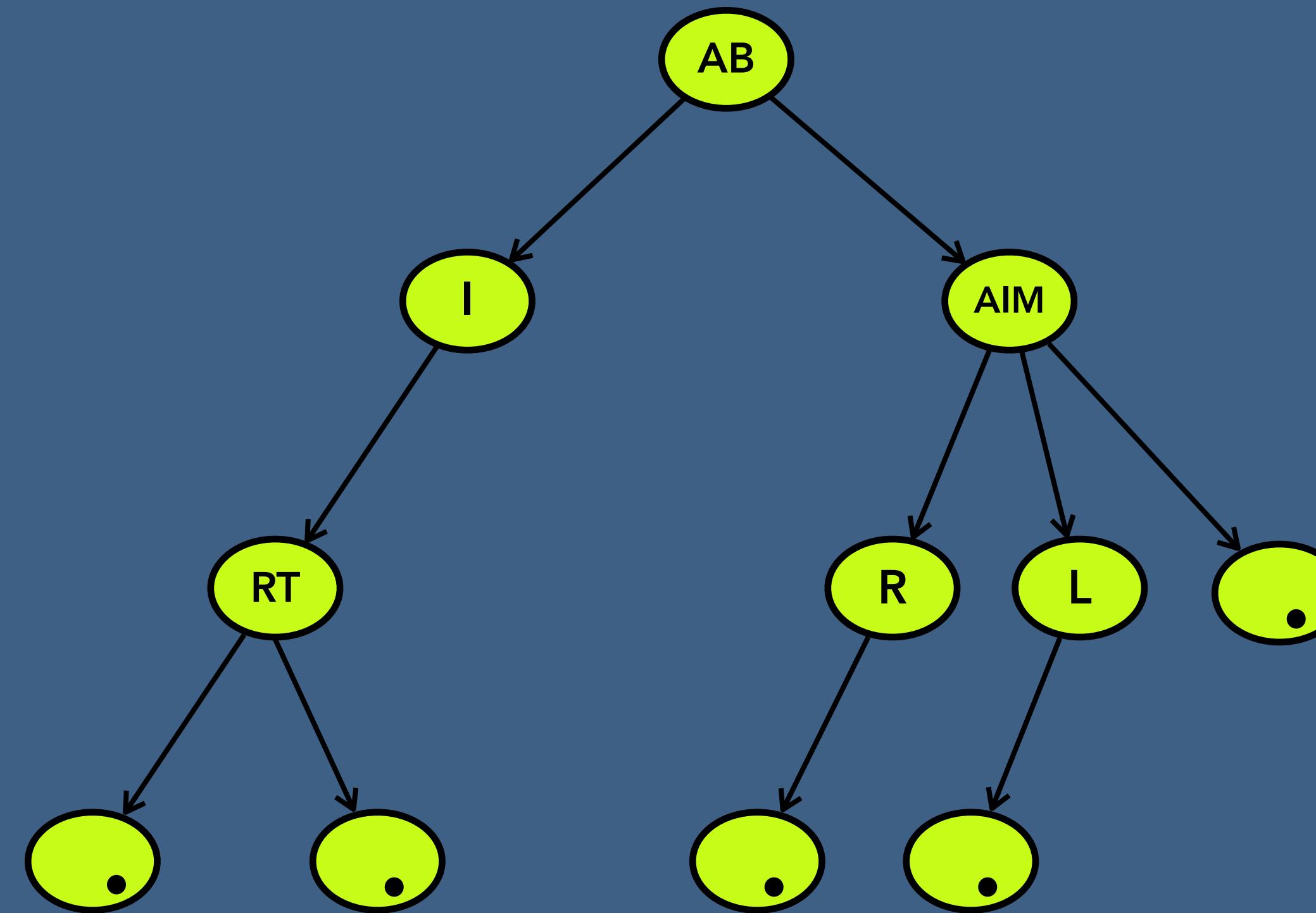
- Spelling checker
- Auto completion

Map	
Characters	Link to Trie Node
End of String	



# Common Operations on Trie

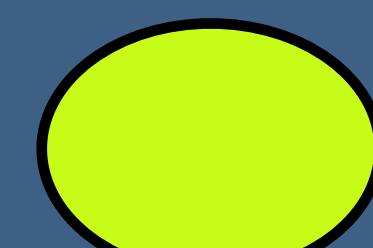
- Creation of Trie
- Insertion in Trie
- Search for a String in trie
- Deletion from Trie



# Common Operations on Trie

- Creation of Trie

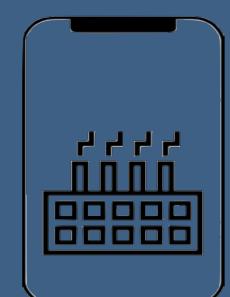
Initialize Trie() class



Logical

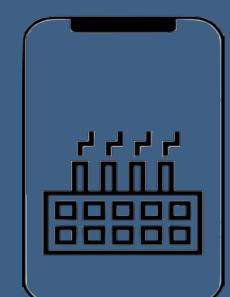
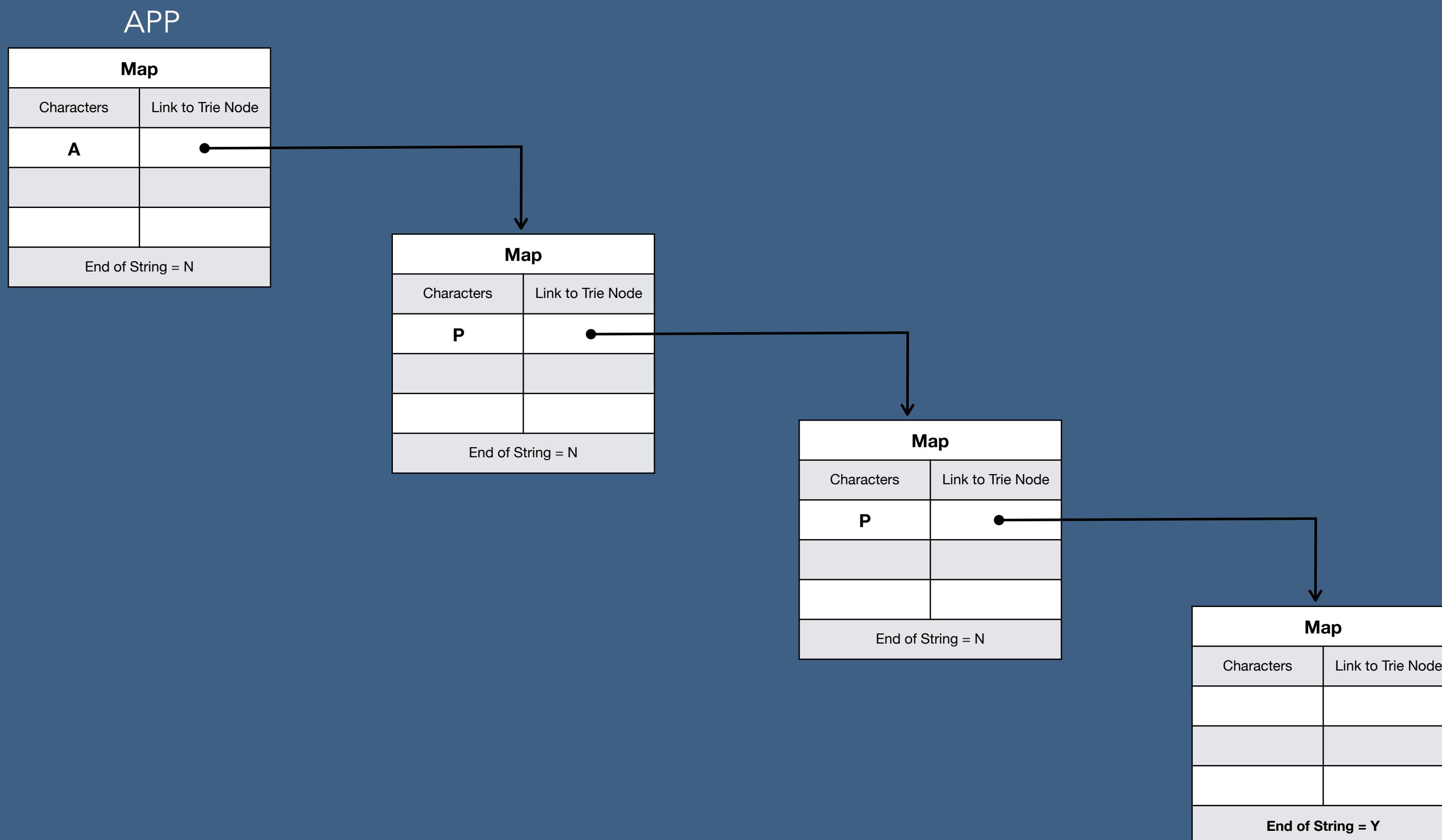
Physically

Map	
Characters	Link to Trie Node
End of String	



# Insert a String in a Trie

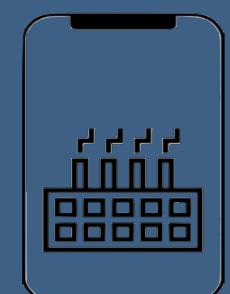
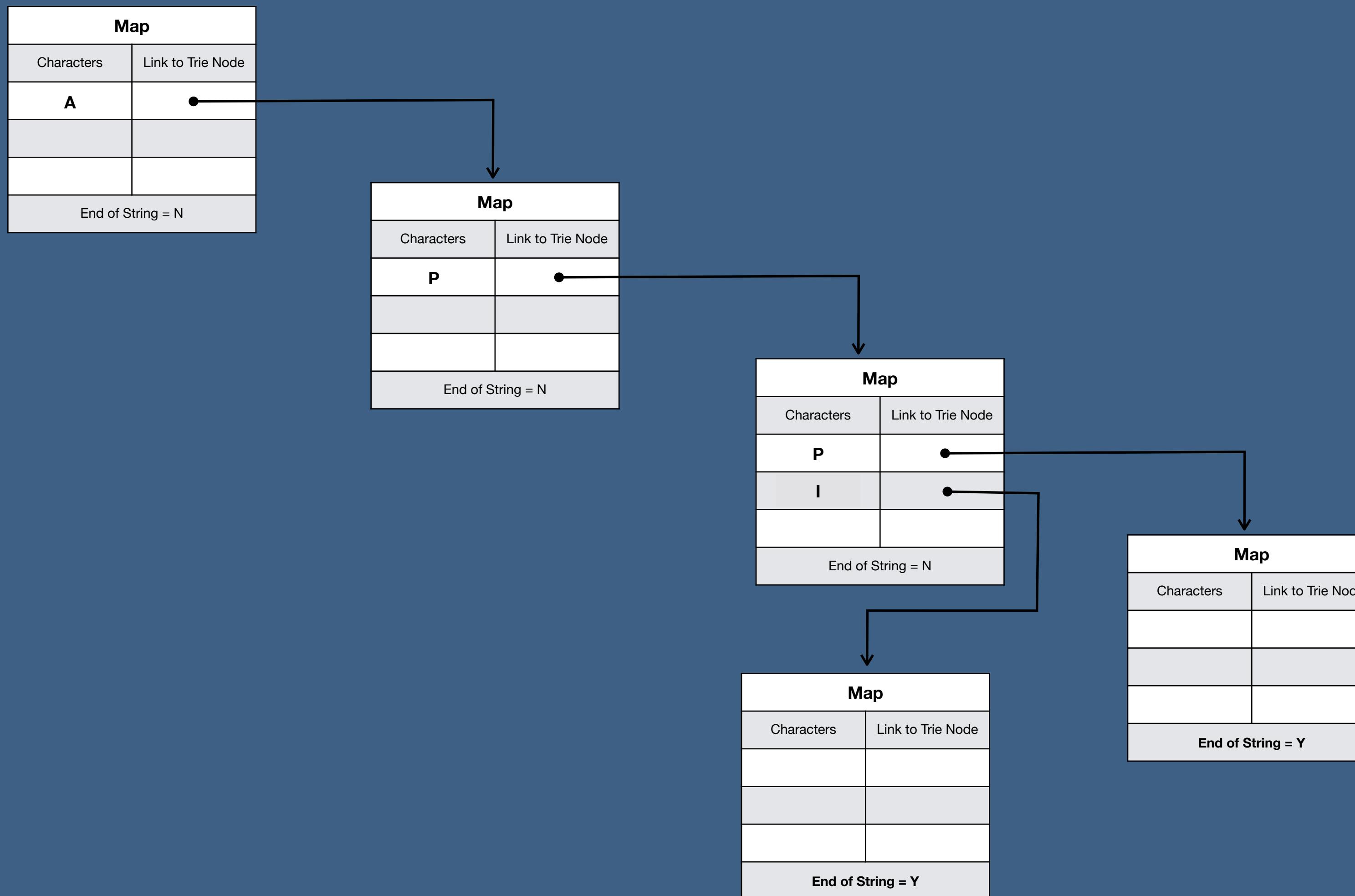
Case 1 : A Trie is Blank



# Insert a String in a Trie

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

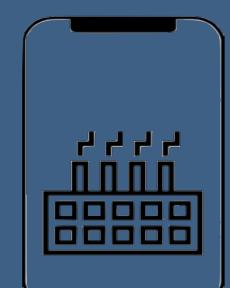
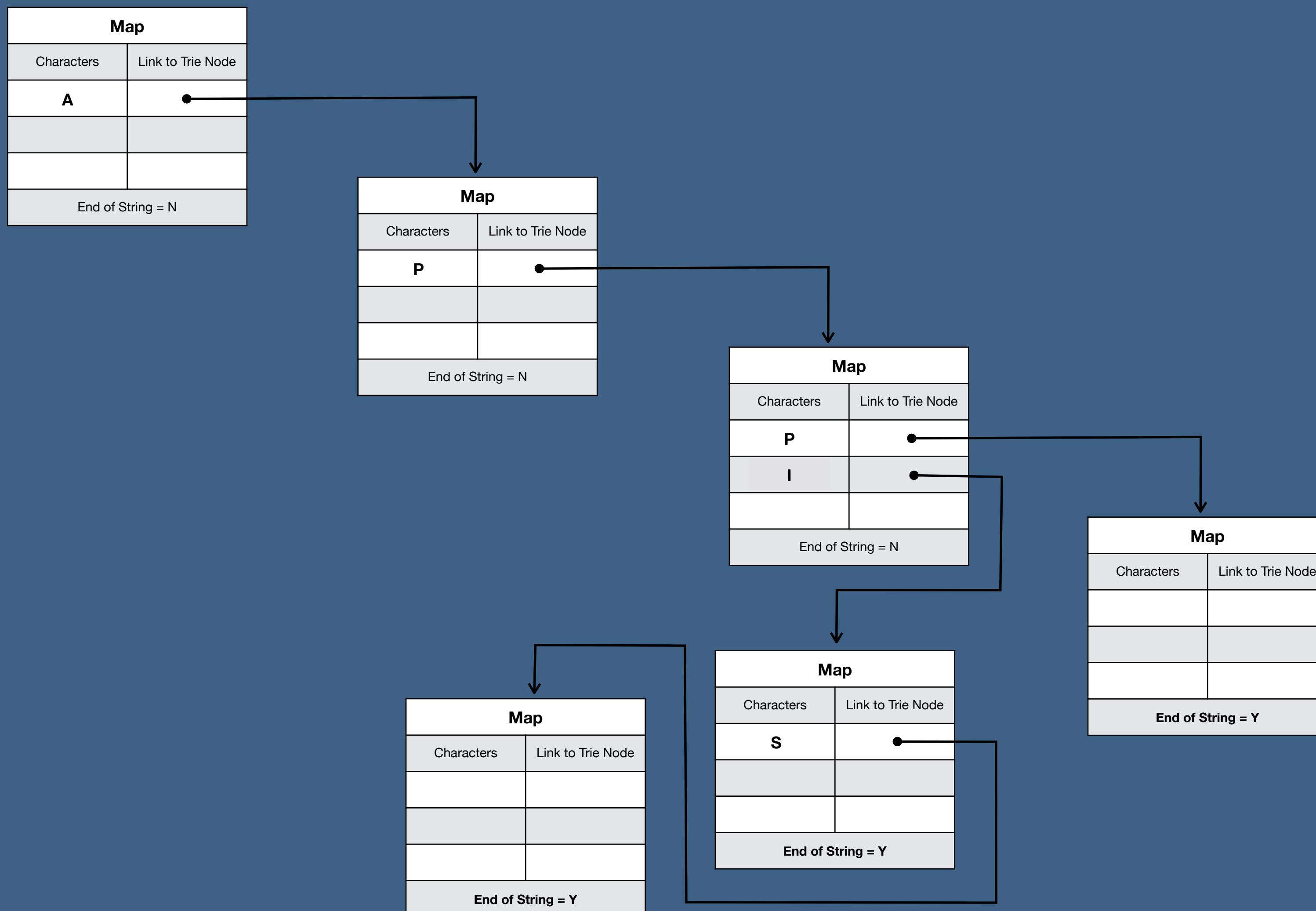
API



# Insert a String in a Trie

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

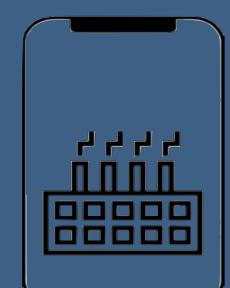
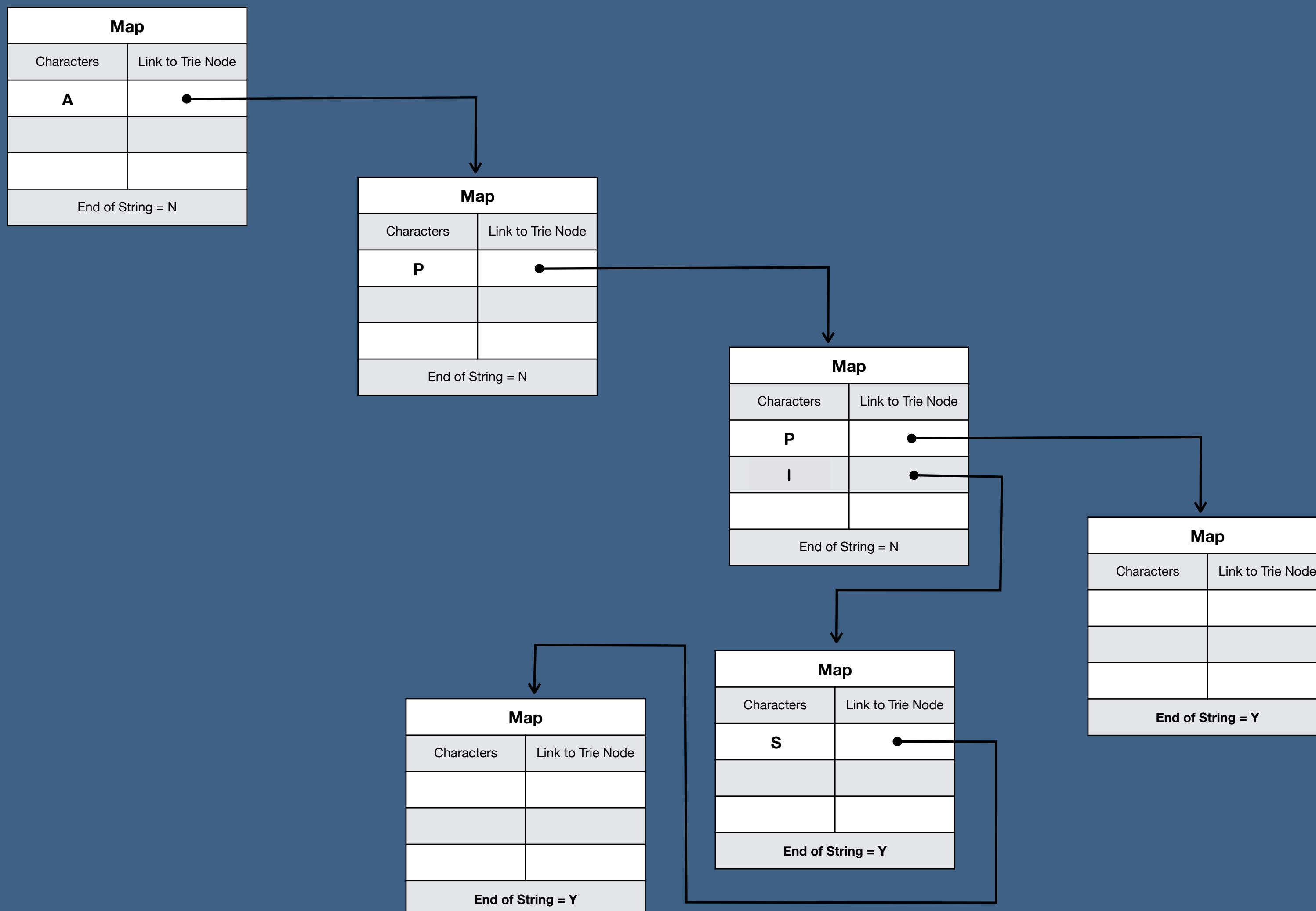
APIS



# Insert a String in a Trie

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

APIS

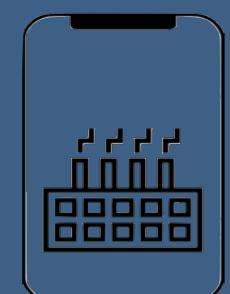
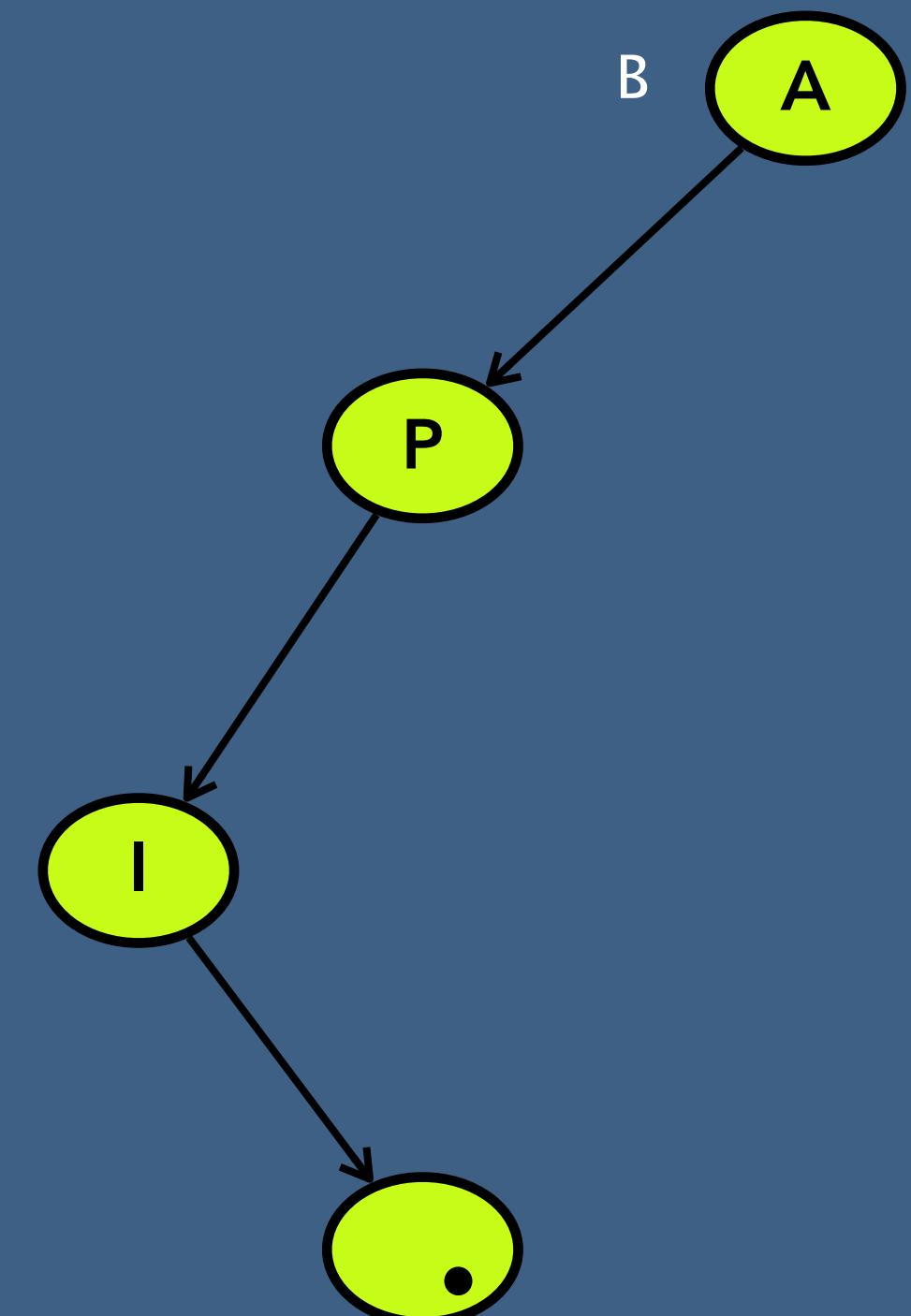


# Search for a String in a Trie

Case 1: String does not exist in Trie

BCD

Return : The string does not exist in Trie

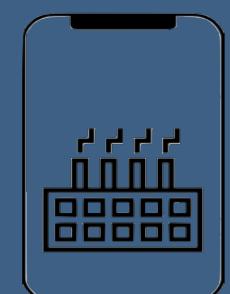
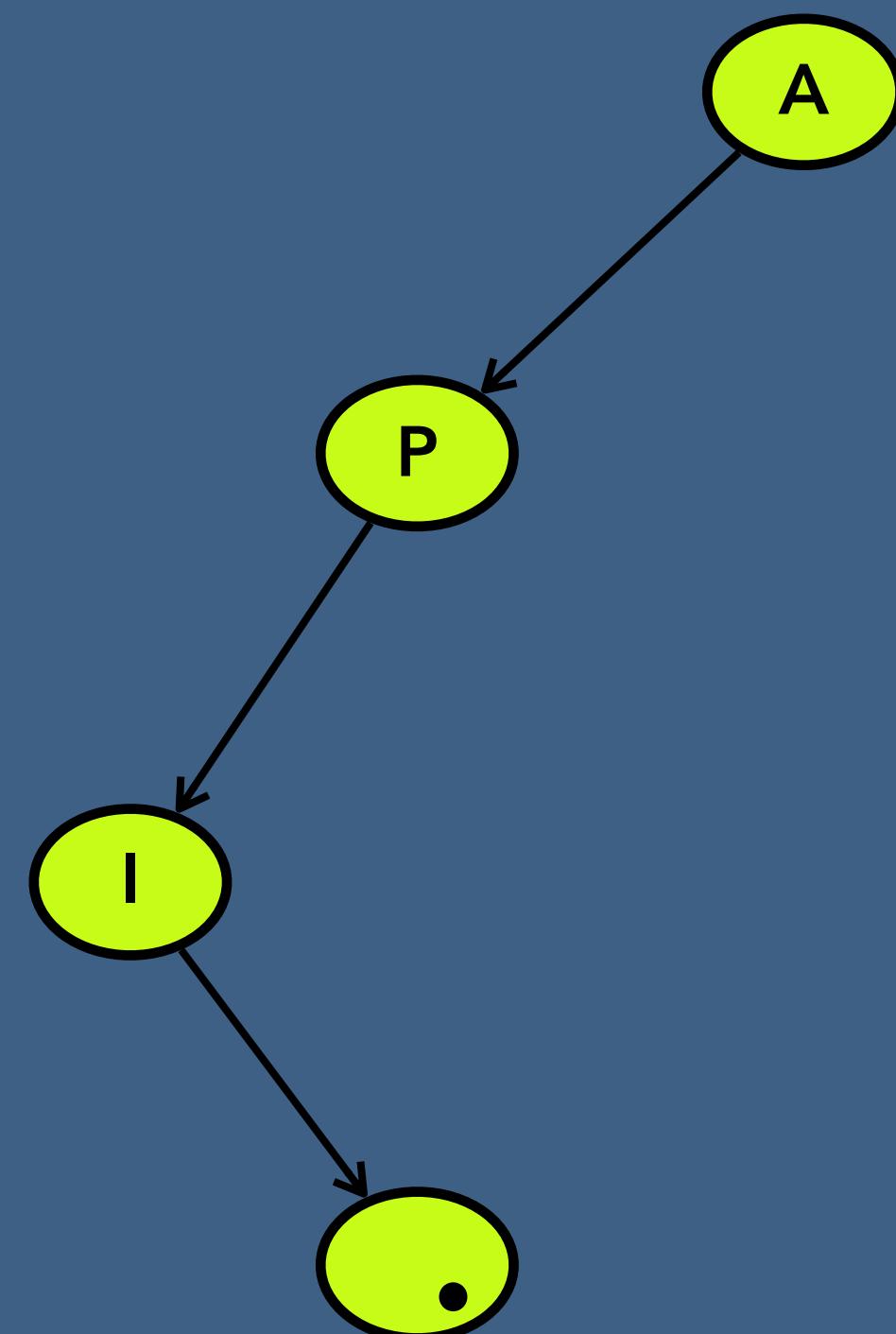


# Search for a String in a Trie

Case 2: String exists in Trie

API

Return : TRUE

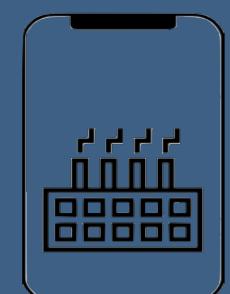
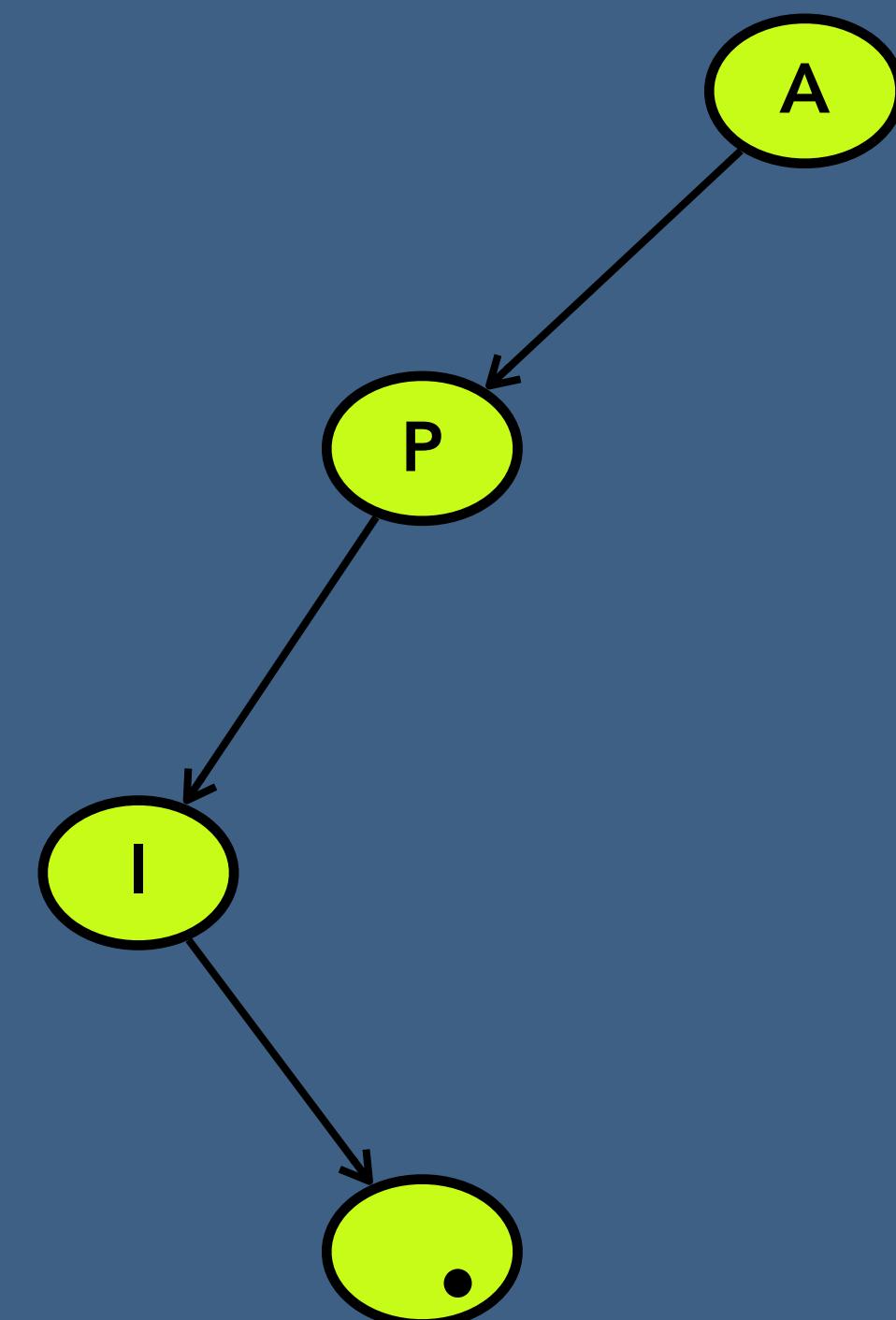


# Search for a String in a Trie

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

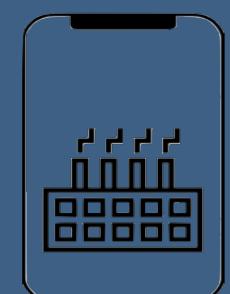
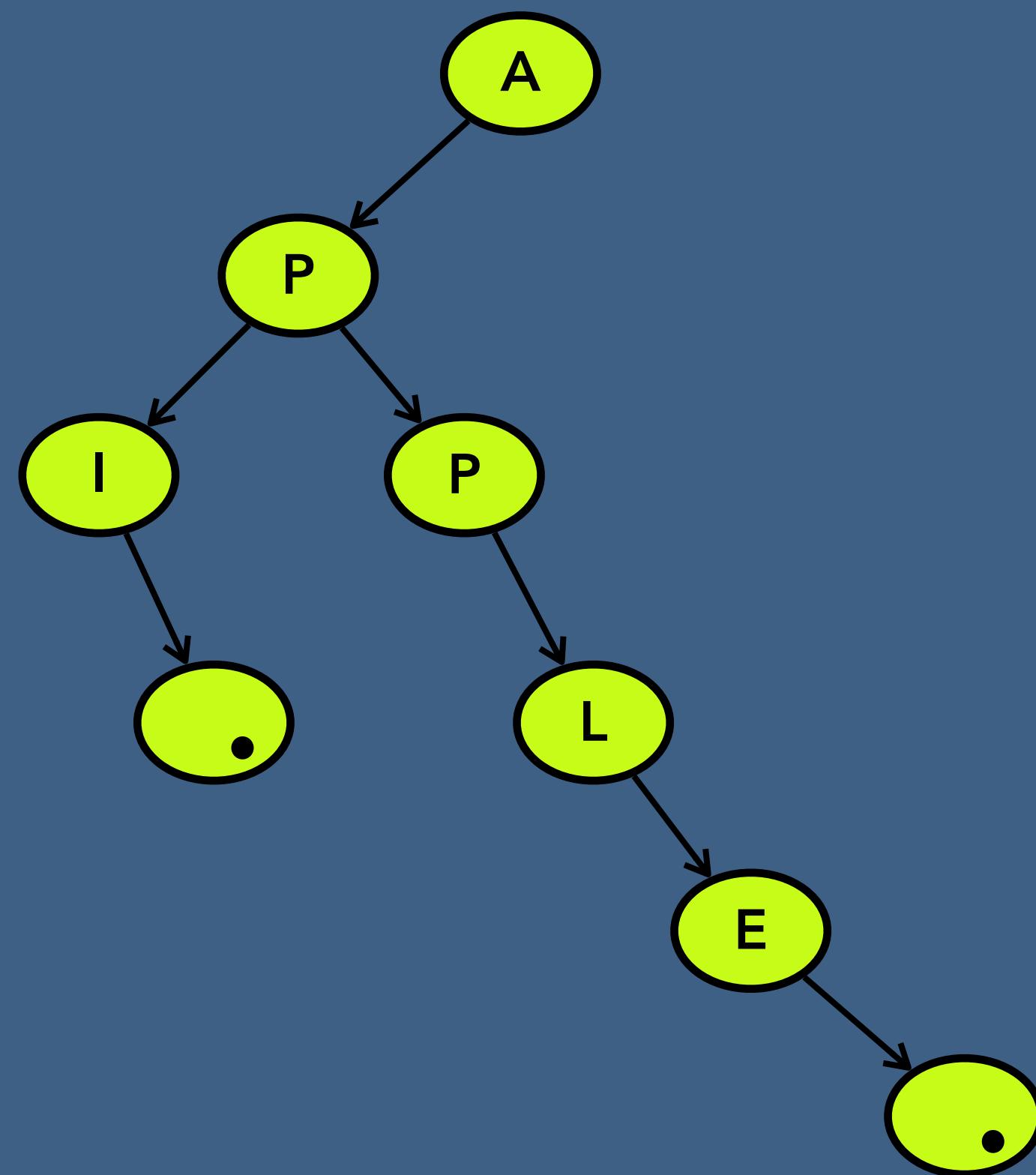
AP

Return : FALSE



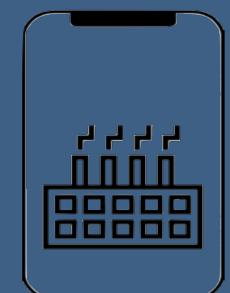
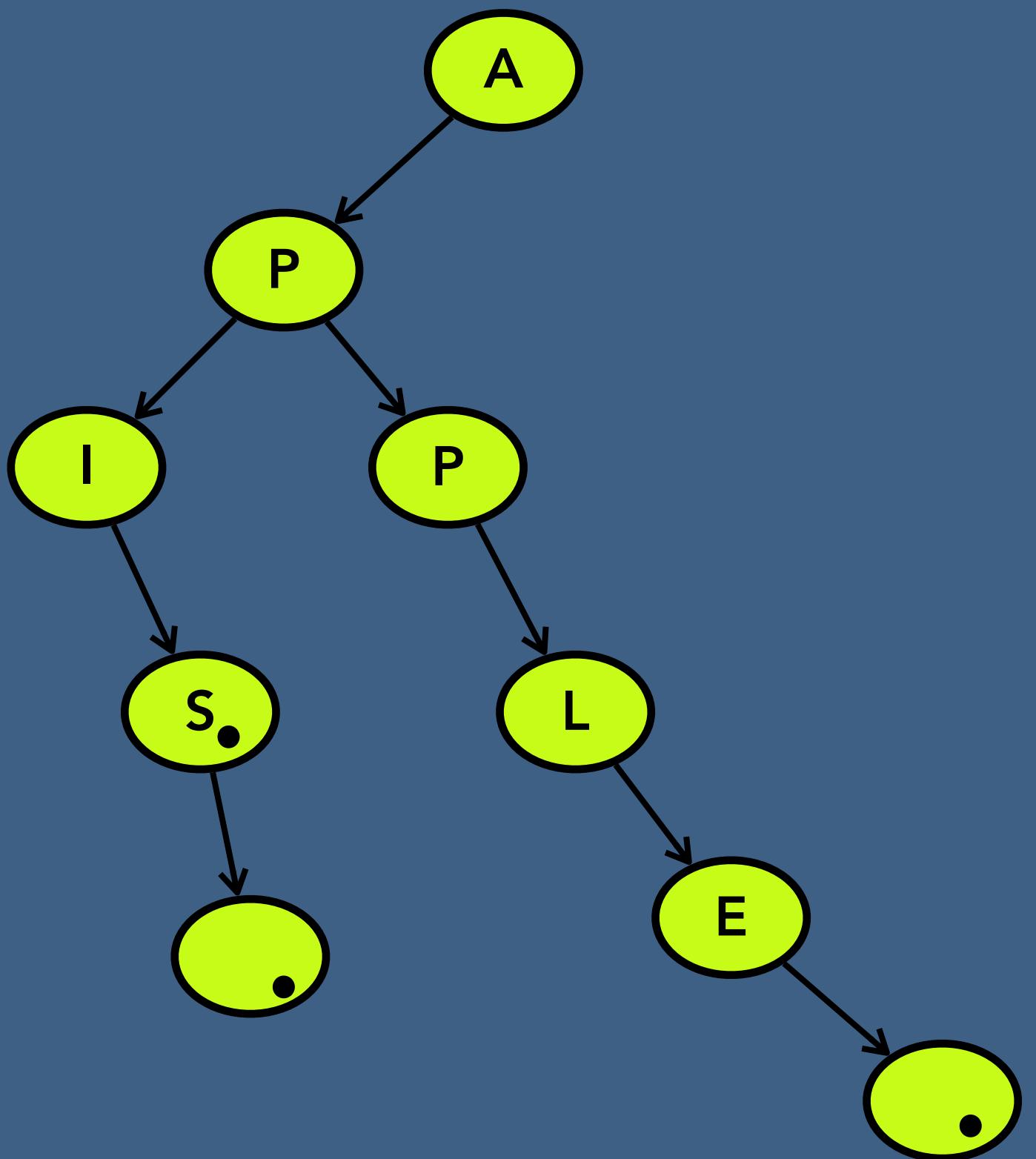
# Delete a String from Trie

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



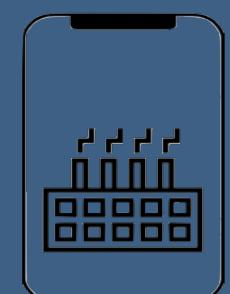
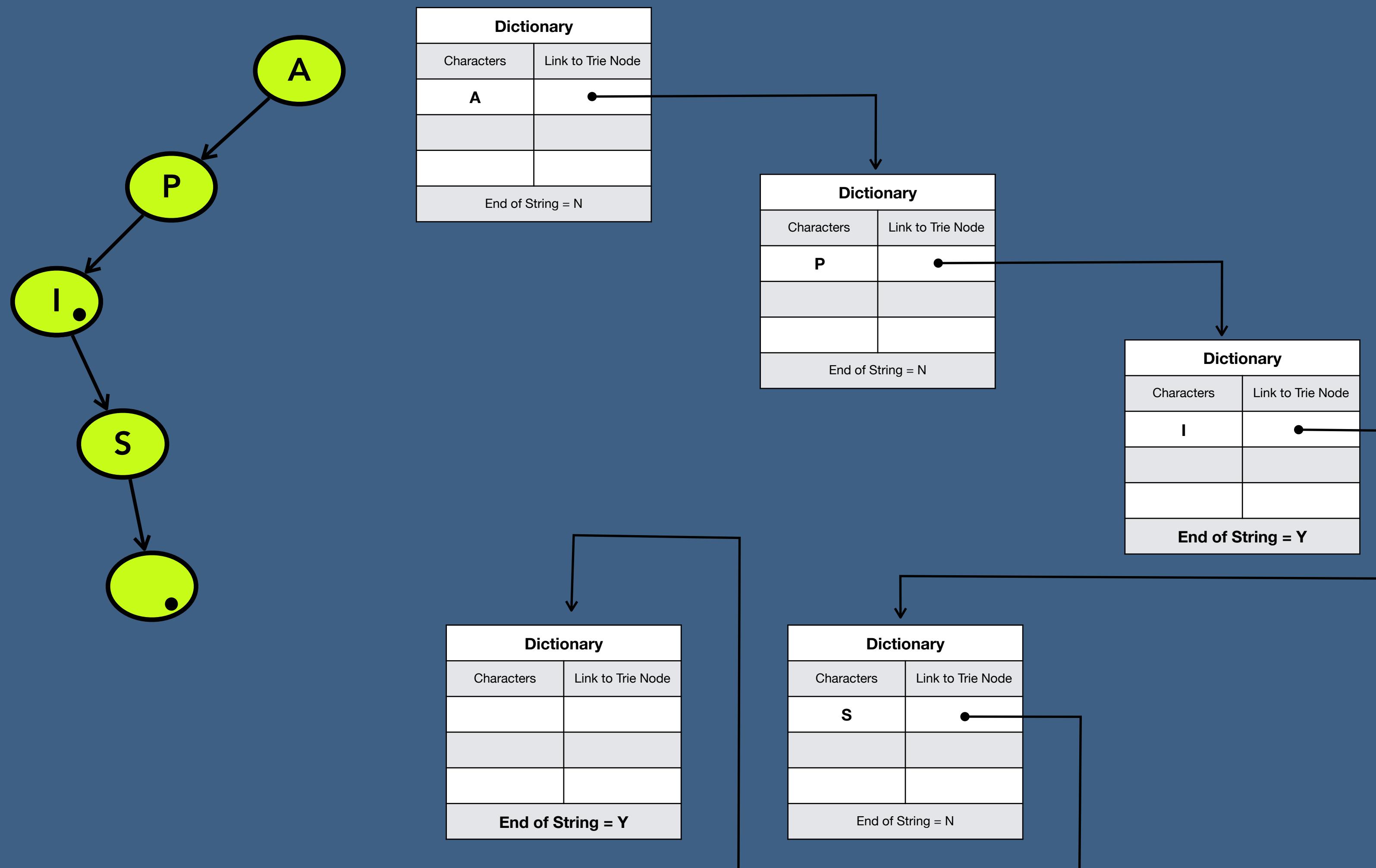
# Delete a String from Trie

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



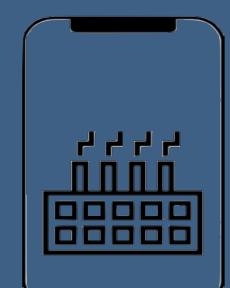
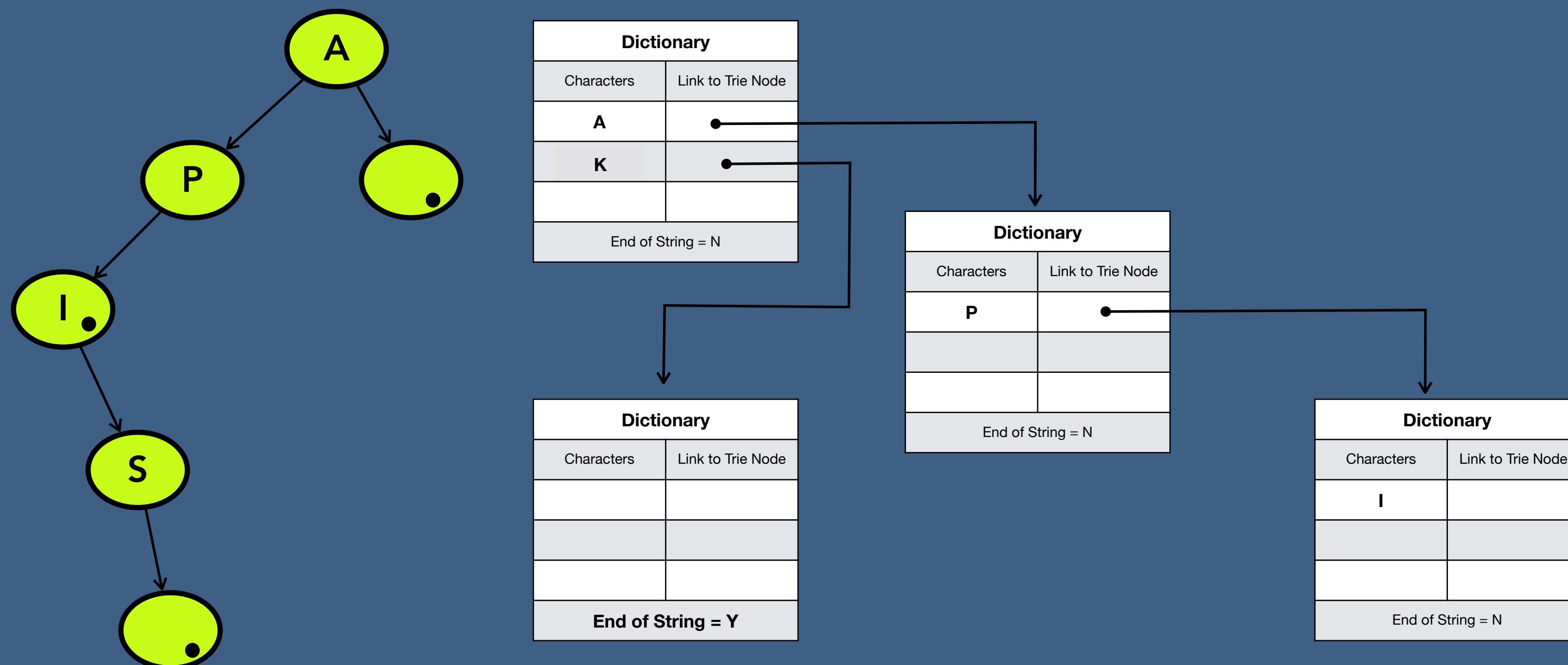
# Delete a String from Trie

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



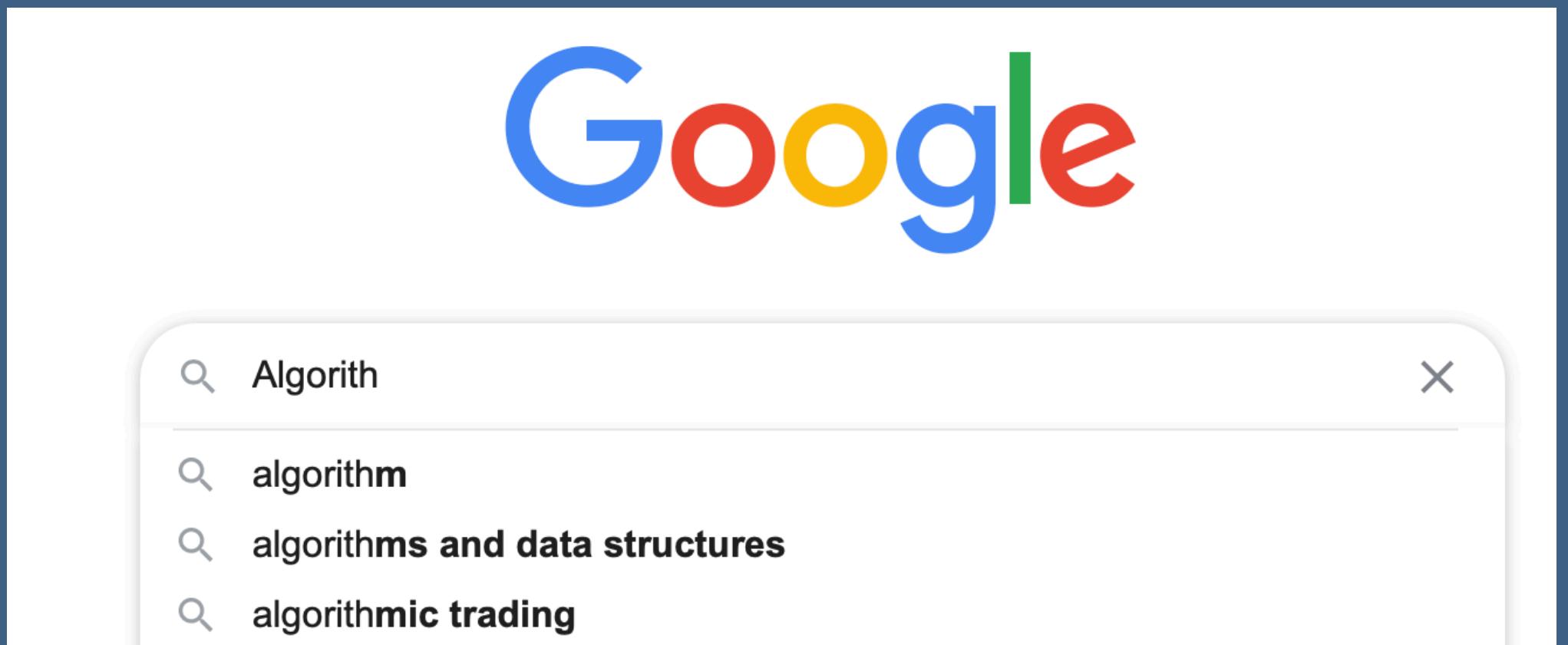
# Delete a String from Trie

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

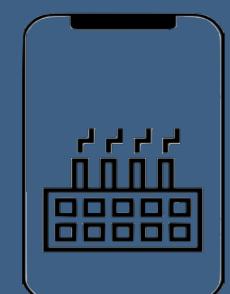


# Practical use of Trie

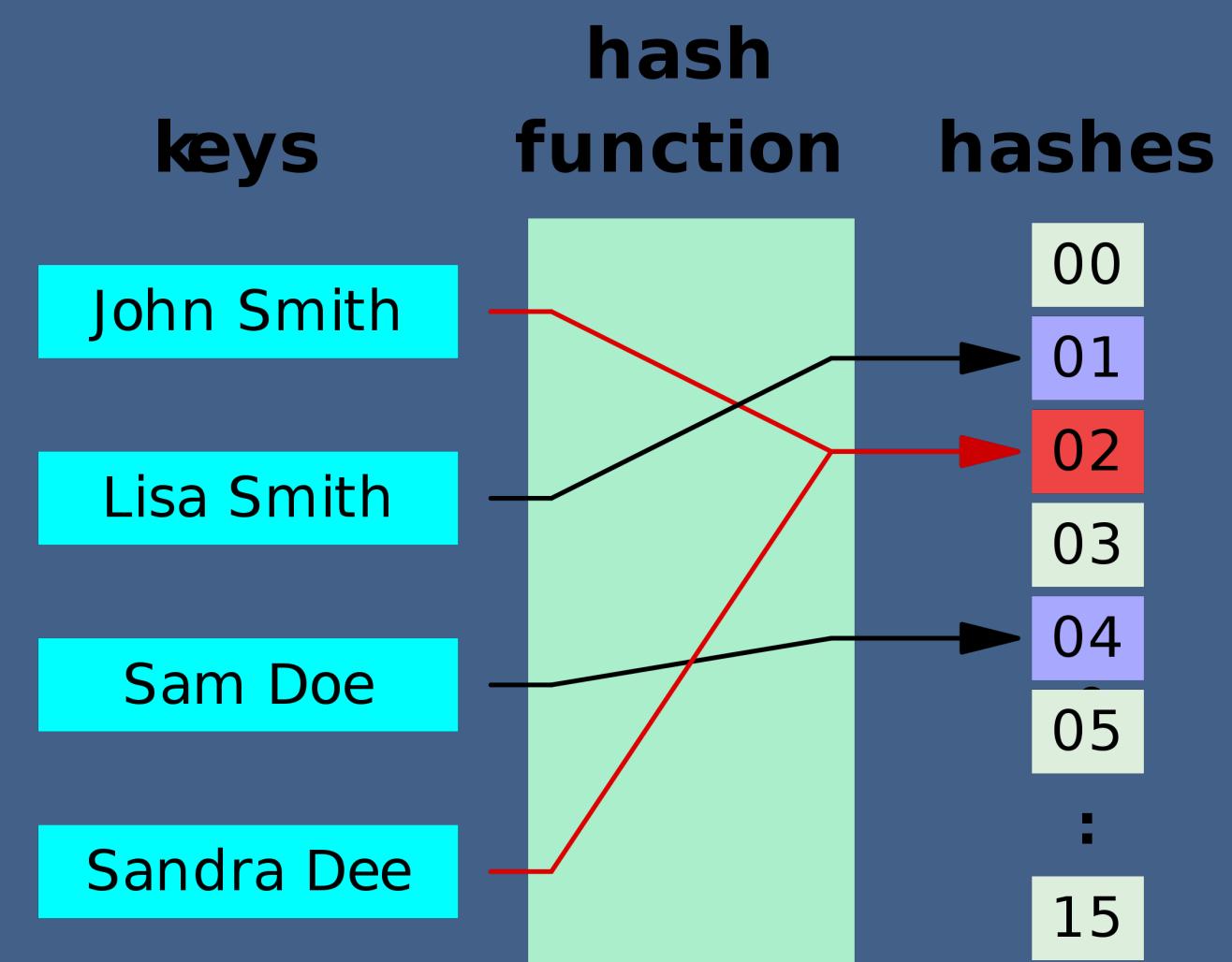
- Auto completion



- Spelling checker

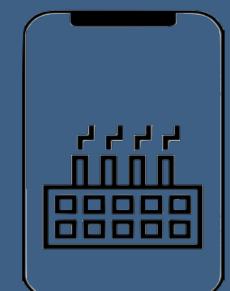
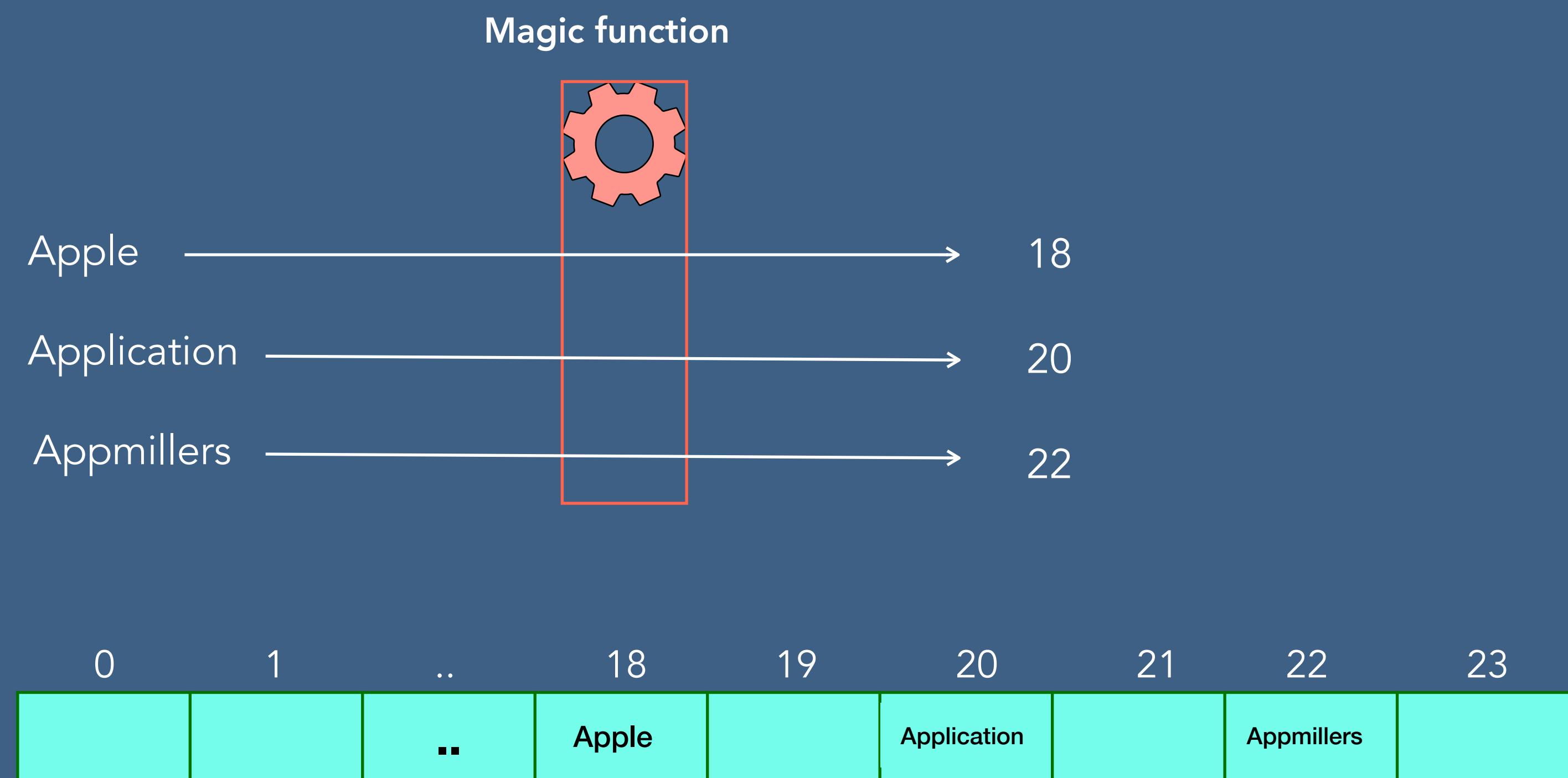


# Hashing



# What is Hashing?

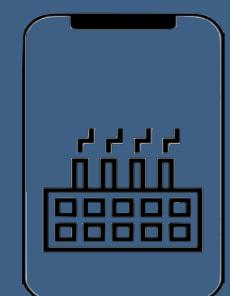
Hashing is a method of sorting and indexing data. The idea behind hashing is to allow large amounts of data to be indexed using keys commonly created by formulas



# Why Hashing?

It is time efficient in case of SEARCH Operation

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



# Hashing Terminology

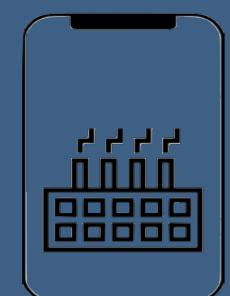
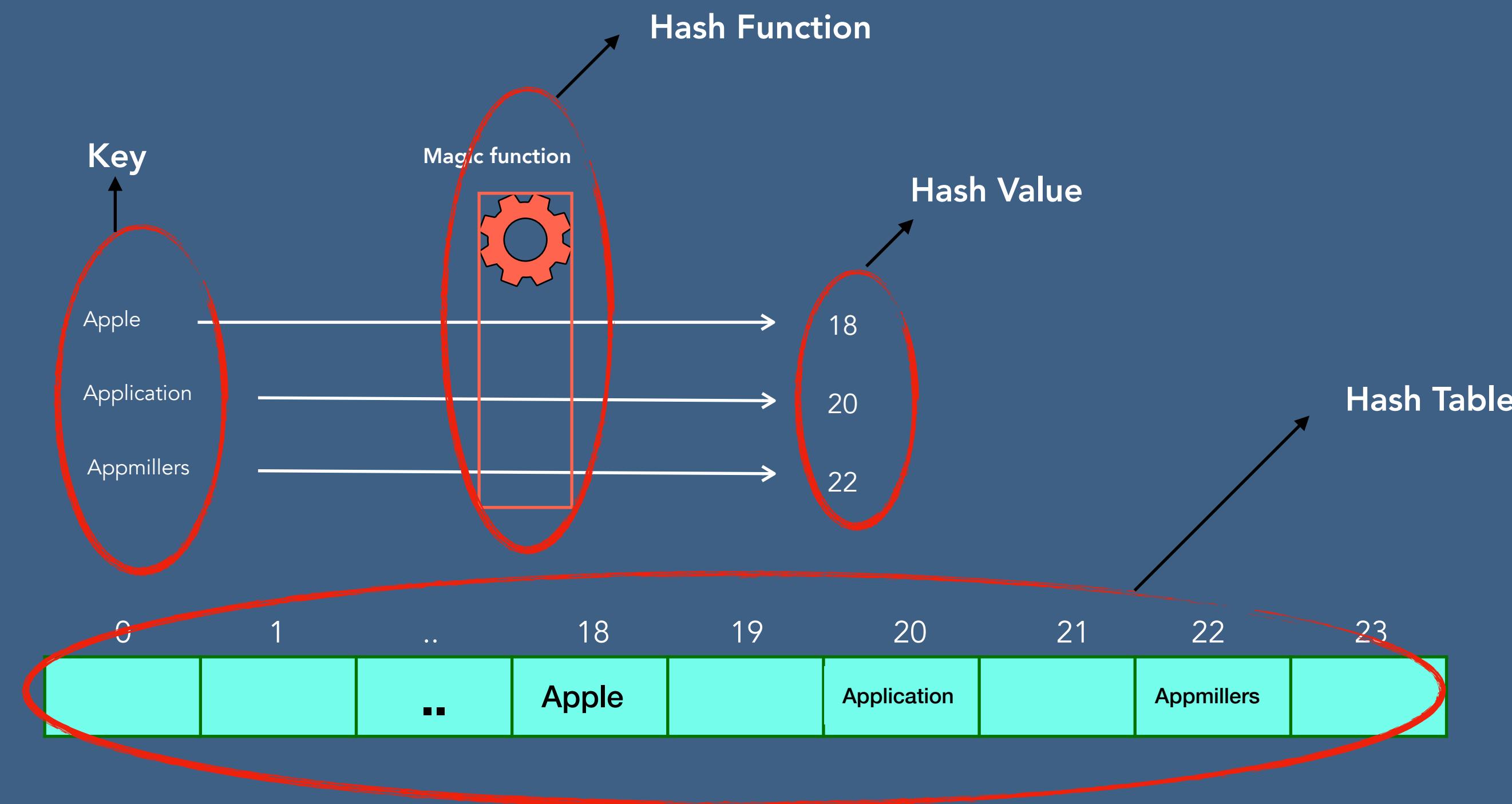
**Hash function** : It is a function that can be used to map of arbitrary size to data of fixed size.

**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

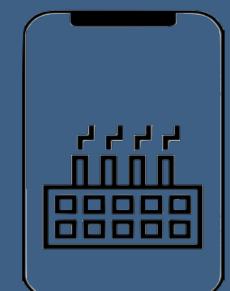
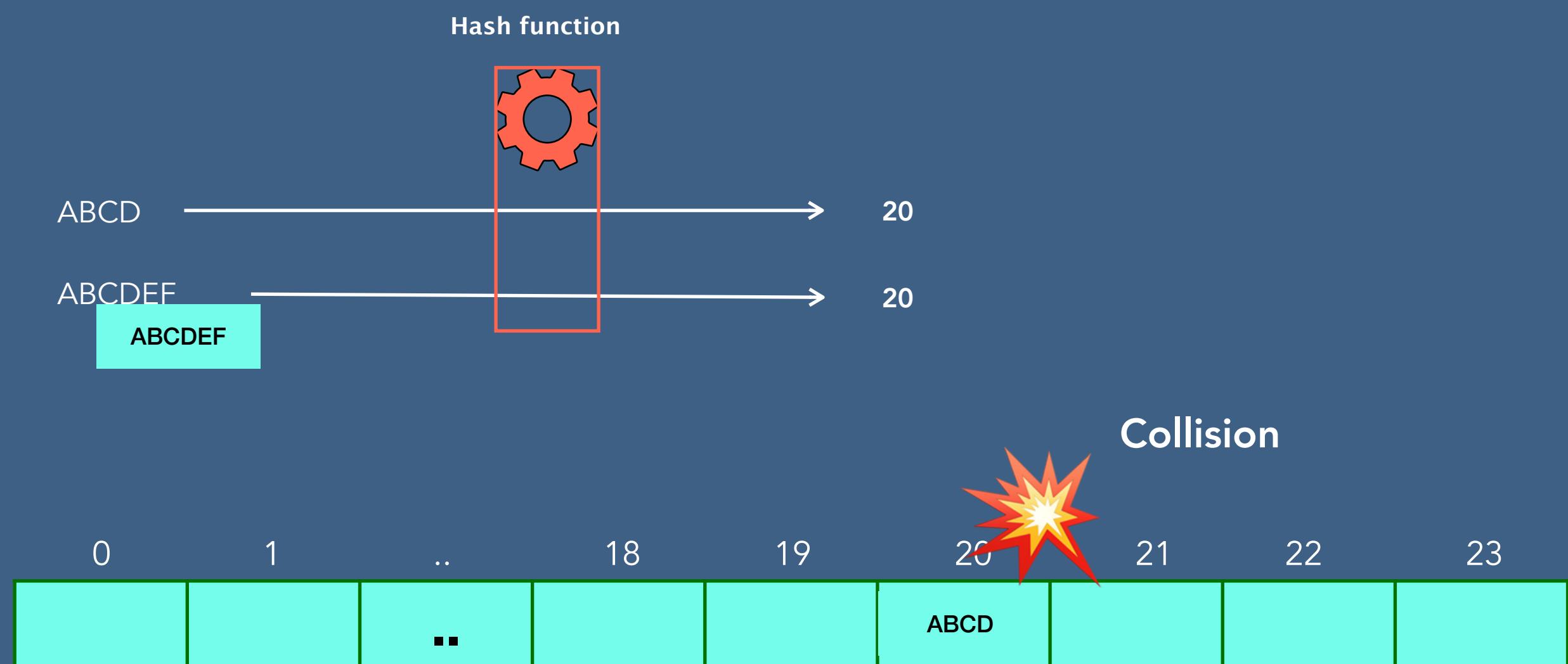
**Hash function** : It is a function that can be used to map of arbitrary size to data of fixed size.

**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.



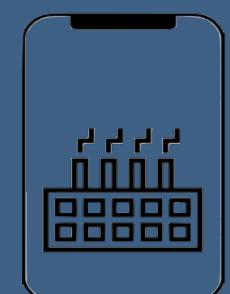
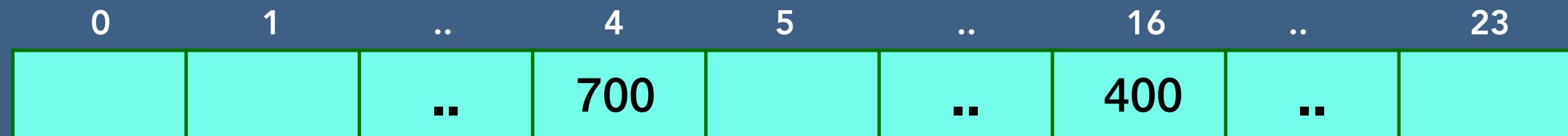
# Hash Functions

## Mod function

```
int mod(int number, int cellNumber) {  
    return number % cellNumber;  
}
```

mod(400, 24) → 16

mod(700, 24) → 4

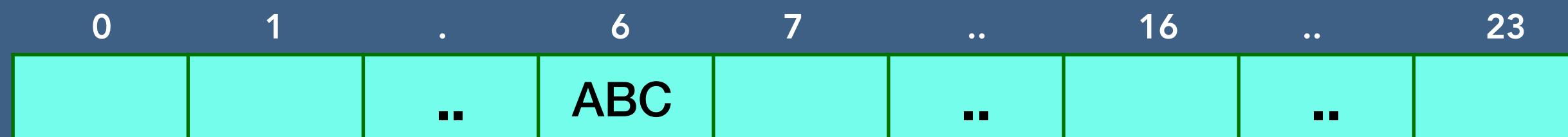


# Hash Functions

## ASCII function

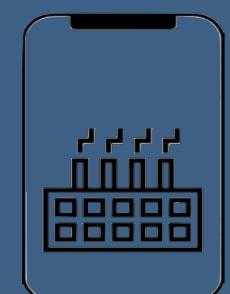
```
public int modASCII(String word, int cellNumber) {
    int total = 0;
    for (int i=0; i<word.length(); i++) {
        total += word.charAt(i);
        System.out.println(total);
    }
    return total % cellNumber;
}
```

$$\begin{array}{l}
 \text{modASCII("ABC", 24)} \longrightarrow 6 \\
 \\
 \begin{array}{ll}
 \text{A} \longrightarrow 65 & 65+66+67 = 198 \quad | \quad 24 \\
 & 192 \quad | \quad 8 \\
 \text{B} \longrightarrow 66 & \\
 & \text{C} \longrightarrow 67
 \end{array}
 \end{array}$$



## ASCII Table

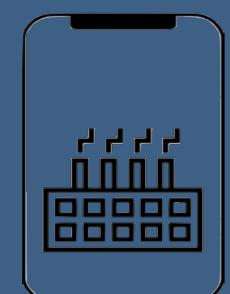
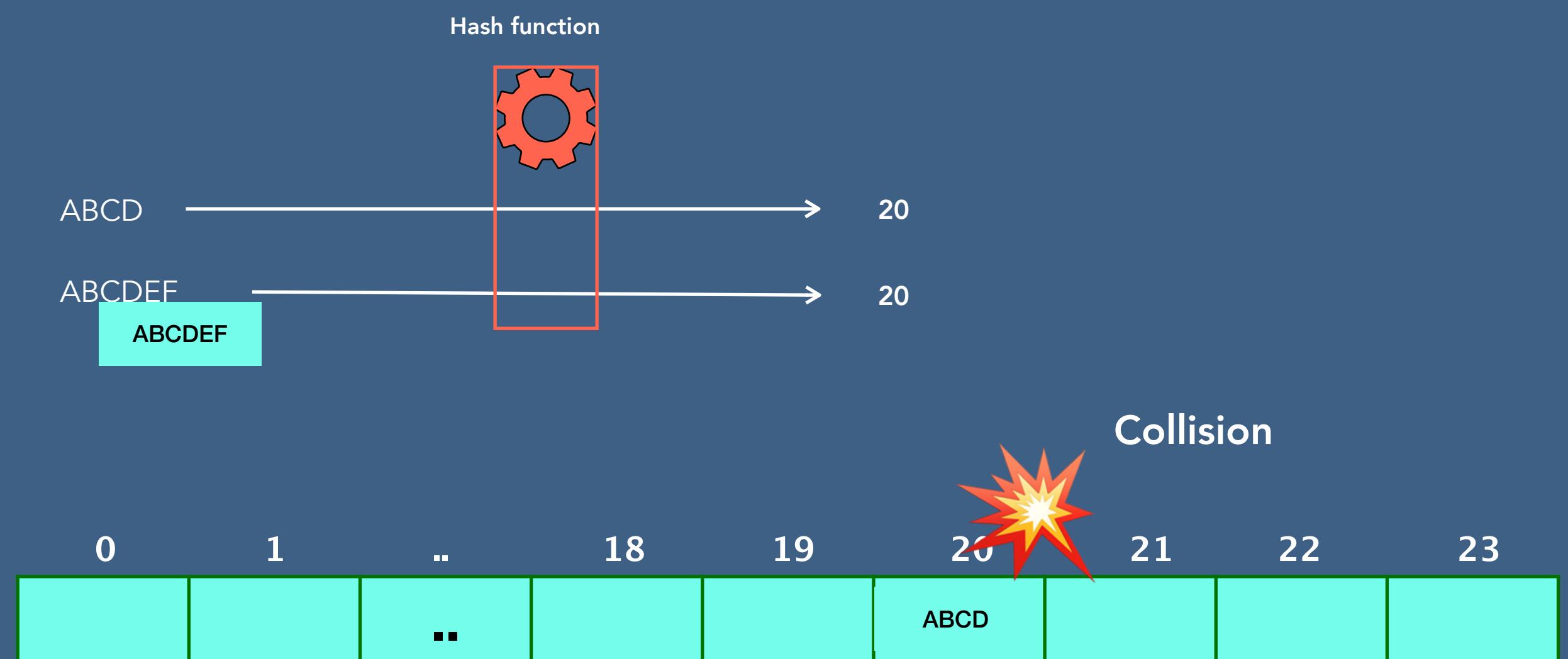
Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@
1	1	1	!	33	21	41	"	65	41	101	A
2	2	2	#	34	22	42	&	66	42	102	B
3	3	3	\$	35	23	43	'	67	43	103	C
4	4	4	(	36	24	44	)	68	44	104	D
5	5	5	*	37	25	45	%	69	45	105	E
6	6	6	*	38	26	46	&	70	46	106	F
7	7	7	+	39	27	47	,	71	47	107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



# Hash Functions

## Properties of good Hash function

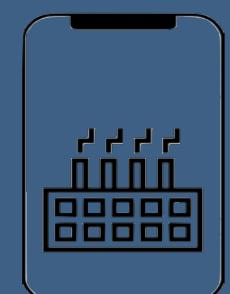
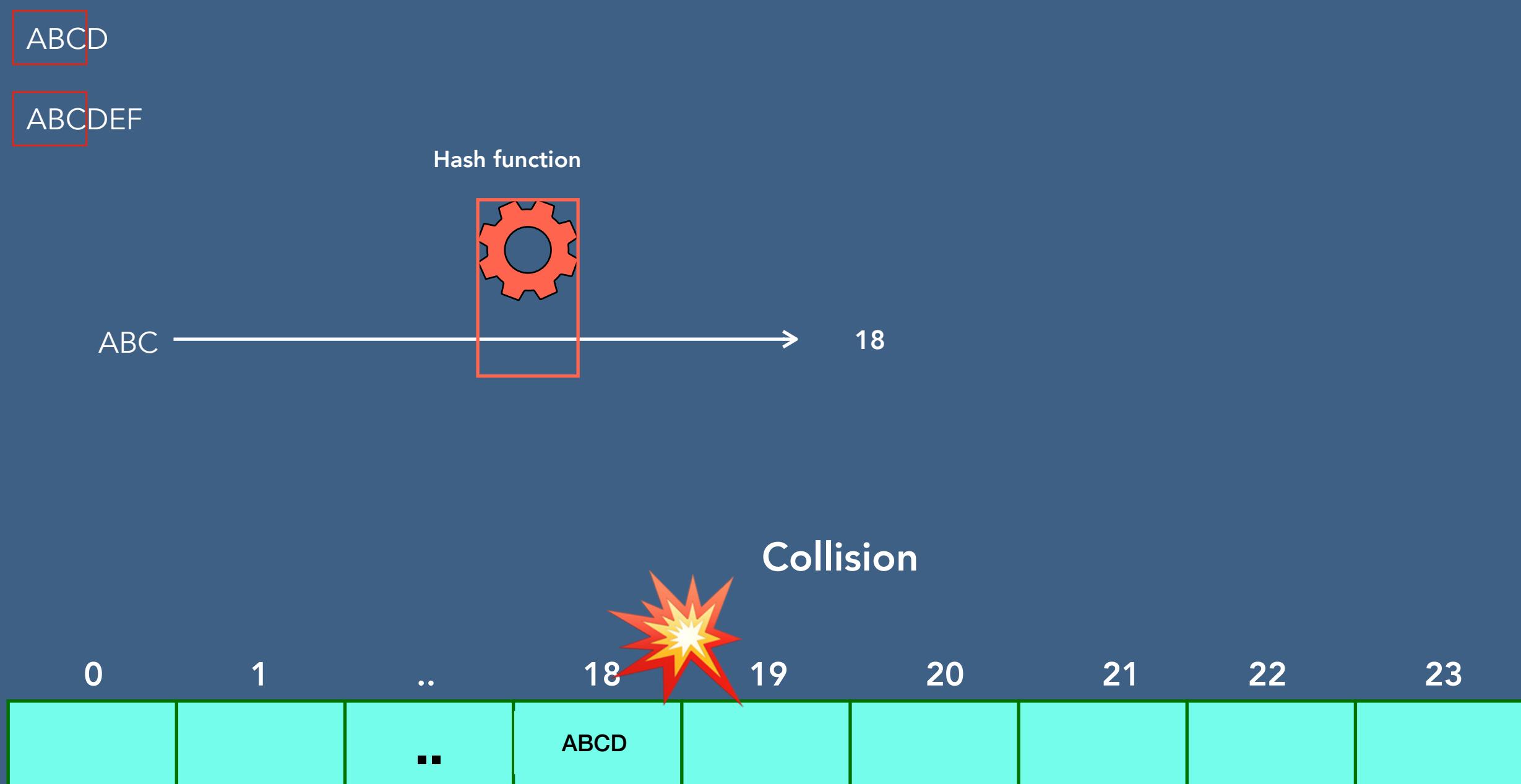
- It distributes hash values uniformly across hash tables



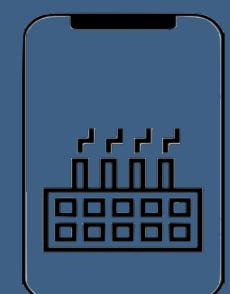
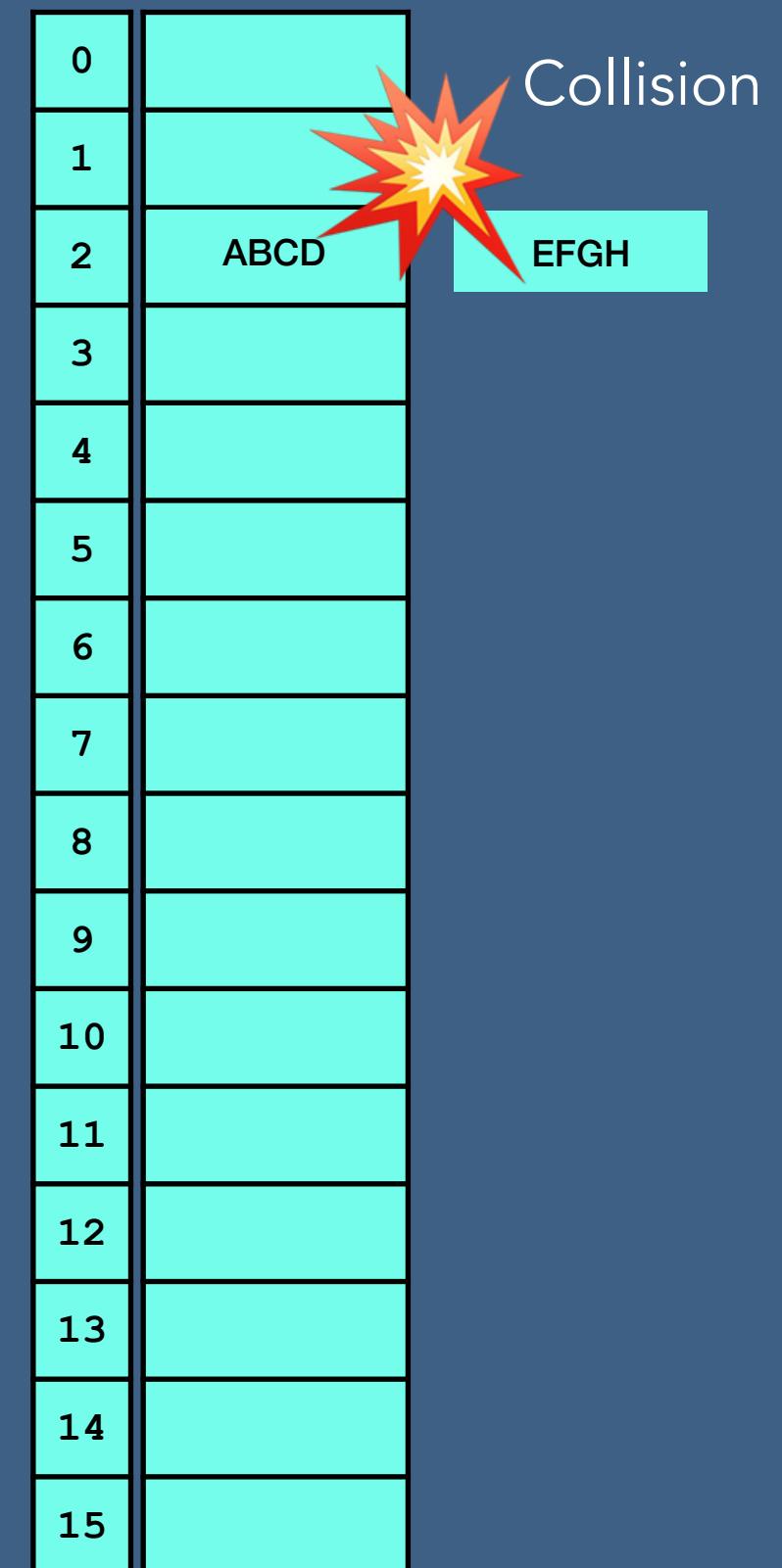
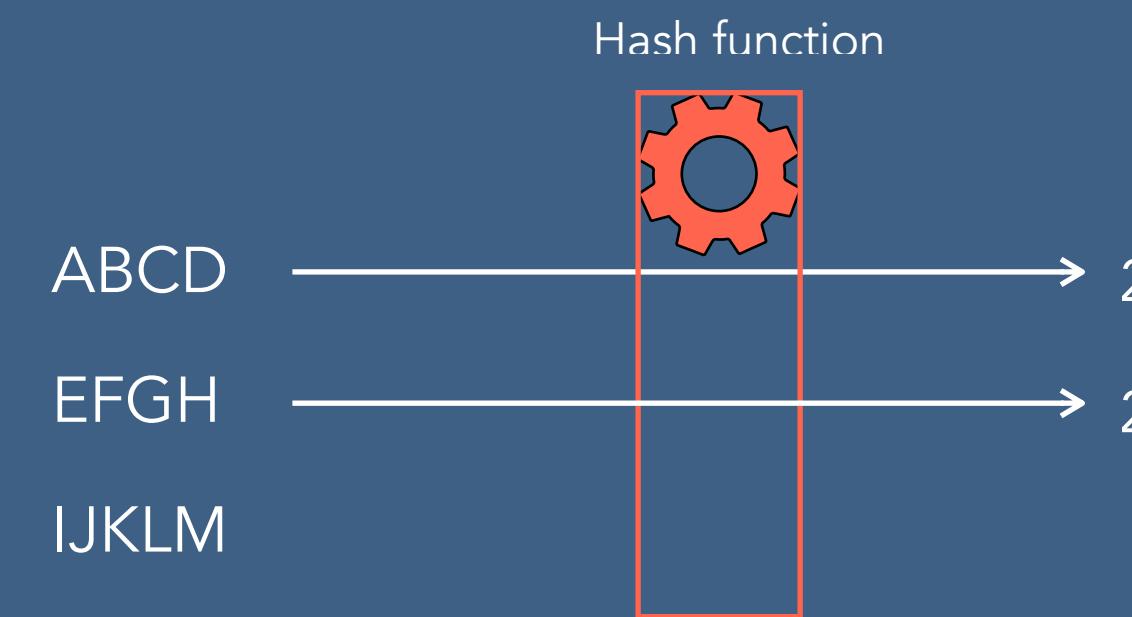
# Hash Functions

## Properties of good Hash function

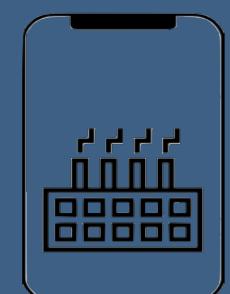
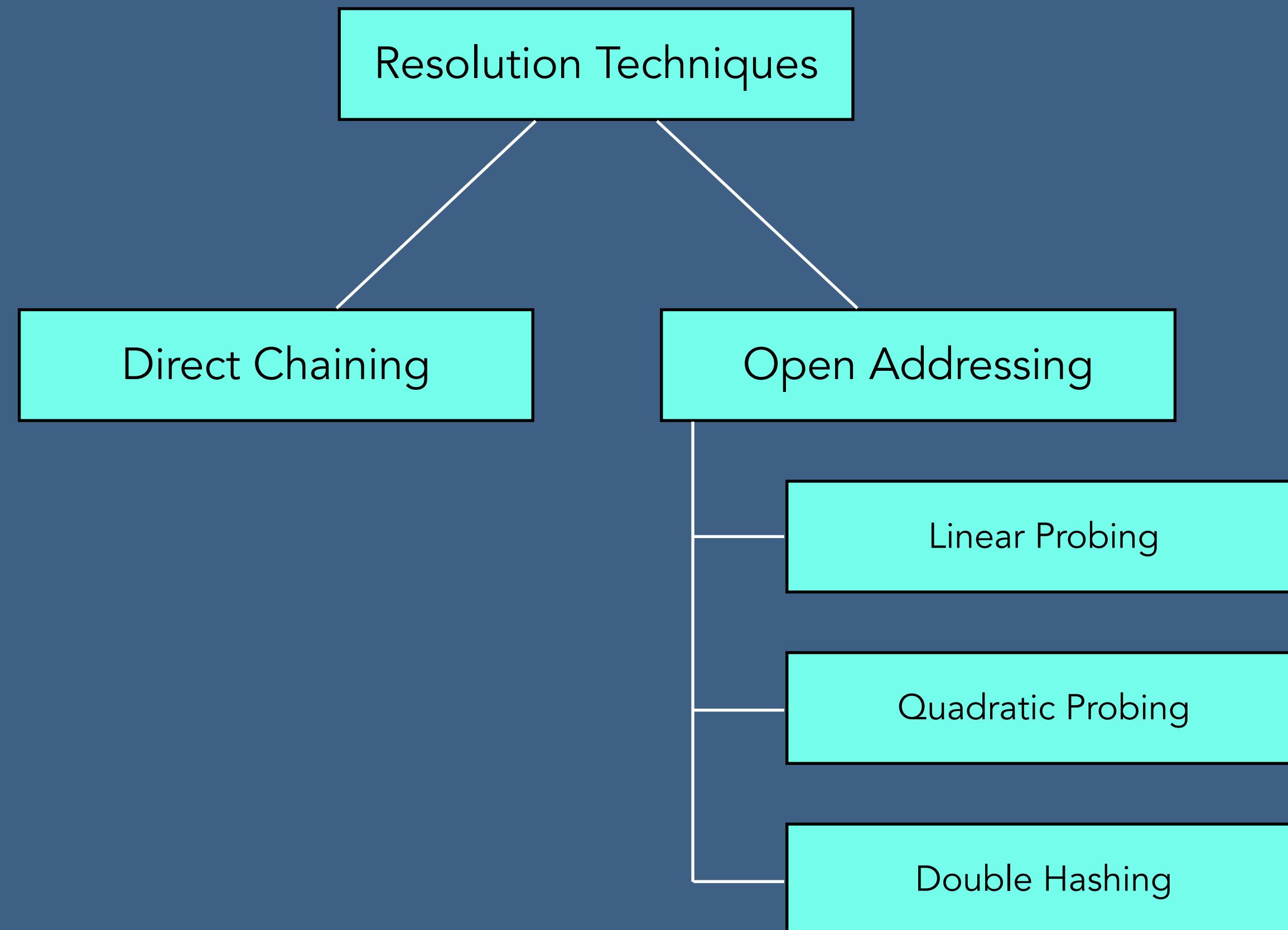
- It distributes hash values uniformly across hash tables
- It has to use all the input data



# Collision Resolution Techniques

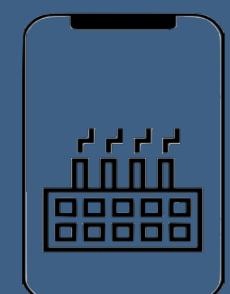
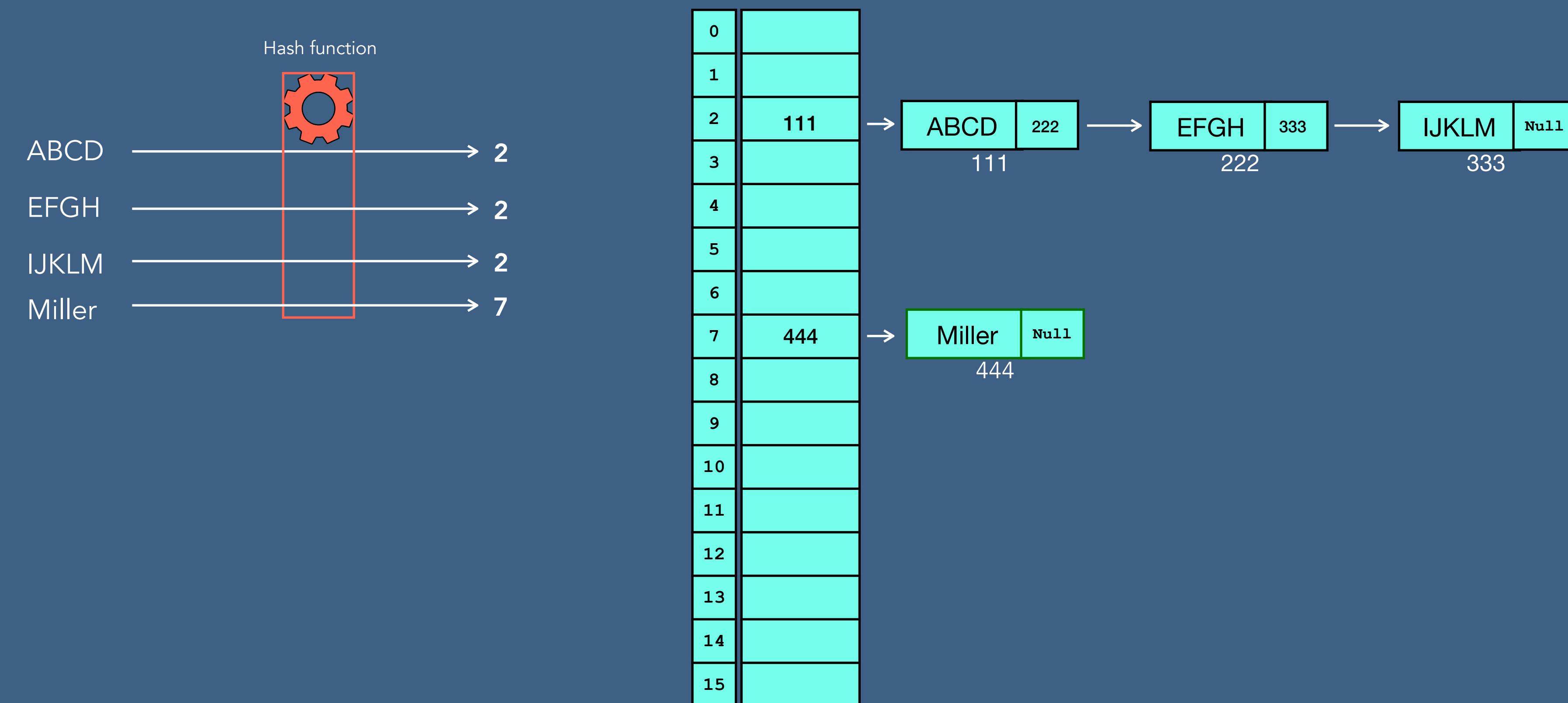


# Collision Resolution Techniques



# Collision Resolution Techniques

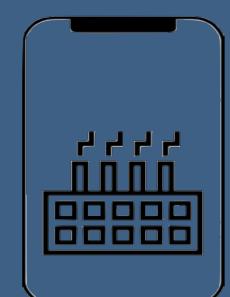
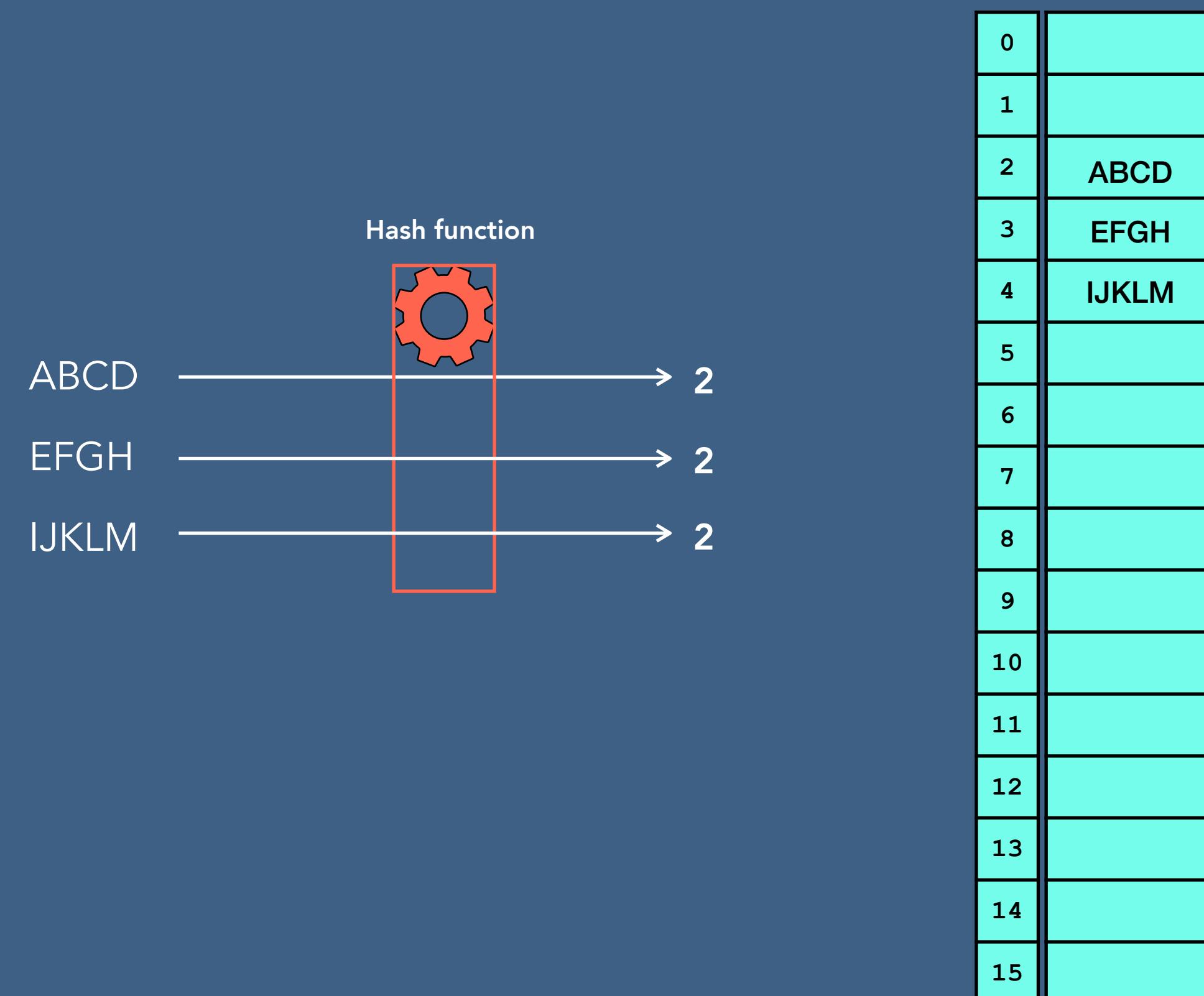
**Direct Chaining :** Implements the buckets as linked list. Colliding elements are stored in this lists



# Collision Resolution Techniques

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

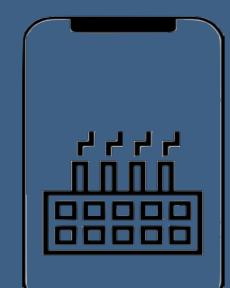
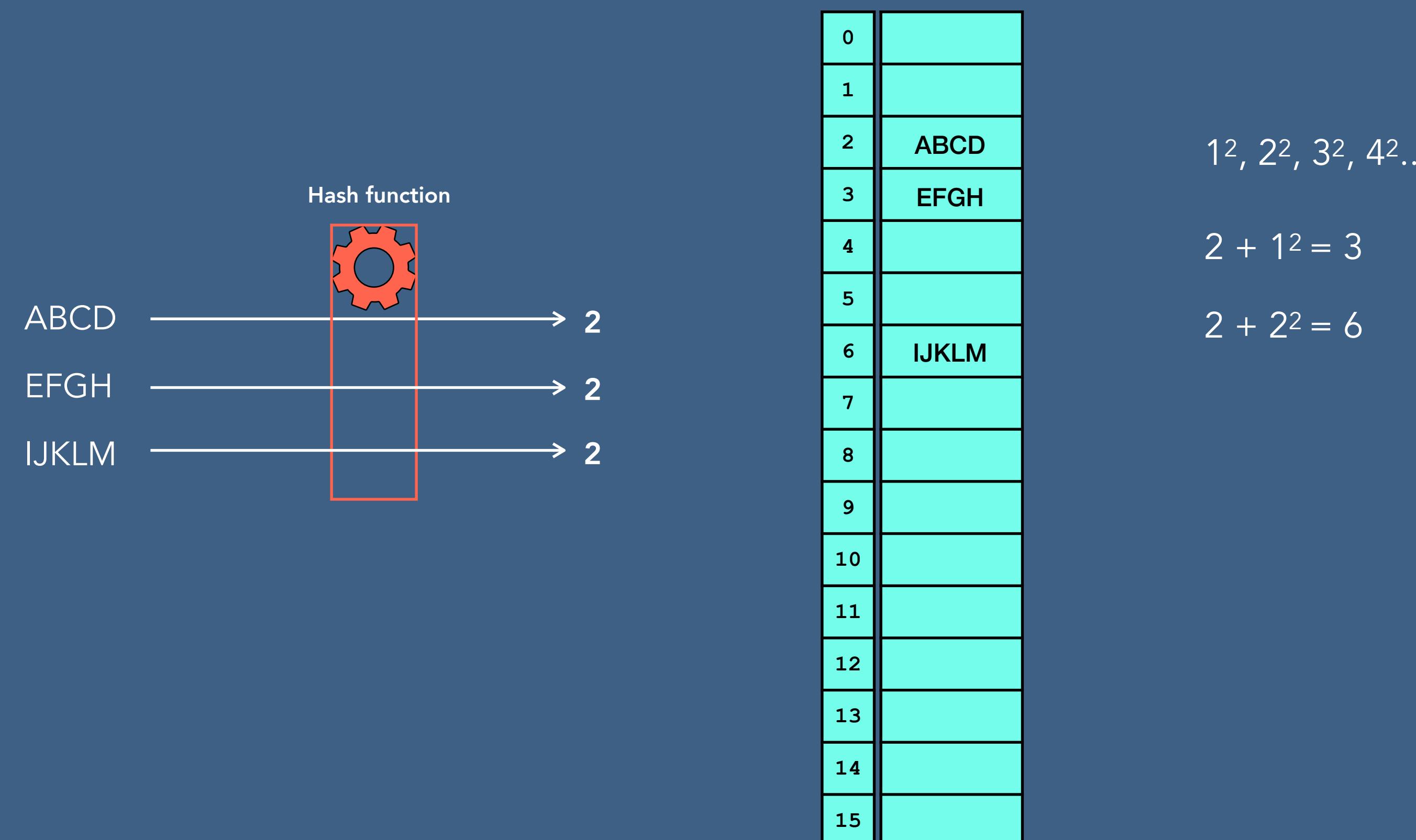
**Linear probing :** It places new key into closest following empty cell



# Collision Resolution Techniques

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

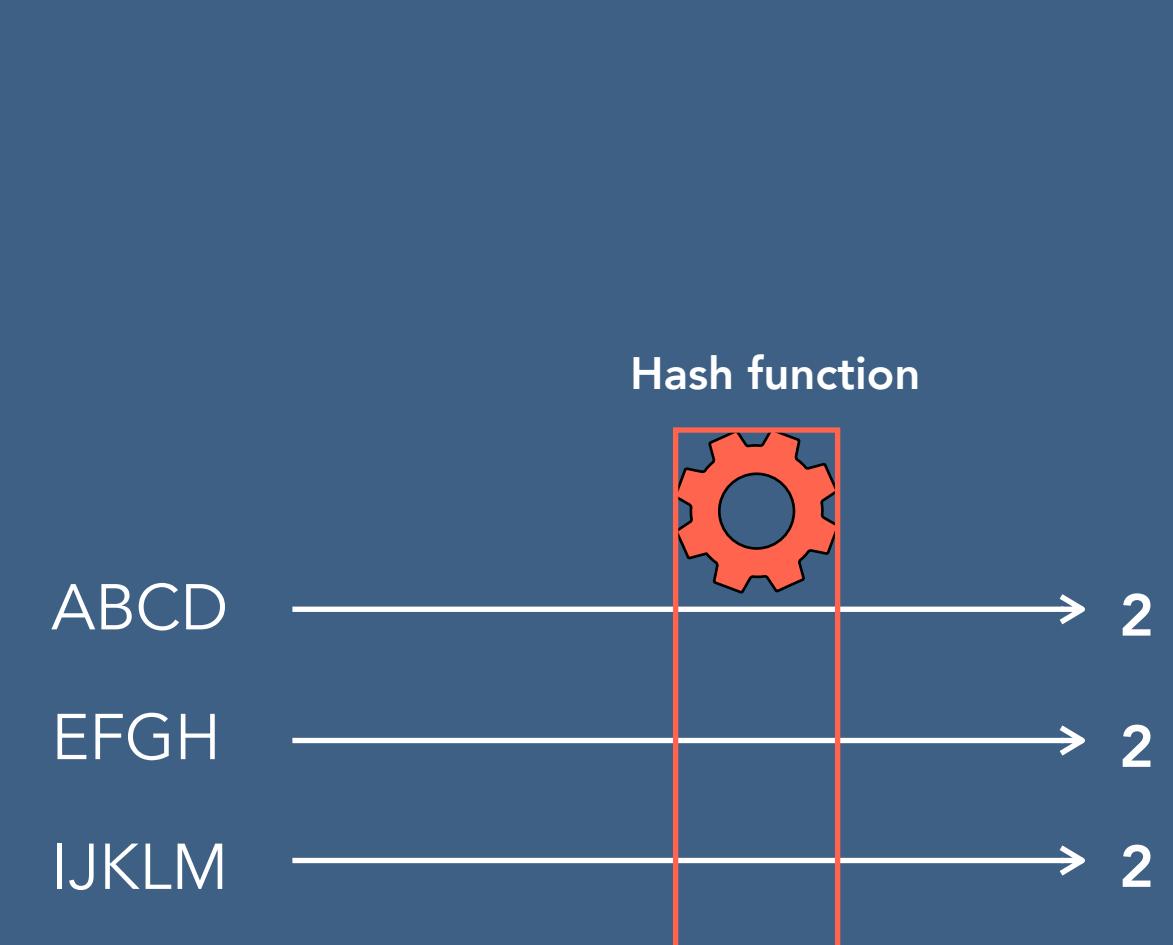
**Quadratic probing :** Adding arbitrary quadratic polynomial to the index until an empty cell is found



# 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

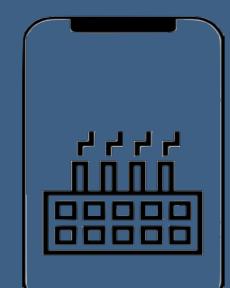
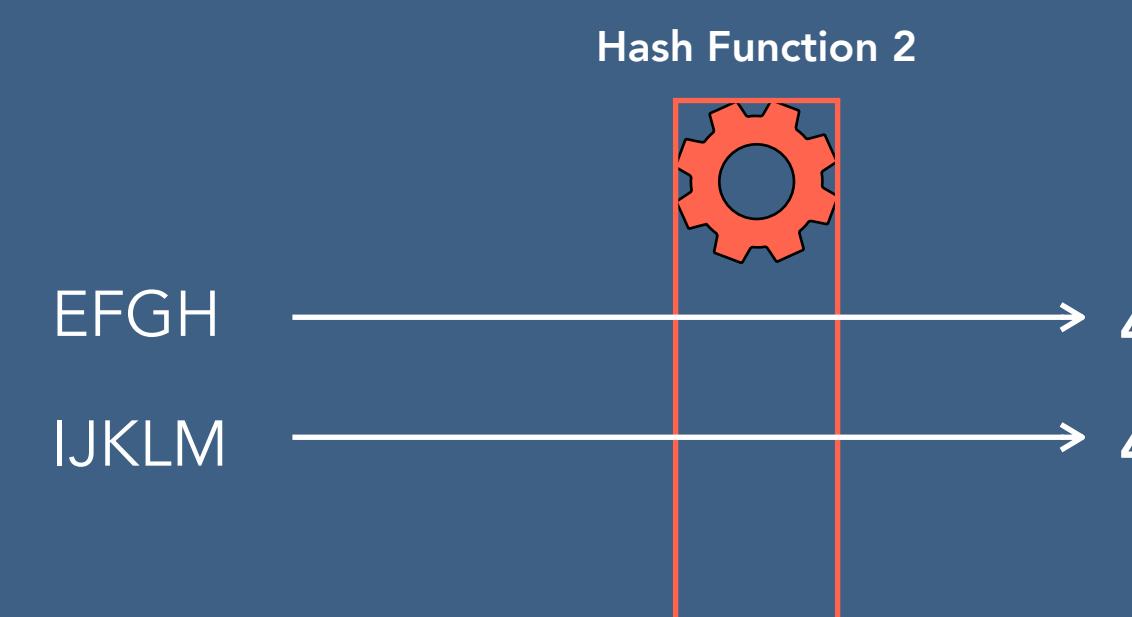


0	
1	
2	ABCD
3	
4	
5	
6	EFGH
7	
8	IJKLM
9	
10	
11	
12	
13	
14	
15	

$$2 + 4 = 6$$

$$2 + 4 = 6$$

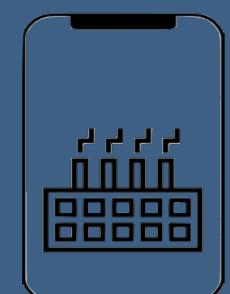
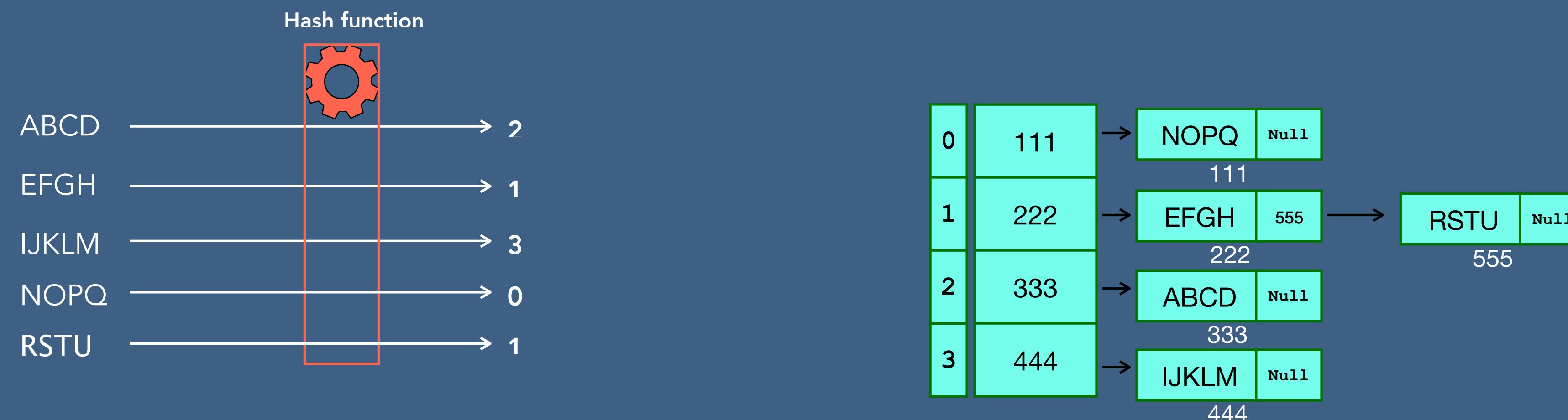
$$2 + (2^4) = 8$$



# Hash Table is Full

## Direct Chaining

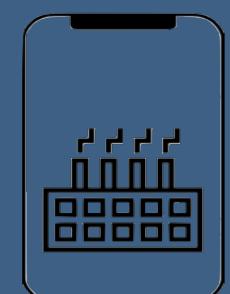
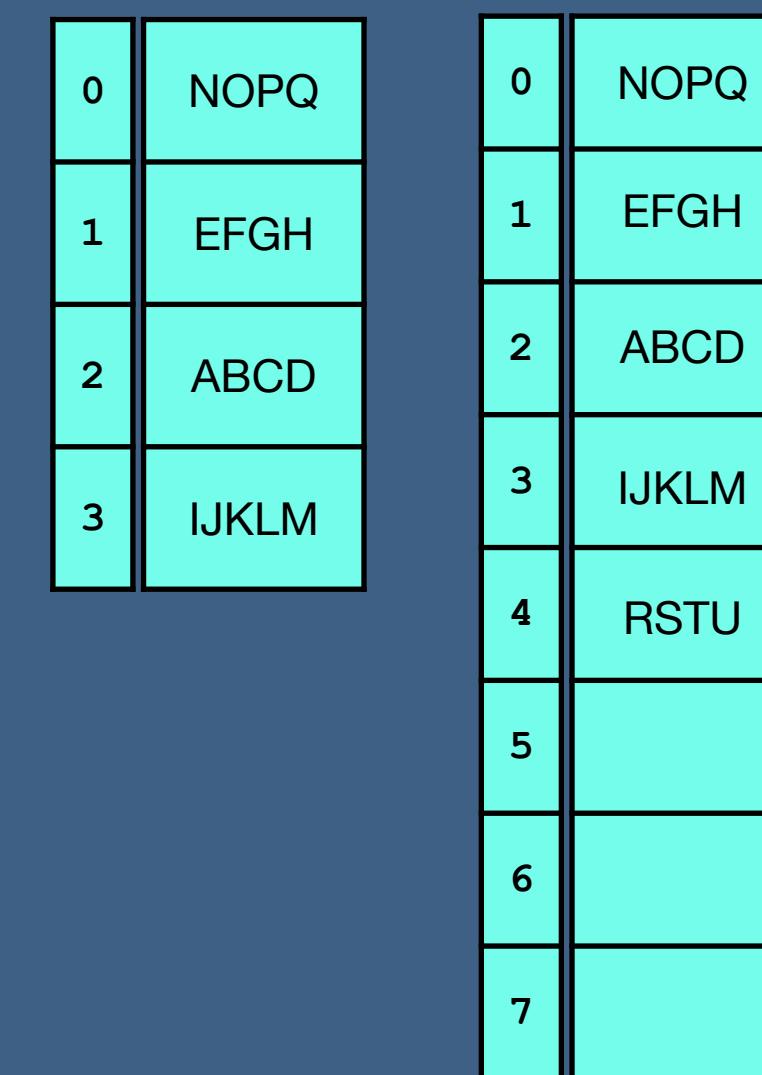
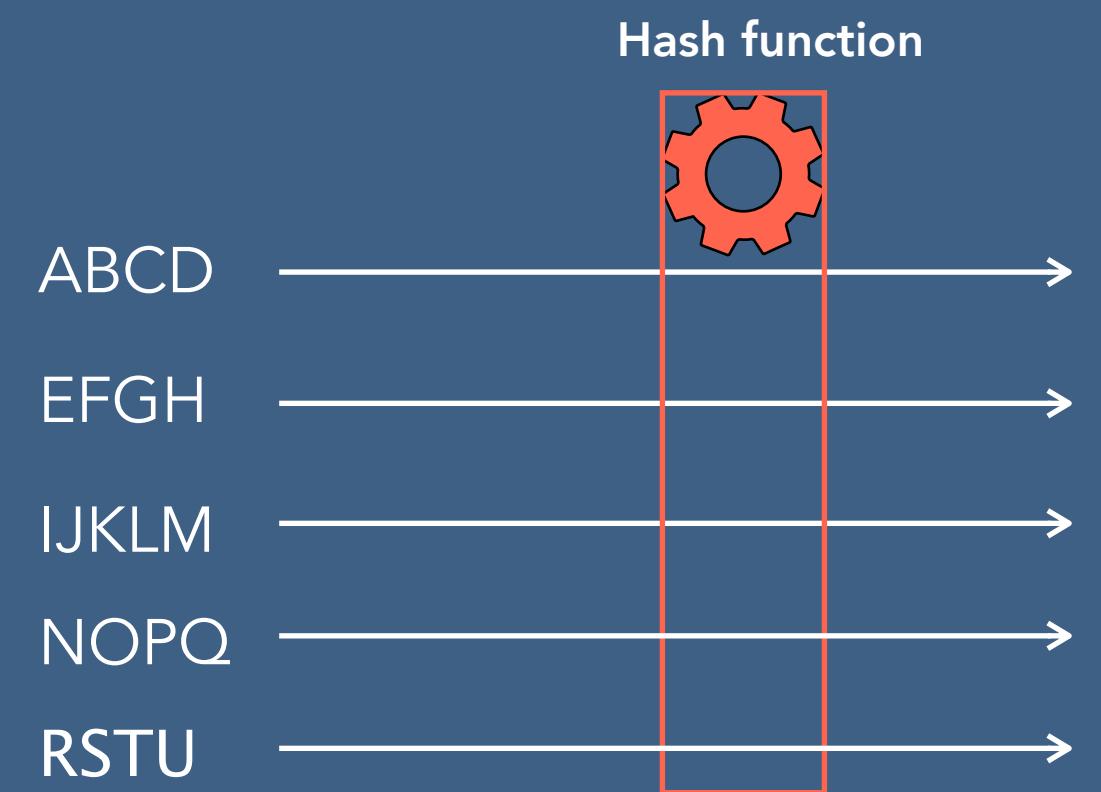
This situation will never arise.



# Hash Table is Full

## Open addressing

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



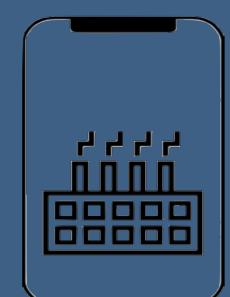
# Pros and Cons of Collision Resolution Techniques

## Direct chaining

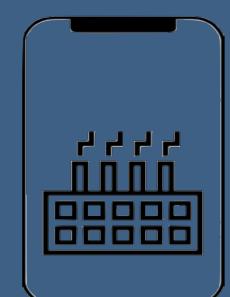
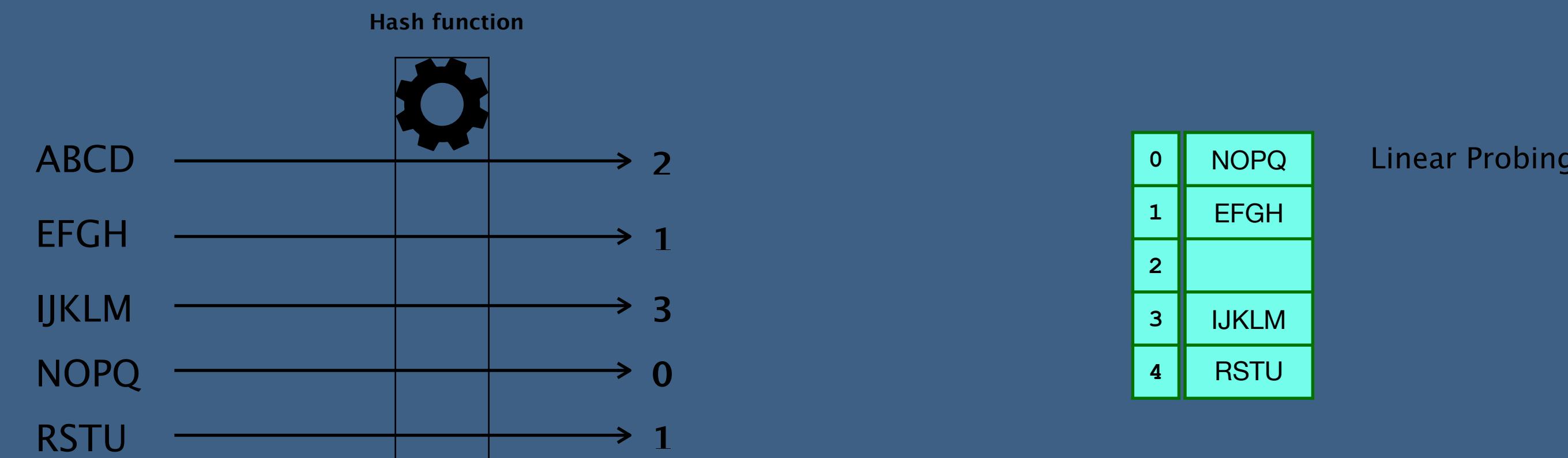
- Hash table never gets full
- Huge Linked List causes performance leaks (Time complexity for search operation becomes  $O(n)$ .)

## 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



# Practical Use of Hashing

## Password verification

Login Required

User:

Password:

Personal Computer



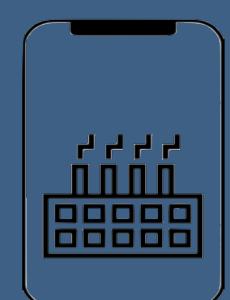
Login : [elshad@google.com](mailto:elshad@google.com)

Password: 123456

Google Servers

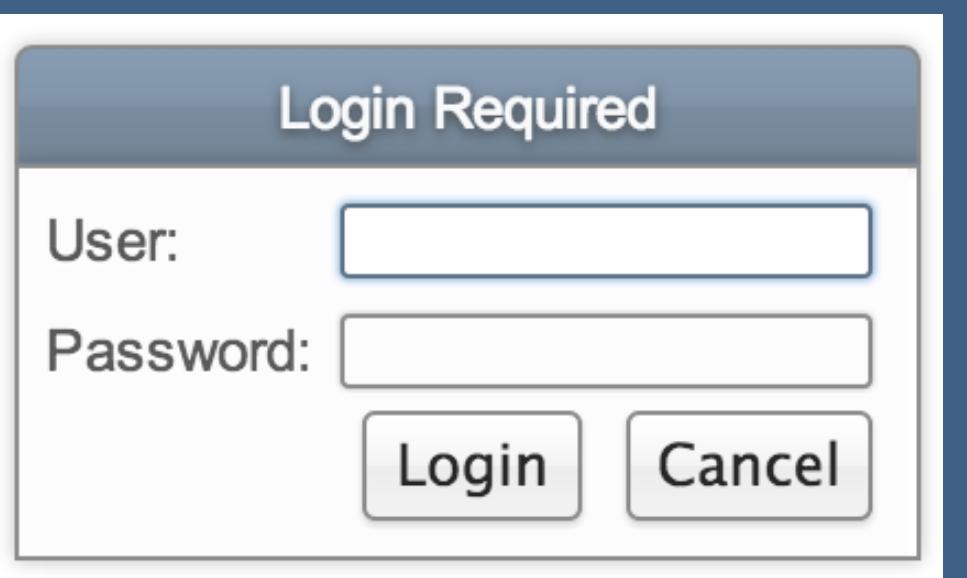


Hash value: \*&71283\*a12

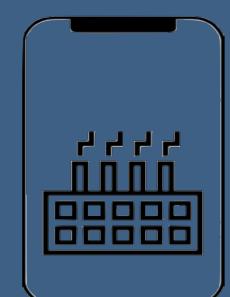
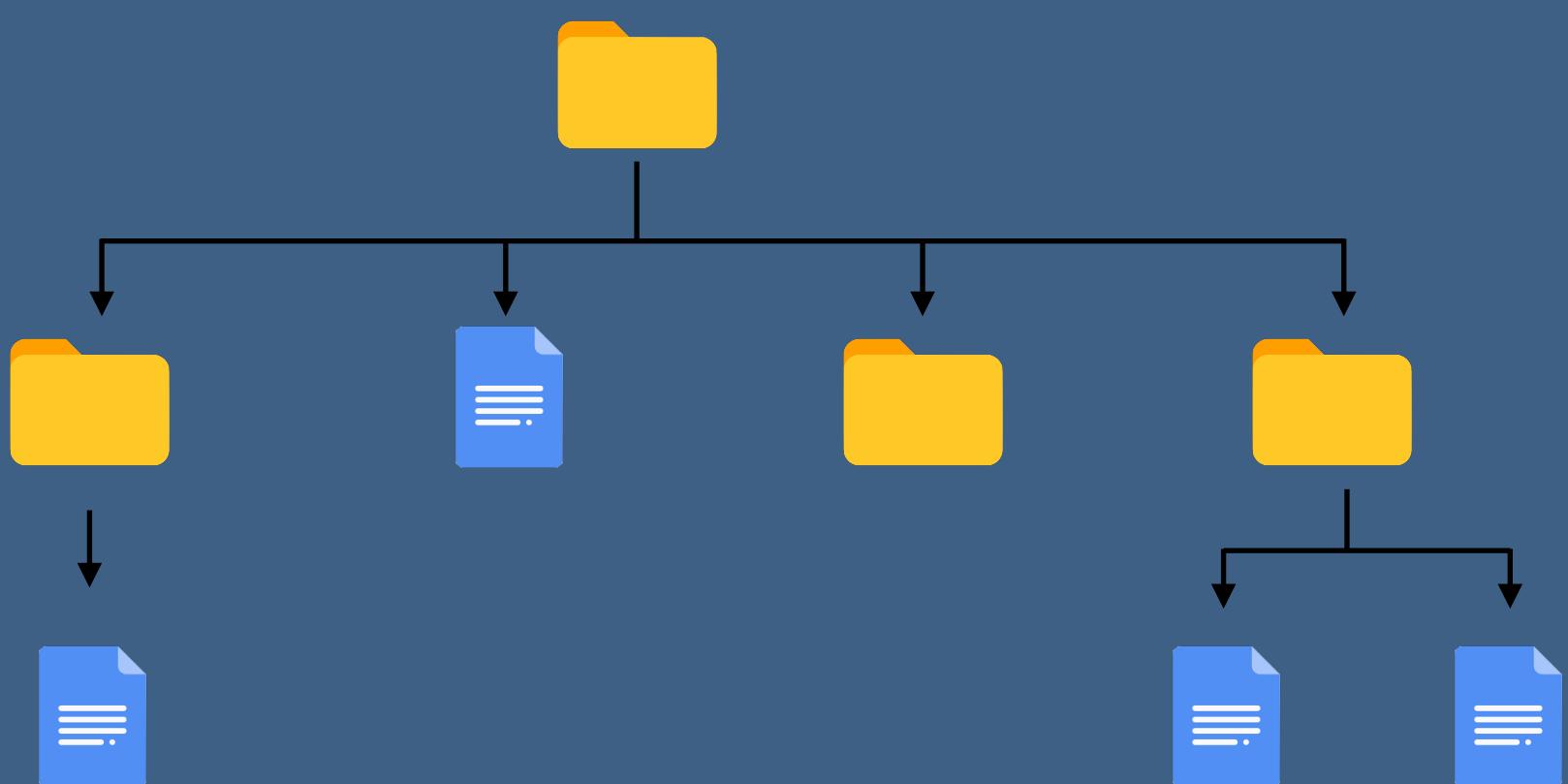


# Practical Use of Hashing

## Password verification

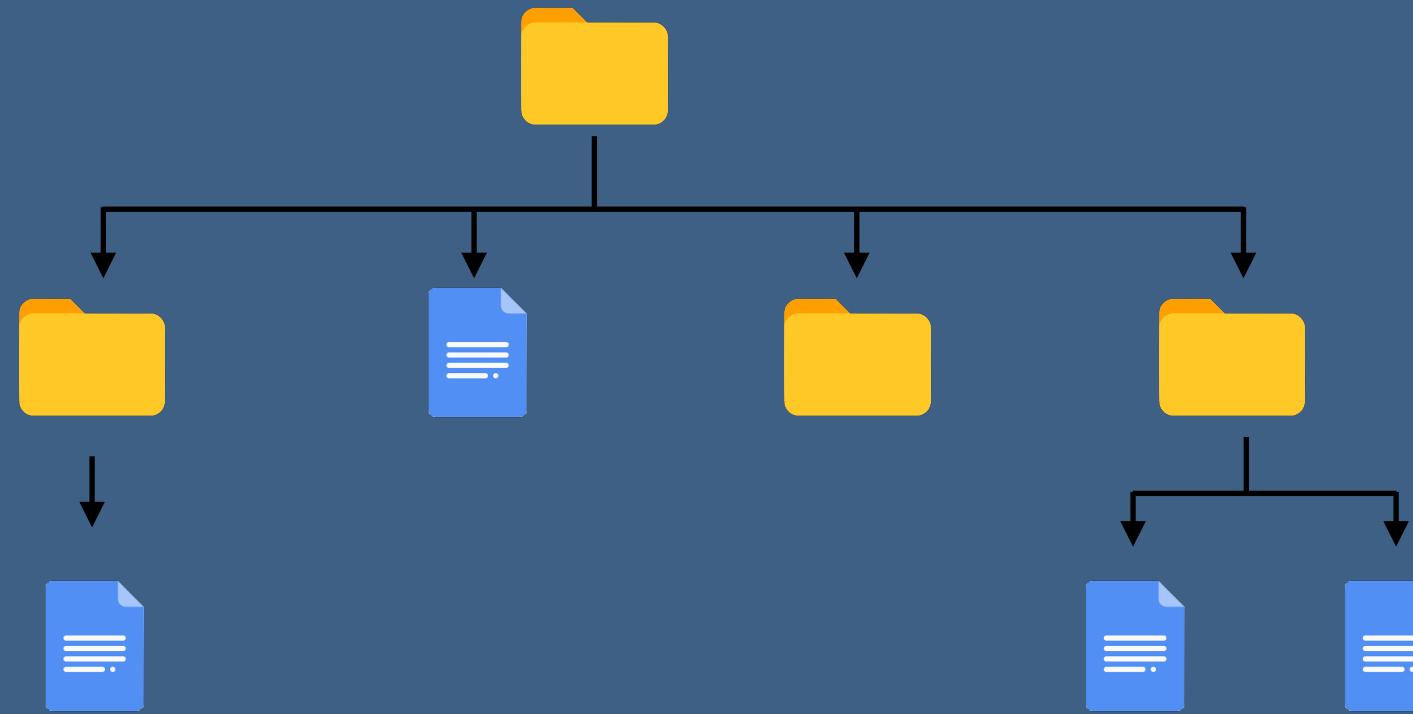


**File system :** File path is mapped to physical location on disk



# Practical Use of Hashing

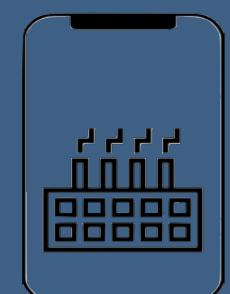
**File system :** File path is mapped to physical location on disk



Path: /Documents/Files/hashing.txt

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

Physical location: sector 4

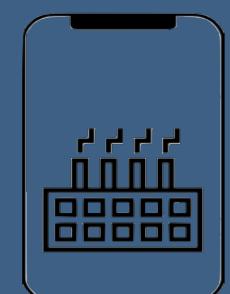


# Pros and Cons of Hashing

✓ 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	Linked List	Tree	Hashing
Insertion	$O(N)$	$O(N)$	$O(\log N)$	$O(1)/O(N)$
Deletion	$O(N)$	$O(N)$	$O(\log N)$	$O(1)/O(N)$
Search	$O(N)$	$O(N)$	$O(\log N)$	$O(1)/O(N)$

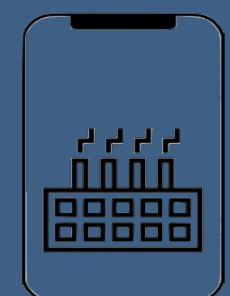


# Sorting Algorithms



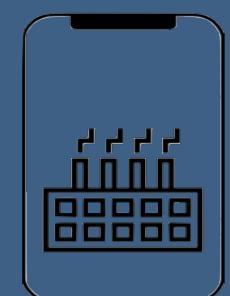
# What is Sorting?

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.



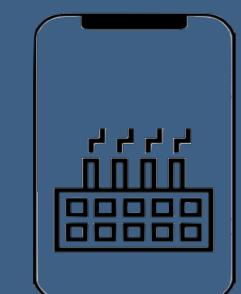
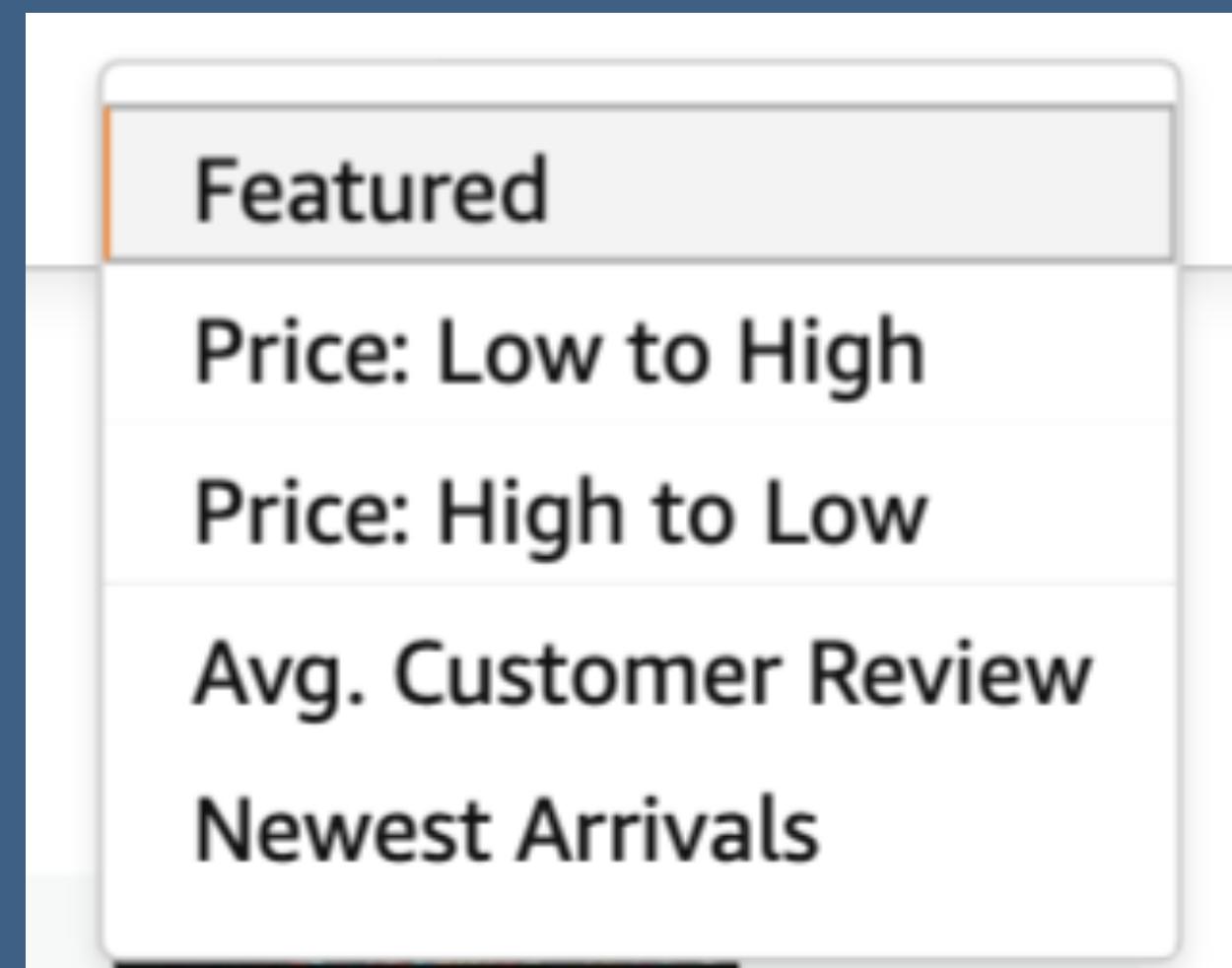
# What is Sorting?

## Practical Use of Sorting

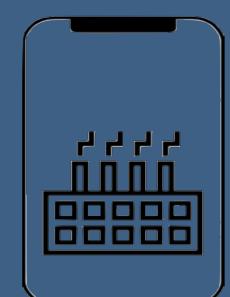
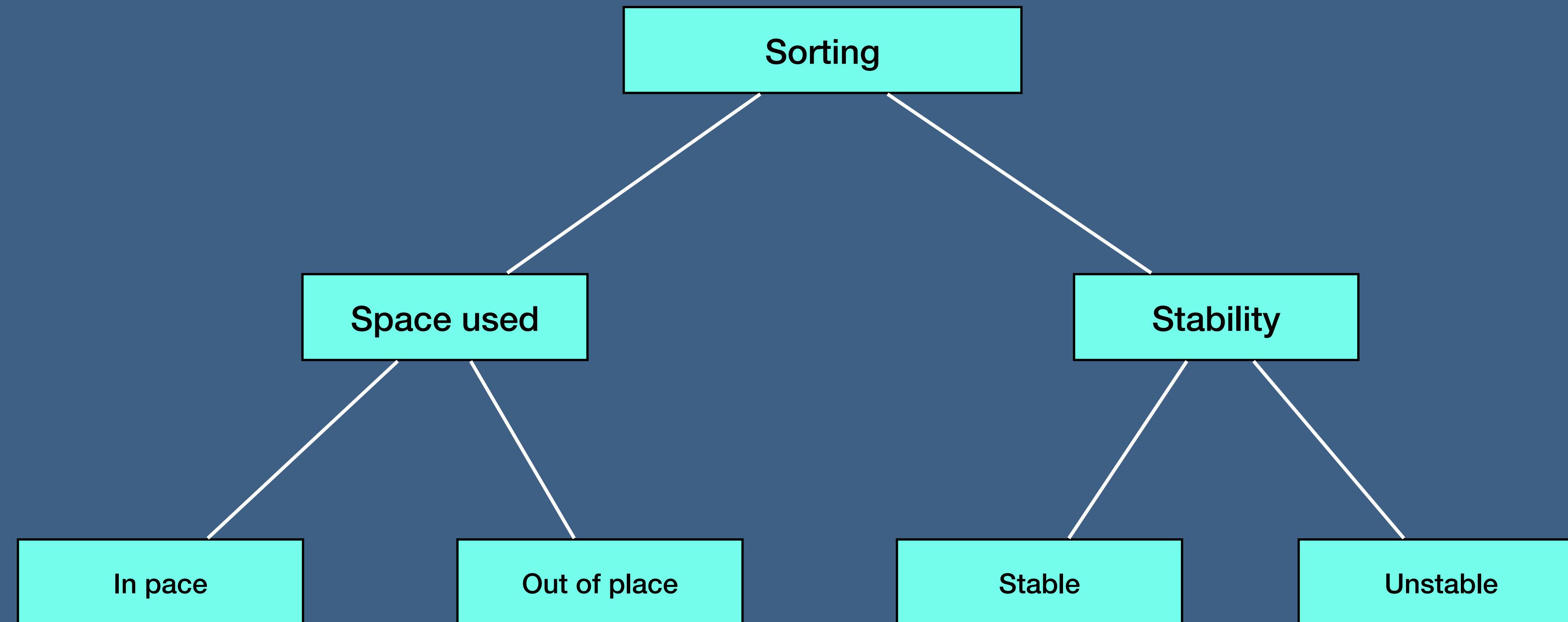
**Microsoft Excel:** Built in functionality to sort data

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

Order Id	Order Date	Employee Name	Reporting Manager	Customer Name	Customer Email	Product Name	Product Price
22	01-Jun-17	Rajkumar Singh	Divya Sharma	Chloe Jones	919-555-2314	beck@email.com	Banana
25	09-Jun-17	Rajkumar Singh	Divya Sharma	Tracey Beckham	919-555-2314	beck@email.com	Banana
23	20-Nov-17						
21	01-Dec-17						
27	12-Jan-17						
26	23-May-17						
28	26-Sep-17						
20	24-Mar-17						
17	30-Jan-17						
24	09-Nov-17						
19	28-Jul-17						
15	16-Sep-17						
6	14-Mar-17						



# Types of Sorting



# 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
----	----	----	----	----	----	----	----	----

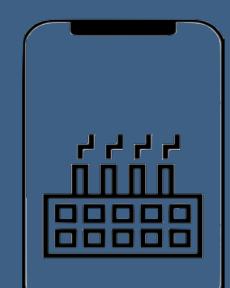
10	20	30	40	50	60	70	80	90
----	----	----	----	----	----	----	----	----

**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
----	----	----	----	----	----	----	----	----

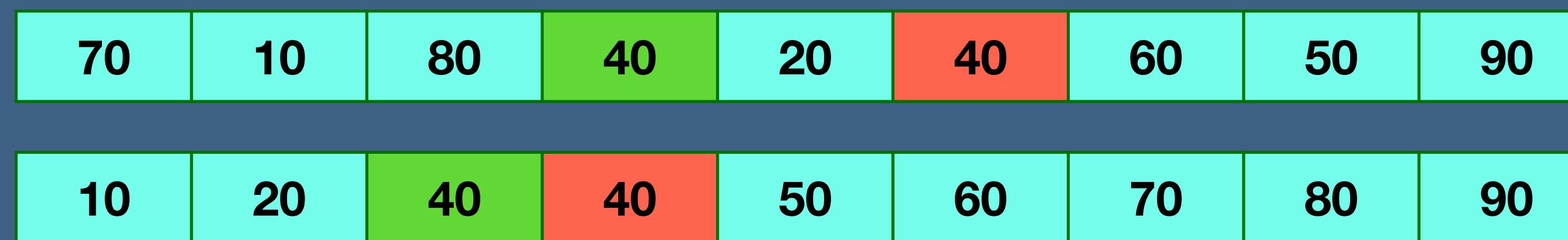


# Types of Sorting

## Stability

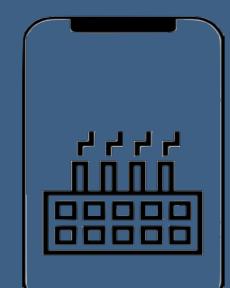
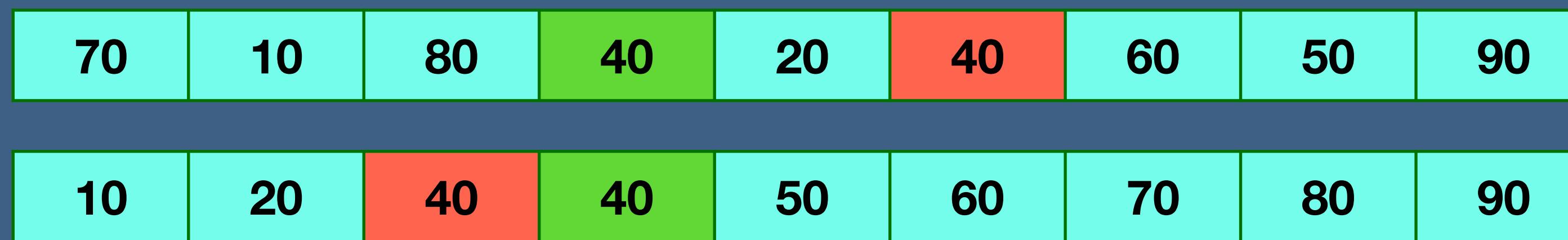
**Stable sorting :** If a sorting algorithm after sorting the contents does not change the sequence of similar content in which they appear, then this sorting is called stable sorting.

**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



# Types of Sorting

## Stability

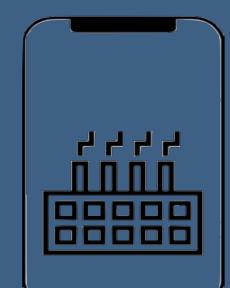
UnStable sorting example

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

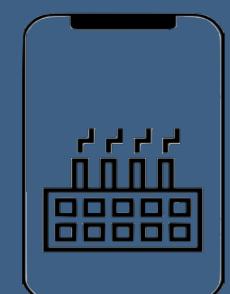
- If successive element is less than the previous one
- Example : 11, 9, 7, 5, 3, 1

## 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**

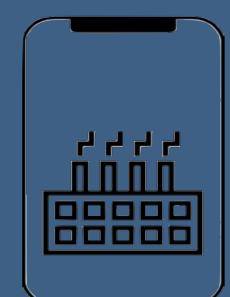
**Merge sort**

**Quick sort**

**Heap sort**

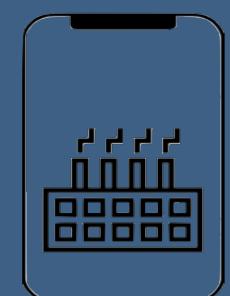
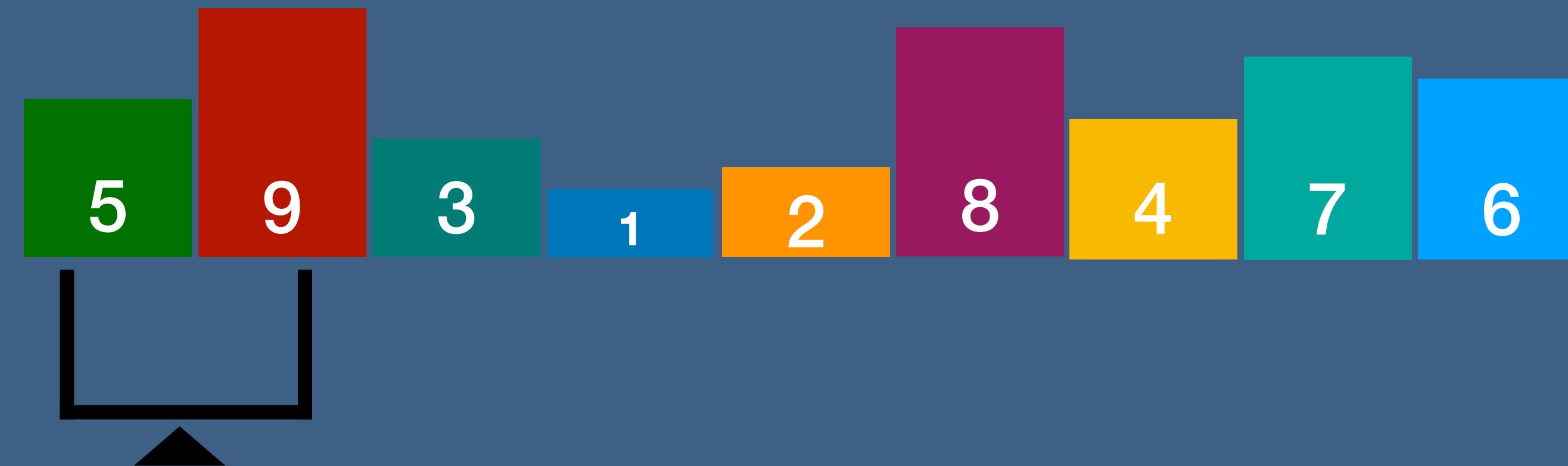
**Which one to select?**

- Stability
- Space efficient
- Time efficient



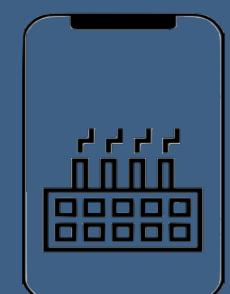
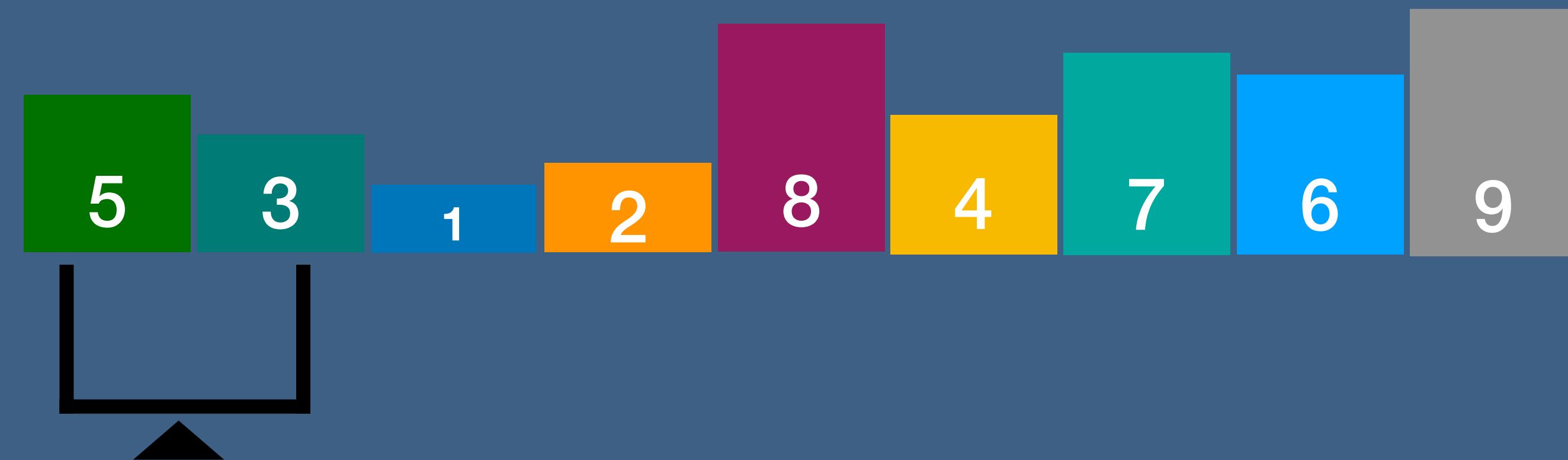
# Bubble Sort

- 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



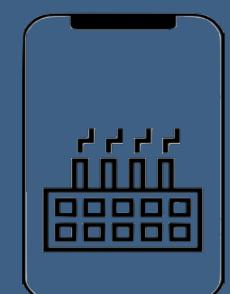
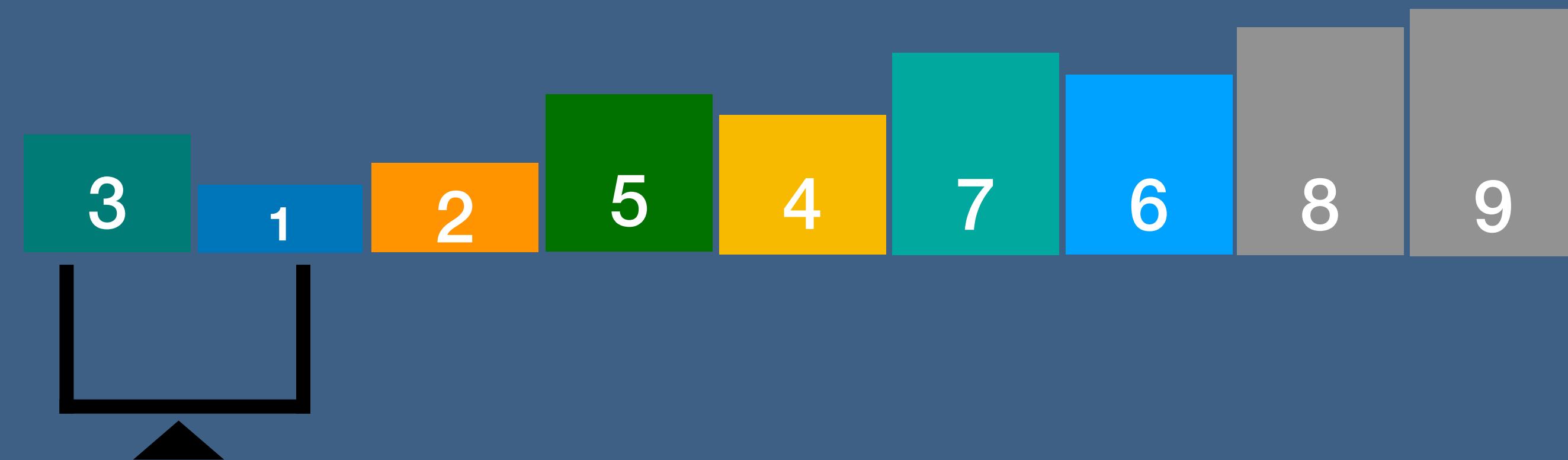
# Bubble Sort

- 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



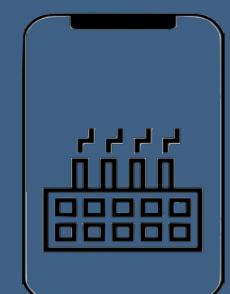
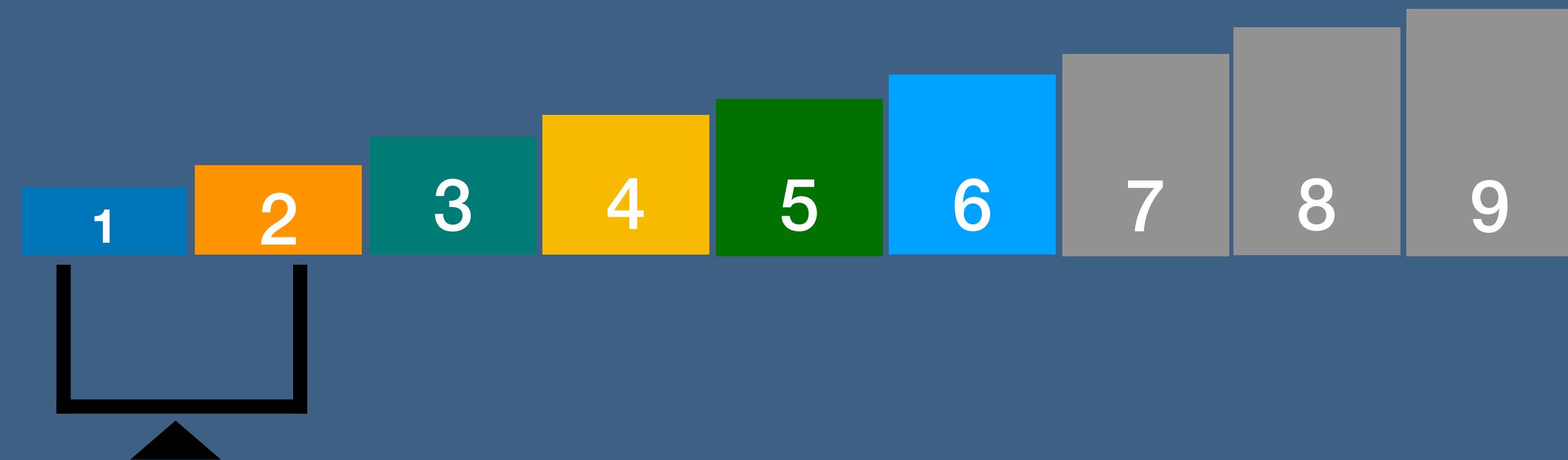
# Bubble Sort

- 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



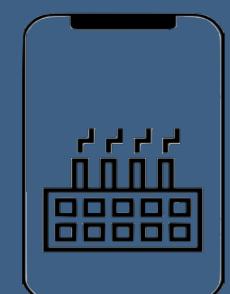
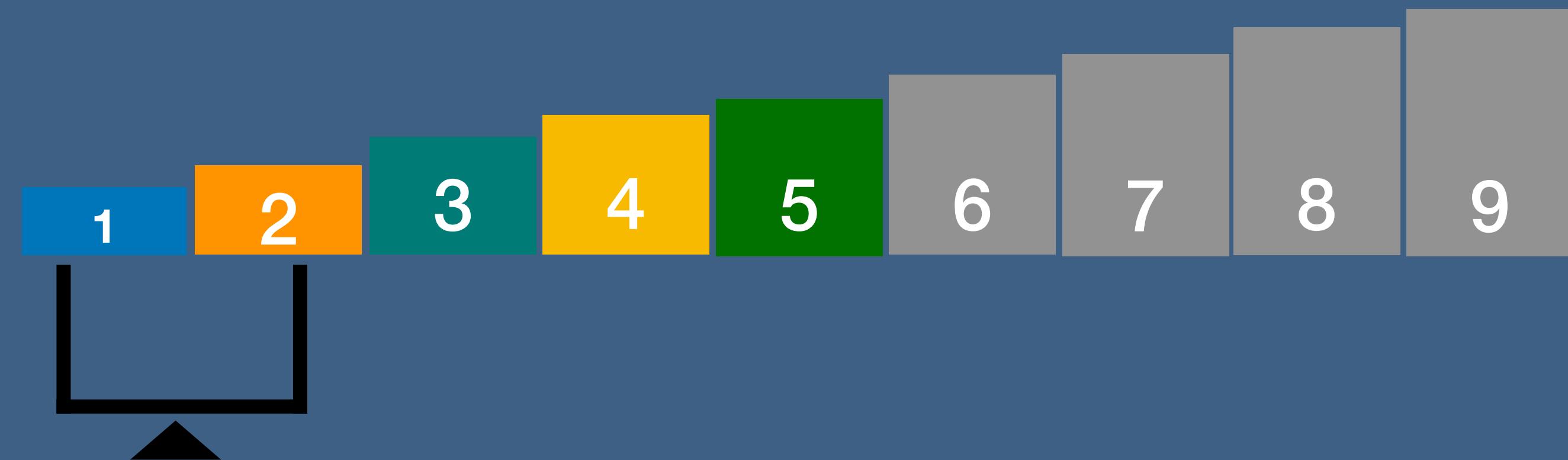
# Bubble Sort

- 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



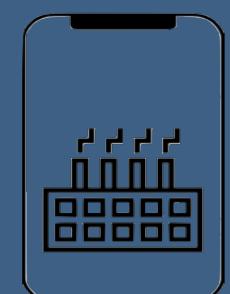
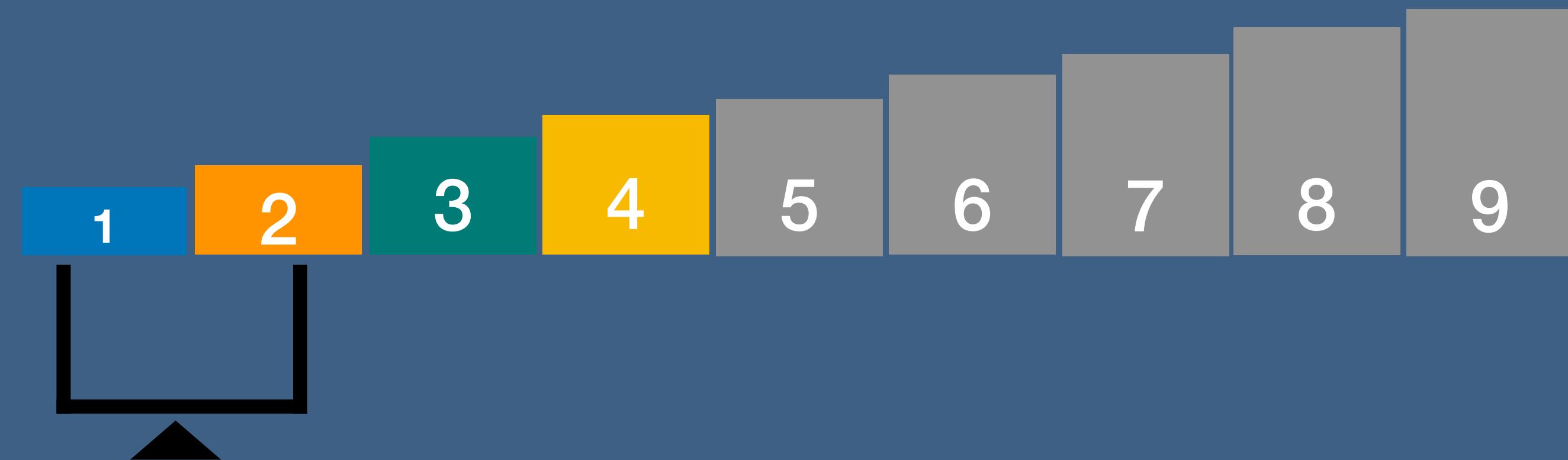
# Bubble Sort

- 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



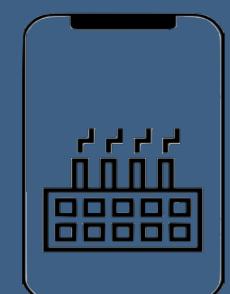
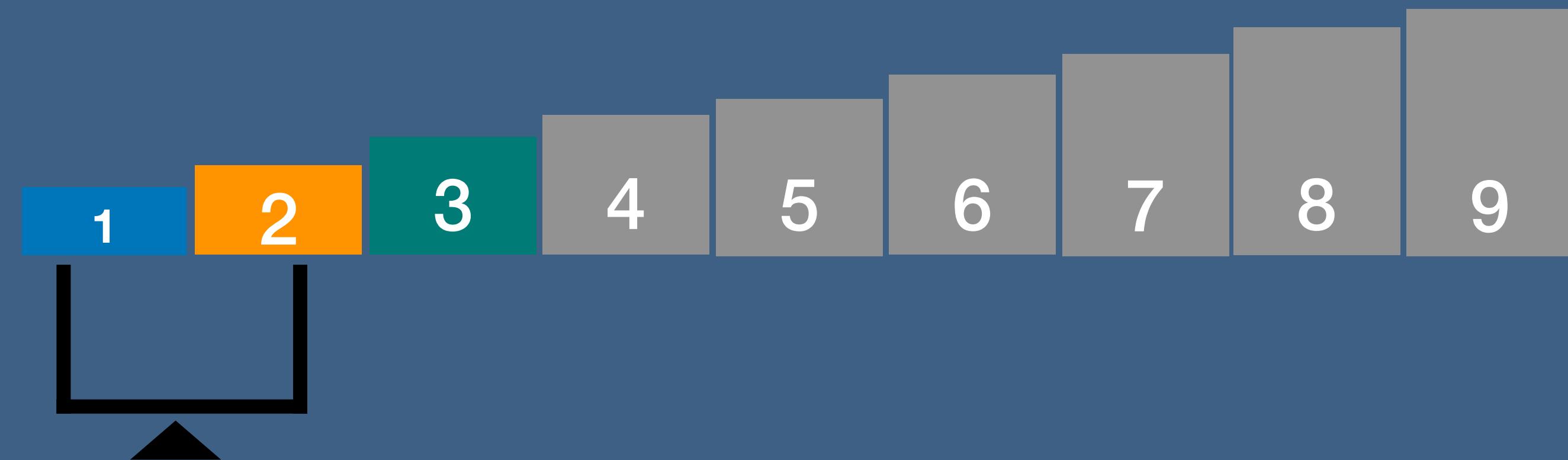
# Bubble Sort

- 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



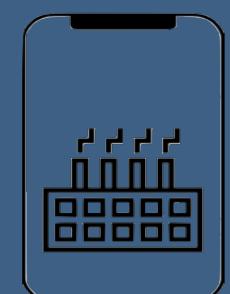
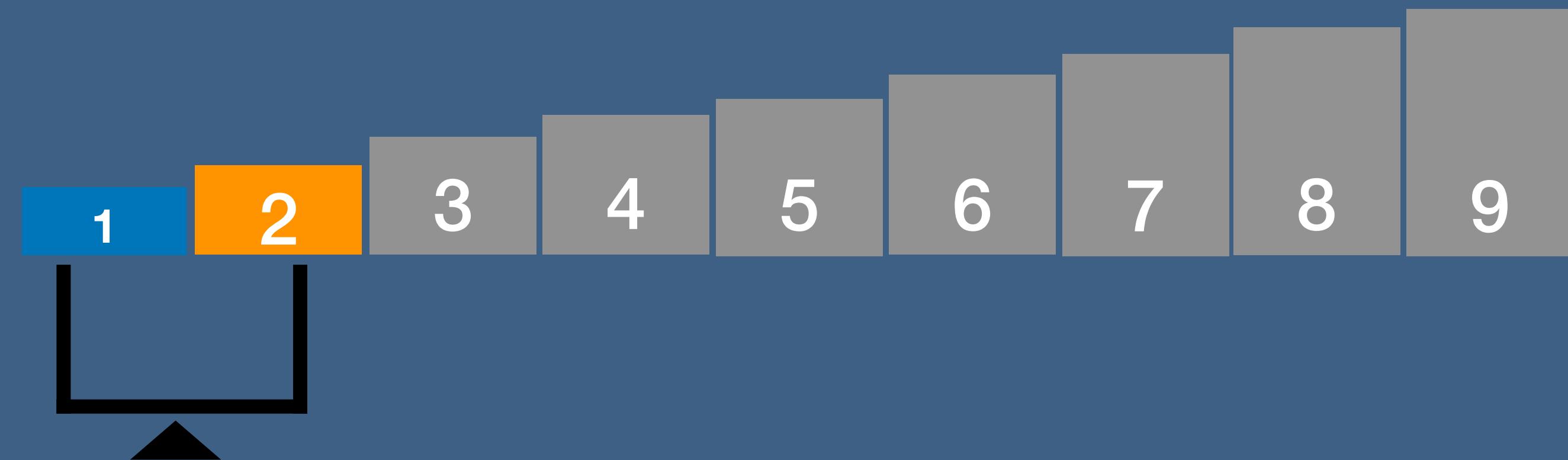
# Bubble Sort

- 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



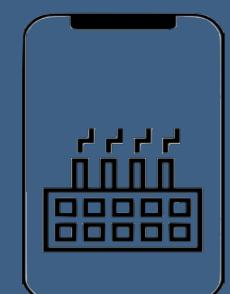
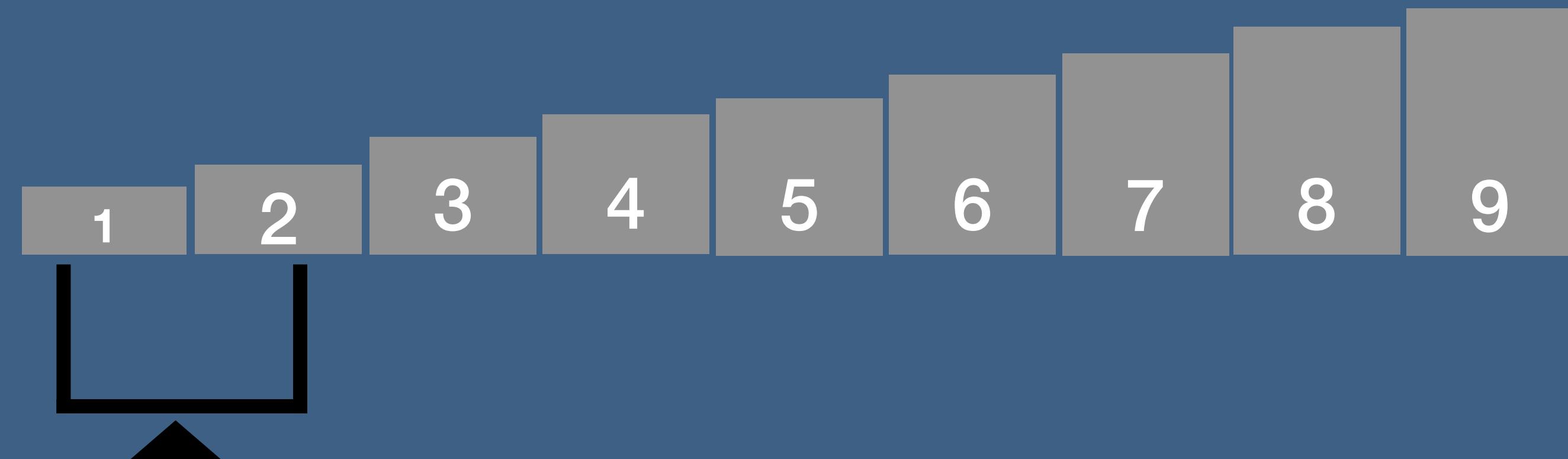
# Bubble Sort

- 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



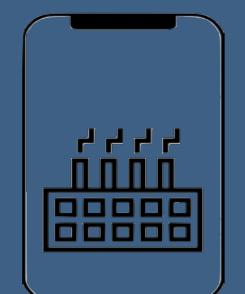
# Bubble Sort

## When to use Bubble Sort?

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

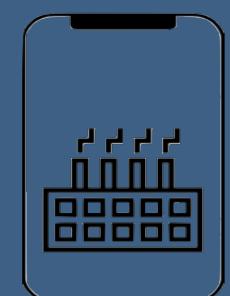
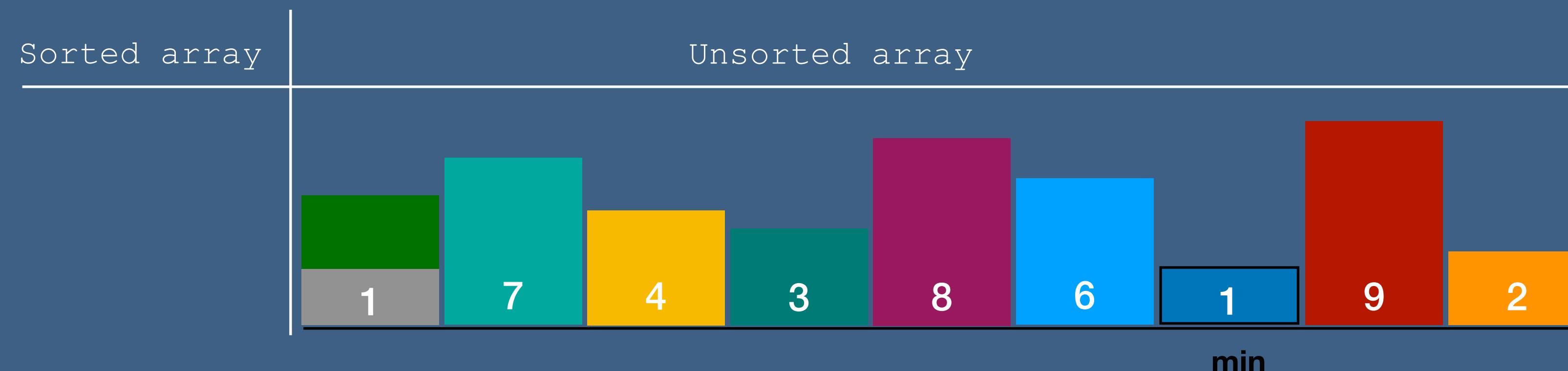
## When to avoid Bubble Sort?

- Average time complexity is poor



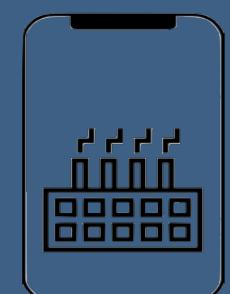
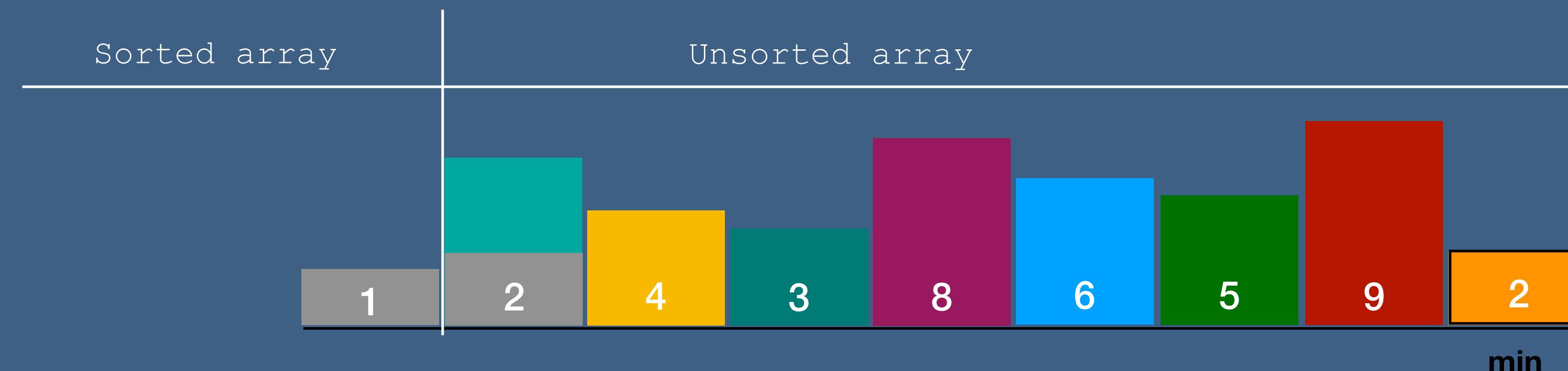
# Selection Sort

- 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.



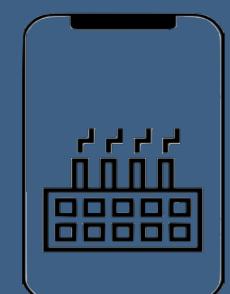
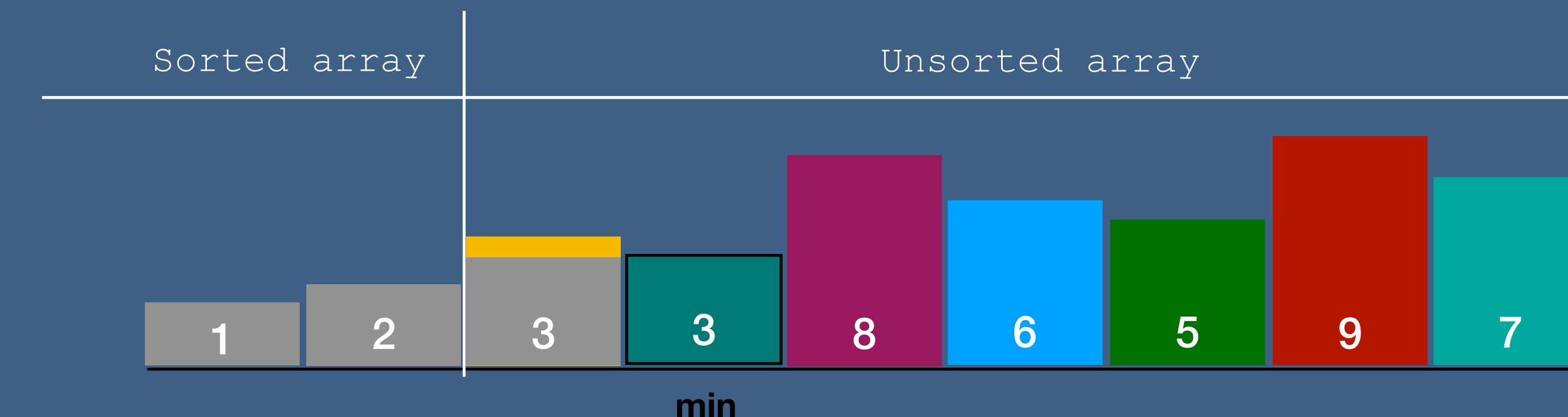
# Selection Sort

- 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.



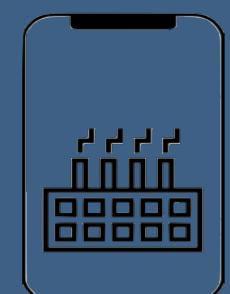
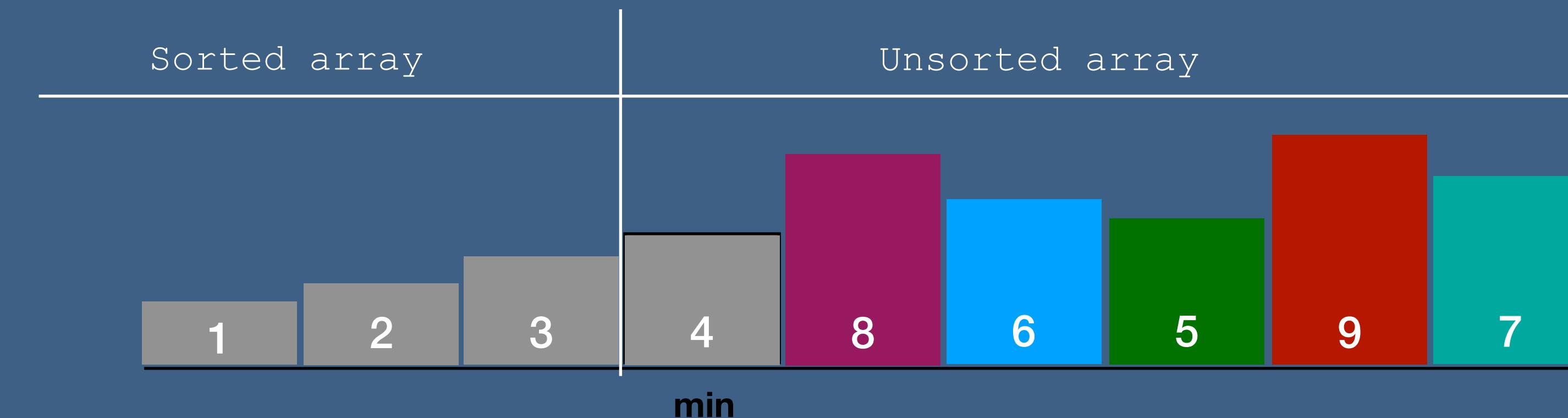
# Selection Sort

- 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.



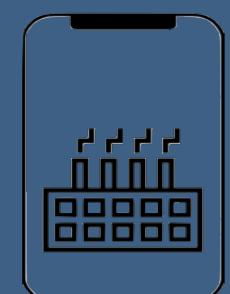
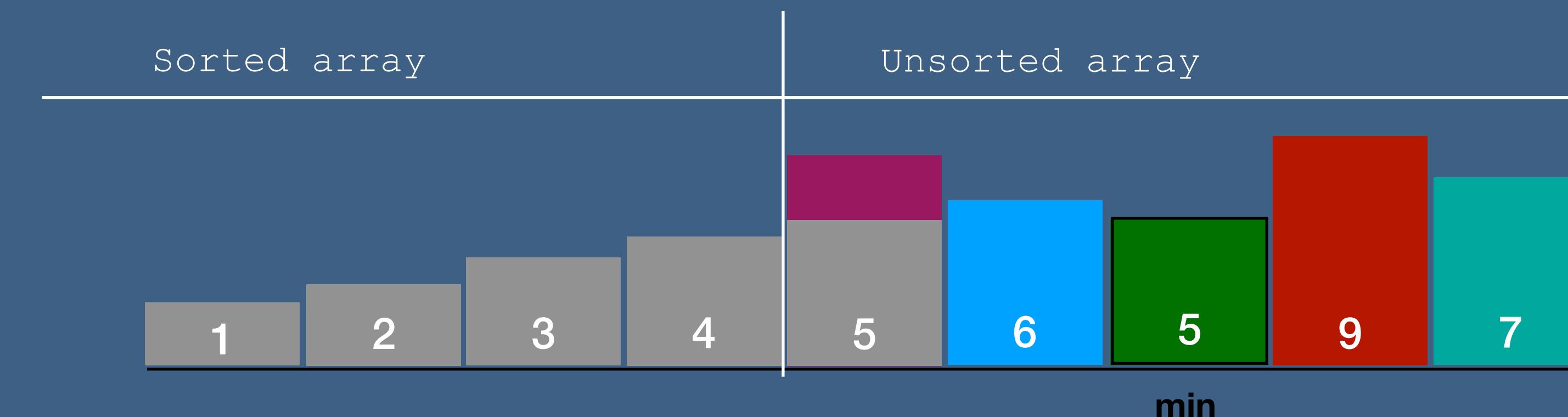
# Selection Sort

- 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.



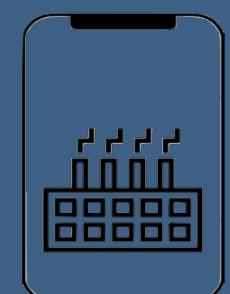
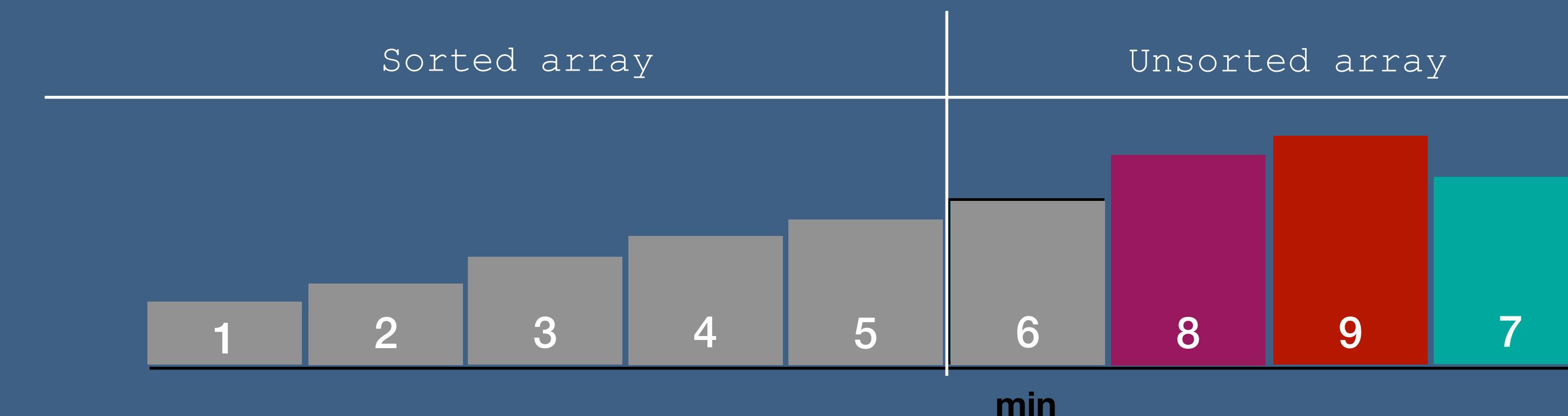
# Selection Sort

- 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.



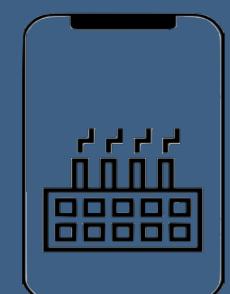
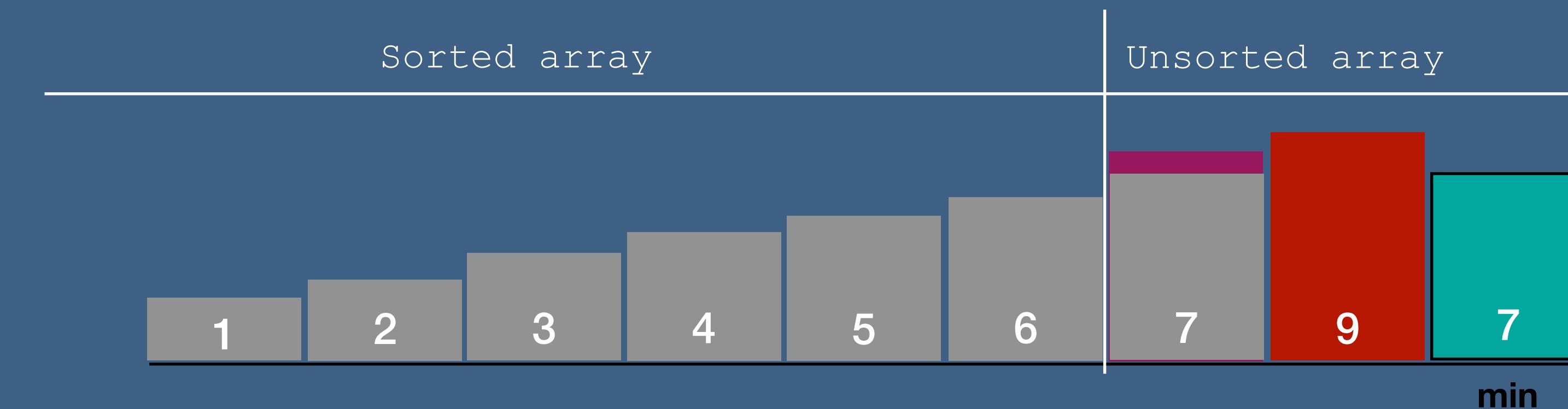
# Selection Sort

- 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.



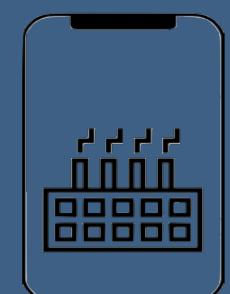
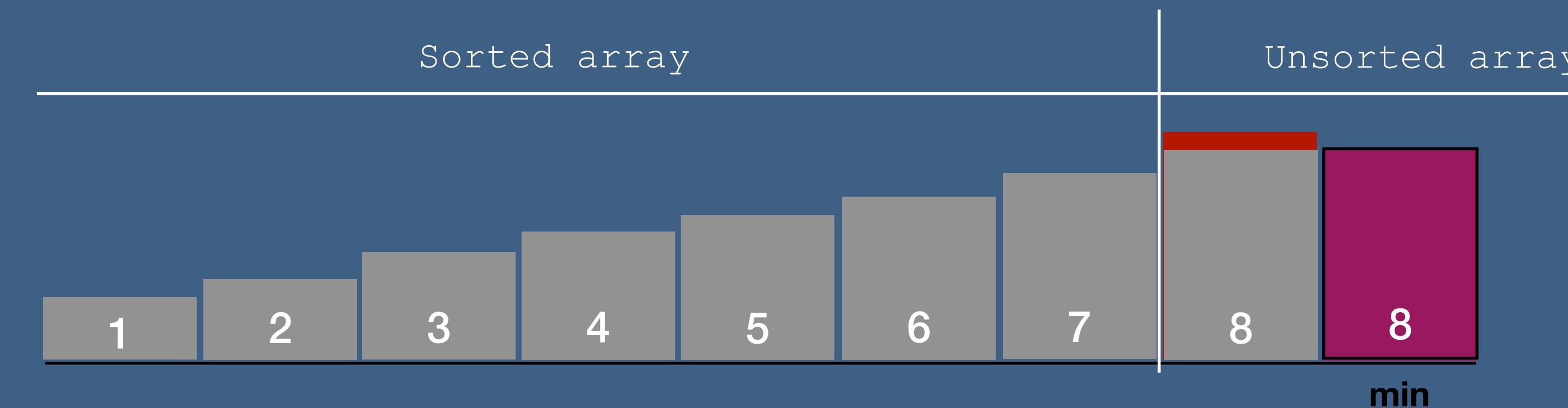
# Selection Sort

- 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.



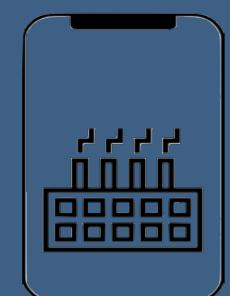
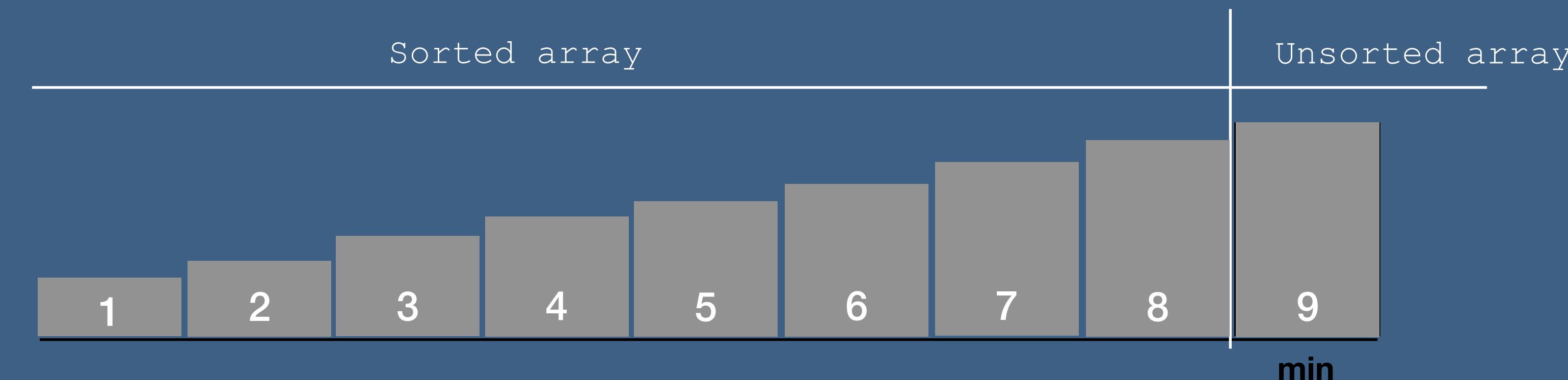
# Selection Sort

- 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.



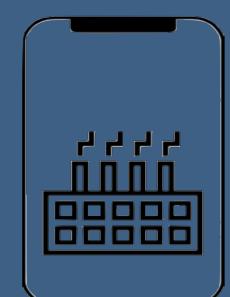
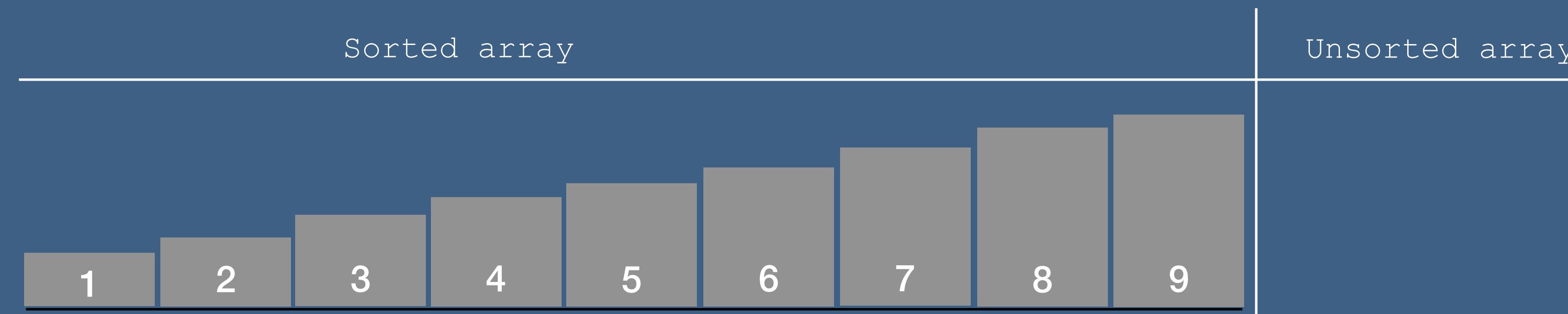
# Selection Sort

- 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.



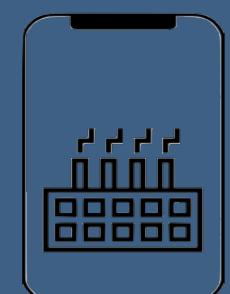
# Selection Sort

## When to use Selection Sort?

- When we have insufficient memory
- Easy to implement

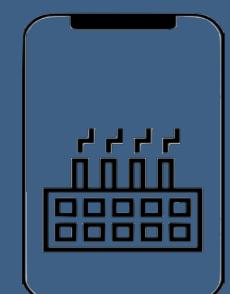
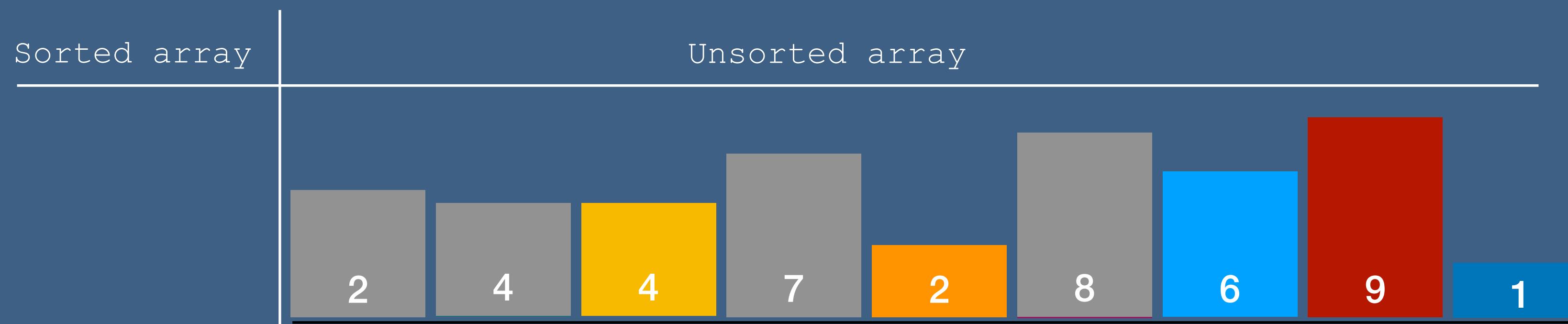
## When to avoid Selection Sort?

- When time is a concern



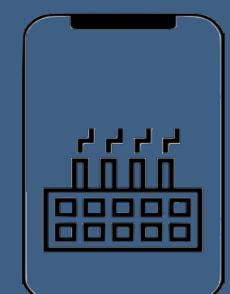
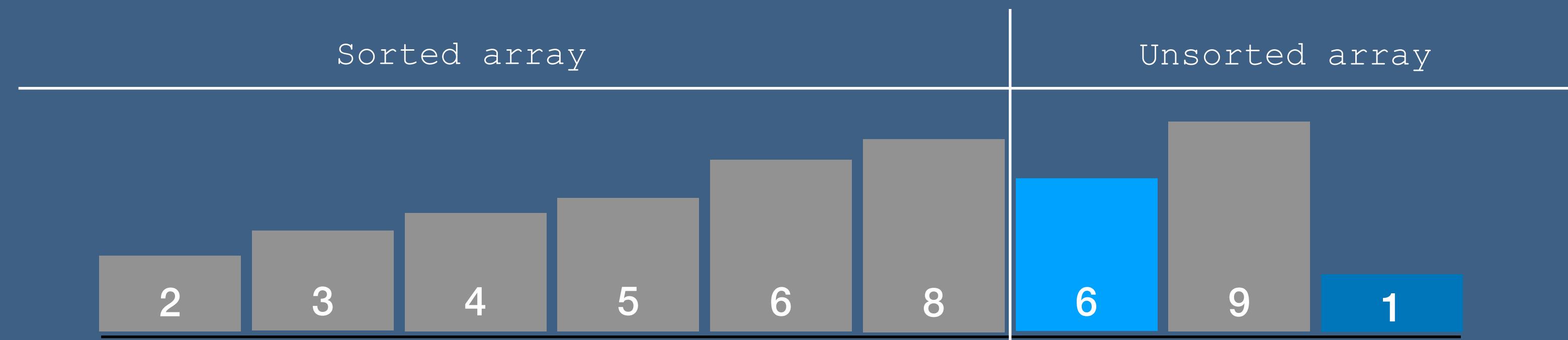
# Insertion Sort

- 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



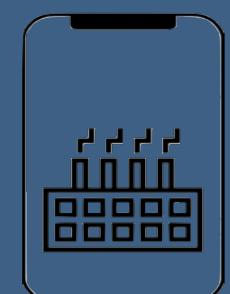
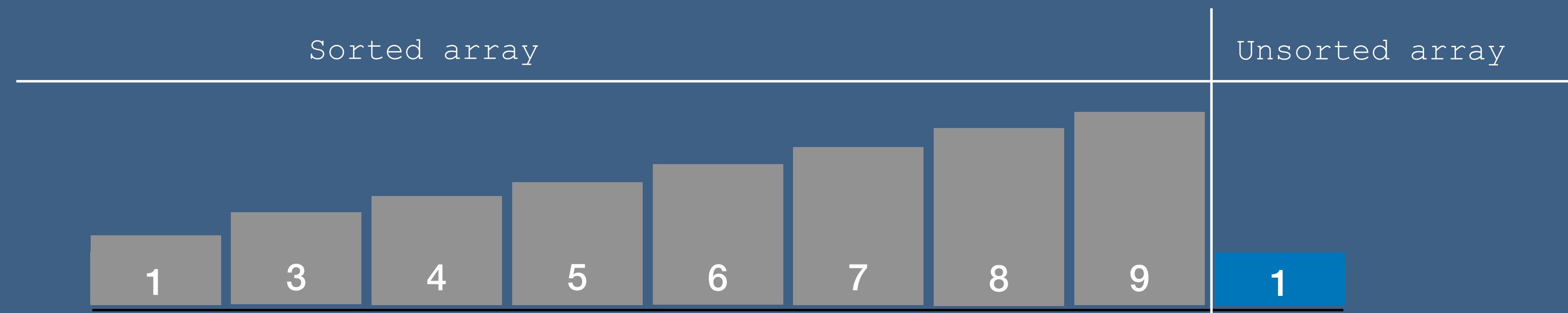
# Insertion Sort

- 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



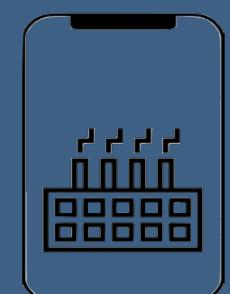
# Insertion Sort

## 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

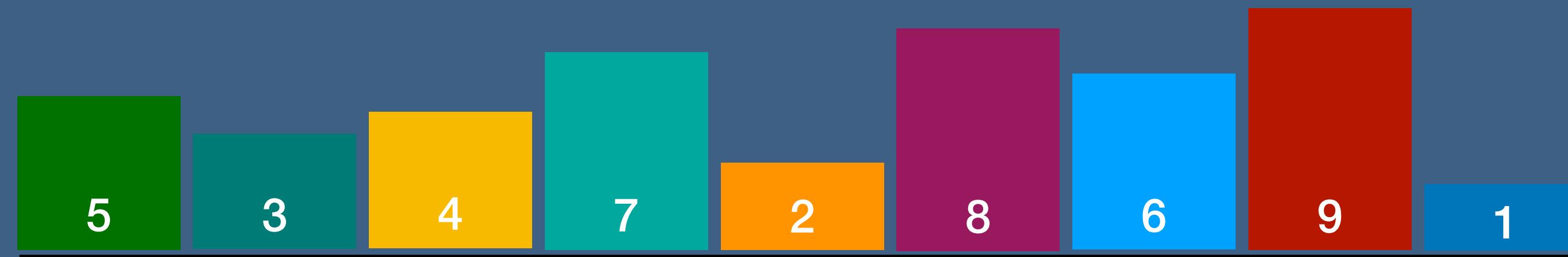
## When to avoid Insertion Sort?

- When time is a concern

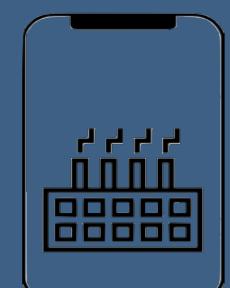
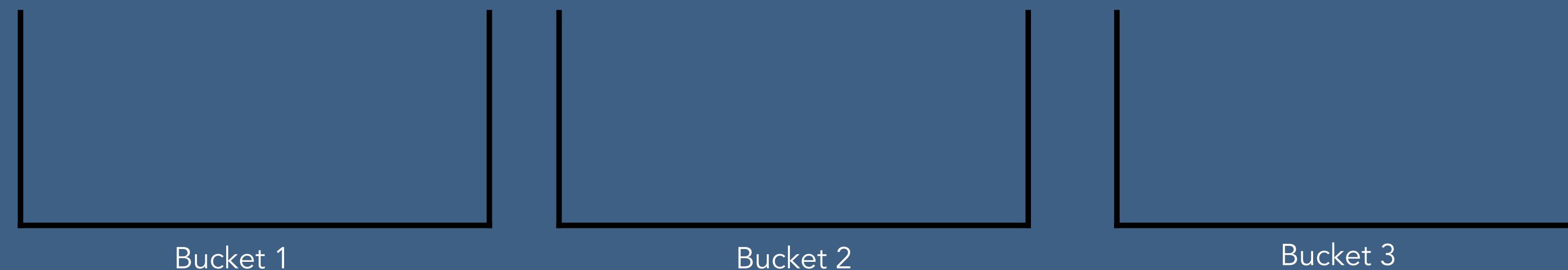


# Bucket Sort

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

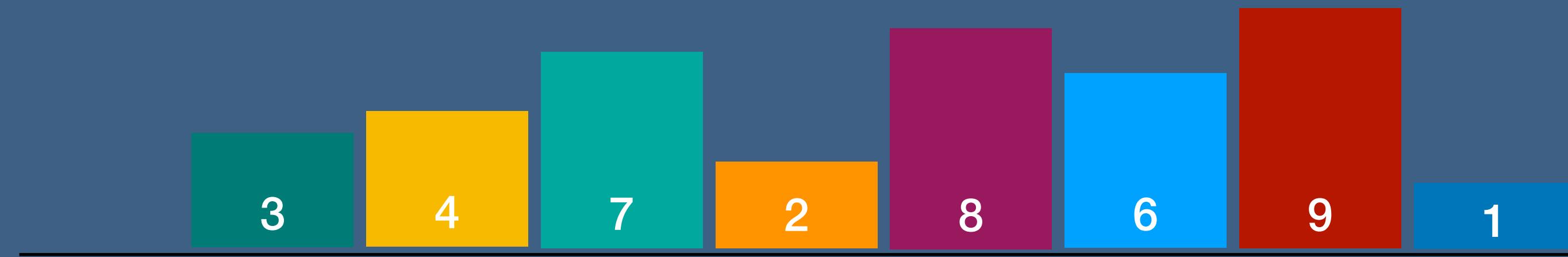


- Number of buckets =  $\text{round}(\text{Sqrt}(\text{number of elements}))$   
 $\text{round}(\sqrt{9}) = 3$
- Appropriate bucket =  $\text{ceil}(\text{Value} * \text{number of buckets} / \text{maxValue})$   
 $\text{ceil}(5 * 3 / 9) = \text{ceil}(1.6) = 2$



# Bucket Sort

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

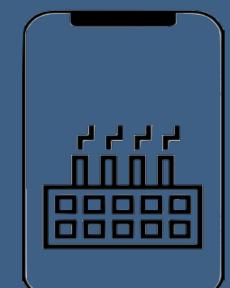
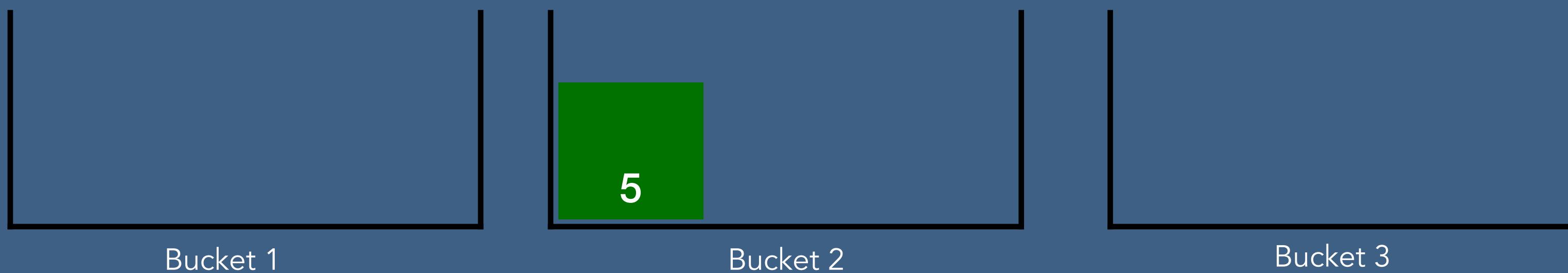


Number of buckets =  $\text{round}(\text{Sqrt}(\text{number of elements}))$

$\text{round}(\sqrt{9}) = 3$

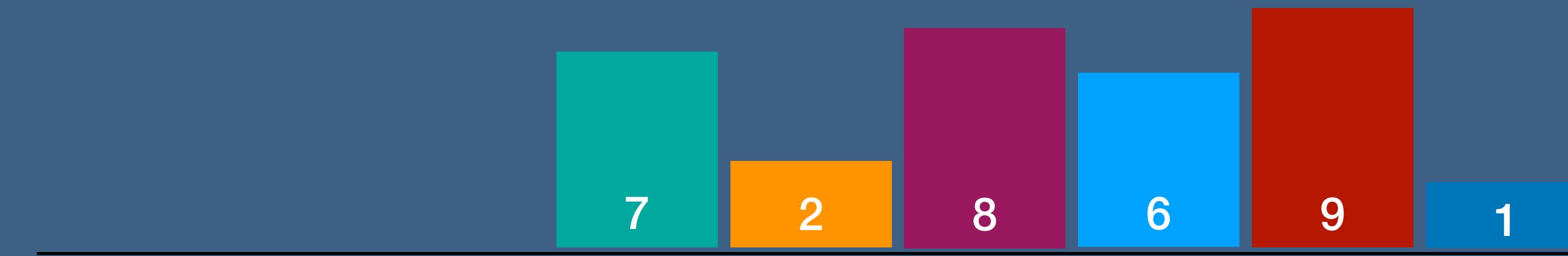
Appropriate bucket =  $\text{ceil}(\text{Value} * \text{number of buckets} / \text{maxValue})$

$\text{ceil}(4 * 3 / 9) = \text{ceil}(1.33) \approx 2$



# Bucket Sort

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

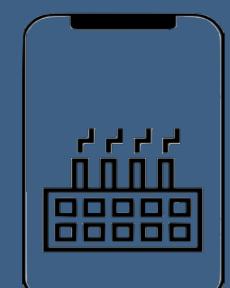
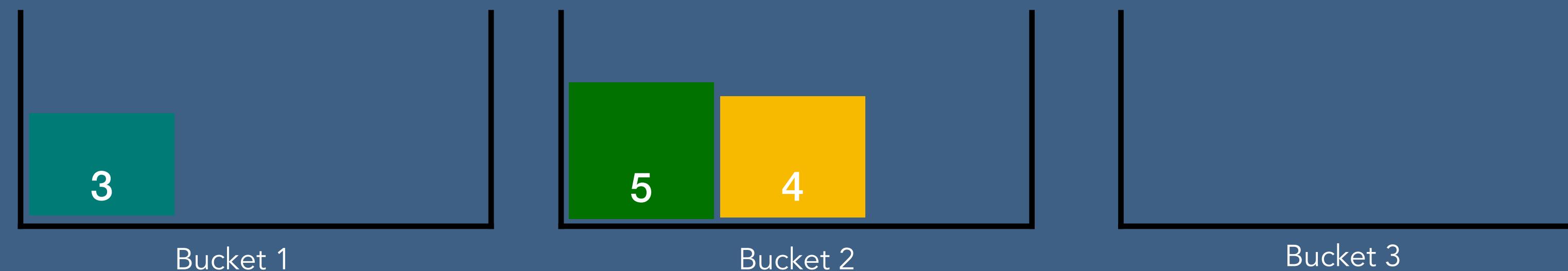


Number of buckets = round(Sqrt(number of elements))

round(sqrt(9)) = 3

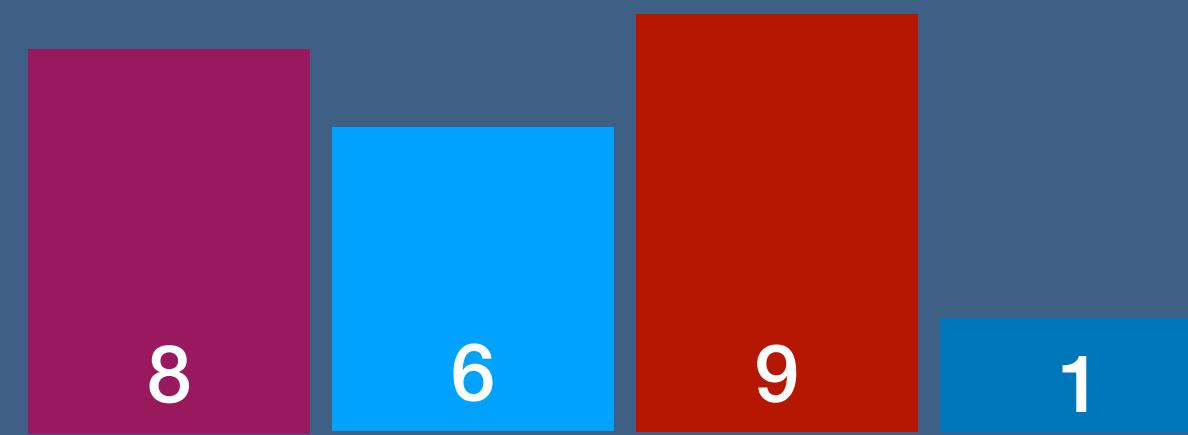
Appropriate bucket = ceil(Value \* number of buckets / maxValue)

ceil(2\*3/9) = ceil(0.8) = 3



# Bucket Sort

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

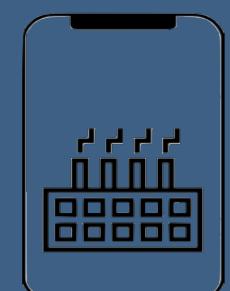
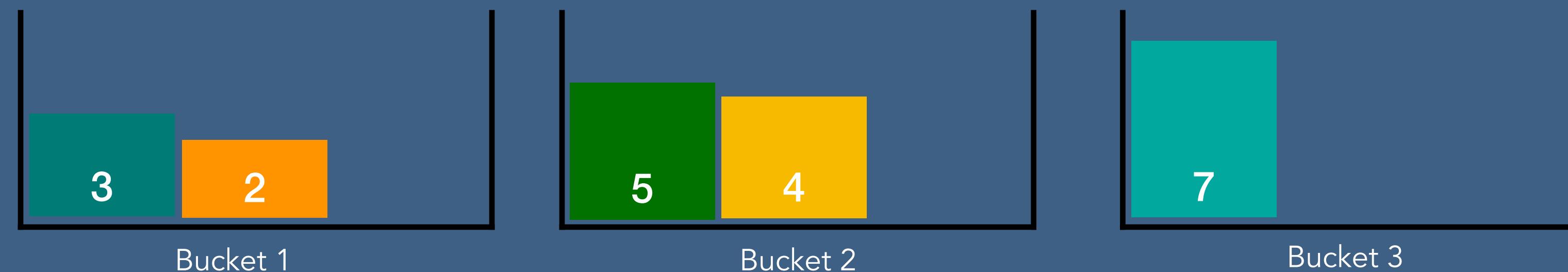


Number of buckets =  $\text{round}(\text{Sqrt}(\text{number of elements}))$

$\text{round}(\sqrt{9}) = 3$

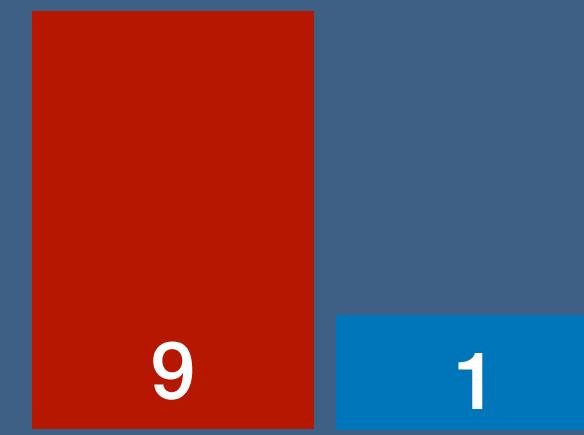
Appropriate bucket =  $\text{ceil}(\text{Value} * \text{number of buckets} / \text{maxValue})$

$\text{ceil}(8 * 3 / 9) = \text{ceil}(2.67) \geq 3$



# Bucket Sort

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

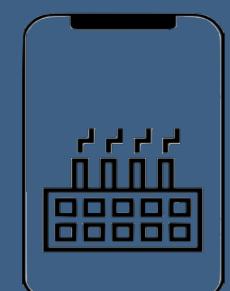
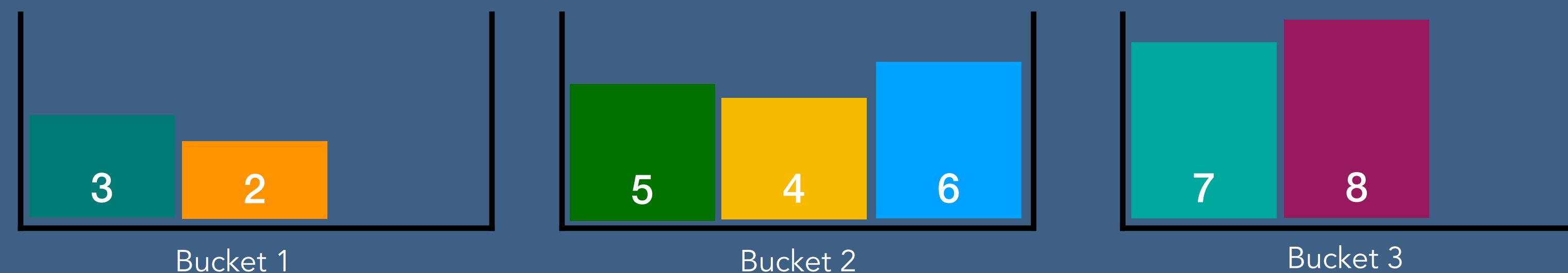


Number of buckets = round(Sqrt(number of elements))

round(sqrt(9)) = 3

Appropriate bucket = ceil(Value \* number of buckets / maxValue)

$\text{ceil}(9 \times 3 / 9) = \text{ceil}(3) \rightarrow 3$   
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

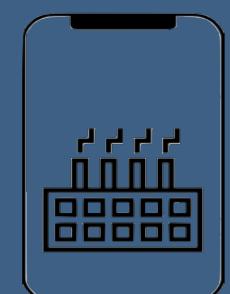
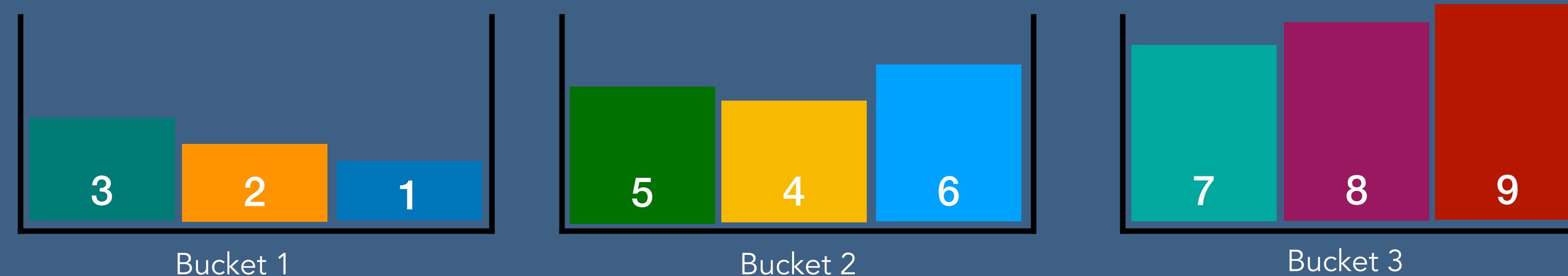
---

Number of buckets =  $\text{round}(\text{Sqrt}(\text{number of elements}))$

$\text{round}(\sqrt{9}) = 3$

Appropriate bucket =  $\text{ceil}(\text{Value} * \text{number of buckets} / \text{maxValue})$

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

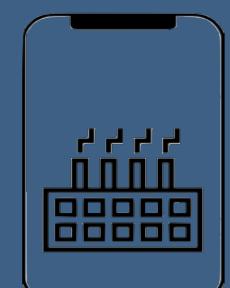
---

Number of buckets =  $\text{round}(\text{Sqrt}(\text{number of elements}))$

$\text{round}(\sqrt{9}) = 3$

Appropriate bucket =  $\text{ceil}(\text{Value} * \text{number of buckets} / \text{maxValue})$

Sort all buckets (using any sorting algorithm)



# Bucket Sort

## When to use Bucket Sort?

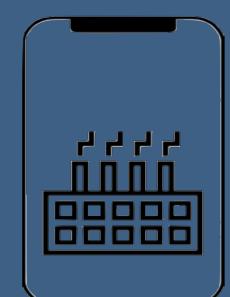
- When input uniformly distributed over range

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

1,2,4,91,3,95

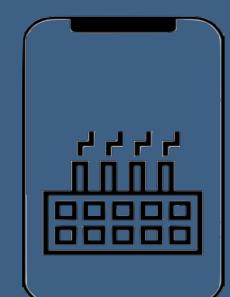
## When to avoid Bucket Sort?

- When space is a concern

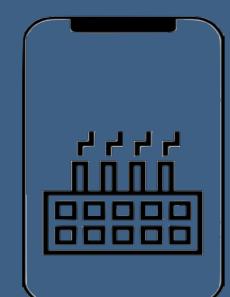
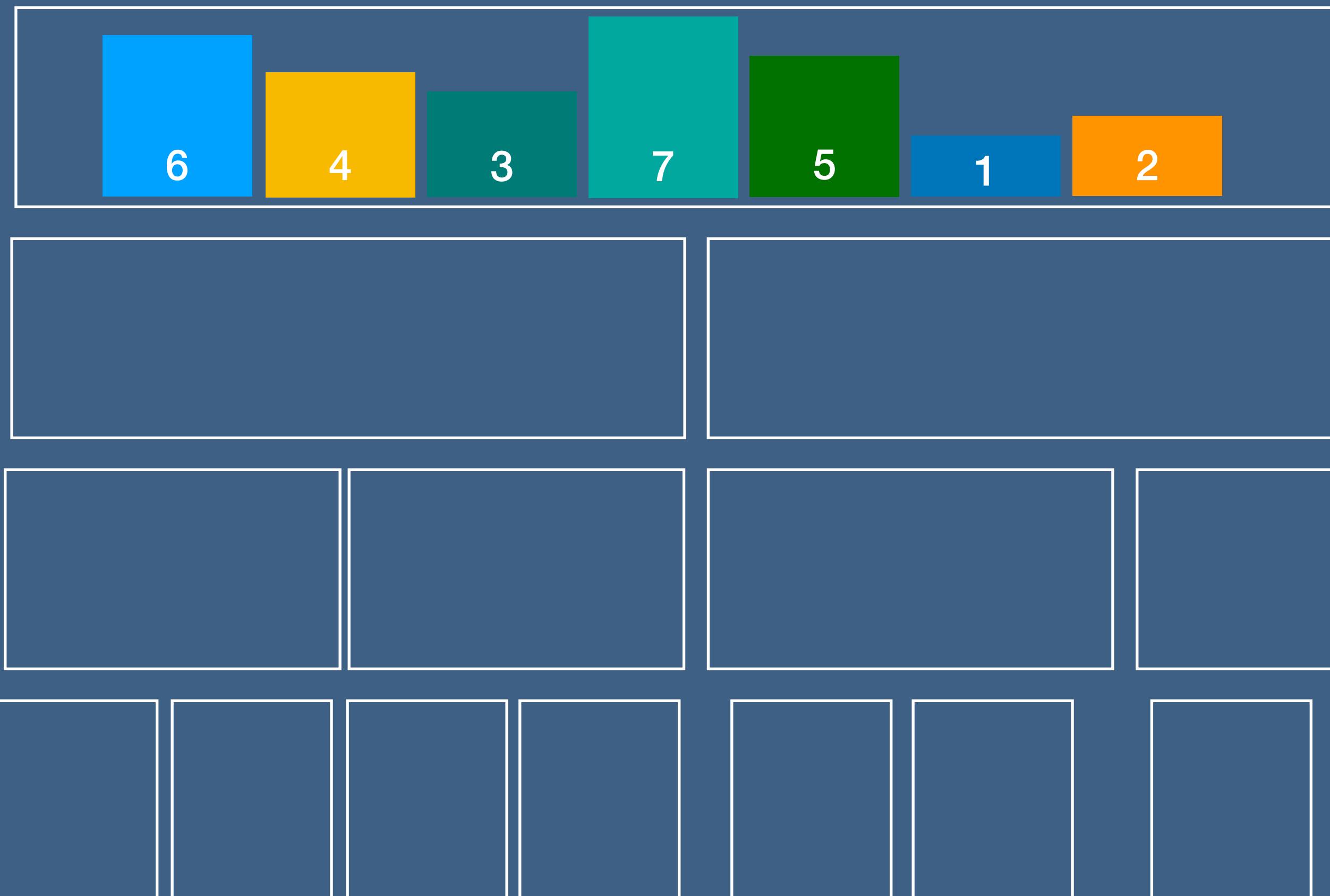


# Merge Sort

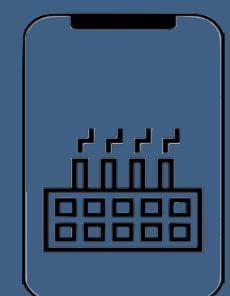
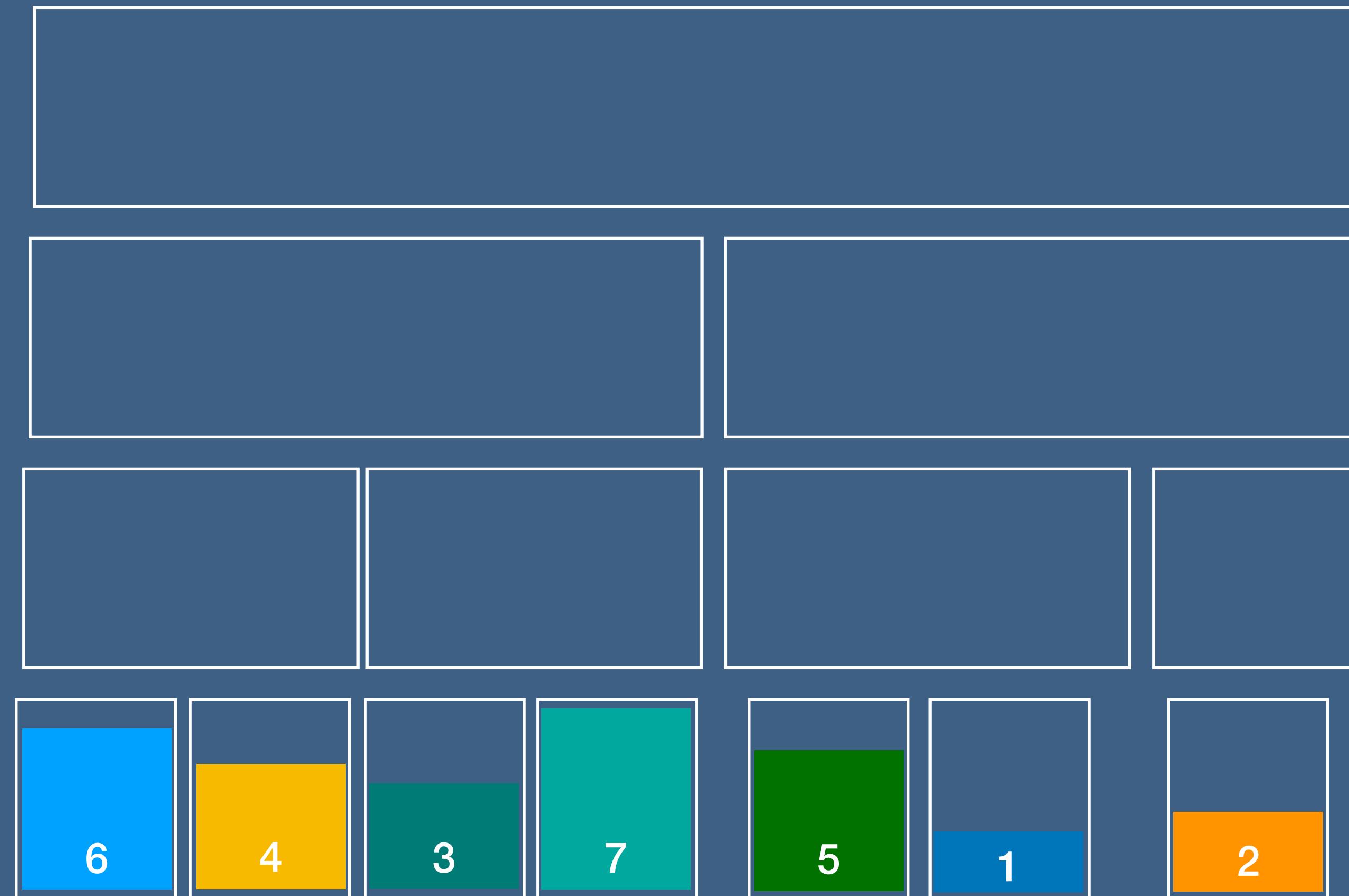
- Merge sort is a divide and conquer algorithm
- Divide the input array in two halves and we keep halving recursively until they become too small that cannot be broken further
- Merge halves by sorting them



# Merge Sort



# Merge Sort



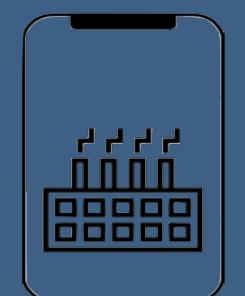
# Merge Sort

## When to use Merge Sort?

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

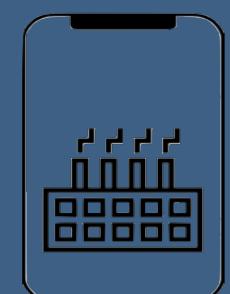
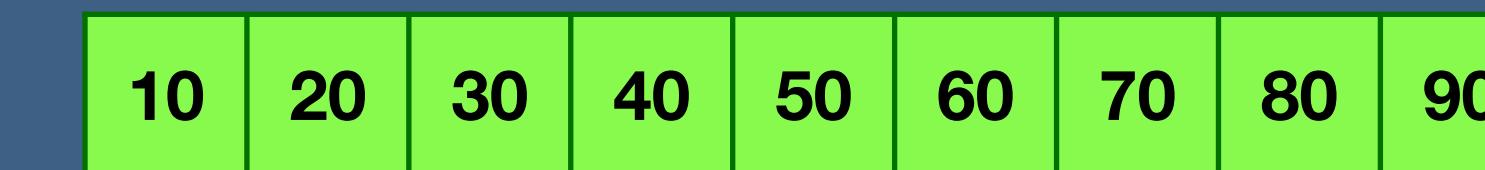
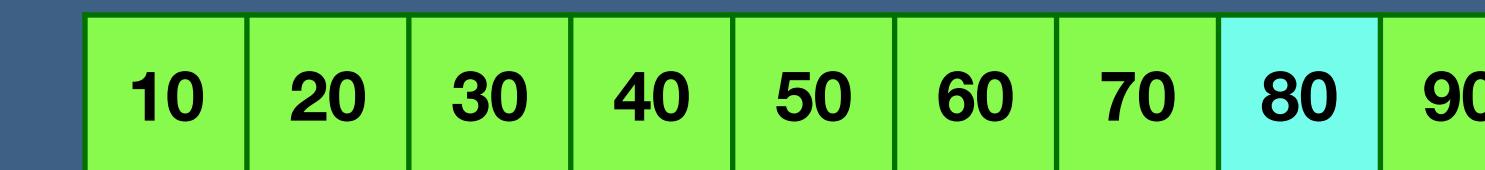
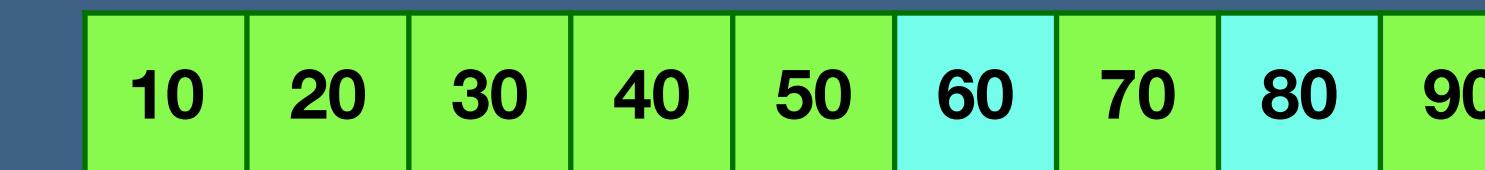
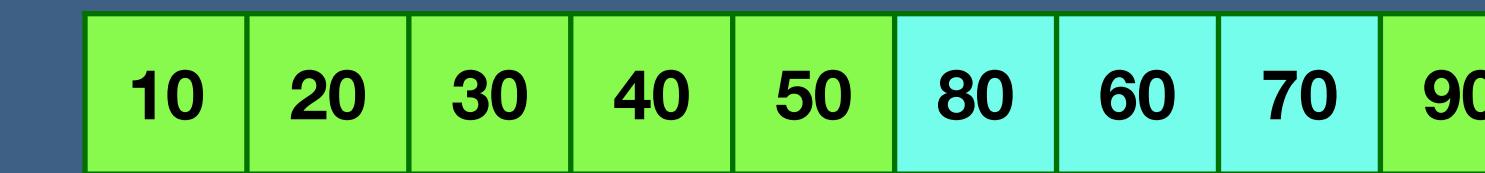
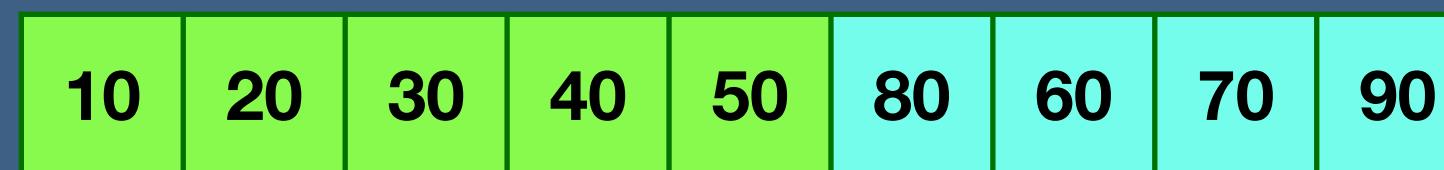
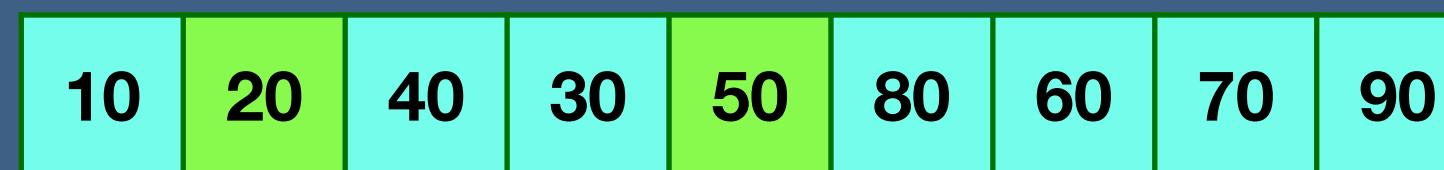
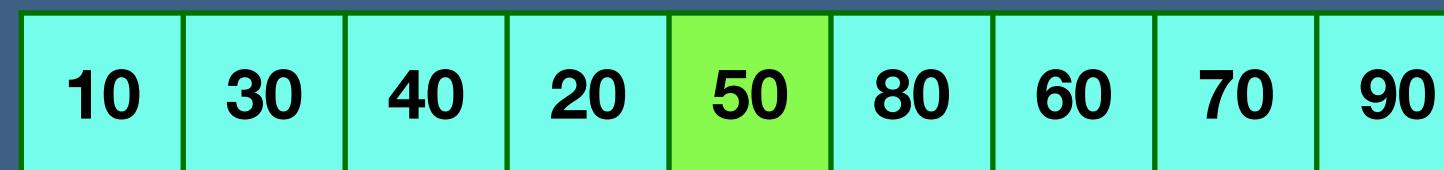
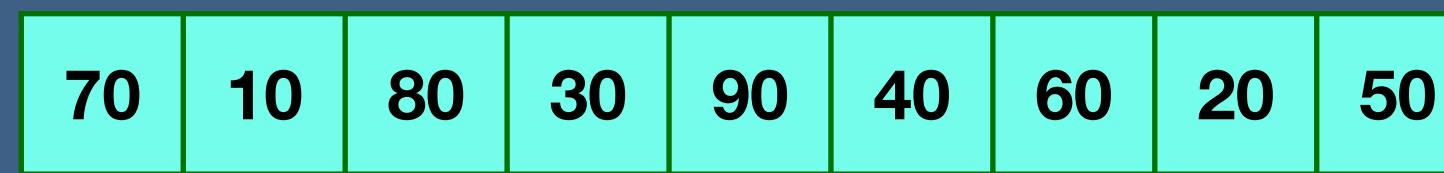
## When to avoid Merge Sort?

- When space is a concern



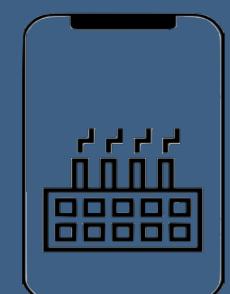
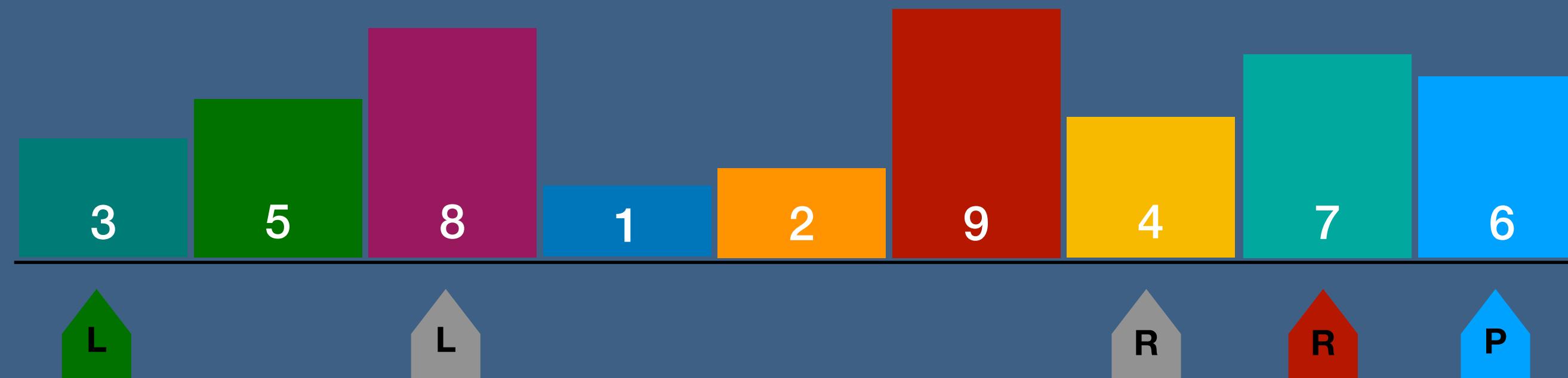
# Quick Sort

- 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



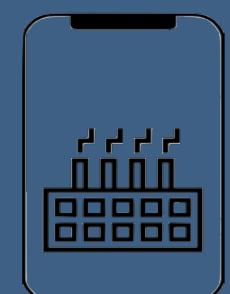
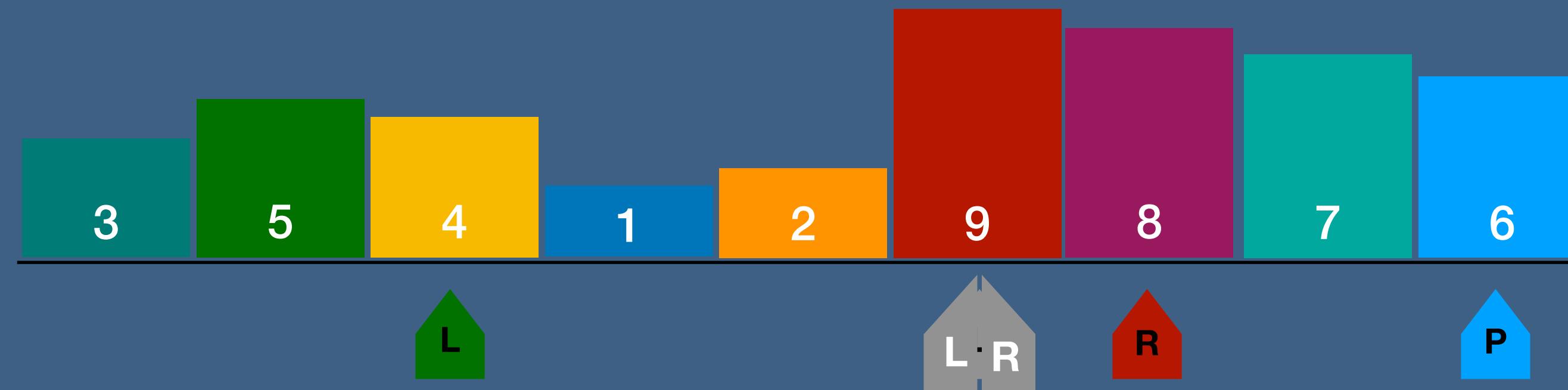
# Quick Sort

- 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



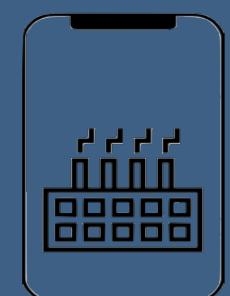
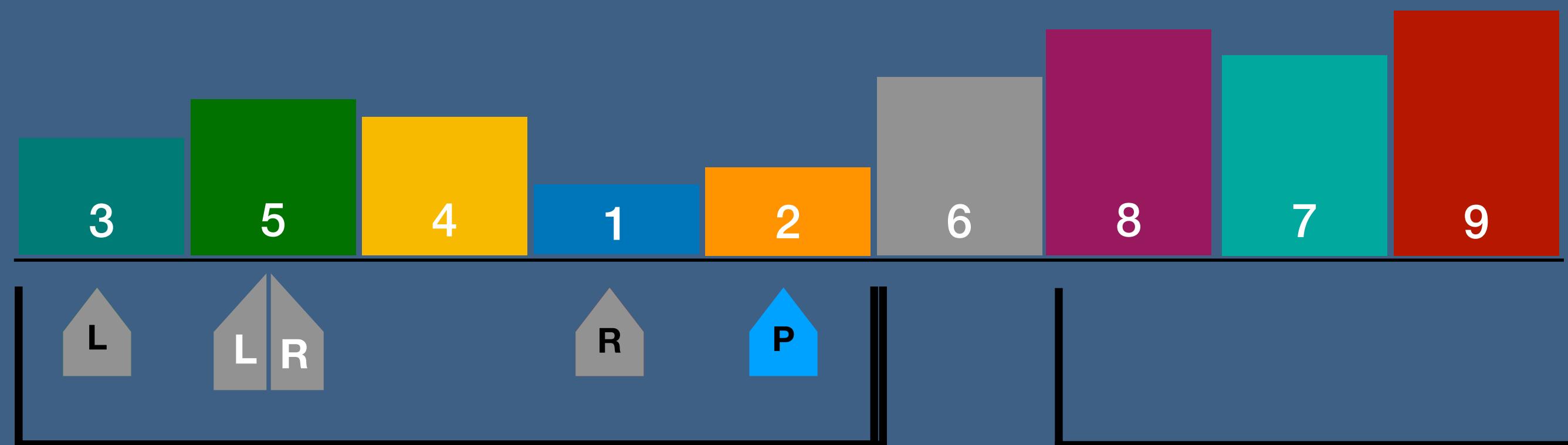
# Quick Sort

- 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



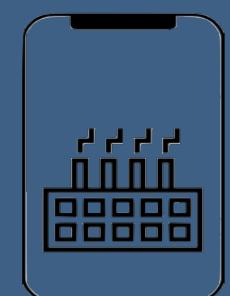
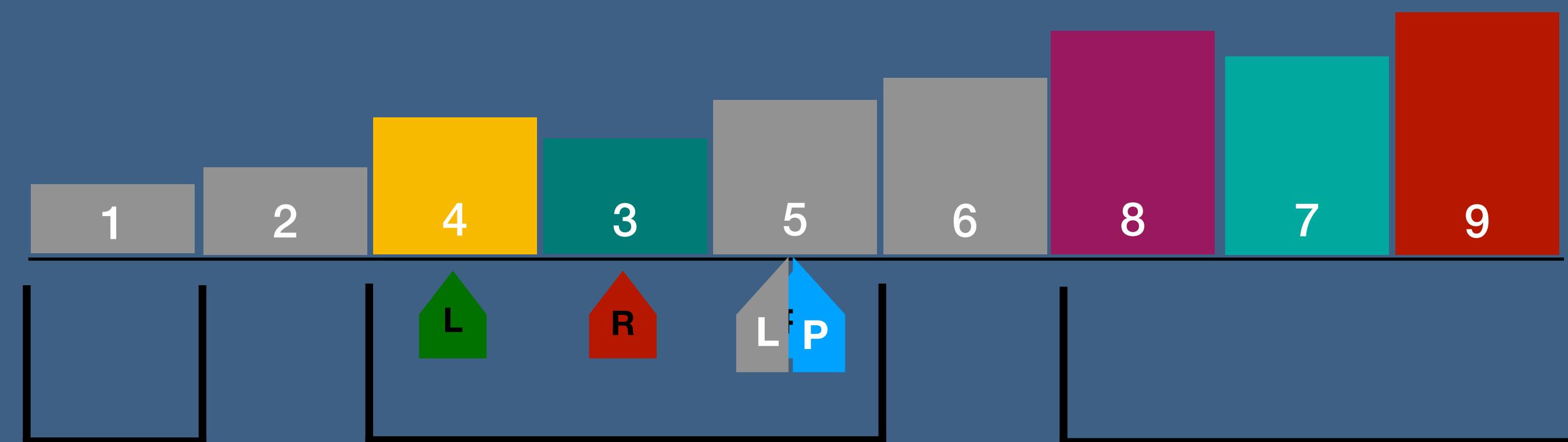
# Quick Sort

- 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



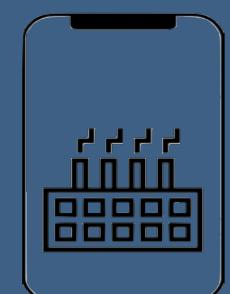
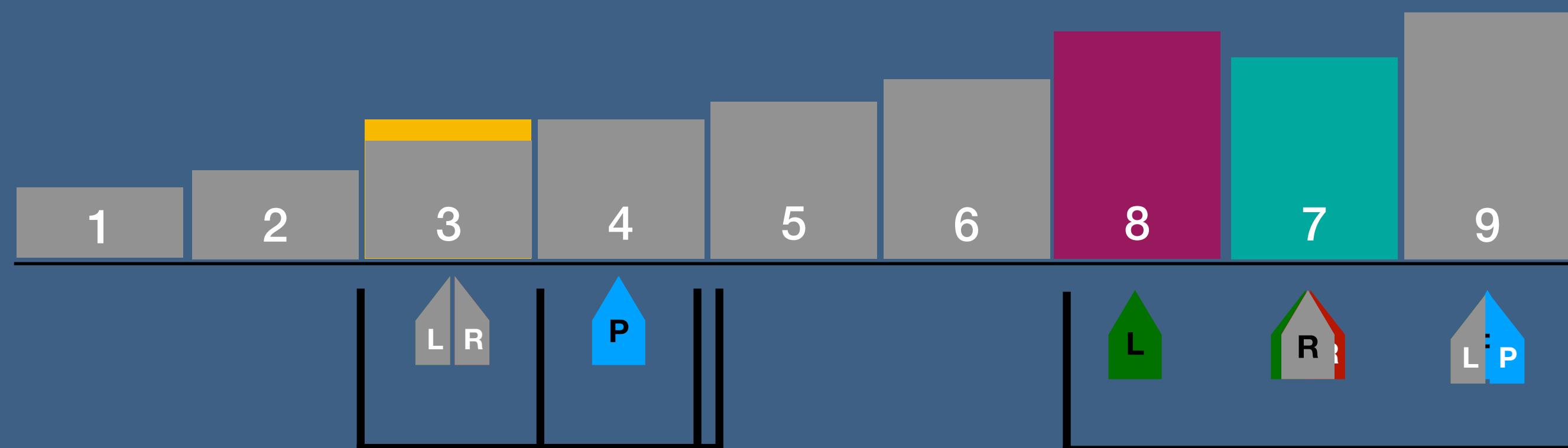
# Quick Sort

- 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



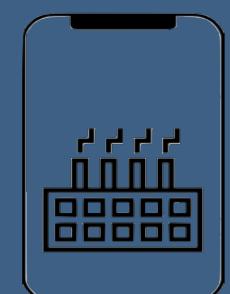
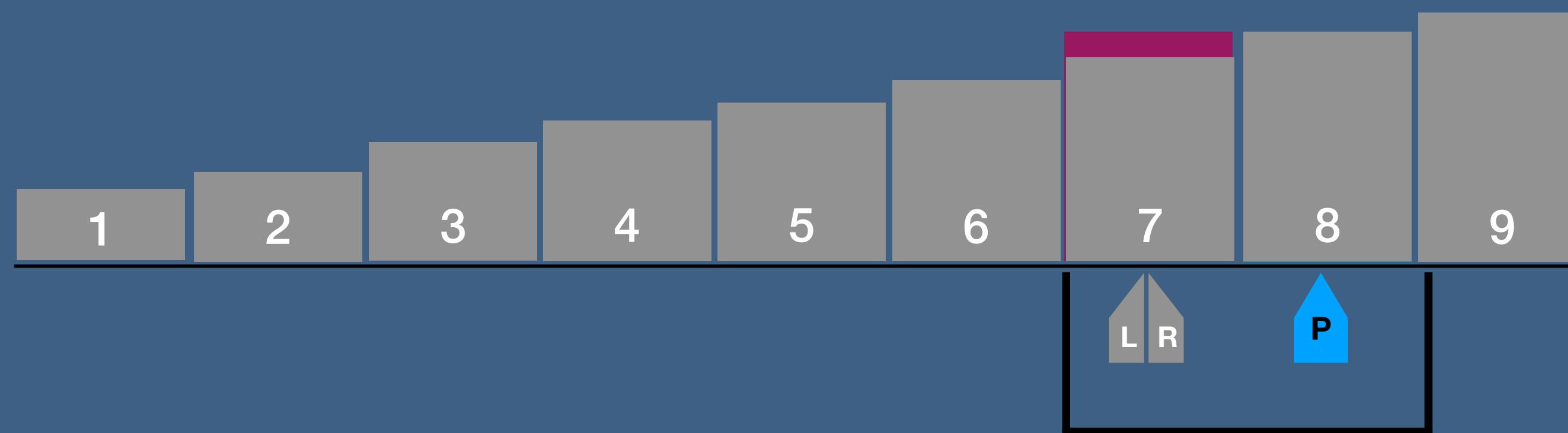
# Quick Sort

- 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



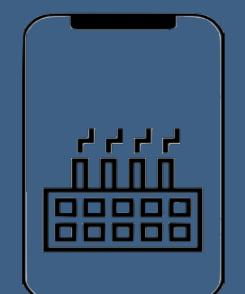
# Quick Sort

## 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

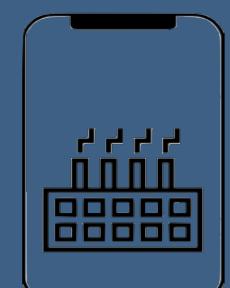
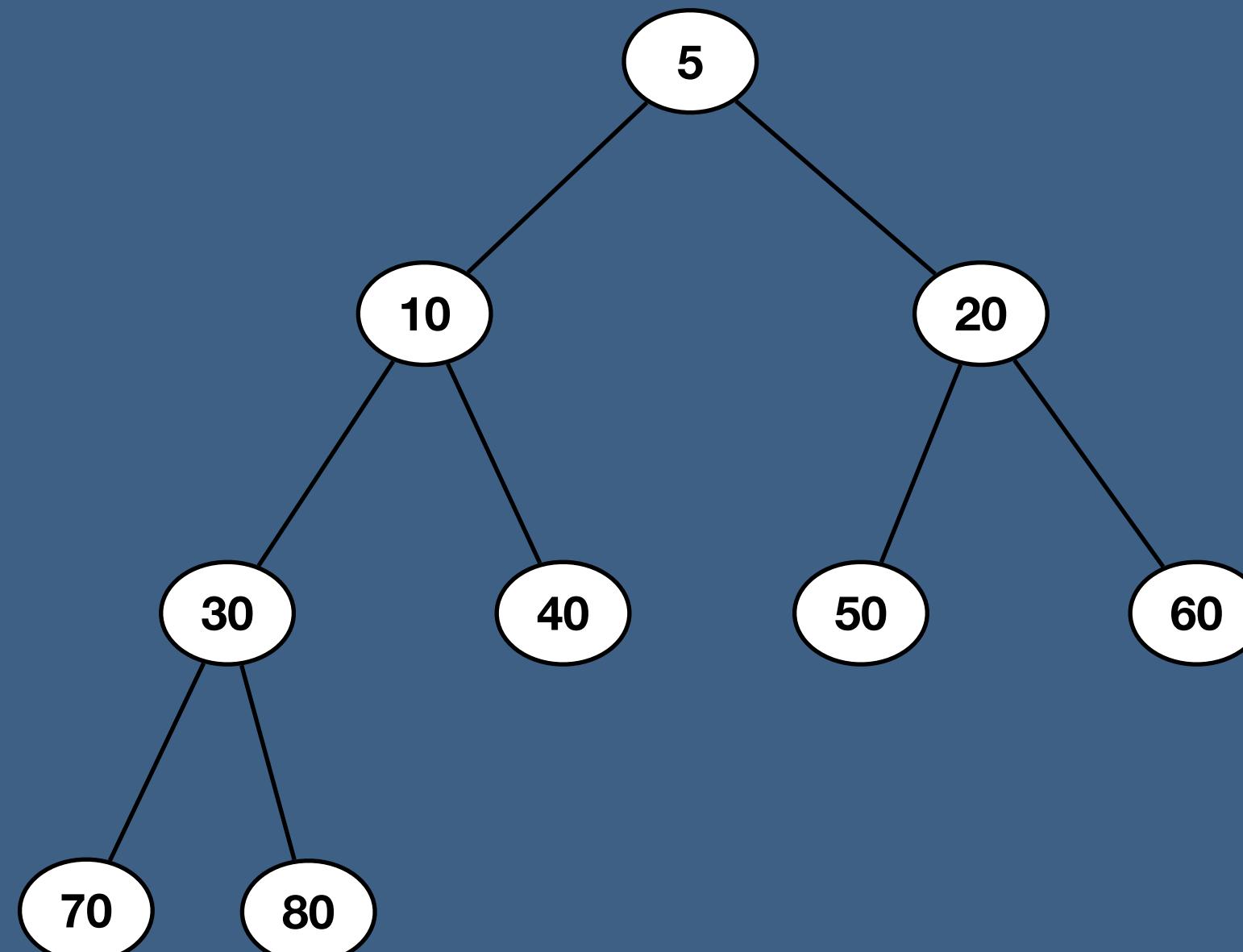


# Heap Sort

- 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

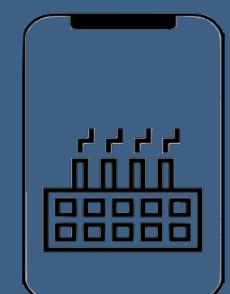
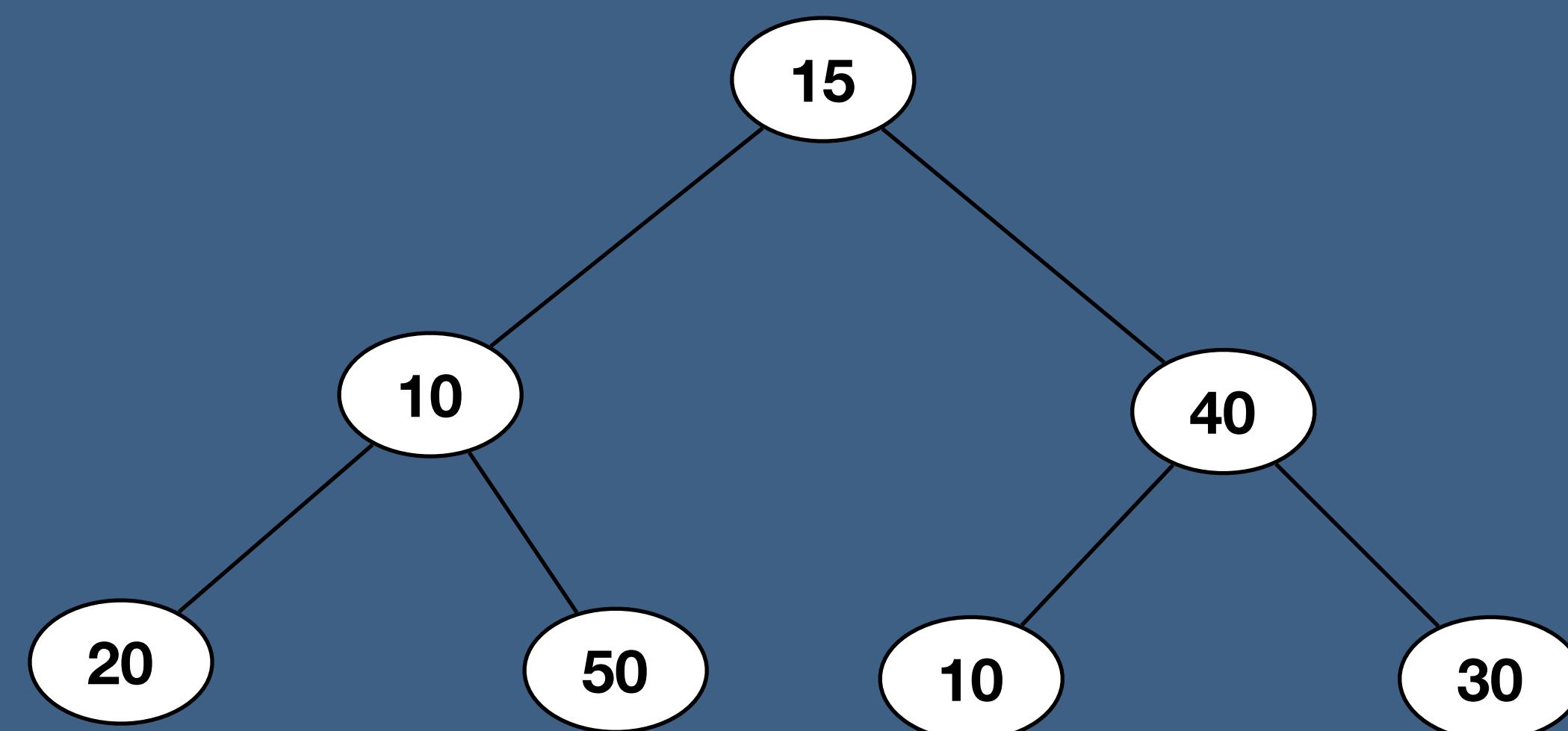
- The value of any given node must be less or equal of its children (min heap)
- The value of any given node must be greater or equal of its children (max heap)



# Heap Sort

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

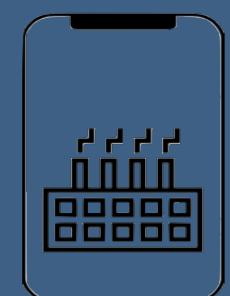
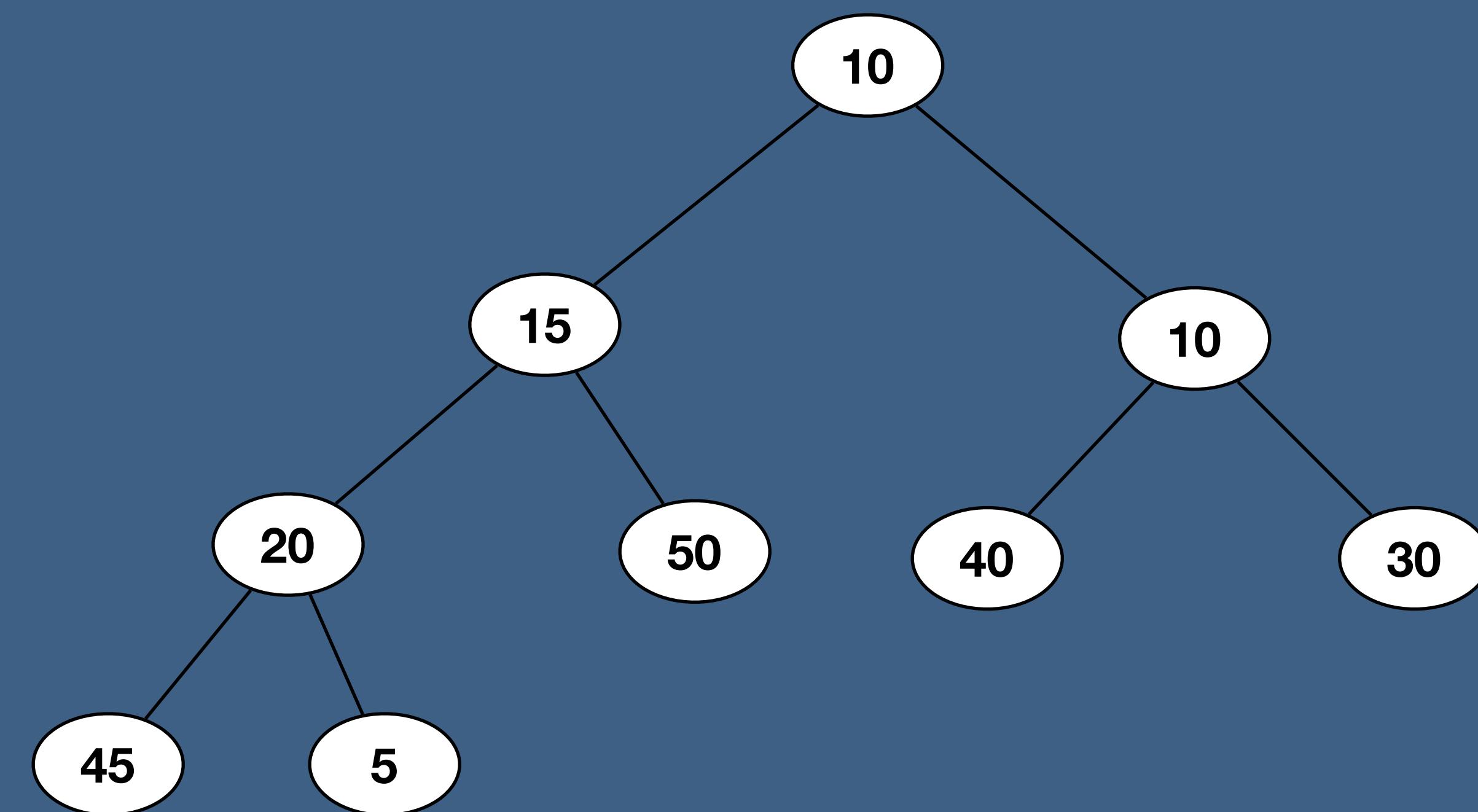
15	10	40	20	50	10	30	45	5
----	----	----	----	----	----	----	----	---



# Heap Sort

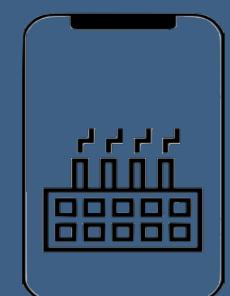
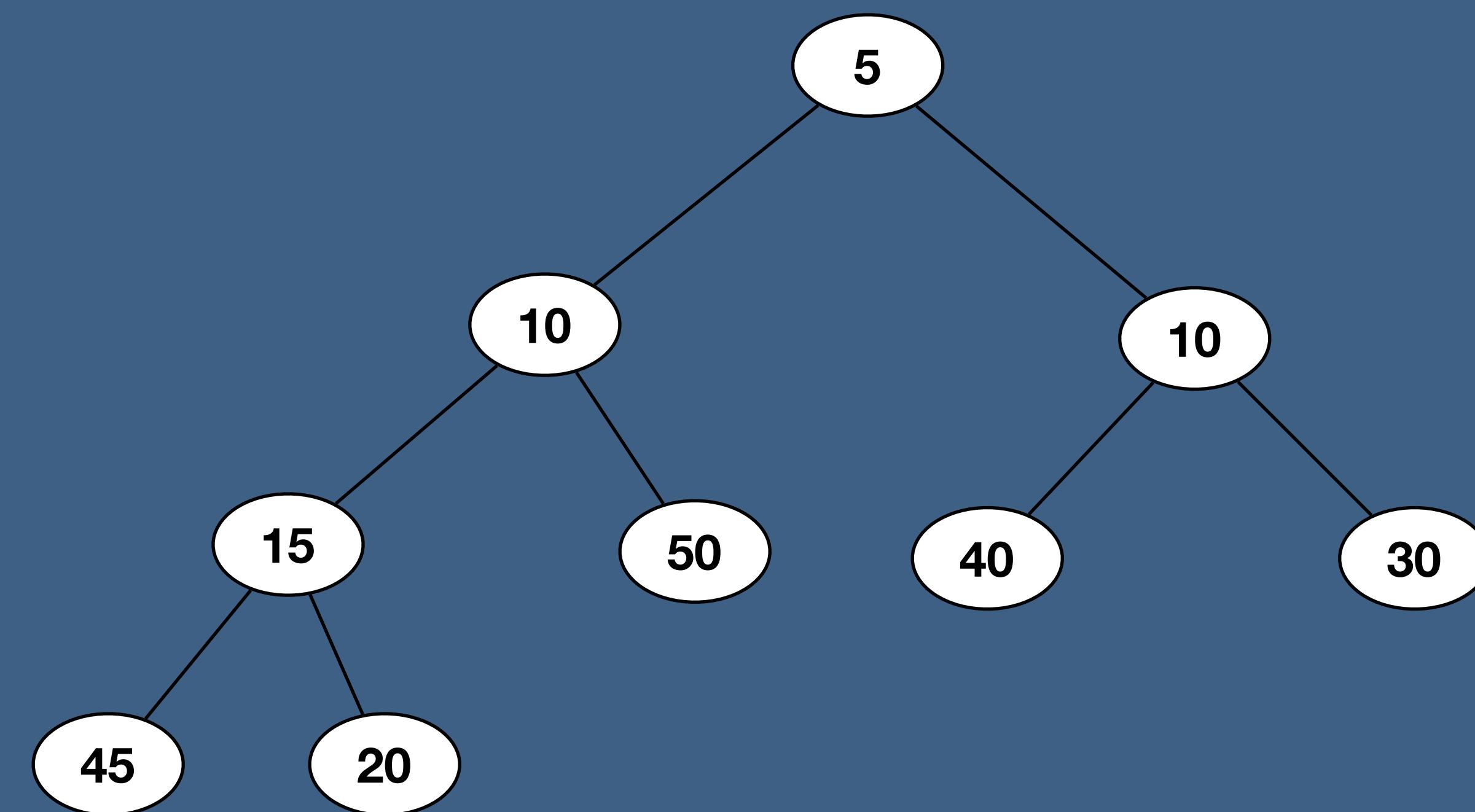
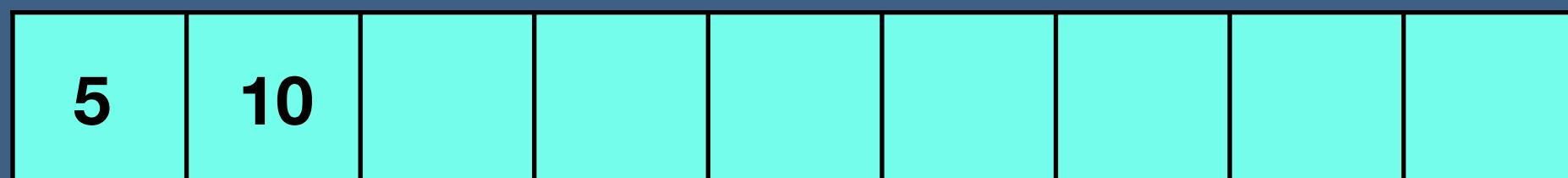
- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap

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



# Heap Sort

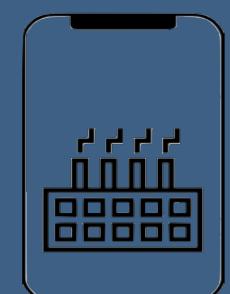
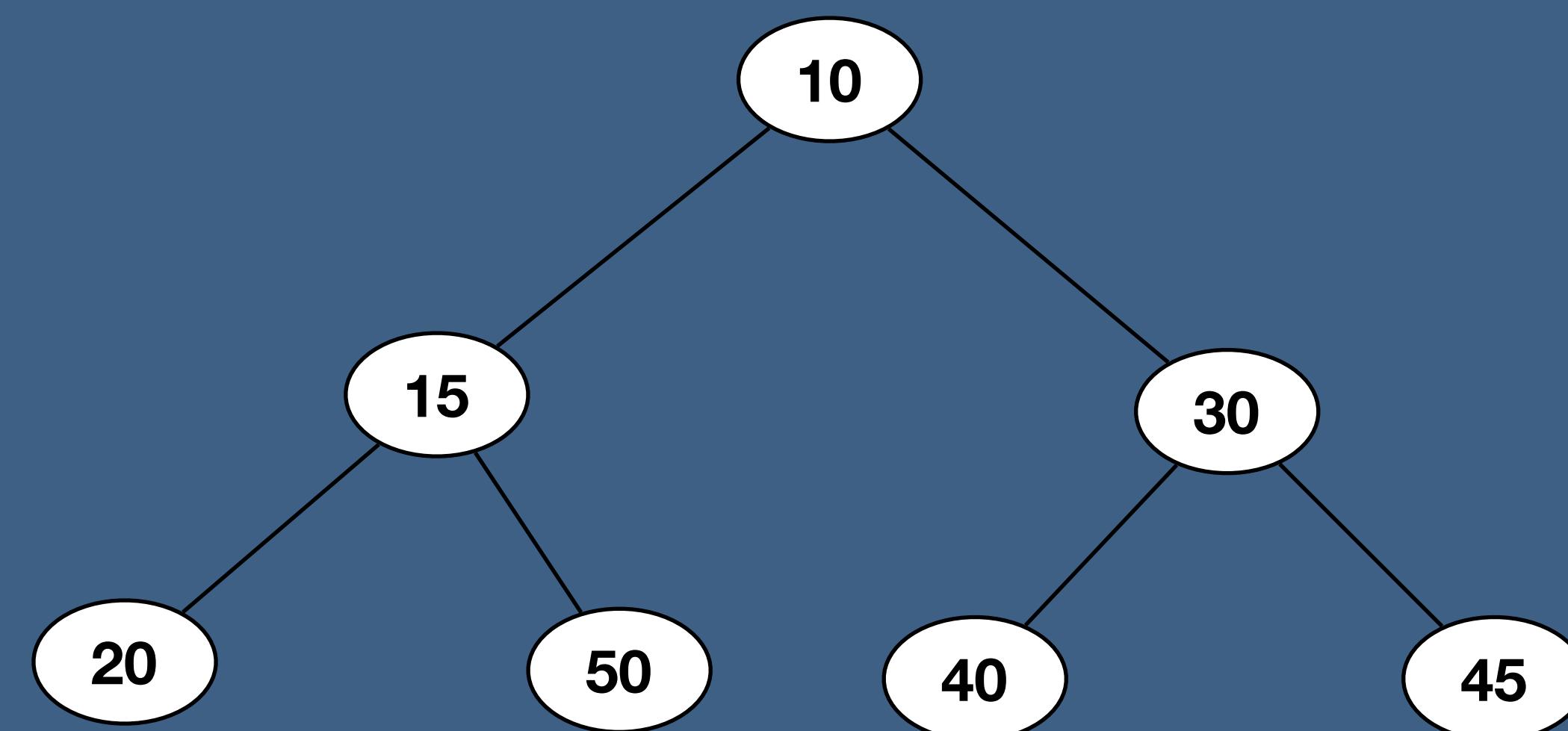
- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap



# Heap Sort

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

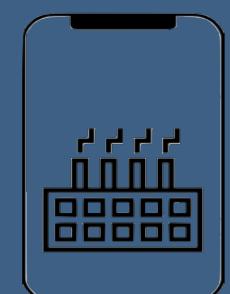
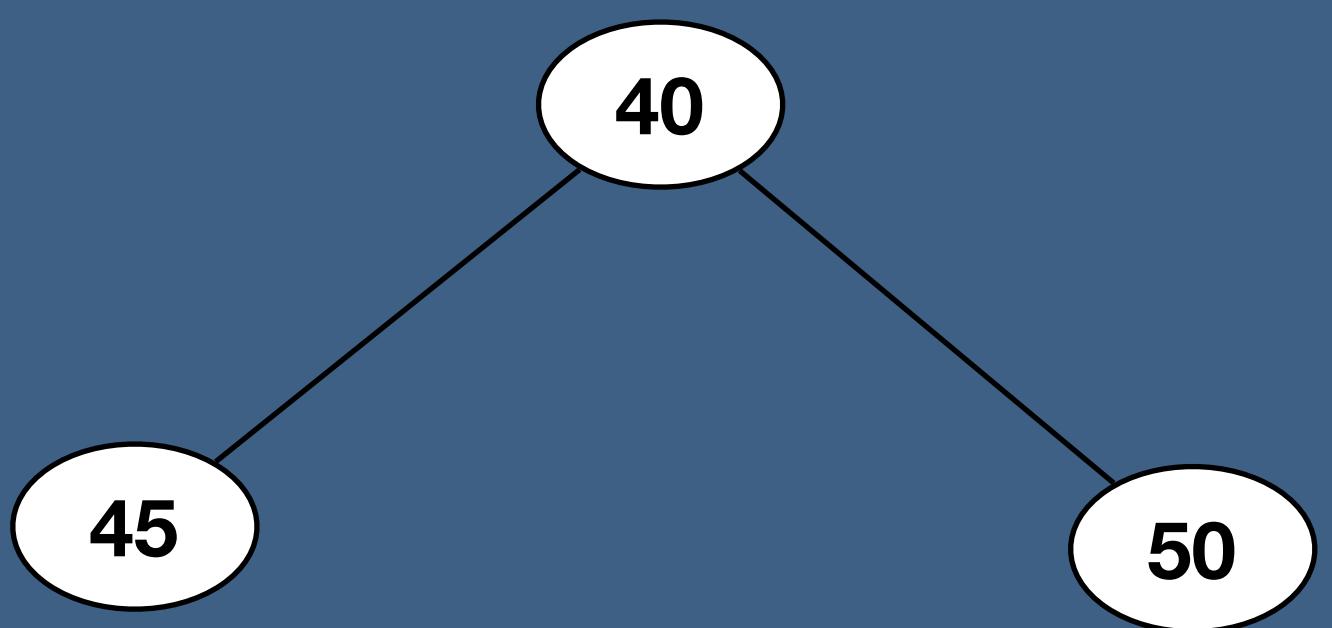
5	10	10	15	20	30			
---	----	----	----	----	----	--	--	--



# Heap Sort

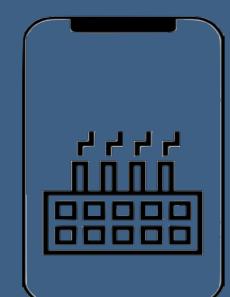
- Step 1 : Insert data to Binary Heap Tree
- Step 2 : Extract data from Binary Heap

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

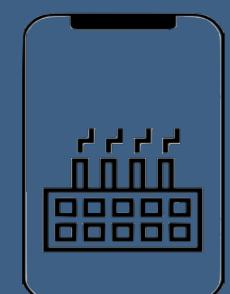


# Sorting Algorithms

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

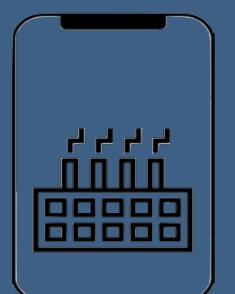


# Linear Search in Java

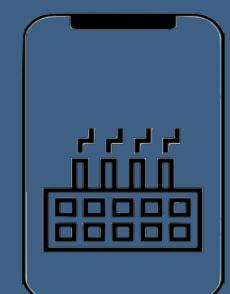


# Linear Search Pseudocode

- 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

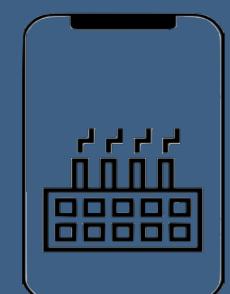


# Binary Search in Java

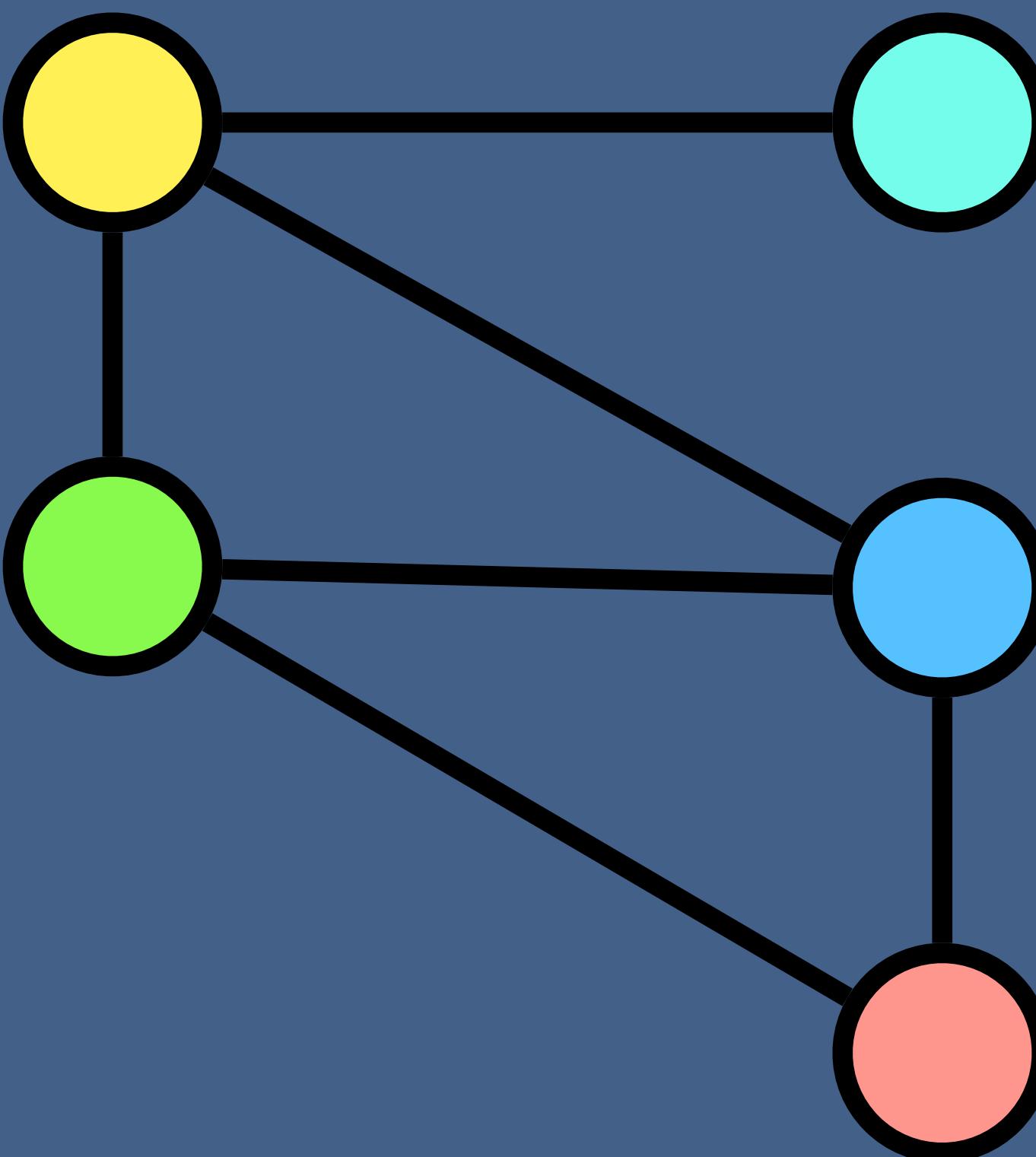


# Binary Search Pseudocode

- 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



# Graph



# What you will learn

What is a graph? Why do we need it?

Graph Terminologies

Types of graphs. Graph Representation

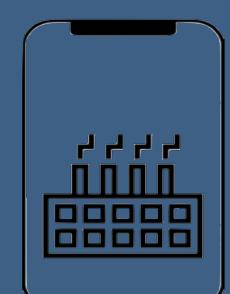
Traversal of graphs. (BFS and DFS)

Topological Sorting

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

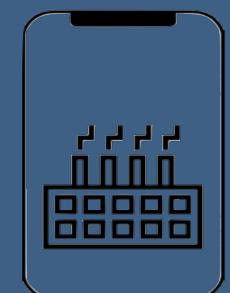
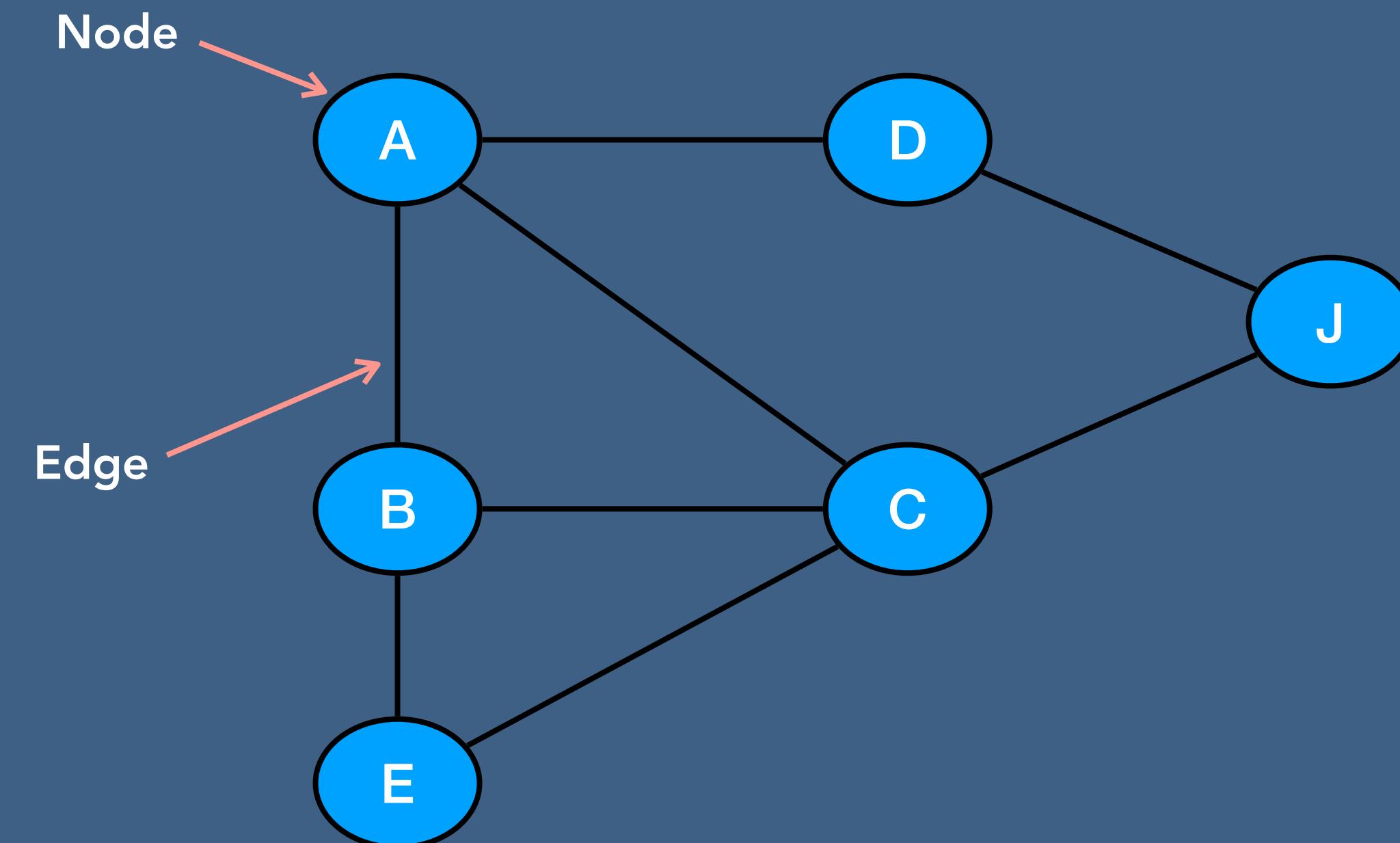
All pairs shortest path (BFS, Dijkstra, Bellman Ford and Floyd Warshall algorithms)

Minimum Spanning Tree (Kruskal and Prim algorithms)

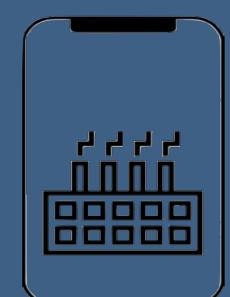
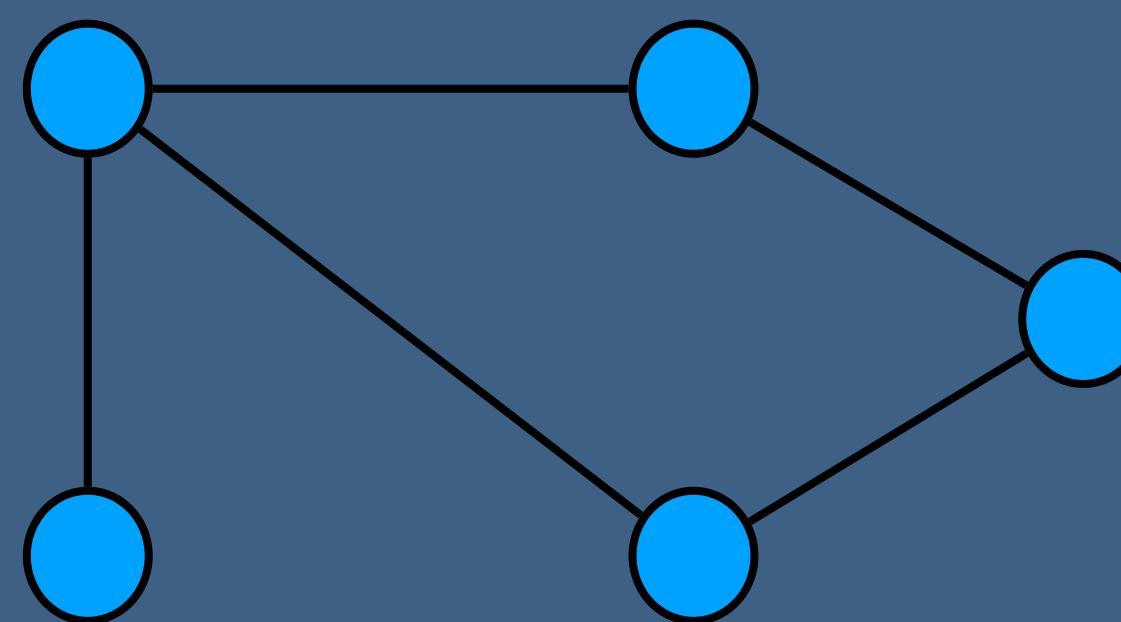


# What is a Graph?

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

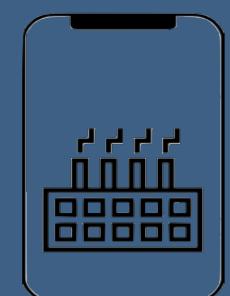
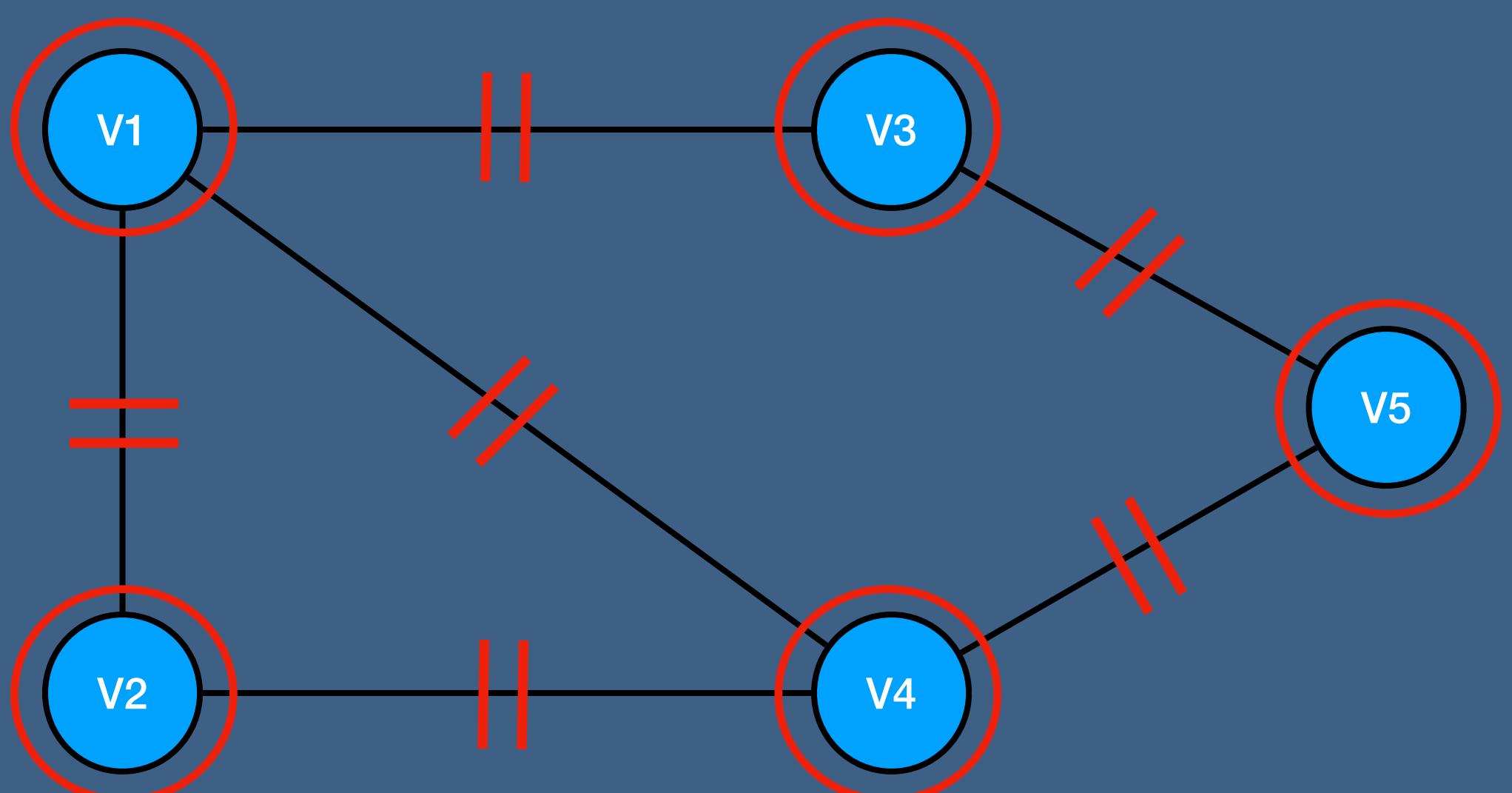


# Why Graph?



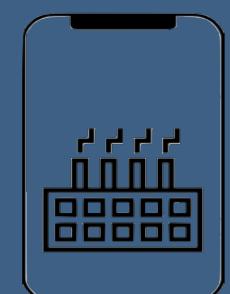
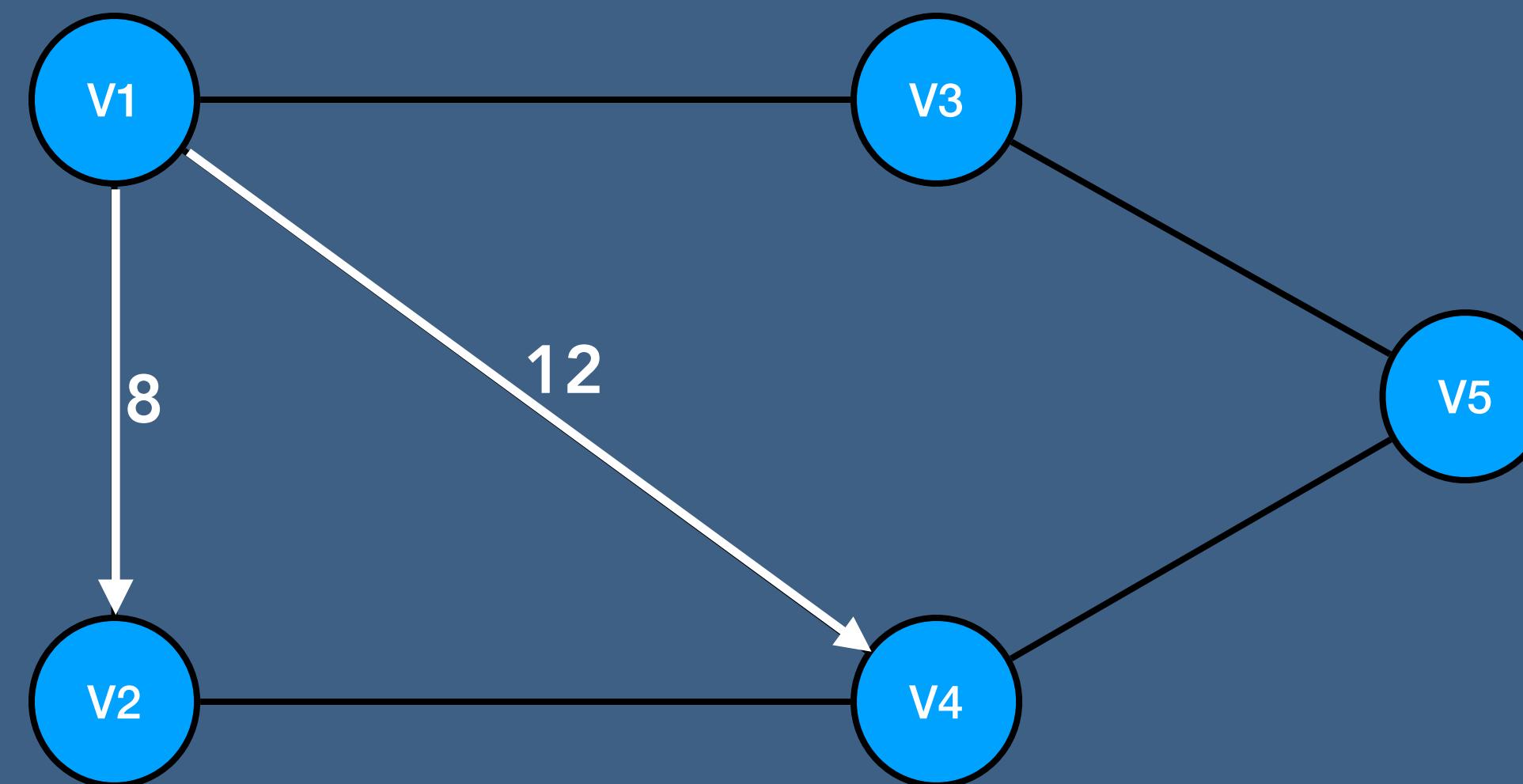
# Graph Terminology

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



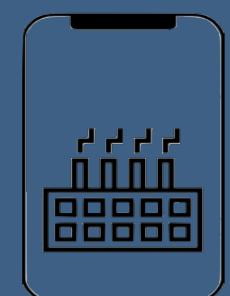
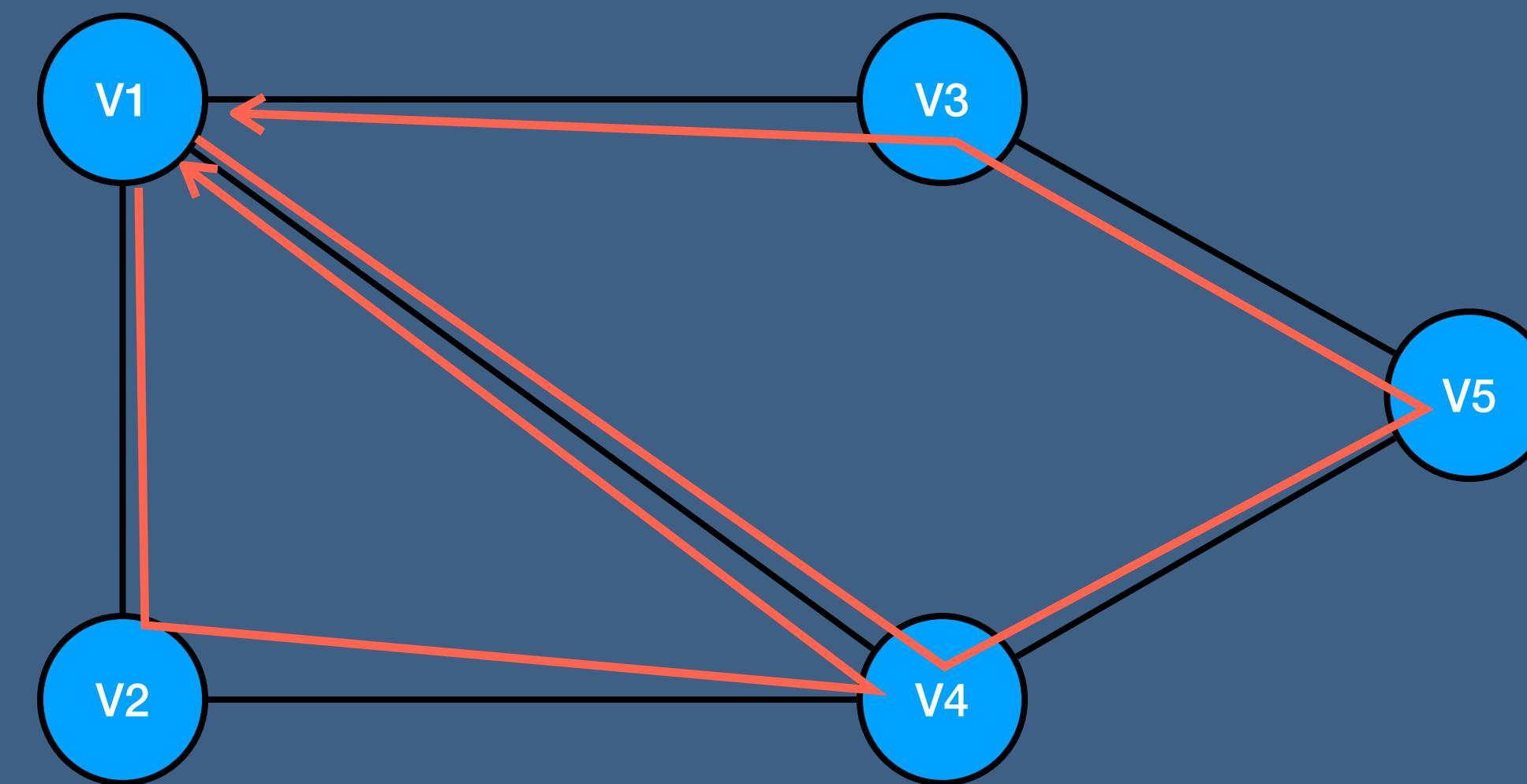
# Graph Terminology

- **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



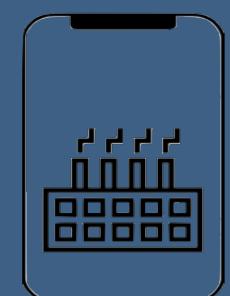
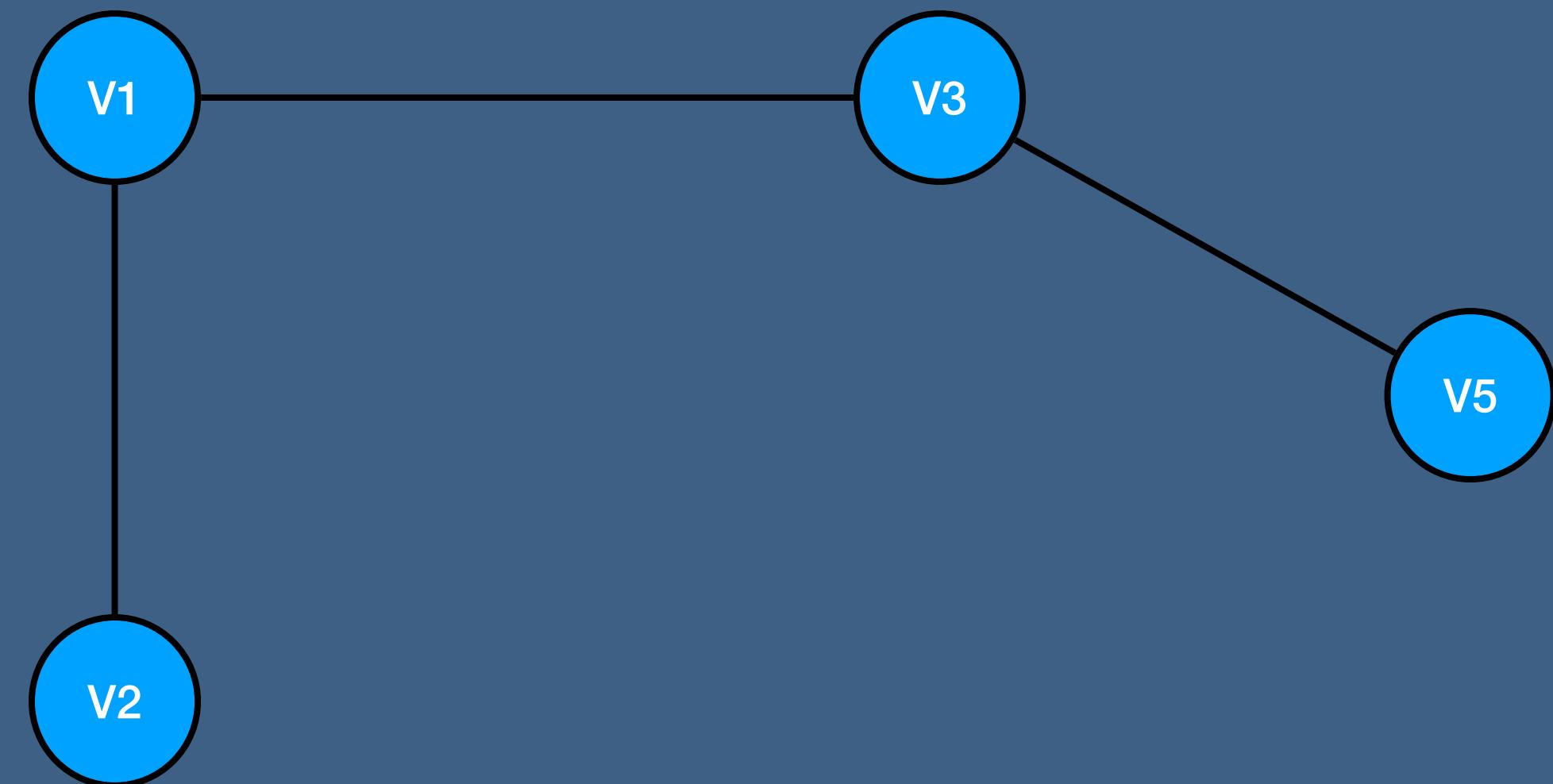
# Graph Terminology

- **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



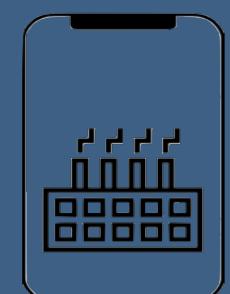
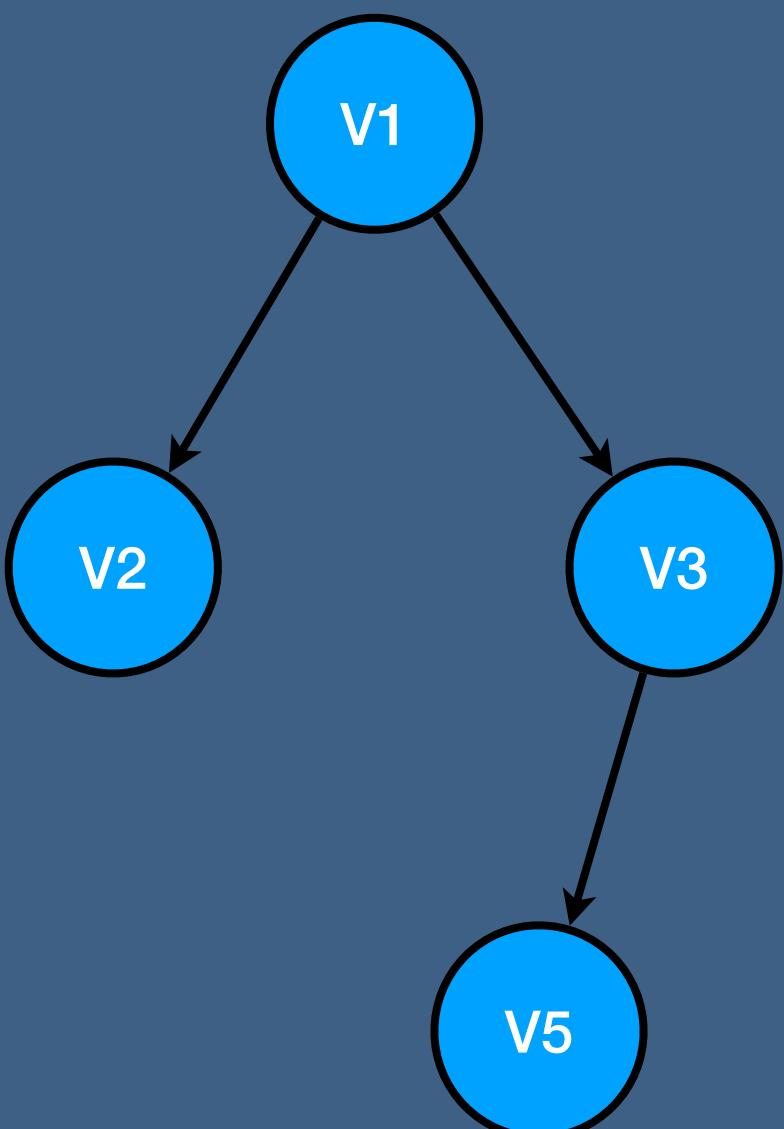
# Graph Terminology

- **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

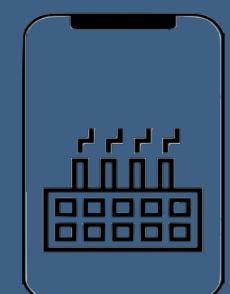
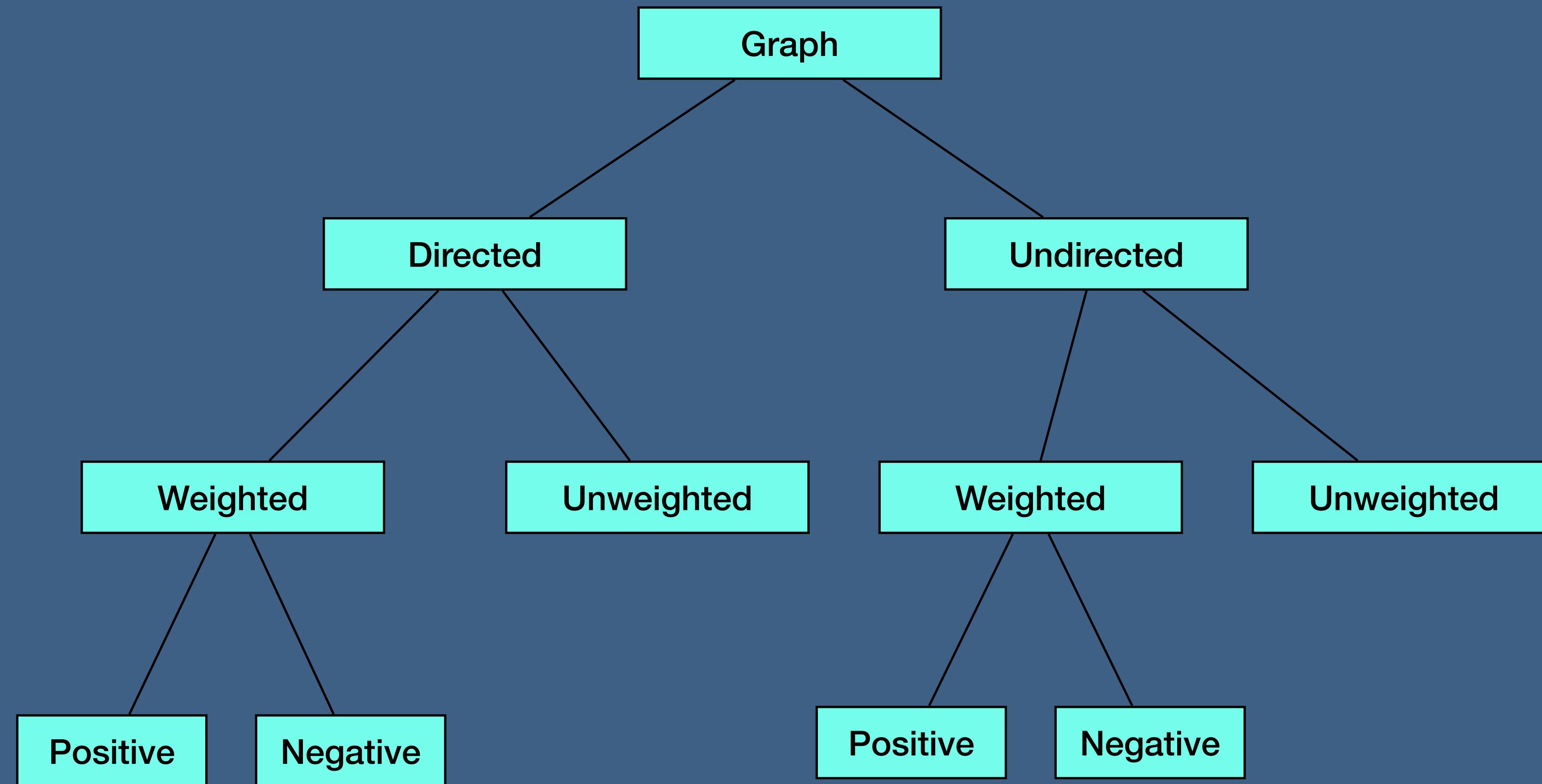


# Graph Terminology

- **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

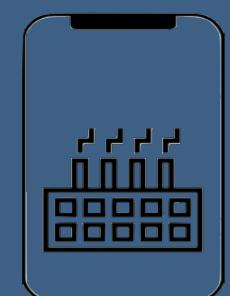
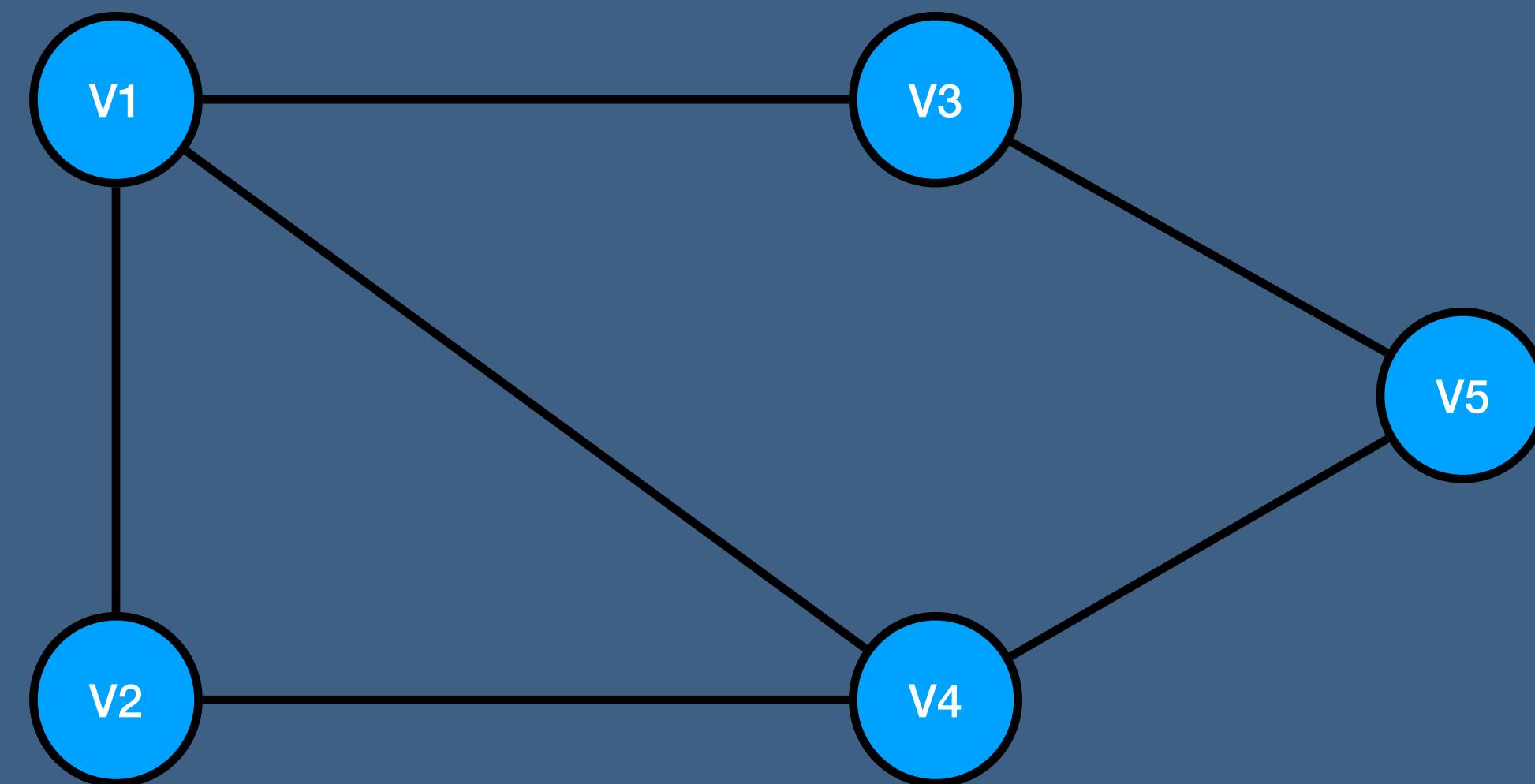


# Graph Types



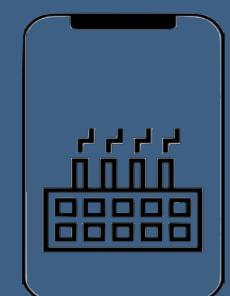
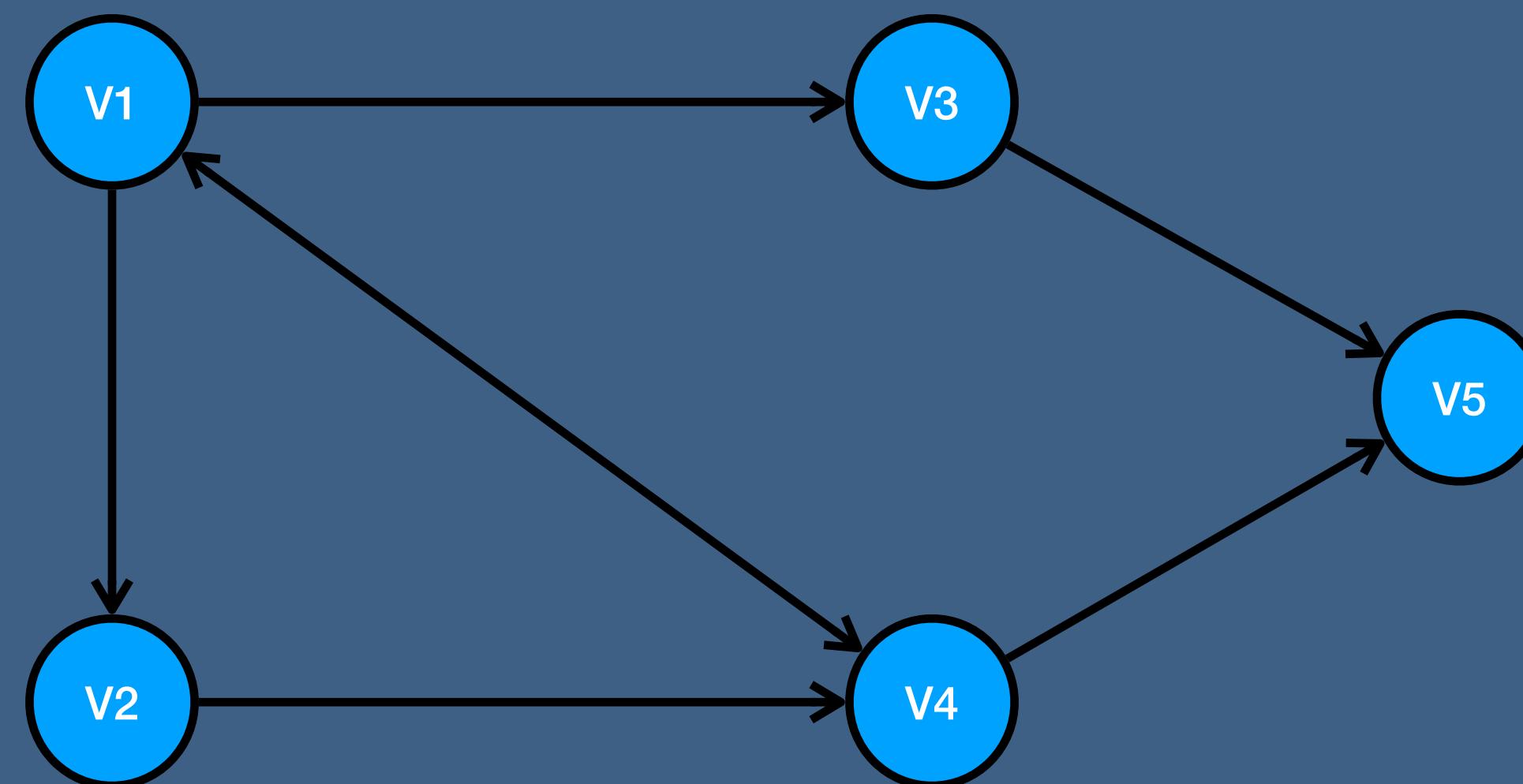
# Graph Types

## 1. Unweighted - undirected



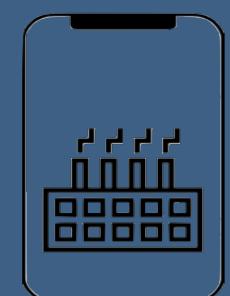
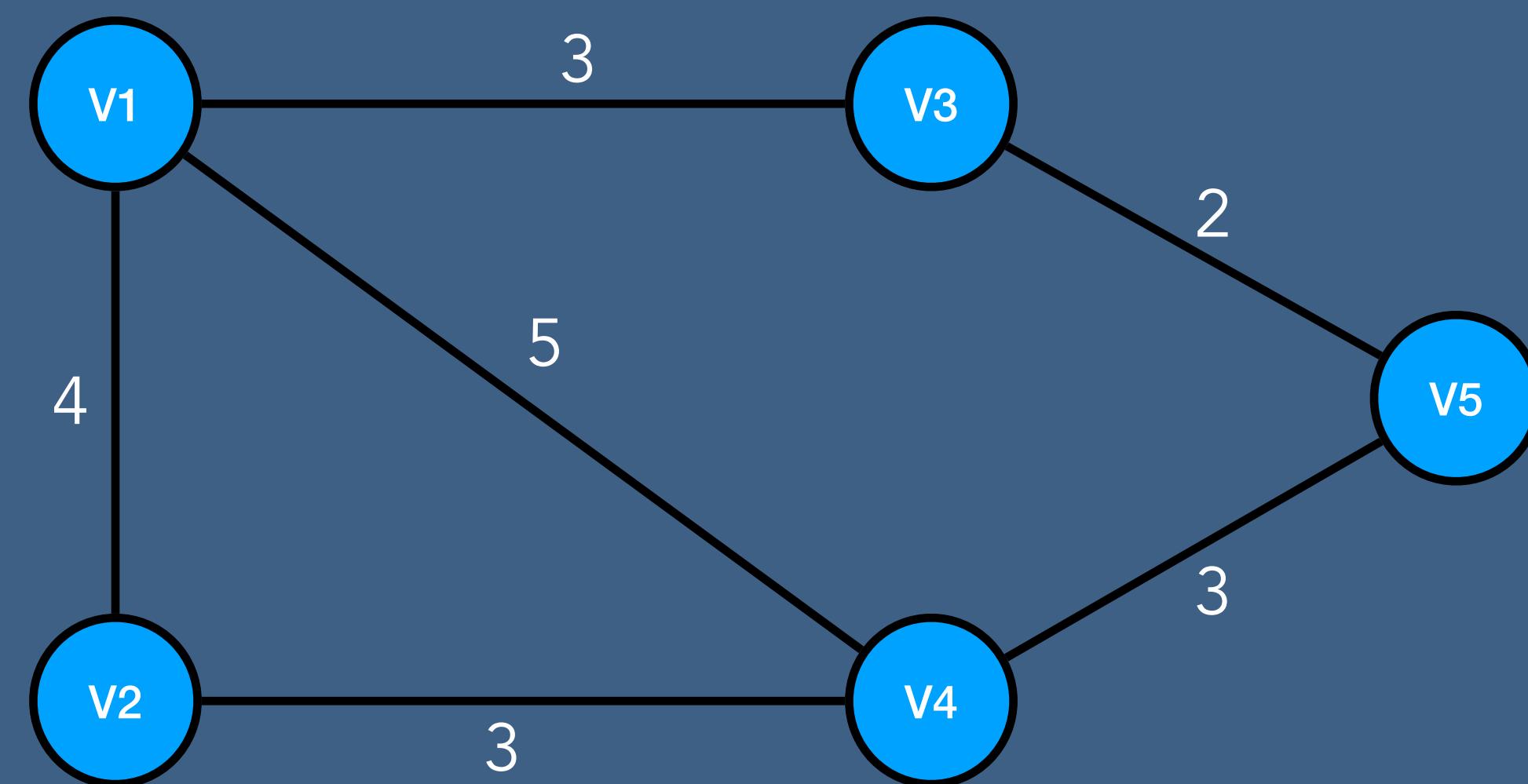
# Graph Types

1. Unweighted - undirected
2. Unweighted - directed



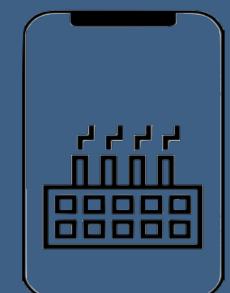
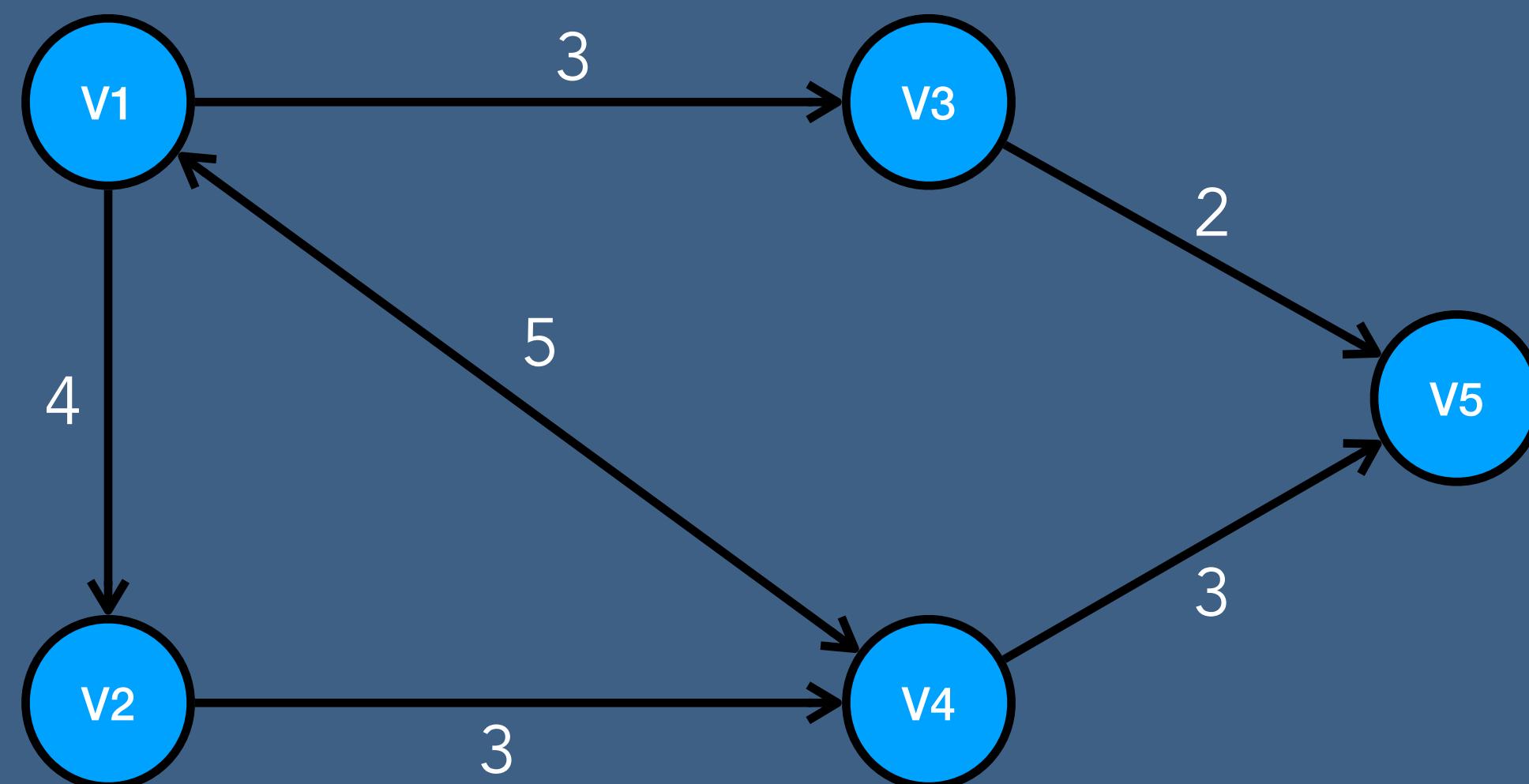
# Graph Types

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



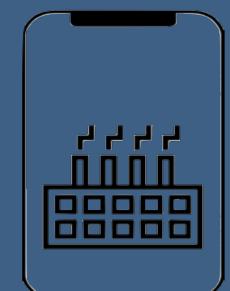
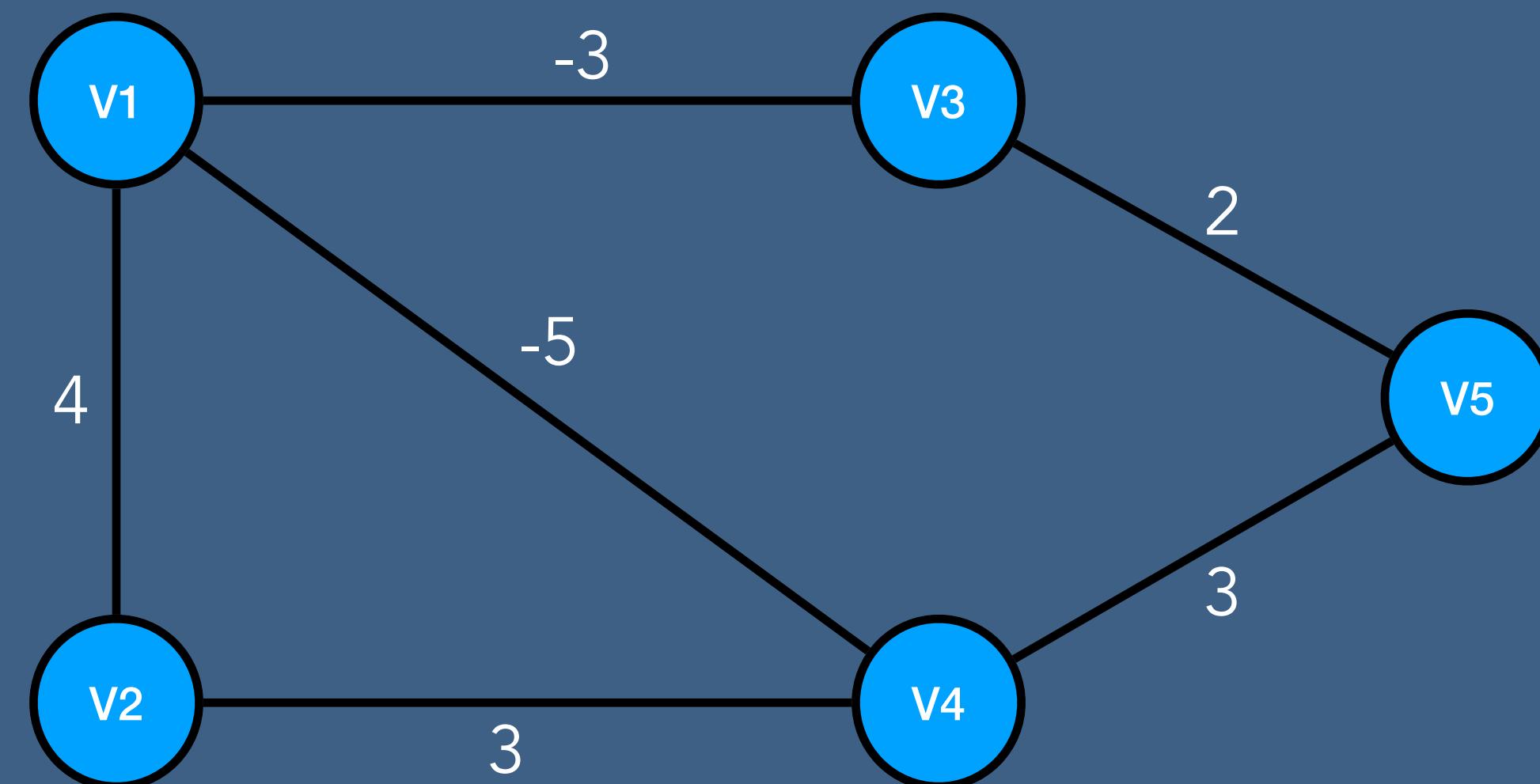
# Graph Types

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



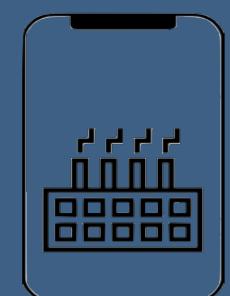
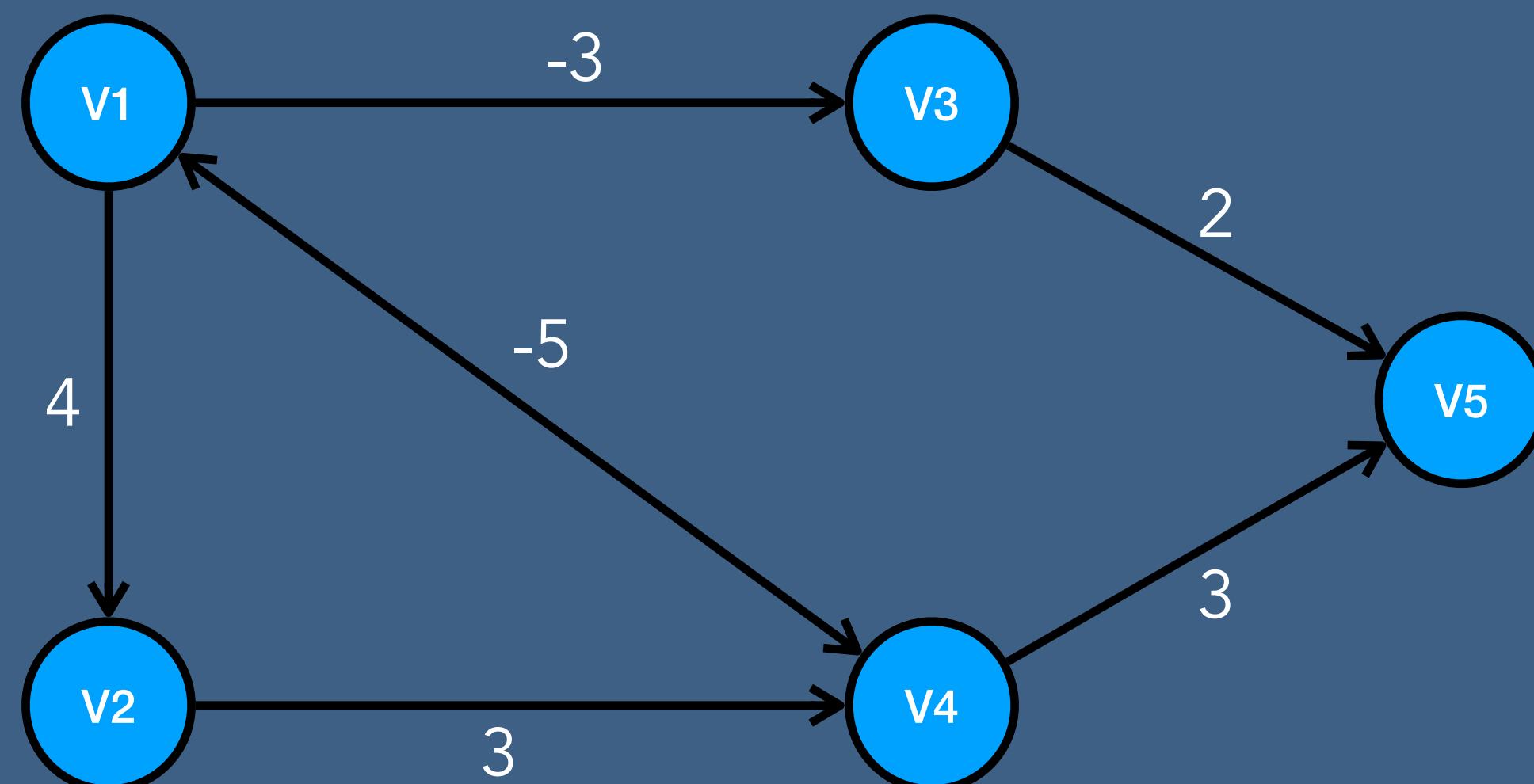
# Graph Types

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



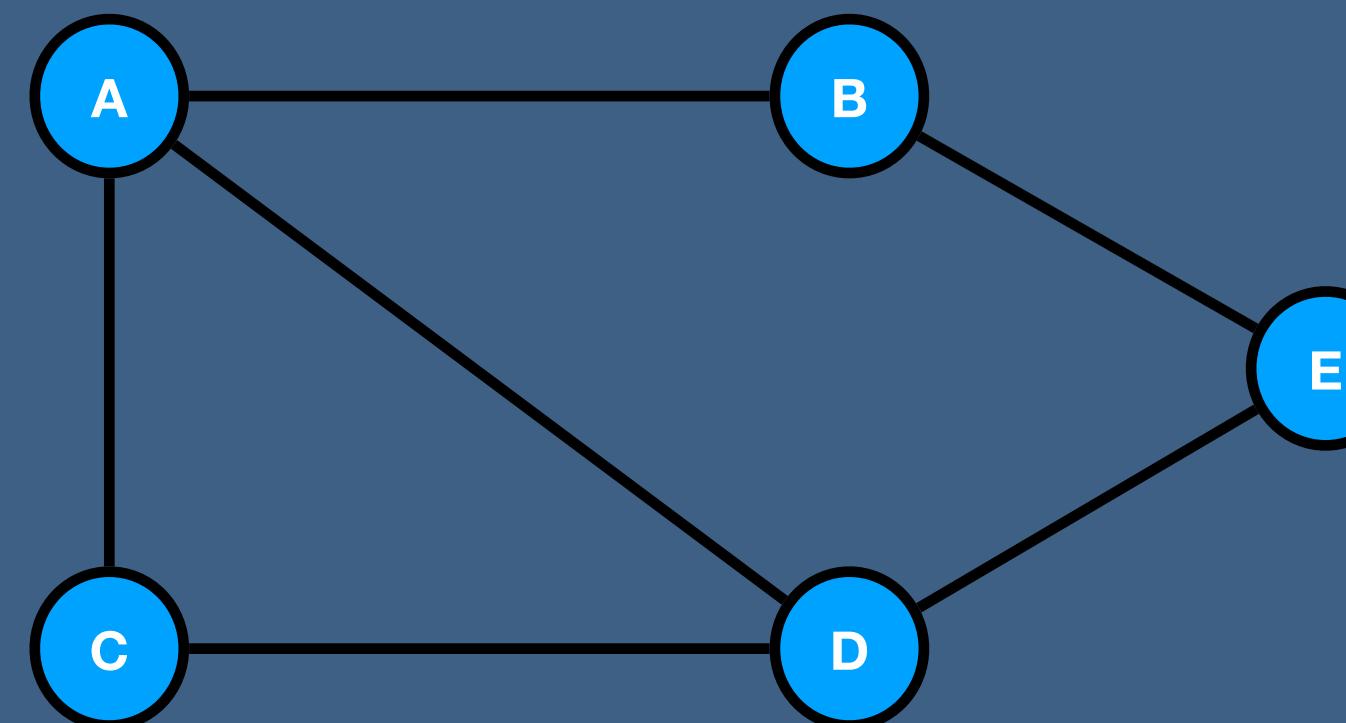
# Graph Types

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

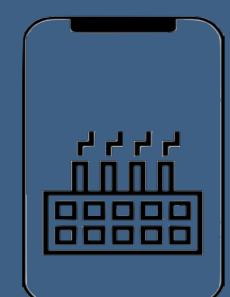


# Graph Representation

**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

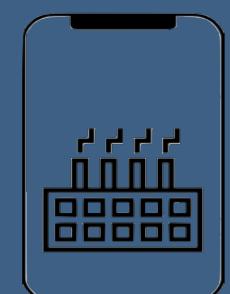
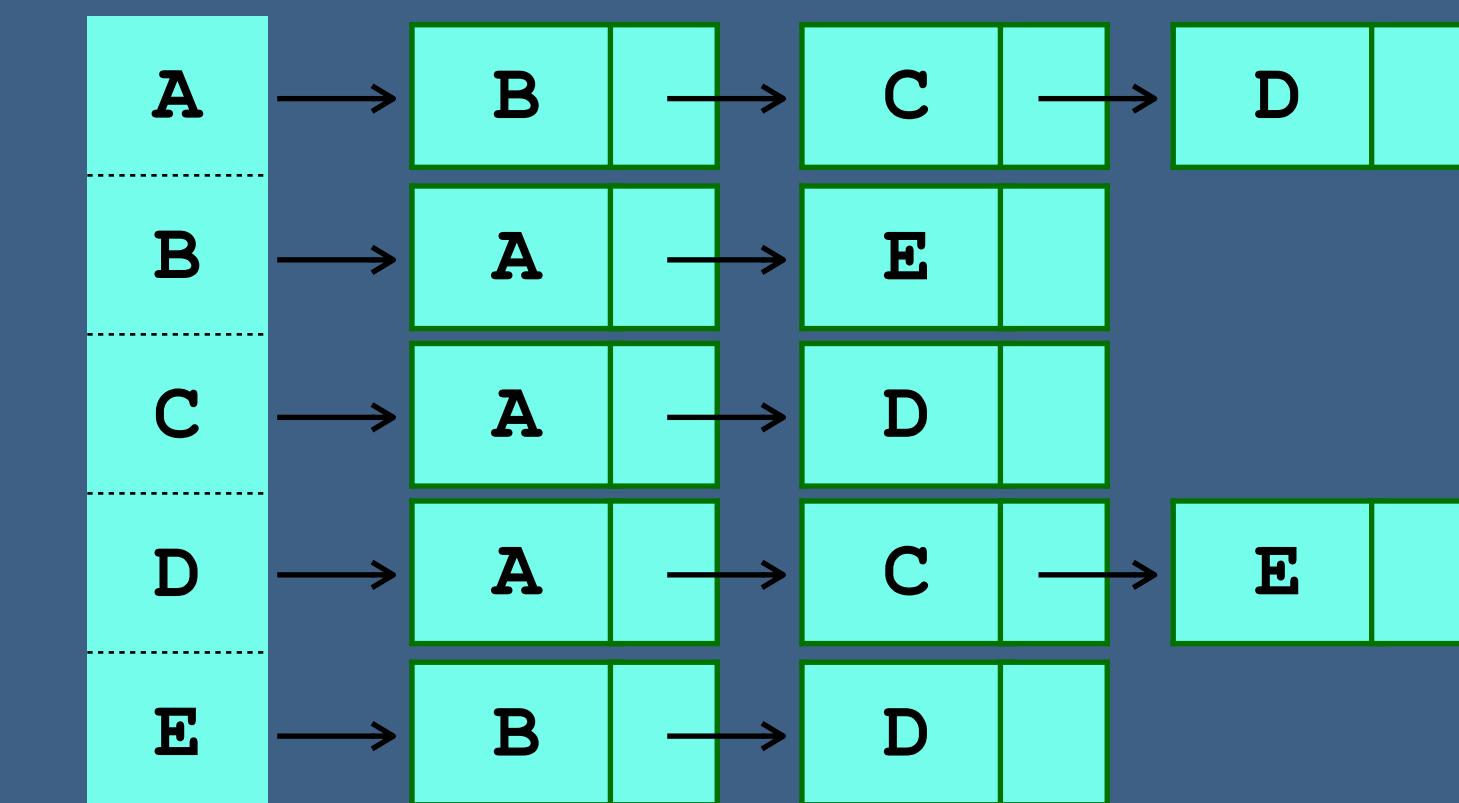
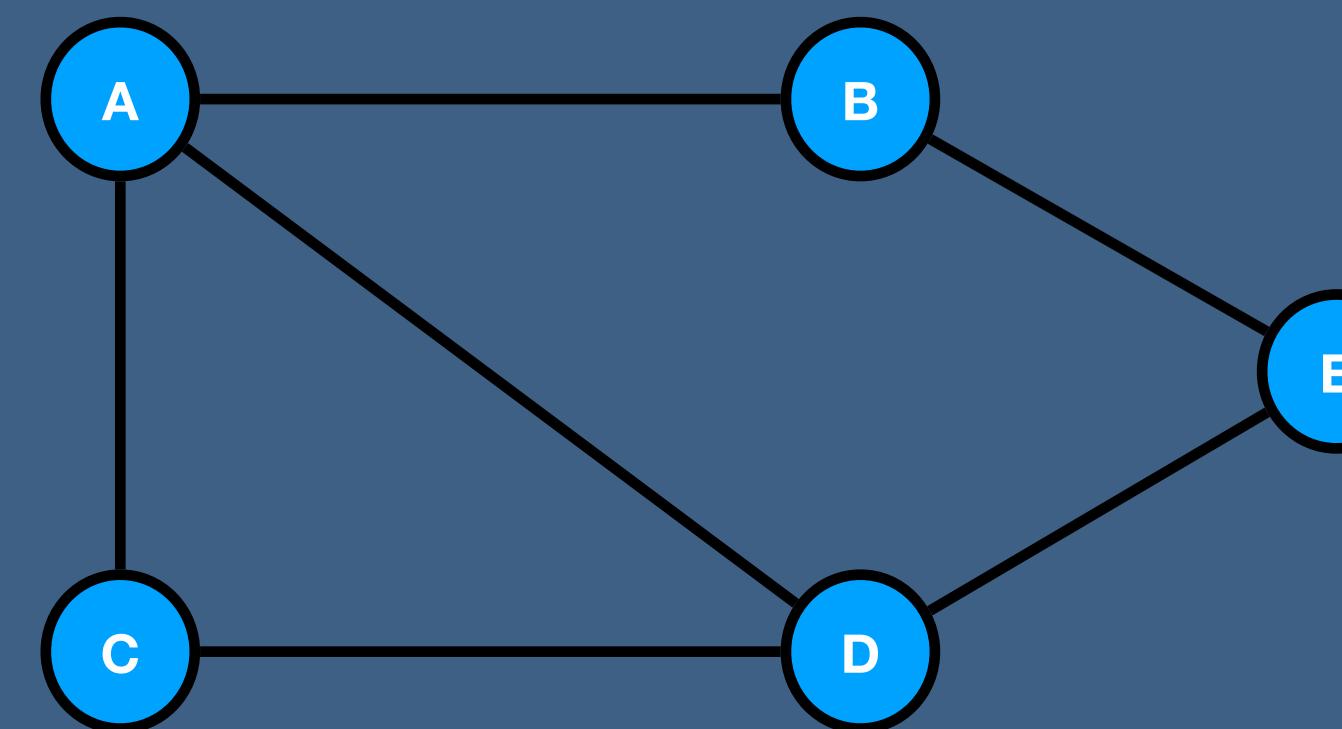


	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	0	1
C	1	0	0	1	0
D	1	0	1	0	1
E	0	1	0	1	0



# 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.

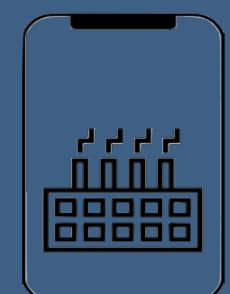
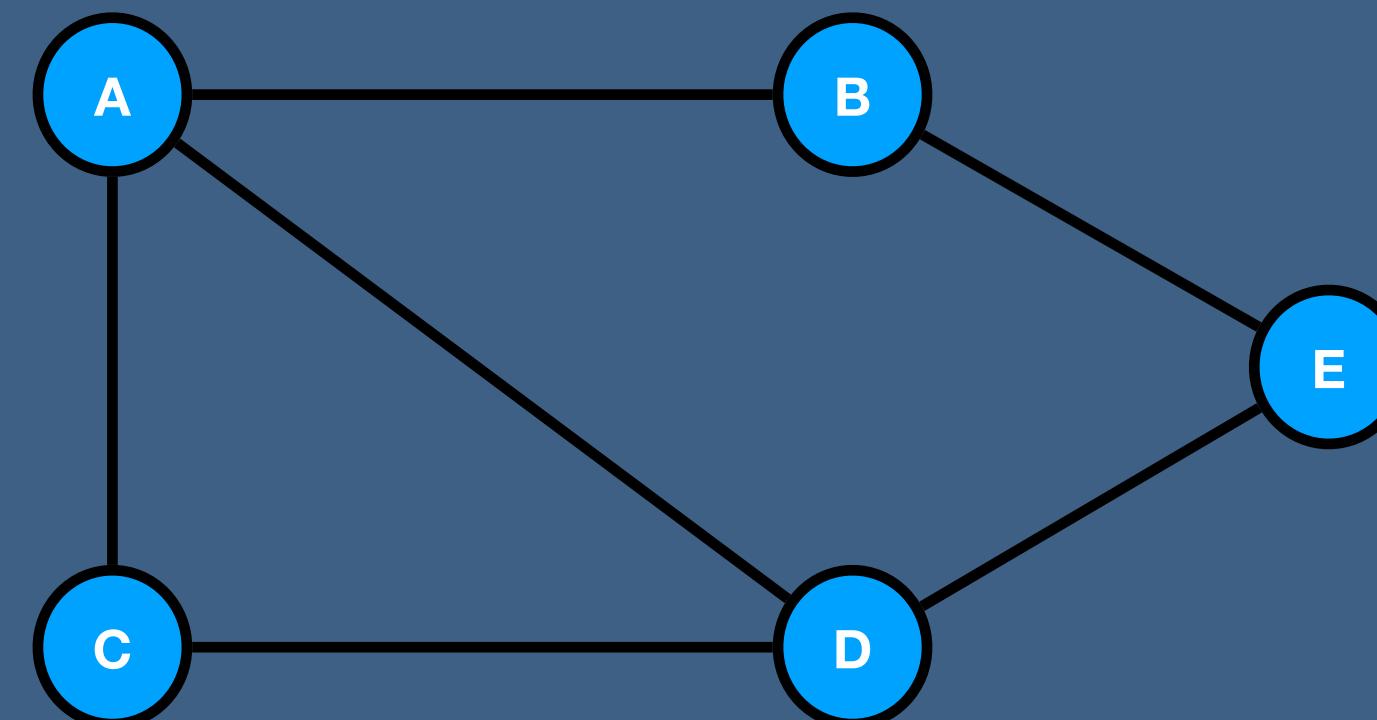
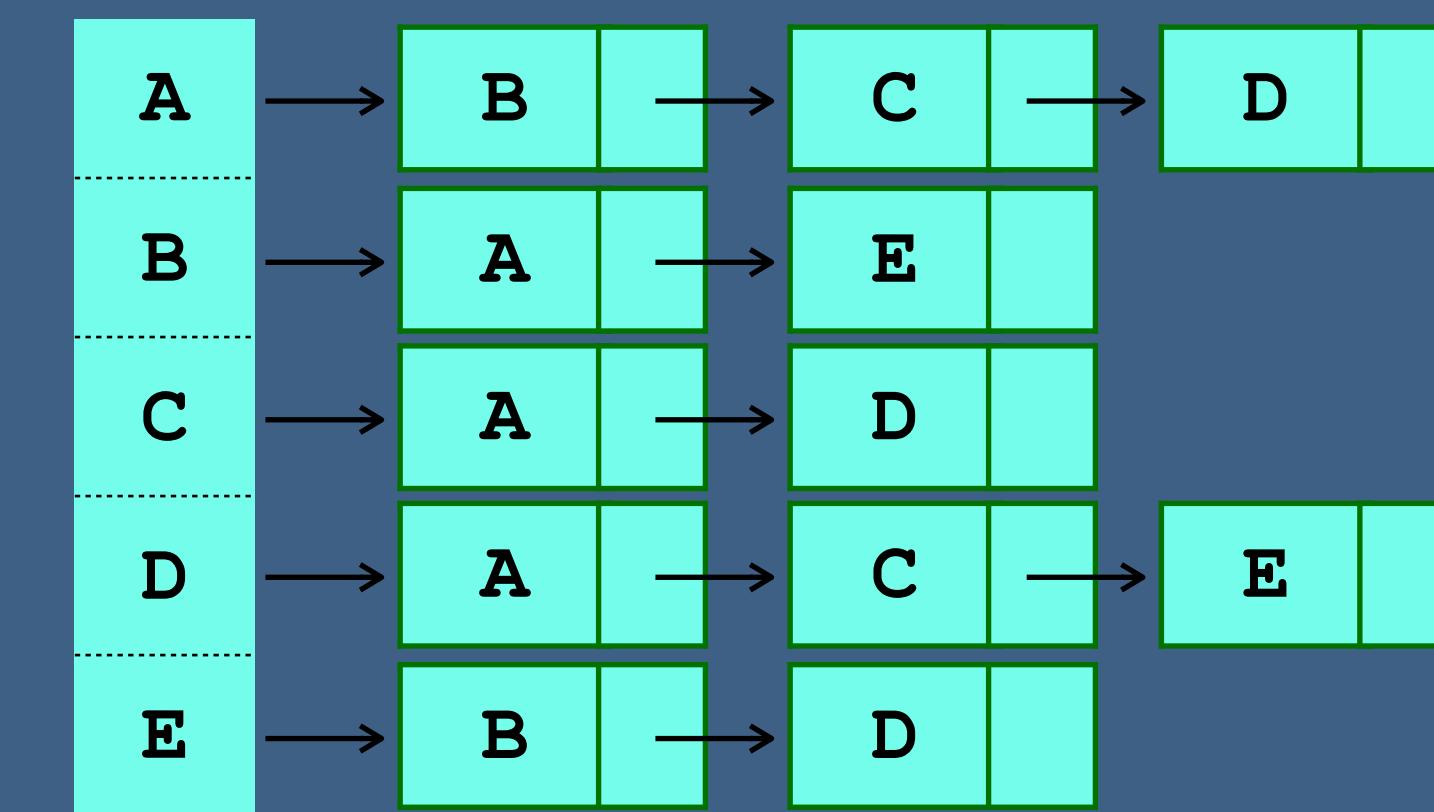


# Graph Representation

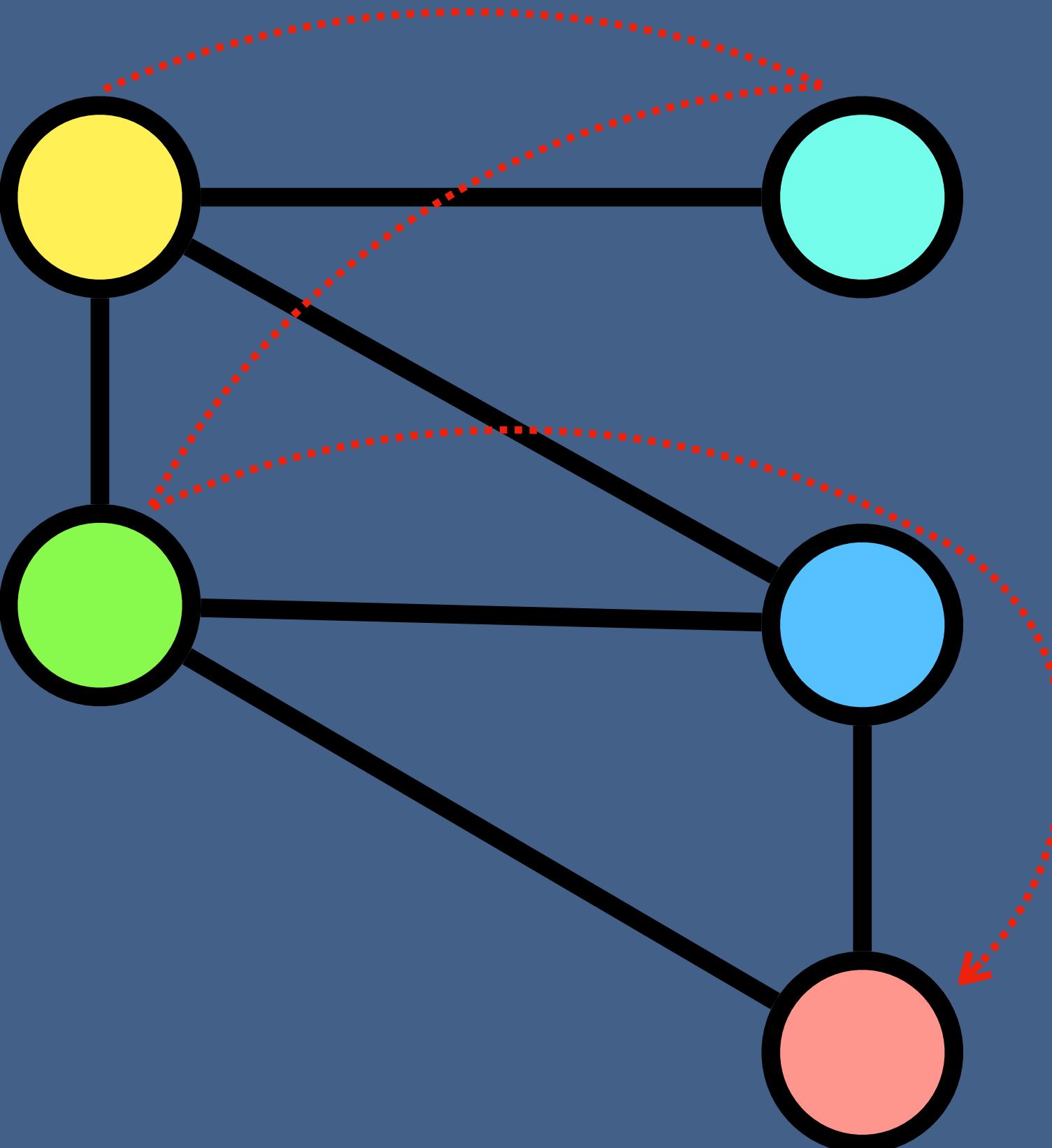
If a graph is complete or almost complete we should use **Adjacency Matrix**

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

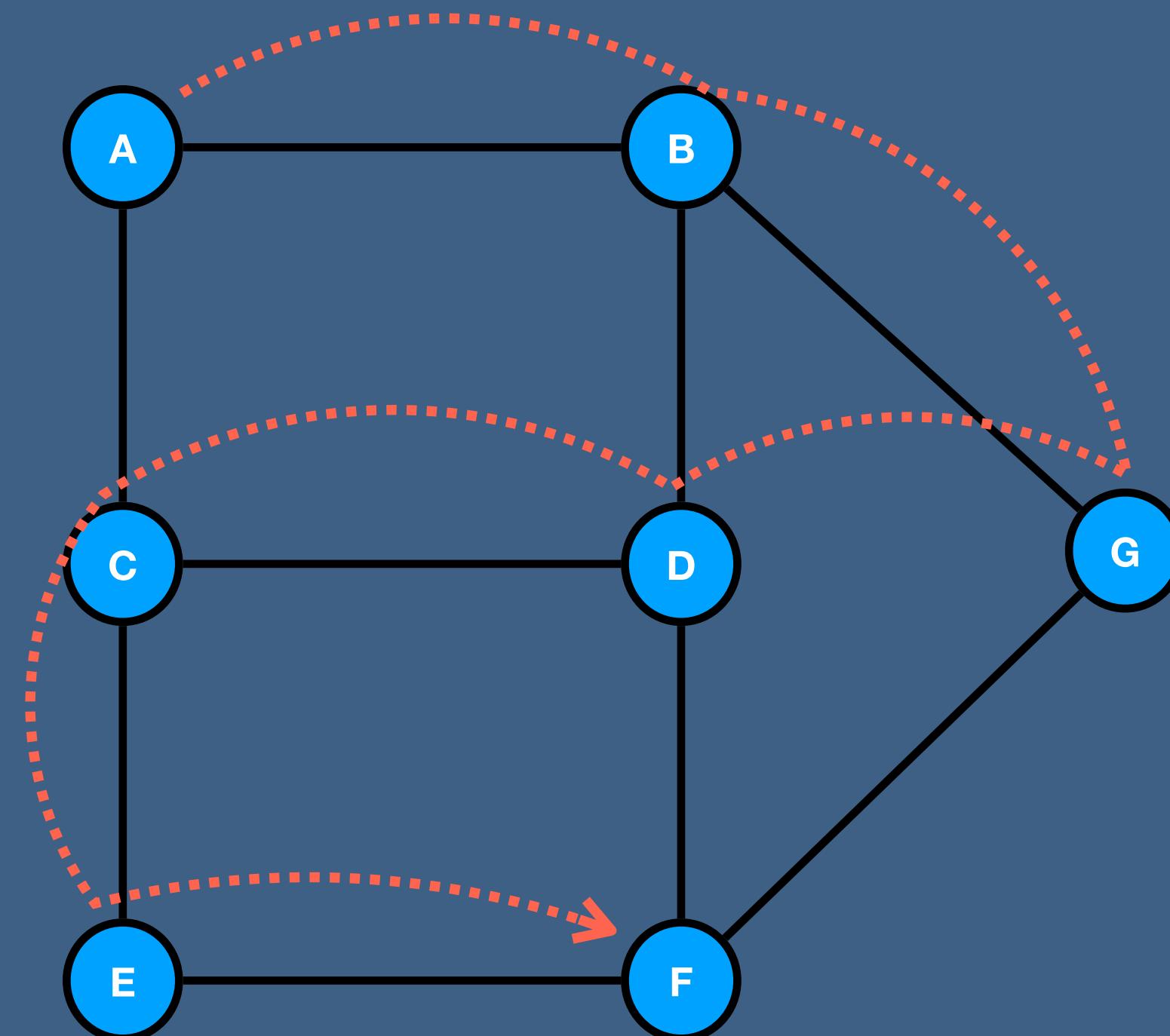


# Graph Traversal

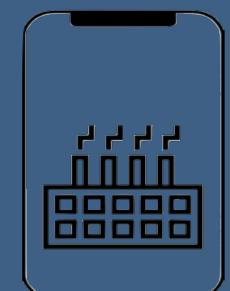


# Graph Traversal

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

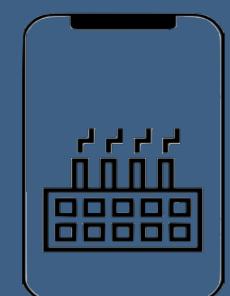
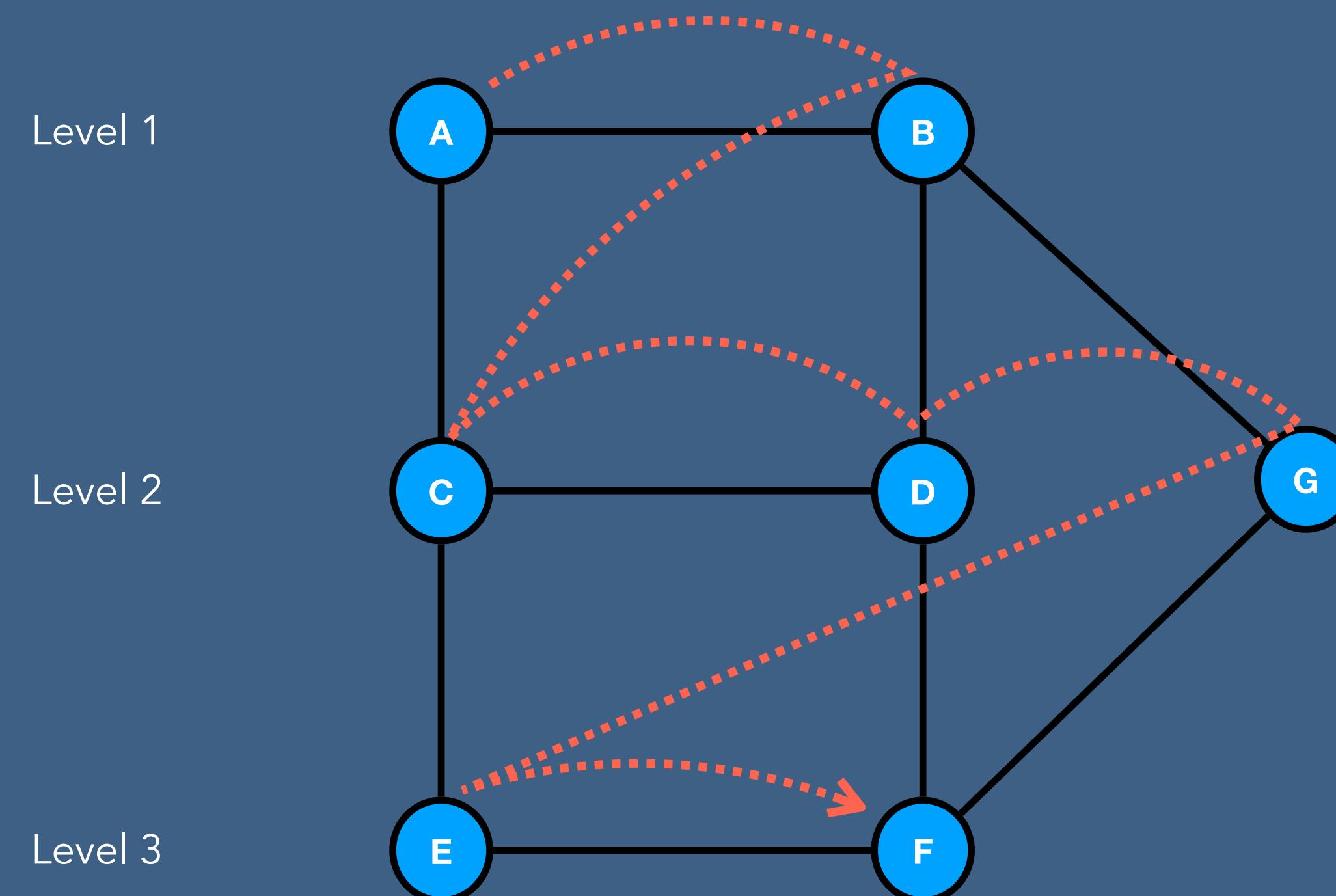


- Breadth First Search
- Depth First Search



# Breadth First Search (BFS)

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.



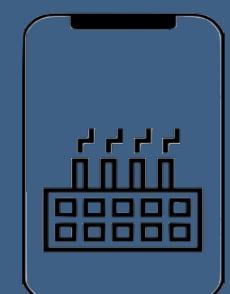
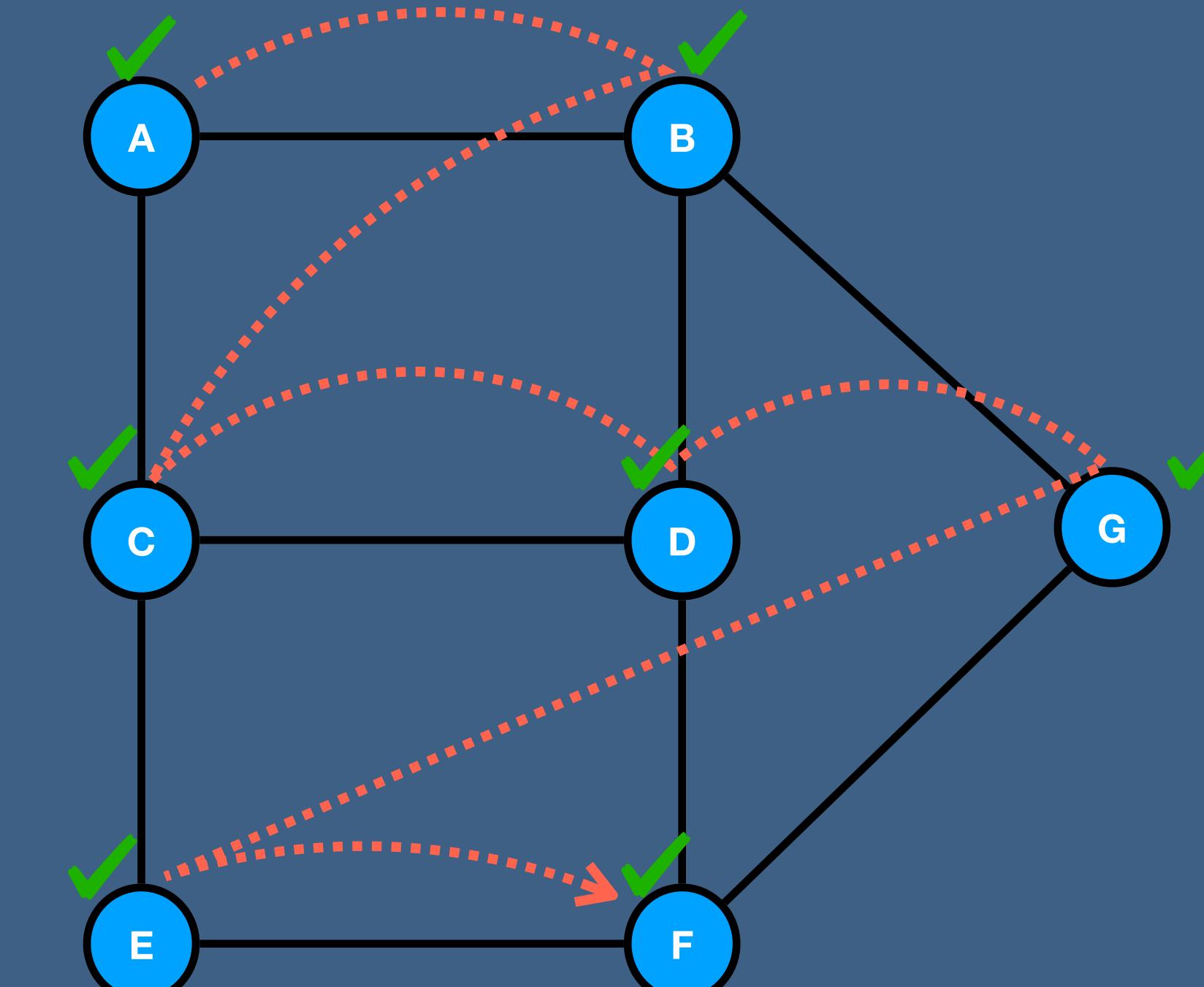
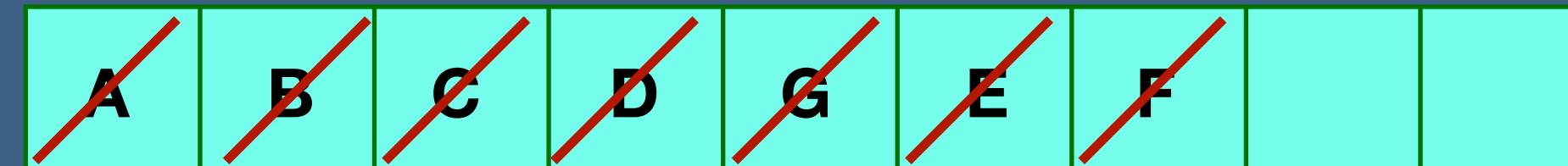
# Breadth First Search Algorithm

**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



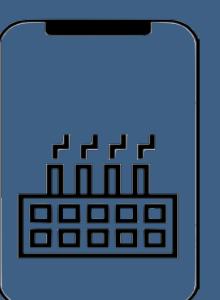
# Time and Space Complexity of BFS

## BFS

```
while all vertices are not explored ..... ➔ O(V)
    enqueue any starting vertex ..... ➔ O(1)
    while queue is not empty ..... ➔ O(V)
        p = dequeue() ..... ➔ O(1)
        if p is unvisited ..... ➔ O(1)
            mark it visited ..... ➔ O(1)
            enqueue unvisited adjacent vertices of p ..... ➔ O(Adj) } O(Adj) } O(E)
```

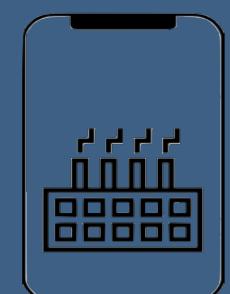
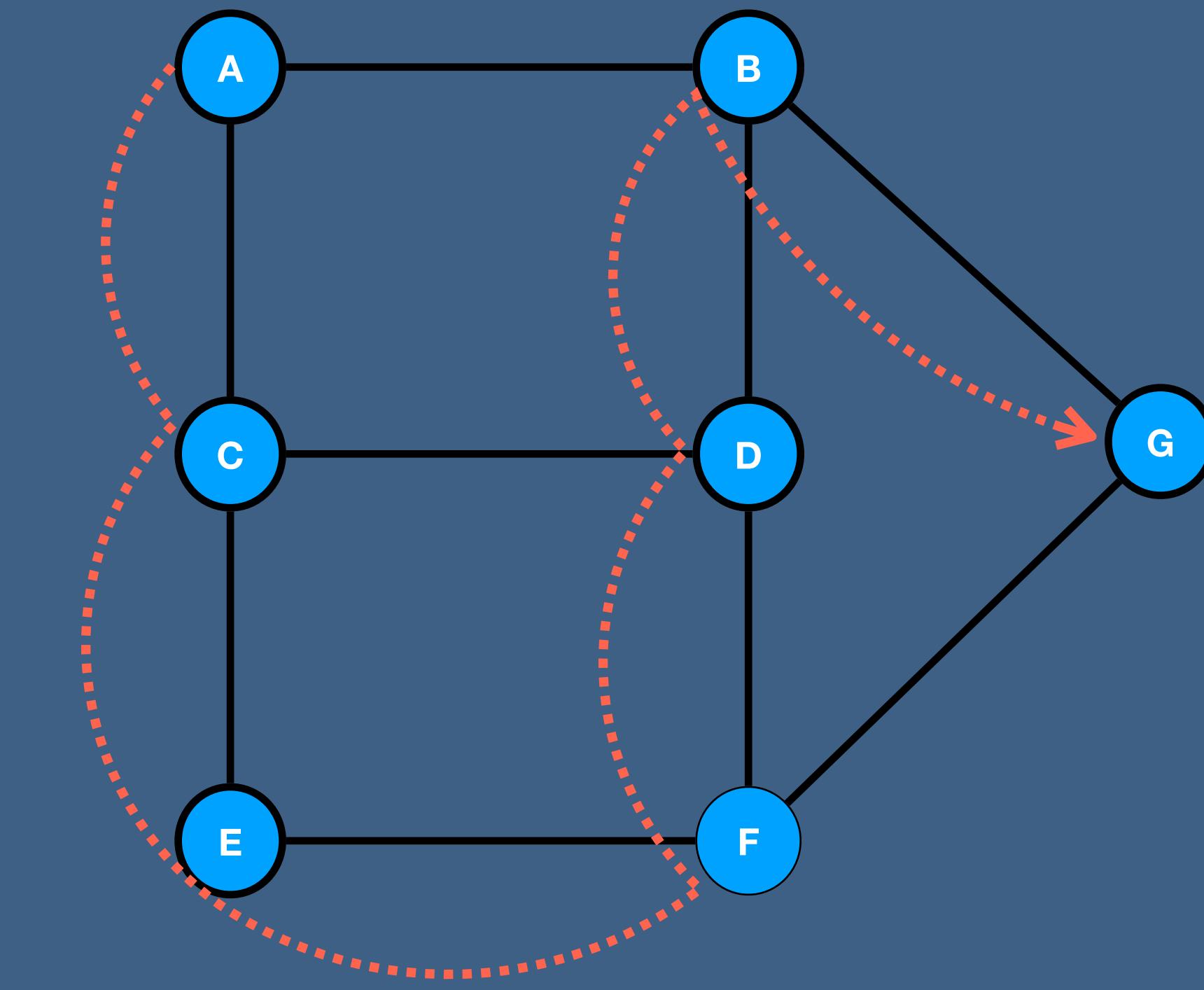
Time Complexity :  $O(V+E)$

Space Complexity :  $O(V+E)$



# Depth First Search (DFS)

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



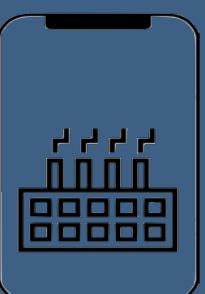
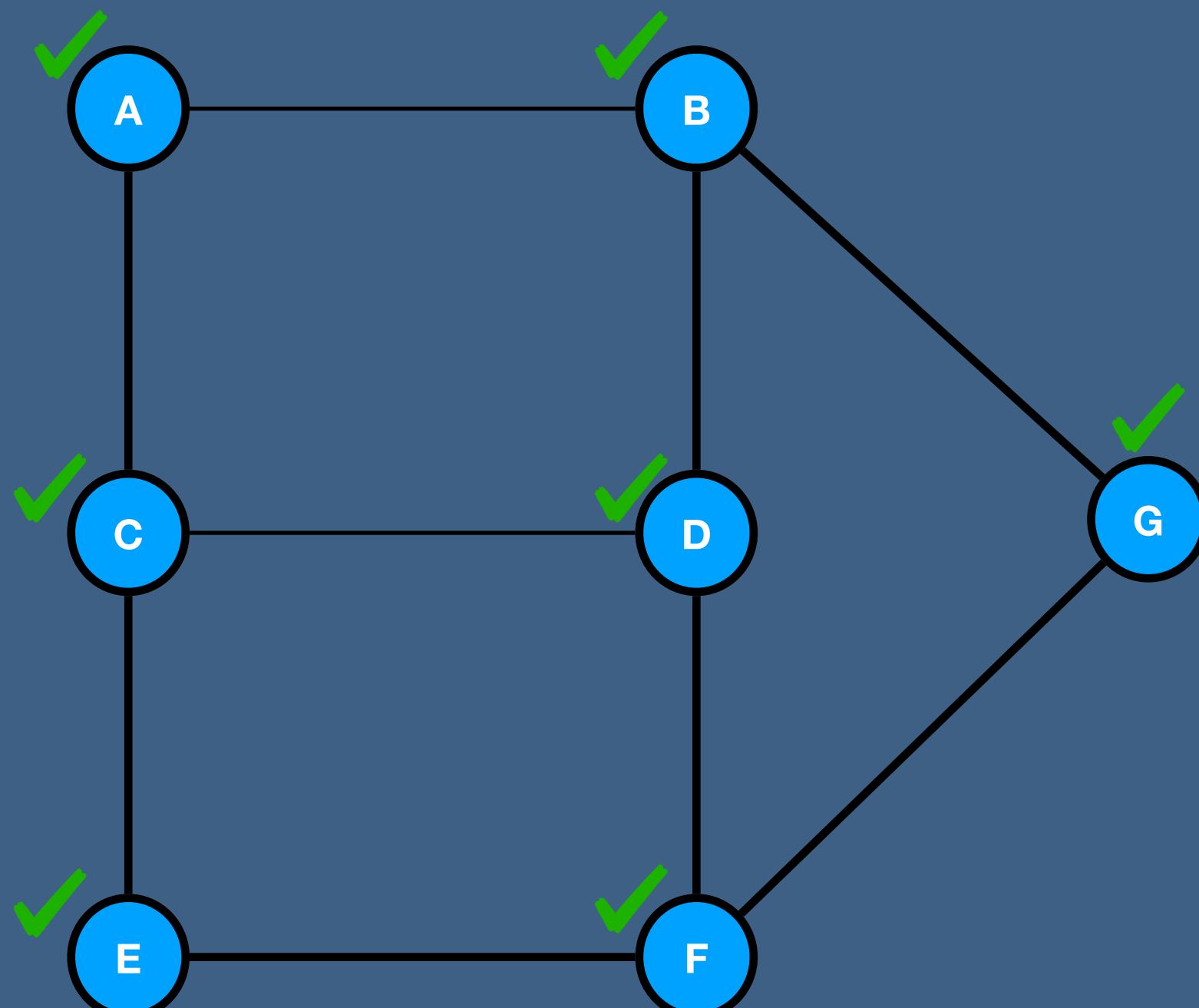
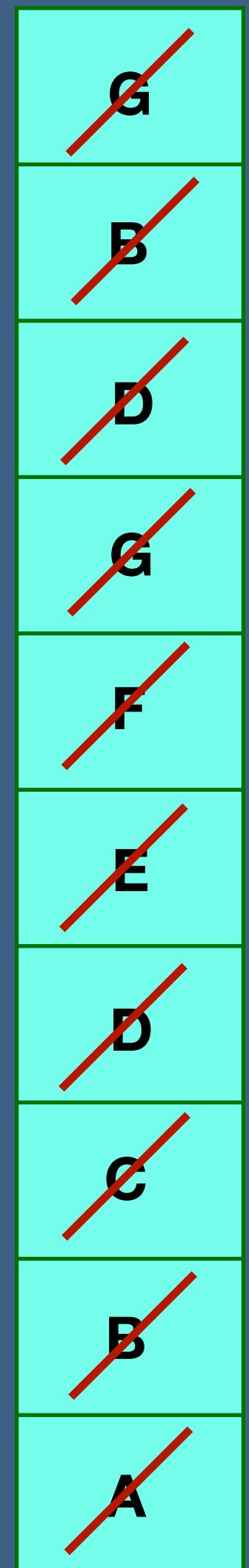
# Depth First Search Algorithm

## 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



# Time and Space Complexity of DFS

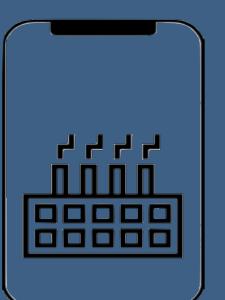
## DFS

```
while all vertices are not explored ..... ➔ O(V)
    push any starting vertex ..... ➔ O(1)
    while stack is not empty ..... ➔ O(V)
        p = pop() ..... ➔ O(1)
        if p is unvisited ..... ➔ O(1)
            mark it visited ..... ➔ O(1)
            push unvisited adjacent vertices of p ..... ➔ O(Adj)
```

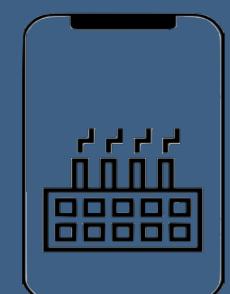
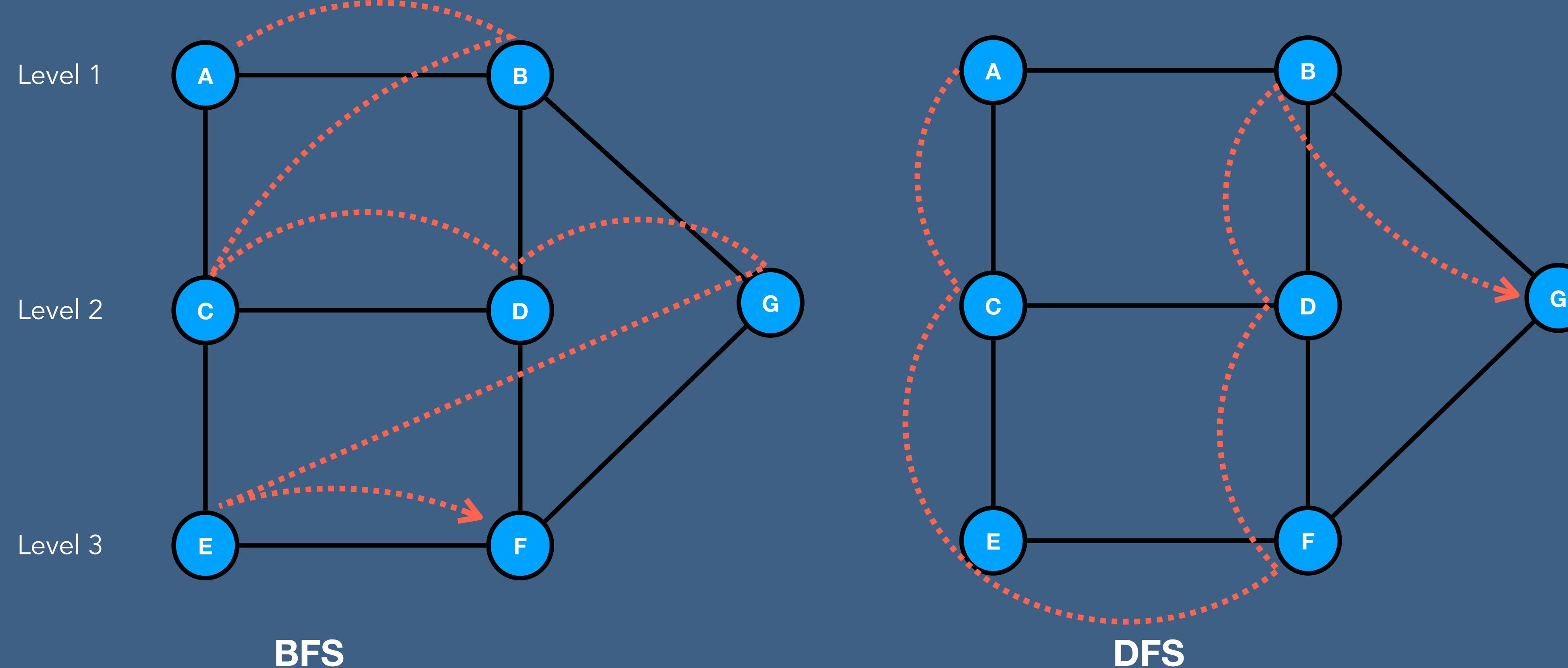
$\left. \begin{matrix} O(1) \\ O(1) \\ O(1) \end{matrix} \right\} O(Adj) \left. \begin{matrix} O(V) \\ O(E) \end{matrix} \right\}$

Time Complexity :  $O(V+E)$

Space Complexity :  $O(V+E)$

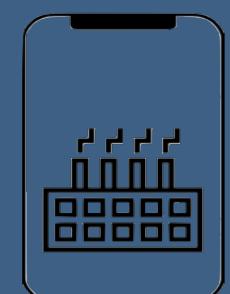


# BFS vs DFS

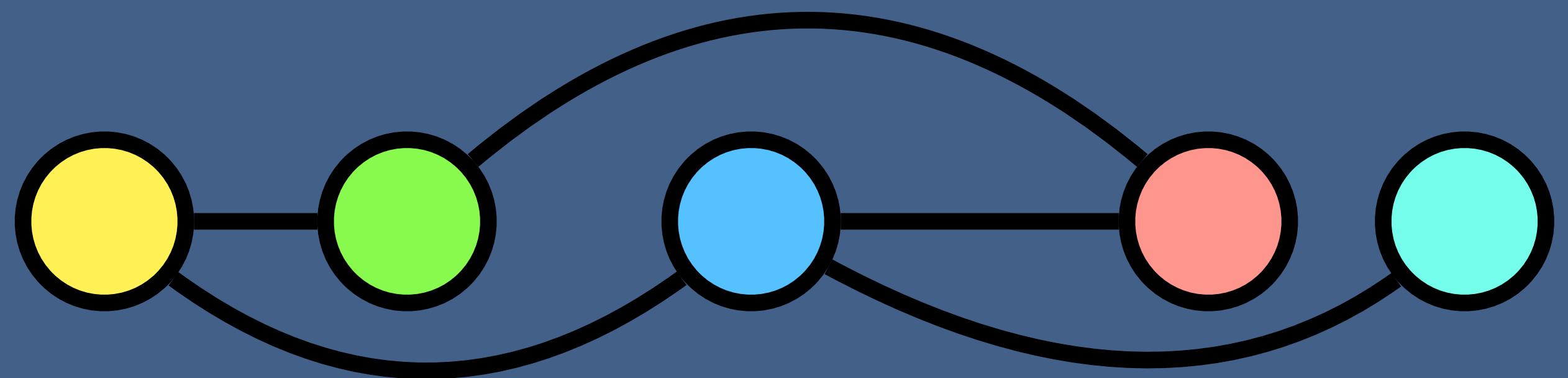


# BFS vs DFS

	<b>BFS</b>	<b>DFS</b>
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

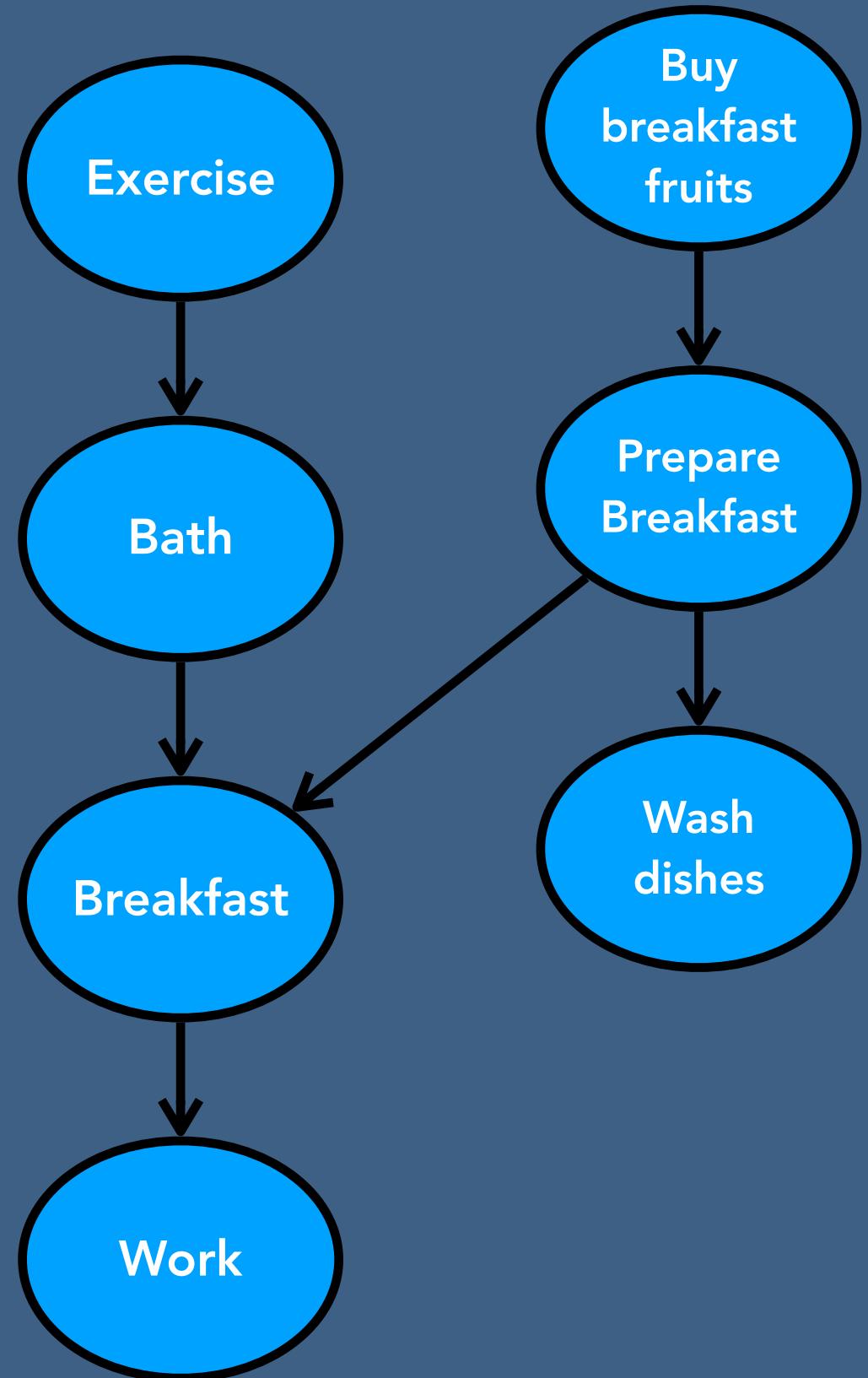


# Topological Sort



# Topological Sort

**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

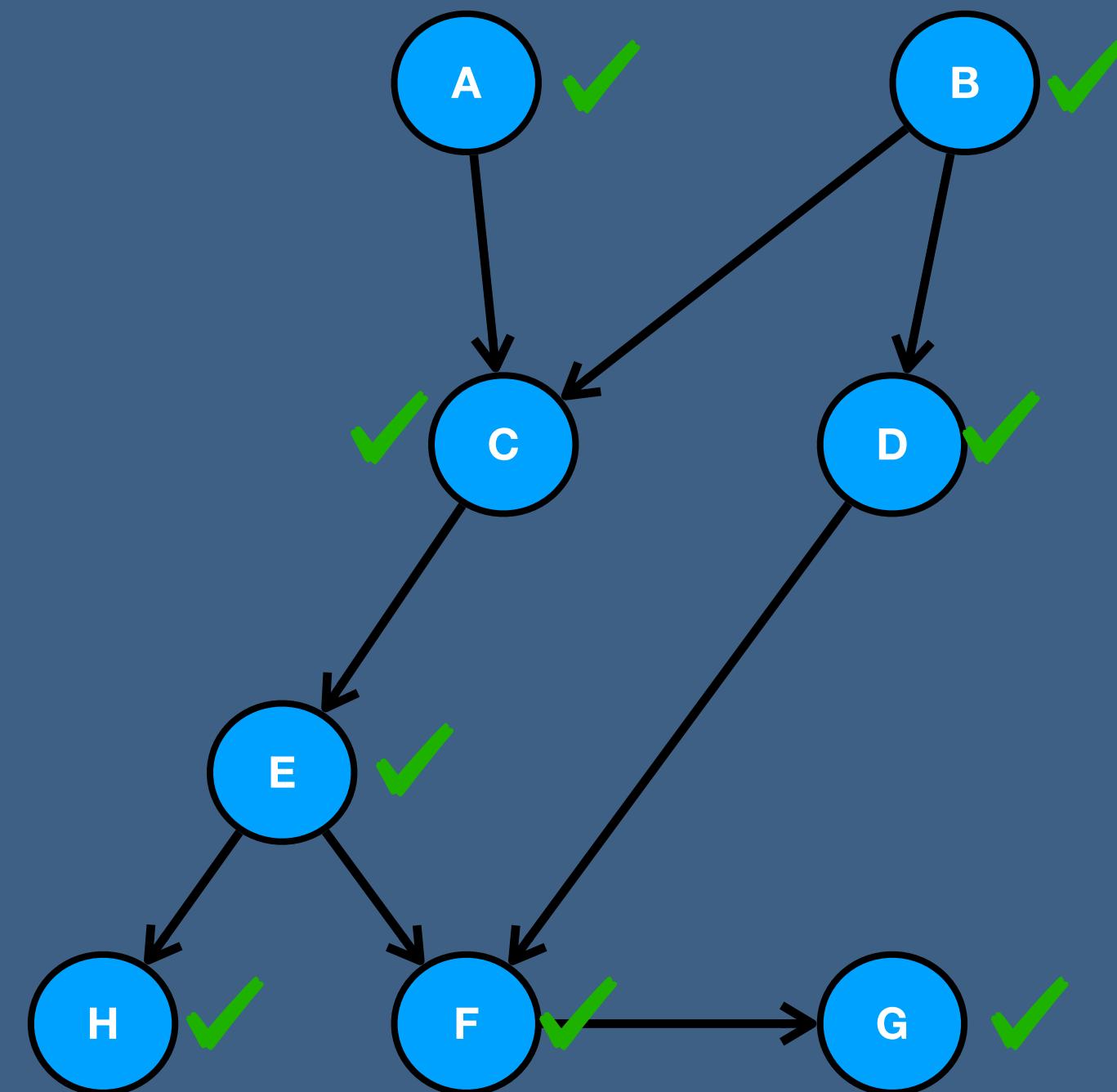
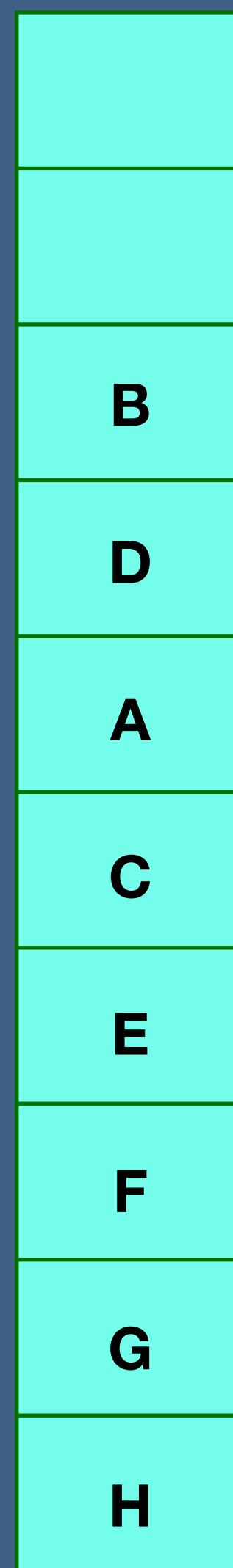
```
if a vertex depends on currentVertex:  
    Go to that vertex and  
    then come back to currentVertex  
else  
    Push currentVertex to Stack
```

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



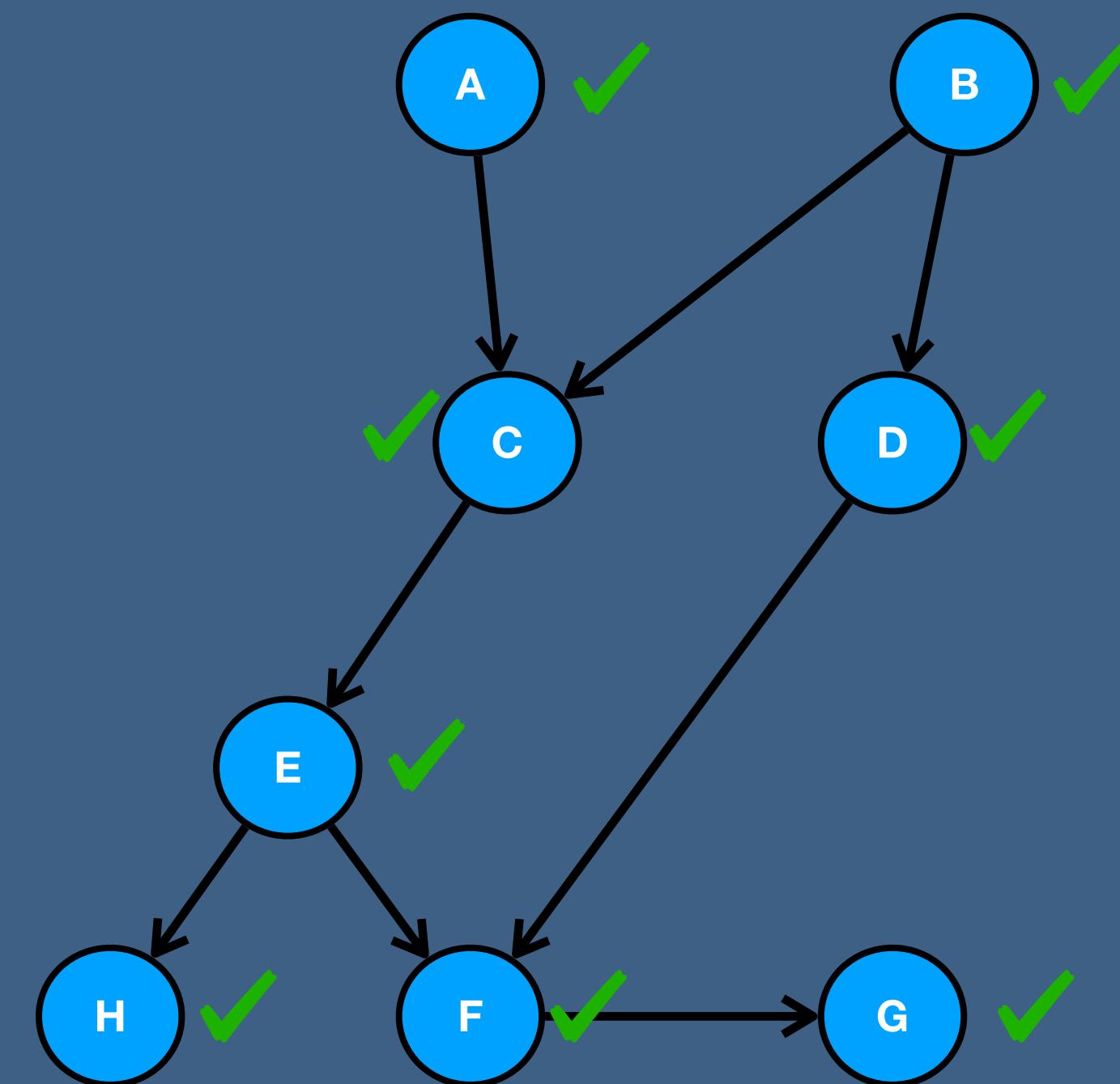
# Time and Space Complexity of Topological Sort

```
topologicalSort(G)
  for all nodes
    if vertex not visited
      topologicalVisit(node)
```

```
topologicalVisit(node)
  for each nodes
    if not visited
      topologicalVisit(neighbor)
```

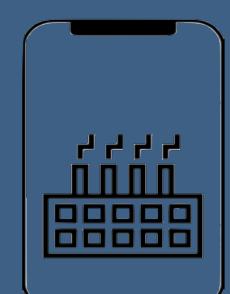
}  $O(V)$

}  $O(E)$

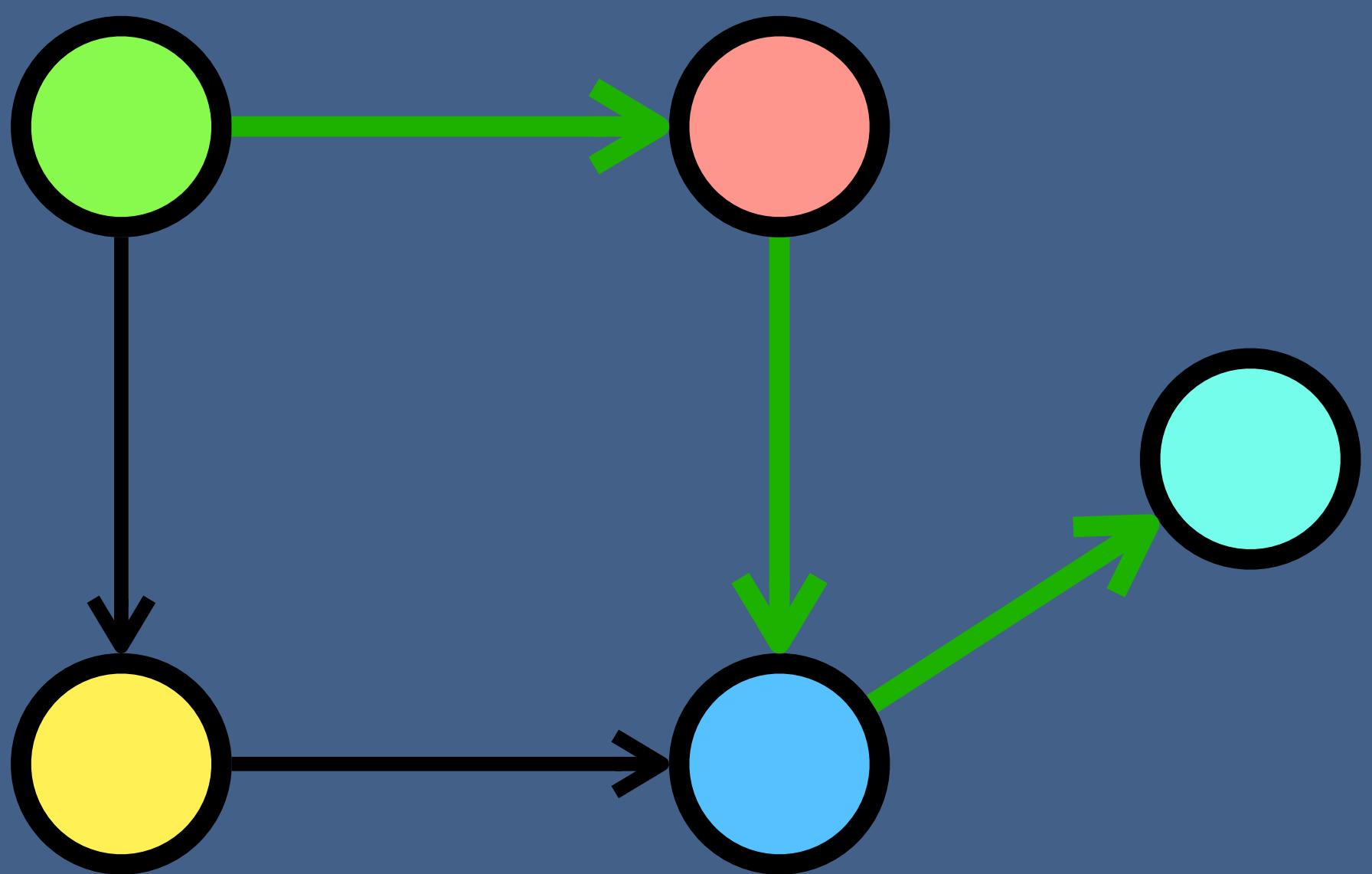


**Time Complexity :  $O(V+E)$**

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



# Single Source Shortest Path Problem (SSSPP)

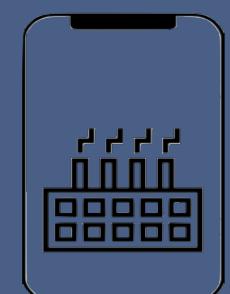
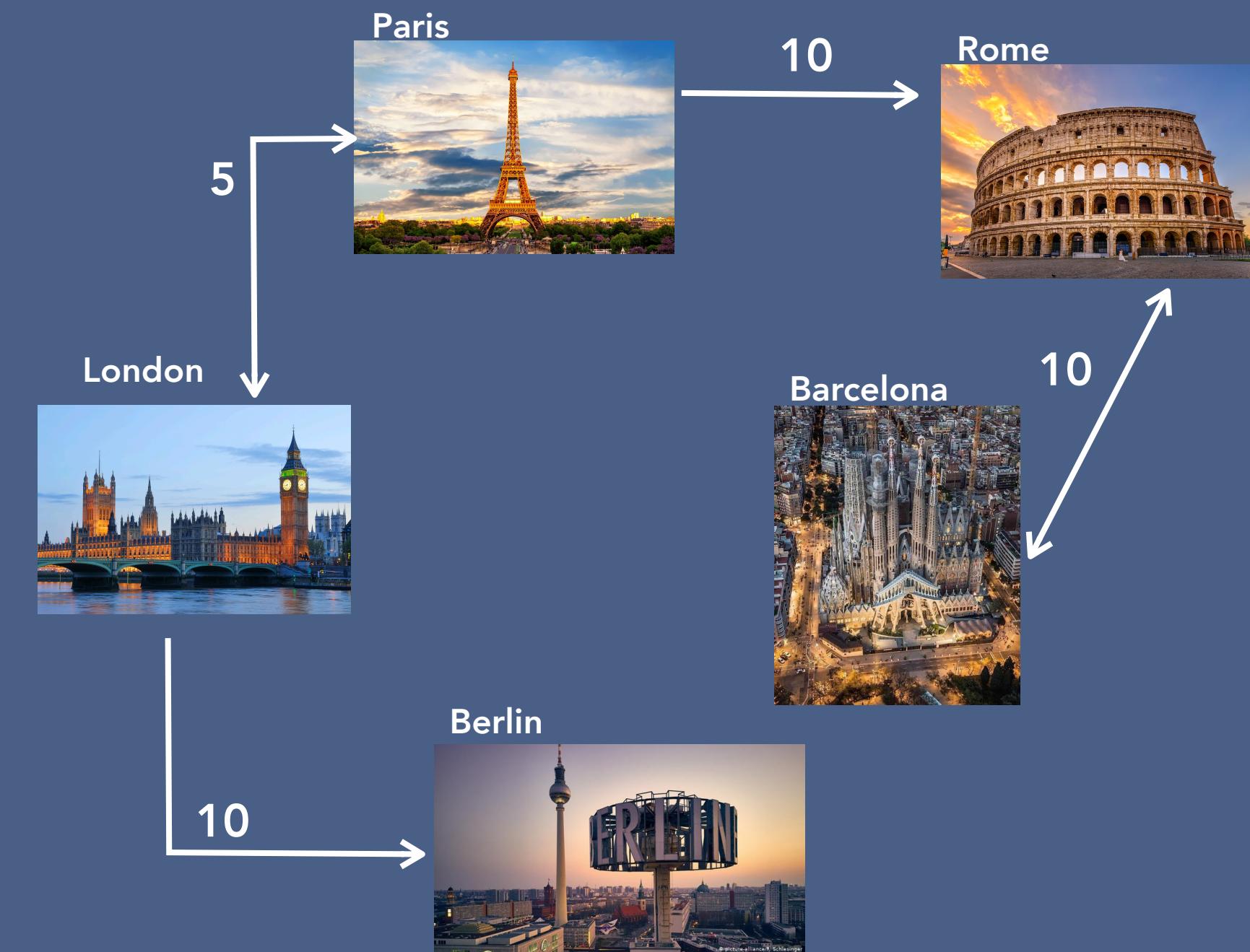
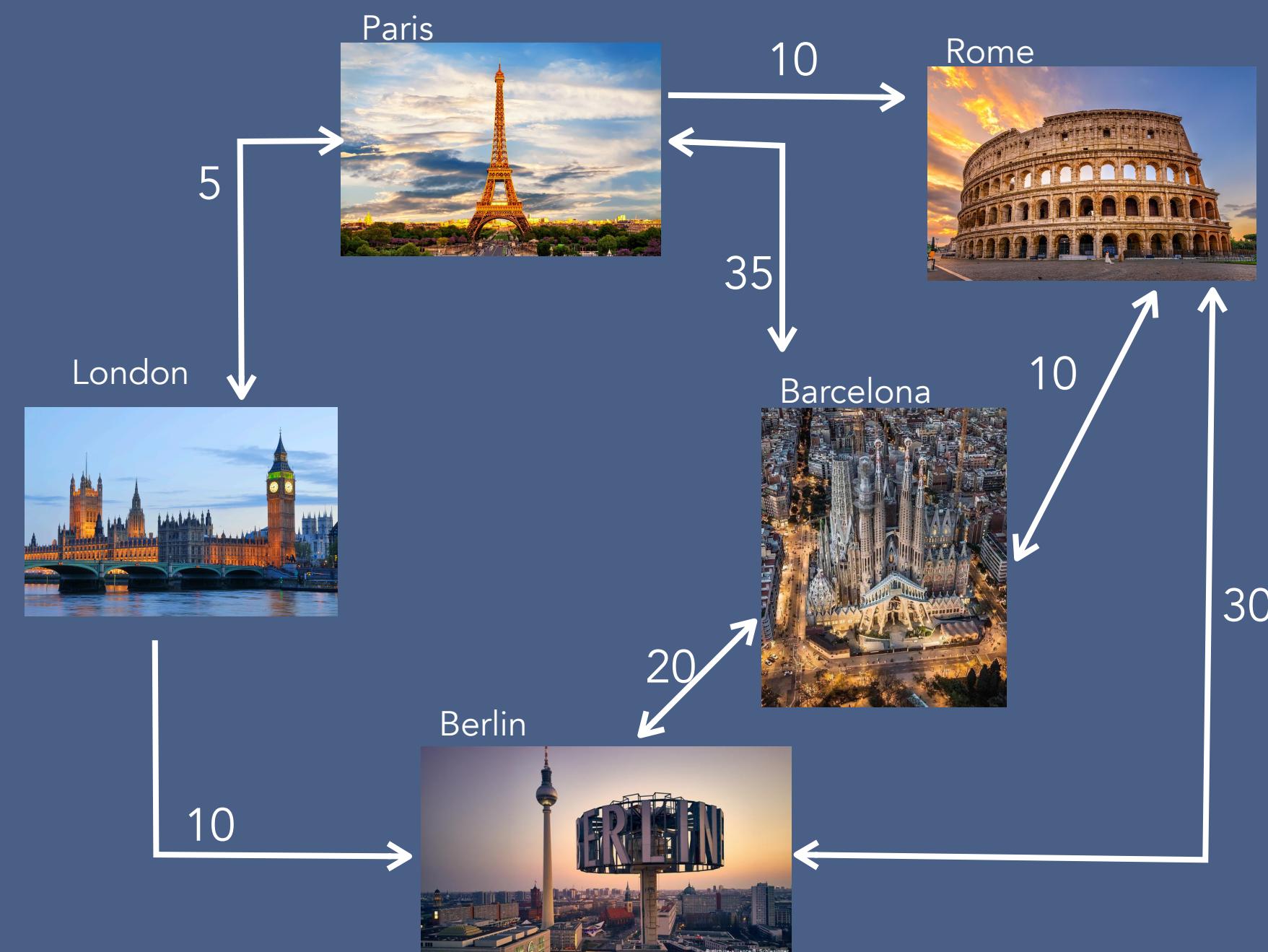


# Single Source Shortest Path Problem (SSSPP)

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:

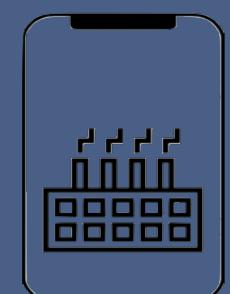
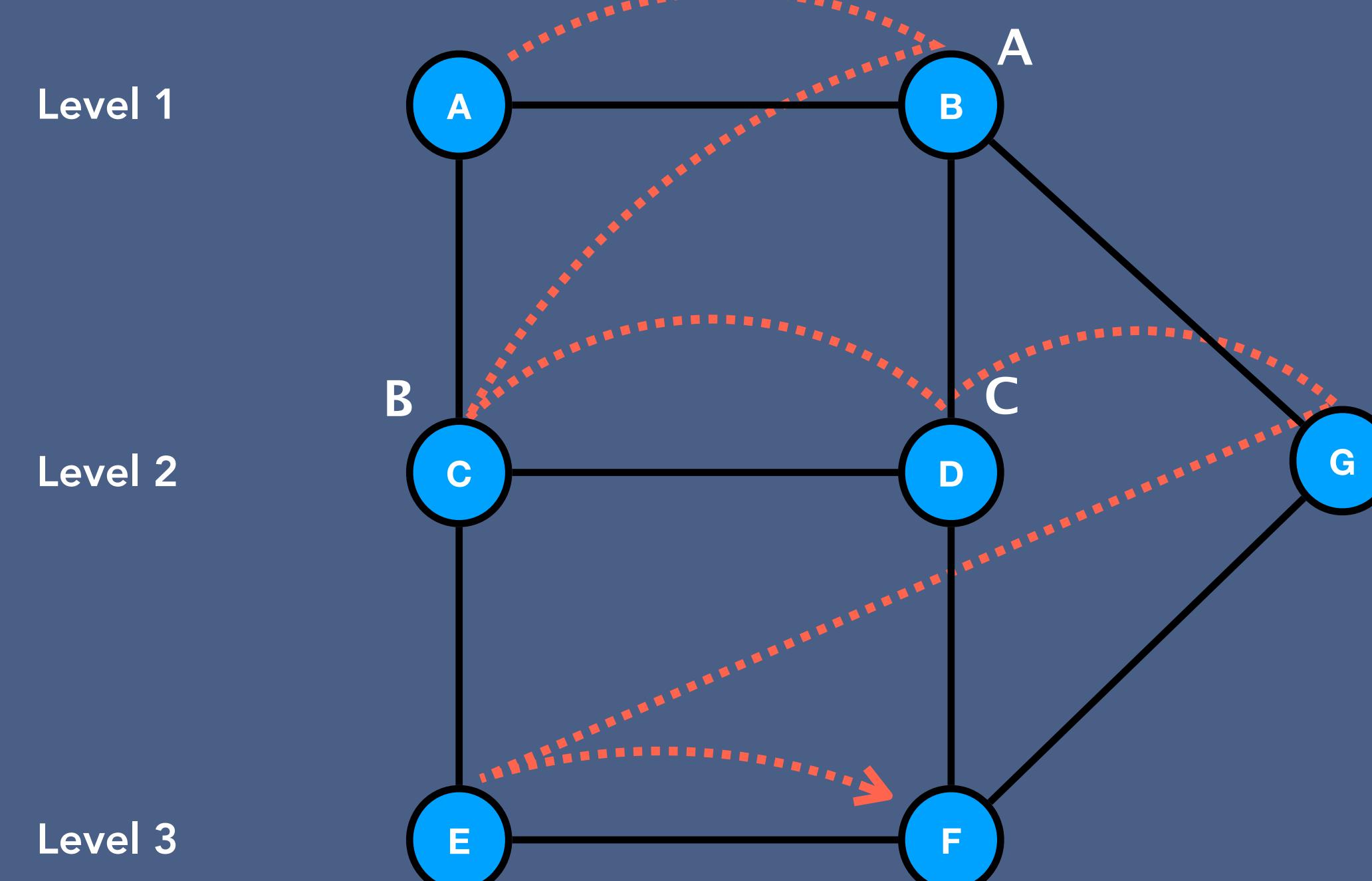
- Five offices in different cities.
- Travel costs between these cities are known.
- Find the cheapest way from head office to branches in different cities



# Single Source Shortest Path Problem (SSSPP)

- BFS
- Dijkstra's Algorithm
- Bellman Ford

BFS - Breadth First Search



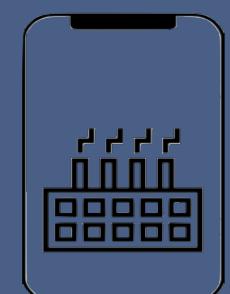
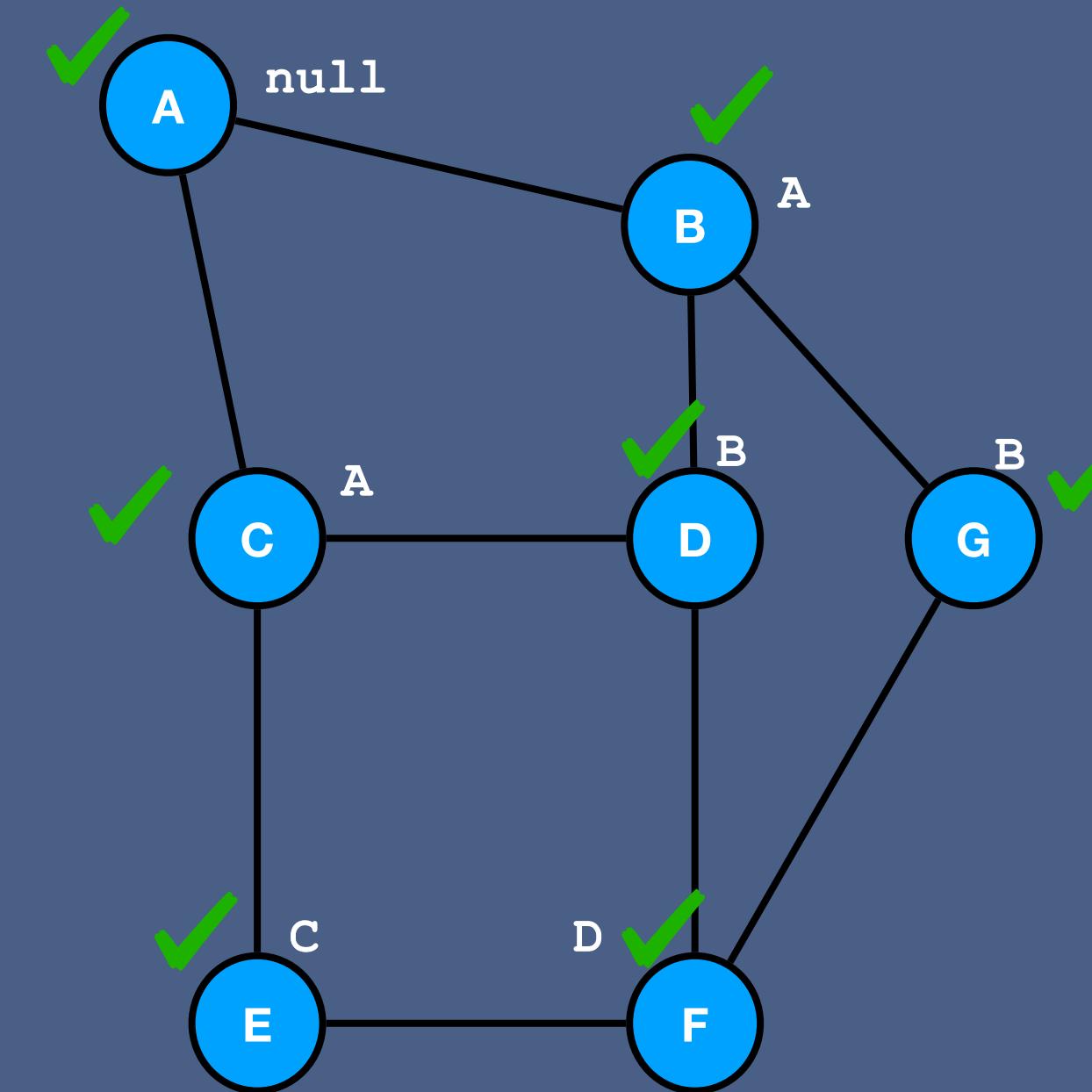
# BFS for SSSPP

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

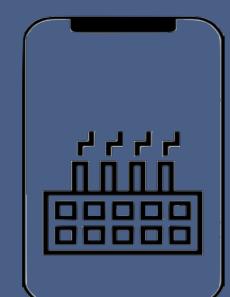


# Time and Space Complexity - BFS for SSSPP

```
create queue ..... ➔ O(1)
enqueue any starting vertex ..... ➔ O(1)
while queue is not empty ..... ➔ O(V)
    p = dequeue() ..... ➔ O(1)
    if p is unvisited ..... ➔ O(1)
        mark it visited ..... ➔ O(1)
        enqueue all adjacent unvisited vertices of p ..... ➔ O(adj)
        update parent of adjacent vertices to curVertex ..... ➔ O(1)
```

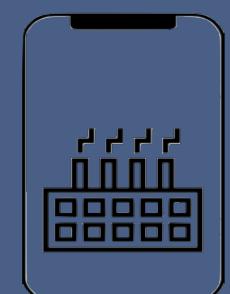
Time Complexity :  $O(E)$

Space Complexity :  $O(V)$

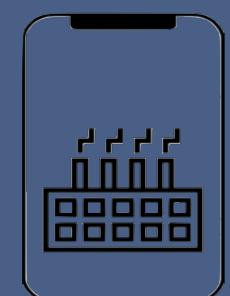
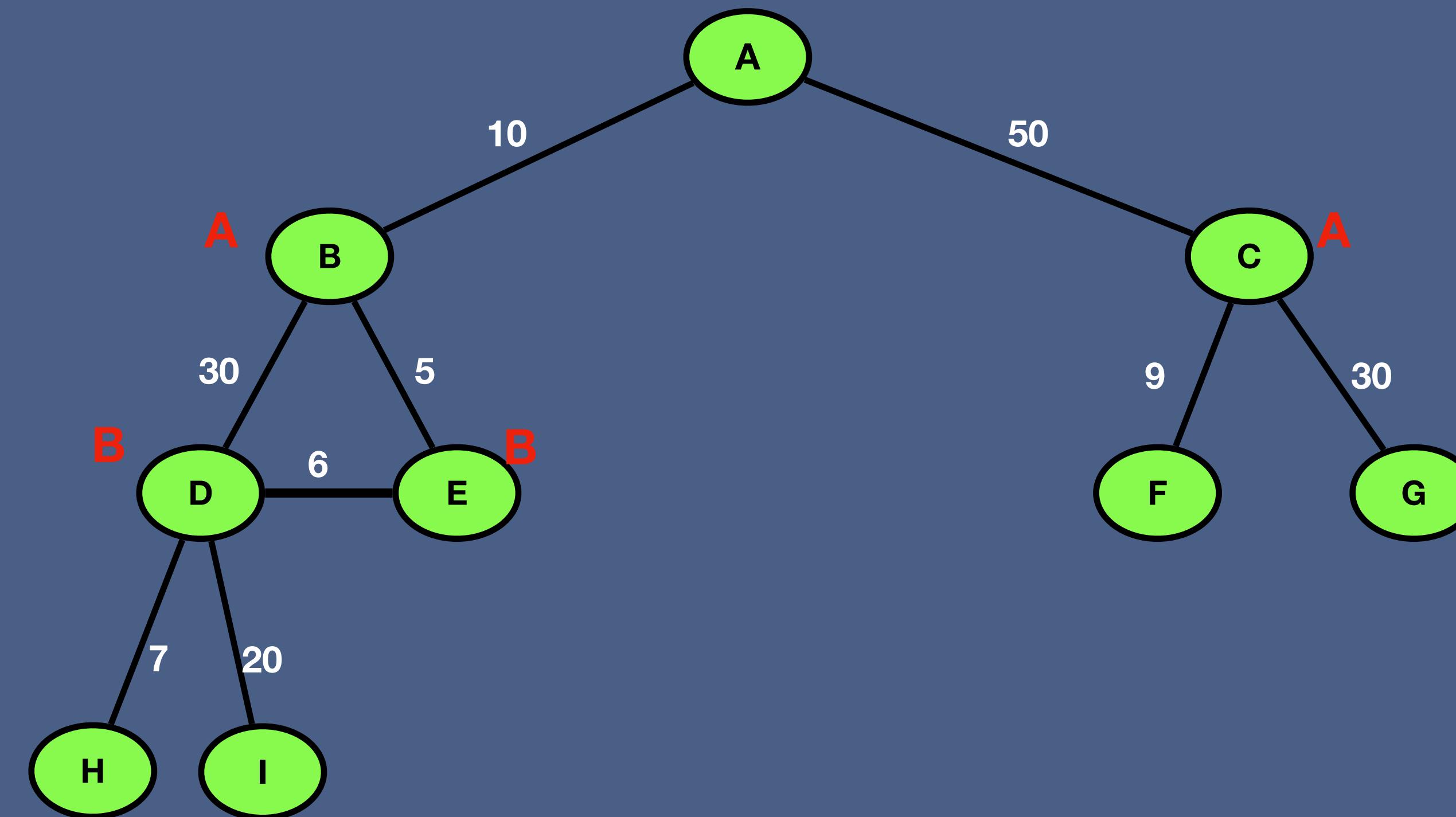


# Why BFS not work with weighted Graph?

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

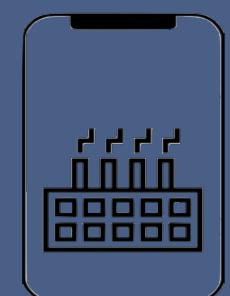
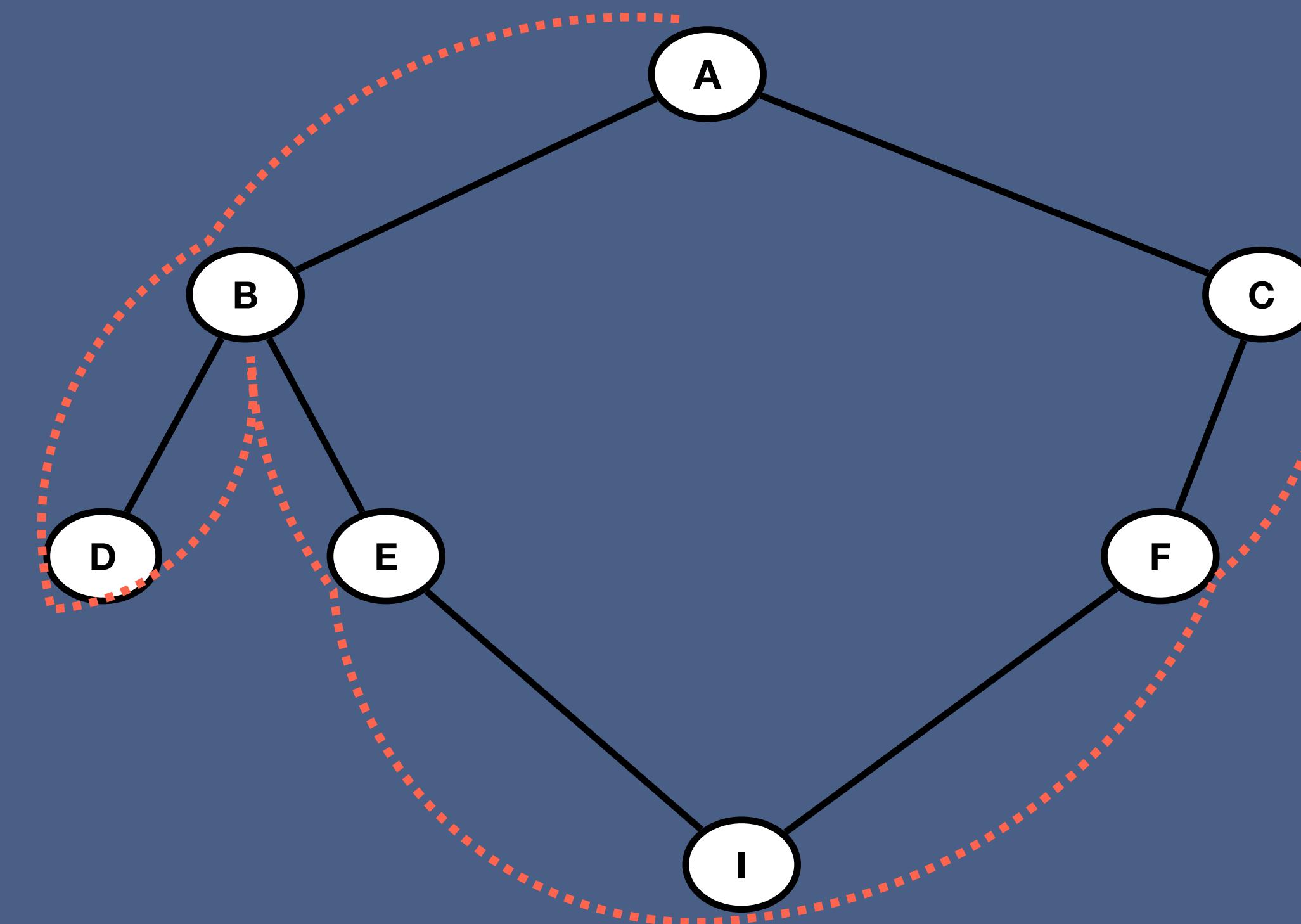


# Why BFS not work with weighted Graph?

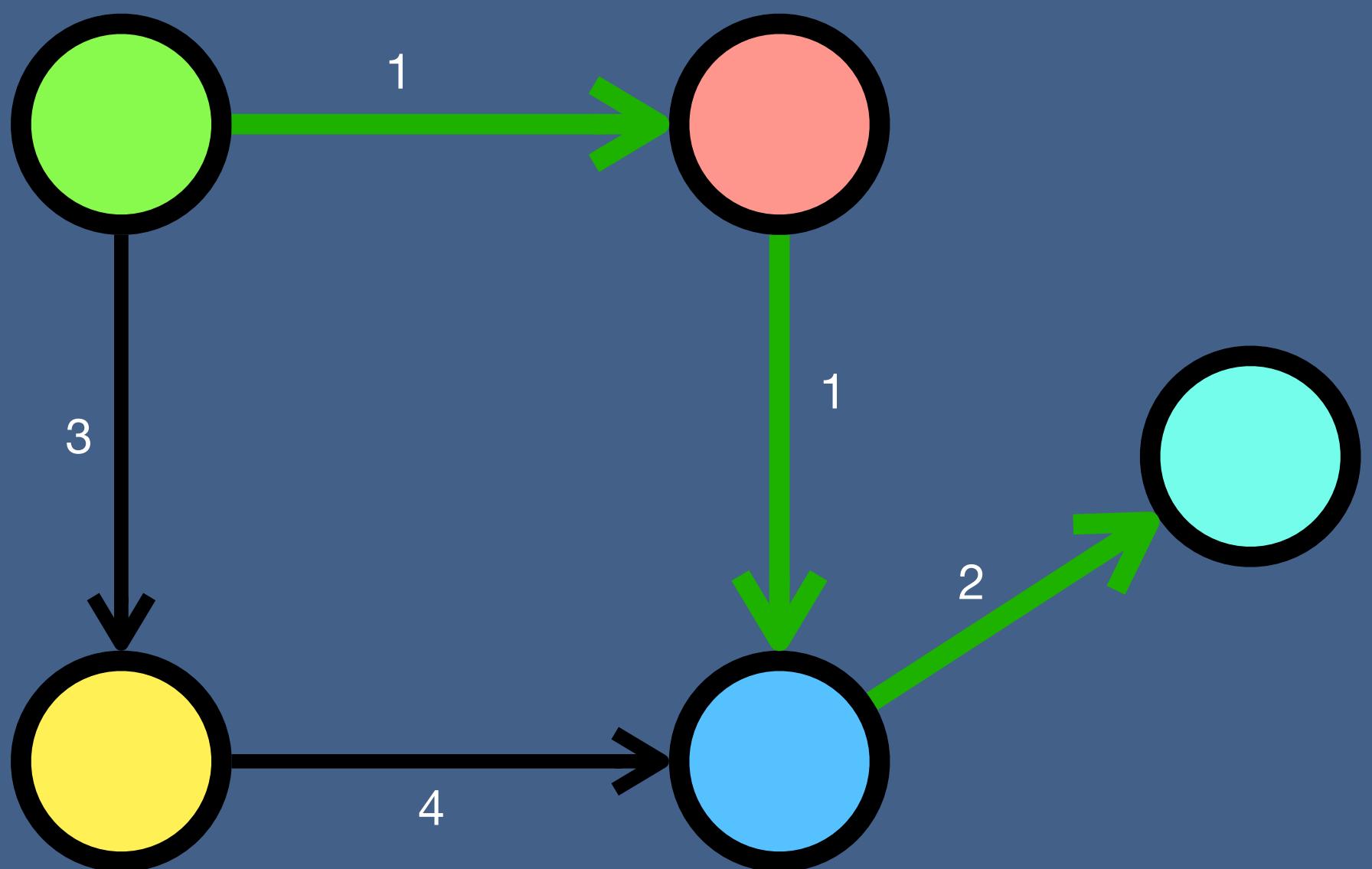


# Why does DFS not work SSSPP?

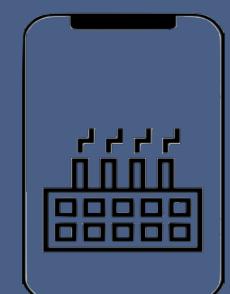
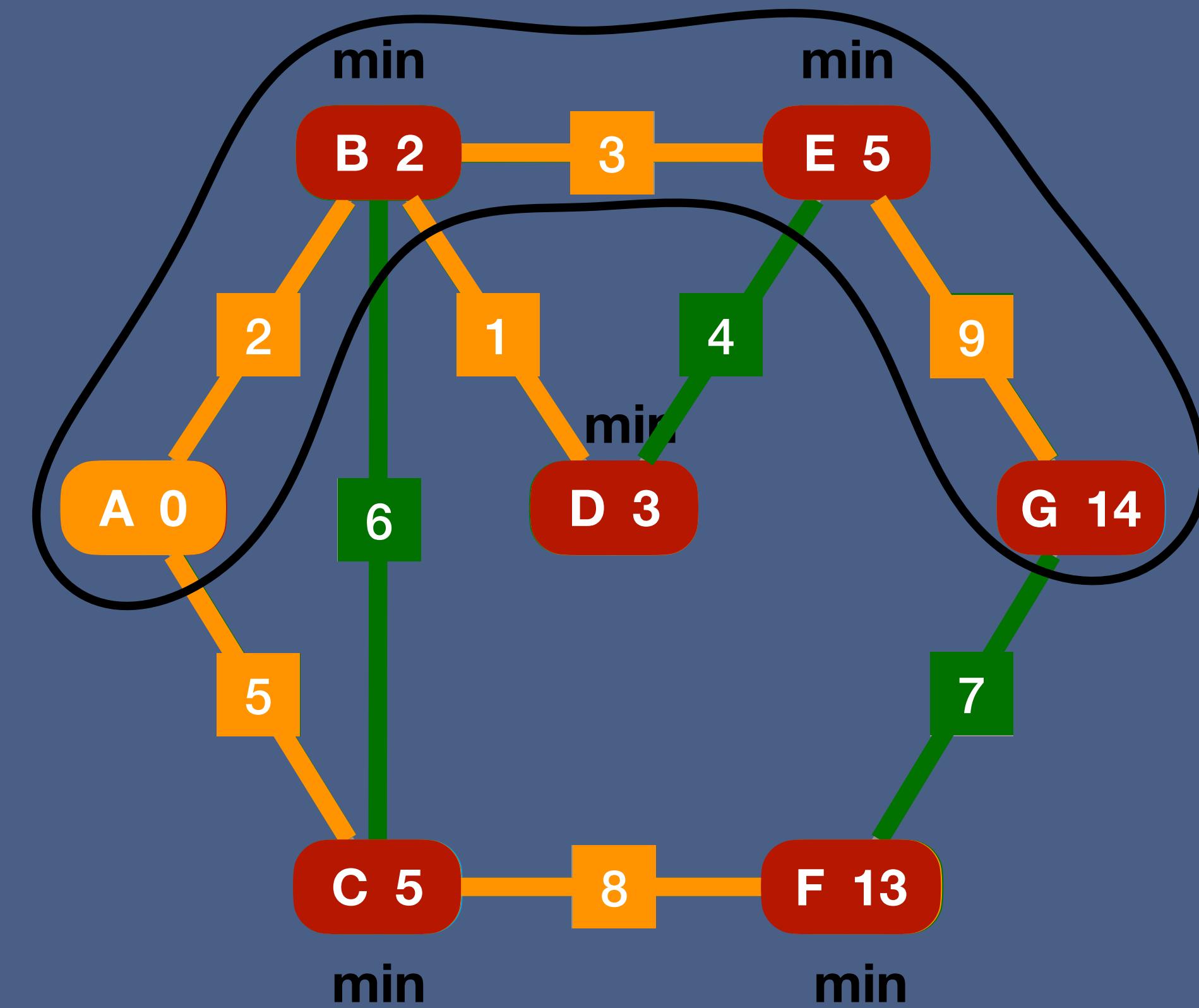
DFS has the tendency to go “as far as possible” from source, hence it can never find “Shortest Path”



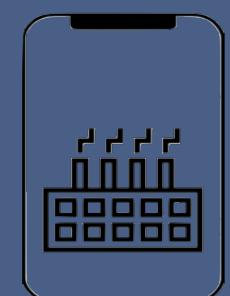
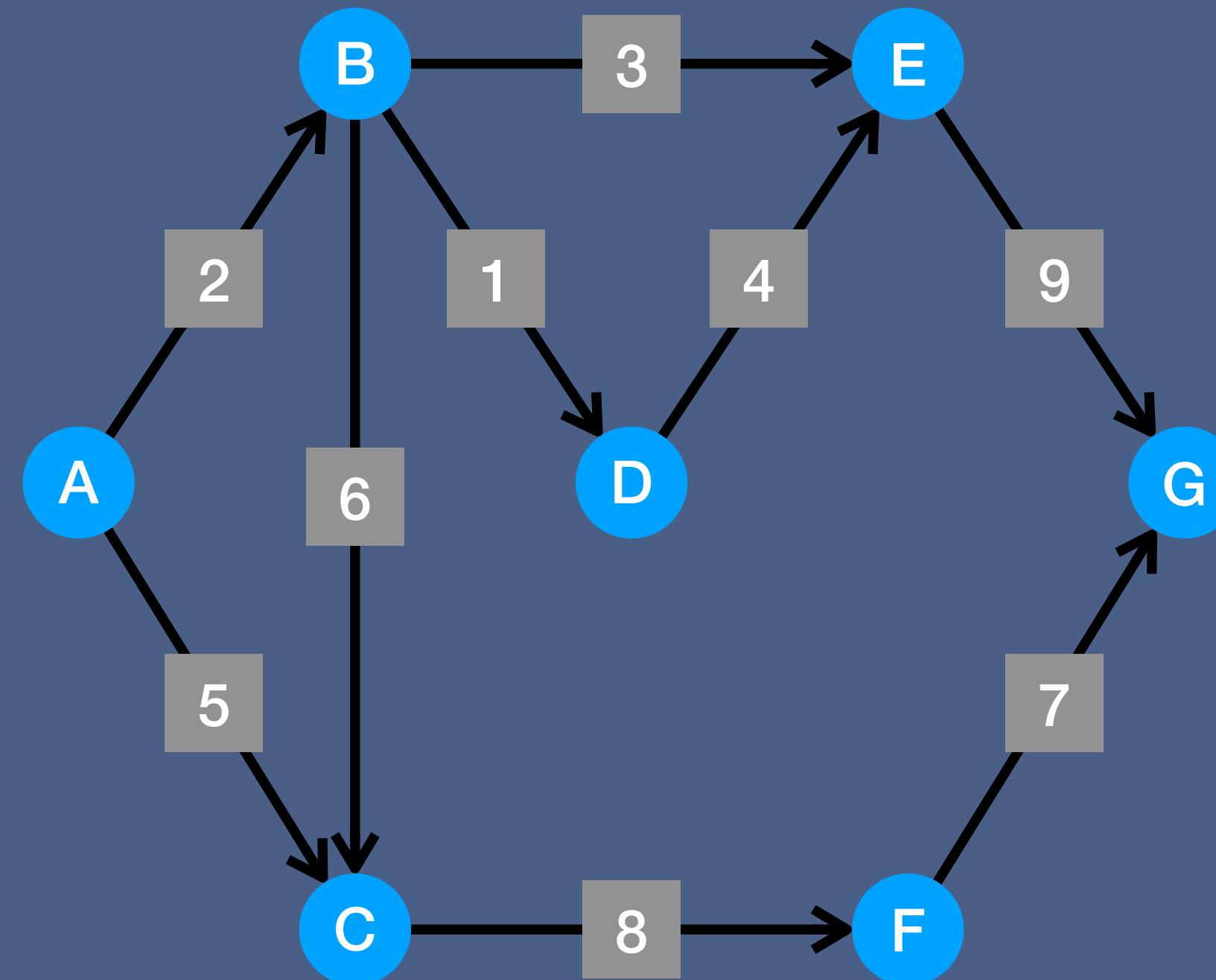
# Dijkstra's Algorithm



# Dijkstra's Algorithm for SSSPP



# Dijkstra's Algorithm for SSSPP

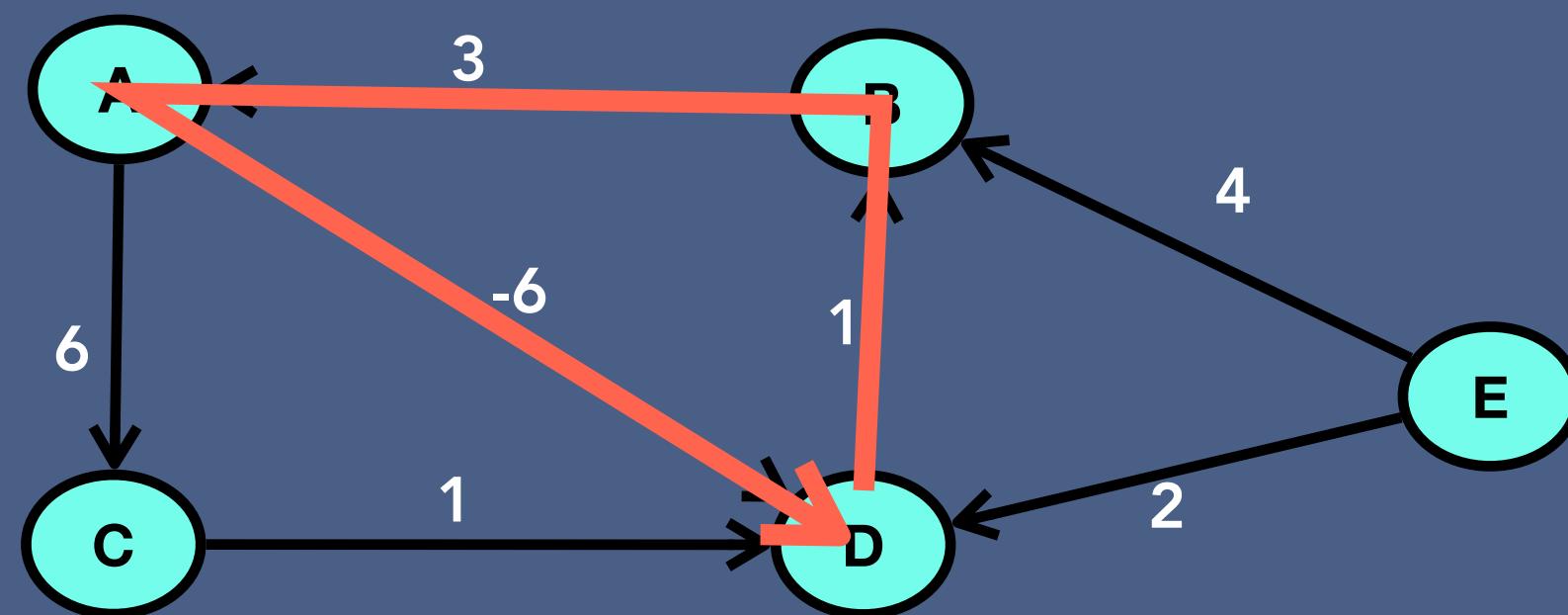


# Dijkstra's Algorithm with Negative Cycle

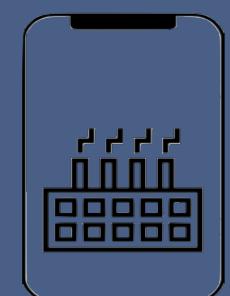
A path is called a negative cycle if:

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

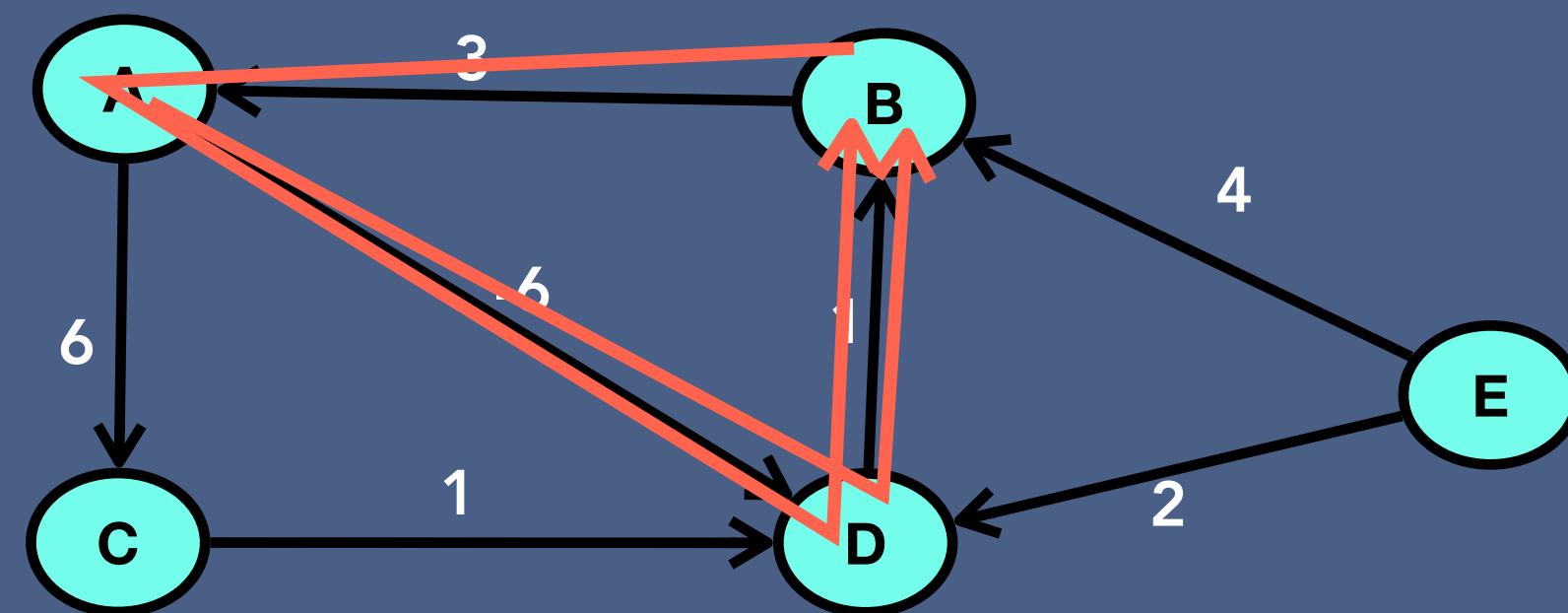


$$1 + 3 + (-6) = -2$$

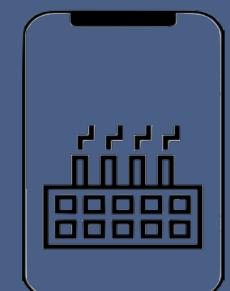


# Dijkstra's Algorithm with Negative Cycle

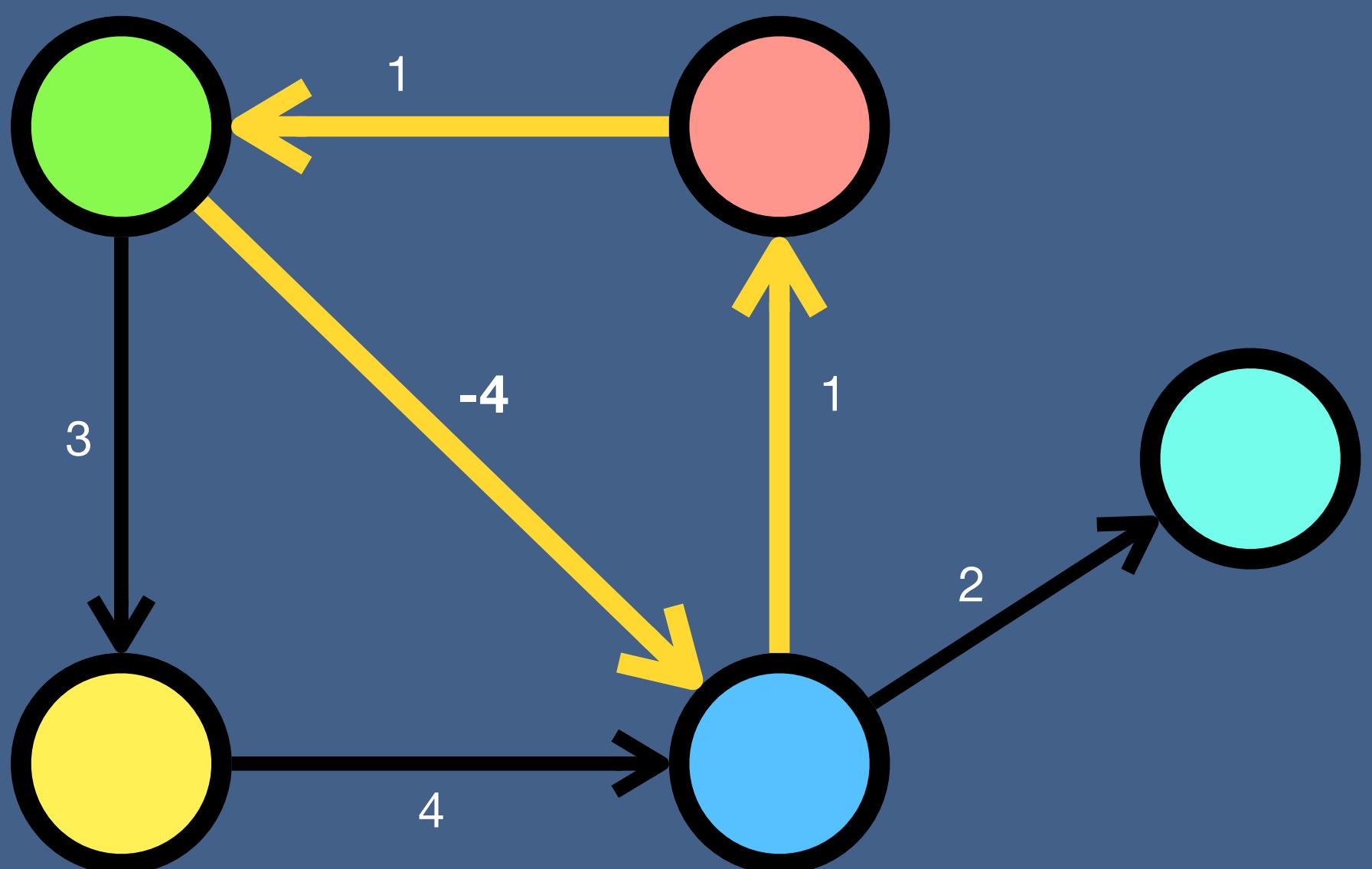
We cannot never find a negative cycle in a graph



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



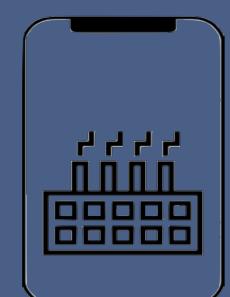
# Bellman Ford Algorithm



# Bellman Ford Algorithm

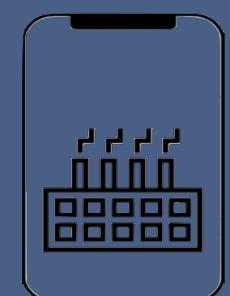
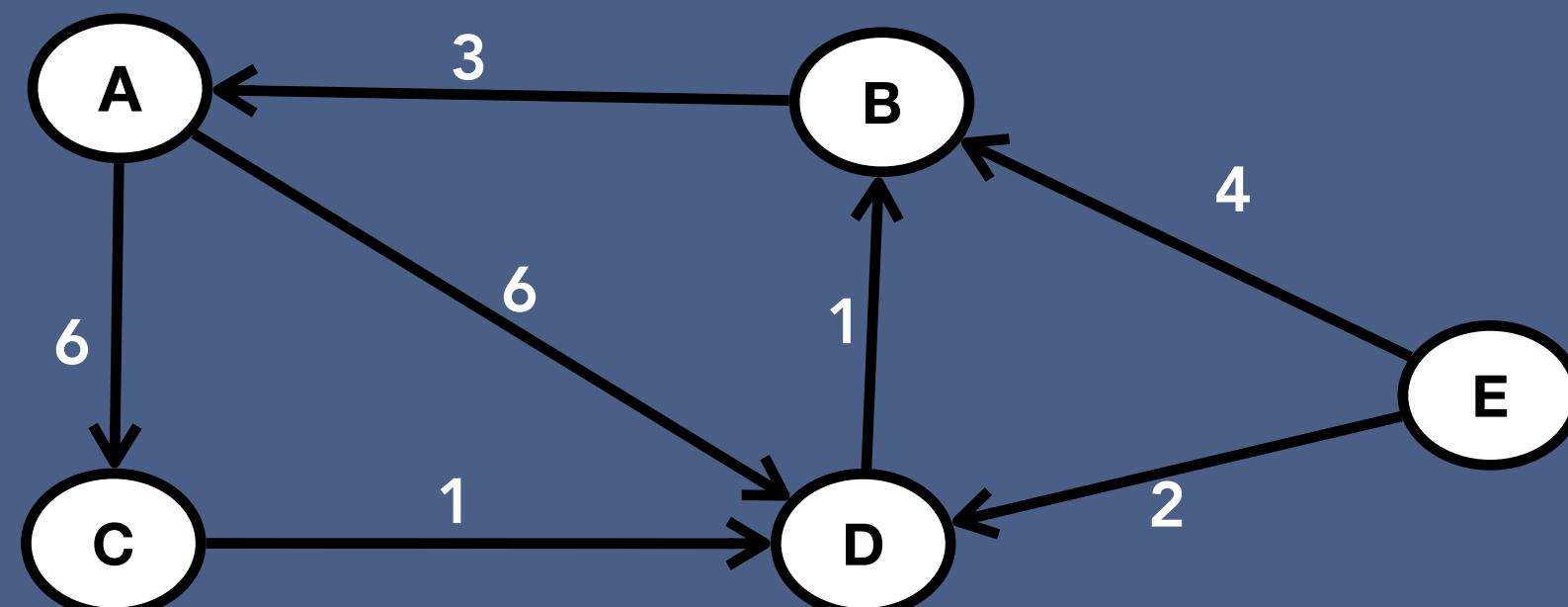
## Single Source Shortest Path Algorithm

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

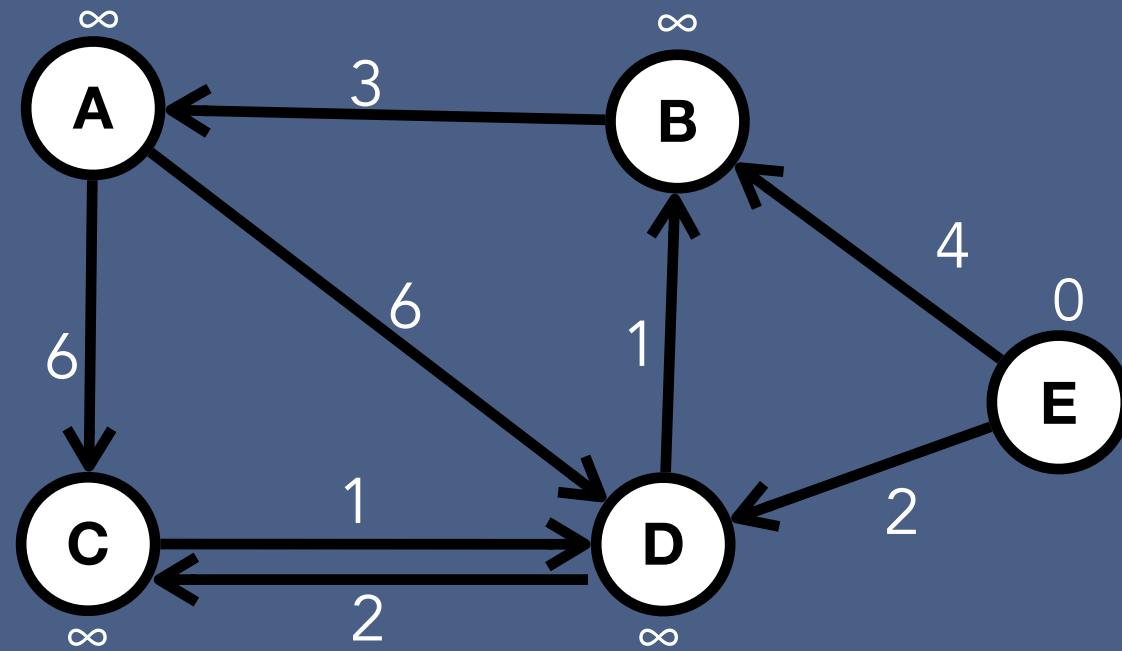


# Bellman Ford Algorithm

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

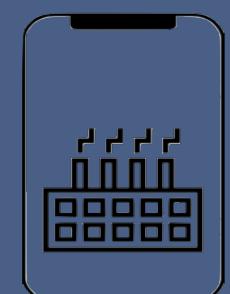


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

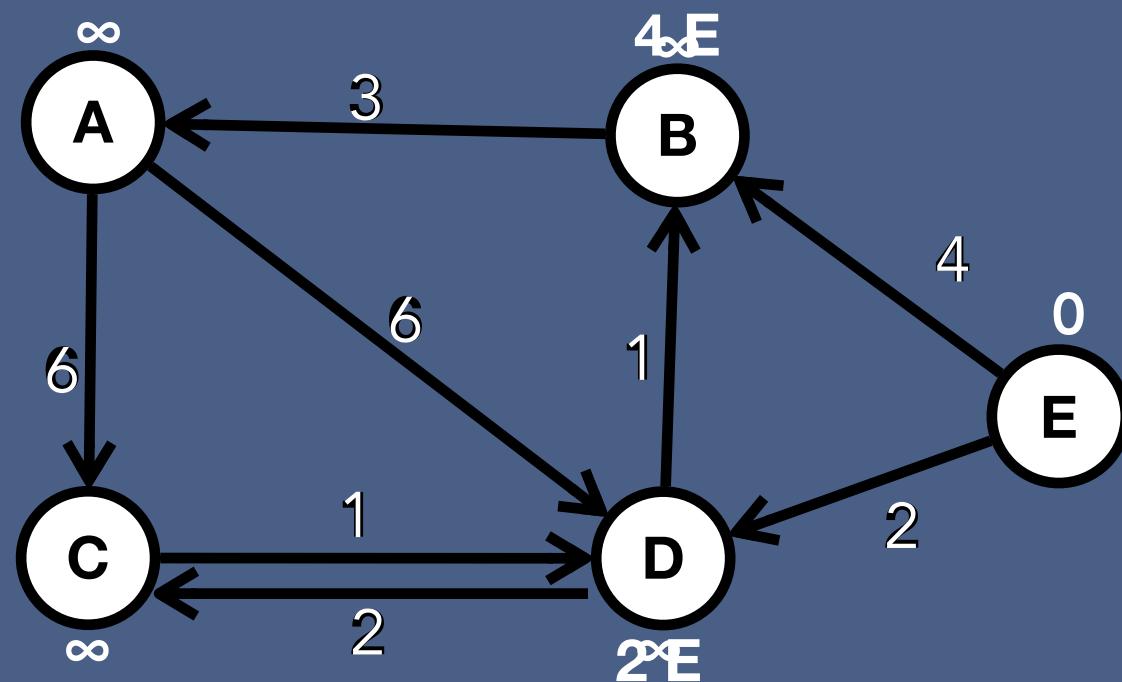
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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	∞
B	∞
C	∞
D	∞
E	0



# Bellman Ford Algorithm

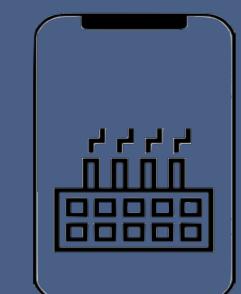


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

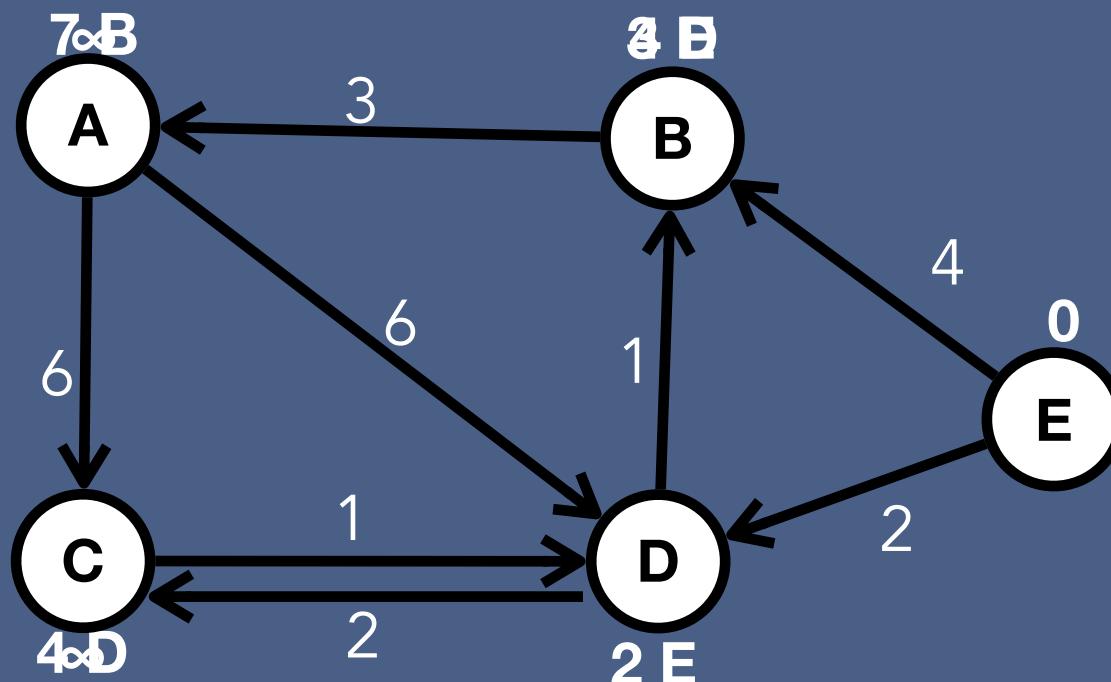
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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	
		Distance	Parent
A	infinity	-	-
B	infinity	E	-
C	infinity	-	-
D	infinity	E	-
E	0	-	-



# Bellman Ford Algorithm

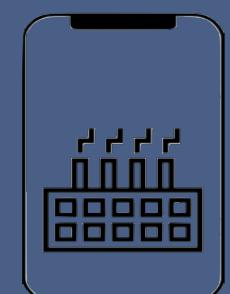


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

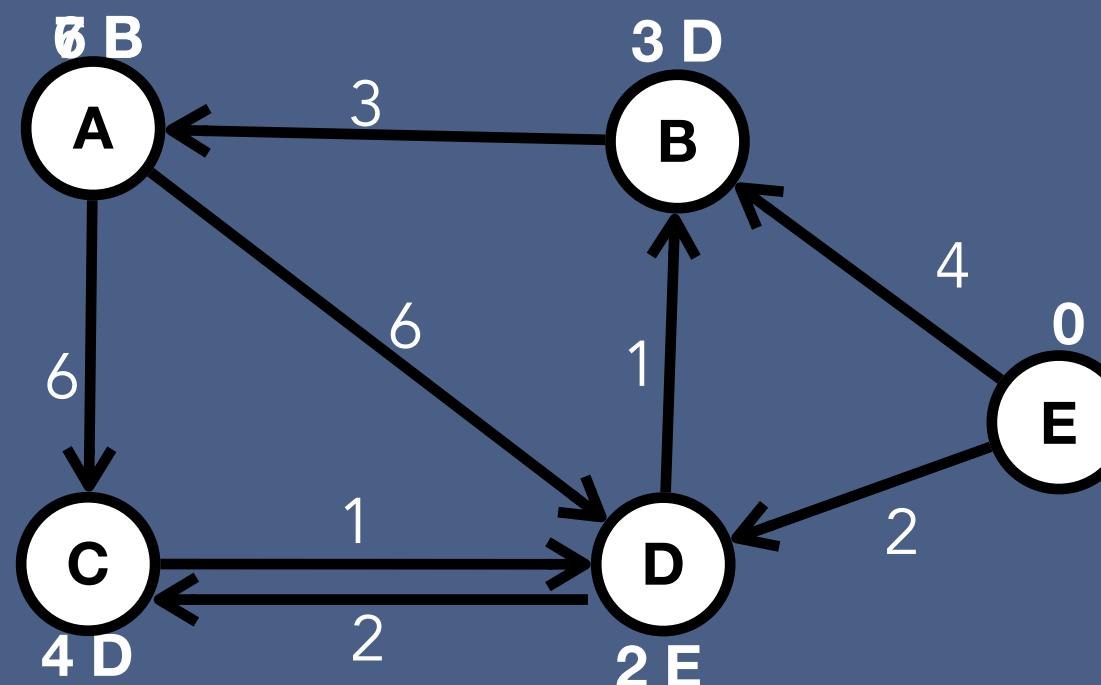
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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	$\infty$	$\infty$	-	$2+2=4$	D
D	$\infty$	2	E	2	E
E	0	0	-	0	-



# Bellman Ford Algorithm

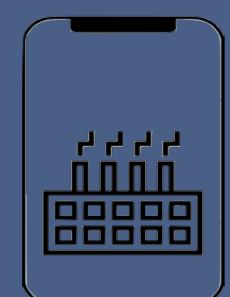


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

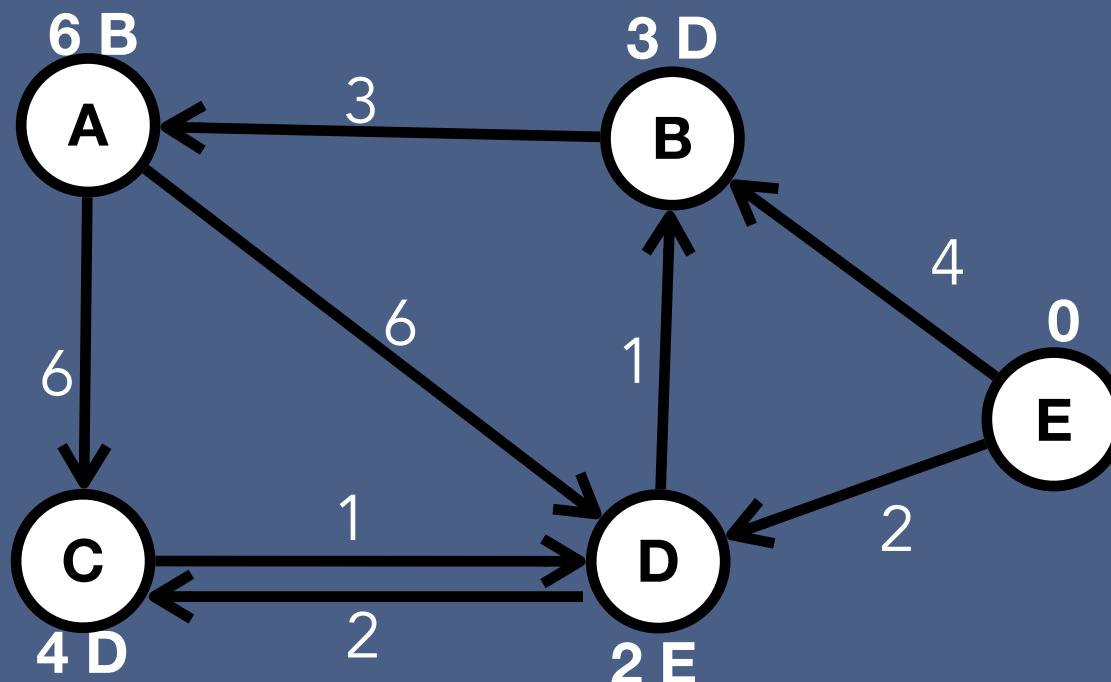
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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	$\infty$	$\infty$	-	$2+2=4$	D	4	D
D	$\infty$	2	E	2	E	2	E
E	0	0	-	0	-	0	



# Bellman Ford Algorithm

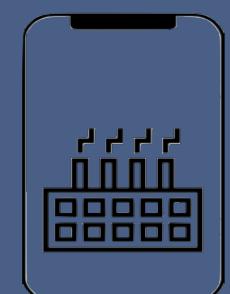


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

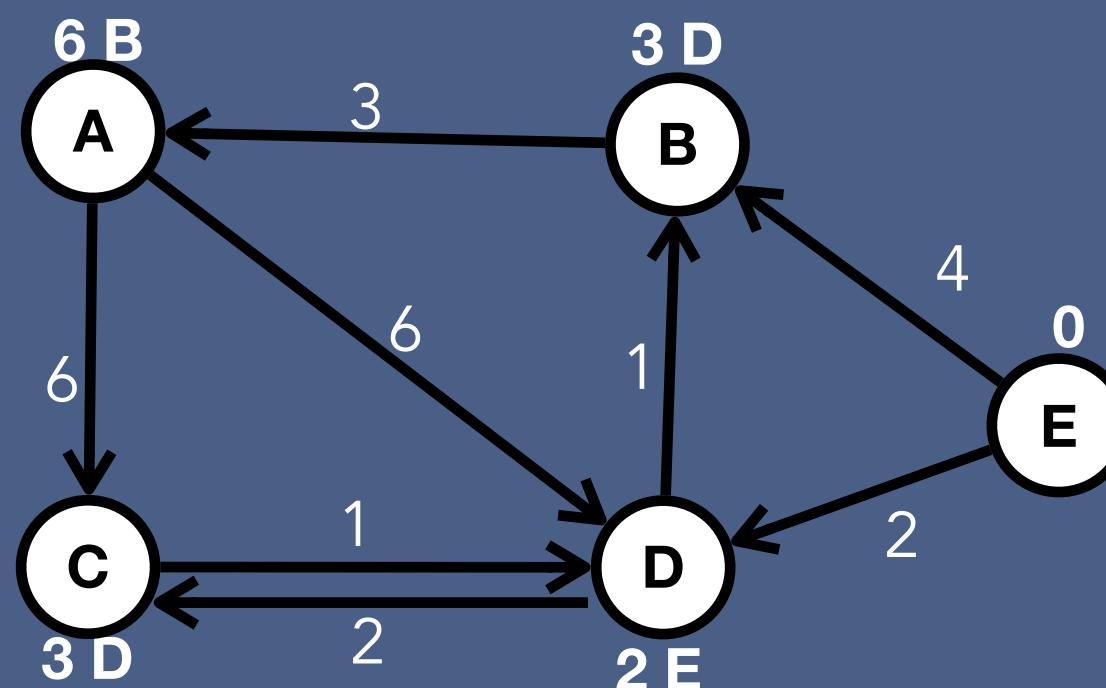
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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

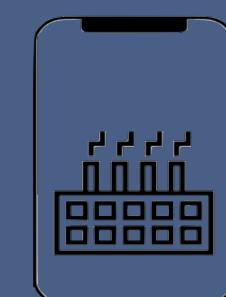


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

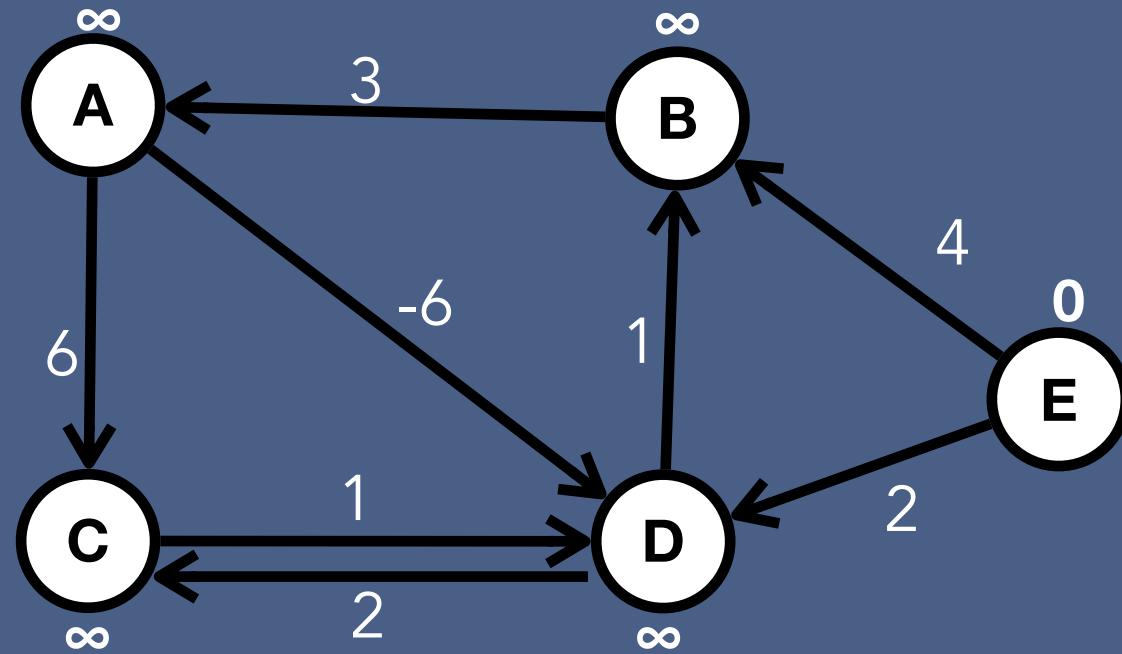
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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 with Negative Cycle

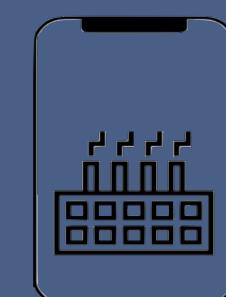


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

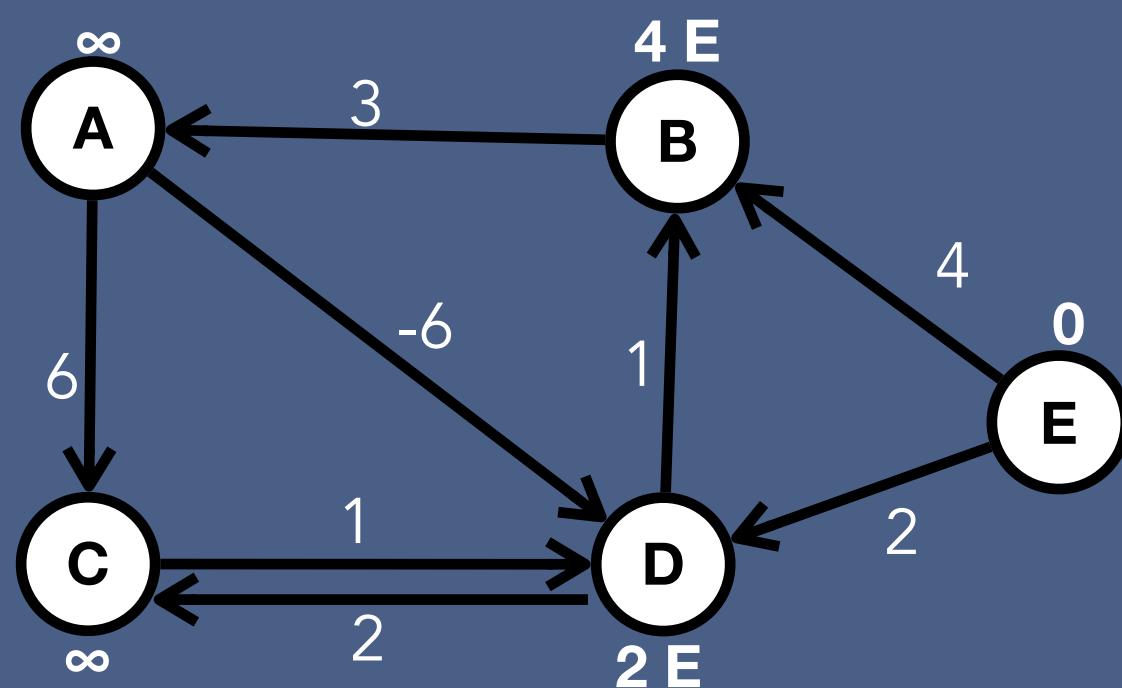
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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 with Negative Cycle

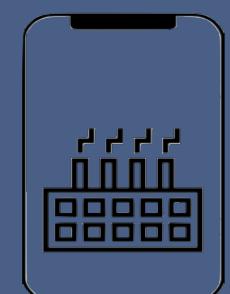


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

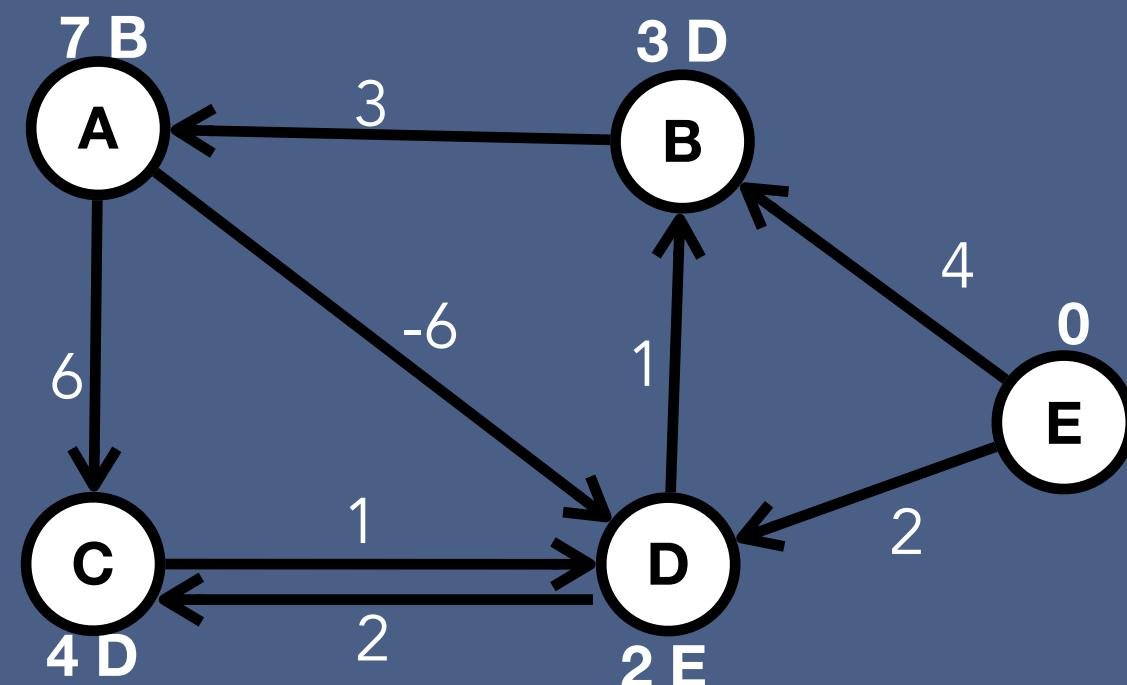
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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	
		Distance	Parent
A	$\infty$	$\infty$	-
B	$\infty$	4	E
C	$\infty$	$\infty$	-
D	$\infty$	2	E
E	0	0	-



# Bellman Ford with Negative Cycle

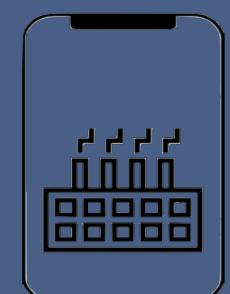


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

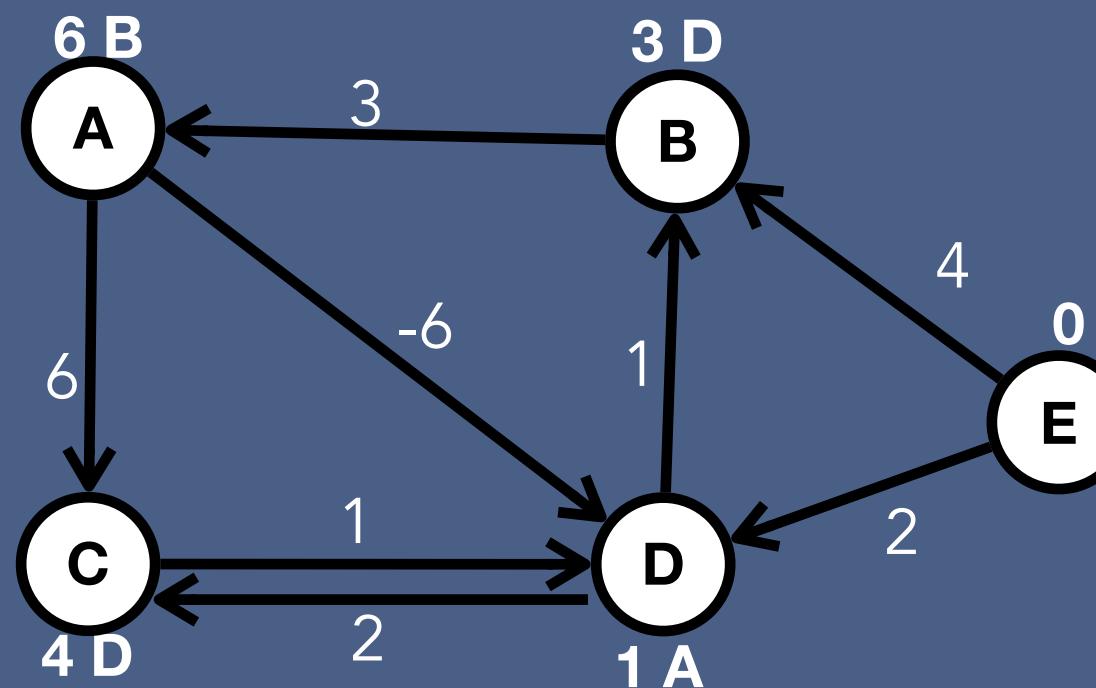
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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 with Negative Cycle

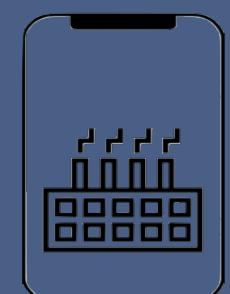


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

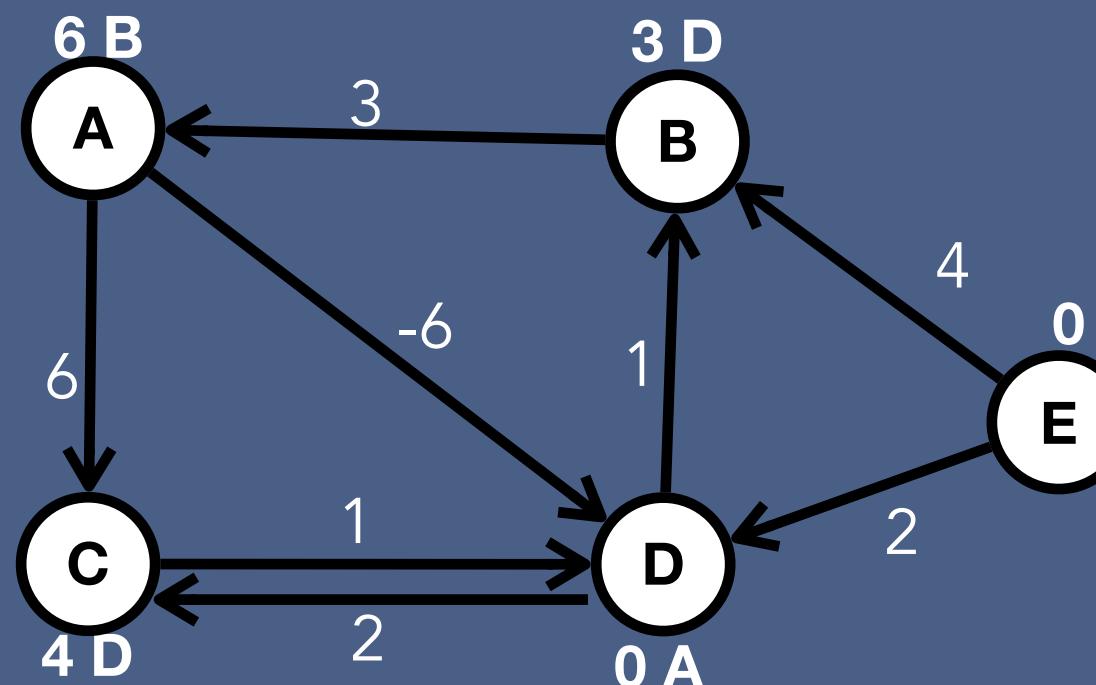
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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	$7+(-6)=1$	A	
E	0	0	-	0	-	0		



# Bellman Ford with Negative Cycle

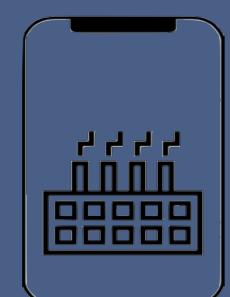


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

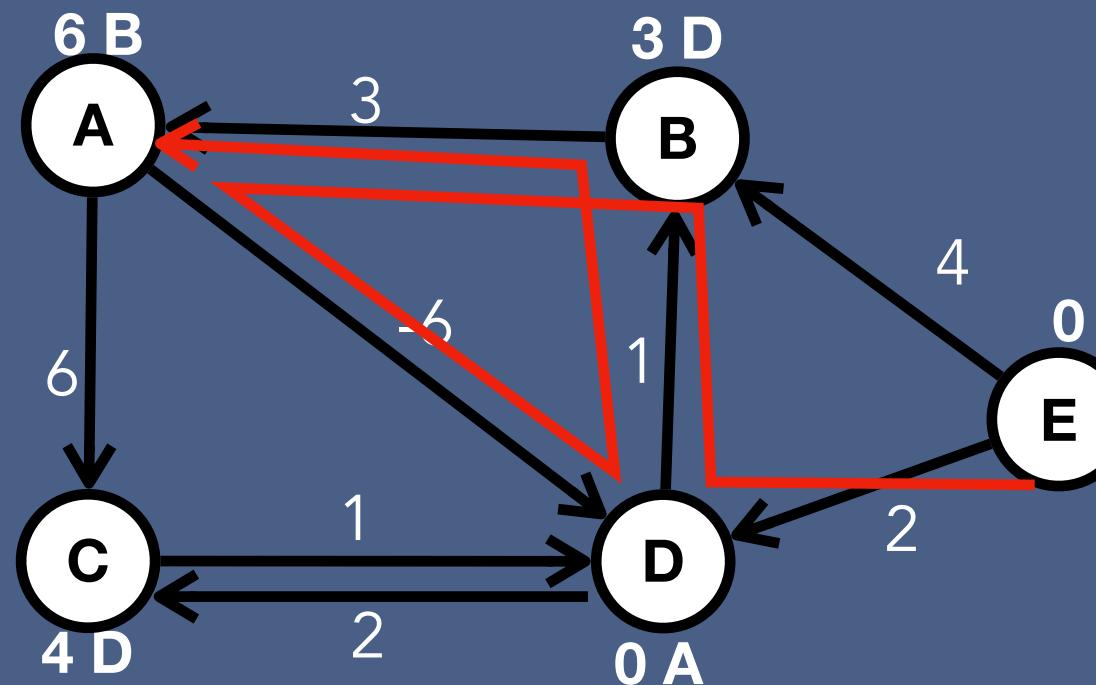
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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		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	$7+(-6)=1$	A	$6+(-6)=0$	A
E	0	0	-	0	-	0	-	0	-



# Bellman Ford with Negative Cycle

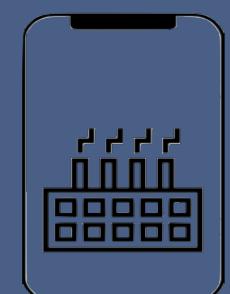


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

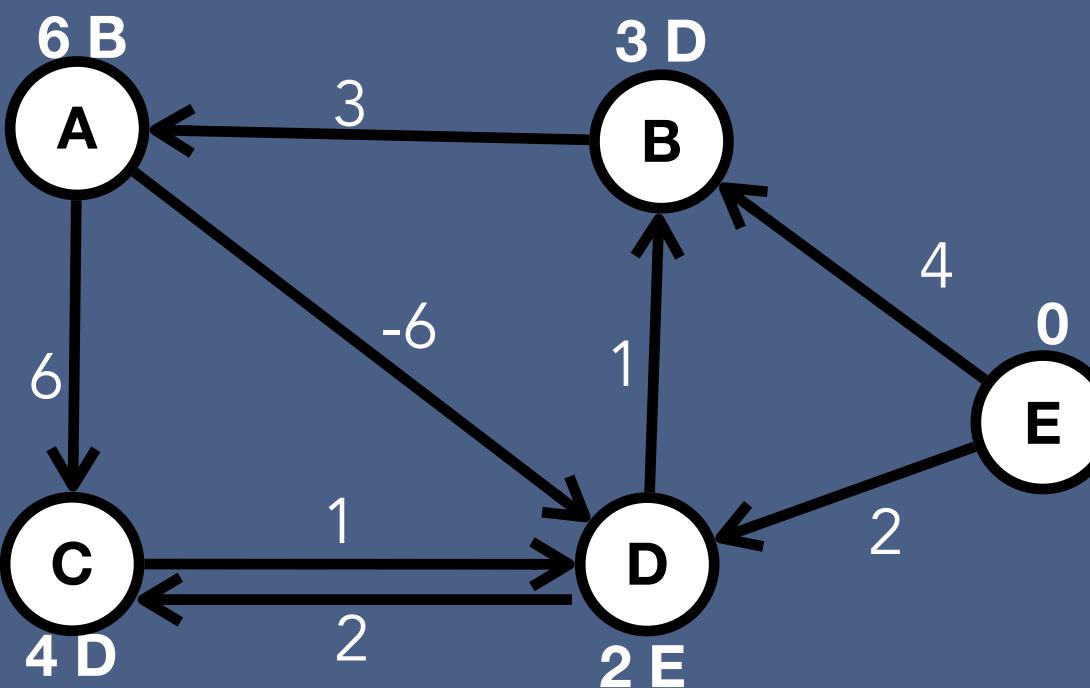
Update distance of destination vertex to (distance of source vertex + weight between source and destination vertex)

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	-		

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 with Negative Cycle

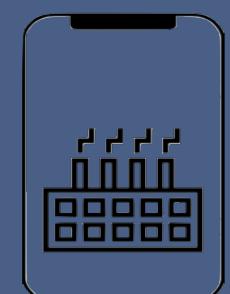


If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex):

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

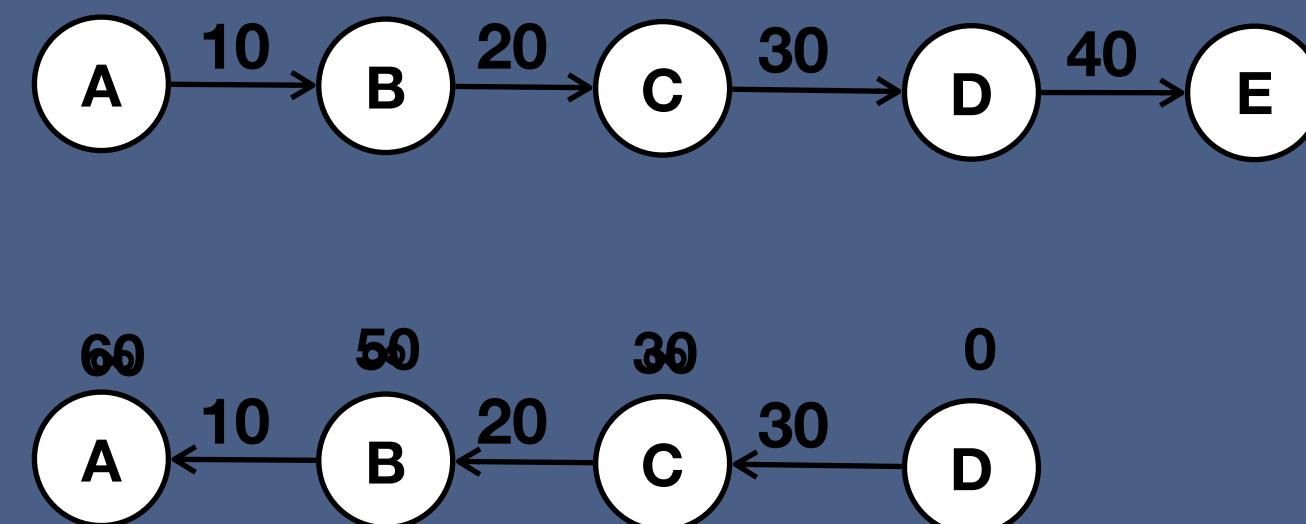
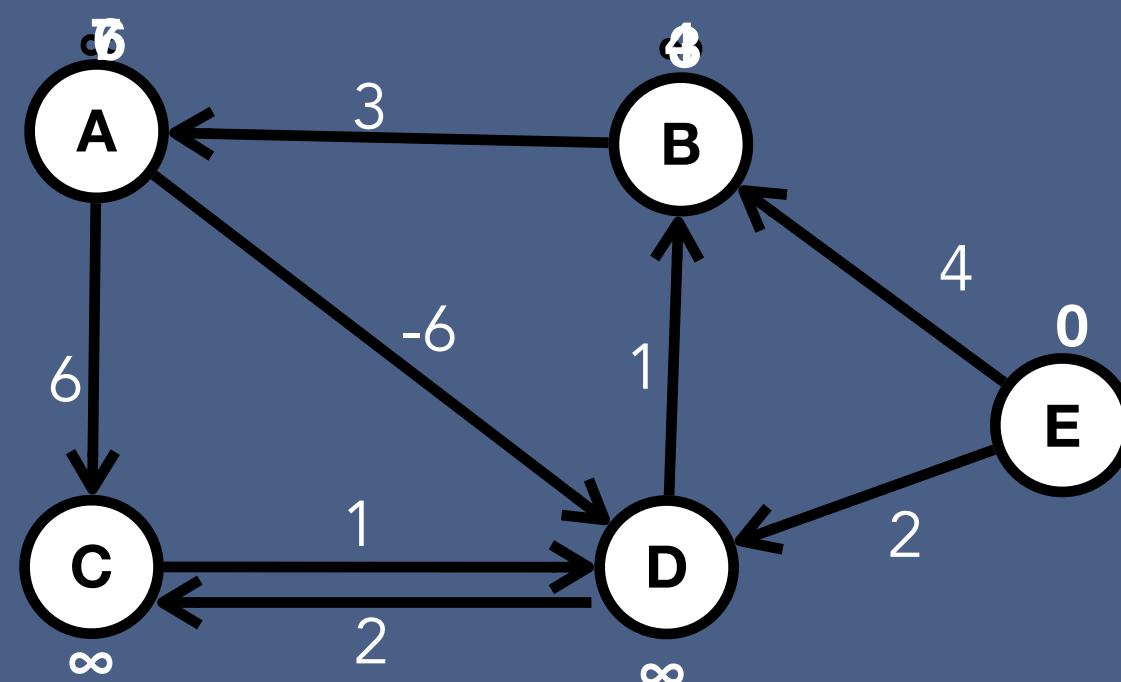
Edge	Weight
A->C	6
A->D	6
B->A	3
C->D	1
D->C	2
D->B	1
E->B	4
E->D	2

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

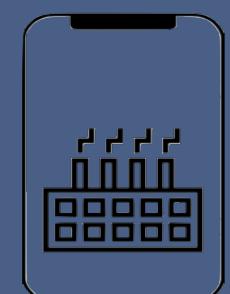


# Why does Bellman Ford run $V-1$ times?

- 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

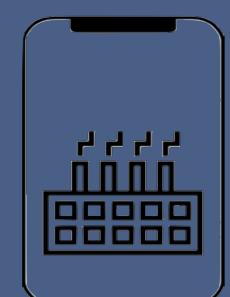


A->B  
B->C  
C->D



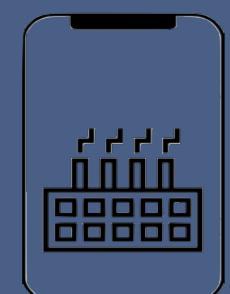
# BFS vs Dijkstra vs Bellman Ford

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

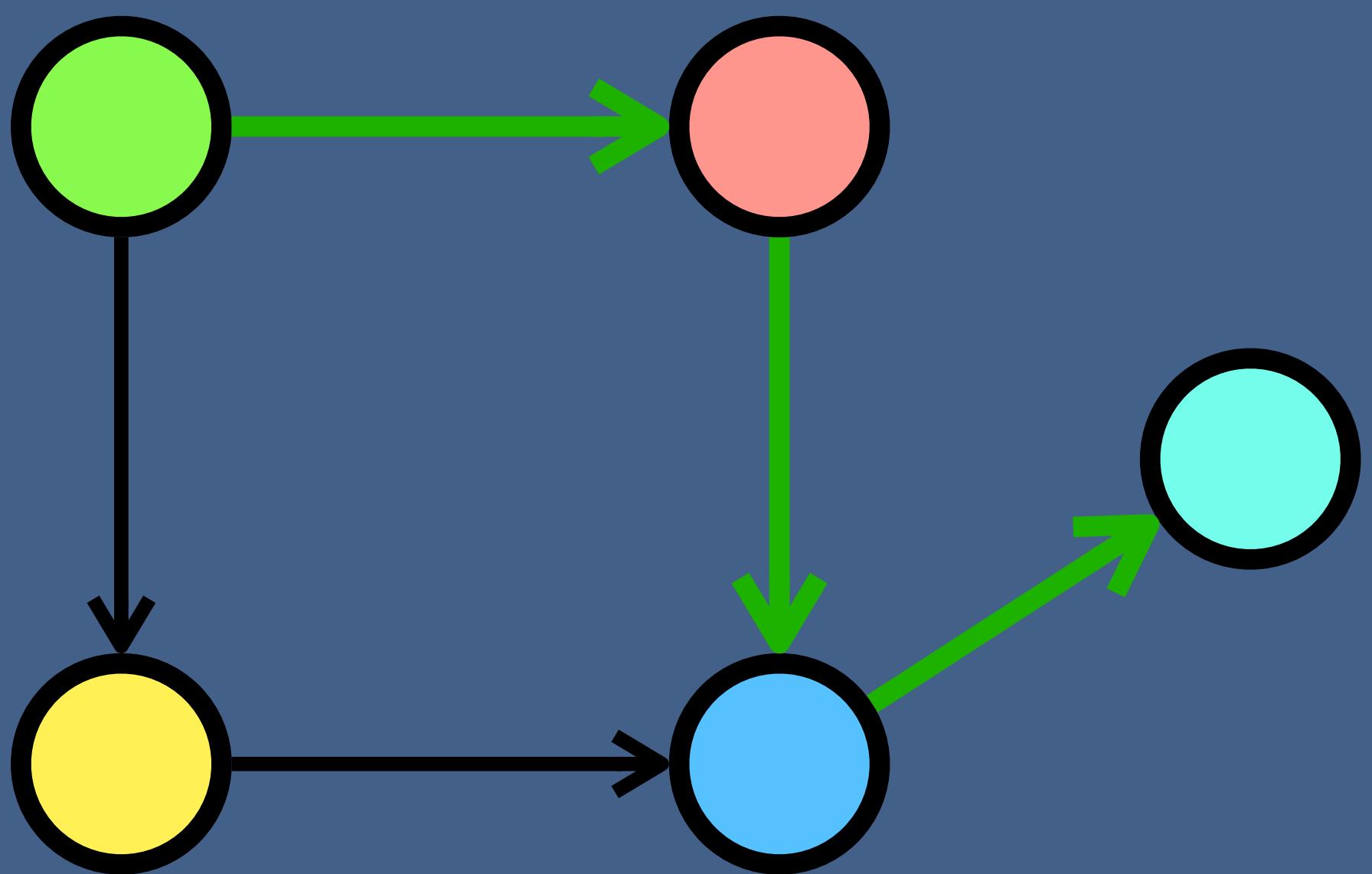


# BFS vs Dijkstra vs Bellman Ford

	<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 Pairs Shortest Path Problem (APSPP)

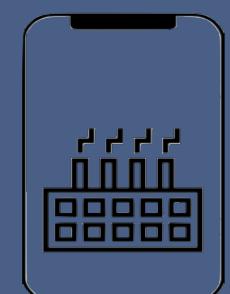
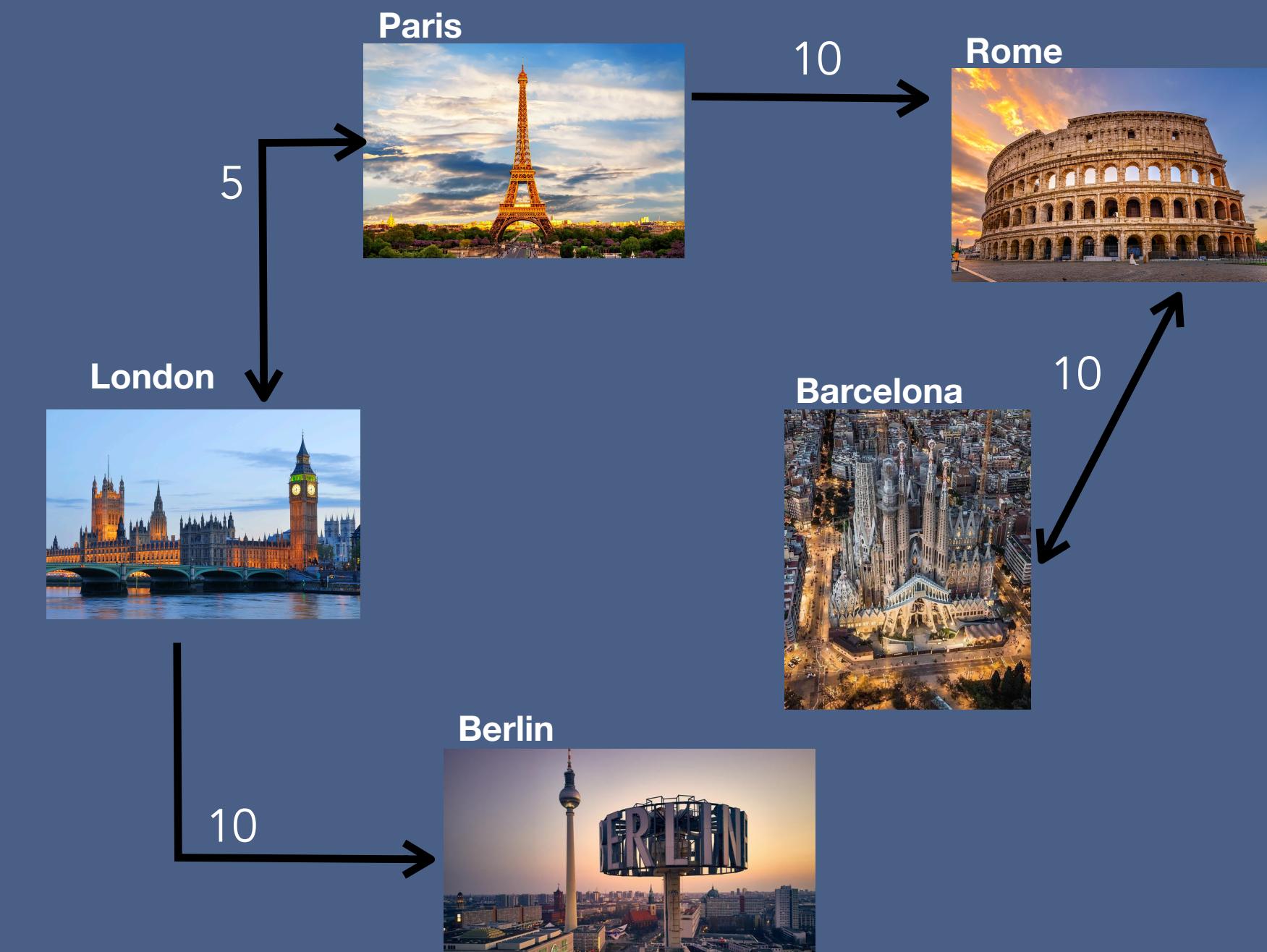
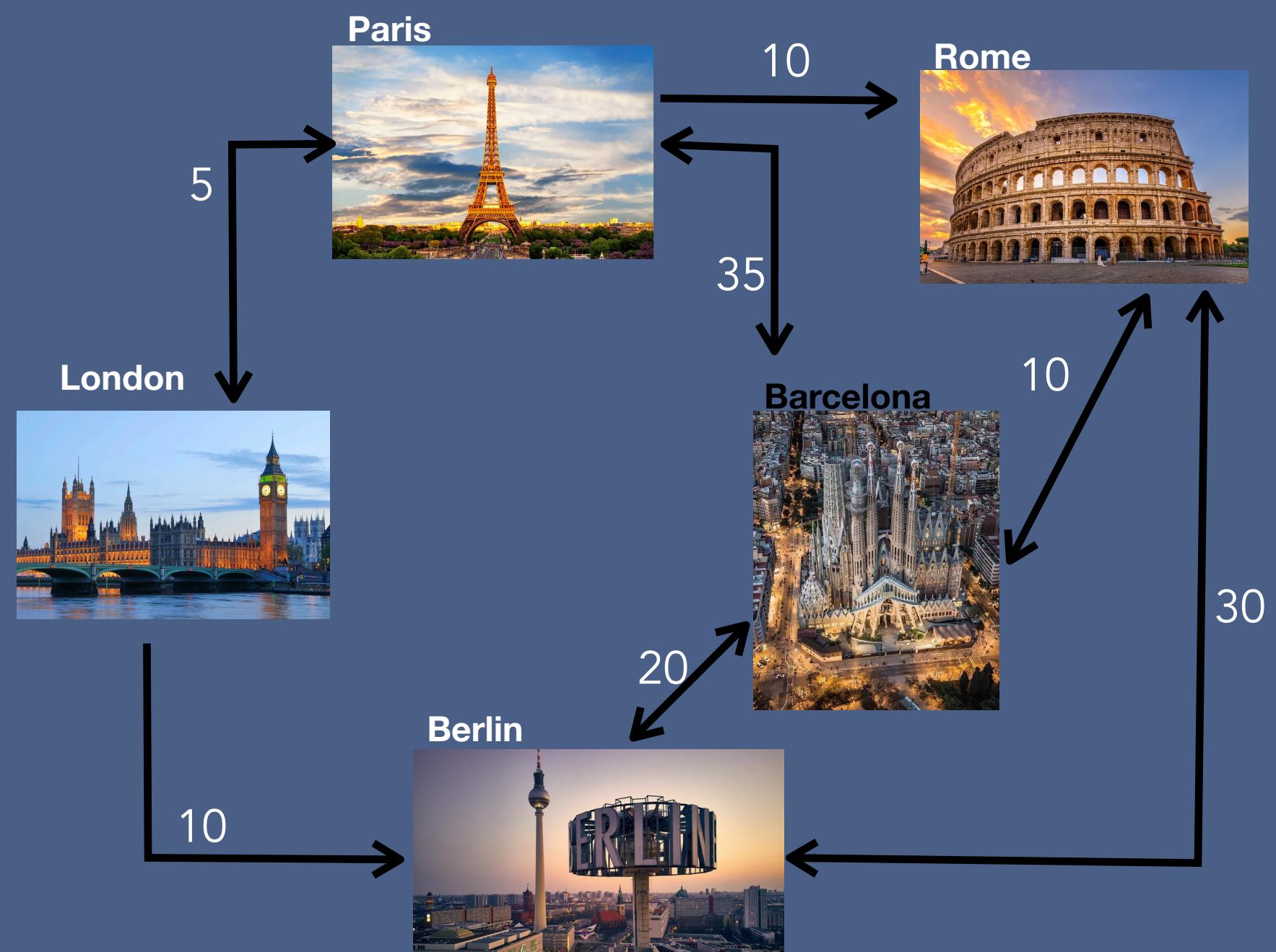


# All Pair Shortest Path Problem

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

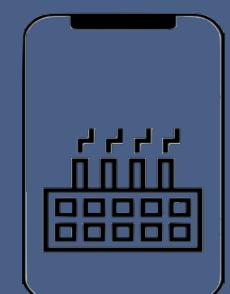
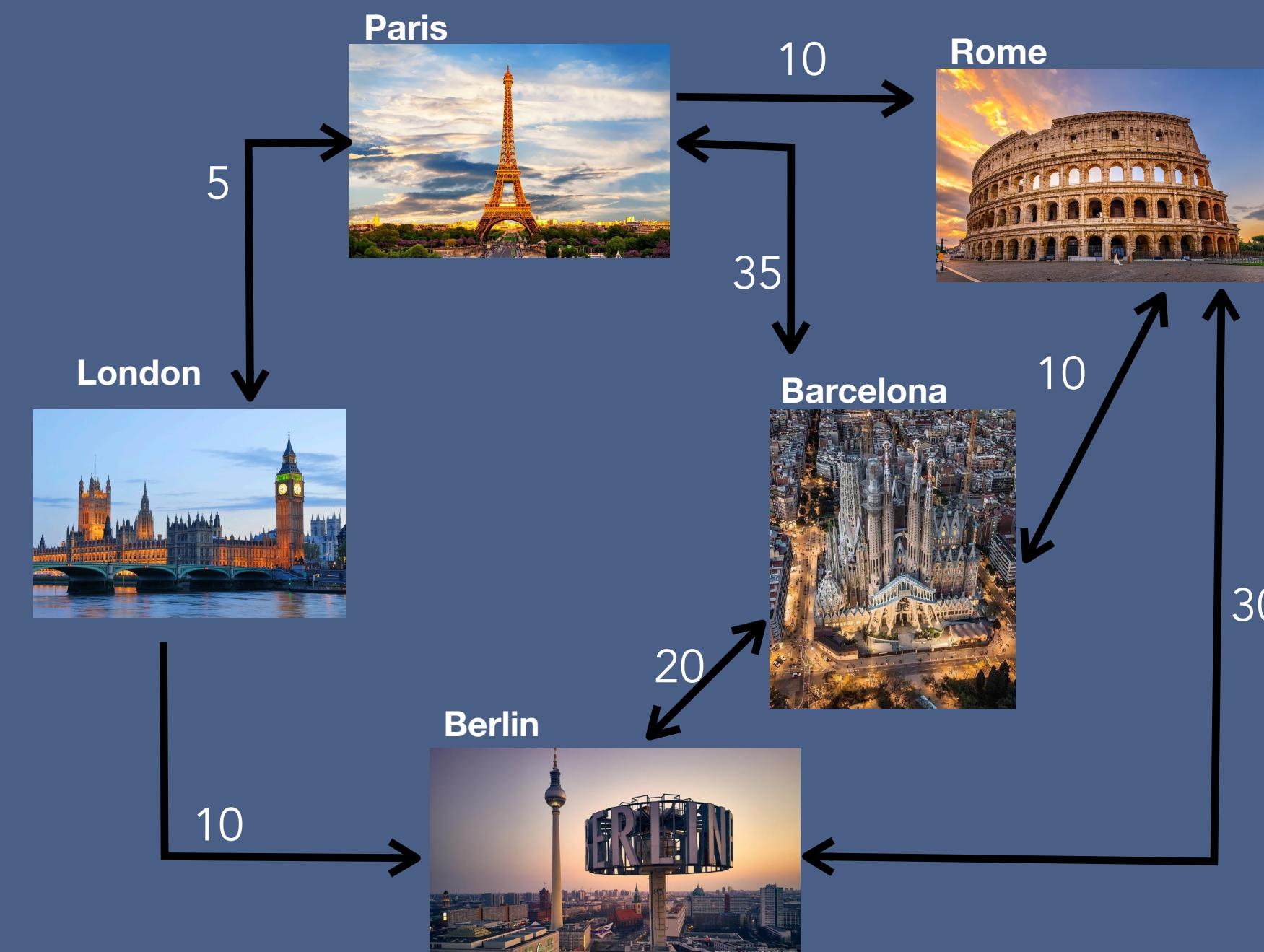


# All Pair Shortest Path Problem

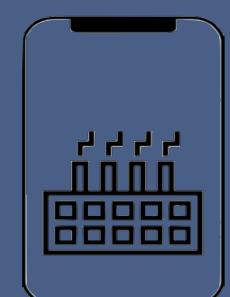
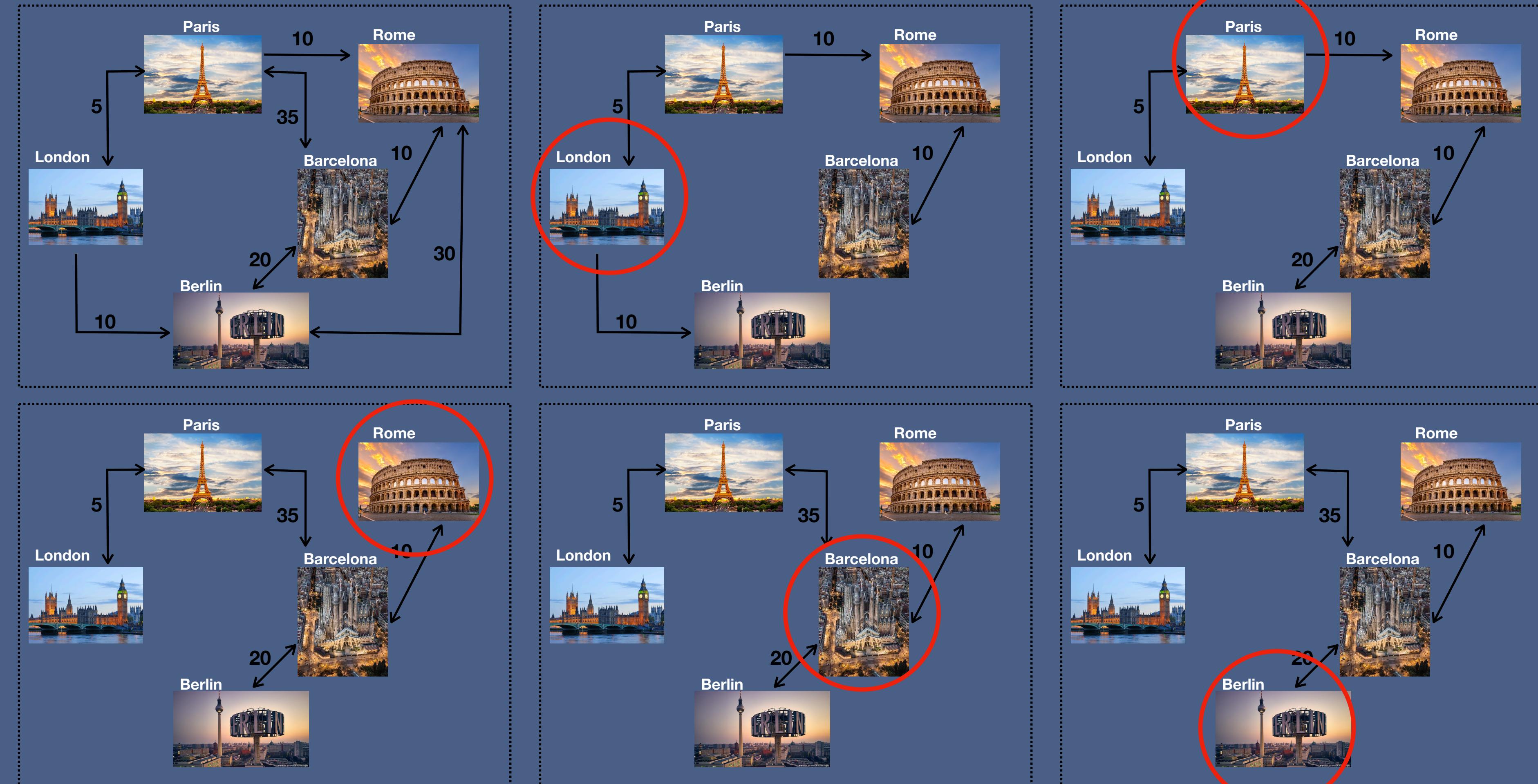
All pair shortest path problem is about finding a path between **every vertex** to all other vertices in a graph such that the total distance between them (source and destination) is minimum.

The problem:

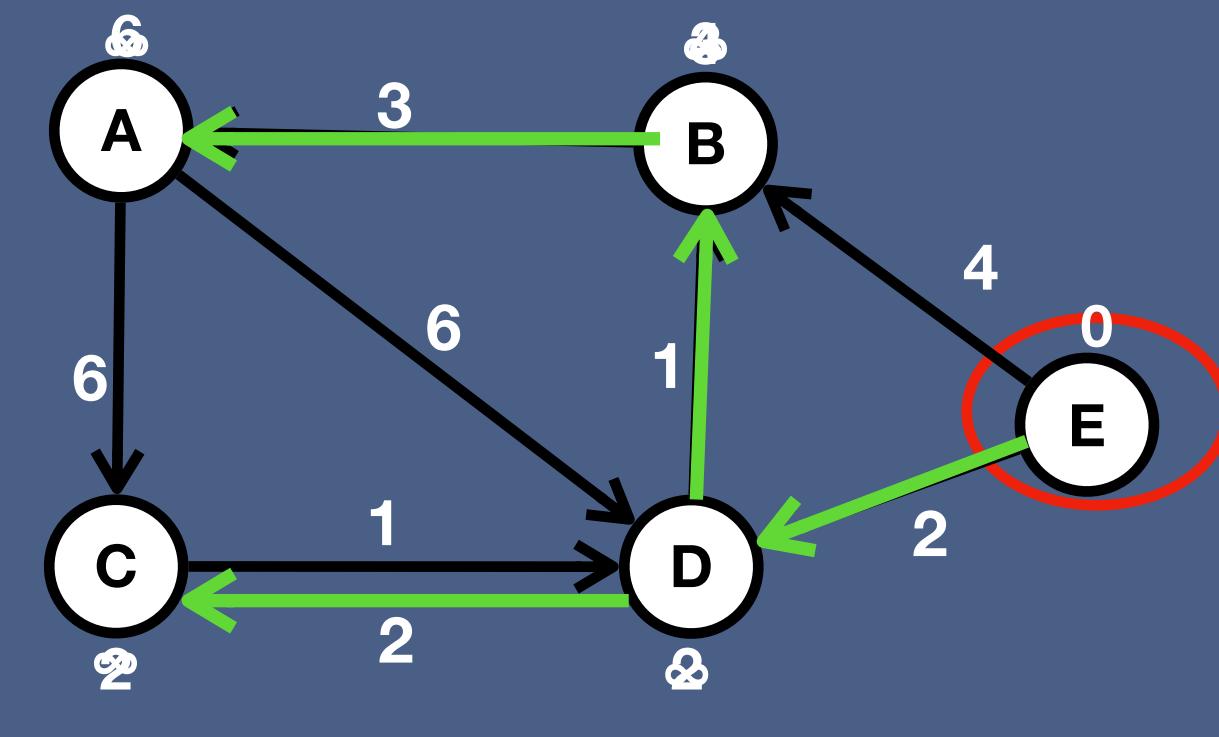
- Five offices in different cities.
- Travel costs between these cities are known.
- Find the cheapest way to reach each office from **every other office**



# All Pair Shortest Path Problem

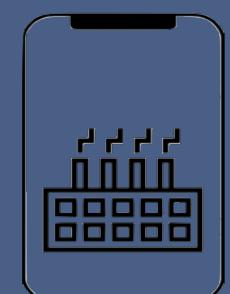


# Dry Run for All Pair Shortest Path Problem

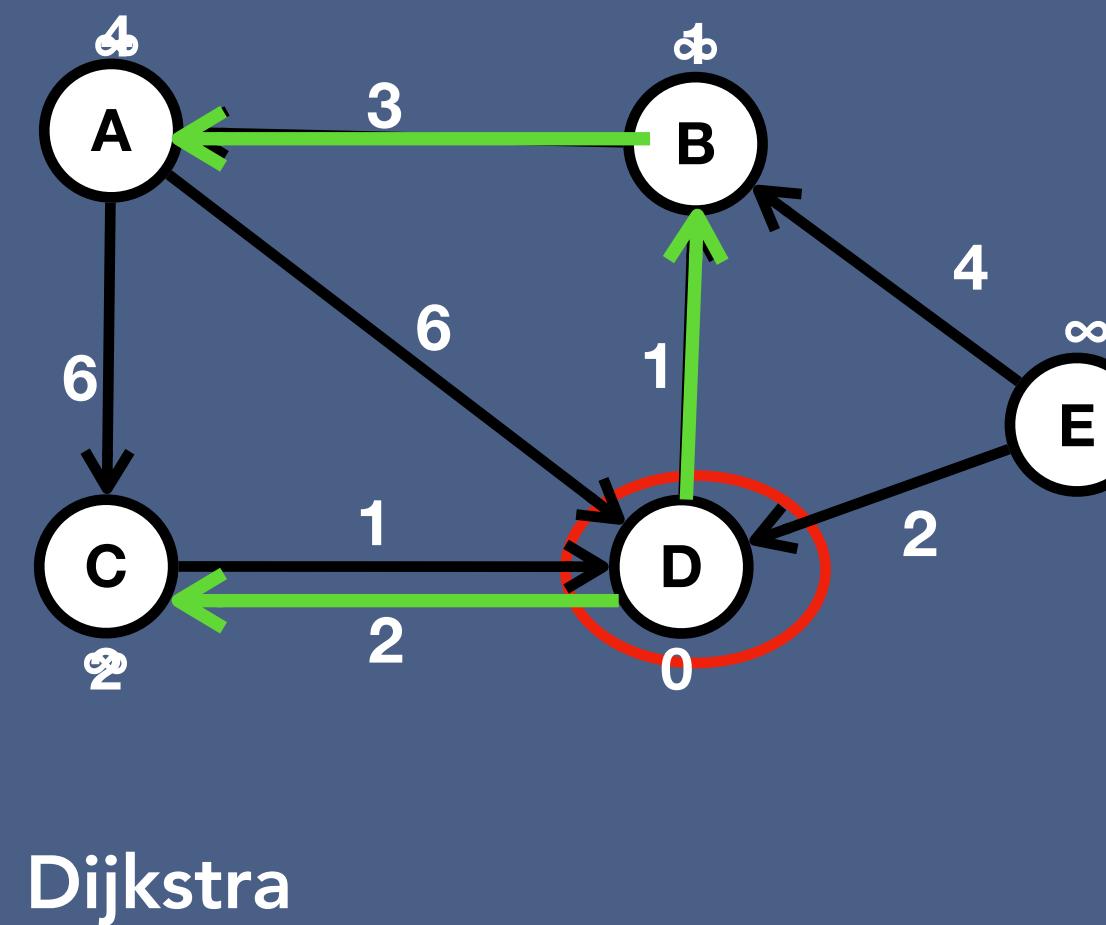


Dijkstra

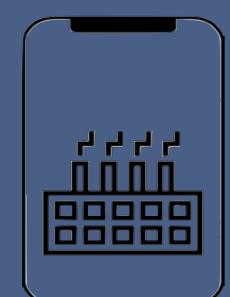
Source Vertex E	Path
A	E->D->B->A
B	E->D->B
C	E->D->C
D	E->D
E	-



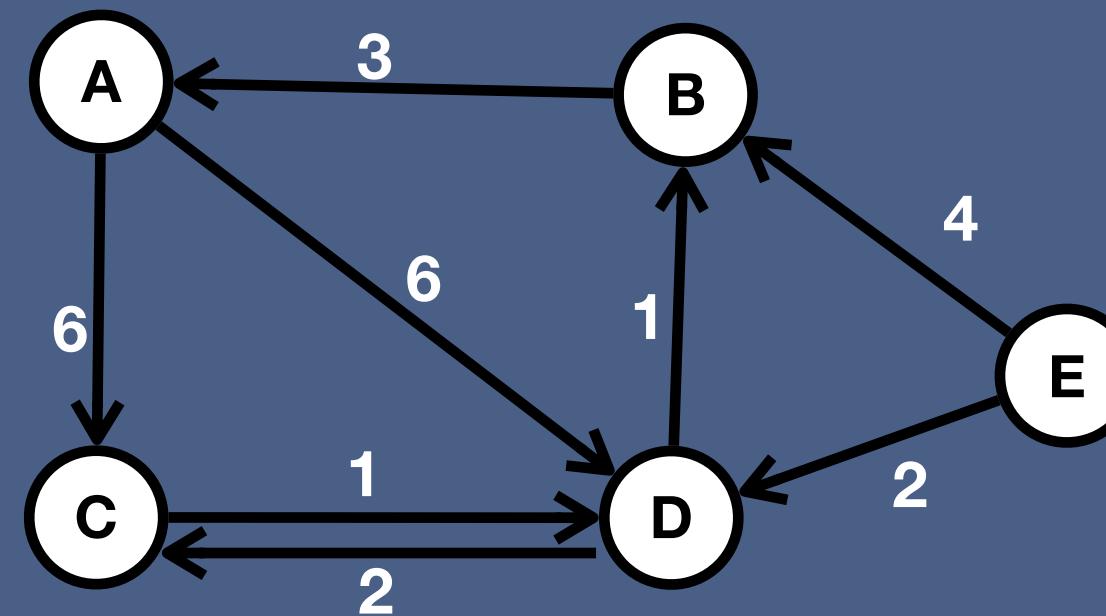
# Dry Run for All Pair Shortest Path Problem



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



# Dry Run for All Pair Shortest Path Problem



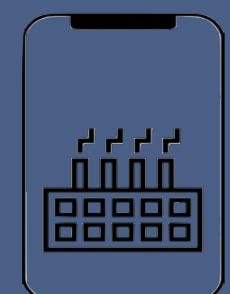
Dijkstra , BFS and Bellman Ford

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

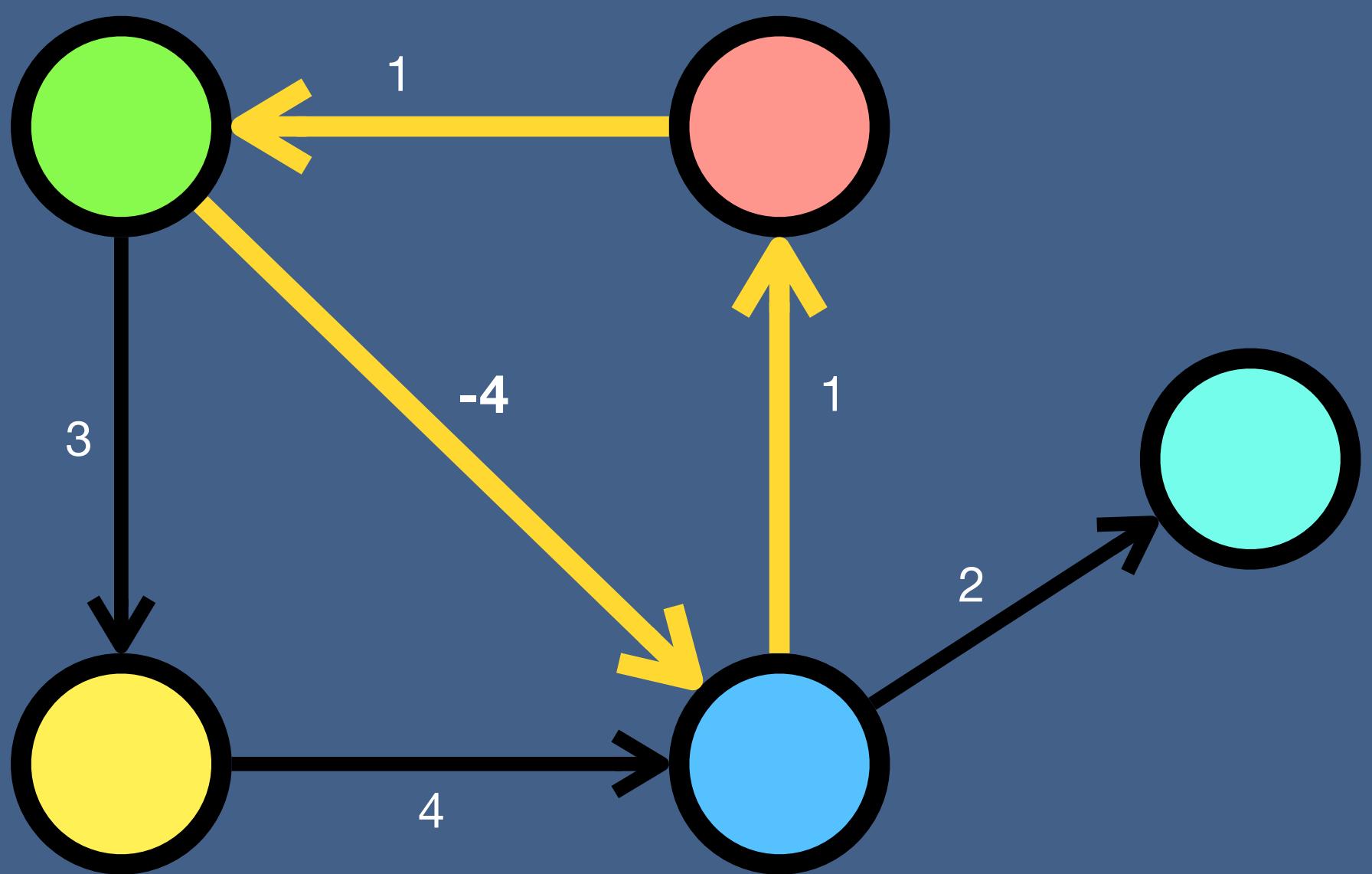
Source Vertex C	Path
A	C->D->B->A
B	C->D->B
C	-
D	C->D
E	N/A

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

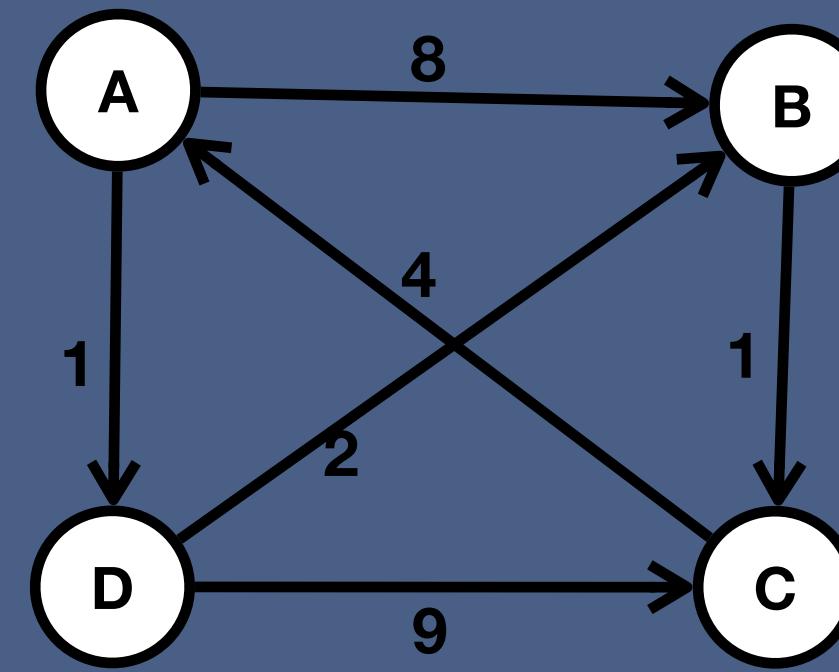
Source Vertex A	Path
A	-
B	A->D->B
C	A->D->C
D	A->D
E	N/A



# Floyd Warshall Algorithm



# 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

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

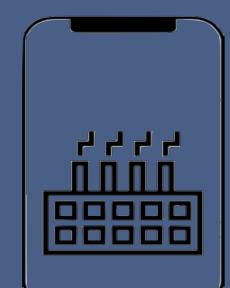
If  $D[C][B] > D[C][A] + D[A][B]$ :  
 $D[C][B] = D[C][A] + D[A][B]$

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

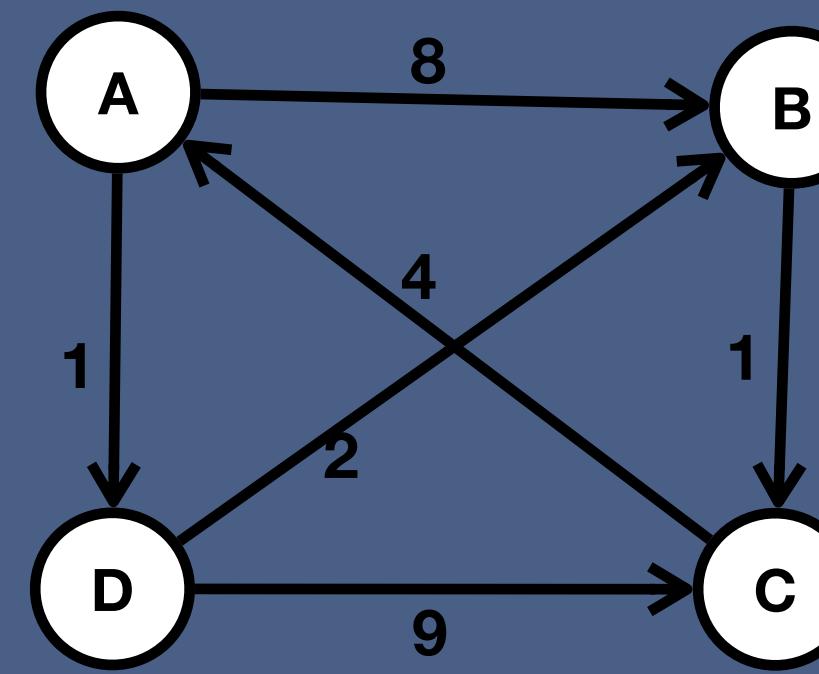
C  $\rightarrow$  D

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 Algorithm



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

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

$D \rightarrow C$

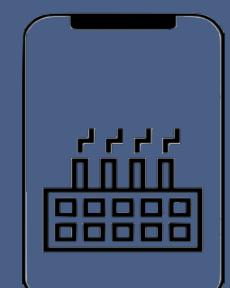
If  $D[D][C] > D[D][B] + D[B][C]$ :  
 $D[D][C] = D[D][B] + D[B][C]$

$$9 > 2 + 1 = 3$$

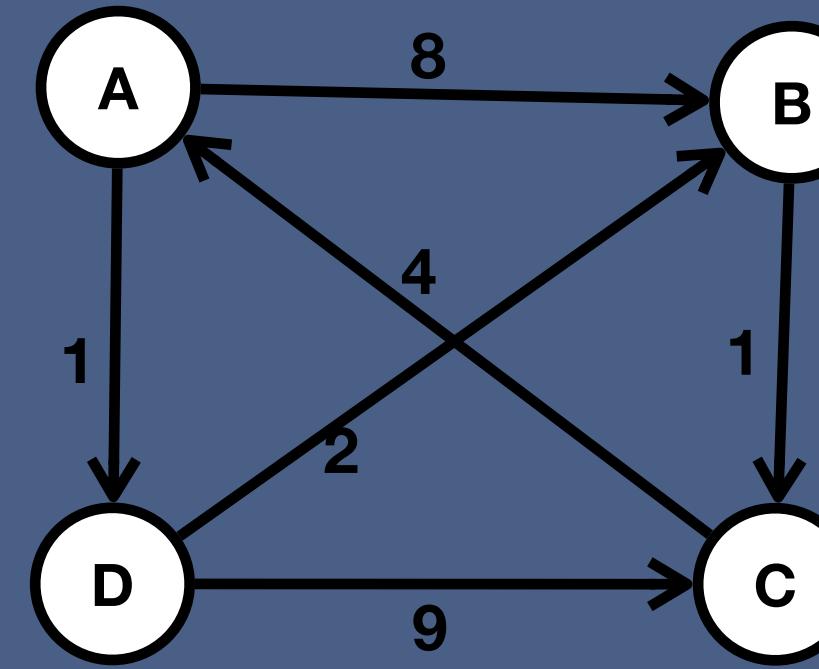
Given				
	A	B	C	D
A	0	8	$\infty$	1
B	$\infty$	0	1	$\infty$
C	4	$\infty$	0	$\infty$
D	$\infty$	2	9	0

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



# Floyd Warshall Algorithm



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

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

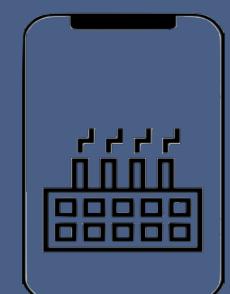
Iteration 1				
via A	A	B	C	D
A	0	8	$\infty$	1
B	$\infty$	0	1	$\infty$
C	4	$4+8=12$	0	$4+1=5$
D	$\infty$	2	9	0

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

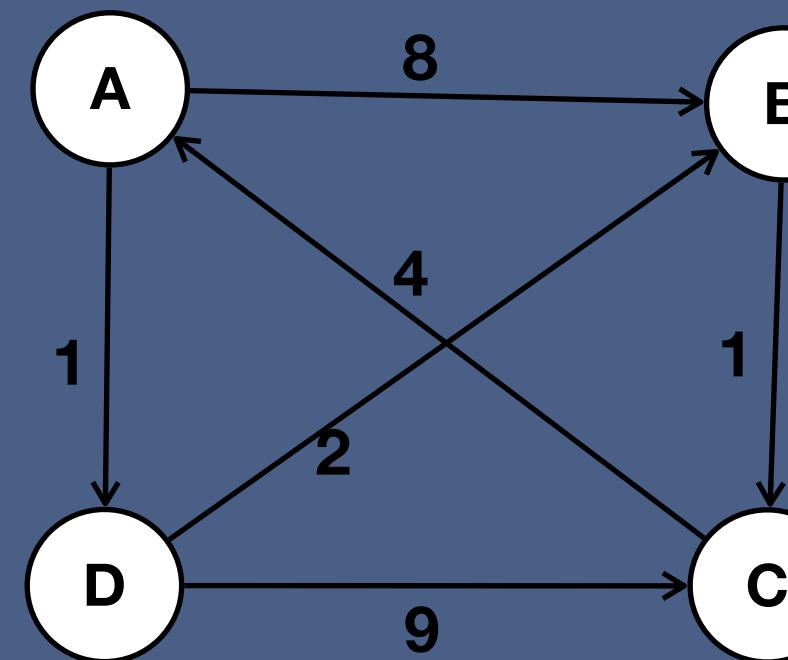
Iteration 3				
via C	A	B	C	D
A	0	8	9	1
B	$1+4=5$	0	1	$1+4+1=6$
C	4	12	0	5
D	$2+1+4=7$	2	3	0

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

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

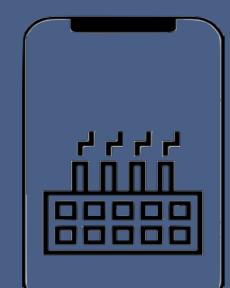
		Final answer			
		A	B	C	D
A	0	3	4	1	
B	5	0	1	6	
C	4	7	0	5	
D	7	2	3	0	

- The vertex is not reachable
- Two vertices are directly connected

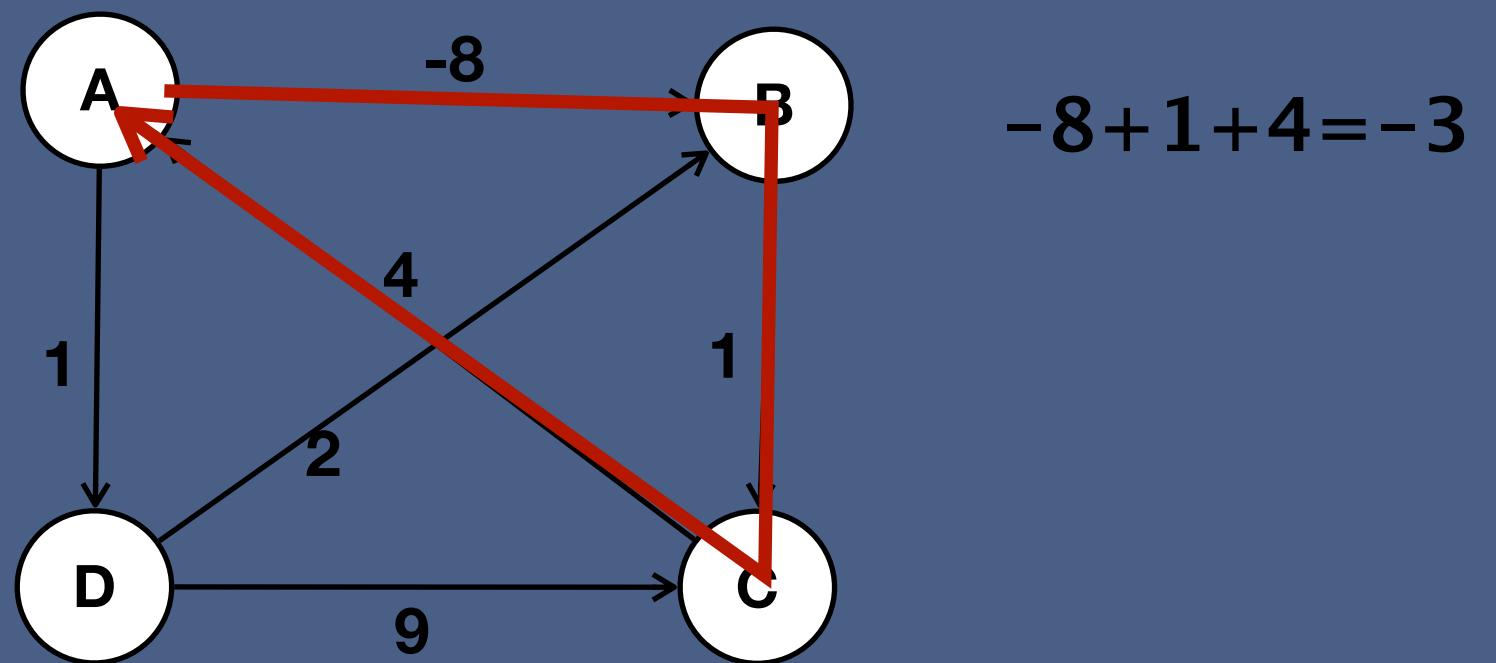
This is the best solution

It can be improved via other vertex

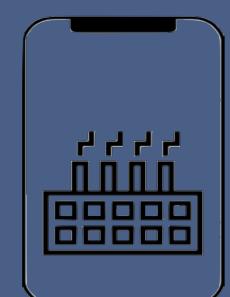
- Two vertices are connected via other vertex



## Floyd Warshall negative cycle

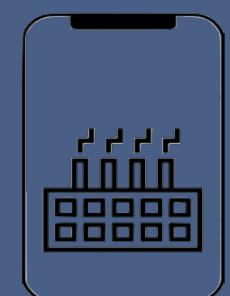


- To go through cycle we need to go via negative cycle participating vertex at least twice
- FW never runs loop twice via same vertex
- Hence, FW can never detect a negative cycle



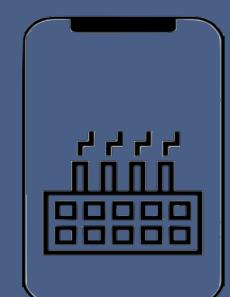
# Which algorithm to use for APSP?

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?

	BFS	Dijkstra	Bellman Ford	Floyd Warshall
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	OK	OK	OK
	Use this as time complexity is good and easy to implement	Not use as priority queue slows it	Not use as time complexity is bad	Can be used
Weighted graph	X	OK	OK	OK
	Not supported	Can be used	Not use as time complexity is bad	Can be preferred as implementation easy
Negative Cycle	X	X	OK	X
	Not supported	Not supported	Use this as others not support	No supported



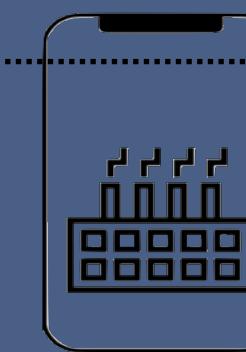
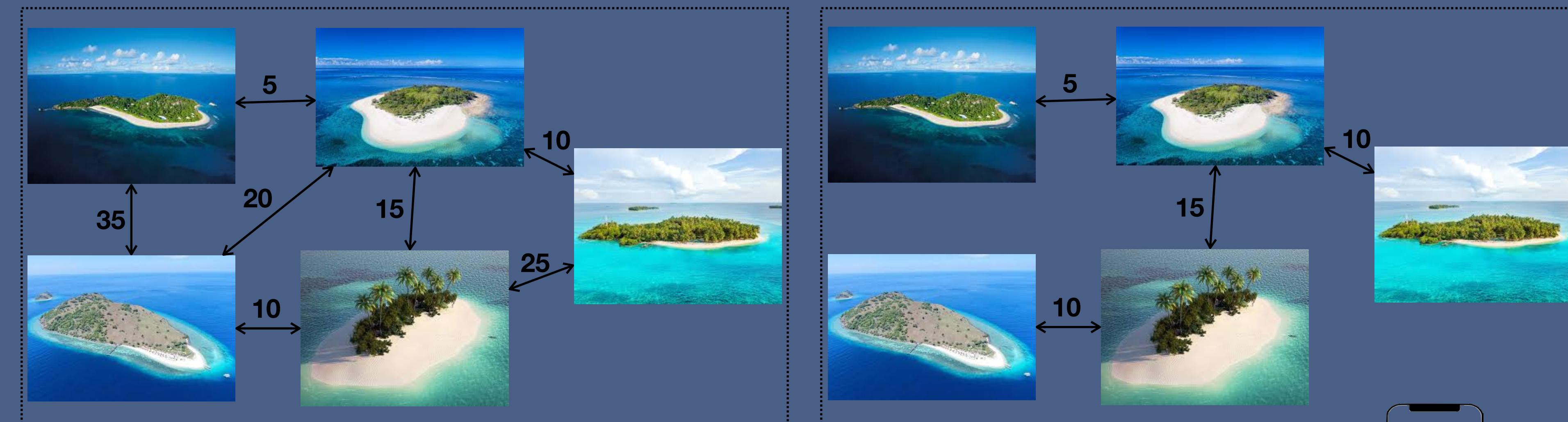
# Minimum Spanning Tree

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

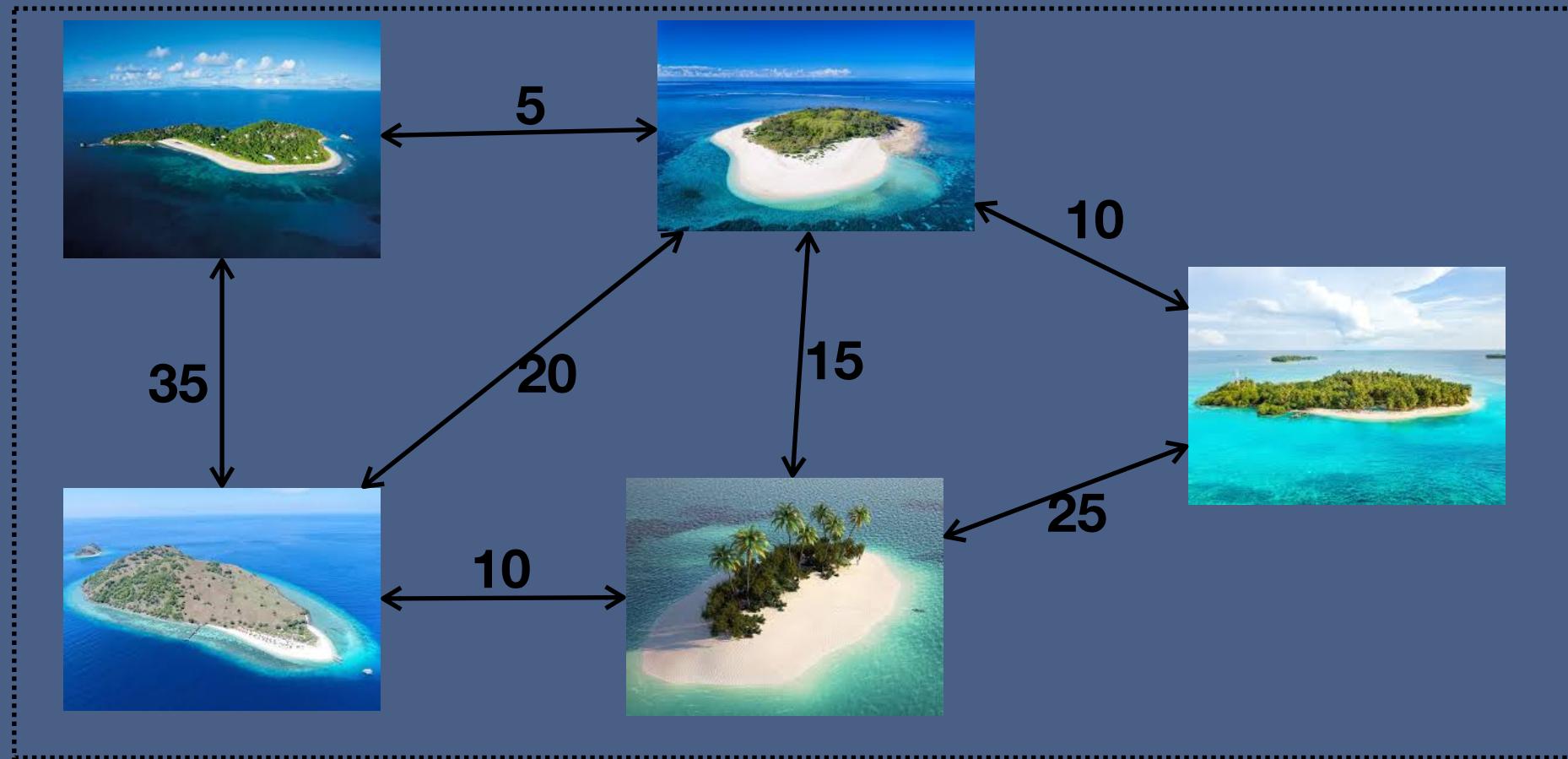
Real life problem

- 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

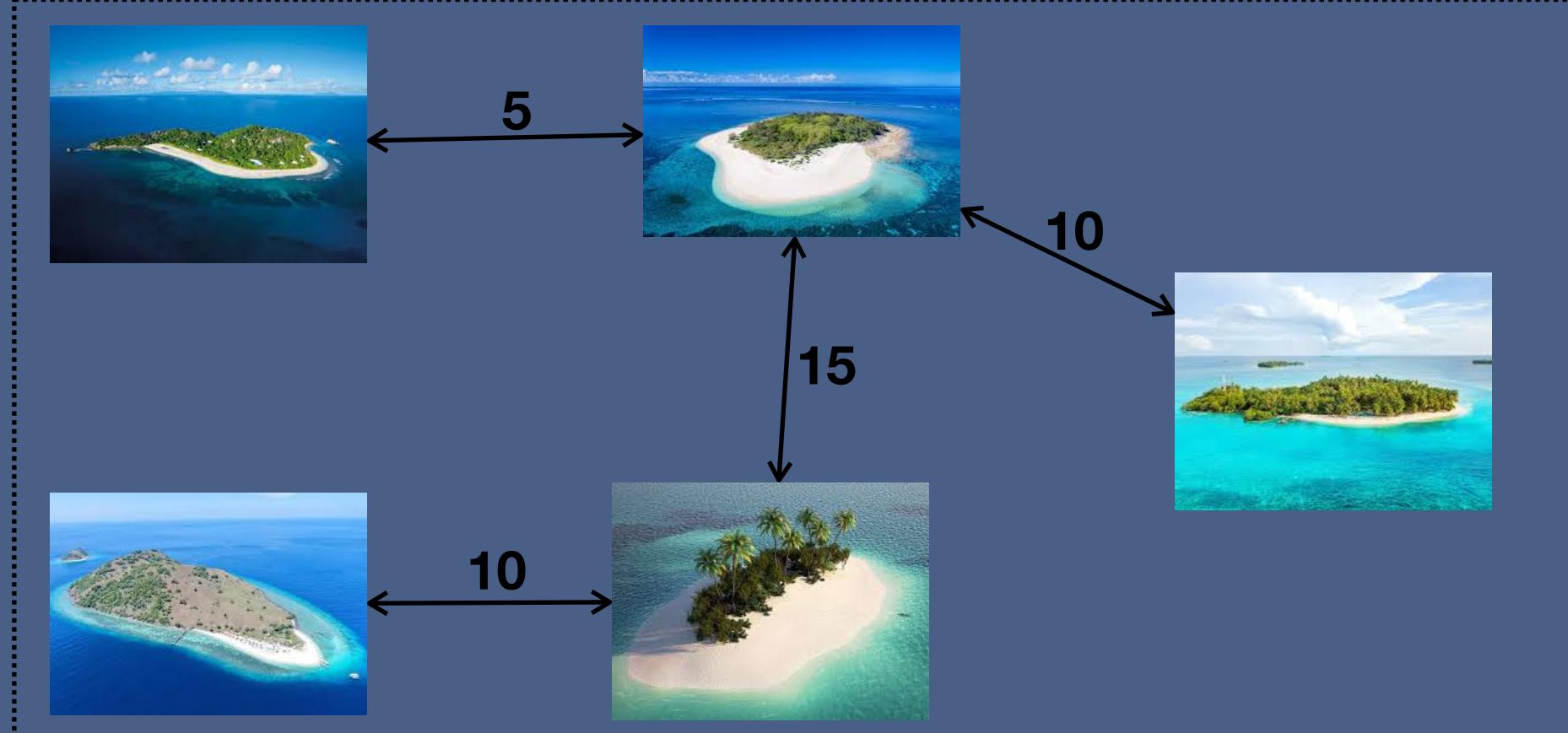


# Minimum Spanning Tree

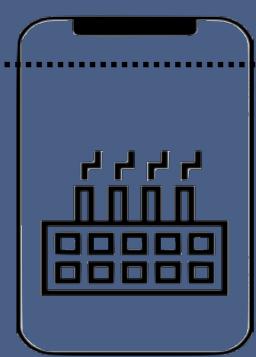
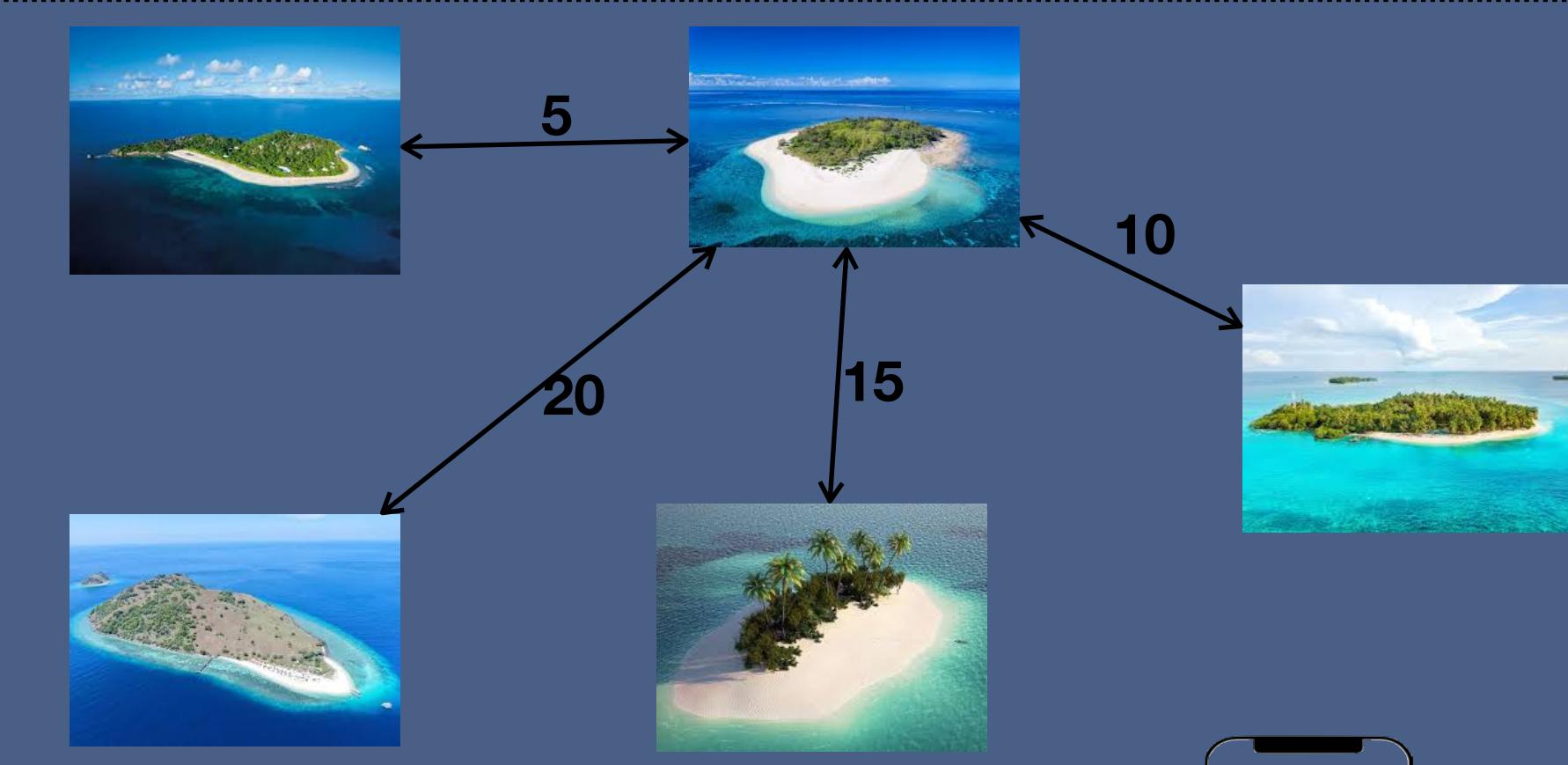
Real life problem



Minimum spanning Tree



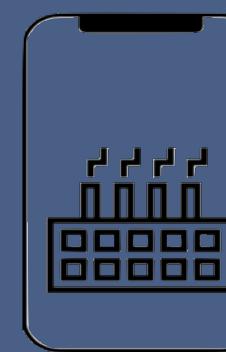
Single source shortest path



# 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

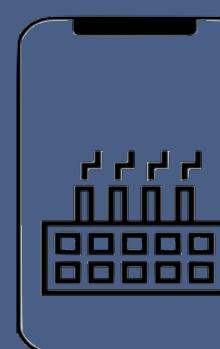
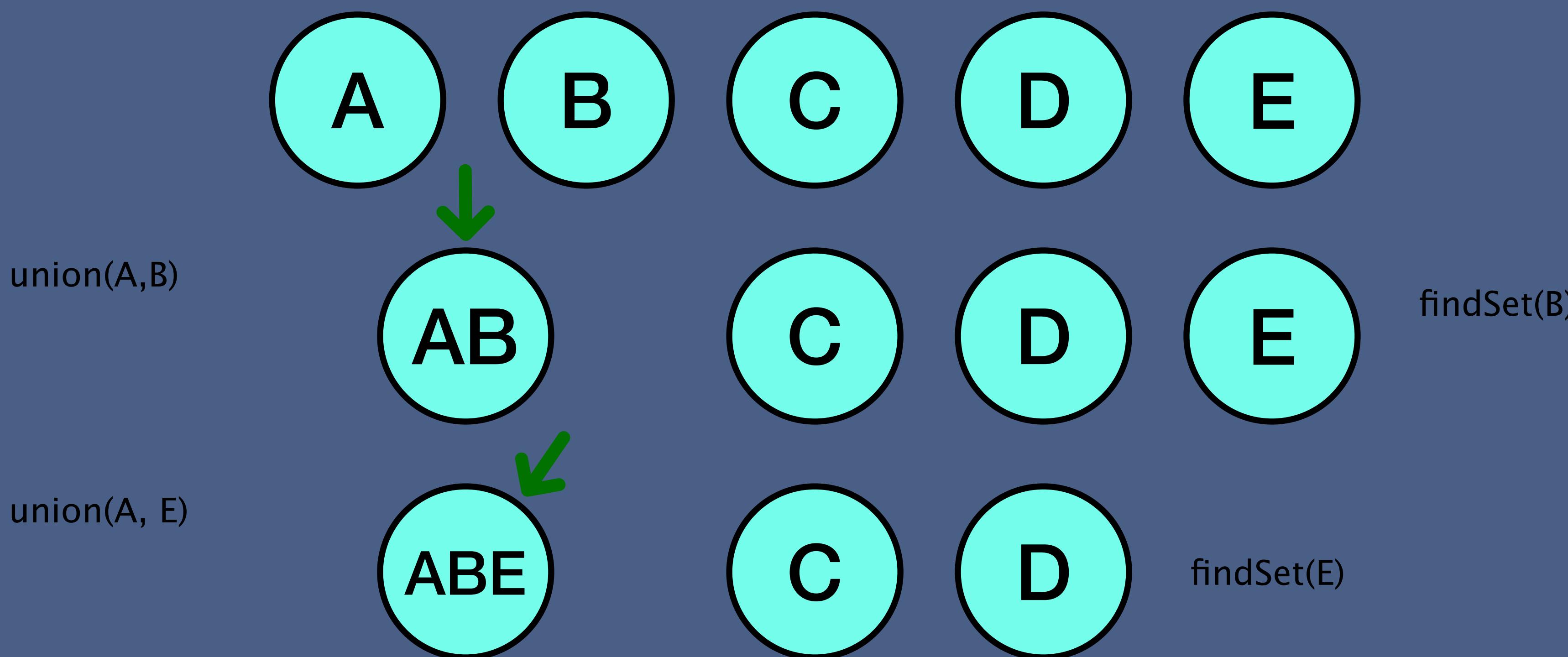


# Disjoint Set

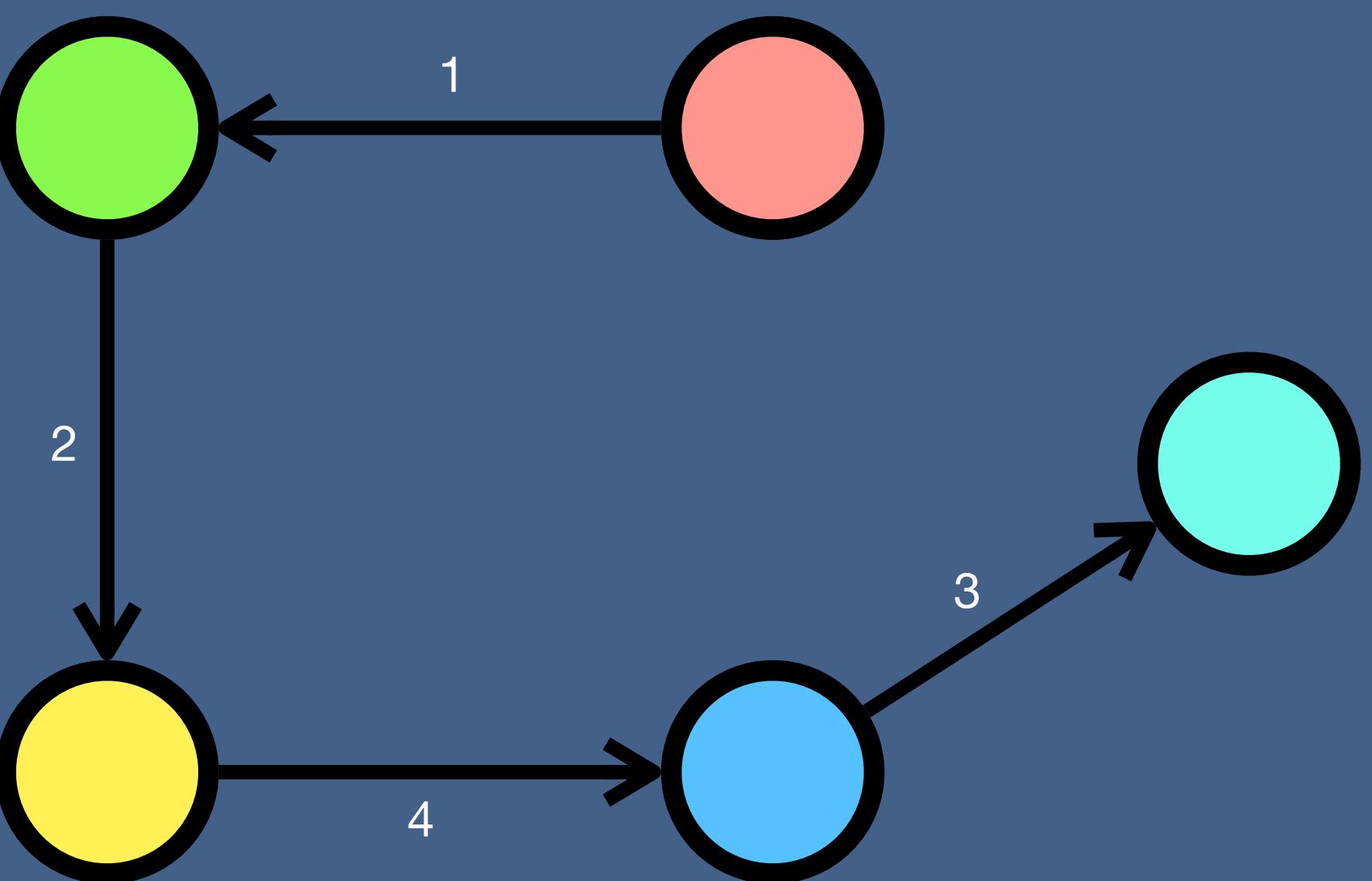
makeSet(N) : used to create initial set

union(x,y): merge two given sets

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



# Kruskal's Algorithm

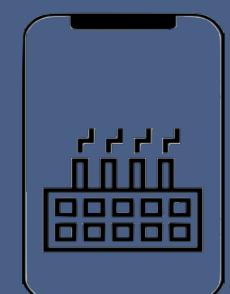
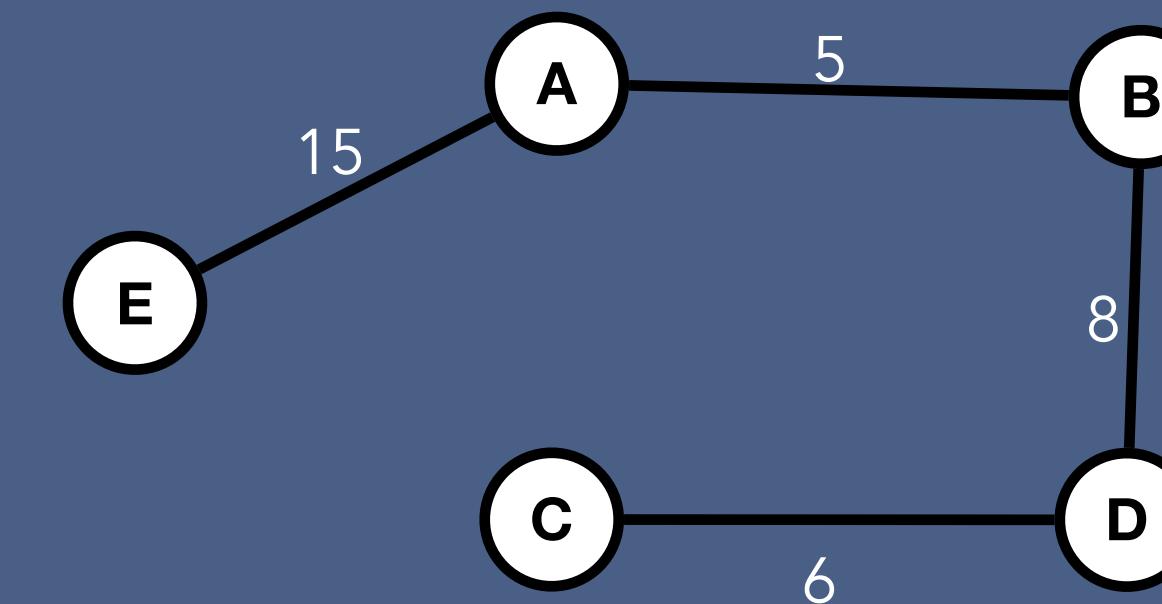
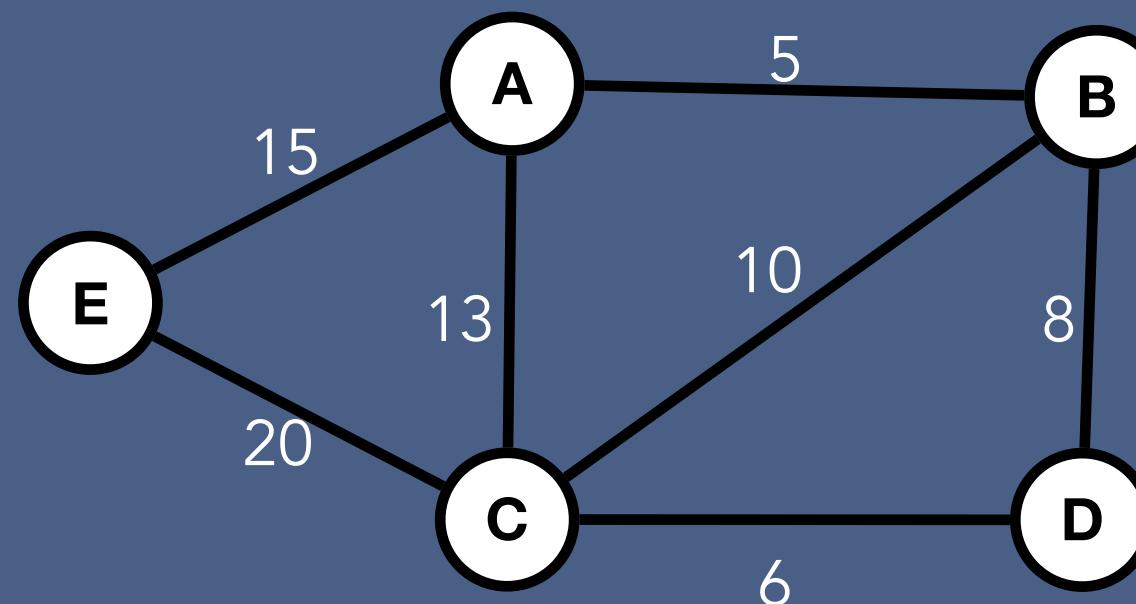


# Kruskal's Algorithm

It is a greedy algorithm

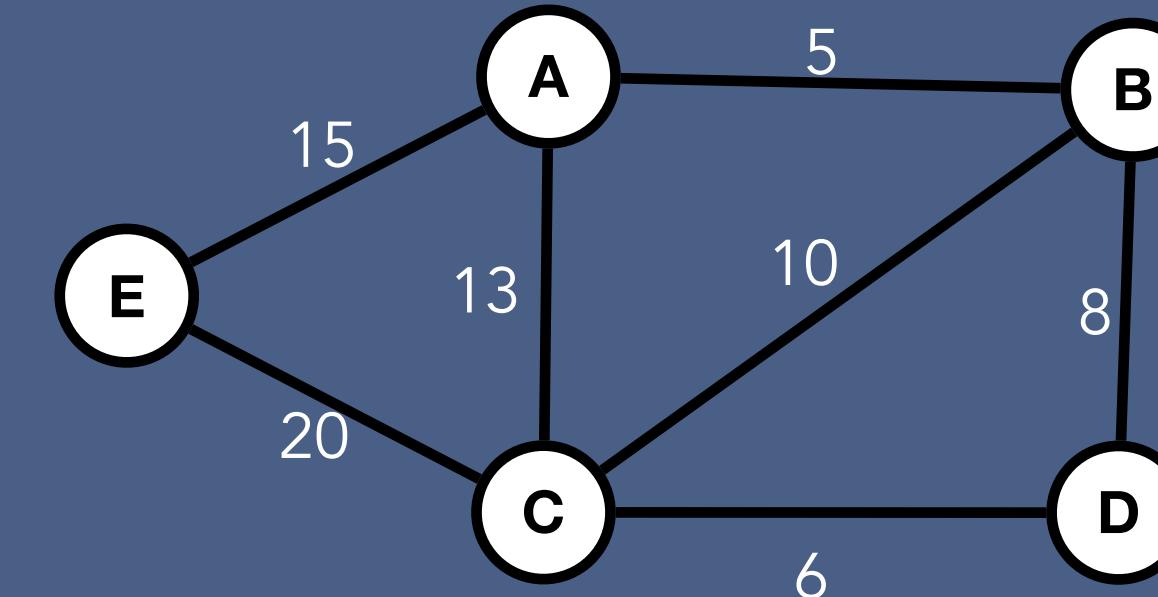
It finds a minimum spanning tree for weighted undirected graphs in two steps

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

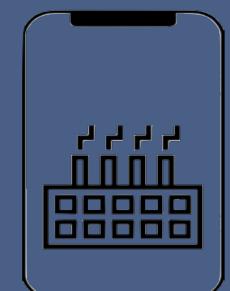
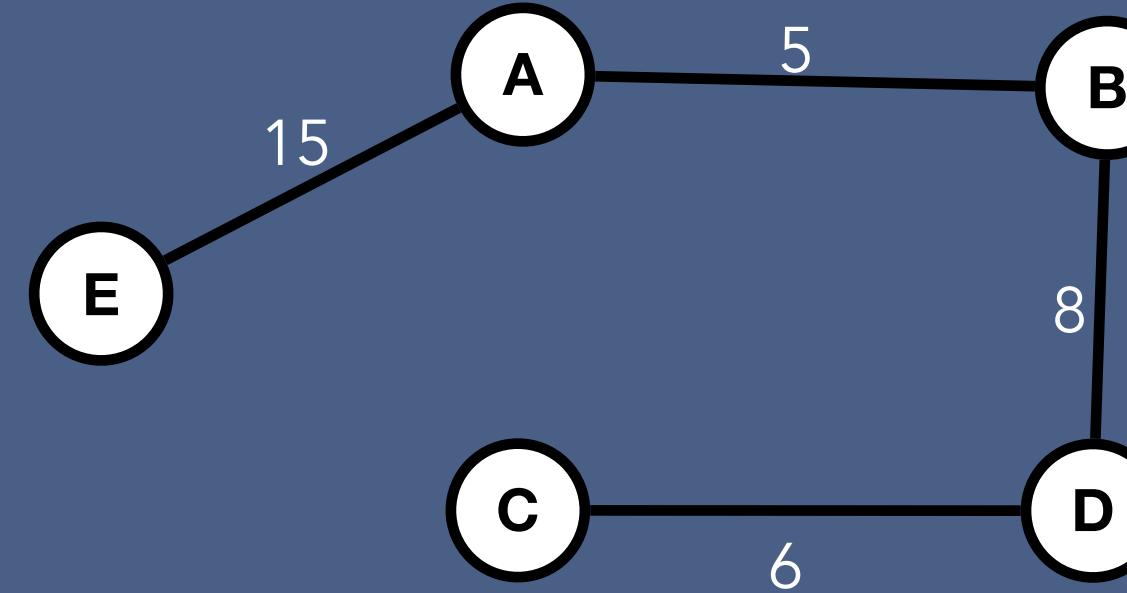


# Kruskal's Algorithm Pseudocode

```
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)
```



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

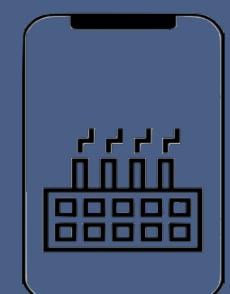


# Time and Space Complexity of Kruskal's Algorithm

```
Kruskal(G)
for each vertex      ..... makeSet(v) ..... → O(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) } ..... → O(ev)
```

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

Space complexity :  $O(V+E)$

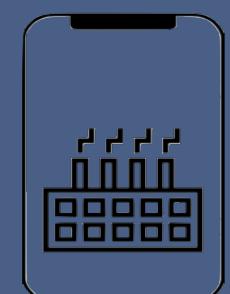
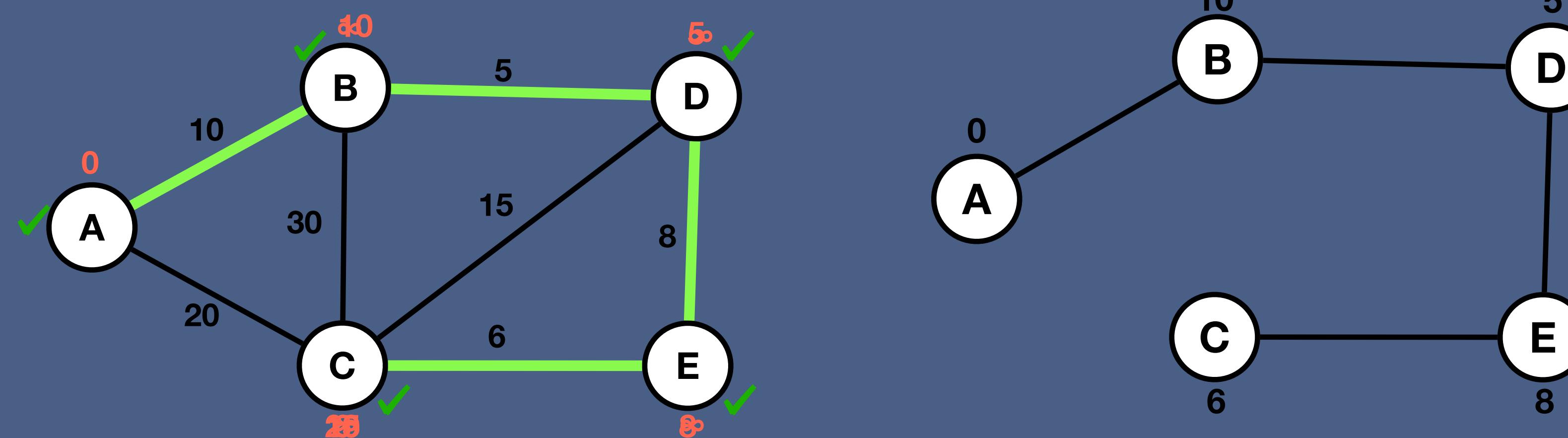


# Prim's Algorithm

It is a greedy algorithm

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

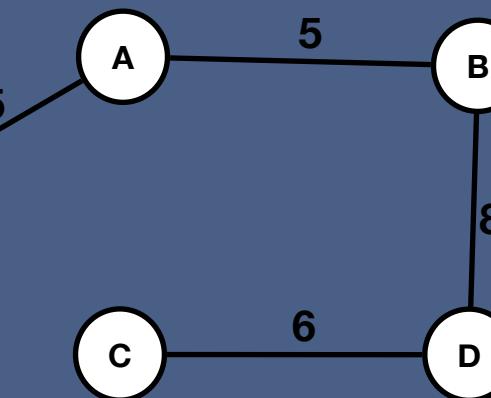
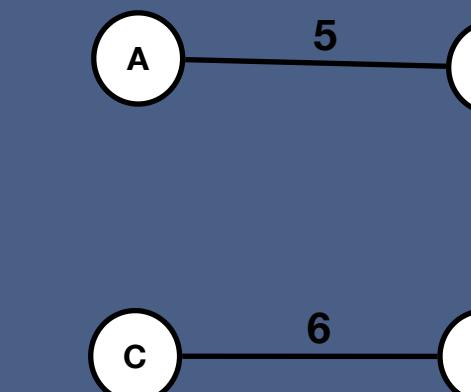
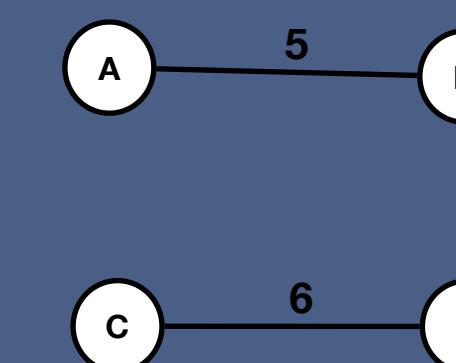
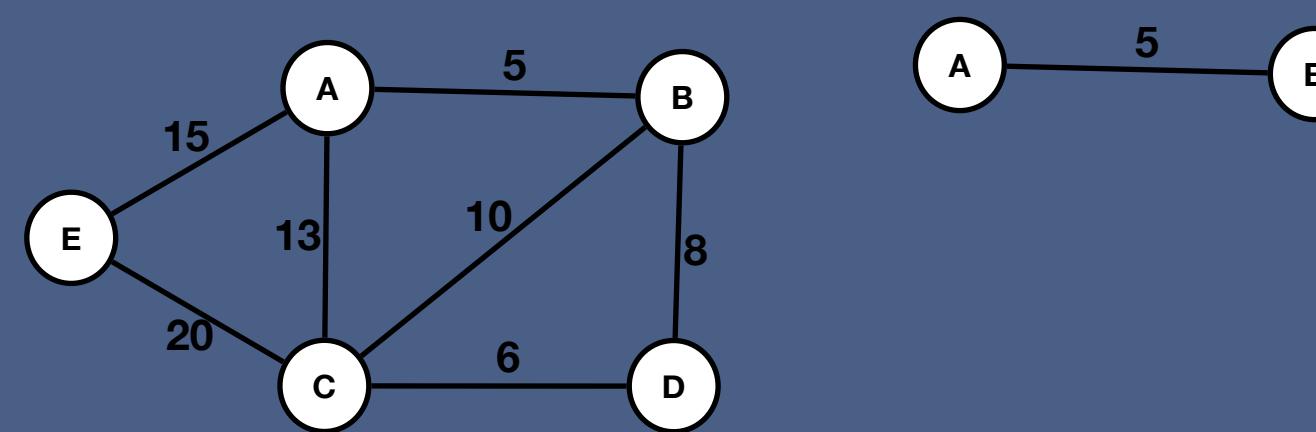
1. Take any vertex as a source set its weight to 0 and all other vertices' weight to infinity
2. For every adjacent vertices if the current weight is more than current edge then we set it to current edge
3. Then we mark current vertex as visited
4. Repeat these steps for all vertices in increasing order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



# Kruskal vs Prim

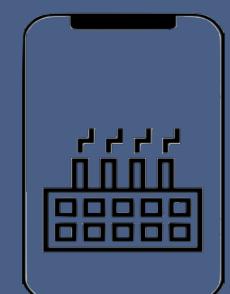
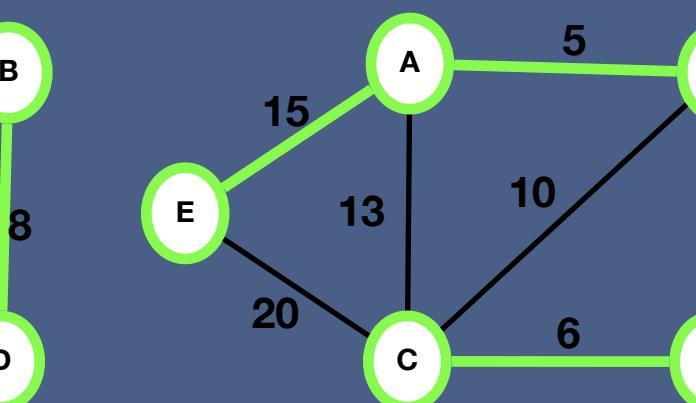
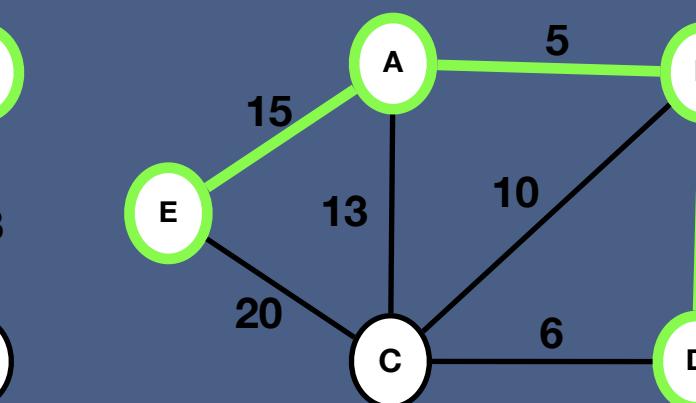
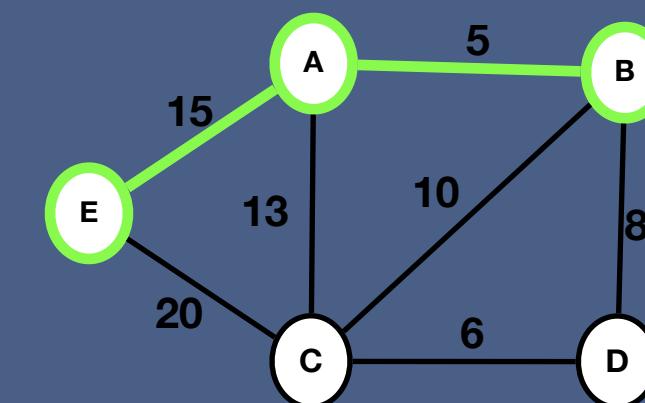
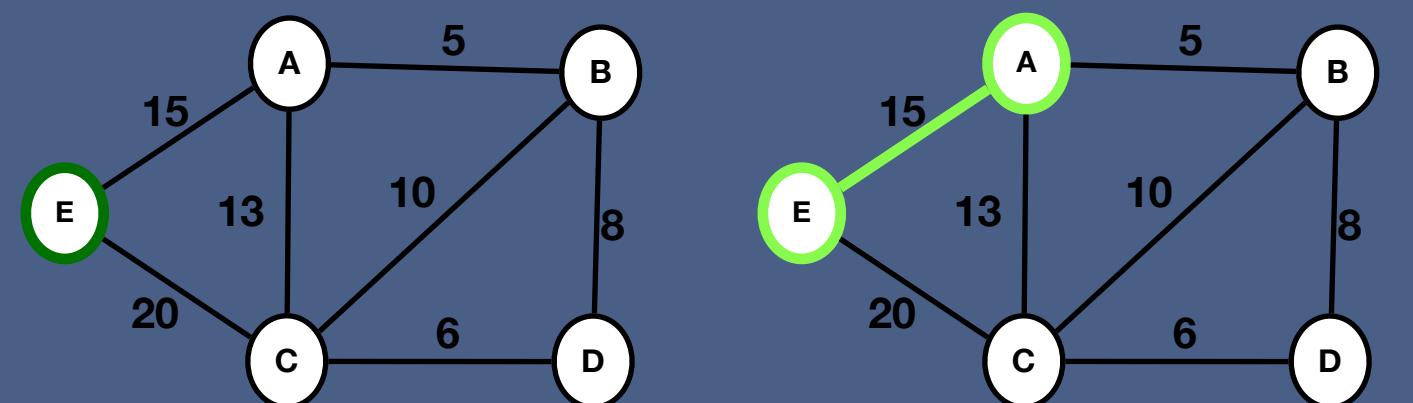
## Kruskal

- Concentrates on Edges
- Finalize edge in each iteration



## Prim's

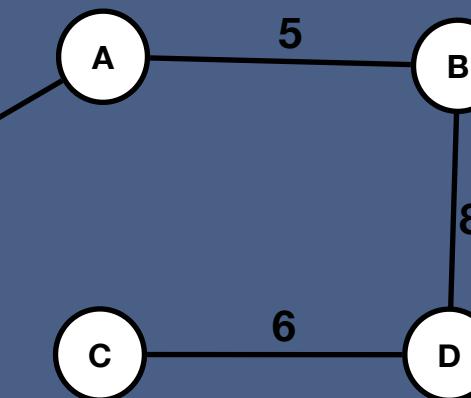
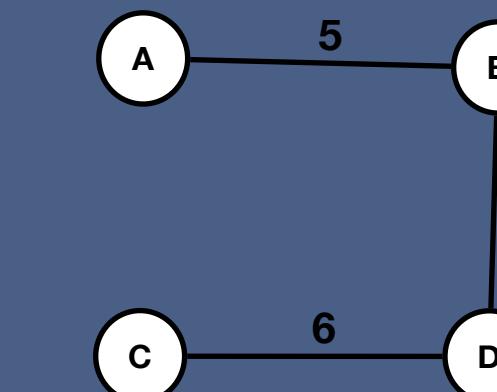
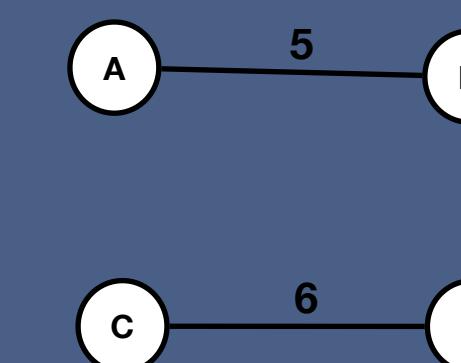
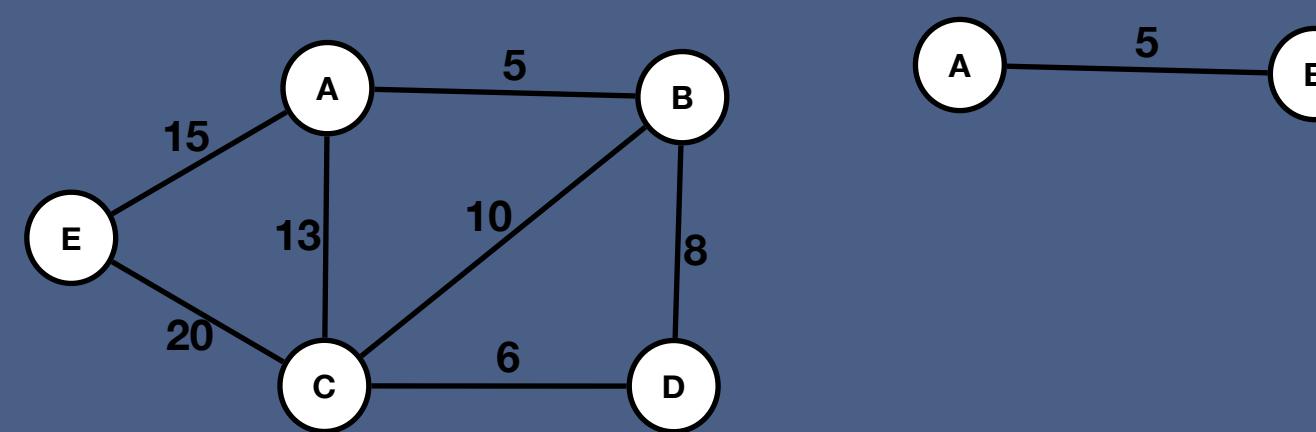
- Concentrates on Vertices
- Finalize Vertex in each iteration



# Kruskal vs Prim

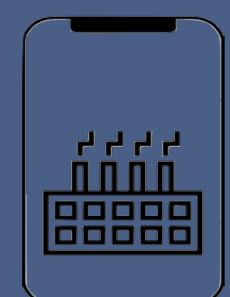
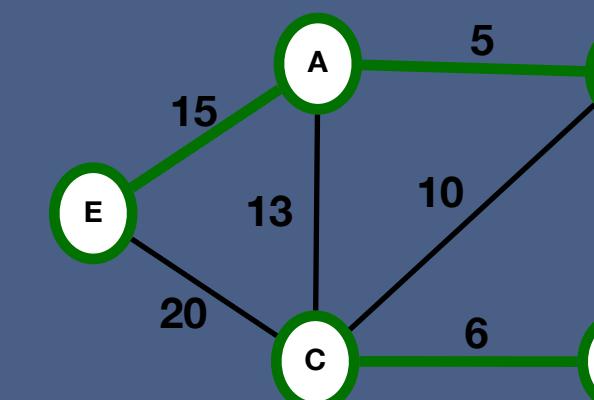
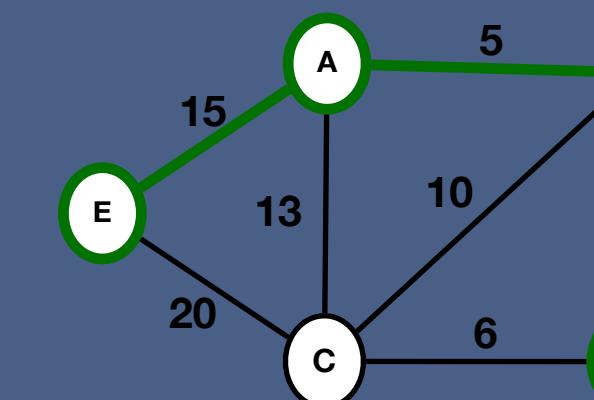
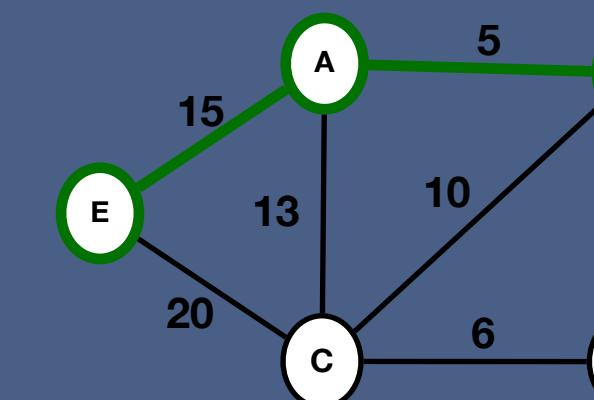
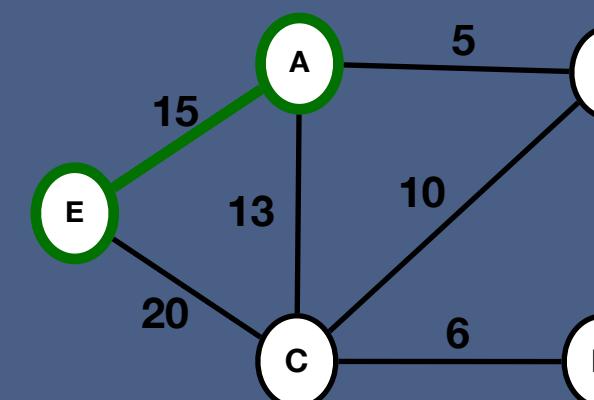
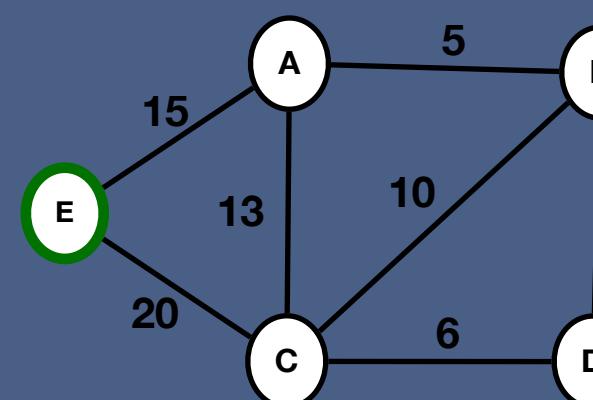
## Kruskal

- Concentrates on Edges
- Finalize edge in each iteration



## Prim's

- Concentrates on Vertices
- Finalize Vertex in each iteration



# Kruskal vs Prim

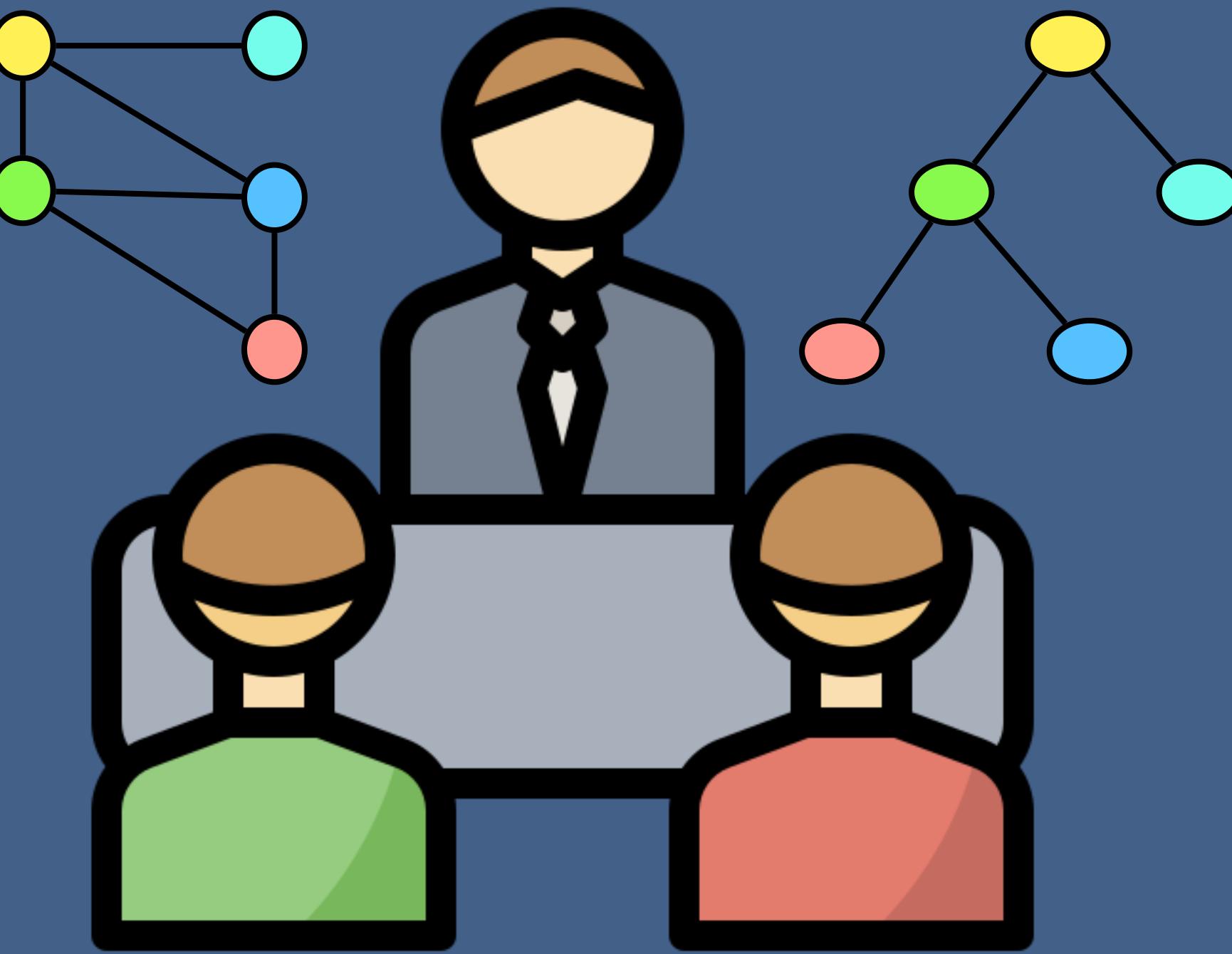
## Kruskal Applications

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

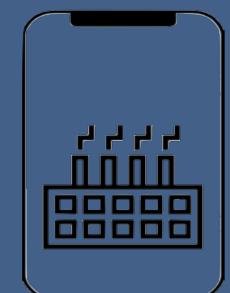
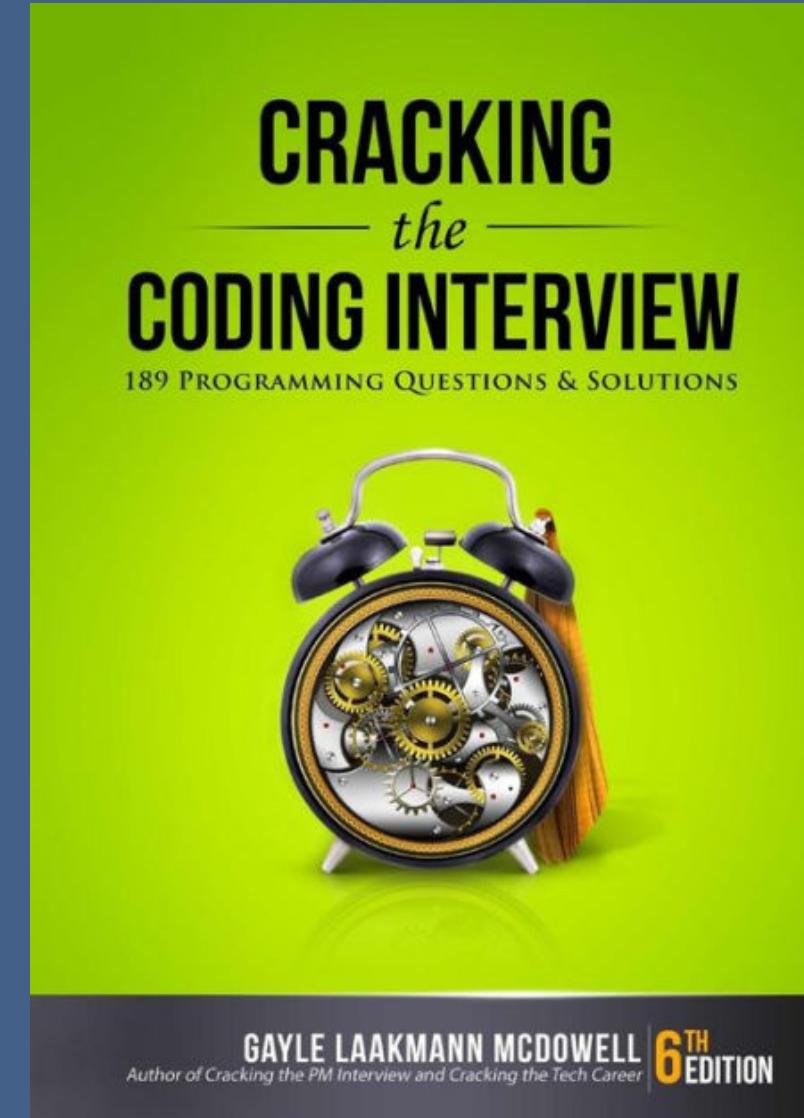
## Prim's Applications

- 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).

# Tree and Graph Interview Questions



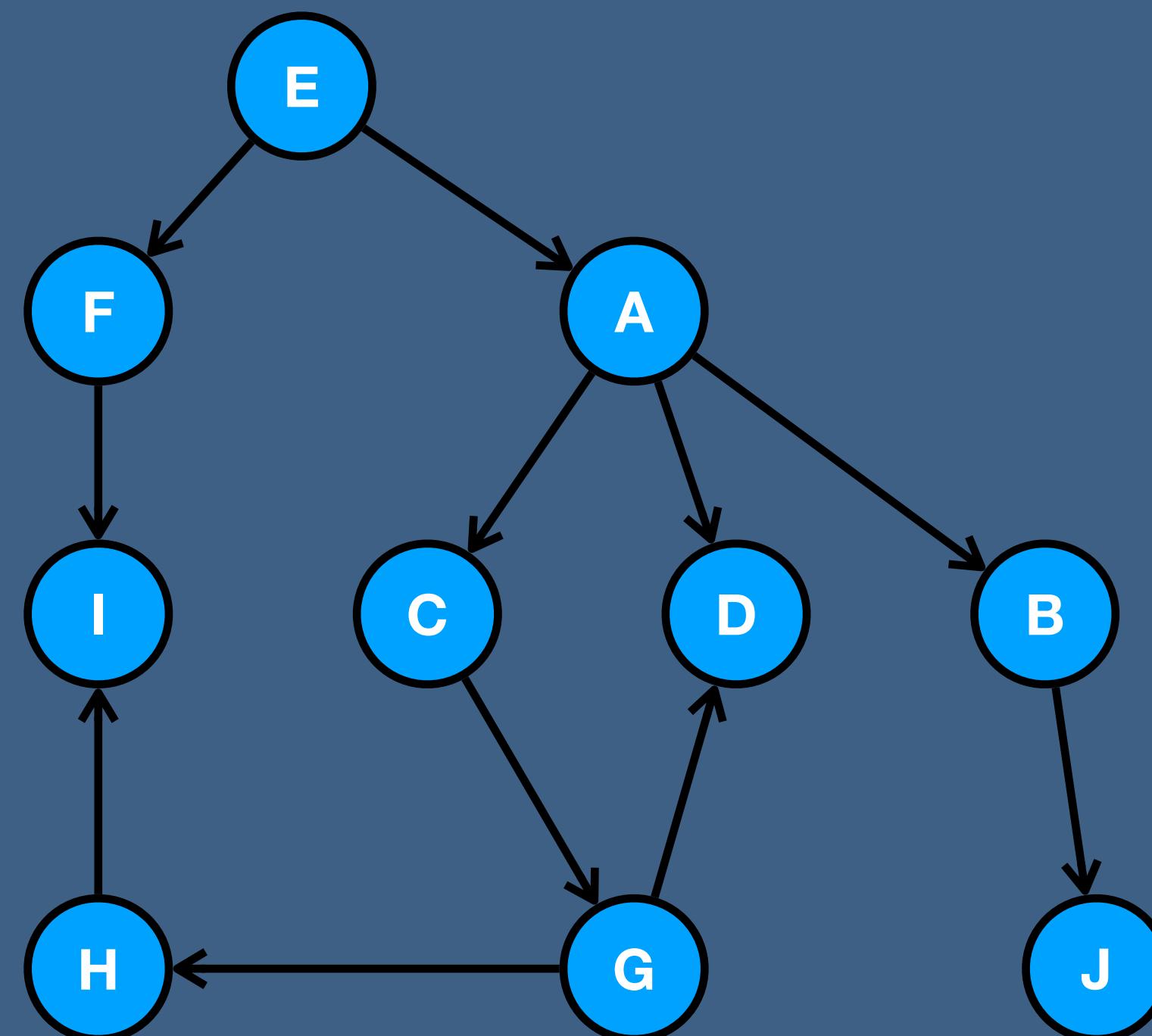
# Tree and Graph Interview Questions



# 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.



$A \rightarrow E$

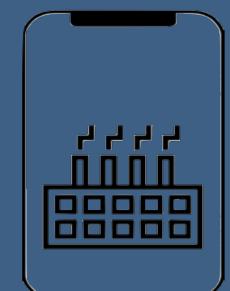
**False**

$A \rightarrow J$

**True**

$A \rightarrow I$

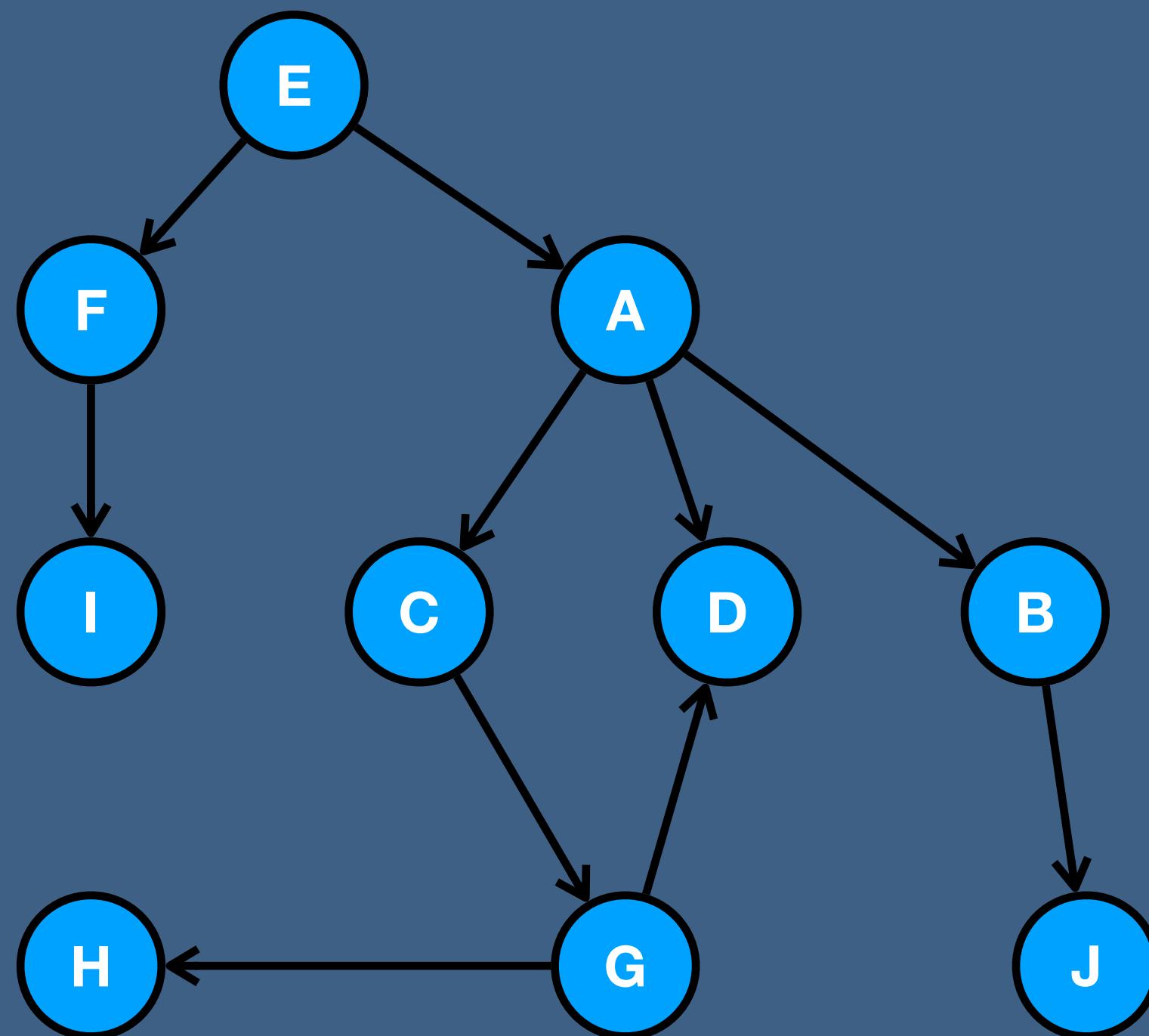
**True**



# Route Between Nodes

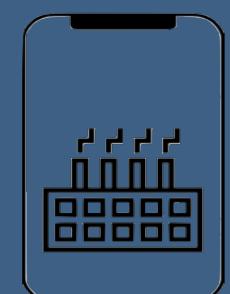
## Problem Statement:

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

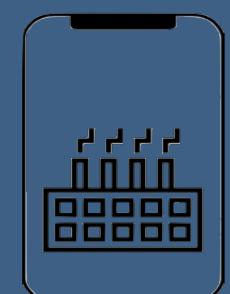
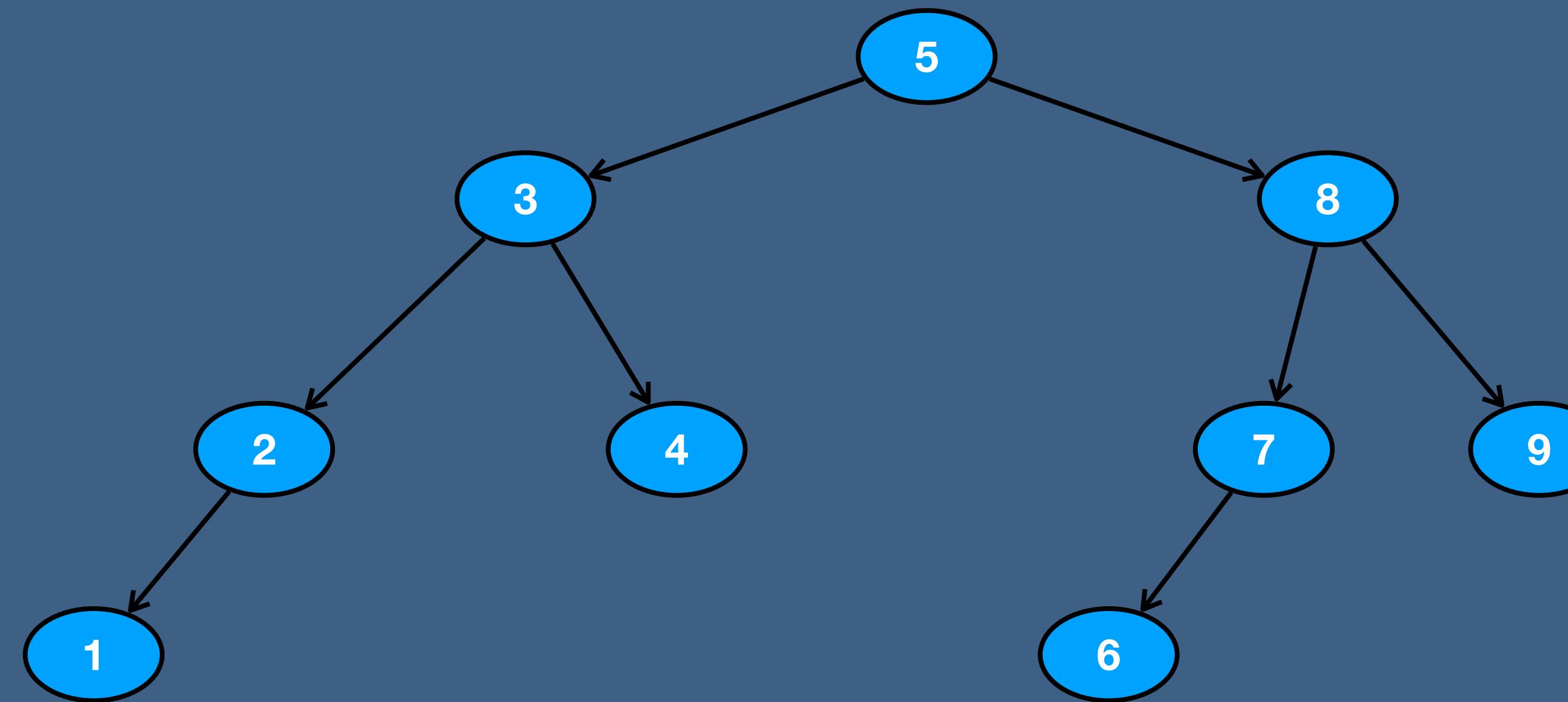
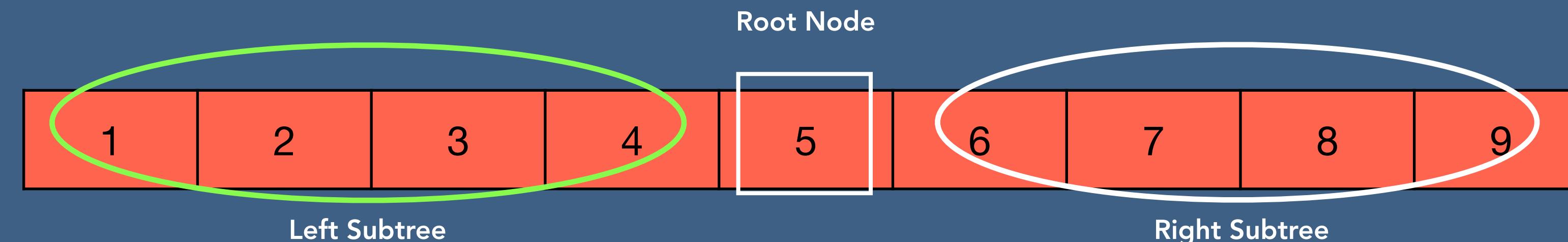
- Create function with two parameters start and end nodes
- Create queue and enqueue start node to it
- Find all the neighbors of the just enqueued node and enqueue them into the queue
- Repeat this process until the end of elements in graph
- If during the above process at some point in time we encounter the destination node, we return True.
- Mark visited nodes as visited



# Minimal Tree

## Problem Statement:

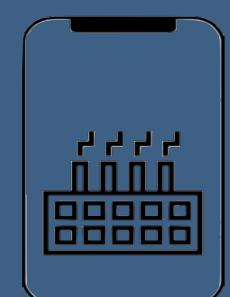
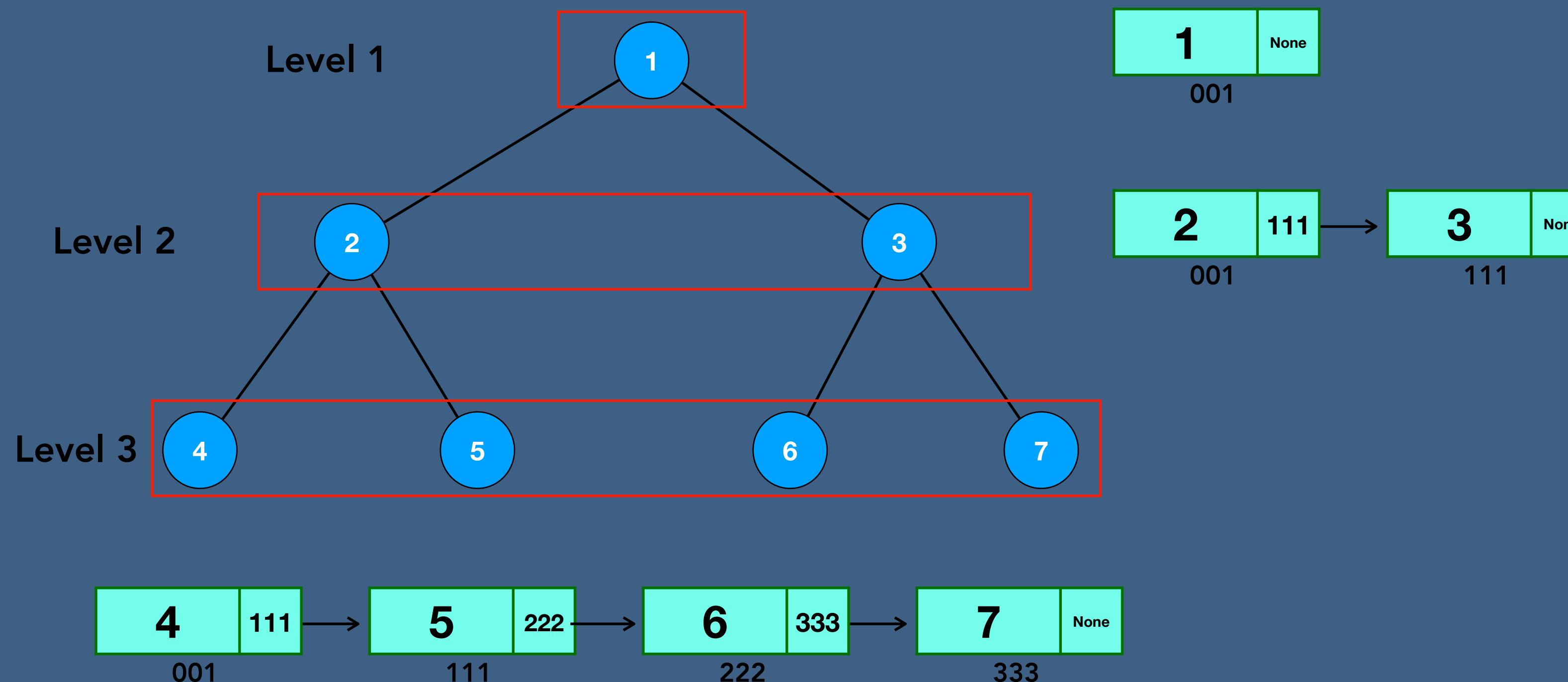
Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.



# 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)

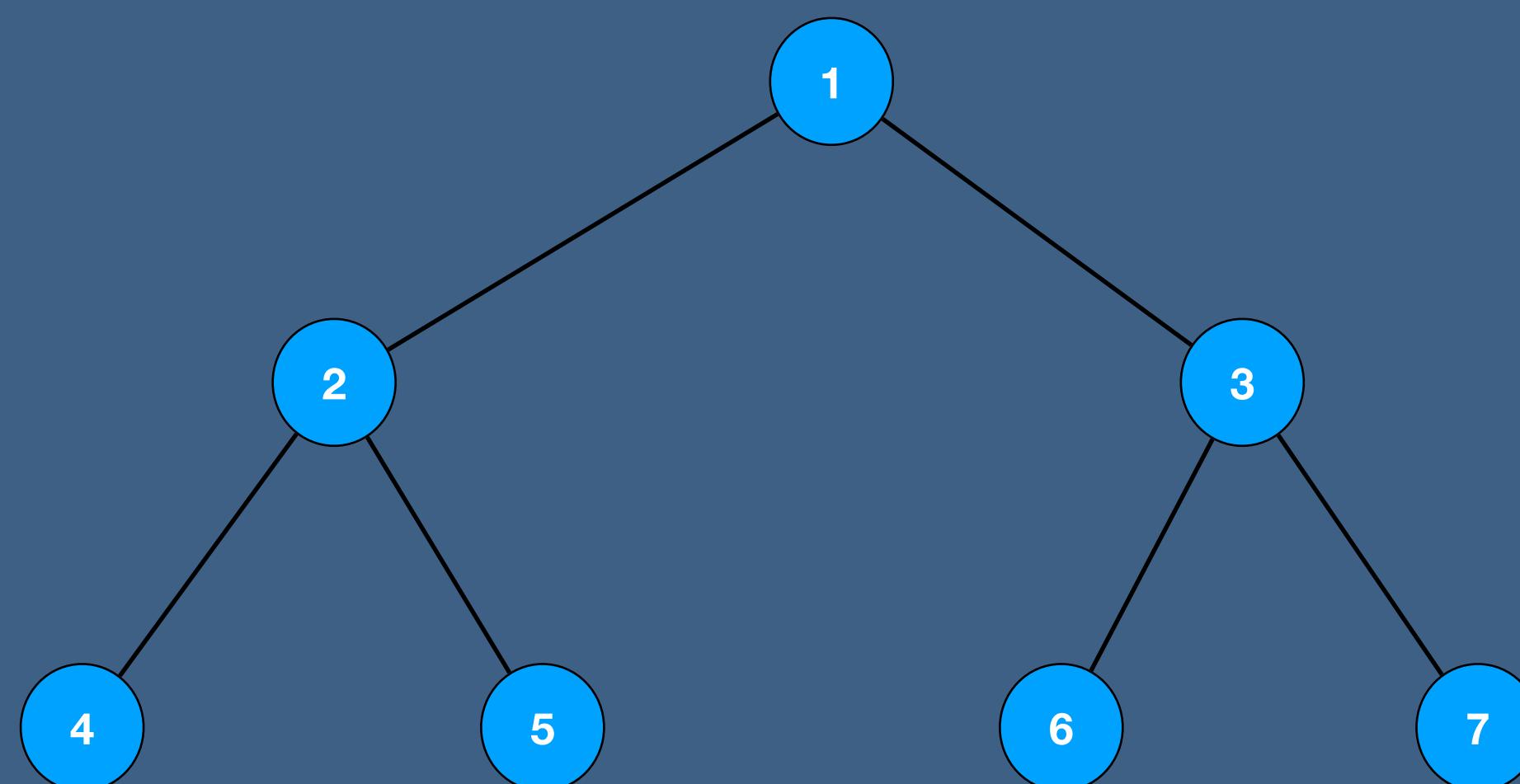


# List of Depths

## 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

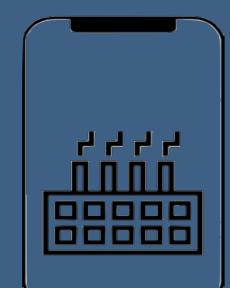
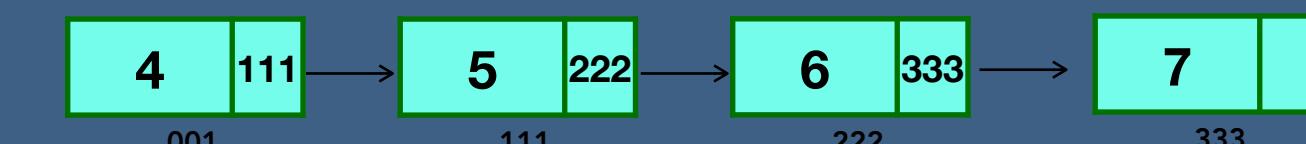
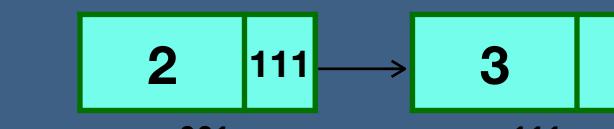
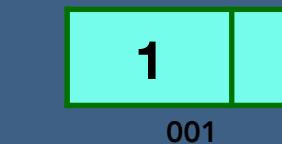
Keys

3

2

1

Values

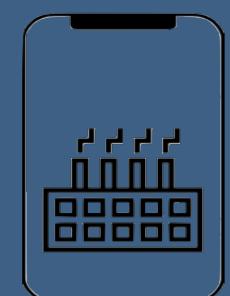
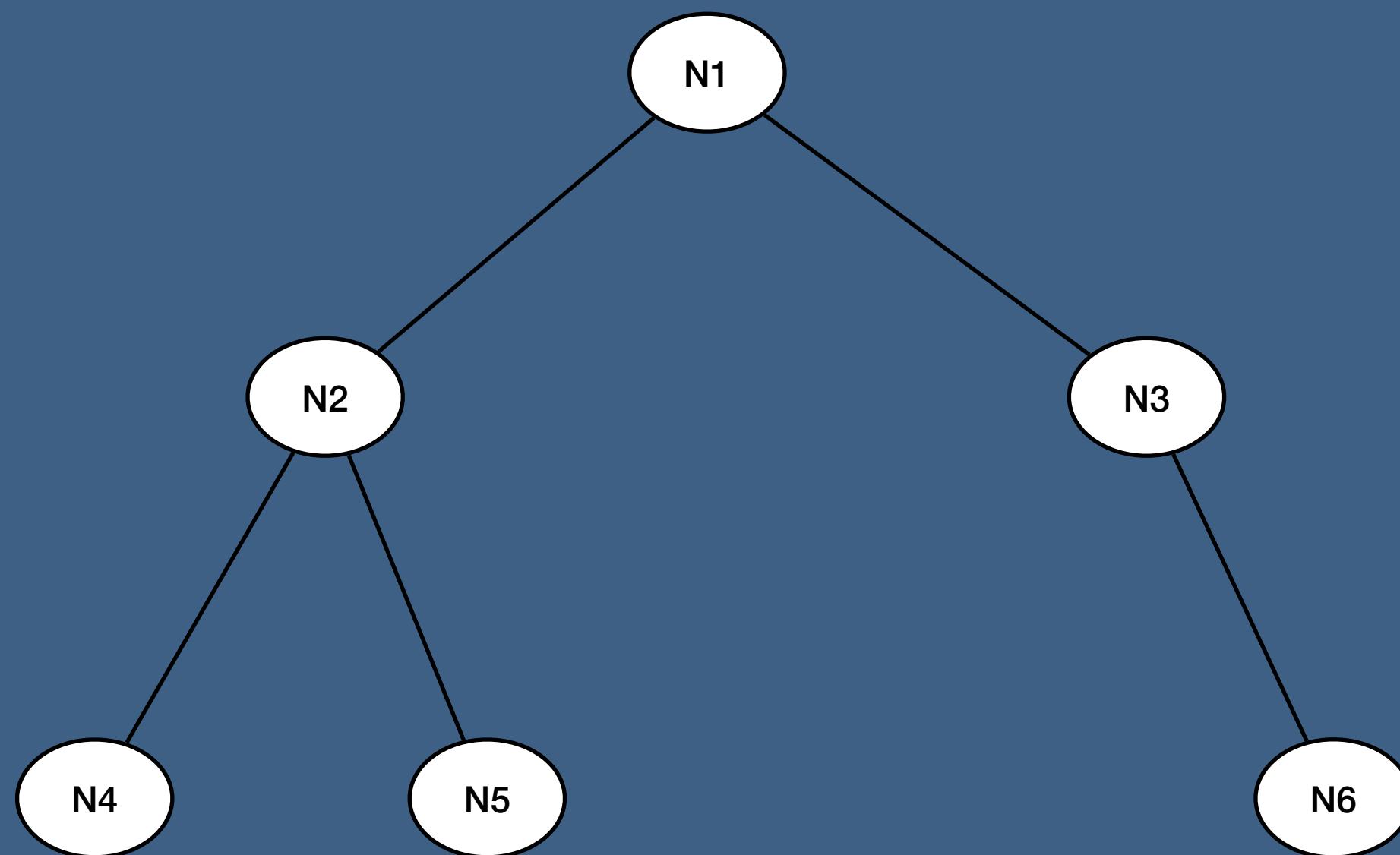


# Check Balanced

## 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

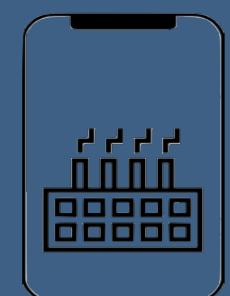
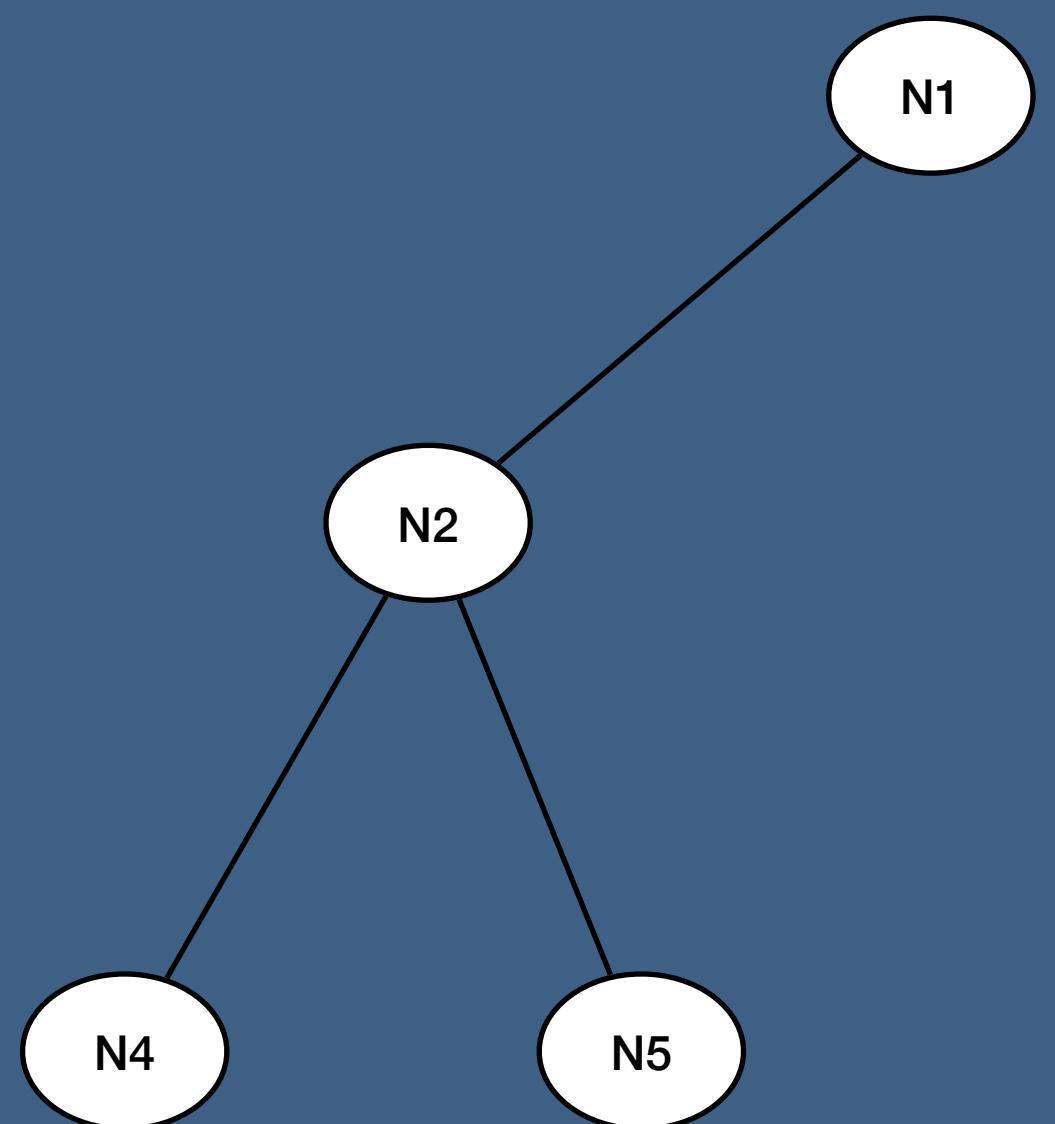


# Check Balanced

## 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



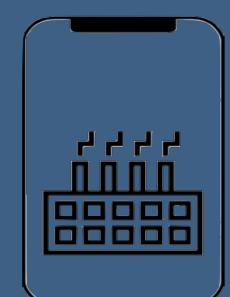
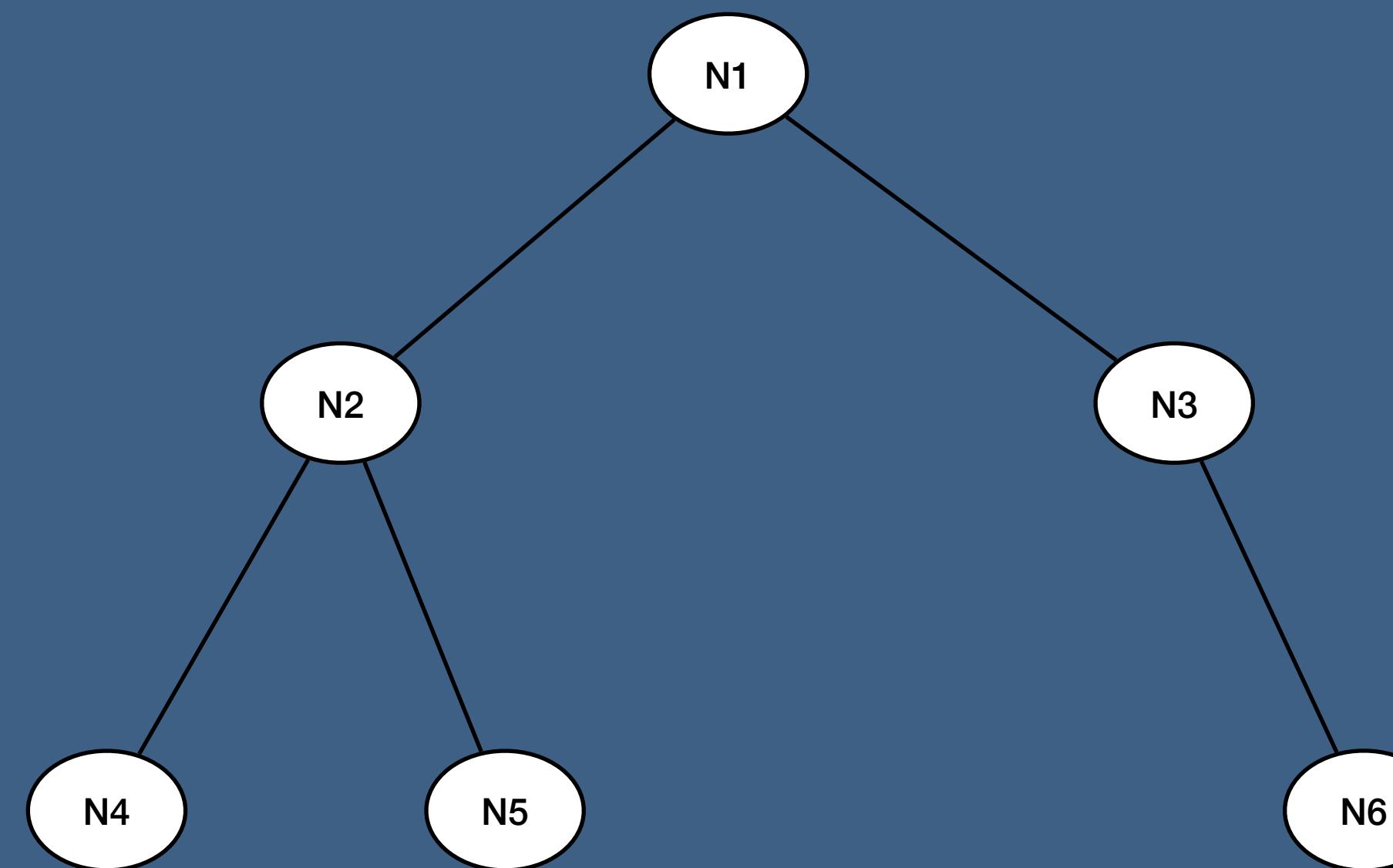
# Check Balanced

## 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



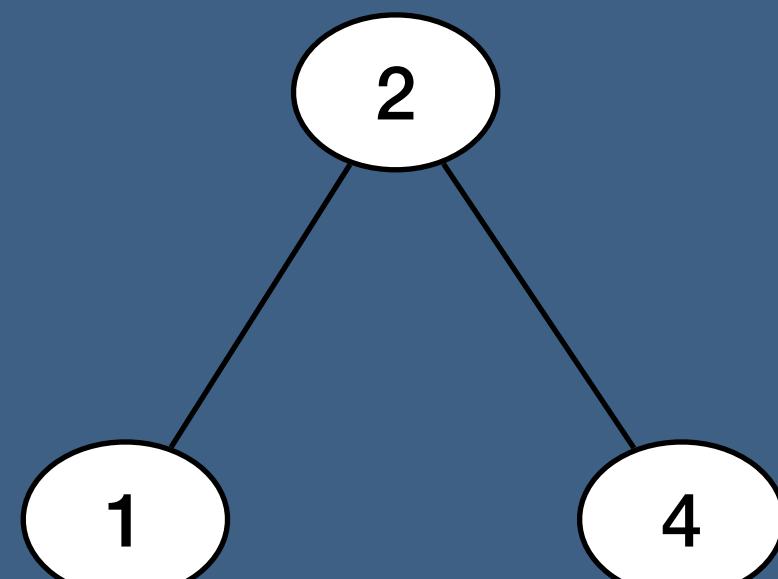
# Validate BST

## Problem Statement:

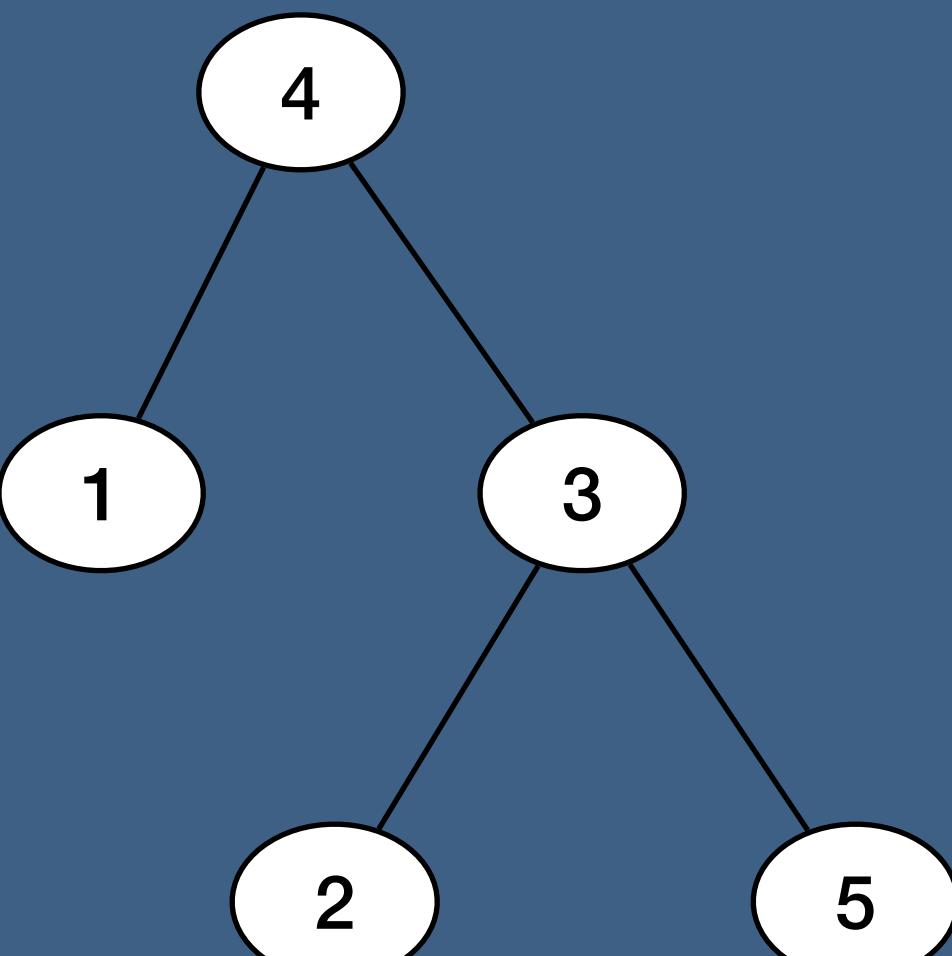
Implement a function to check if a binary tree is a Binary Search Tree.

The Binary Tree is Binary Search Tree if:

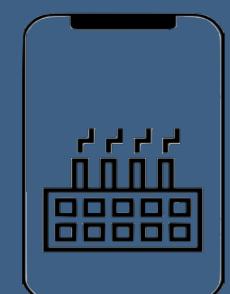
- The left subtree of a node contains only nodes with keys less than the node's key
- The right subtree of a node contains only nodes with keys greater than the node's key
- These conditions are applicable for both left and right subtrees



True



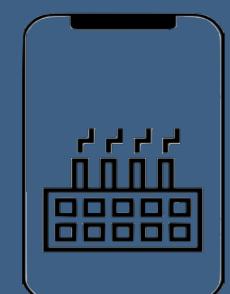
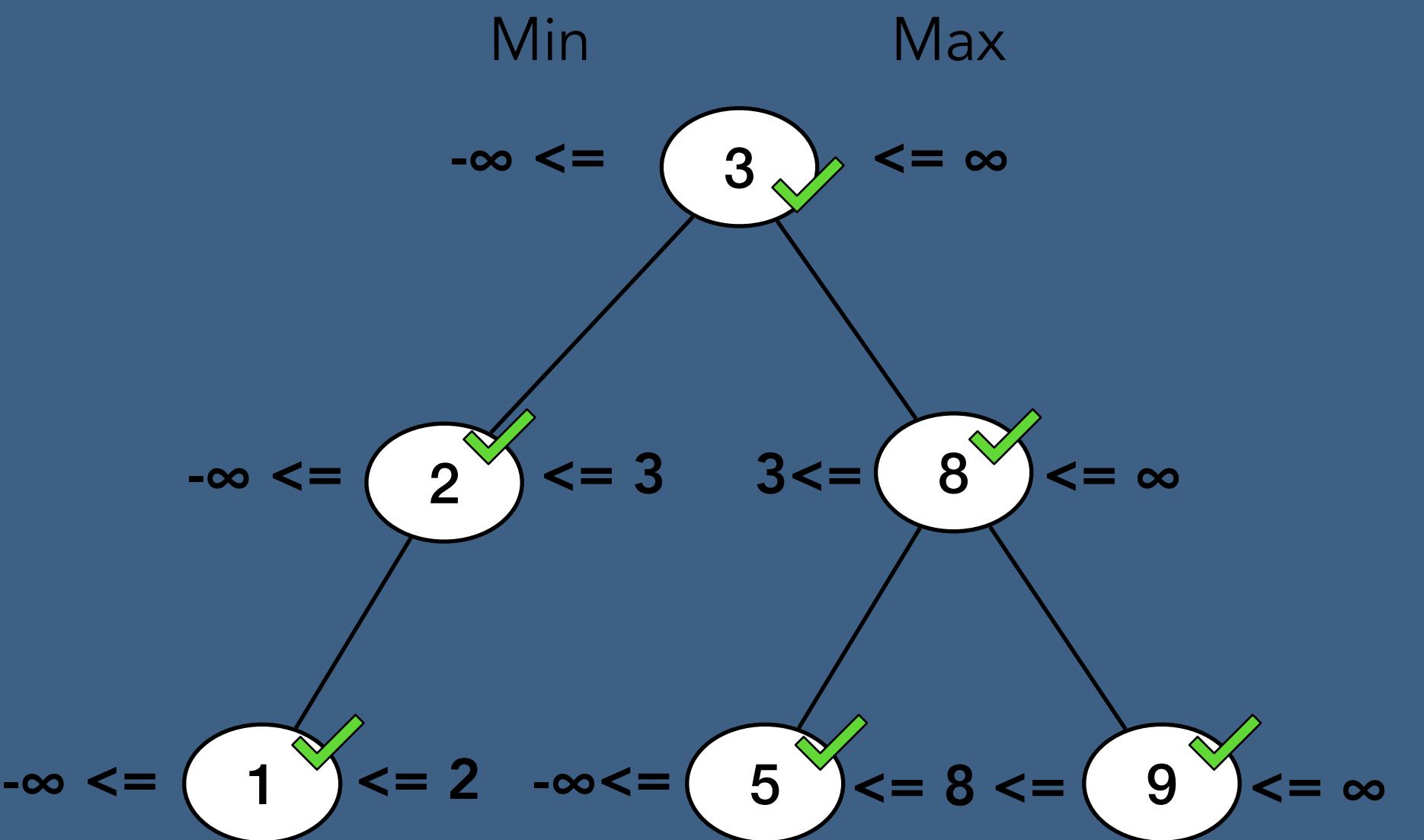
False



# Validate BST

## Problem Statement:

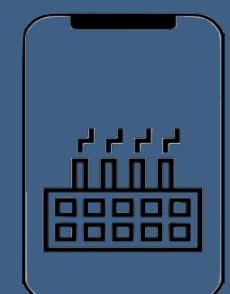
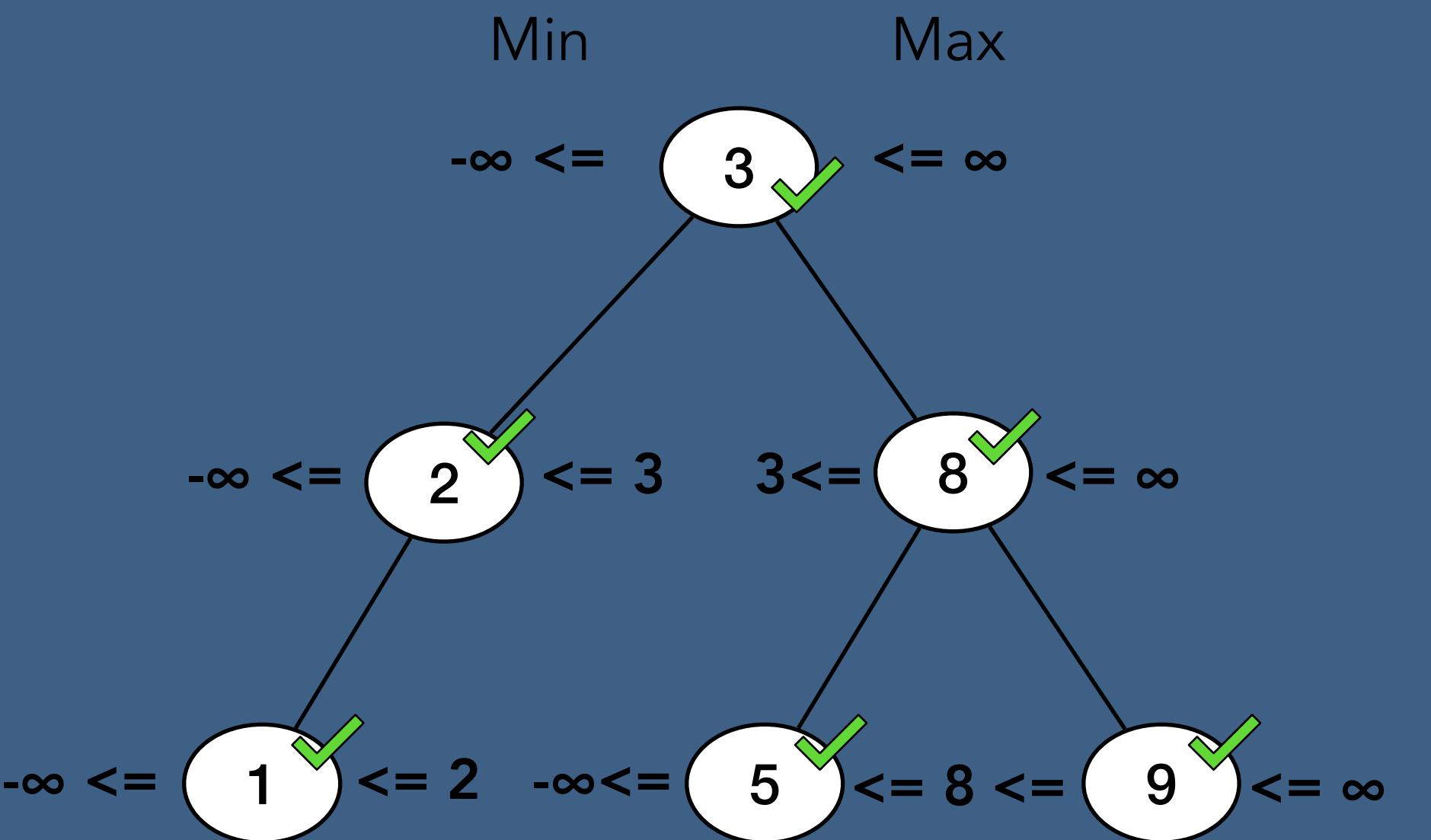
Implement a function to check if a binary tree is a Binary Search Tree.



# Validate BST

## Problem Statement:

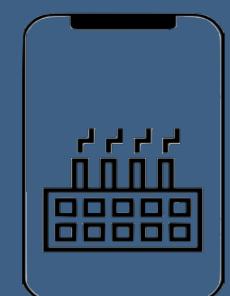
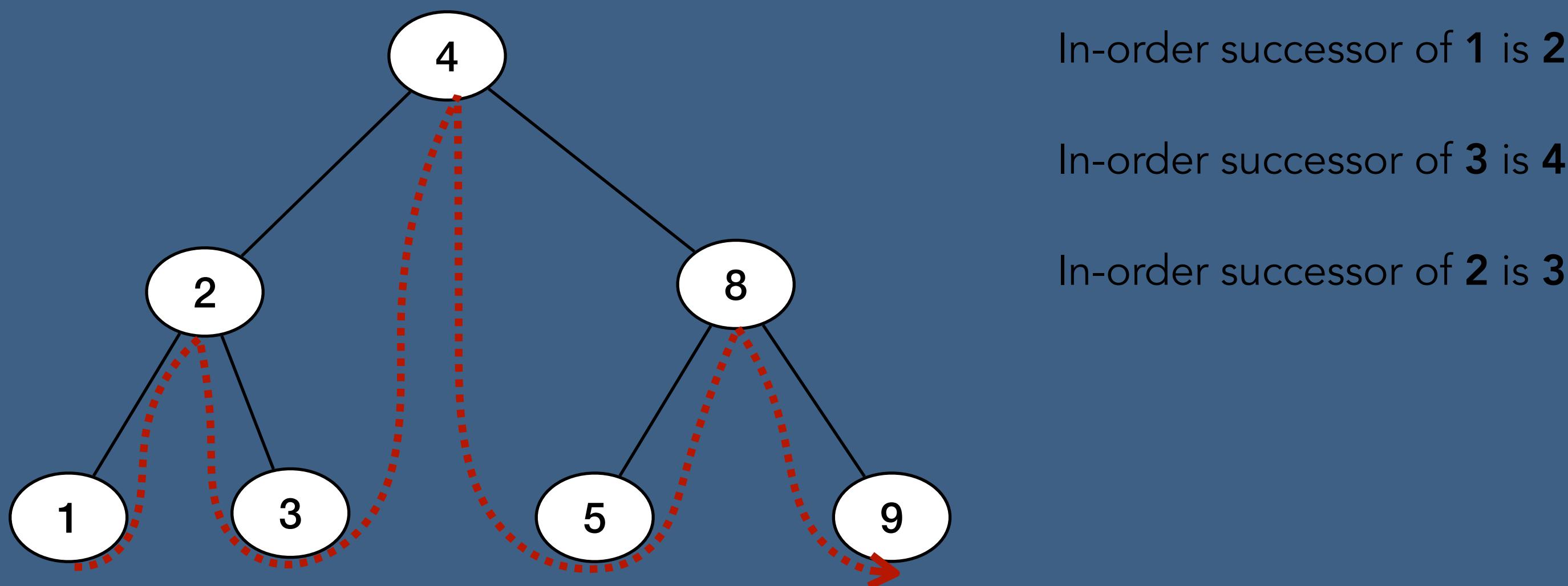
Implement a function to check if a binary tree is a Binary Search Tree.



# Successor

## Problem Statement:

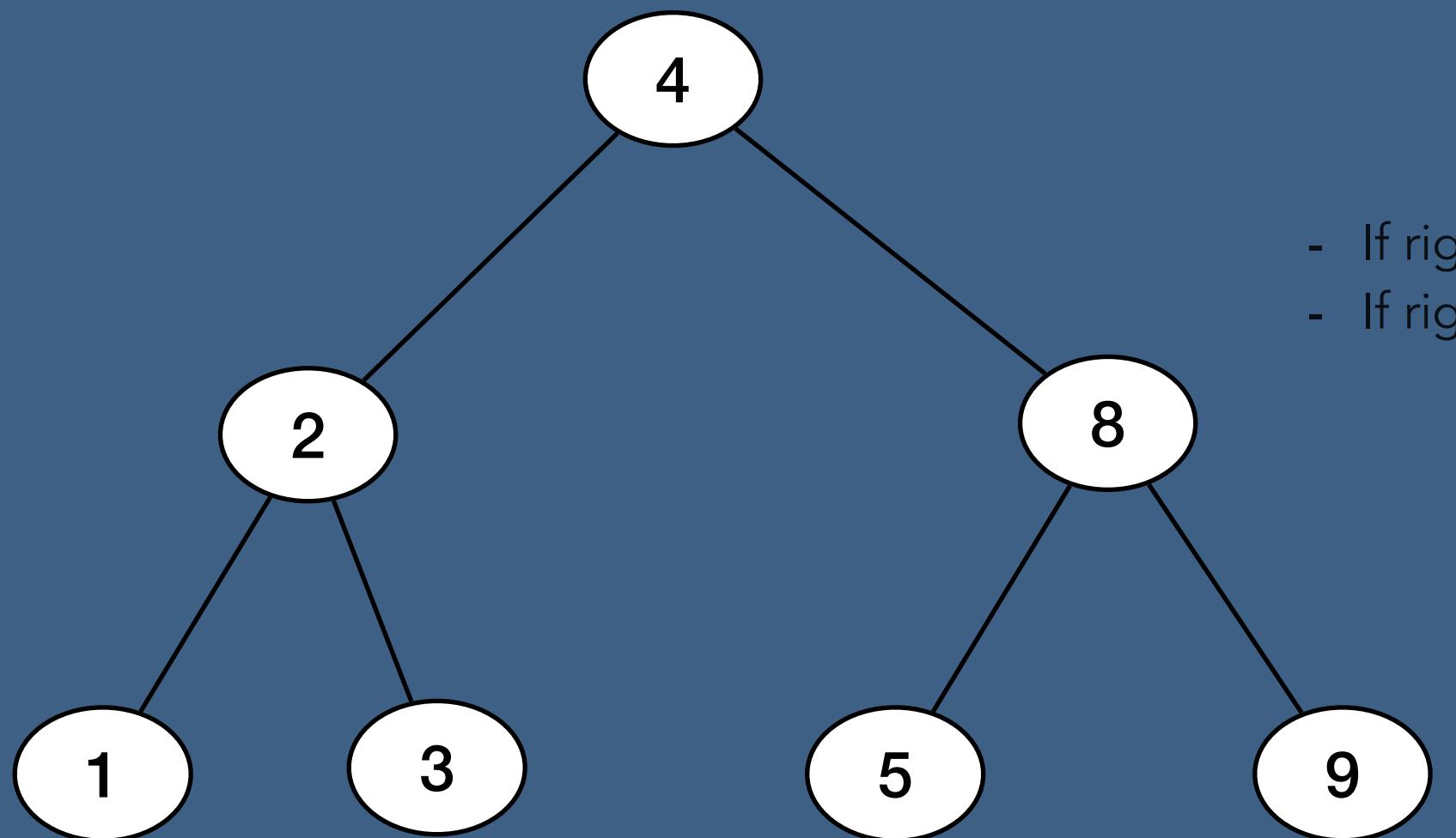
Write an algorithm to find the next node (i.e in-order successor) of given node in a binary search tree. You may assume that each node has a link to its parent.



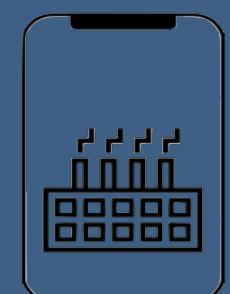
# Successor

## 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.

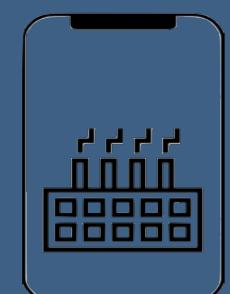
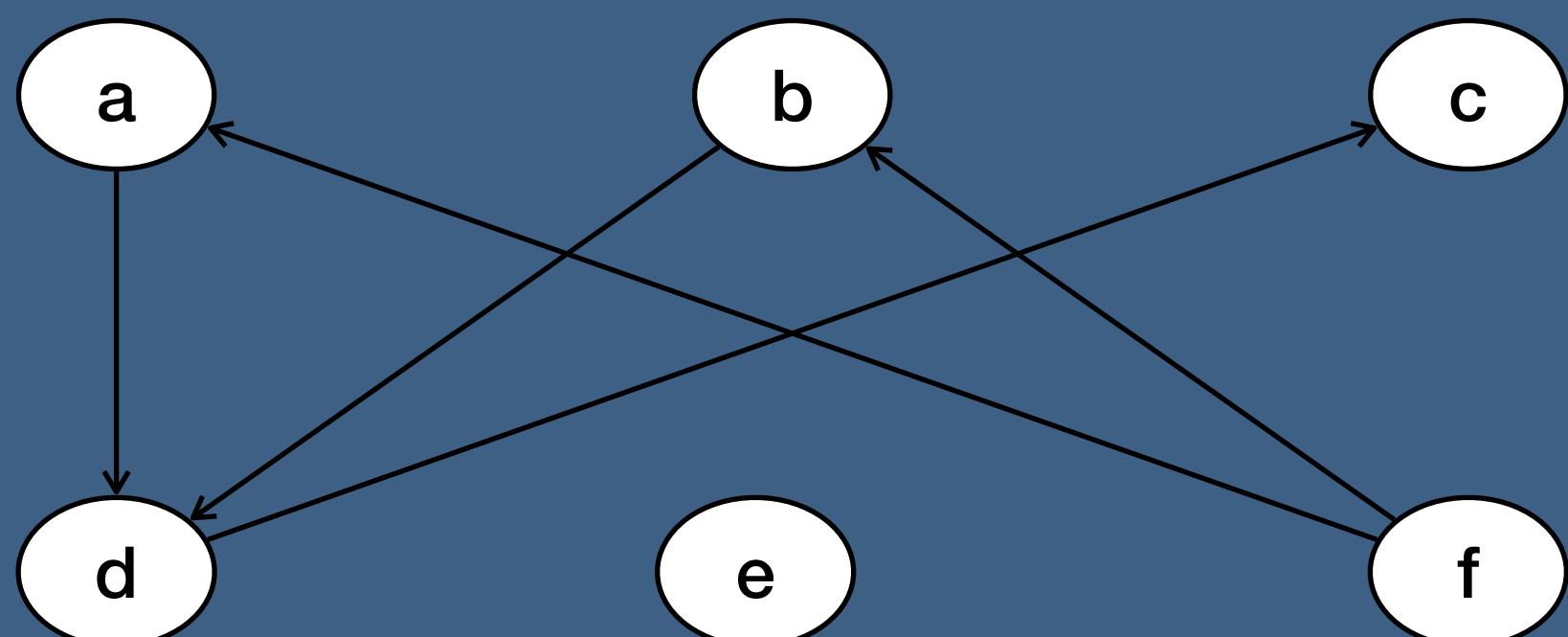
### **Input:**

Projects : a, b, c, d, e, f

Dependencies: (a, d), (f, b), (b, d), (f, a), (d, c)

### **Output:**

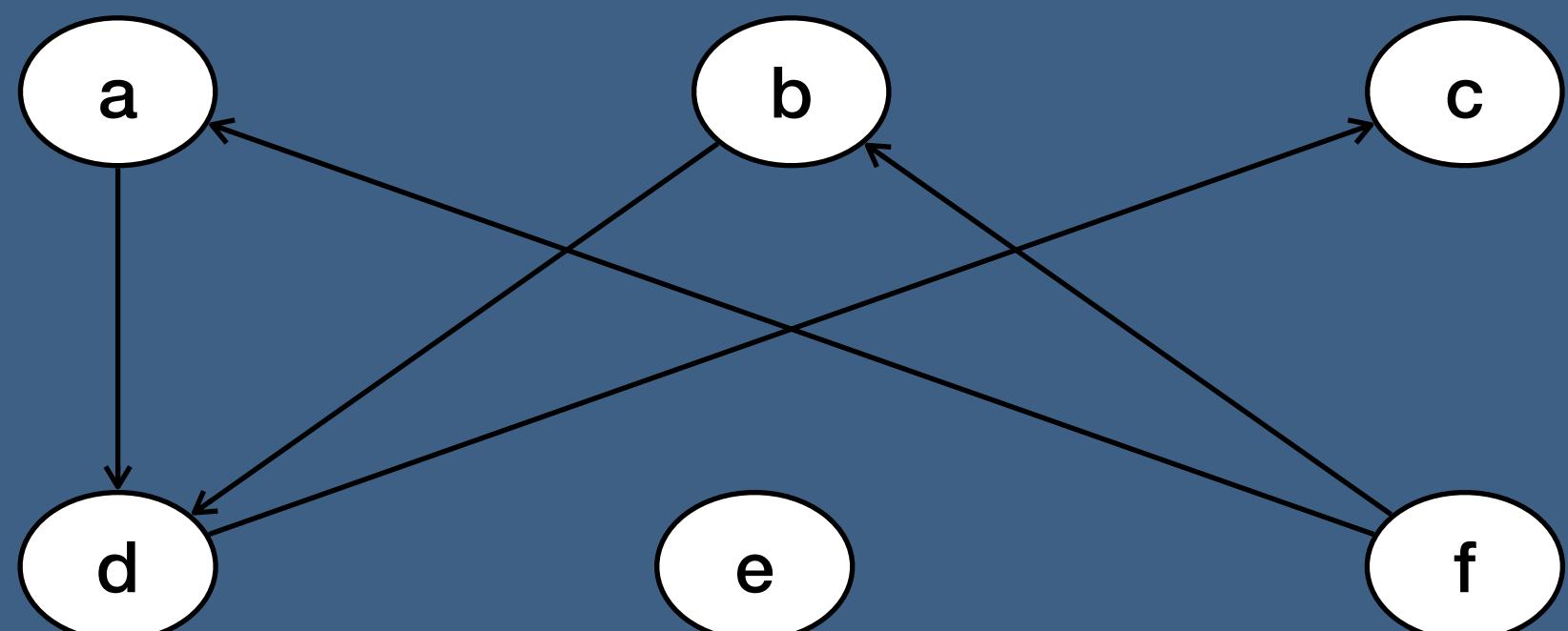
e, f, a, b, d, c



# 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.



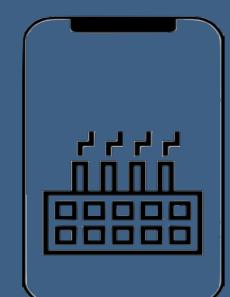
Find nodes with dependencies

a, b, c, d

Find nodes **without** dependencies

e, f

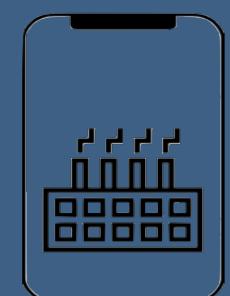
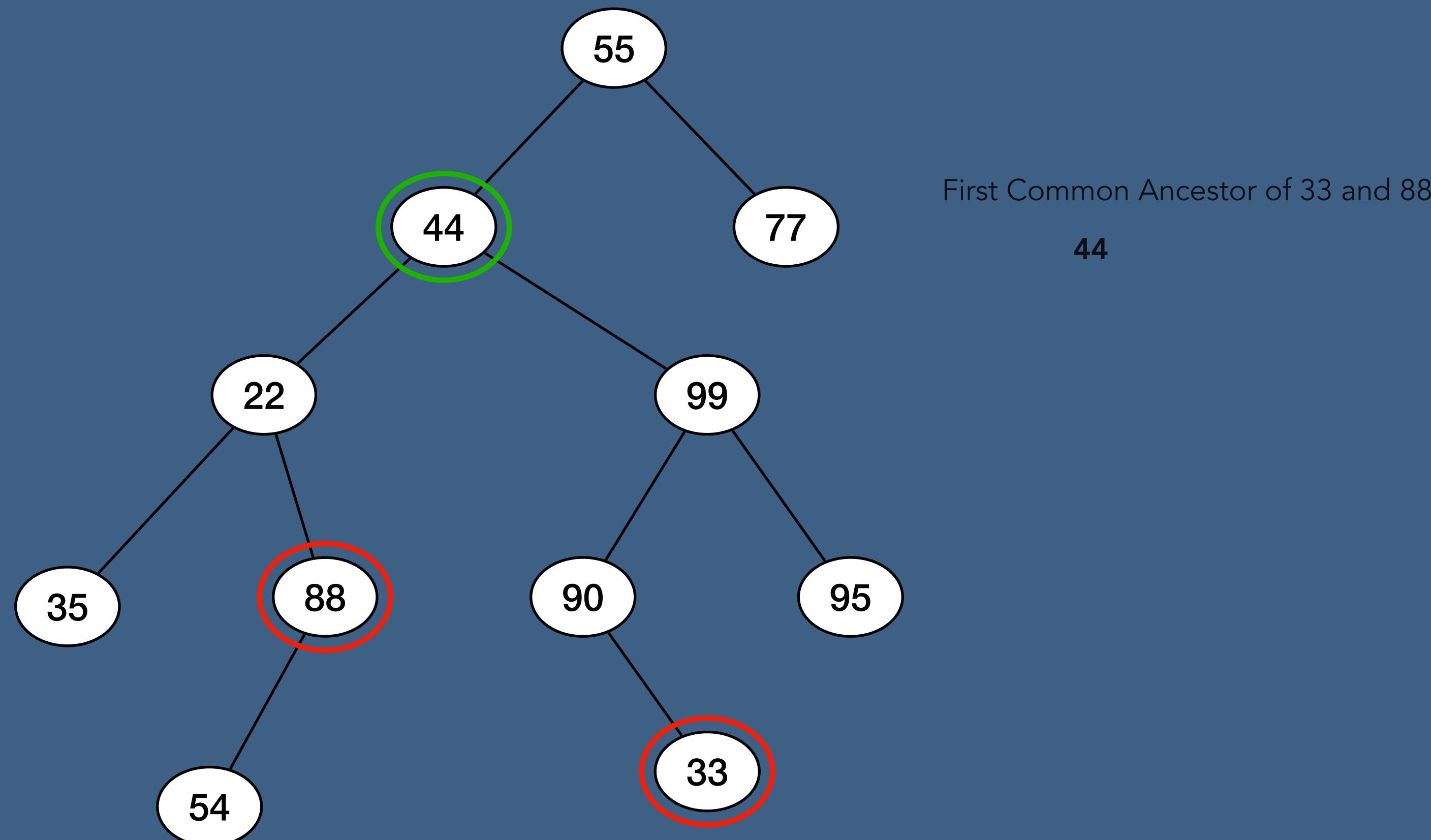
Find order by taking out nodes **without** dependencies.



# First Common Ancestor

## Problem Statement:

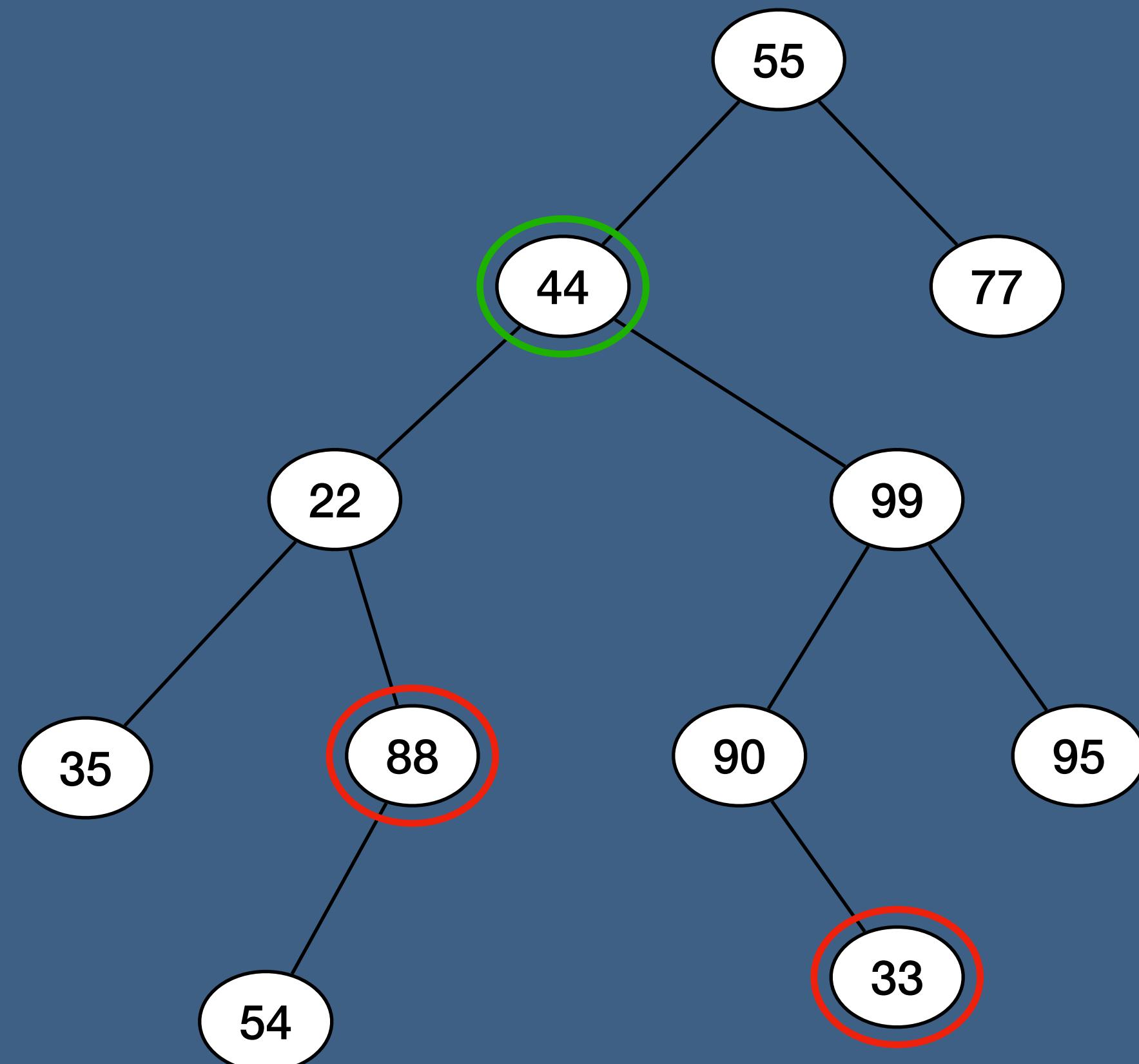
Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.



# 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)

findNodeinTree(33, left-44)

findNodeinTree(88, left-22)

findNodeinTree(33, left-22)

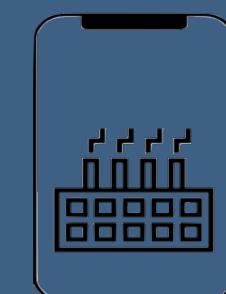
→ True

→ True

→ True

→ False

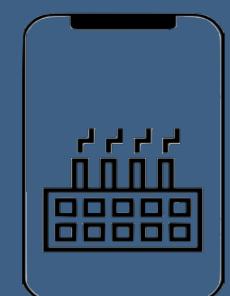
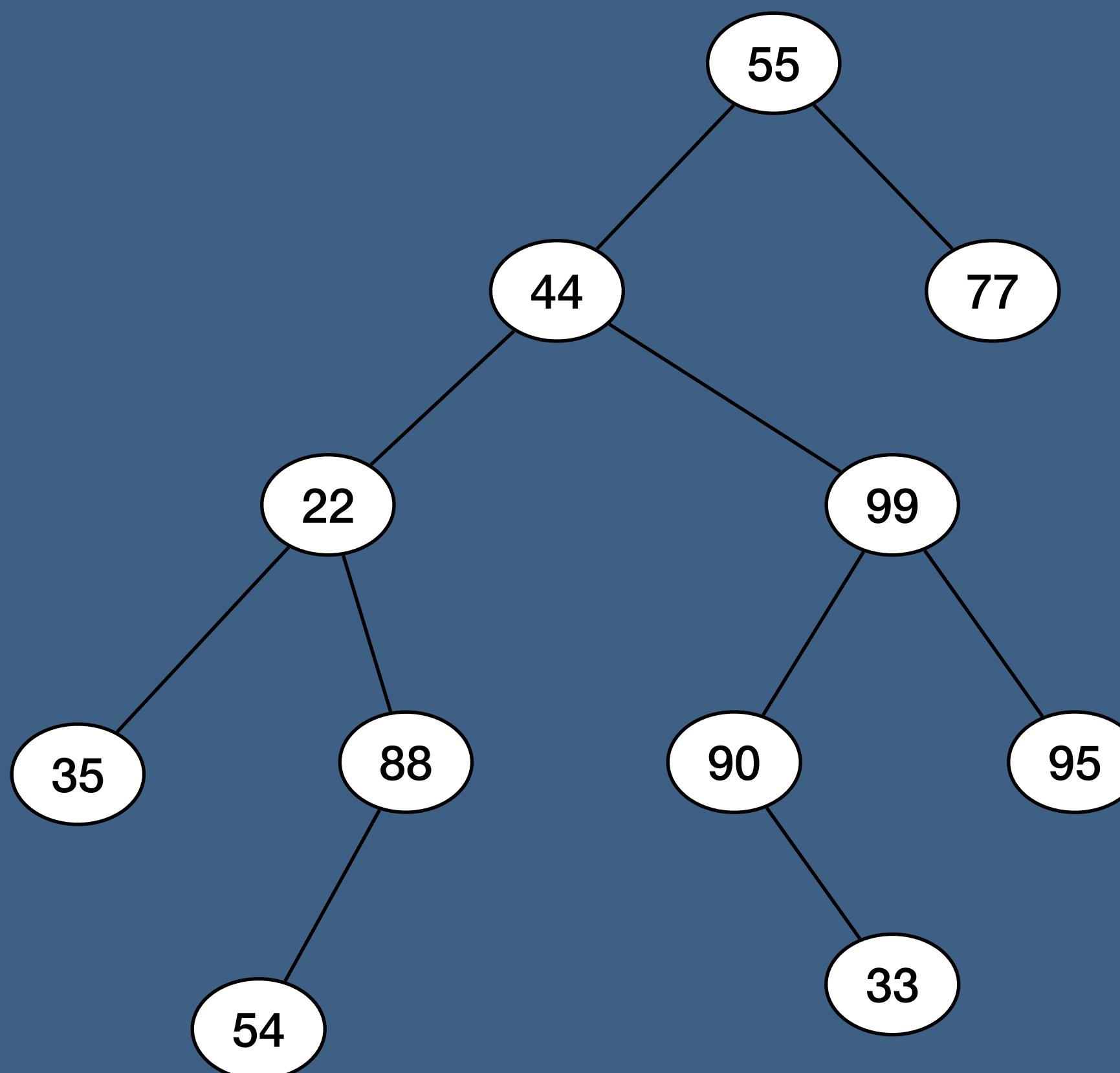
**Return root (44)**



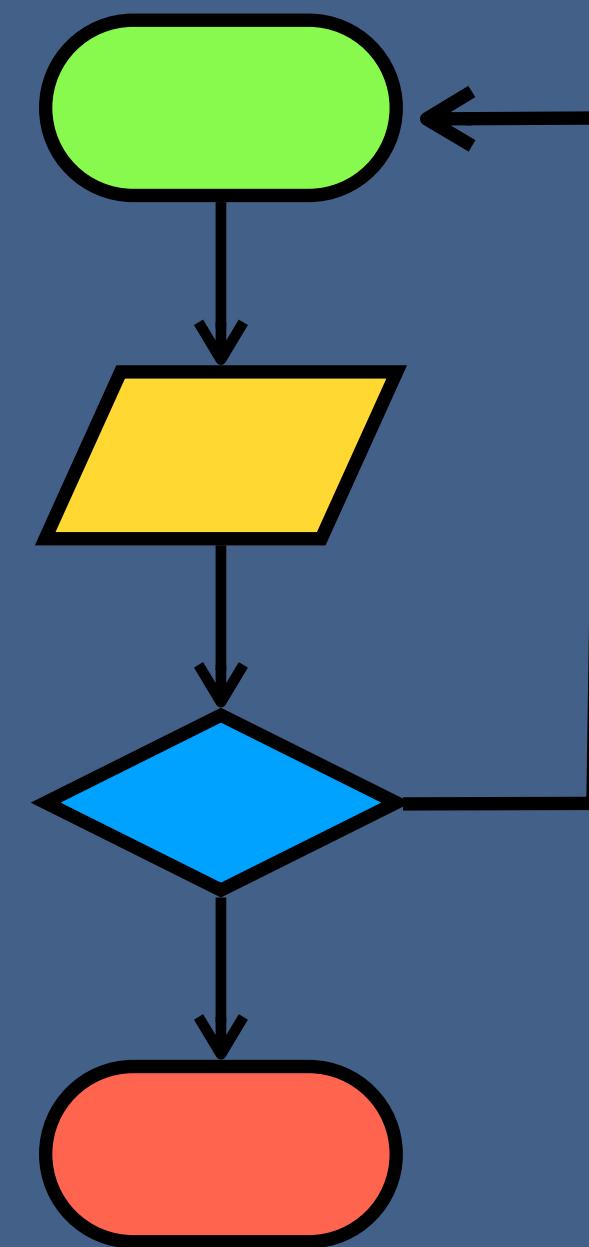
# First Common Ancestor

## 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.

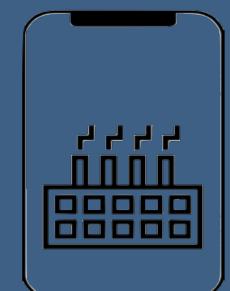
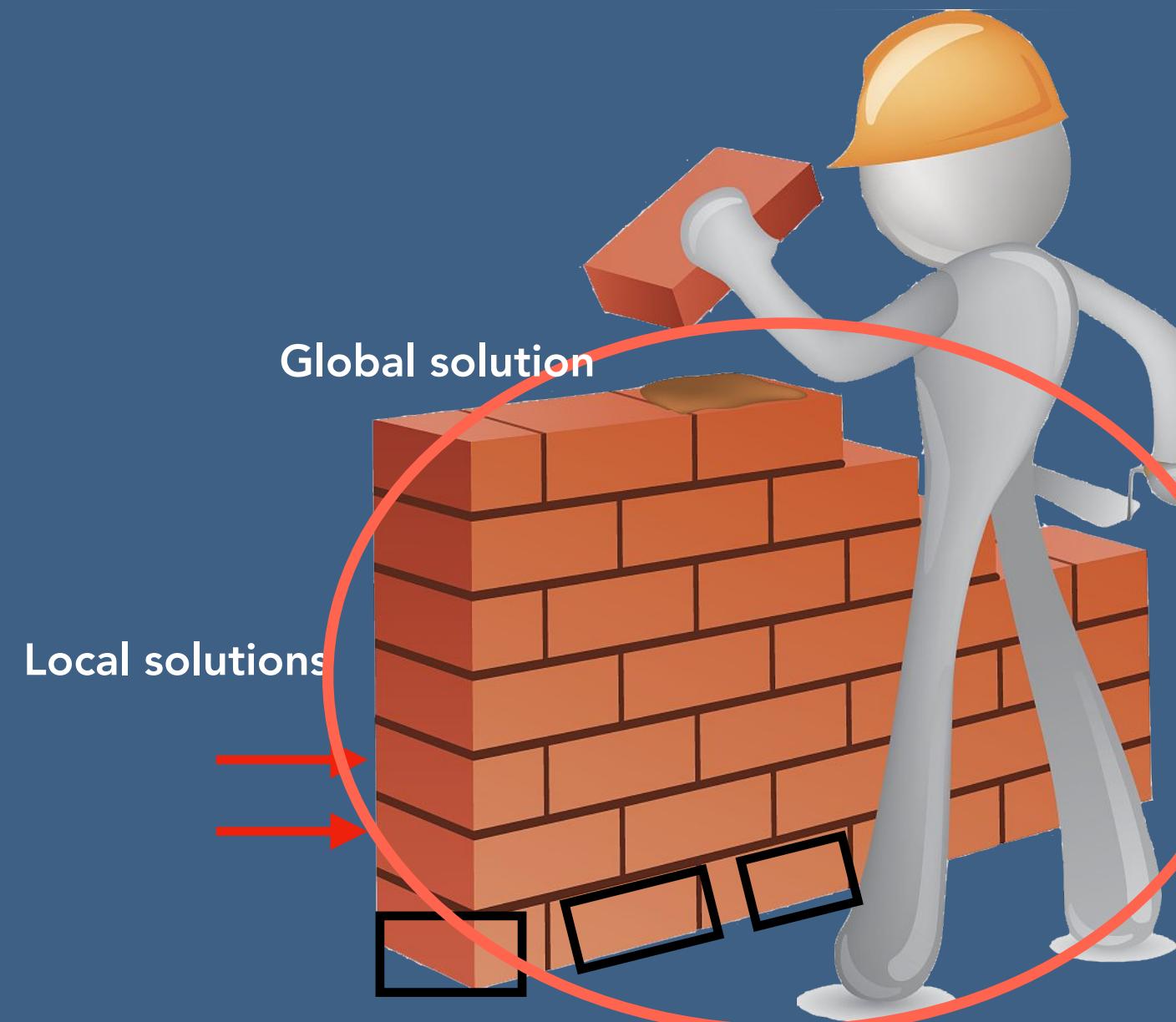


# Greedy Algorithms



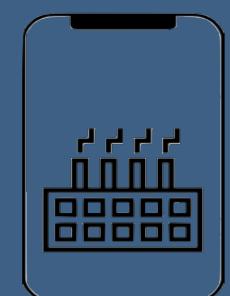
# What is Greedy Algorithm?

- 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



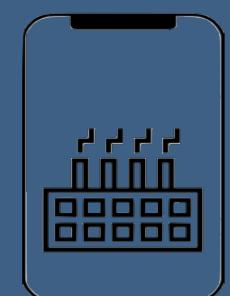
# What is Greedy Algorithm?

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



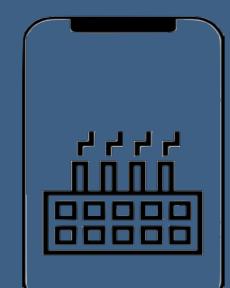
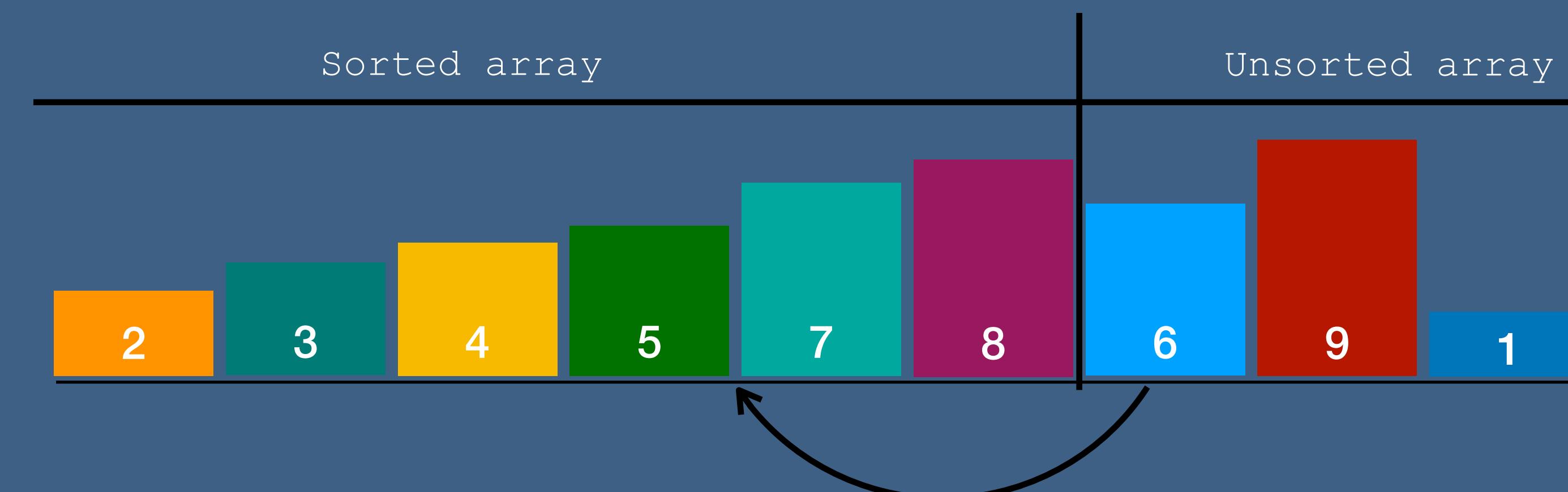
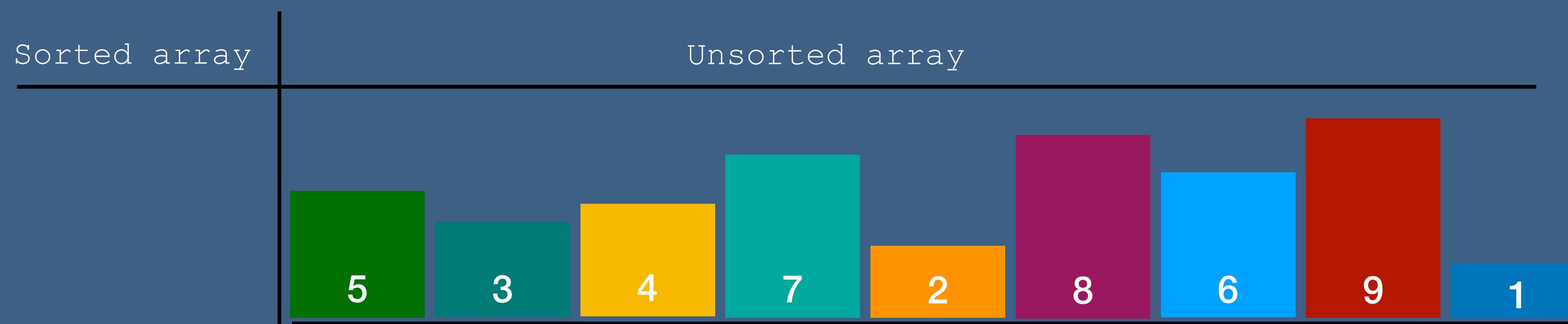
# Greedy Algorithms

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



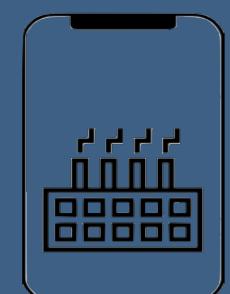
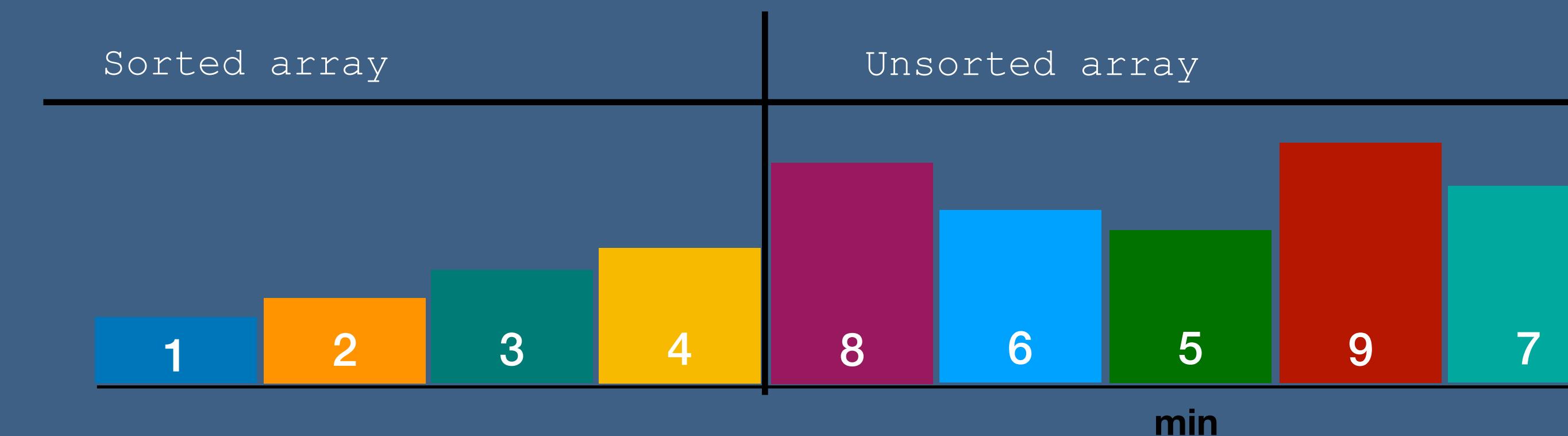
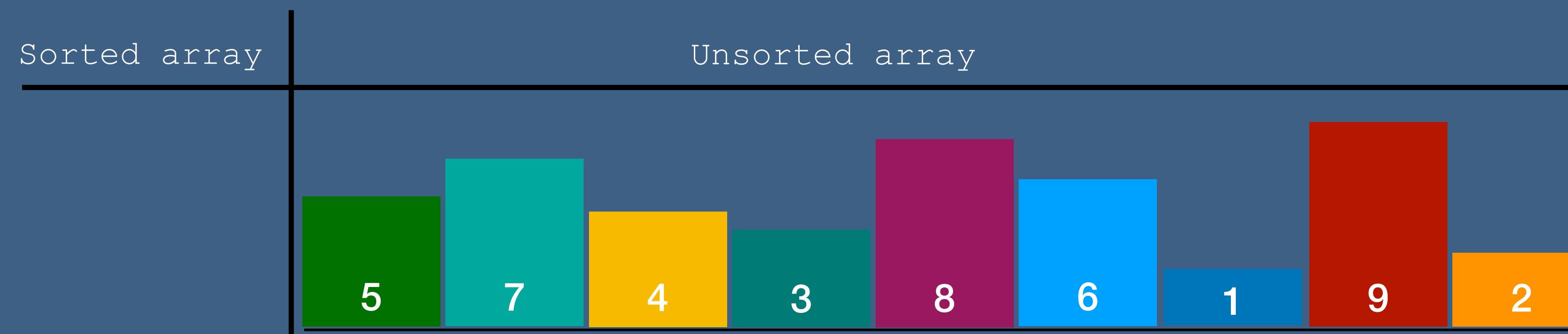
# Greedy Algorithms

## Insertion Sort



# Greedy Algorithms

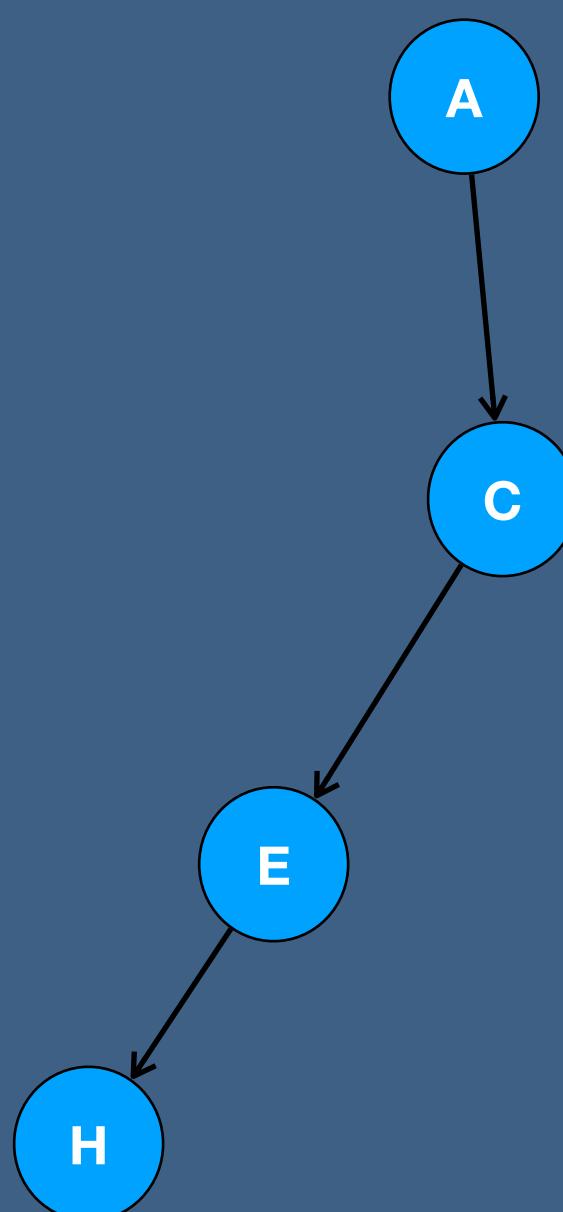
## Selection Sort



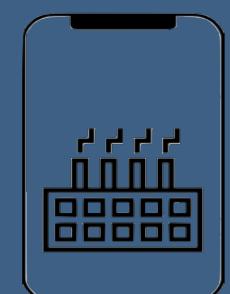
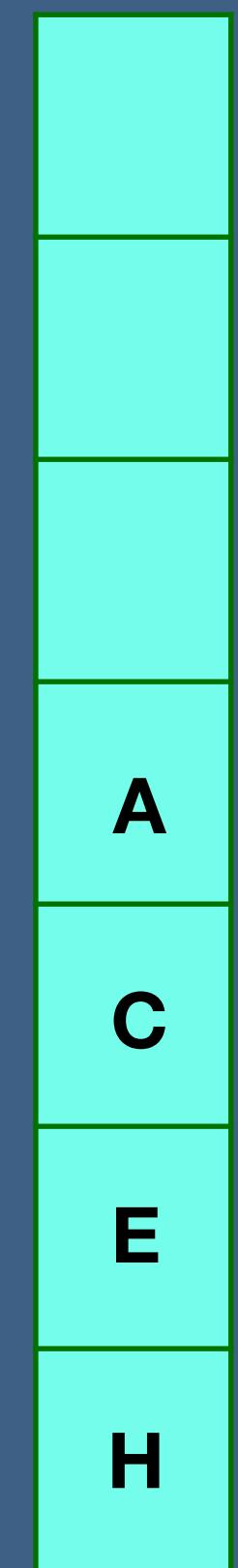
# Greedy Algorithms

## Topological Sort

A C E H



## Stack



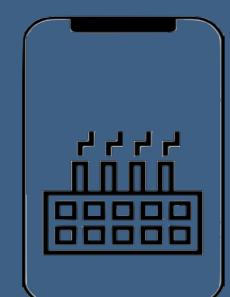
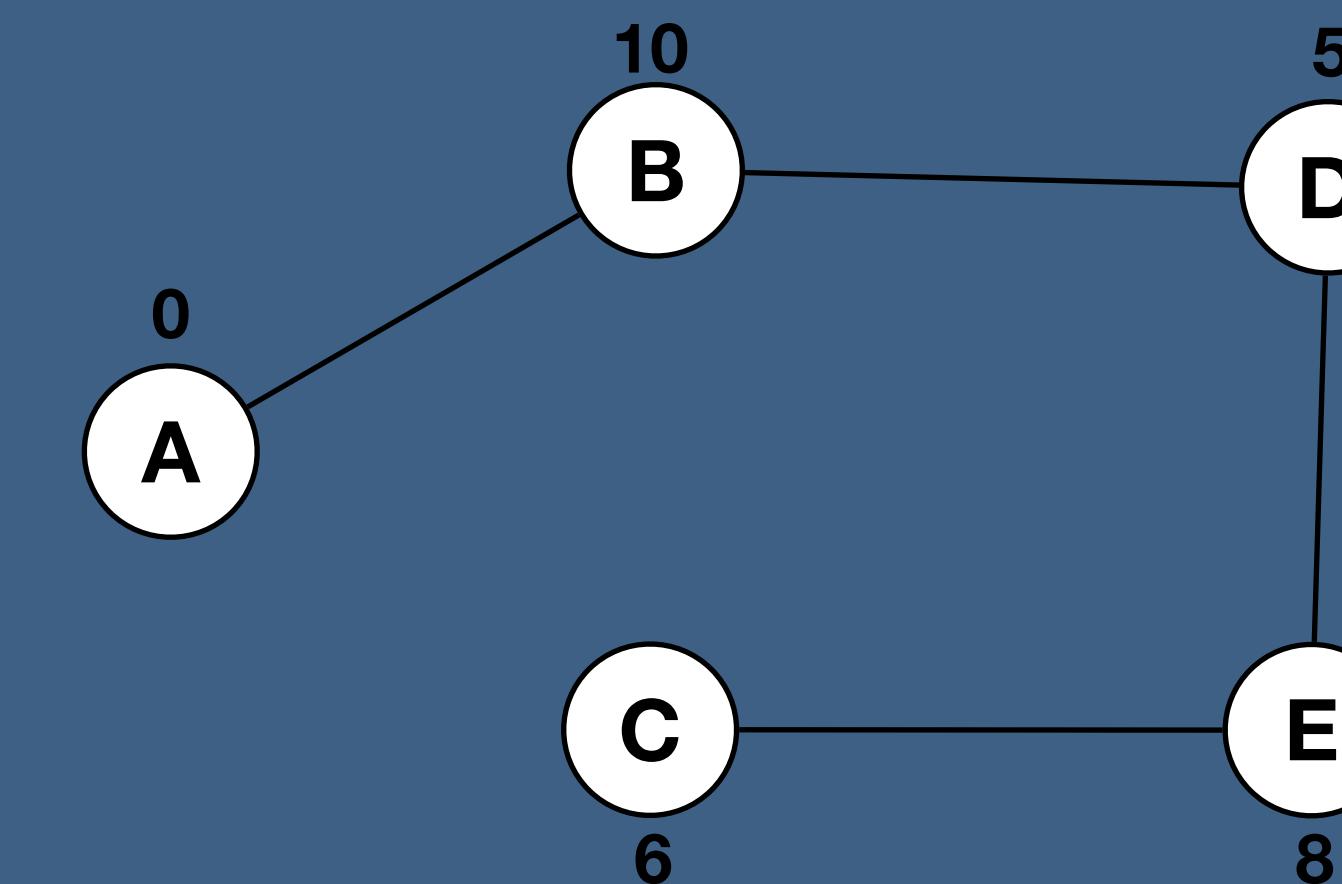
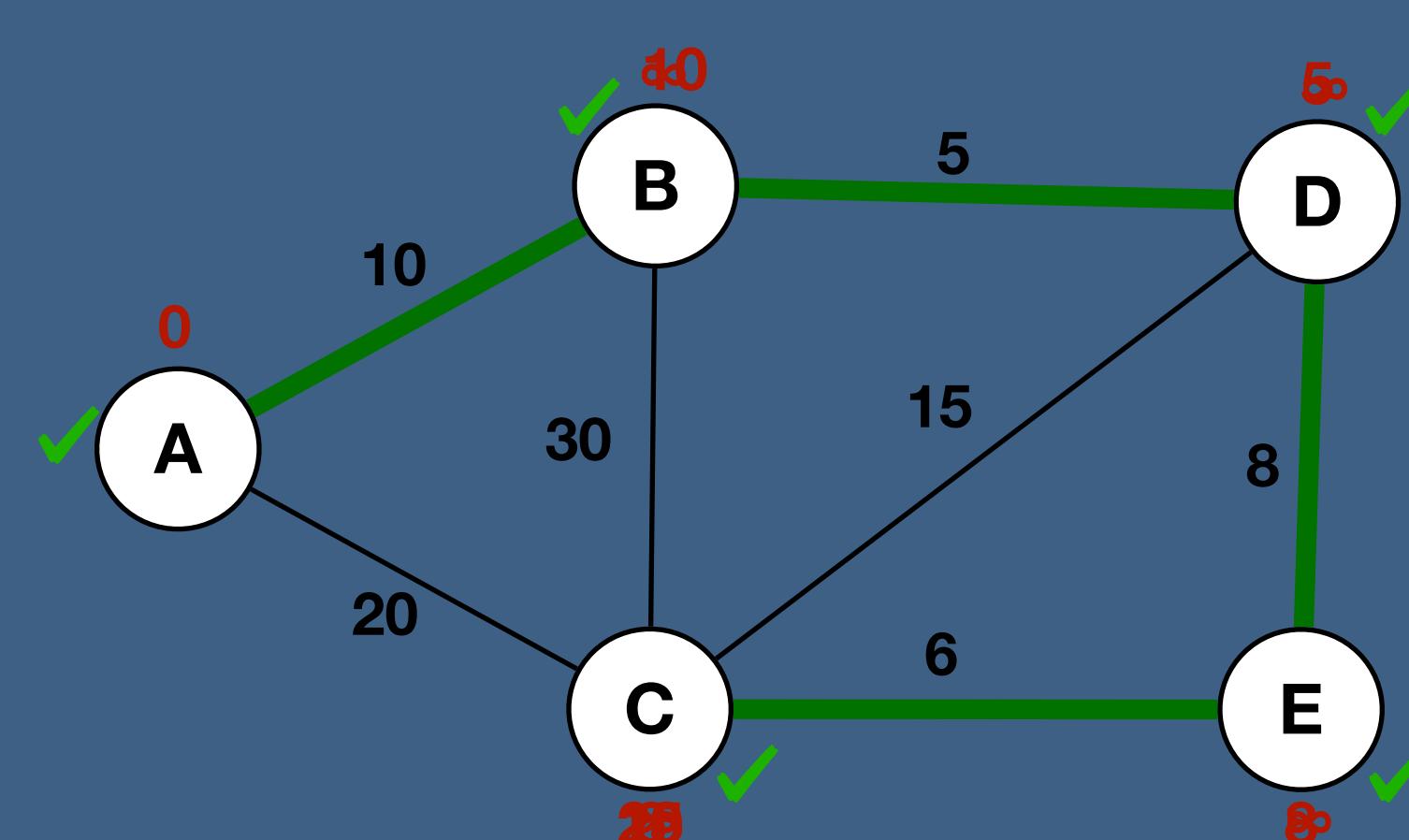
# Greedy Algorithms

## 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



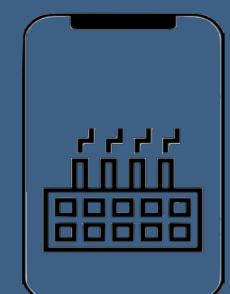
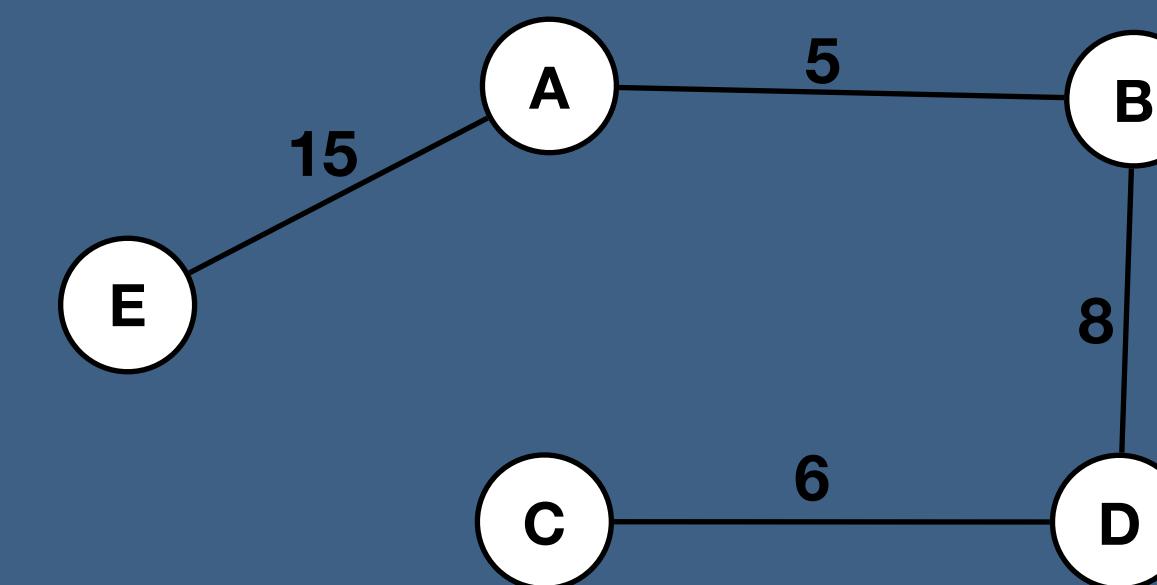
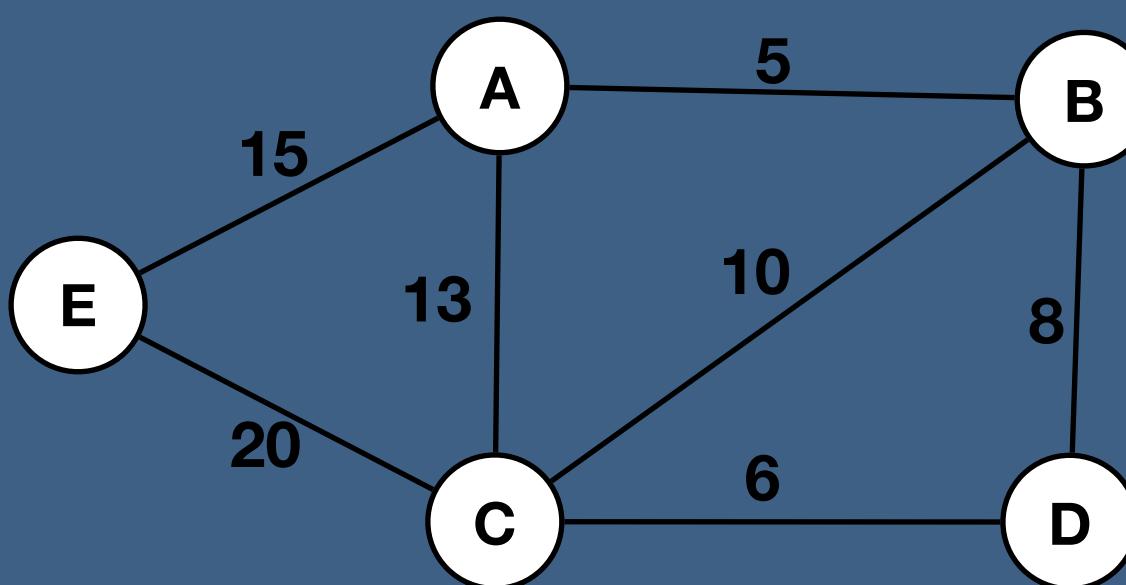
# Greedy Algorithms

## 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

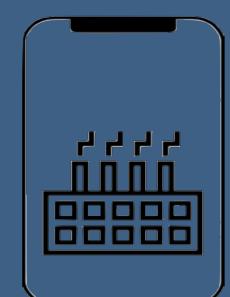
Given N number of activities with their start and end times. We need to select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

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



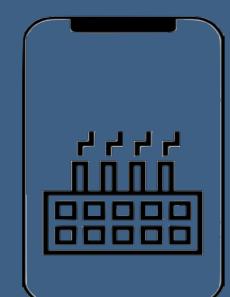
# Activity selection problem

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 yo make up the given amount.

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

## Example 1

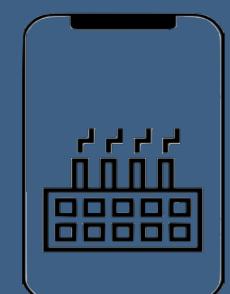
Total amount : 70

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 yo 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

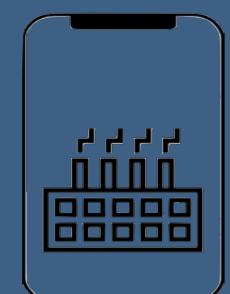
$$1035 - 1000 = 35$$

$$35 - 20 = 15$$

Answer: 5

$$15 - 10 = 5$$

$$5 - 5 = 0$$



# Coin Change Problem

Find the biggest coin that is less than given total number

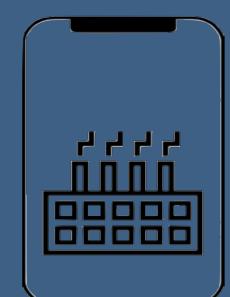
Add coin to the result and subtract coin from total number

If V is equal to zero:

    Then print result

else:

    Repeat Step 2 and 3

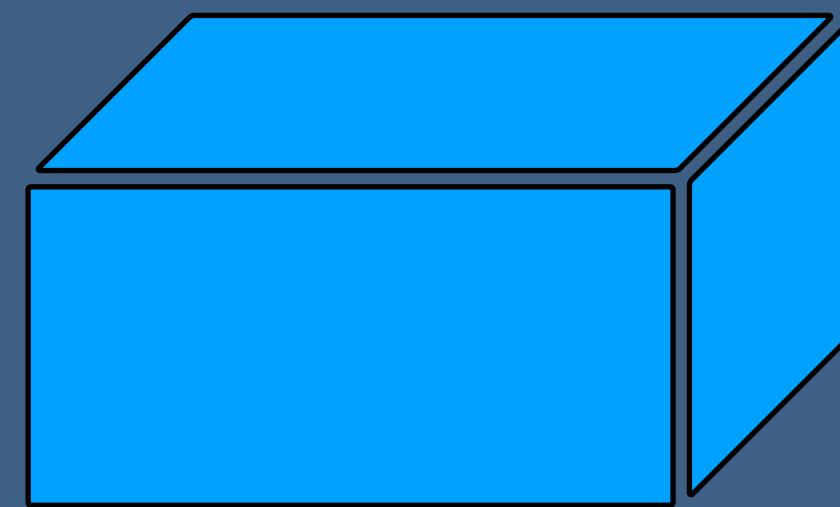


# Fractional Knapsack Problem

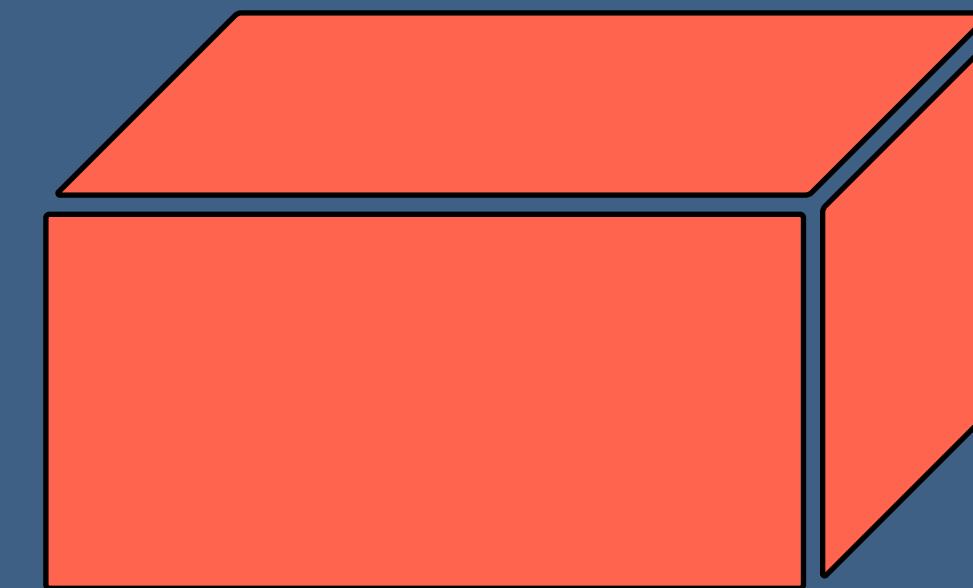
Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



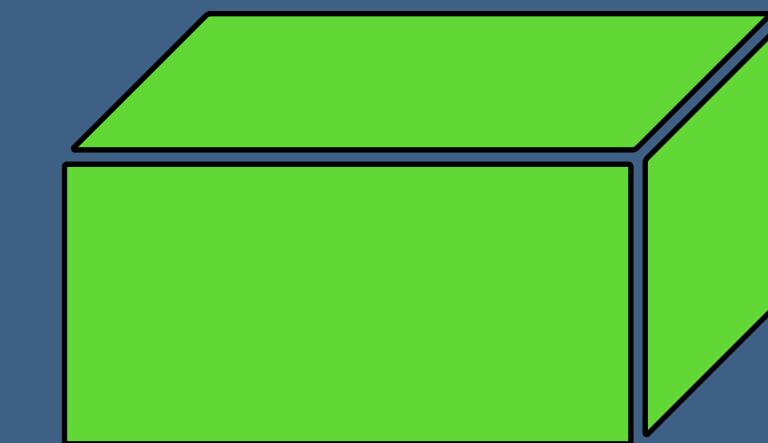
30 kg



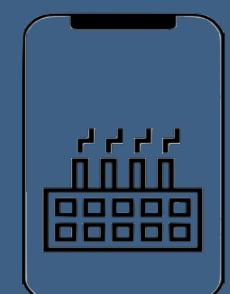
10 kg, Value : 20



20 kg, Value : 10

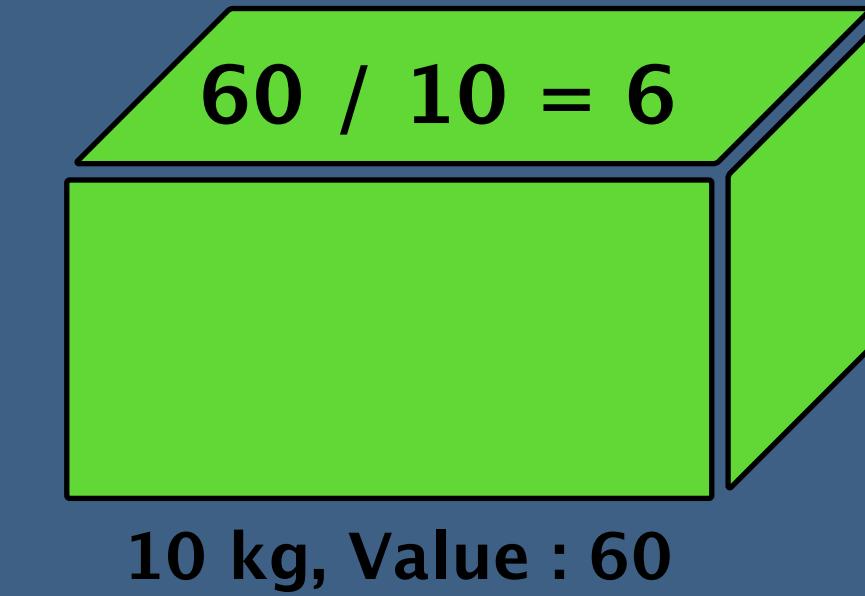
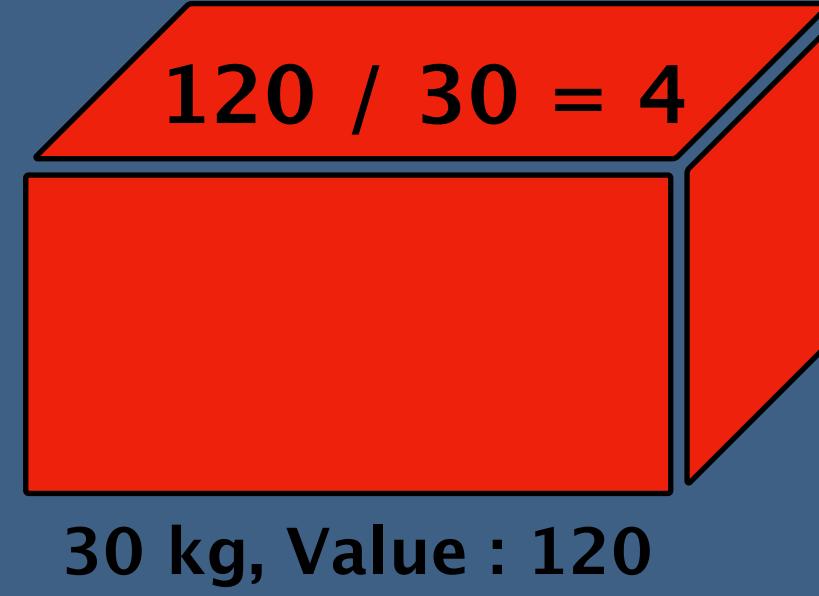
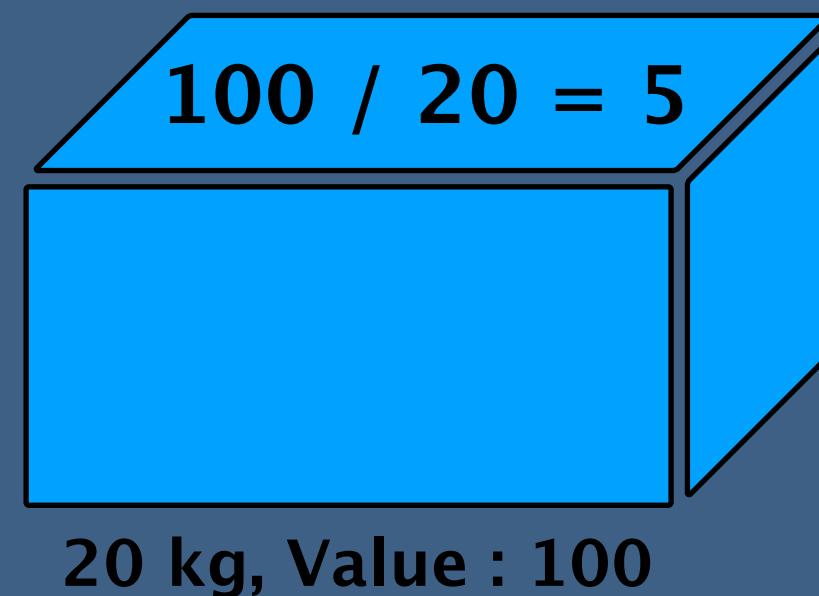


10 kg, Value : 30



# Fractional Knapsack Problem

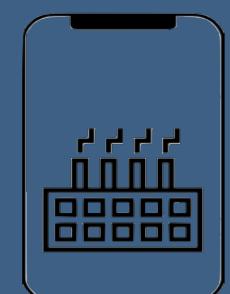
Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



50 kg

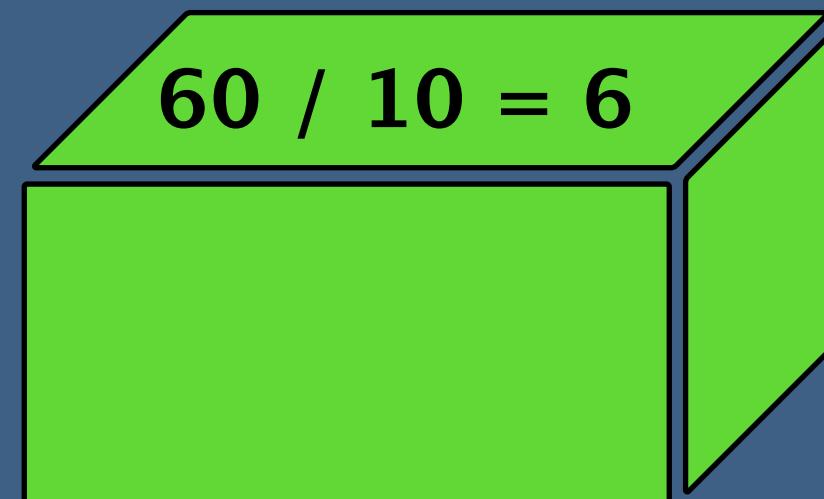
$$100 + 120 = 220$$

$$60 + 100 + 120 \cdot 2/3 = 240$$

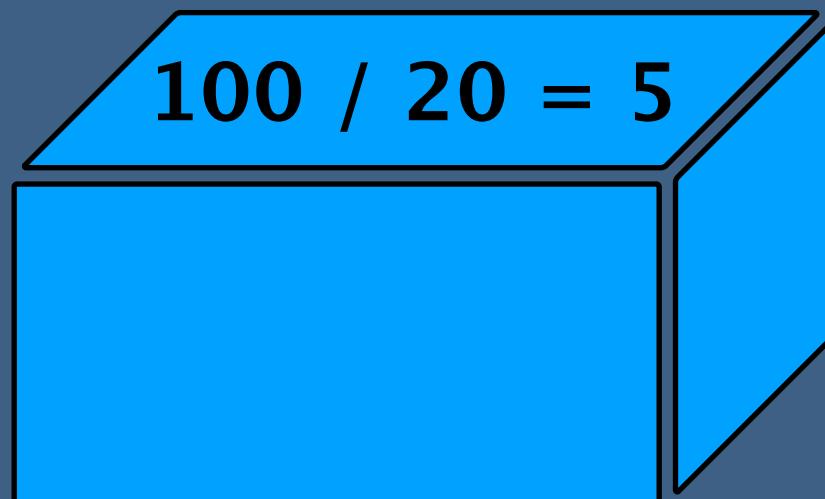


# Fractional Knapsack Problem

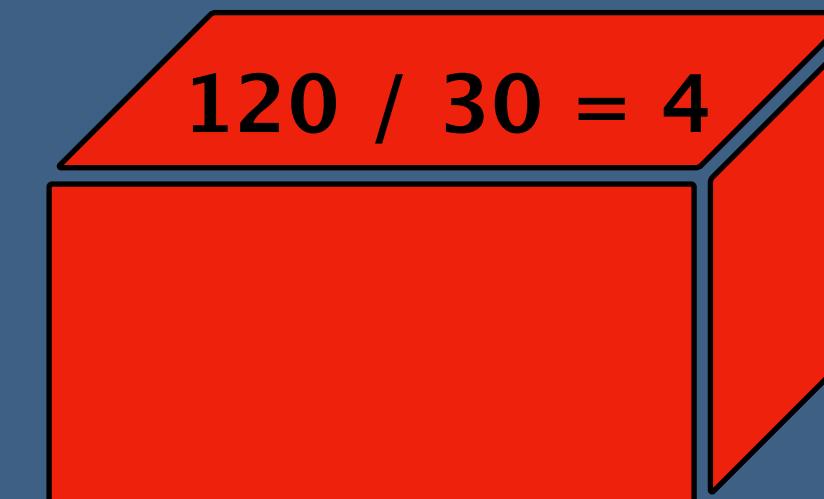
Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.



10 kg, Value : 60



20 kg, Value : 100



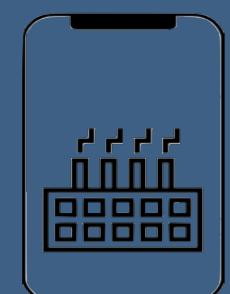
30 kg, Value : 120



50 kg

$$100 + 120 = 220$$

$$60 + 100 + 120 \cdot 2/3 = 240$$



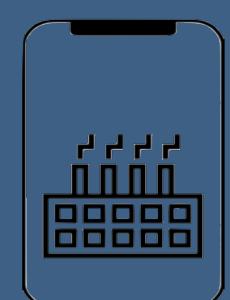
# Fractional Knapsack Problem

Calculate the density or ratio for each item

Sort items based on this ratio

Take items with the highest ratio sequentially until weight allows

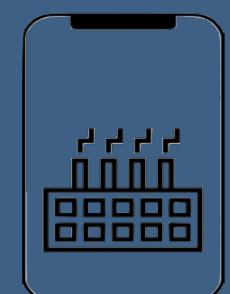
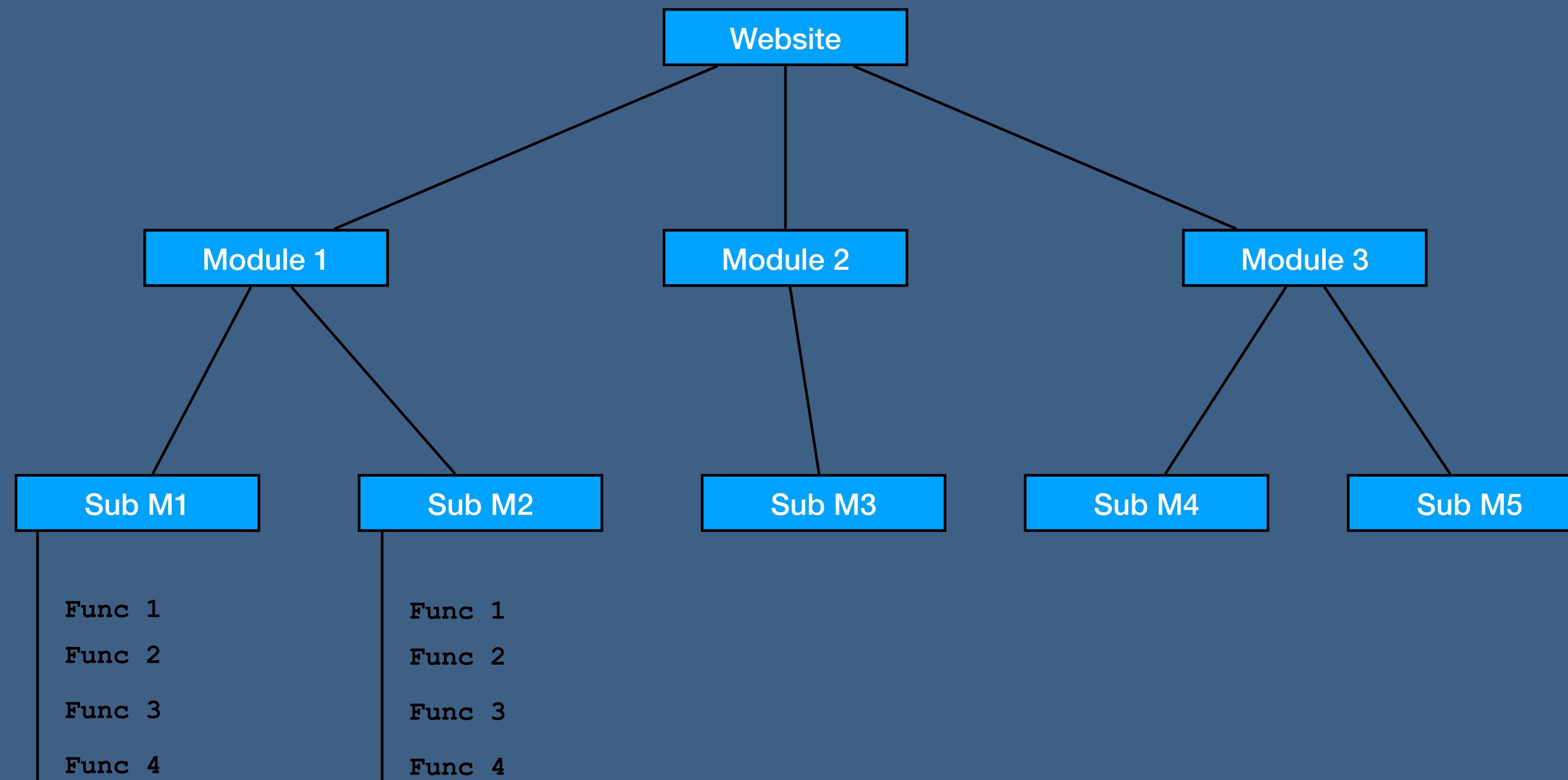
Add the next item as much (fractional) as we can



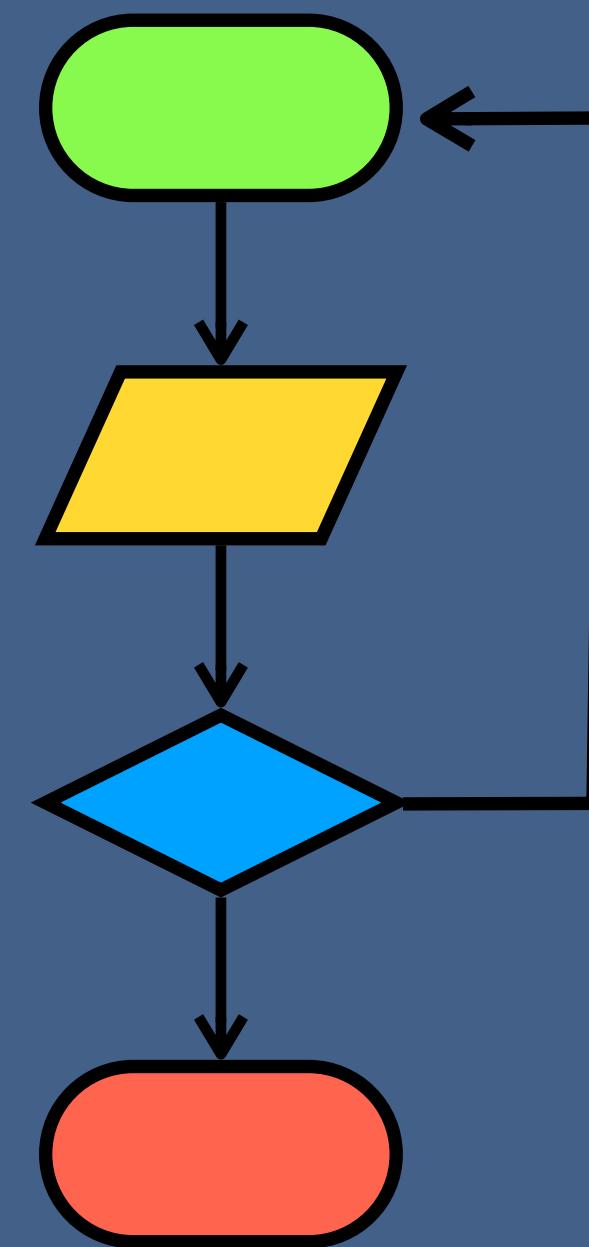
# What is Divide and Conquer Algorithm?

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



# Dynamic Programming



# Dynamic Programming - Number Factor

## **Problem Statement:**

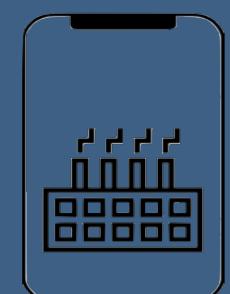
Given N, find the number of ways to express N as a sum of 1, 3 and 4.

### **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}

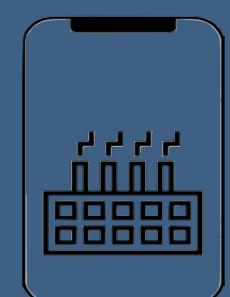


# Dynamic Programming - Number Factor

## Problem Statement:

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

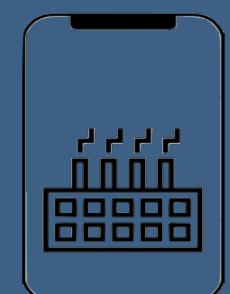
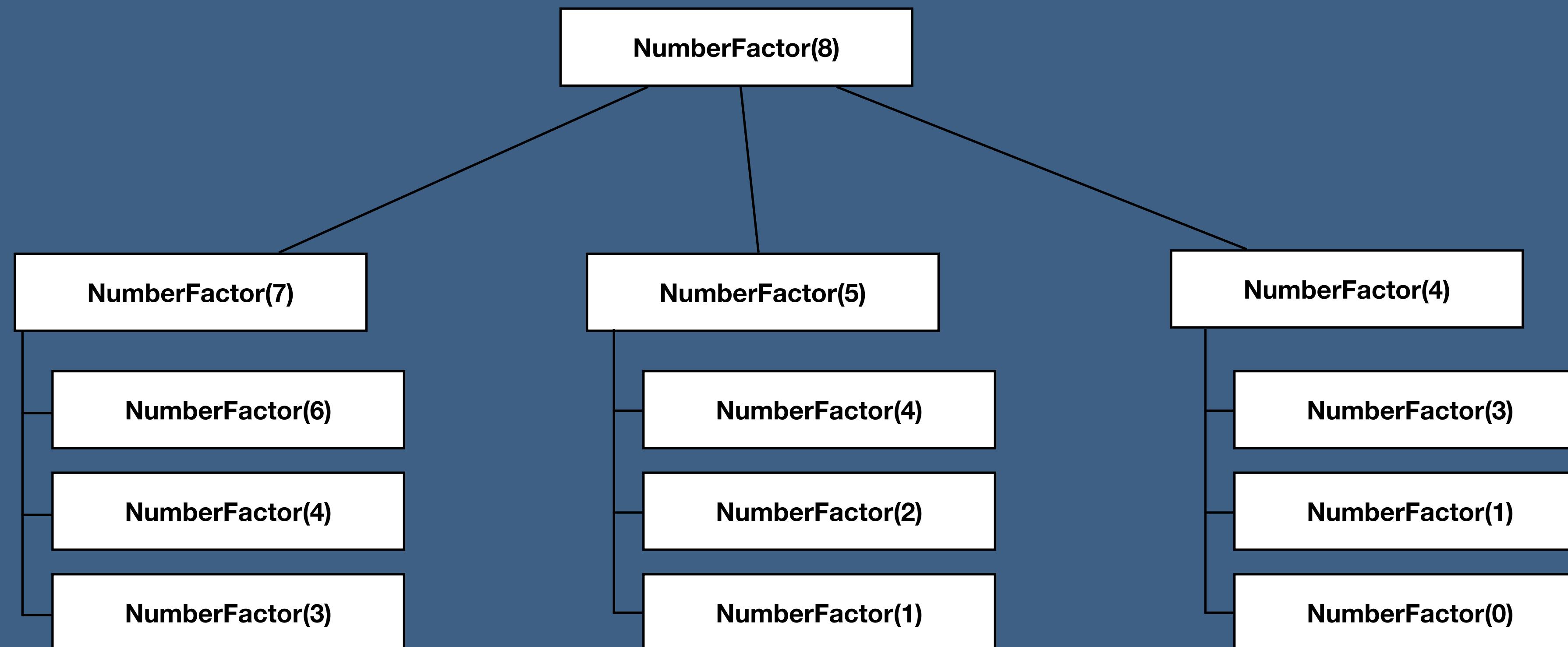
```
NumberFactor(N)
    If N in (0,1,2) return 1
    If N = 3 return 2
    Else
        return NumberFactor(N-1) + NumberFactor(N-3) + NumberFactor(N-4)
```



# Dynamic Programming - Number Factor

## Problem Statement:

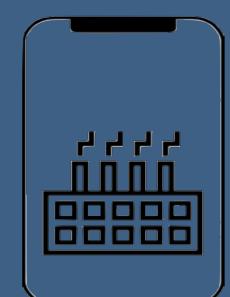
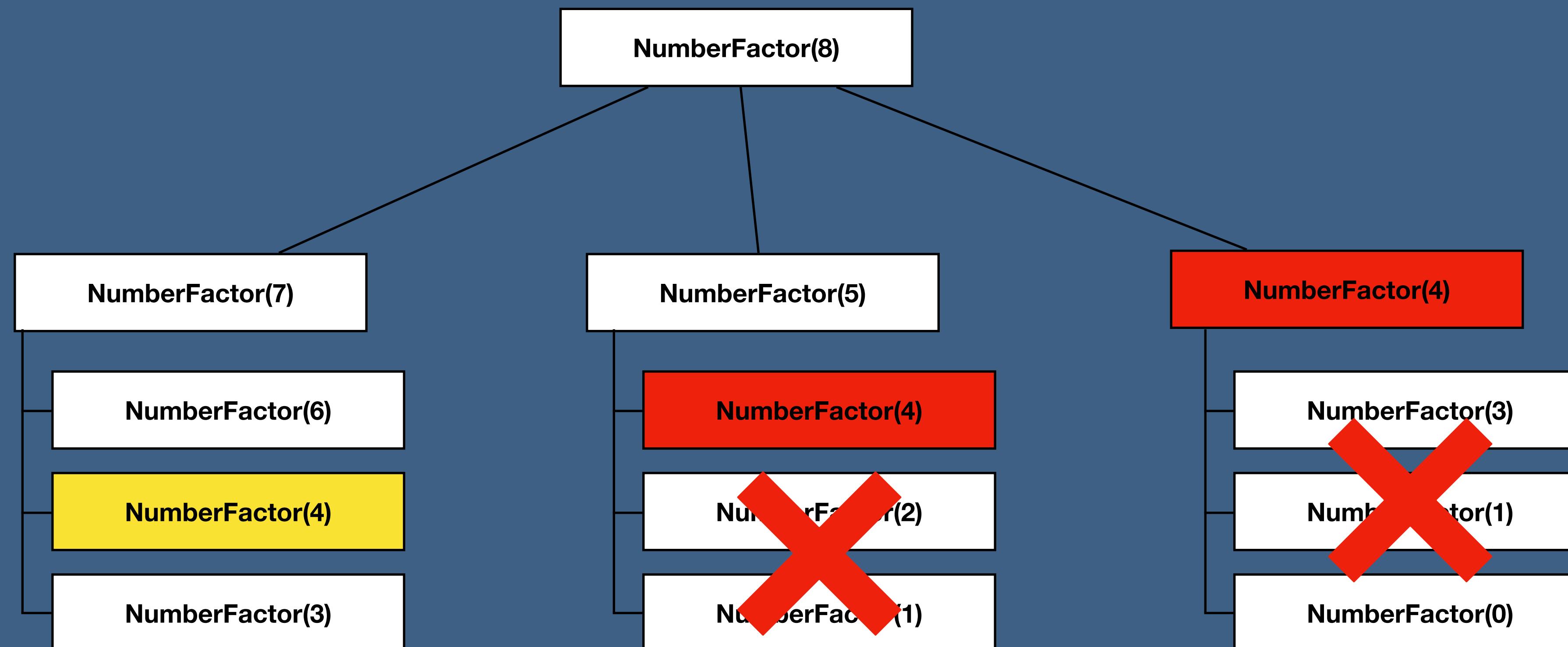
Given N, find the number of ways to express N as a sum of 1, 3 and 4.



# Dynamic Programming - Number Factor

## Problem Statement:

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

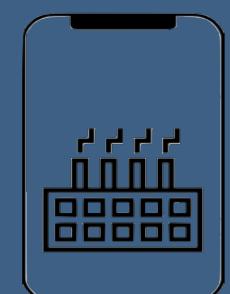


# Dynamic Programming - Number Factor

## 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  
        rec1 = NumberFactor(N-1)  
        rec2 = NumberFactor(N-3)  
        rec3 = NumberFactor(N-4)  
  
        return rec1 + rec2 + rec3
```

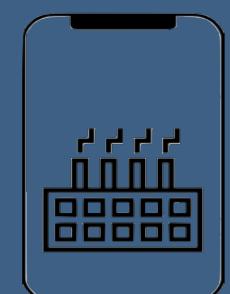


# Dynamic Programming - Number Factor

## 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
```

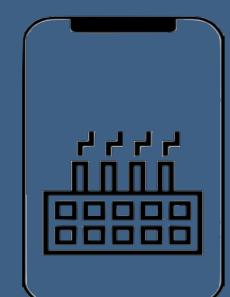


# Dynamic Programming - Number Factor

## Problem 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
```

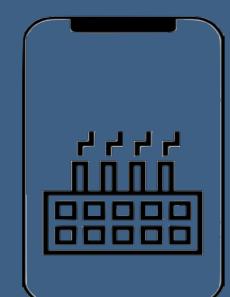


# Dynamic Programming - Number Factor

## Problem 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
```

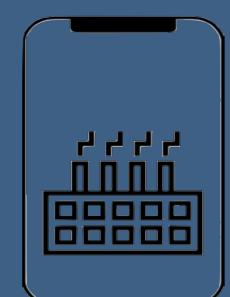


# Dynamic Programming - Number Factor

## Problem 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]
```

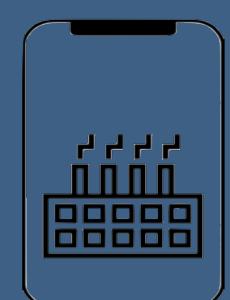


# Dynamic Programming - Number Factor

## Problem 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] → Step 2  
    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 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)
1	1	1	2				NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	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)
1	1	1	2		NF5+NF3+NF2	NF6+NF4+NF3	

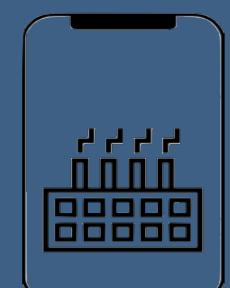
NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	4+1+1=6	NF5+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	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)
1	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)
1	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)
1	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	9+4+2=15



# Dynamic Programming - Number Factor

## 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)
1	1	1	2	NF3+NF1+NF0			

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	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)
1	1	1	2	NF3+NF1+NF0	NF4+NF2+NF1		

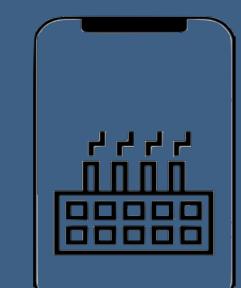
NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	2+1+1=4	4+1+1=6	NF6+NF3+NF2	NF6+NF4+NF3

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	1	1	2	NF3+NF1+NF0	NF4+NF2+NF1	NF6+NF3+NF2	

NF(0)	NF(1)	NF(2)	NF(3)	NF(4)	NF(5)	NF(6)	NF(7)
1	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)
1	1	1	2	NF3+NF1+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)
1	1	1	2	2+1+1=4	4+1+1=6	6+2+1=9	9+4+2=15



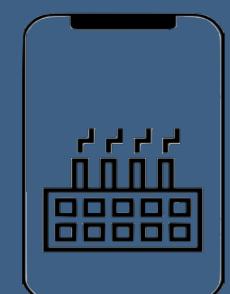
# Dynamic Programming - Number Factor

## Problem Statement:

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

## **Bottom Up Approach**

```
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]
```



# Dynamic Programming - 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**

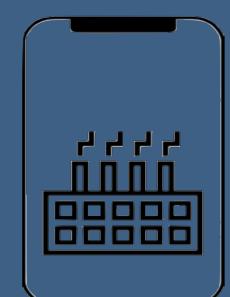
- Maximum amount = 41
- Houses that are stolen : 7, 30, 4

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



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

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

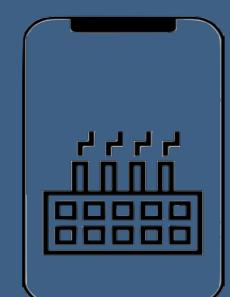


# Dynamic Programming - House Robber

## 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

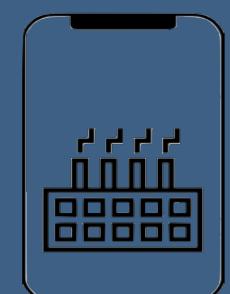
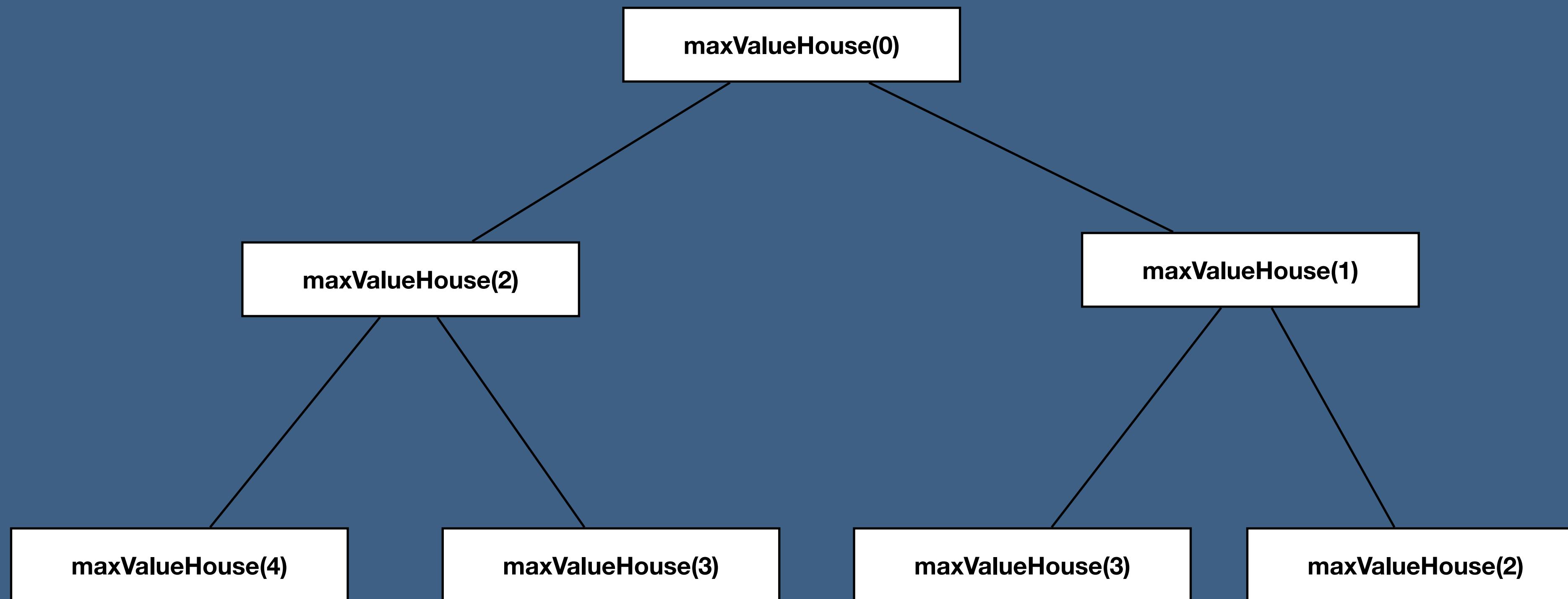
```
maxValueHouse(houses, currentHouse)
    If currentHouse > length of houses
        return 0
    Else
        stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)
        skipFirstHouse = maxValueHouse(houses, currentHouse+1)
        return max(stealFirstHouse, skipFirstHouse)
```



# Dynamic Programming - House Robber

## Problem Statement:

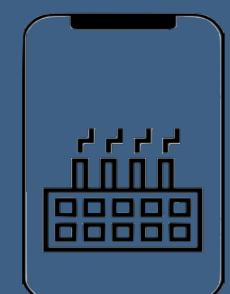
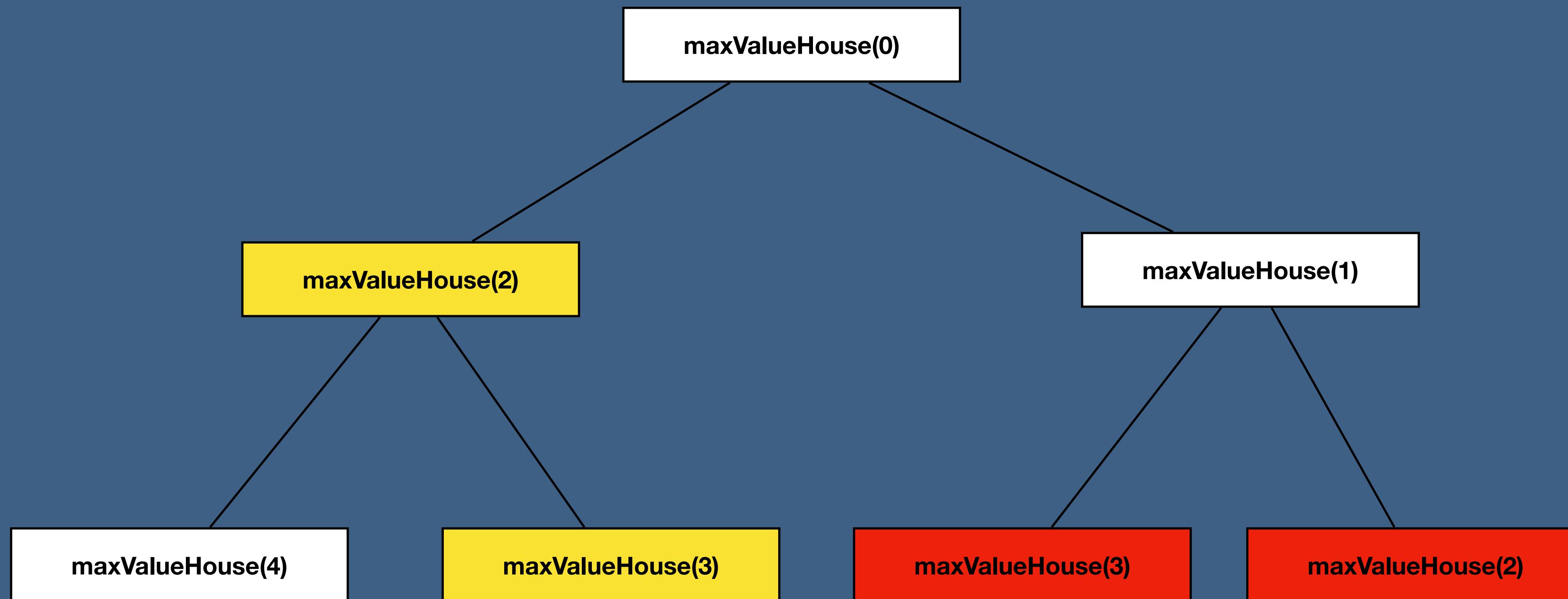
- Given N number of houses along the street with some amount of money
- Adjacent houses cannot be stolen
- Find the maximum amount that can be stolen



# Dynamic Programming - 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

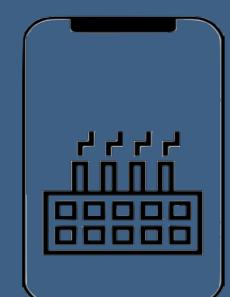


# Dynamic Programming - House Robber

## 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)
```

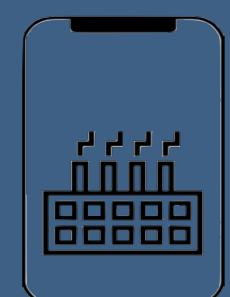


# Dynamic Programming - House Robber

## 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, dp): —————→ 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)
```

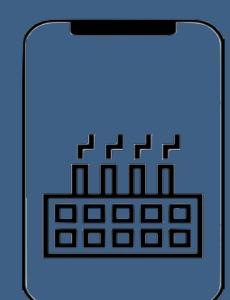


# Dynamic Programming - House Robber

## 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, dp):           —————→ Step 1
    If currentHouse > length of houses
        return 0
    Else
        If currentHouse not in dp:                  —————→ Step 2
            stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)
            skipFirstHouse = maxValueHouse(houses, currentHouse+1)
        return max(stealFirstHouse, skipFirstHouse)
```

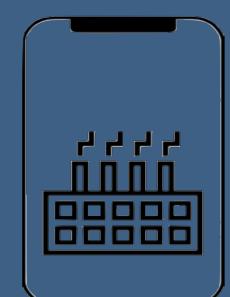


# Dynamic Programming - House Robber

## 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, dp): → Step 1
    If currentHouse > length of houses
        return 0
    Else
        If currentHouse not in dp: → Step 2
            stealFirstHouse = currentHouse + maxValueHouse(houses, currentHouse+2)
            skipFirstHouse = maxValueHouse(houses, currentHouse+1)
            dp[currentHouse] = max(stealFirstHouse, skipFirstHouse) → Step 3
        return dp[currentHouse] → Step 4
```



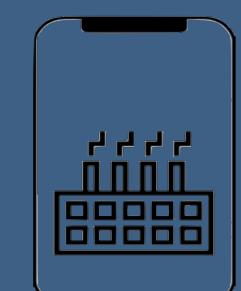
# Dynamic Programming - House Robber

## 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)



# Dynamic Programming - 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

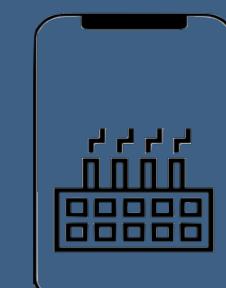
HR(0)	HR(1)	HR(2)	HR(3)	HR(4)	HR(5)	HR(6)
max(H0+HR2, HR1)	max(H1+HR3, HR2)	max(H2+HR4, HR3)	max(H3+HR5, HR4)	max(H4+HR6, HR5)	max(2+0, 4)=4	4

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



# Dynamic Programming - House Robber

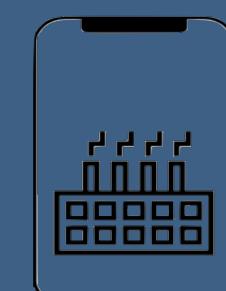
## Bottom Up Approach

### 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



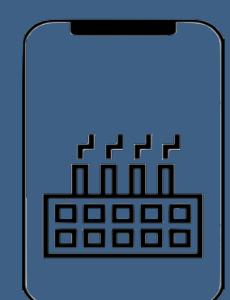
# Dynamic Programming - House Robber

## 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**

```
houseRobberBU(houses, currentIndex)
    tempAr = [0]*(length(houses)+2)
    for i in length(houses)-1 until -1
        tempAr[i] = max(houses[i]+tempAr[i+2], tempAr[i+1])
    return tempAr[0]
```



# Dynamic Programming - Convert String

## Problem Statement:

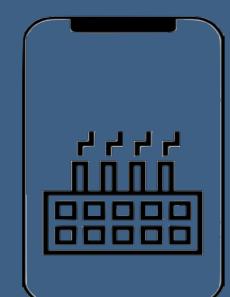
- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations

## **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”



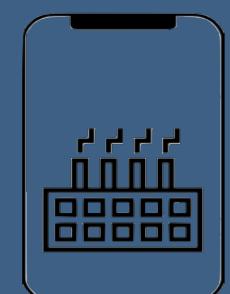
# Dynamic Programming - Convert String

## 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)

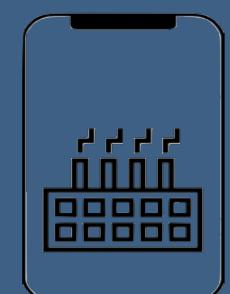
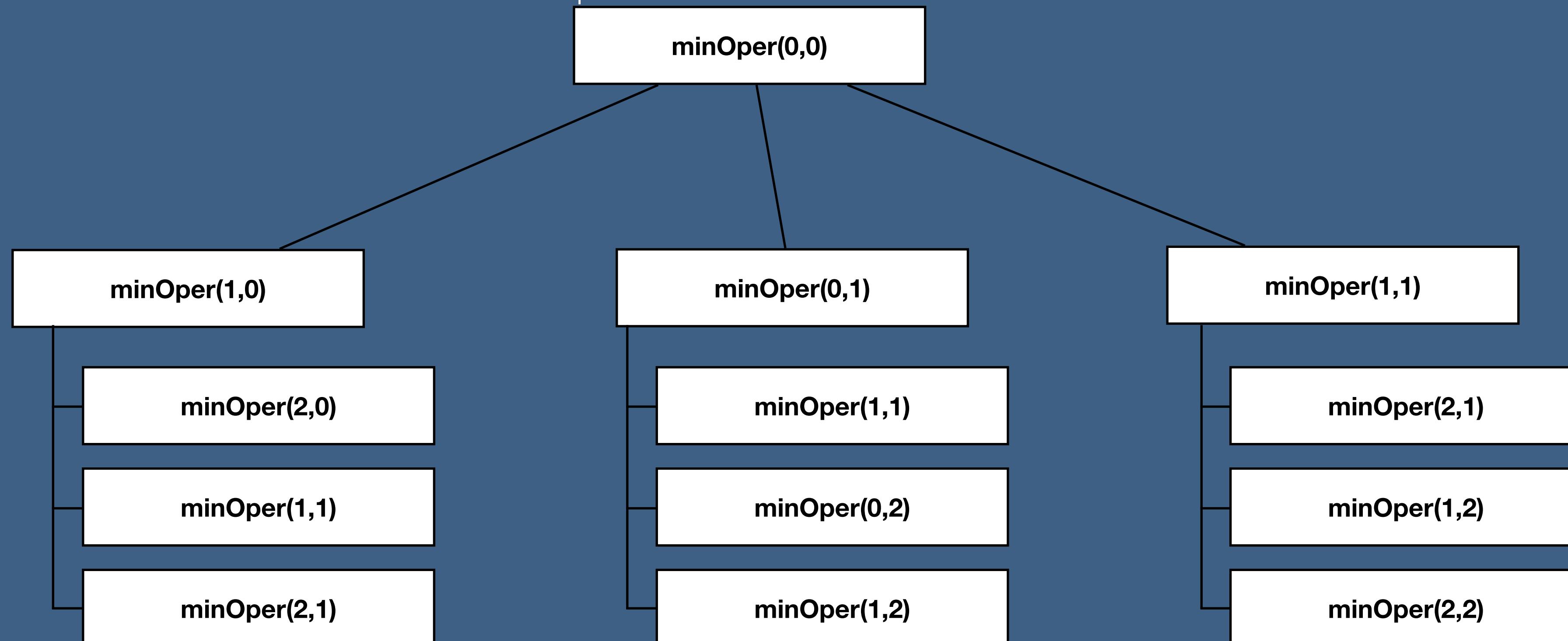
    Else
        deleteOp = 1 + findMinOperation(s1, s2, index1, index2+1)
        insertOp = 1 + findMinOperation(s1, s2, index1+1, index2)
        replaceOp = 1 + findMinOperation(s1, s2, index1+1, index2+1)
        return min(deleteOp, insertOp, replaceOp)
```



# Dynamic Programming - Convert String

## Problem Statement:

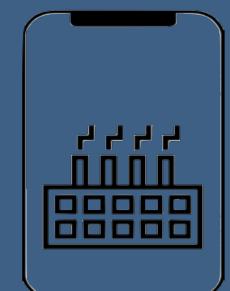
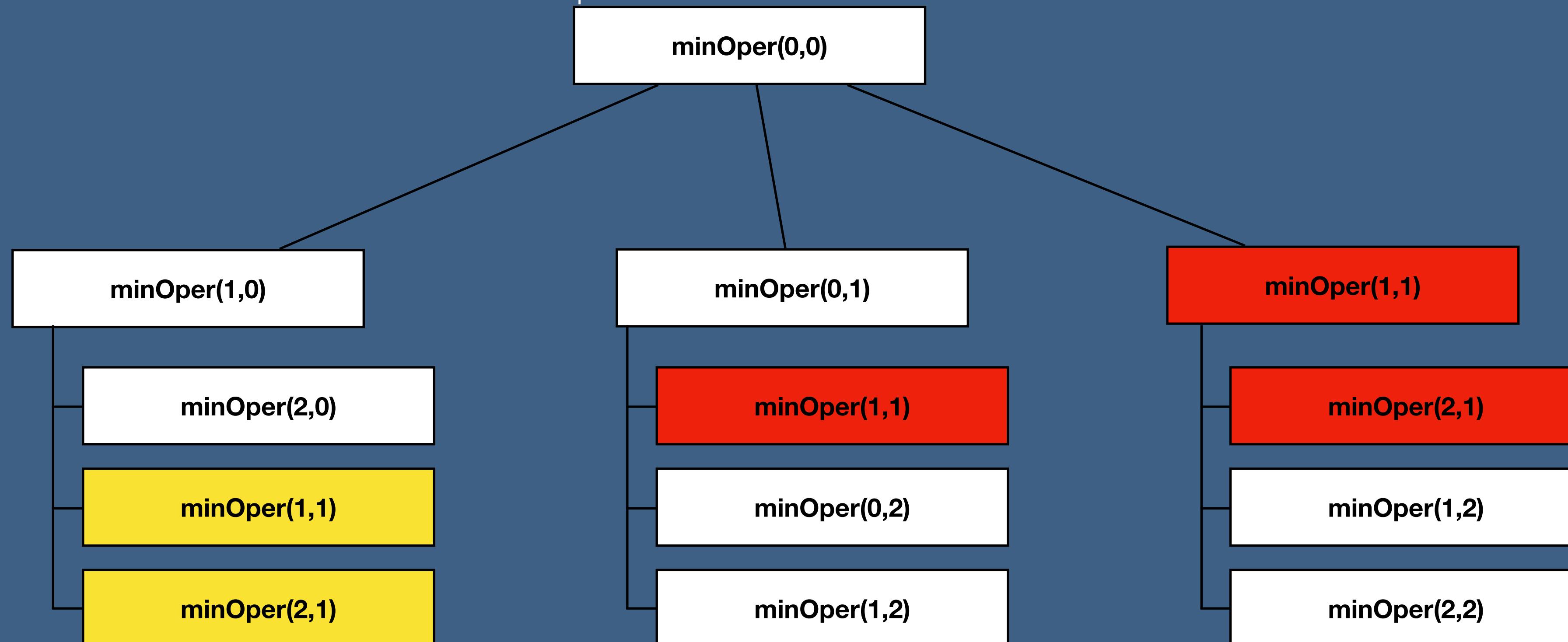
- S1 and S2 are given strings
- Convert S2 to S1 using delete, insert or replace operations
- Find the minimum count of edit operations



# Dynamic Programming - 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



# Dynamic Programming - 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

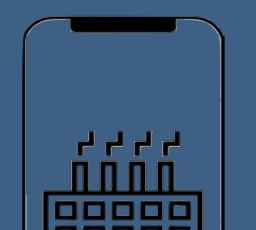
## Top Down Approach

```
findMinOperation(s1, s2, index1, index2, dp[][])
    if index1 == s1.length
        return s2.length-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 1
        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 2
        return tempDict[dictKey] → Step 3
```

```
if(dp[i1][i2] == null) {
    if(i1 == s1.length()) // if we have reached the end of s1, then insert all the remaining characters of s2
        dp[i1][i2] = s2.length() - i2;

    else if(i2 == s2.length()) // if we have reached the end of s2, then delete all the remaining characters of s1
        dp[i1][i2] = s1.length() - i1;

    else if(s1.charAt(i1) == s2.charAt(i2)) // If the strings have a matching character, recursively match for the remaining lengths.
        dp[i1][i2] = findMinOperationsTD(dp, s1, s2, i1+1, i2+1);
```



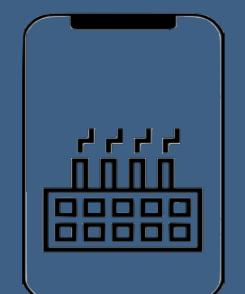
# Dynamic Programming - Convert String

## 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**

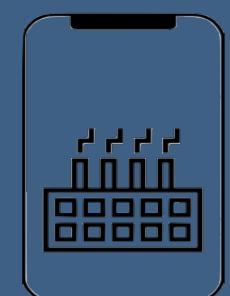
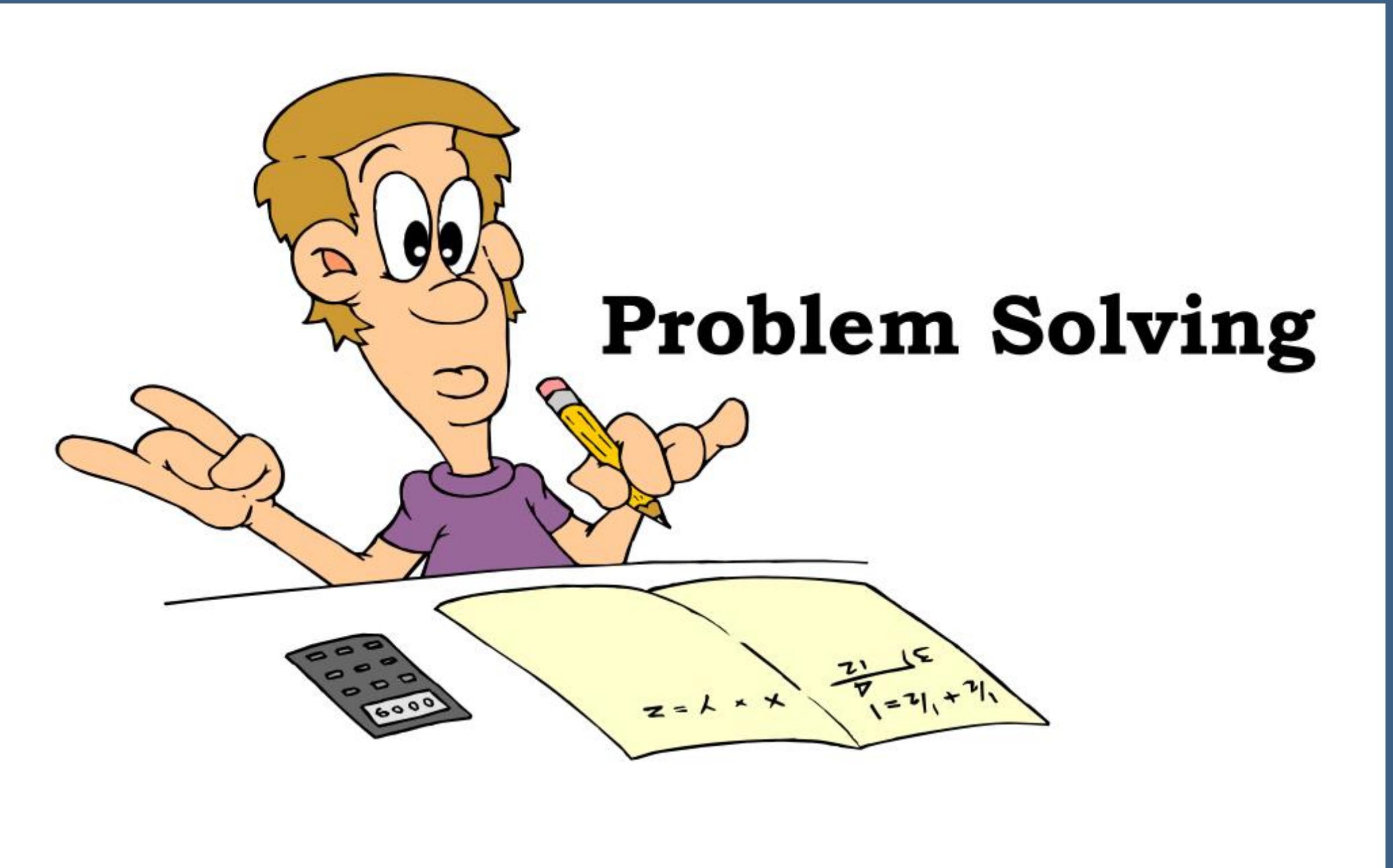
```
def findMinOperationBU(s1, s2, tempDict):  
    for i1 in range(len(s1)+1):  
        dictKey = str(i1) + '0'  
        tempDict[dictKey] = i1  
    for i2 in range(len(s2)+1):  
        dictKey = '0' + str(i2)  
        tempDict[dictKey] = i2  
  
    for i1 in range(1, len(s1)+1):  
        for 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]
```



# A Recipe for Problem Solving



# A Recipe for Problem Solving

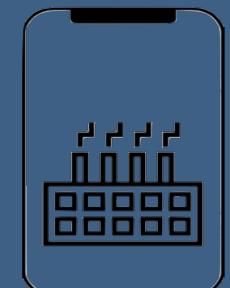


# A Recipe for Problem Solving

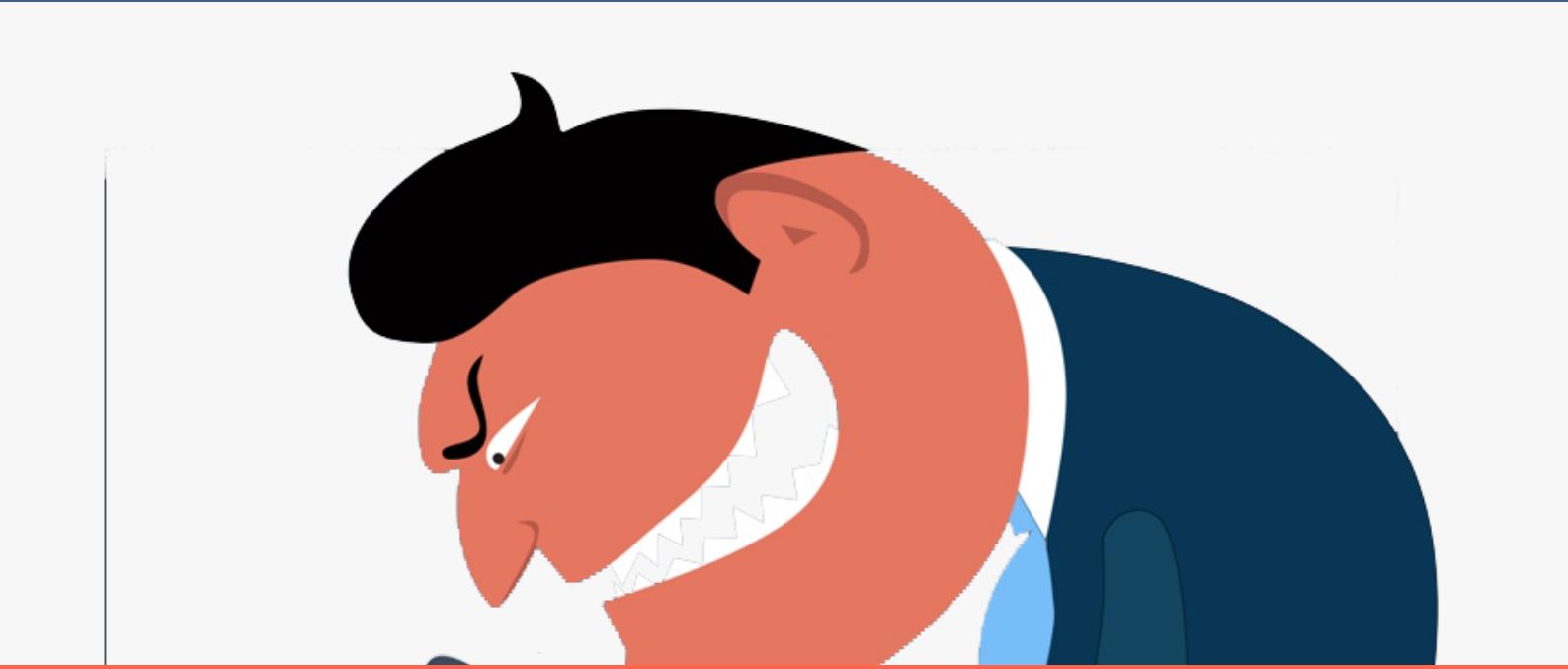
Algorithm : is a set of steps to accomplish a certain task

$$(x + a)^n = \sum_{k=0}^n \binom{n}{k} x^k a^{n-k}$$

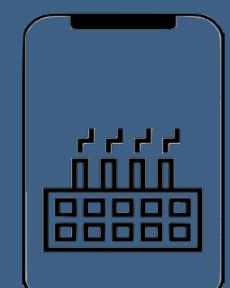
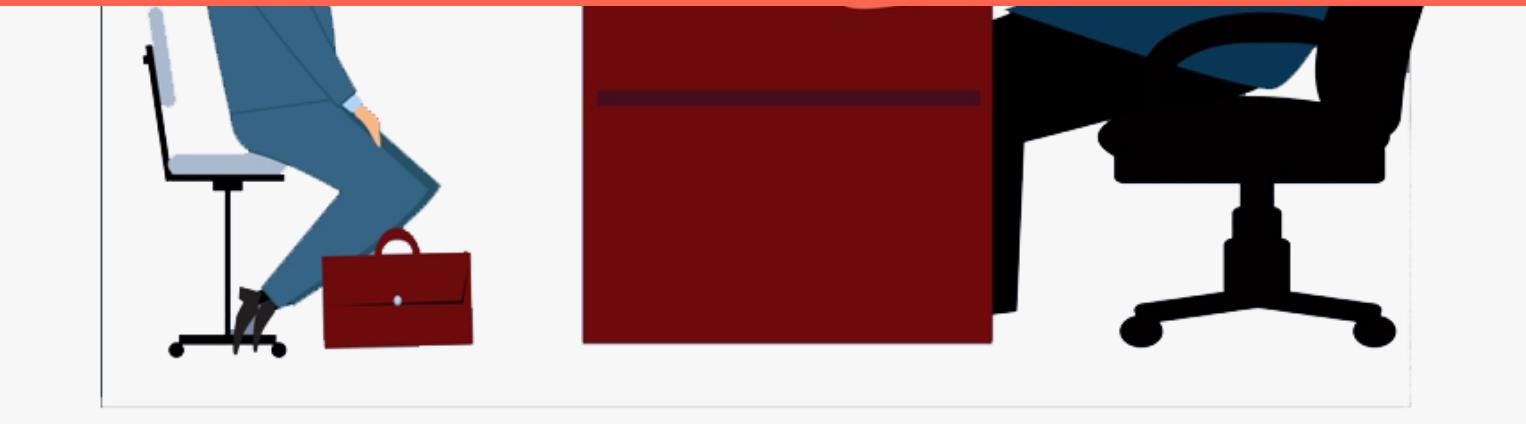
$$\begin{aligned} & x^2 + 2x - 1 \quad y = \frac{1}{2}(-12x^2 + 6x + 2) \\ & 1) \quad y = 1 \quad (\cos x + 1) + x \quad (-\sin x) = \cos x - x \sin x + 1 \\ & x_0 = ? \quad y(x_0) \approx 1 \quad y = 6x^2 - 4x + 1 \quad x_0 = 0 \quad O+1 \\ & x_1 = \frac{2}{3}, A(0, -t) \quad B\left(\frac{2}{3}, -\frac{17}{27}\right) \quad y \approx \frac{3}{2} \quad t \approx \frac{\sqrt{5}}{2} \\ & x_2 \approx 3 \quad y(x_2) \approx 6x^2 - 4x + 1 \quad x_2 \approx 2 \approx 0.3x^2 \\ & x \approx \frac{1}{3} \quad A(t, O) \quad B\left(-\frac{1}{3}, -\frac{49}{27}\right) \quad x \approx 2x \quad y \approx \frac{1}{y} \\ & x = t - O_1 \quad t_0 \approx 3 \times (t) \quad t_0 \approx 1 \quad (-) \\ & \approx \frac{1}{3} \quad x_0 = 1 \quad f = \frac{x}{x^2 - 1} \approx \frac{1}{2} \\ & 3 \times \frac{1}{3} \times (3) = 2 \times 3 \times \frac{1}{2} \quad 1, 3 \approx 0, 0214 \\ & x \approx x + 1 \quad y(1) \approx 1 \quad \text{if } a = y(1) = 1 \\ & y = \frac{2}{x+1} \quad y = -\frac{2}{(x+1)^2} \quad \int \frac{dx}{x^3} \\ & \int x \sqrt{x} dx = \int x^{\frac{3}{2}} dx = \frac{2}{7} x^{\frac{5}{2}} \Big|_0^1 = \frac{2}{7} \end{aligned}$$



# A Recipe for Problem Solving



5 STEPS FOR PROBLEM SOLVING



# A Recipe for Problem Solving

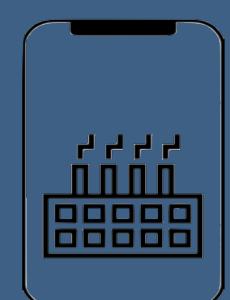
UNDERSTAND THE PROBLEM

EXPLORE EXAMPLES

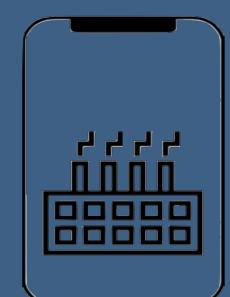
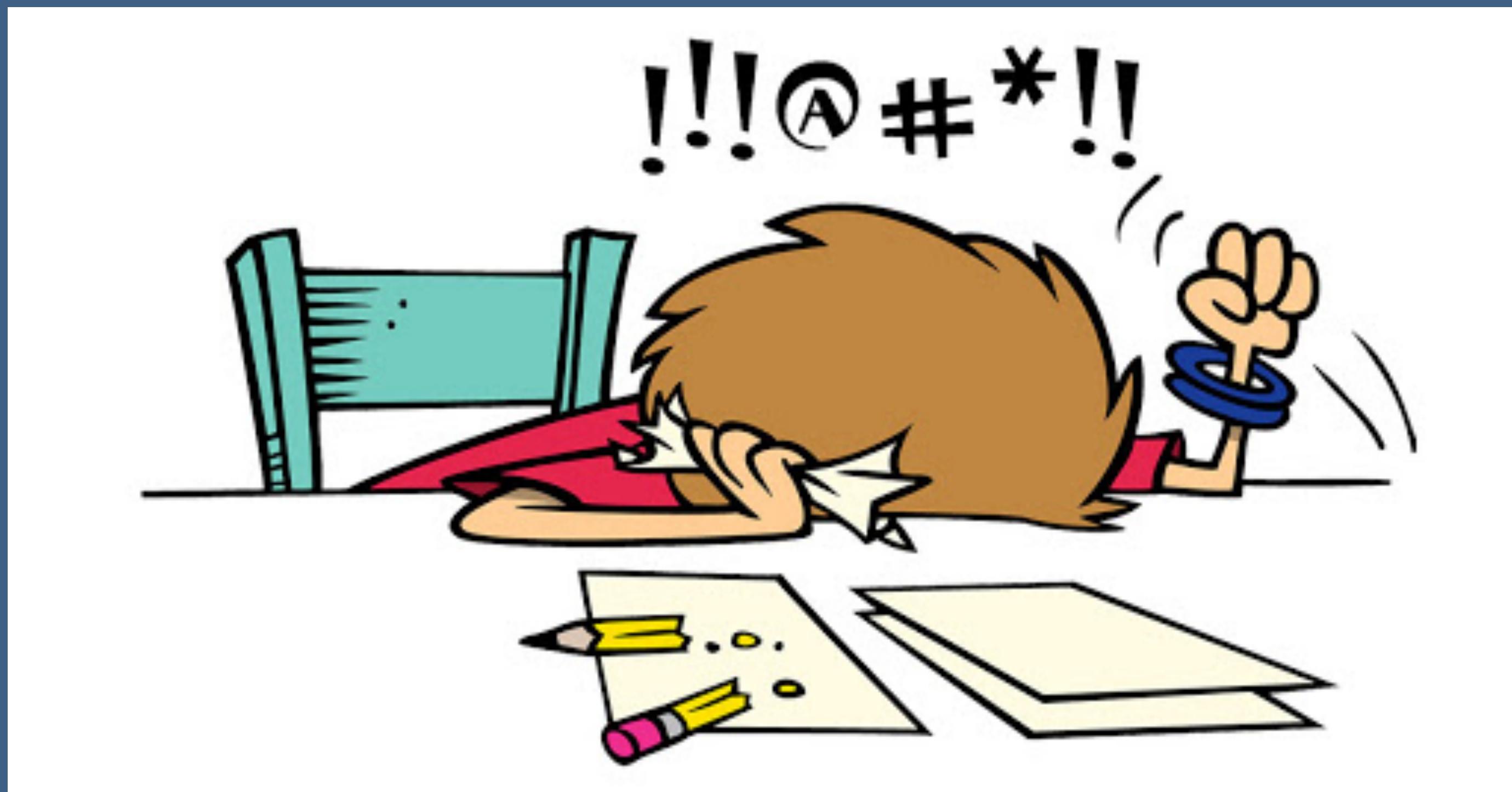
BREAK IT DOWN

SOLVE / SIMPLIFY

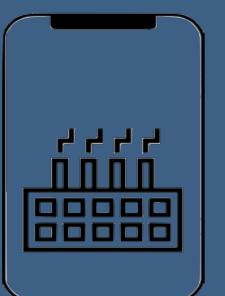
LOOK BACK REFACTOR



## Step 1 - UNDERSTAND THE PROBLEM

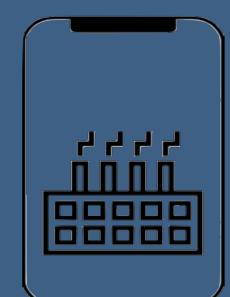


# Step 1 - UNDERSTAND THE PROBLEM



# 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?

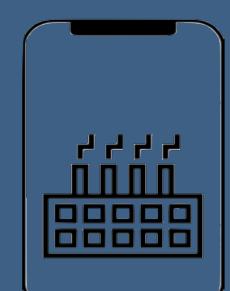
Integer? Float? Or?

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?

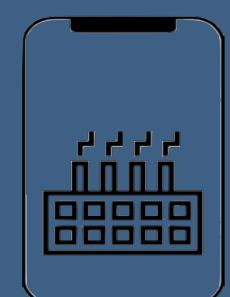
Add, Sum



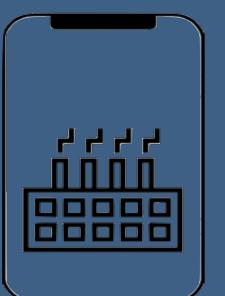
# A Recipe for Problem Solving

UNDERSTAND THE PROBLEM

EXPLORE EXAMPLES

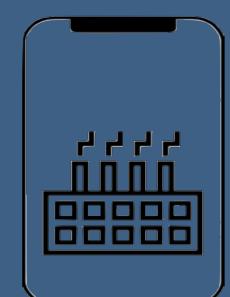


## Step 2 - EXPLORE EXAMPLES



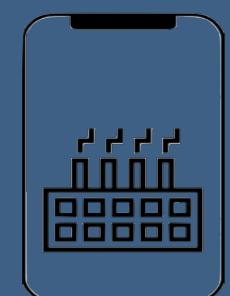
## Step 2 - EXPLORE EXAMPLES

1. Start with simple examples
2. Progress to more complex examples
3. Explore examples with empty
4. Explore the examples with invalid inputs



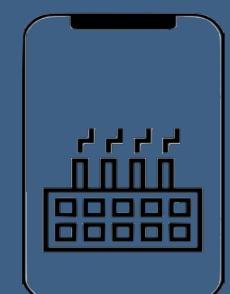
## Step 2 - EXPLORE EXAMPLES

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

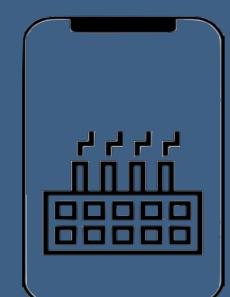


## SOLVE / SIMPLIFY

Solve the Problem

If you cannot...

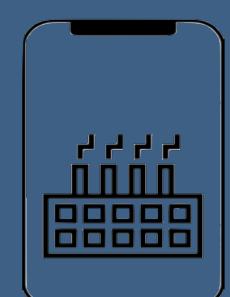
Simplify the Problem



## SOLVE / SIMPLIFY

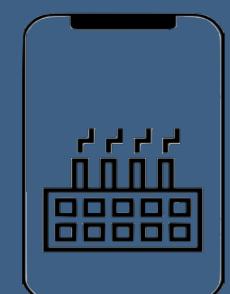
### Simplify the Problem

- 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

