

# Laboratorio de Programación

## 2016 -2

Juan Camilo Rada<sup>1</sup>

<sup>1</sup>Departamento de Electronica y ciencias de la computación

Octubre de 2016

# Outline

## 1 Function Pointers

- Basics
- Functions in structures

## 2 Tree

- Basics
- Binary Tree
  - Inserting a node
  - Traversing a tree

# Outline

## 1 Function Pointers

- Basics
- Functions in structures

## 2 Tree

- Basics
- Binary Tree
  - Inserting a node
  - Traversing a tree

# Function Pointers

## Basics

- Is a characteristic of the *Third Generation* of programming languages [Read More!](#)

# Function Pointers

## Basics

- Is a characteristic of the *Third Generation* of programming languages [Read More!](#)
- The pointer points to executable code within memory

# Function Pointers

## Basics

- Is a characteristic of the *Third Generation* of programming languages [Read More!](#)
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function

# Function Pointers

## Basics

- Is a characteristic of the *Third Generation* of programming languages [Read More!](#)
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*

# Function Pointers

## Basics

- Is a characteristic of the *Third Generation* of programming languages [Read More!](#)
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*
- Functions passed as an argument are widely known as *Callbacks*

# Function Pointers

## Basics

- Is a characteristic of the *Third Generation* of programming languages [Read More!](#)
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*
- Functions passed as an argument are widely known as *Callbacks*
- The function is expected to be called back at some convenient time

# Function Pointers

```
int myFunction(double a, char b)
```

# Function Pointers

```
int myFunction(double a, char b)  
int (*pointer) (double, char)
```

## Declaring a pointer

- `int (*pointer) → int myFunction`

The pointer type has to be equal to the function return type

# Function Pointers

```
int myFunction(double a, char b)  
int (*pointer) (double, char)
```

## Declaring a pointer

- `int (*pointer) → int myFunction`  
The pointer type has to be equal to the function return type
- `(double, char) → (double a, double b)`  
Types of the function members

# Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

## Declaring a pointer

- `int (*pointer) → int myFunction`  
The pointer type has to be equal to the function return type
- `(double, char) → (double a, double b)`  
Types of the function members
- `int (*pointer) (double, char) = myFunction`  
The pointer points to myFunction

# Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

## Declaring a pointer

- `int (*pointer) → int myFunction`  
The pointer type has to be equal to the function return type
- `(double, char) → (double a, double b)`  
Types of the function members
- `int (*pointer) (double, char) = myFunction`  
The pointer points to myFunction
- `void process(void (*funcp)(int), int a, int b)`  
A function which receives a function as a parameter

# Function Pointers

## Test the code

```
#include <stdio.h>

void callback(int value)
{
    printf("This Callback function prints the value = %i", value);
}
void process(void (*funcp)(int), int a, int b)
{
    int c = a + b;
    funcp(c);
}
int main()
{
    void (*functionPointer)(int);
    functionPointer = callback;
    process(functionPointer, 2, 3);
}
```

# Function Pointers

Test the code

```
#include <stdio.h>
void callback()
{
    printf("The process has finished!!!\n");
}
void process(void (*funcp)(), int *a, int b)
{
    *a = (*a * *a) + b;
    funcp();
}
int main()
{
    int var = 5;
    void (*functionPointer)();
    functionPointer = callback;
    process(functionPointer, &var, 3);
    printf("a = %d", var);
}
```

# Function Pointers

## Test the code

```
#include <stdio.h>
int sumCallback(int a, int b)
{
    return a + b;
}
void process(int (*funcp)(int, int), int a, int b)
{
    printf("a = %d, b = %d, res = %d", a, b, funcp(a, b));
}
int main()
{
    int (*functionPointer)(int, int);
    functionPointer = sumCallback;
    process(functionPointer, 5, 3);
}
```

# Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

## The map function

- Is a *High-order* function

# Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

## The map function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments

# Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

## The `map` function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments
- The `map` function applies the input function over all the elements in an array

# Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

## The `map` function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments
- The `map` function applies the input function over all the elements in an array
- *Keep in mind:* It works also for sequential *containers*

# Function Pointers

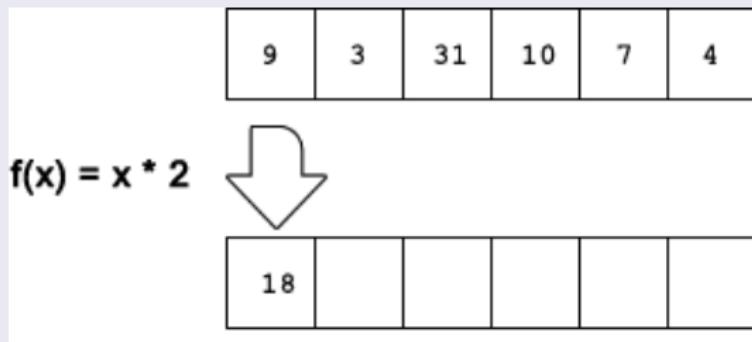
The map function - How it works?

9	3	31	10	7	4
---	---	----	----	---	---

$$f(x) = x * 2$$

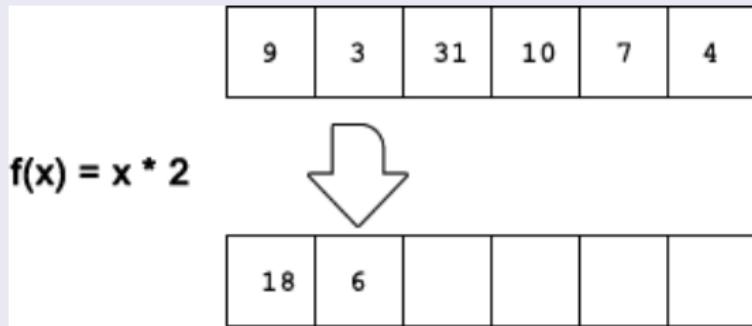
# Function Pointers

The `map` function - How it works?



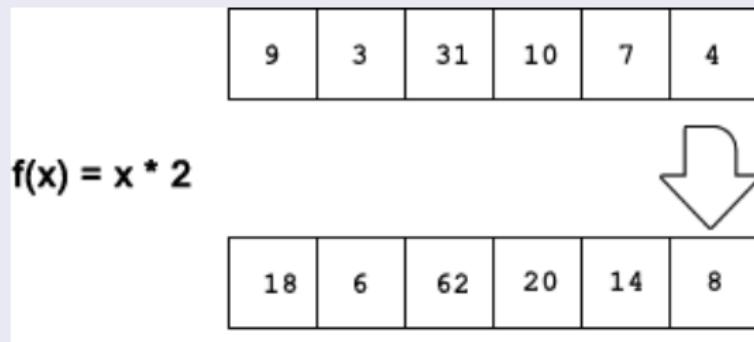
# Function Pointers

The `map` function - How it works?



# Function Pointers

The `map` function - How it works?



# Function Pointers

## The map function - Implementation

```
int* map(int (*funPtr)(int), int* ptr, int size)
{
    int* ptrRes = calloc(size, sizeof(int));
    for (int i=0; i<size; i++)
    {
        *(ptrRes+i) = funPtr(ptr[i]);
    }
    return ptrRes;
}
```

# Function Pointers - Test the code with the map function

```
#include <stdio.h>
#include <stdlib.h>

int getDouble(int a)
{
    return a*2;
}

int main()
{
    int *a = calloc(2, sizeof(int));
    int (*fun)(int) = getDouble;
    a[0] = 5;
    a[1] = 9;

    int* array = map(fun, a,2);
    printf("%d, %d", array[0], array[1]);
}
```

# Outline

## 1 Function Pointers

- Basics
- Functions in structures

## 2 Tree

- Basics
- Binary Tree
  - Inserting a node
  - Traversing a tree

# Functions in structures

## The Queue structure

```
#include <stdio.h>
#include <stdlib.h>

struct queue
{
```

# Functions in structures

## The Queue structure

```
#include <stdio.h>
#include <stdlib.h>

struct queue
{
```

The basic structure contains:

- A pointer to the head of the queue

# Functions in structures

## The Queue structure

```
#include <stdio.h>
#include <stdlib.h>

struct queue
{
```

The basic structure contains:

- A pointer to the head of the queue
- A pointer to the tail of the queue

# Functions in structures

## The Queue structure

```
#include <stdio.h>
#include <stdlib.h>

struct queue
{
```

- What if we want to *encapsulate* all the queue functions inside the structure?

# Functions in structures

## The Queue structure

```
#include <stdio.h>
#include <stdlib.h>

struct queue
{
```

- What if we want to *encapsulate* all the queue functions inside the structure?
- We can use pointers to functions

# Functions in structures

## The Queue structure

```
    Node * tail;  
};  
struct queue  
{  
    Node * head;  
    Node * tail;  
    void (*enqueue)(struct queue *, int);
```

- **void** (\*enqueue)(**struct** queue \*, **int**);

# Functions in structures

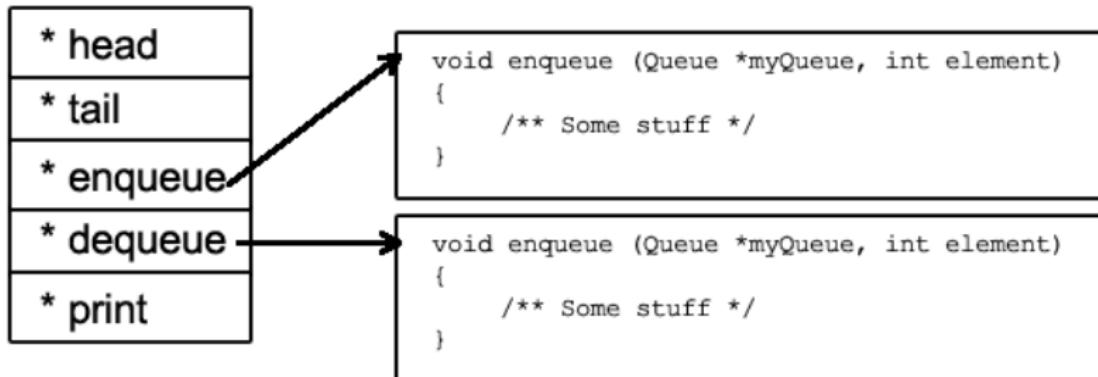
## The Queue structure

```
    Node * tail;  
};  
struct queue  
{  
    Node * head;  
    Node * tail;  
    void (*enqueue)(struct queue *, int);
```

- **void** (\*enqueue)(**struct** queue \*, **int**);
- We create three new pointers to every handled operation

# Functions in structures

## Queue



# Functions in structures

## The Queue structure - The Constructor function

```
    void (*print)(struct queue *);  
};  
typedef struct queue Queue;  
  
void enqueue (Queue *myQueue, int element)  
{
```

- The constructor will provide us an instance of a Queue

# Functions in structures

## The Queue structure - The Constructor function

```
    void (*print)(struct queue *);  
};  
typedef struct queue Queue;  
  
void enqueue (Queue *myQueue, int element)  
{
```

- The constructor will provide us an instance of a Queue
- We must associate every pointer to its respective function

# Functions in structures

## The Queue structure - The Constructor function

```
myNode->value = element;  
myNode->next = NULL;
```

# Functions in structures

## The Queue structure - The Constructor function

```
myNode->value = element;  
myNode->next = NULL;
```

- We have to call constructor() every time we need a new instance of Queue

# Functions in structures

## The Queue structure - The Constructor function

```
myNode->value = element;  
myNode->next = NULL;
```

- We have to call `constructor()` every time we need a new instance of Queue
- All the functions has to be called with the `->` operator

# Functions in structures

## The Queue structure - The Constructor function

```
myNode->value = element;  
myNode->next = NULL;
```

- We have to call `constructor()` every time we need a new instance of Queue
- All the functions has to be called with the `->` operator

# Functions in structures

## The Queue structure - The Constructor function

```
myNode->value = element;  
myNode->next = NULL;
```

Be careful!. The pointer to every function can be modified so, it can be unsafe!!!

# Outline

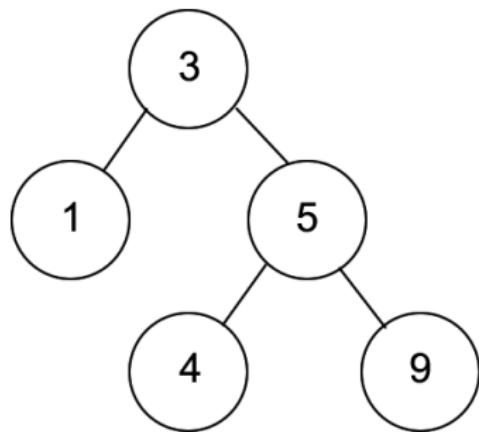
## 1 Function Pointers

- Basics
- Functions in structures

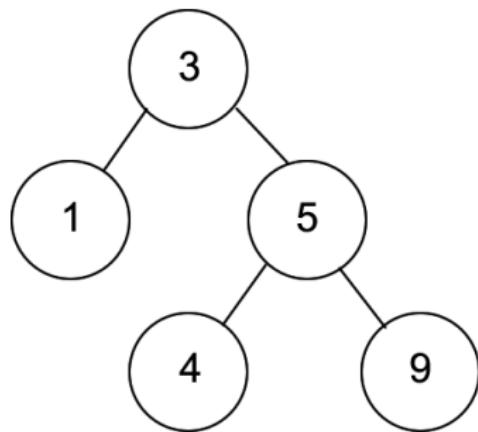
## 2 Tree

- Basics
- Binary Tree
  - Inserting a node
  - Traversing a tree

## Trees



# Trees



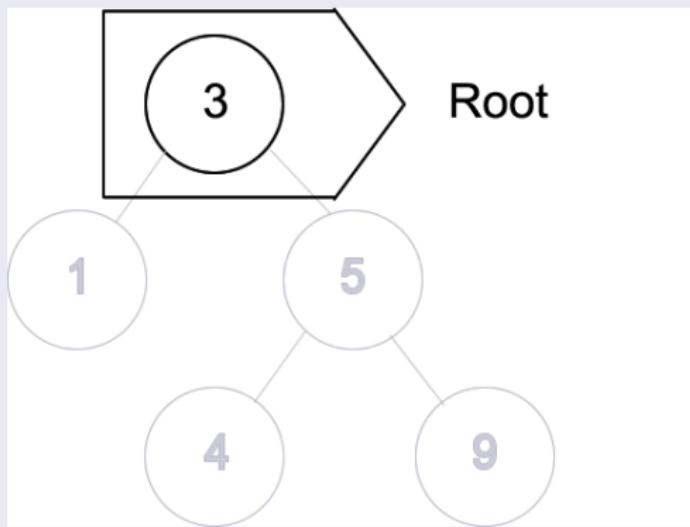
## Trees

Trees can be defined as a

- *Nonlinear, two-dimensional data structure*

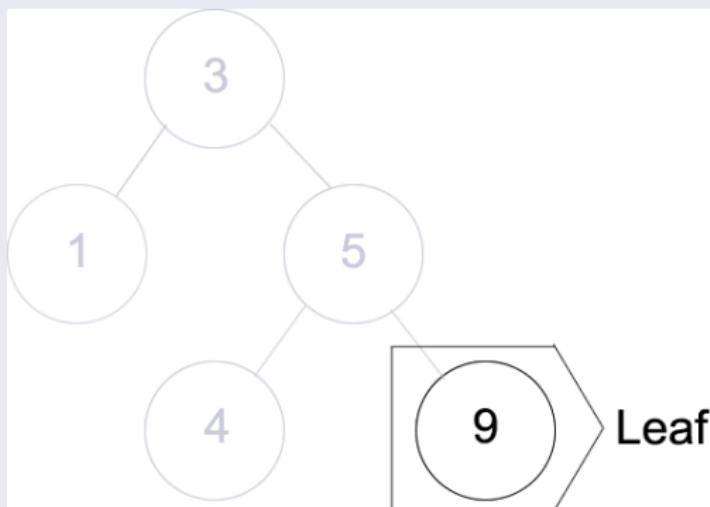
# Trees

## Parts



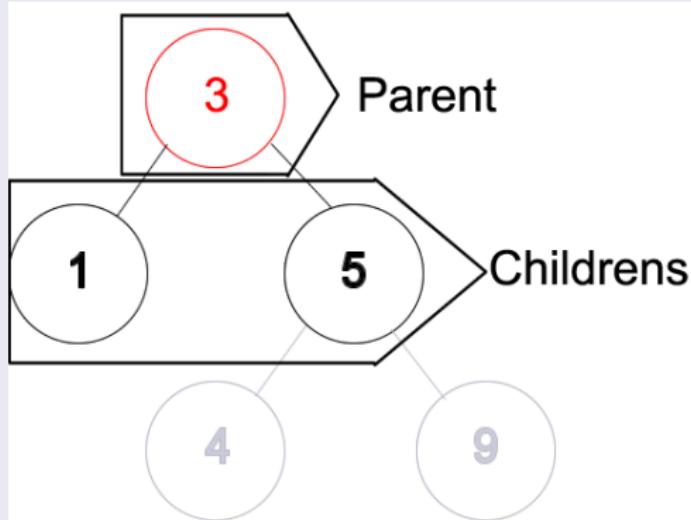
# Trees

## Parts



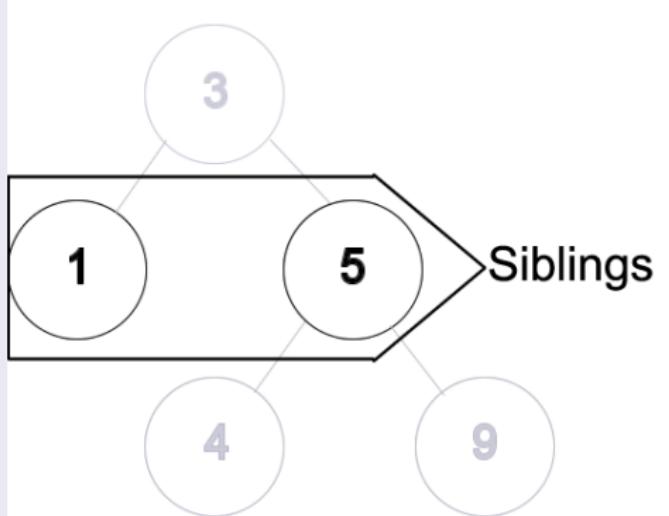
# Trees

## Parts



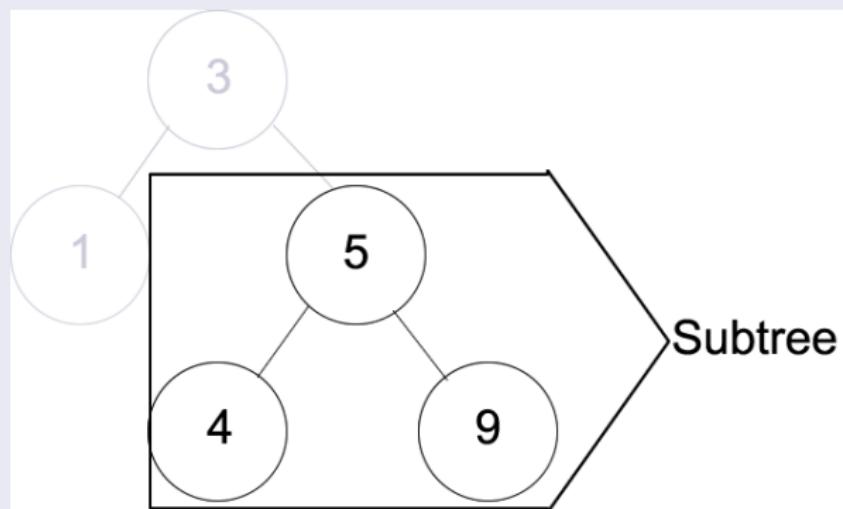
# Trees

## Parts



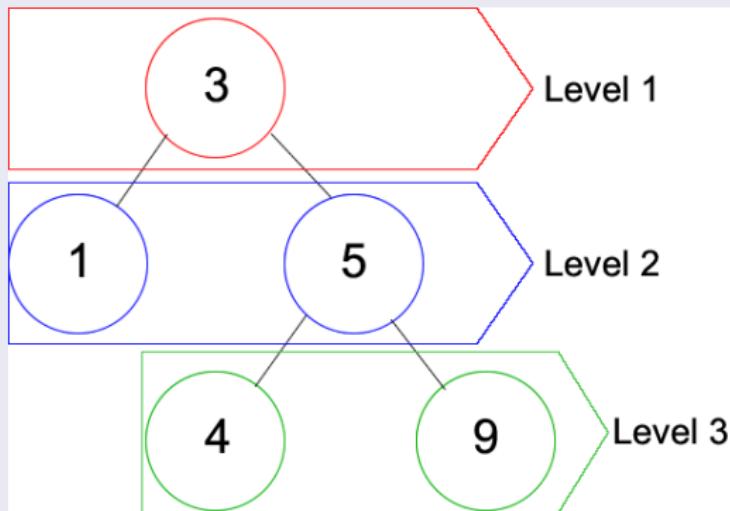
# Trees

## Parts



# Trees

## Parts



# Outline

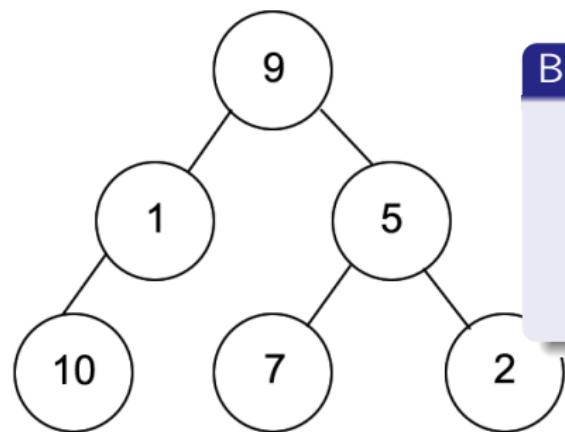
## 1 Function Pointers

- Basics
- Functions in structures

## 2 Tree

- Basics
- Binary Tree
  - Inserting a node
  - Traversing a tree

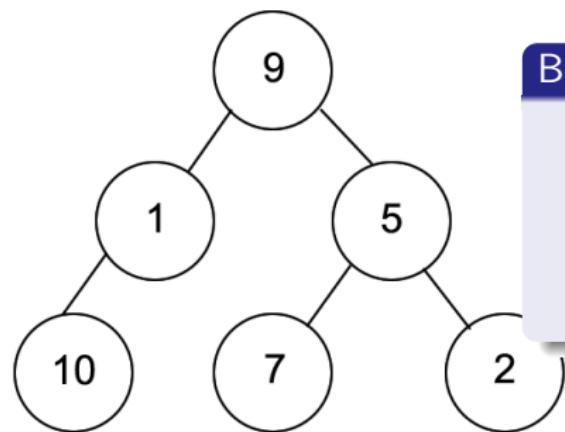
## Trees



## Binary Tree

- Each node has at most two children

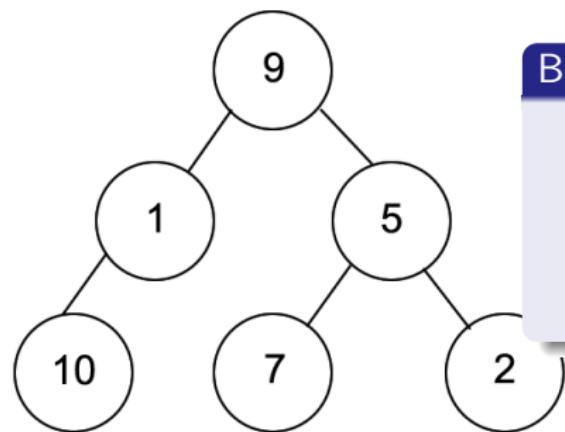
# Trees



## Binary Tree

- Each node has at most two children
- Left child

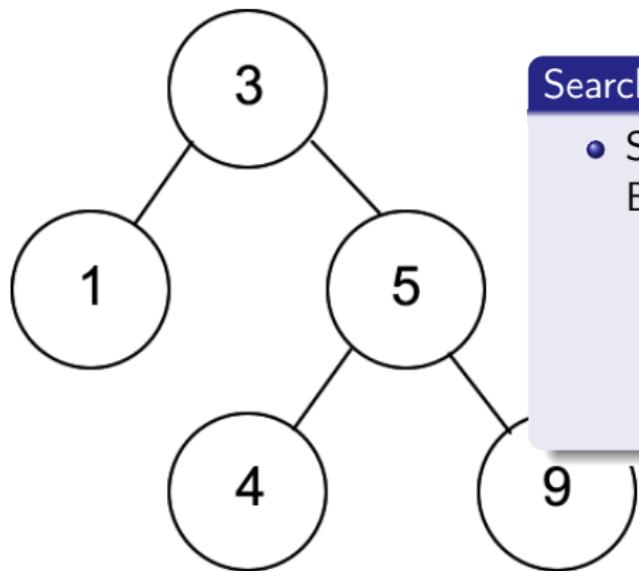
# Trees



## Binary Tree

- Each node has at most two children
- Left child
- Right child

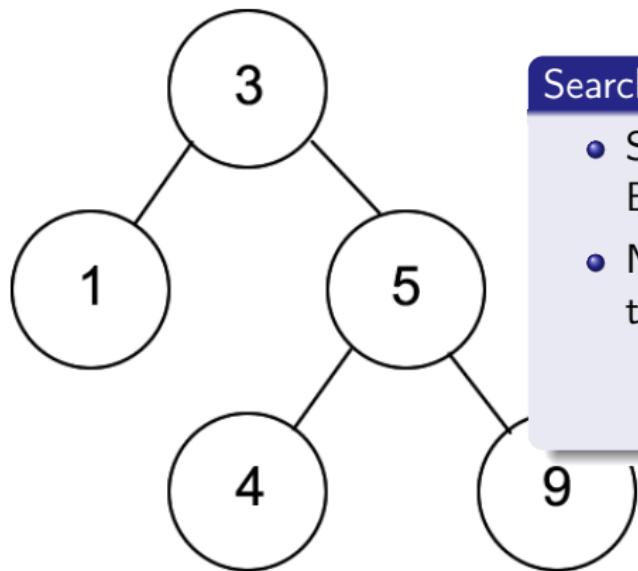
## Trees



## Search Binary Tree

- Share the basic characteristics of a Binary Tree

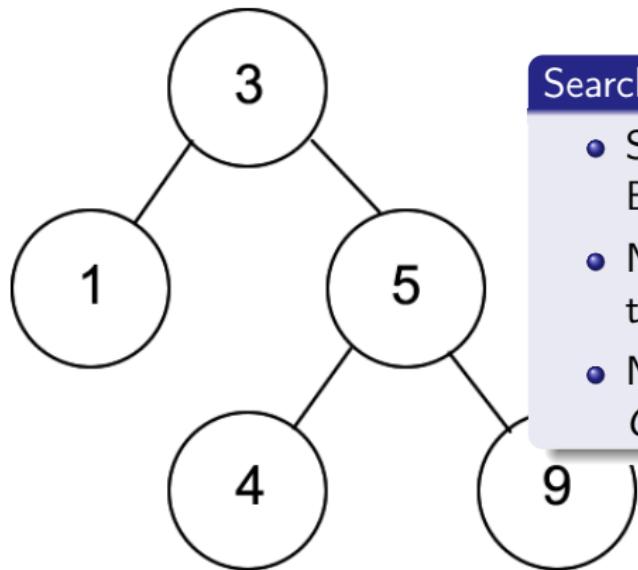
## Trees



## Search Binary Tree

- Share the basic characteristics of a Binary Tree
- Nodes at the left subtree are *Lower* than the root value

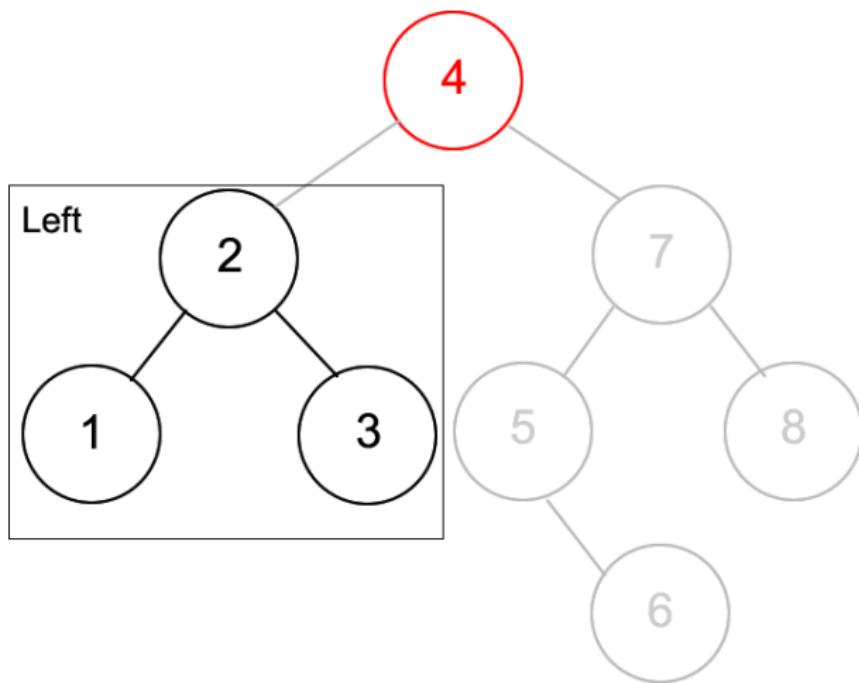
## Trees



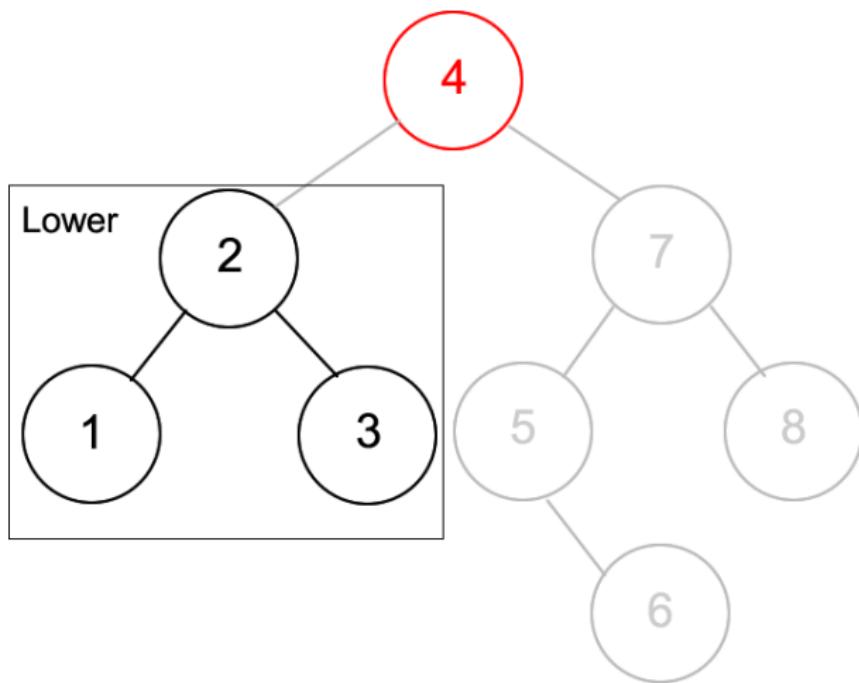
## Search Binary Tree

- Share the basic characteristics of a Binary Tree
- Nodes at the left subtree are *Lower* than the root value
- Nodes at the right subtree are *Greater* than the root value

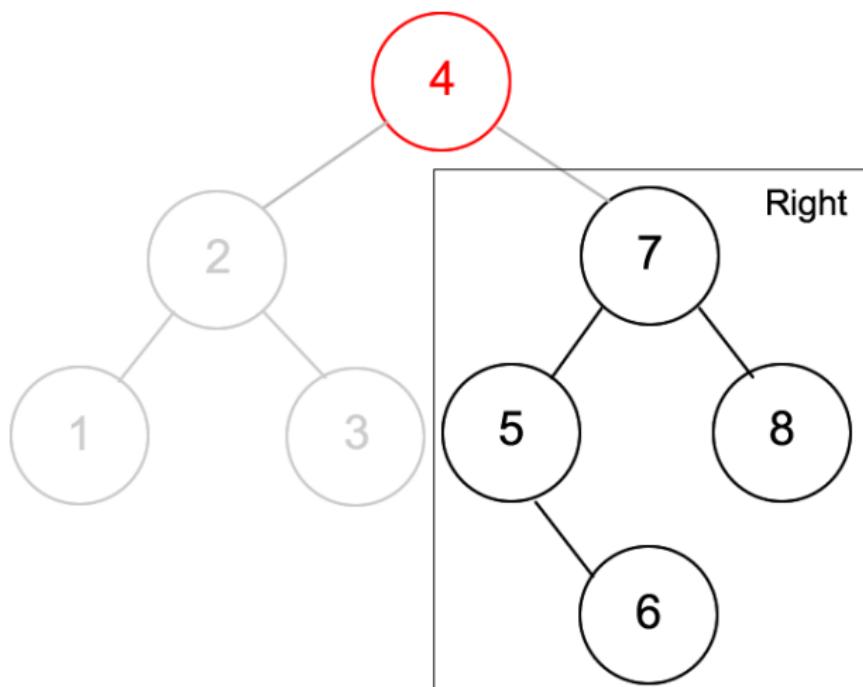
## Trees



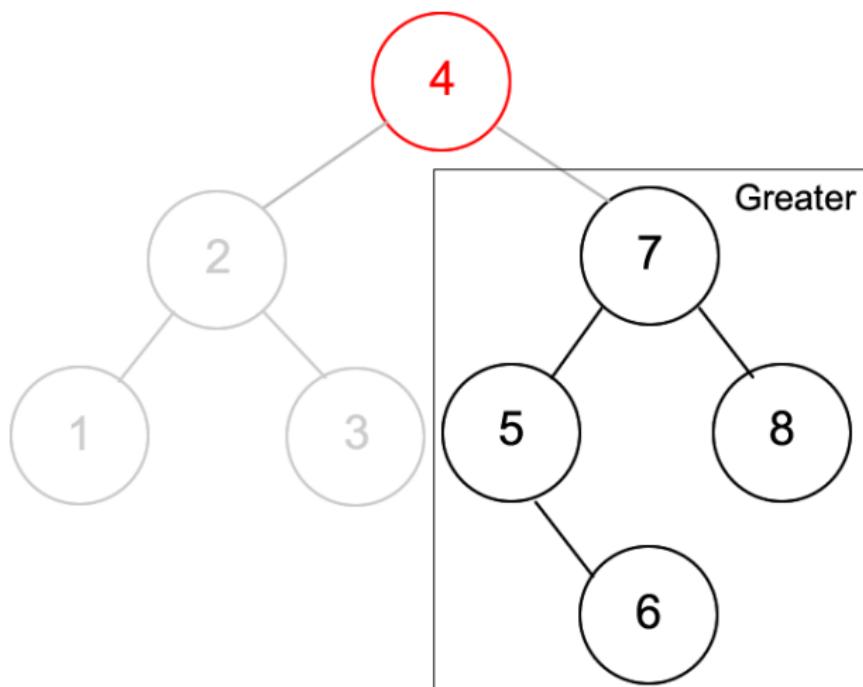
## Trees



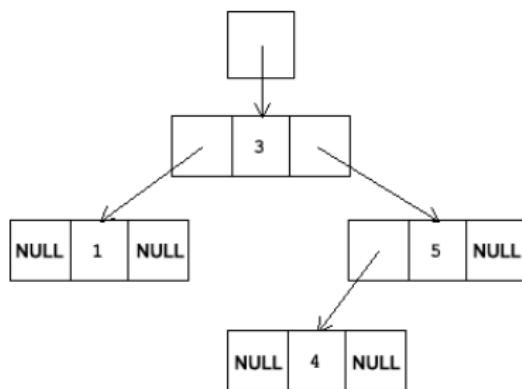
## Trees



## Trees



# Trees

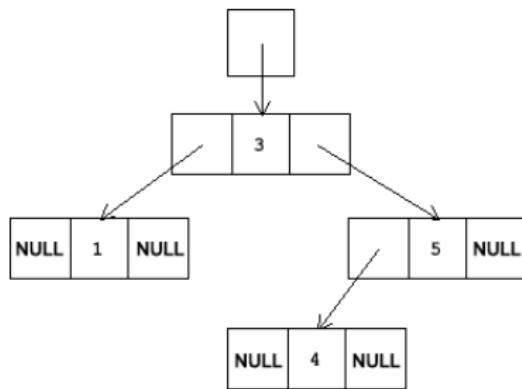


## Structure

Parts of the Node structure:

- The value to store

# Trees

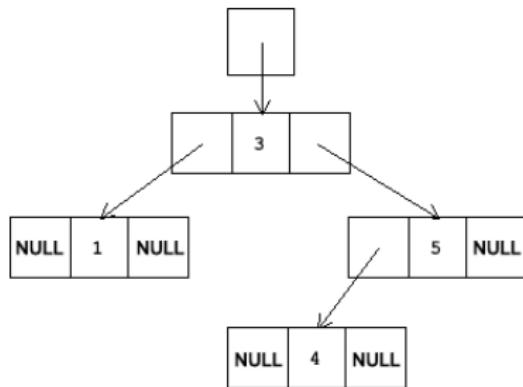


## Structure

Parts of the Node structure:

- The value to store
- left pointer to a Node

# Trees



## Structure

Parts of the Node structure:

- The value to store
- left pointer to a Node
- right pointer to a Node

# Trees

## Node (Leaf)



# Queue

## Operations

- **insert**

Input: A pointer to the tree, and the element to insert

Do: Inserts the element in the tree

# Queue

## Operations

- **insert**

Input: A pointer to the tree, and the element to insert

Do: Inserts the element in the tree

- **Traversing/Searchng an element**

Input: A pointer to the tree, and the element to search

Do: Transverse the tree searching the element

Return: True/False (If the element was found). Return (Alt):

A pointer to the node where the value is the searched one

# Queue

## Insert

- Let tree a pointer to the root of the tree, and value the value to insert

# Queue

## Insert

- Let tree a pointer to the root of the tree, and value the value to insert
- Check if the input node is null

# Queue

## Insert

- Let tree a pointer to the root of the tree, and value the value to insert
- Check if the input node is null
- If so, create a new Node with the input value, and point the input tree to the created Node

# Queue

## Insert

- Let tree a pointer to the root of the tree, and value the value to insert
- Check if the input node is null
- If so, create a new Node with the input value, and point the input tree to the created Node
- Else. Check if the value to insert is lower than the value of the Node. If so, call the insert function with the left subtree

# Queue

## Insert

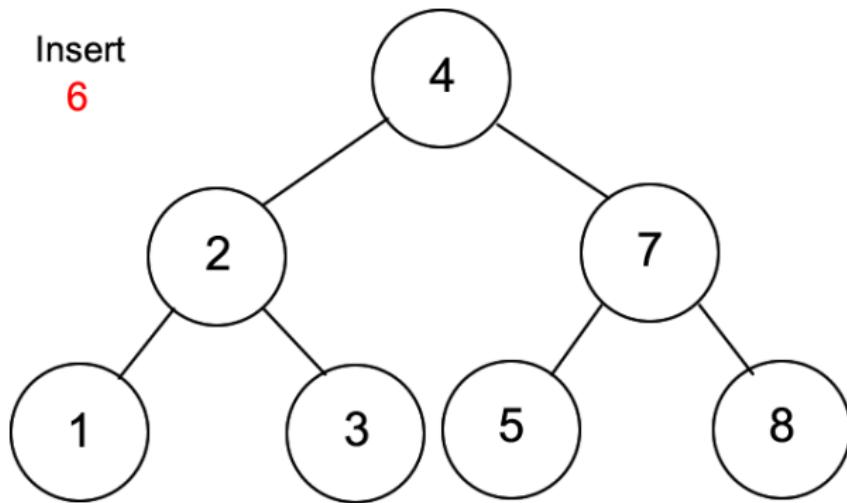
- Let tree a pointer to the root of the tree, and value the value to insert
- Check if the input node is null
- If so, create a new Node with the input value, and point the input tree to the created Node
- Else. Check if the value to insert is lower than the value of the Node. If so, call the insert function with the left subtree
- Else if the value to insert is greater than the value of the Node, call the insert function with the right subtree

# Queue

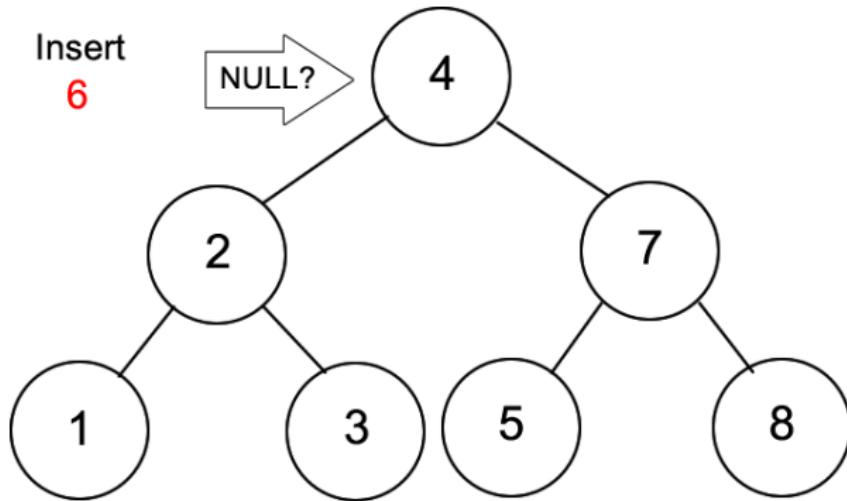
## Insert

- Let tree a pointer to the root of the tree, and value the value to insert
- Check if the input node is null
- If so, create a new Node with the input value, and point the input tree to the created Node
- Else. Check if the value to insert is lower than the value of the Node. If so, call the insert function with the left subtree
- Else if the value to insert is greater than the value of the Node, call the insert function with the right subtree
- If the value to insert is equal to the value of the Node, discard the value and stop the algorithm

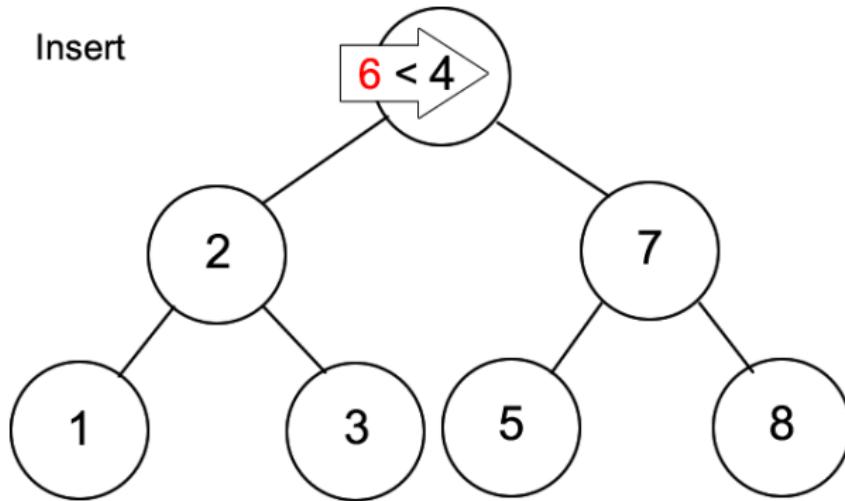
# Binary Search Tree - Insert



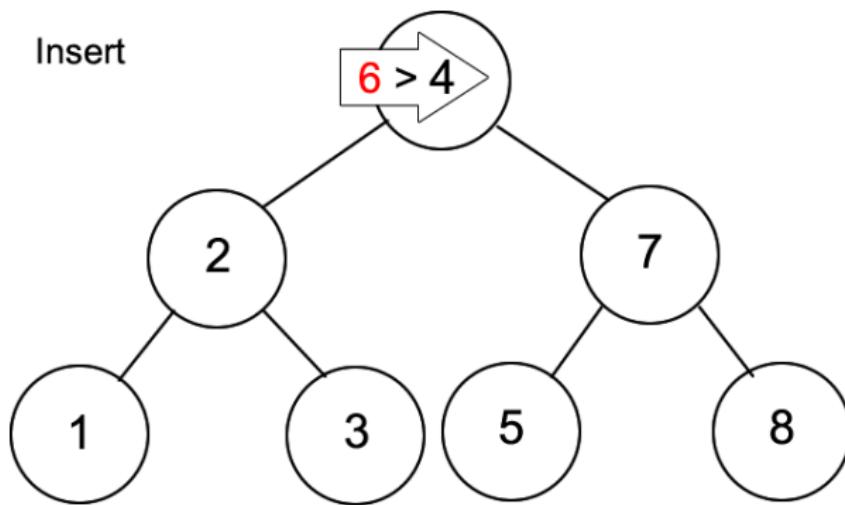
## Binary Search Tree - Insert



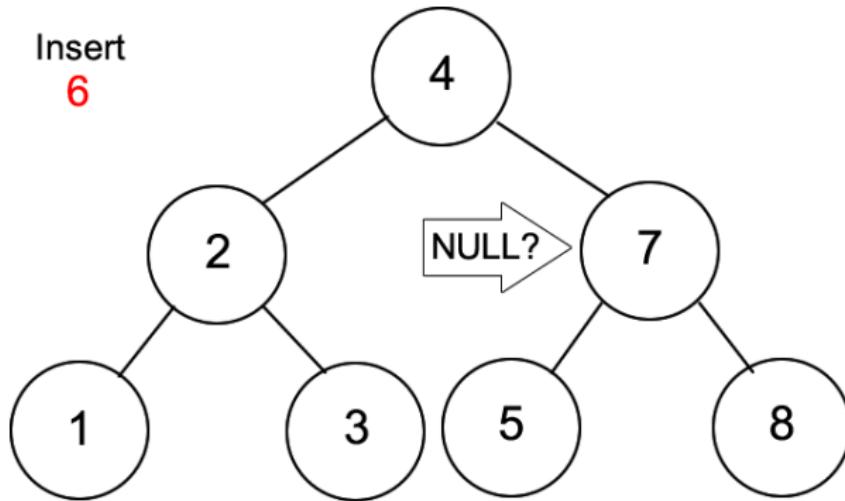
## Binary Search Tree - Insert



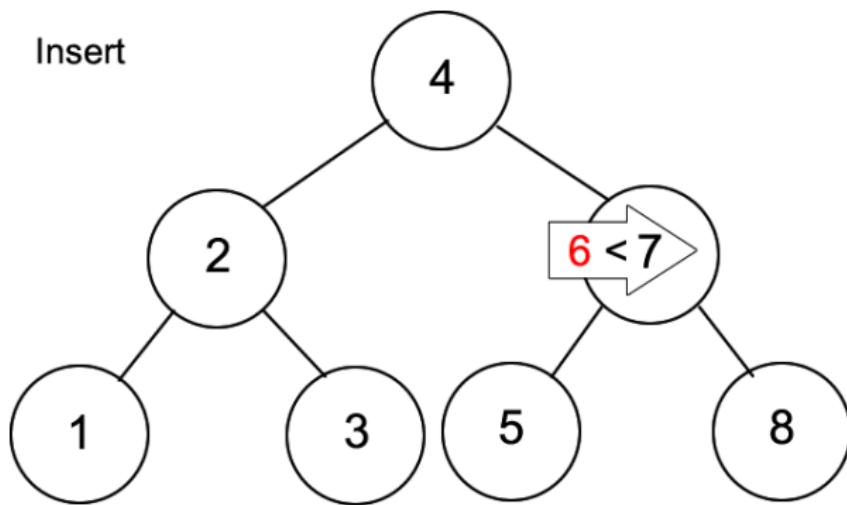
## Binary Search Tree - Insert



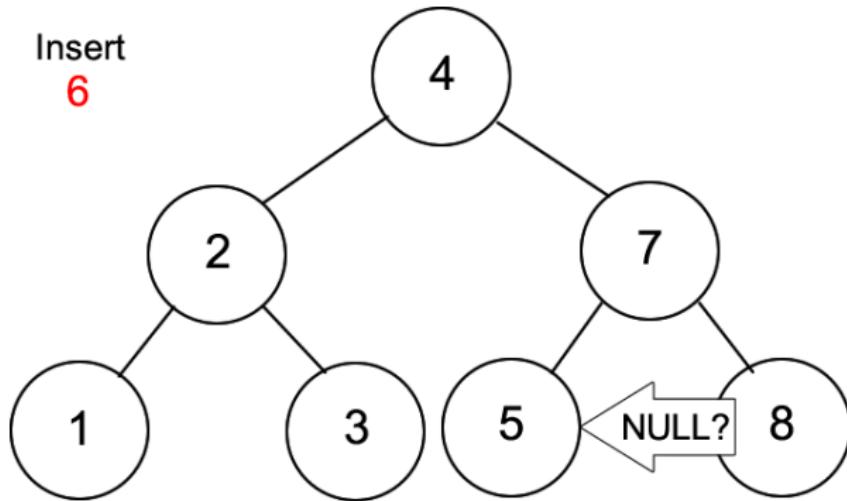
## Binary Search Tree - Insert



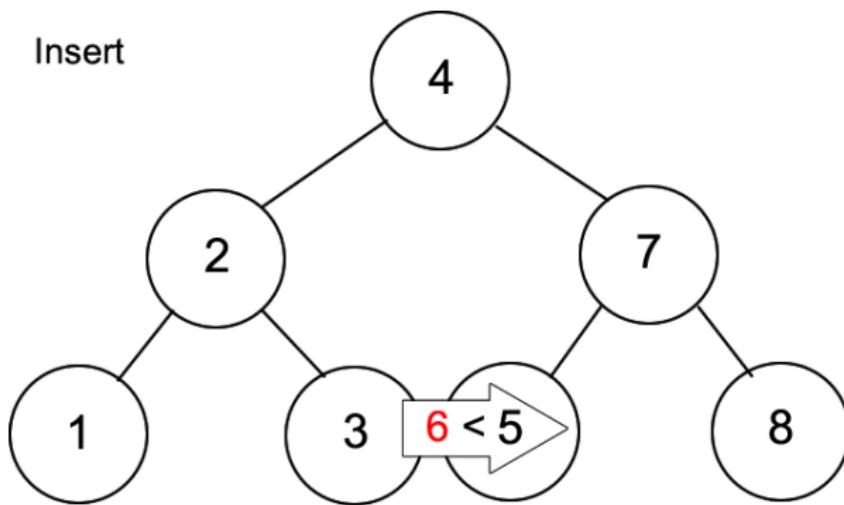
## Binary Search Tree - Insert



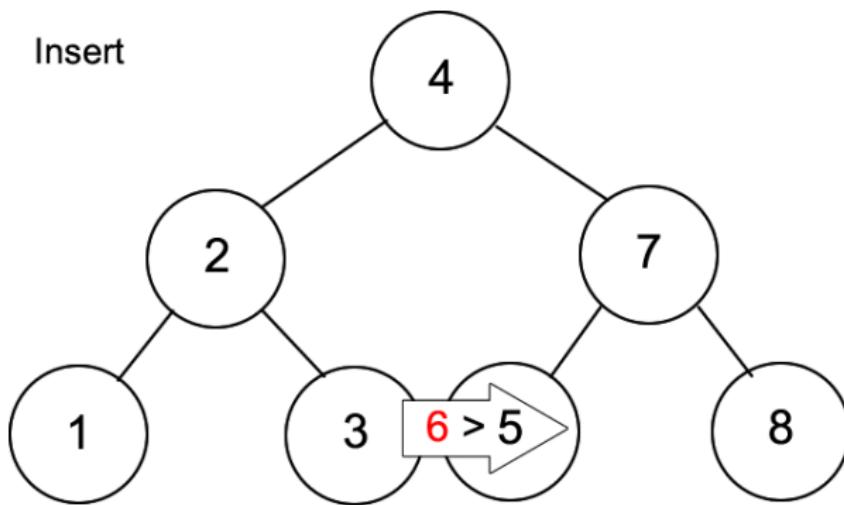
## Binary Search Tree - Insert



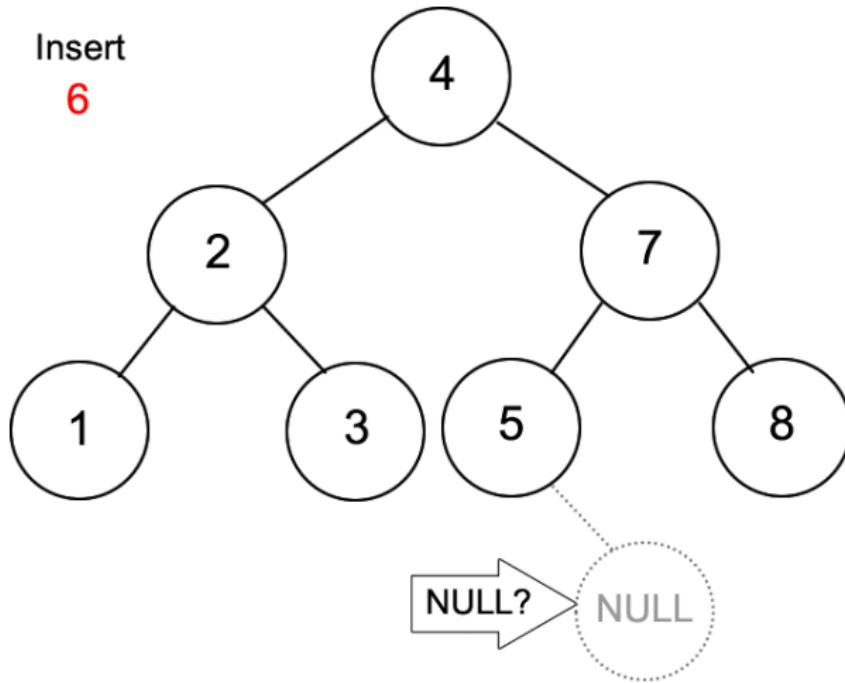
## Binary Search Tree - Insert



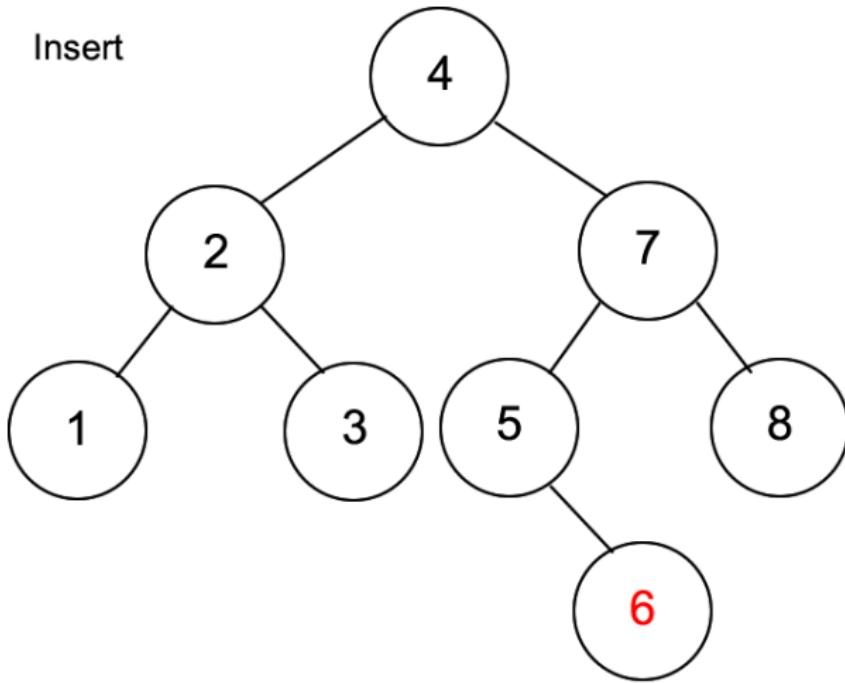
## Binary Search Tree - Insert



## Binary Search Tree - Insert



## Binary Search Tree - Insert



# Trees - Traversing

## Depth-first search

Is an algorithm for traversing a tree. Exploration starts at the root node and made a depth exploration as far as possible before *backtracking*. Algorithm vary according the exploration order:

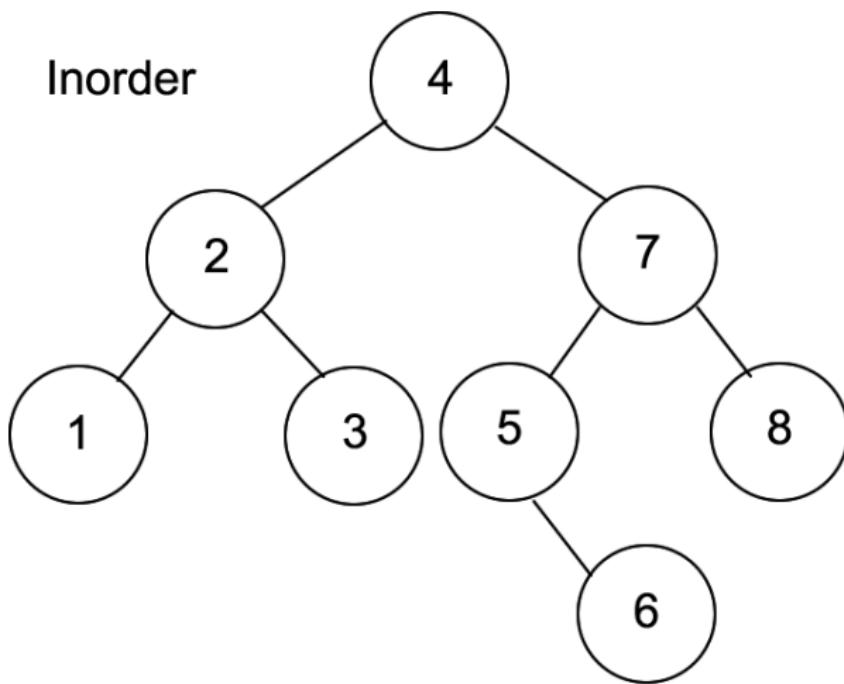
- In-order
- Pre-order
- Post-order

# Trees - InOrder

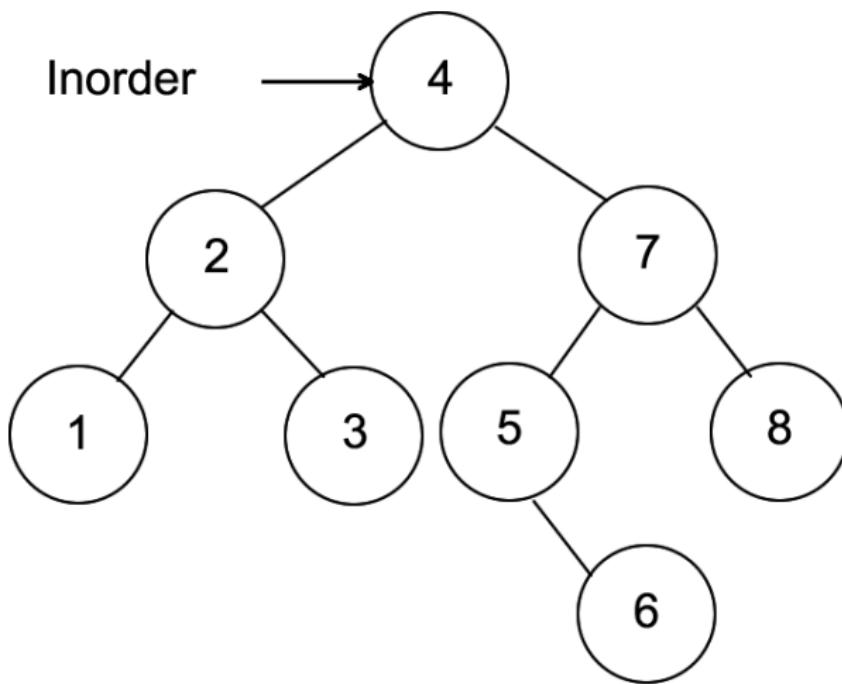
## Traversing a tree In-order

- Check if the node is null
- Traverse the left subtree InOrder
- Display the data at the node
- Traverse the right subtree InOrder

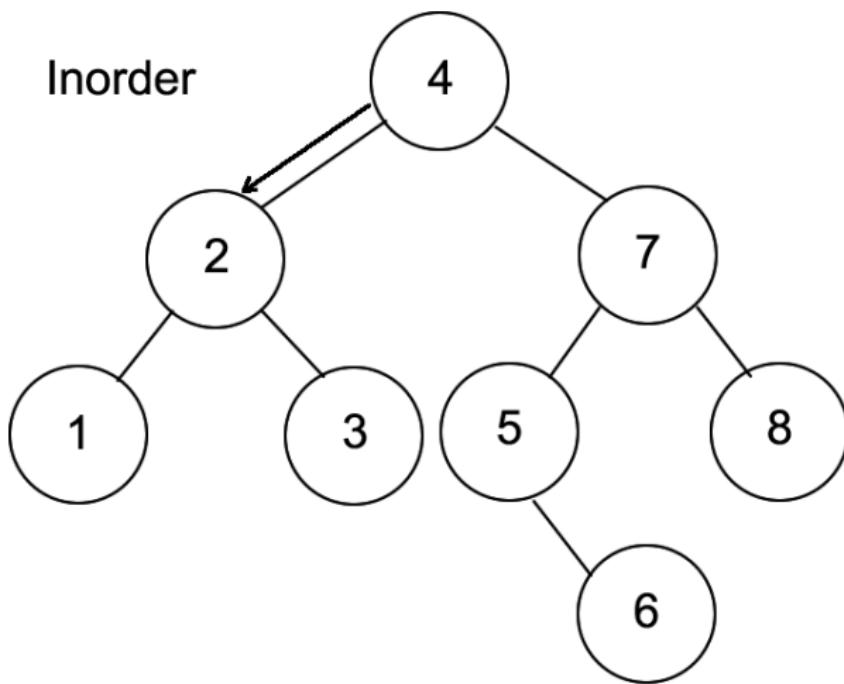
## Trees - InOrder



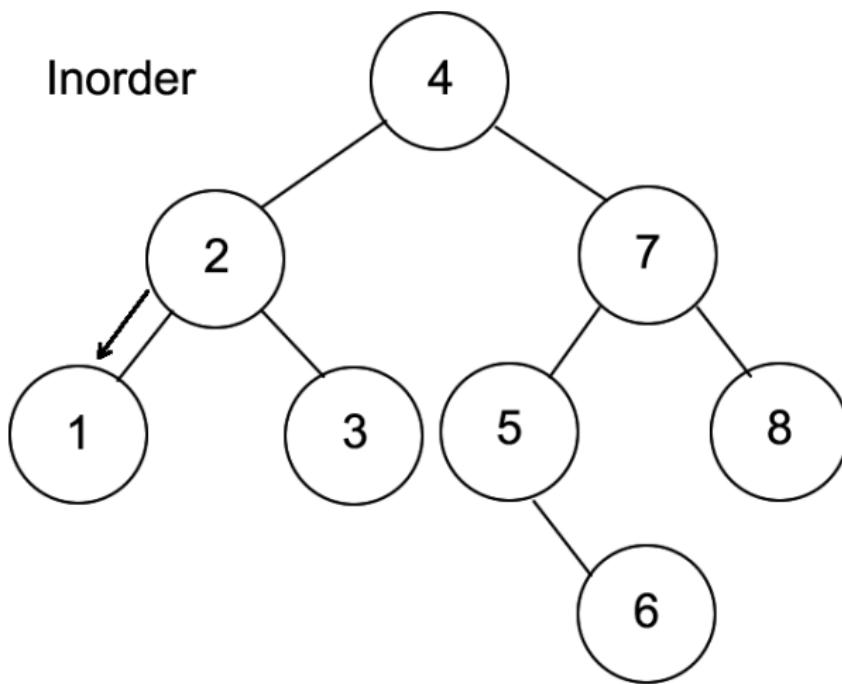
## Trees - InOrder



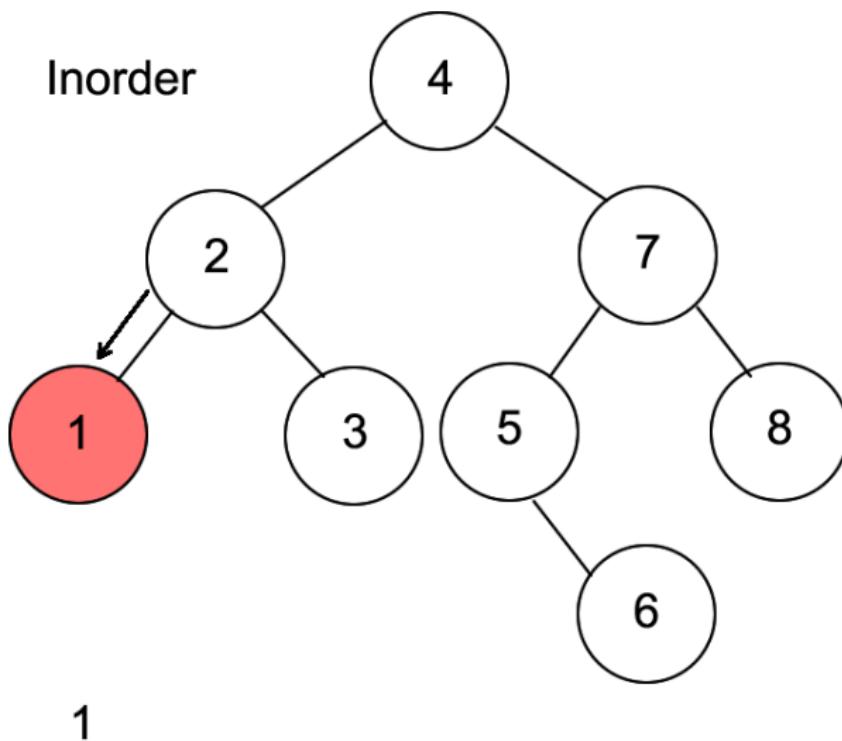
## Trees - InOrder



## Trees - InOrder

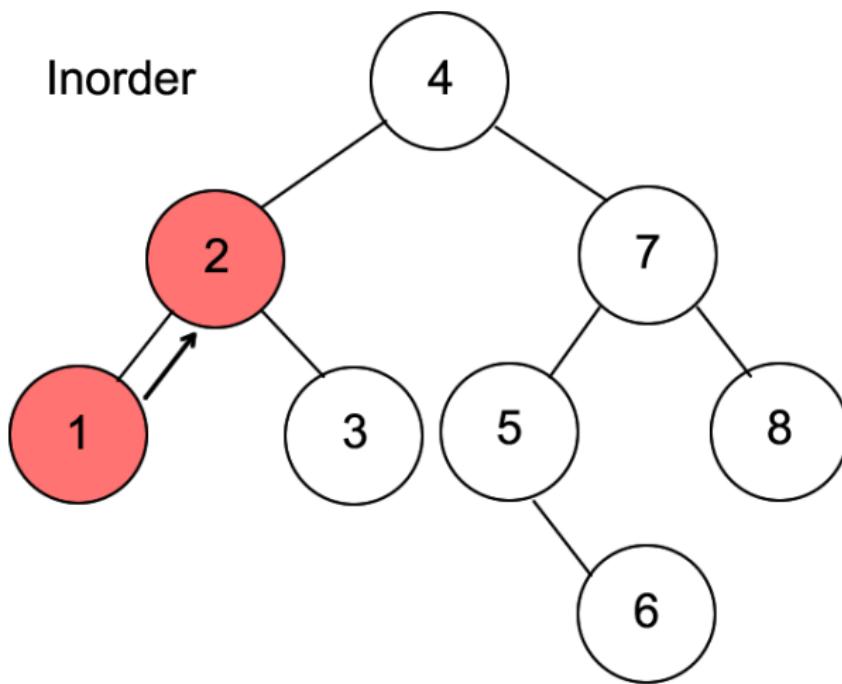


## Trees - InOrder



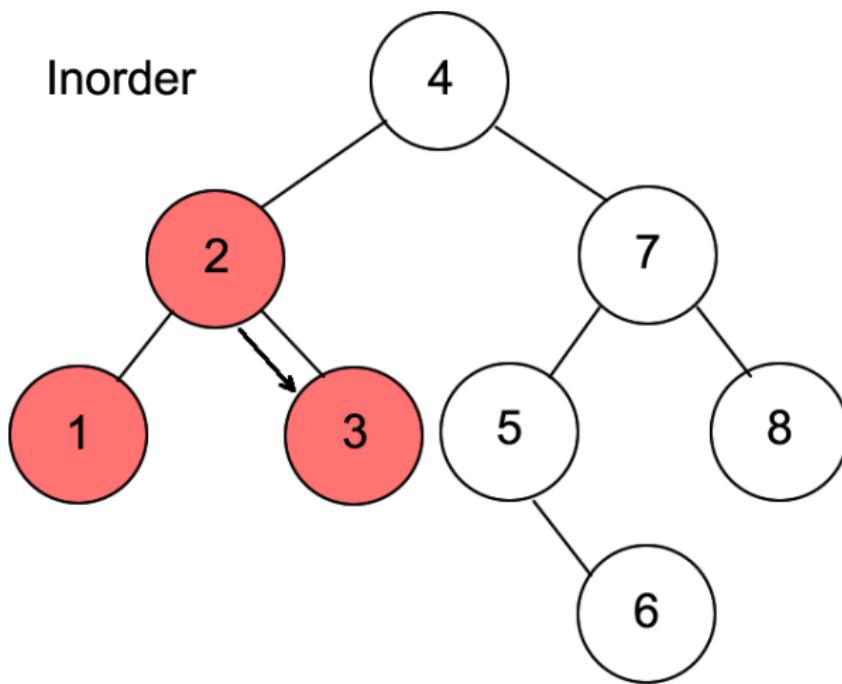
1

## Trees - InOrder



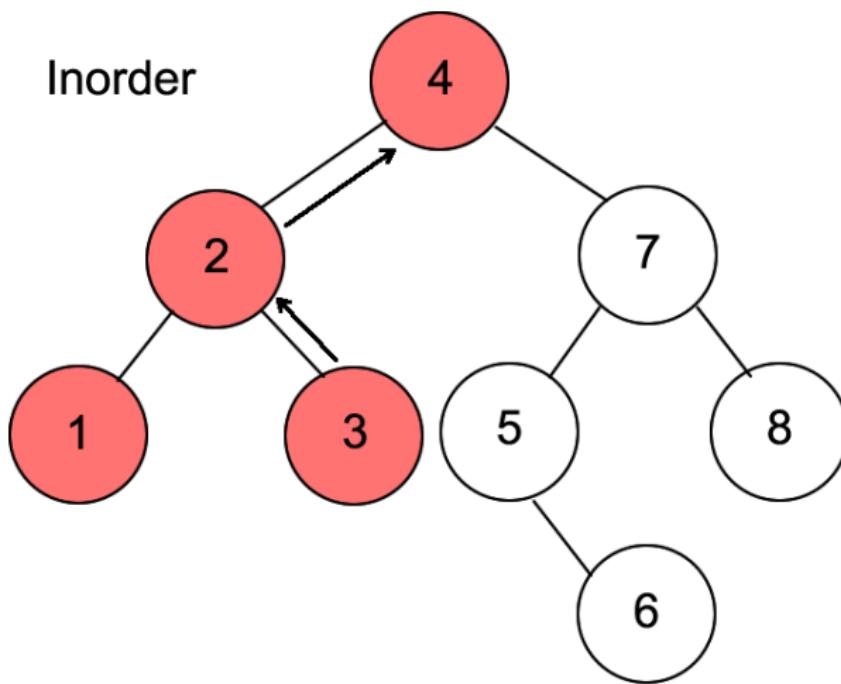
1 - 2

## Trees - InOrder



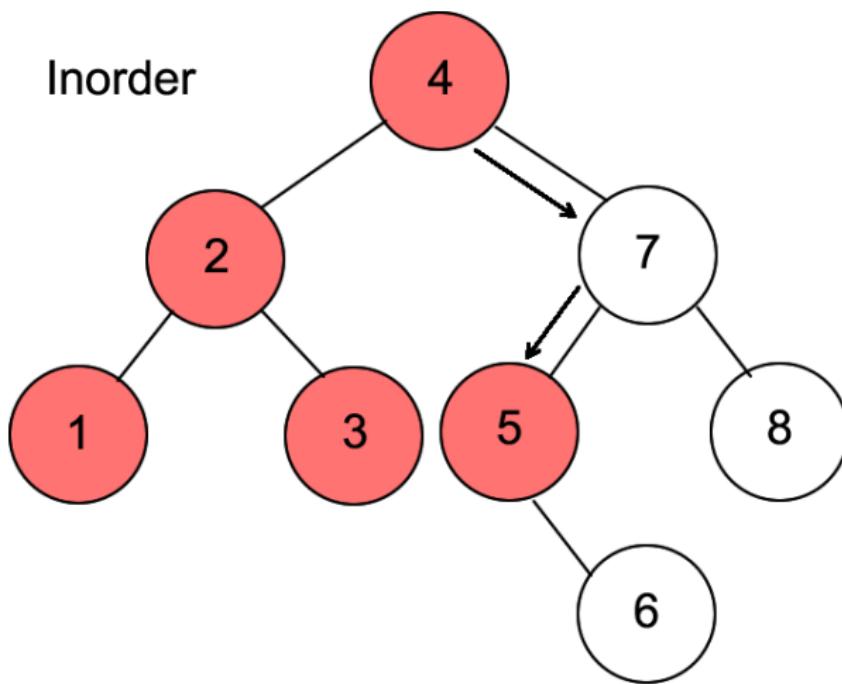
1 - 2 - 3

## Trees - InOrder



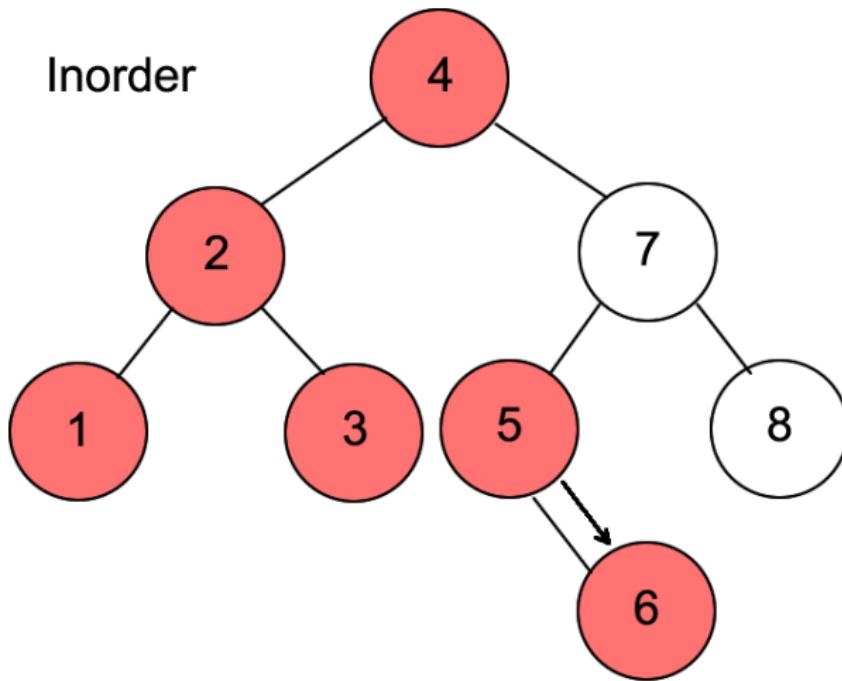
1 - 2 - 3 - 4

## Trees - InOrder



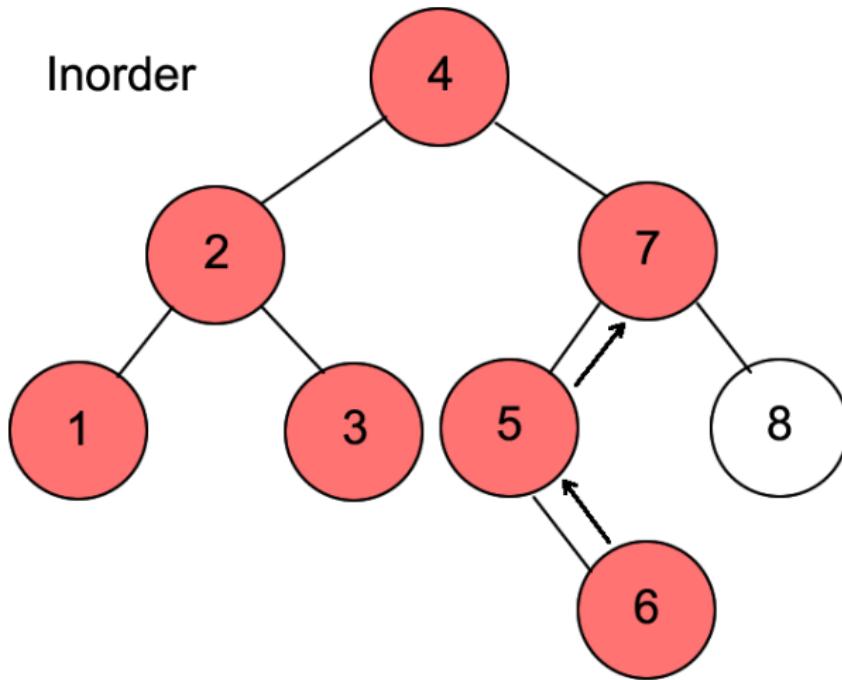
1 - 2 - 3 - 4 - 5

## Trees - InOrder



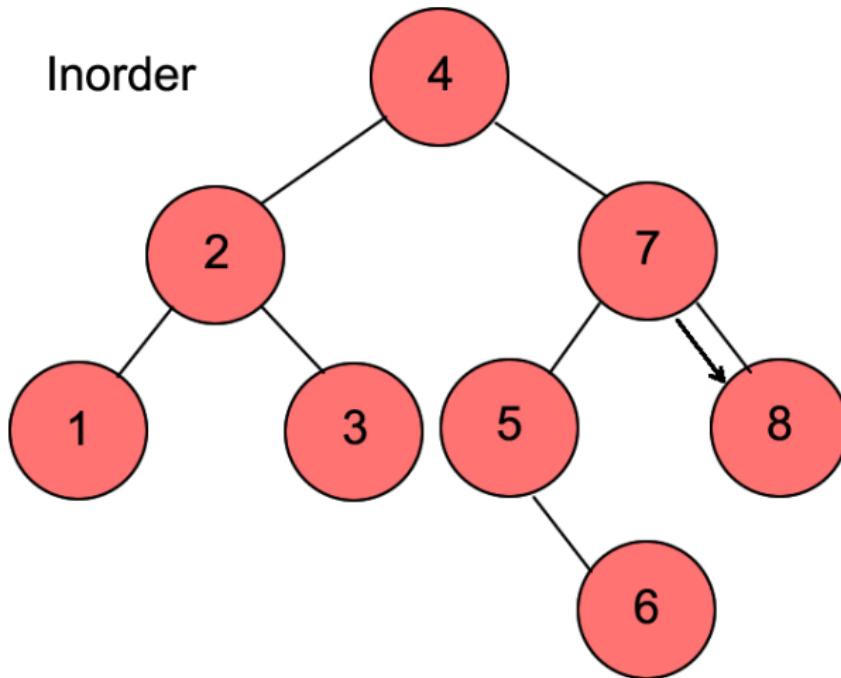
1 - 2 - 3 - 4 - 5 - 6

## Trees - InOrder



1 - 2 - 3 - 4 - 5 - 6 - 7

## Trees - InOrder



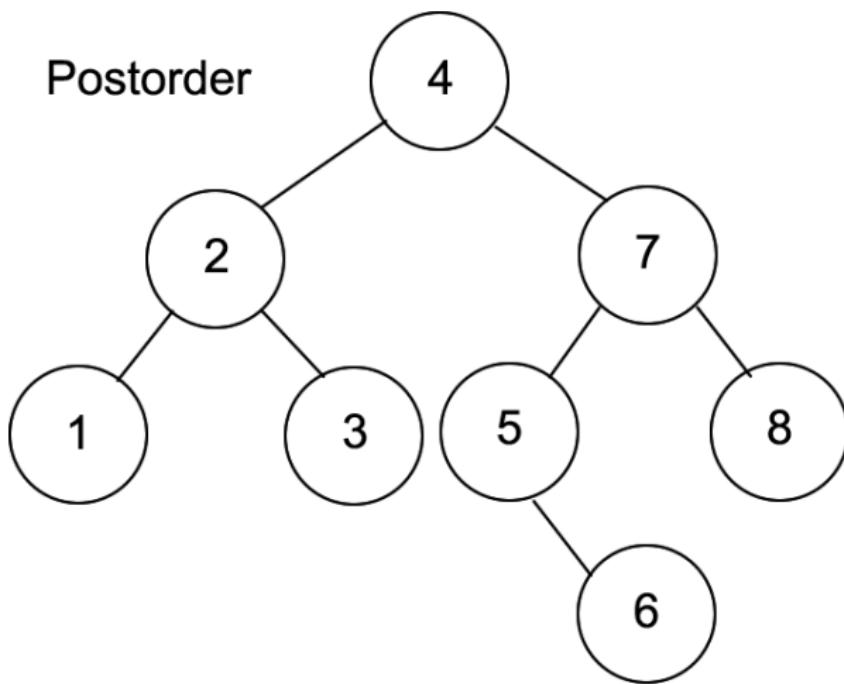
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8

# Trees - PostOrder

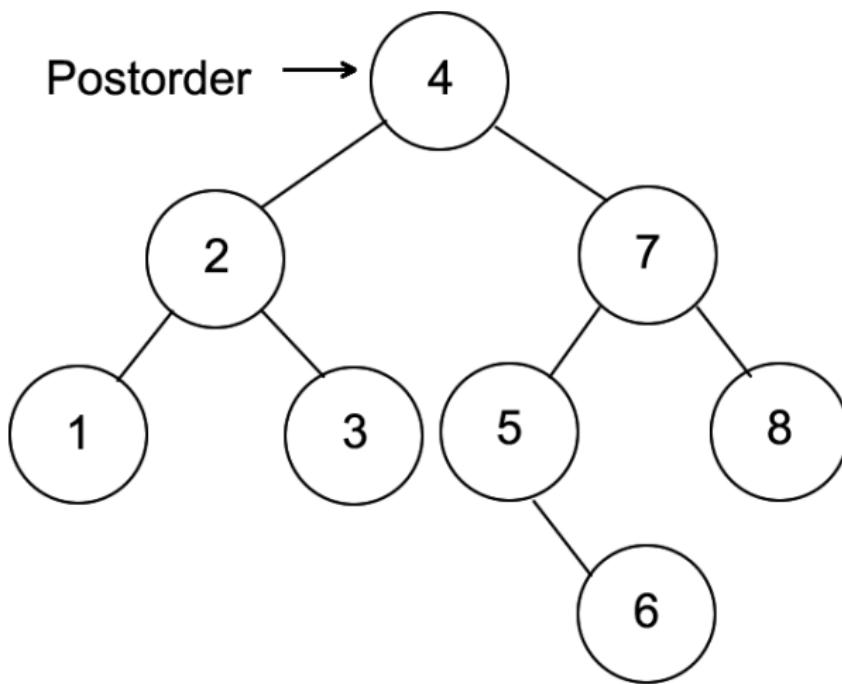
## Traversing a tree Post-order

- Check if the node is null
- Traverse the left subtree PostOrder
- Traverse the right subtree PostOrder
- Display the data at the node

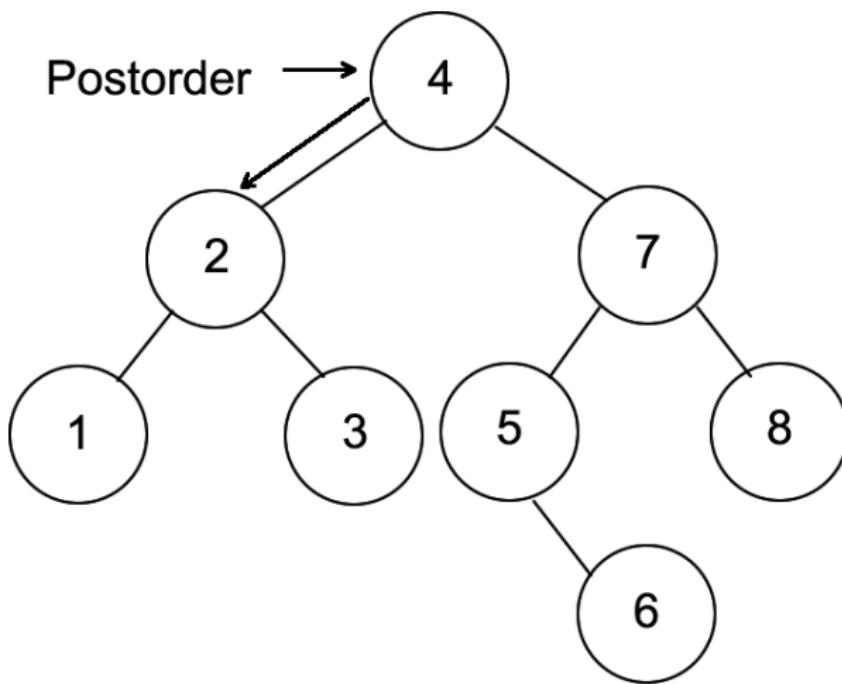
## Trees - PostOrder



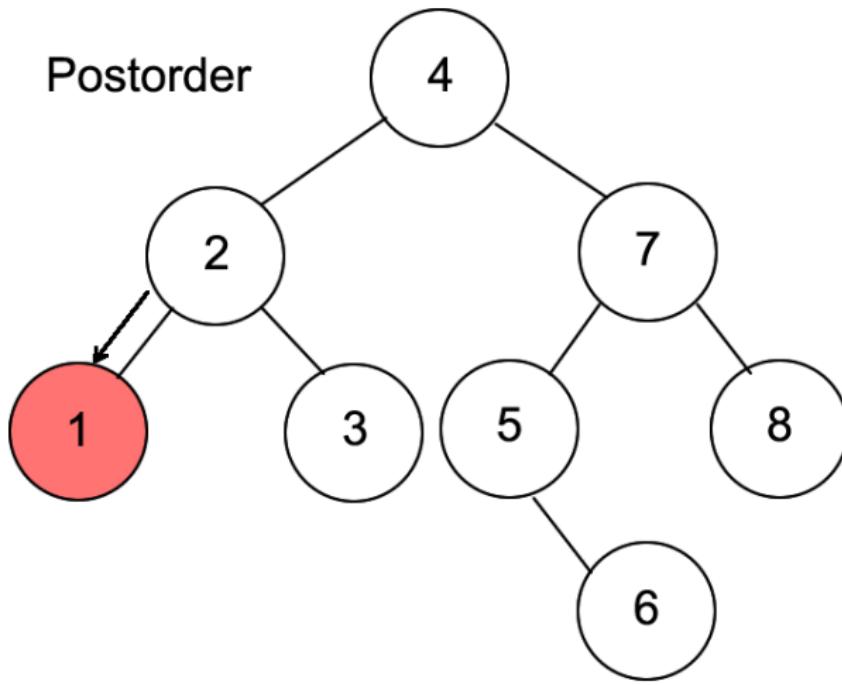
## Trees - PostOrder



## Trees - PostOrder

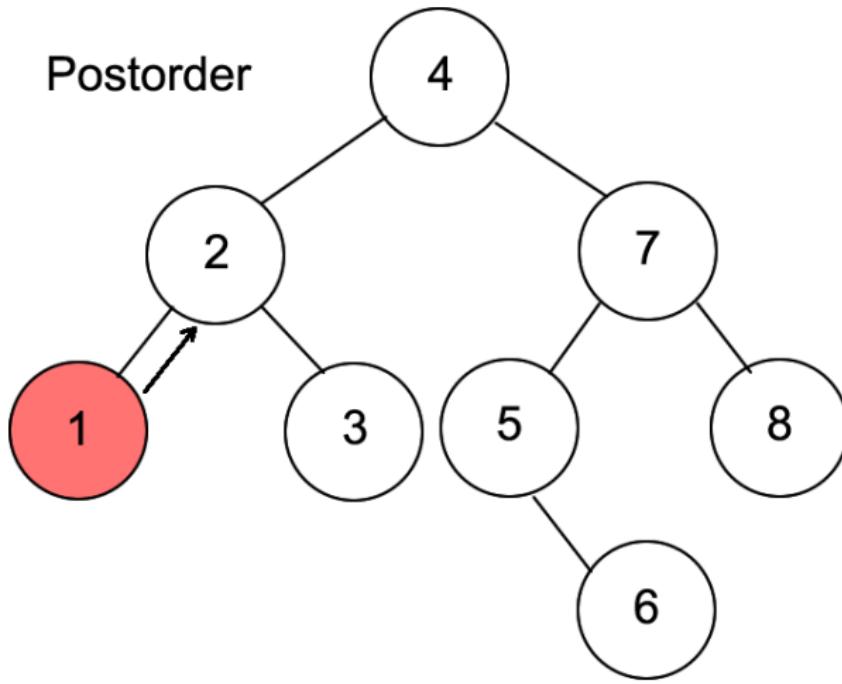


## Trees - PostOrder



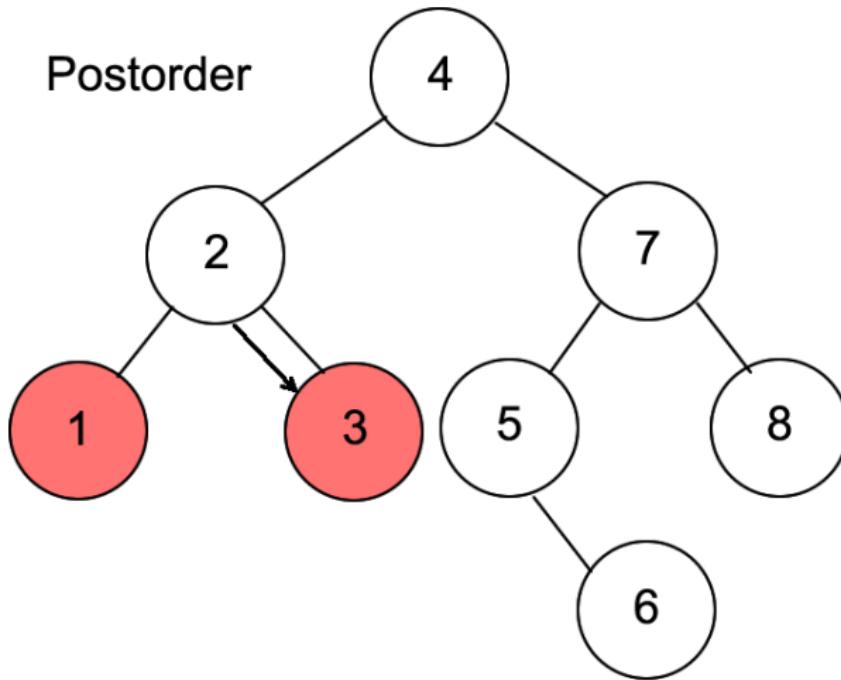
1

## Trees - PostOrder



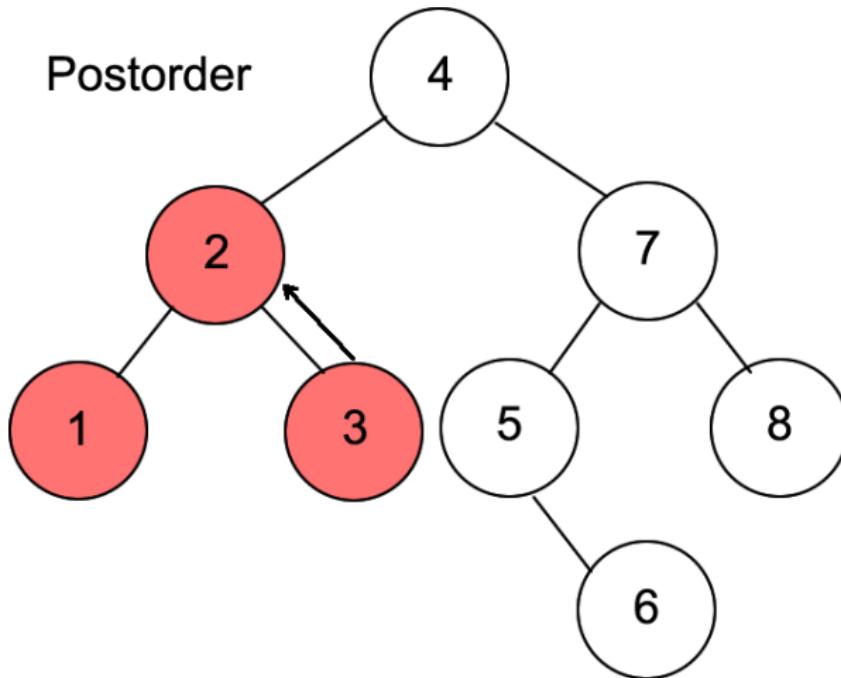
1

## Trees - PostOrder



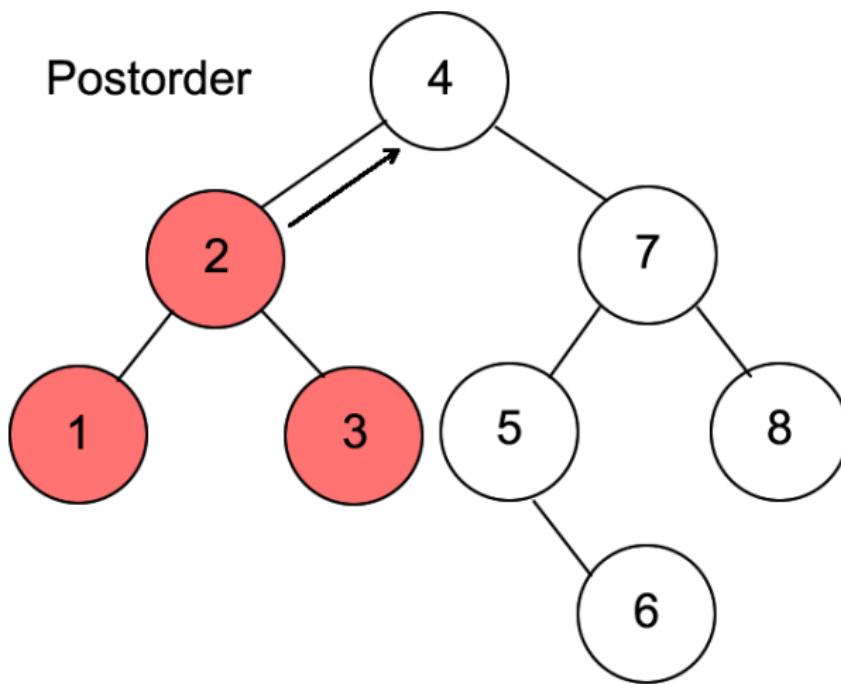
1 - 3

## Trees - PostOrder



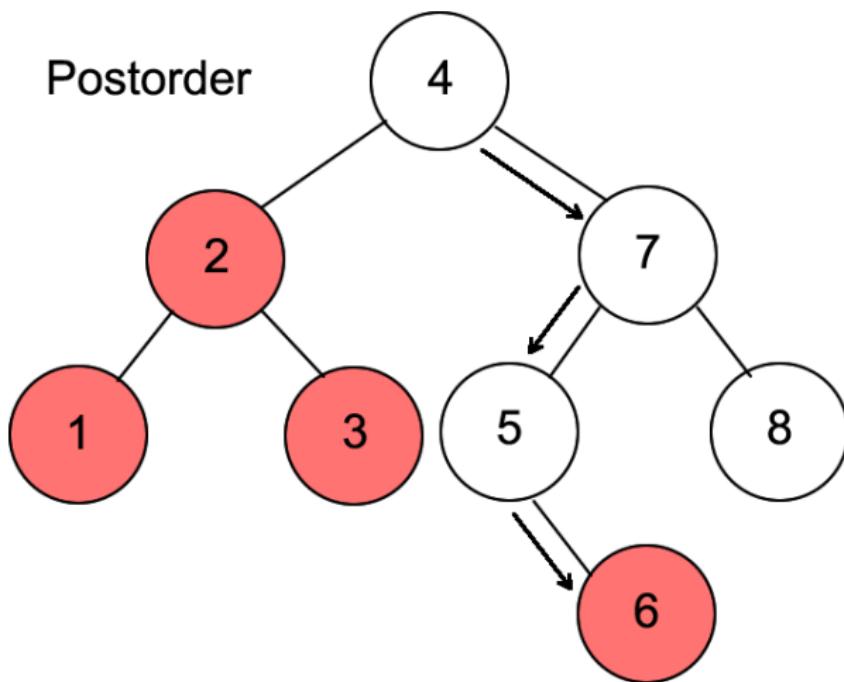
1 - 3 - 2

## Trees - PostOrder



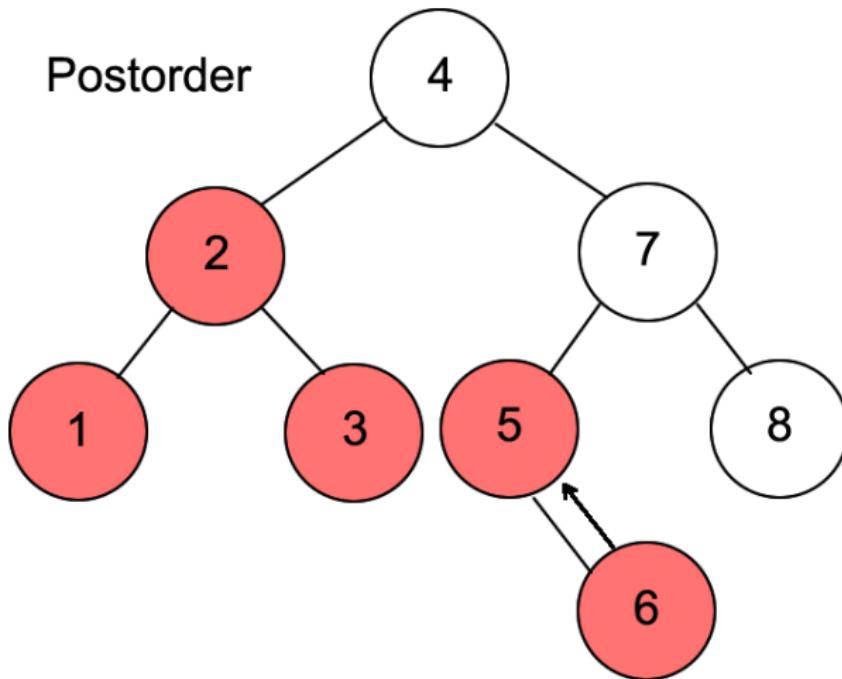
1 - 3 - 2

## Trees - PostOrder



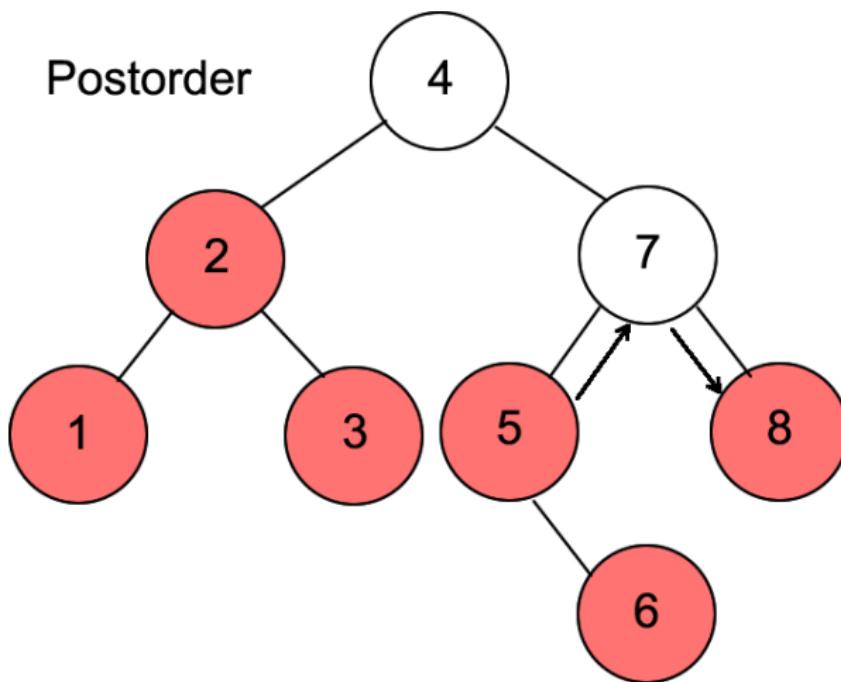
1 - 3 - 2 - 6

## Trees - PostOrder



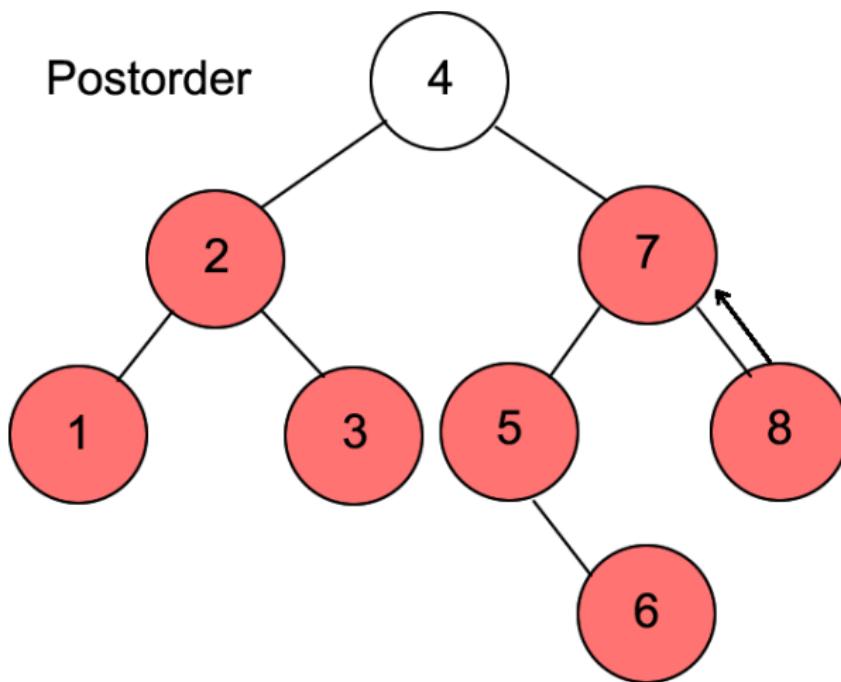
1 - 3 - 2 - 6 - 5

## Trees - PostOrder



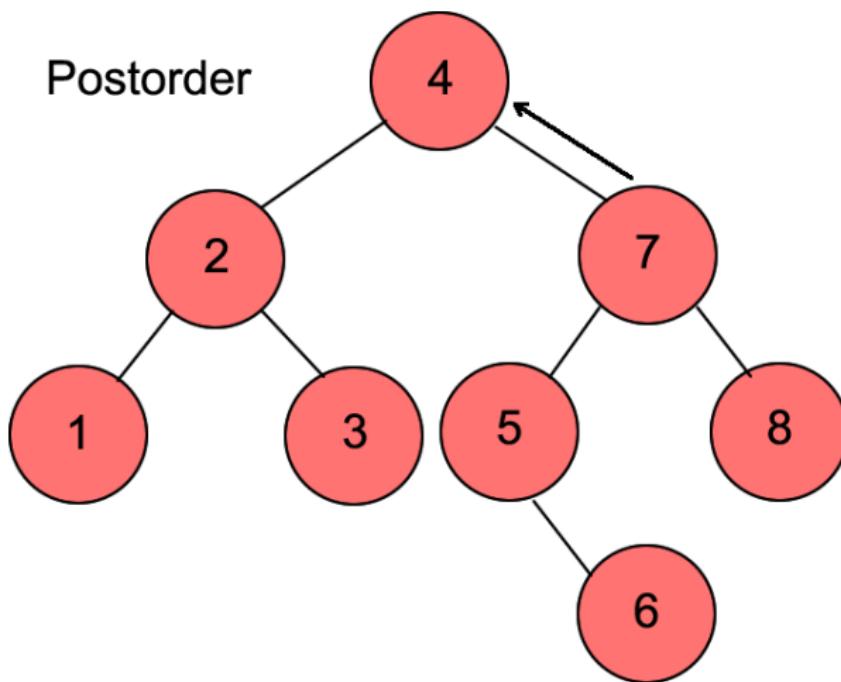
1 - 3 - 2 - 6 - 5 - 8

## Trees - PostOrder



1 - 3 - 2 - 6 - 5 - 8 - 7

## Trees - PostOrder



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

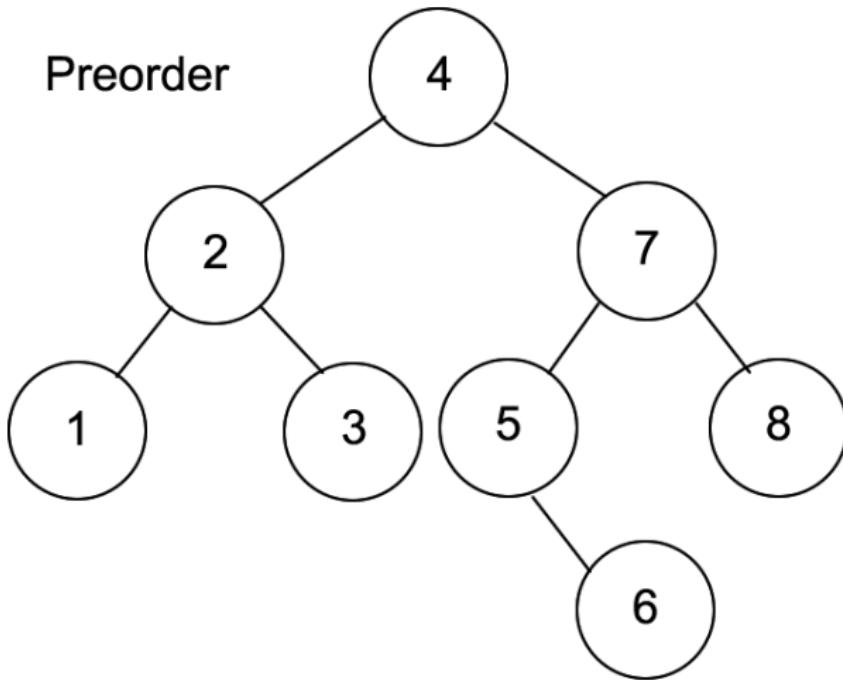
# Trees - PreOrder

## Traversing a tree Pre-order

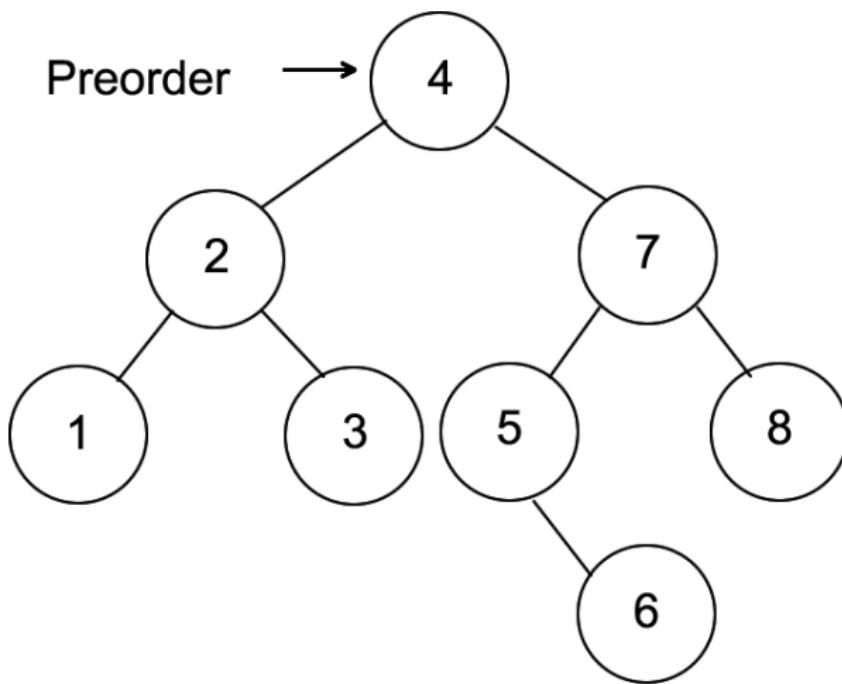
- Check if the node is null
- Display the data at the node
- Traverse the left subtree PreOrder
- Traverse the right subtree PreOrder

## Trees - PreOrder

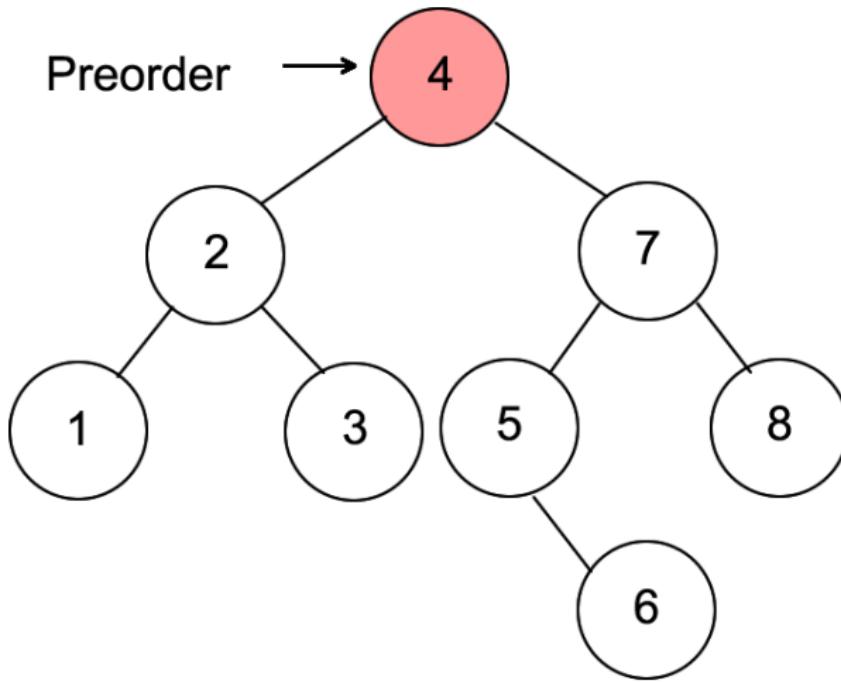
Preorder



## Trees - PreOrder

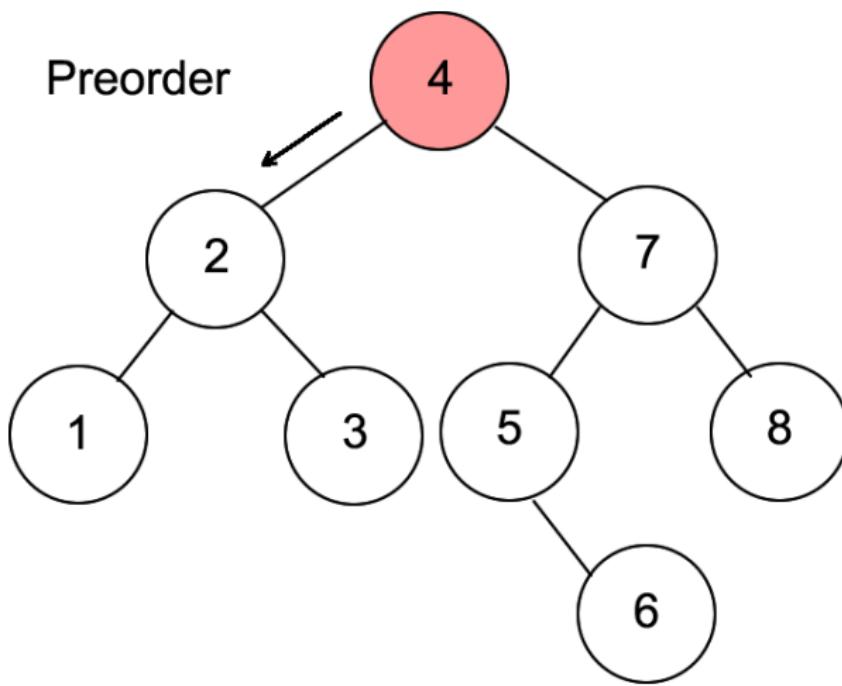


## Trees - PreOrder



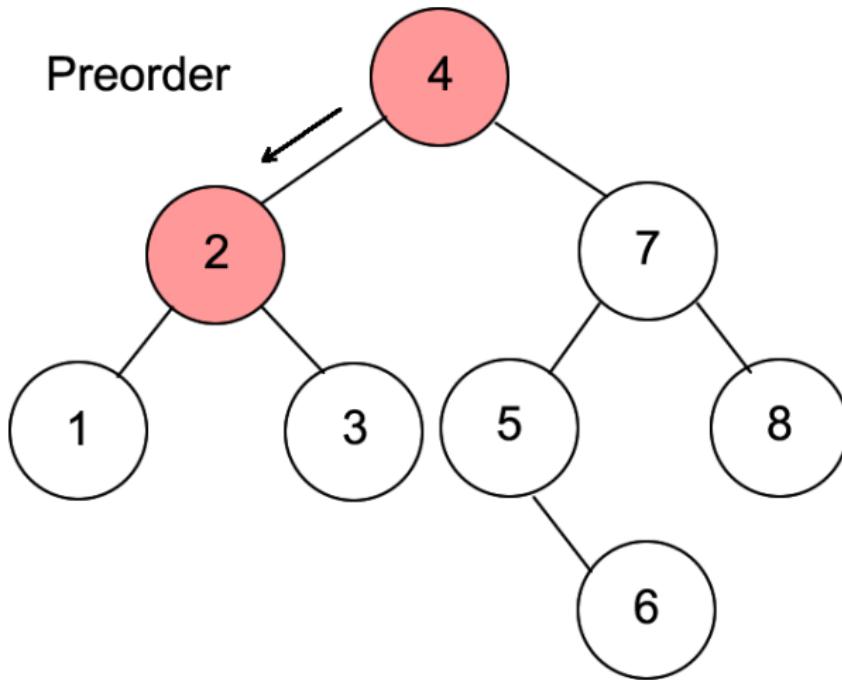
4

## Trees - PreOrder



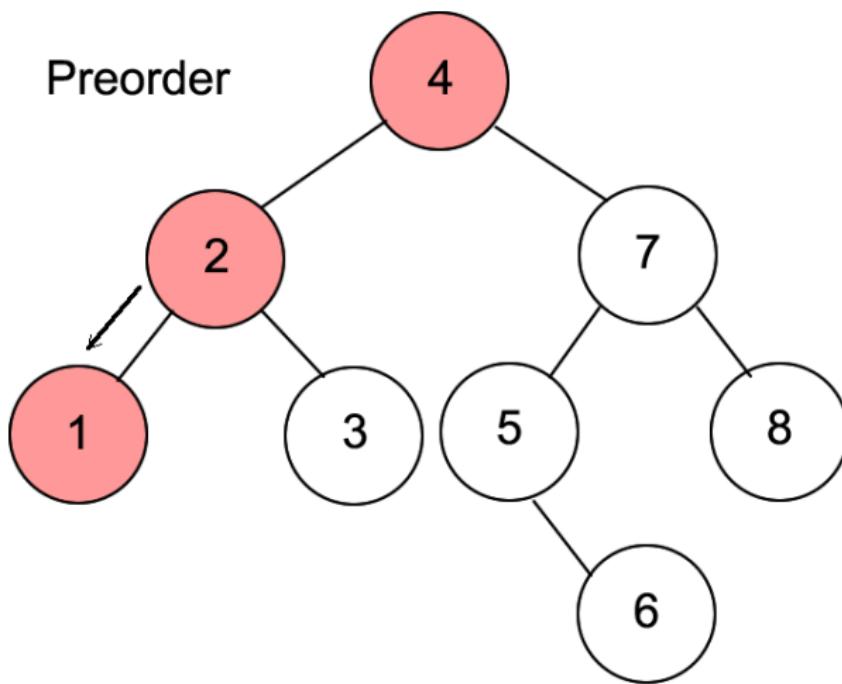
4

## Trees - PreOrder



4 - 2

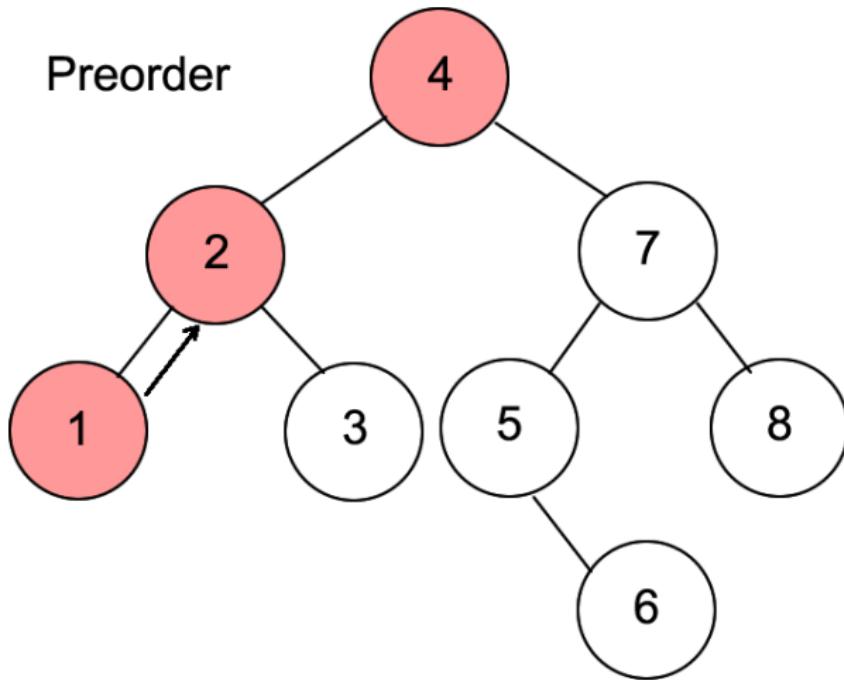
## Trees - PreOrder



4 - 2 - 1

## Trees - PreOrder

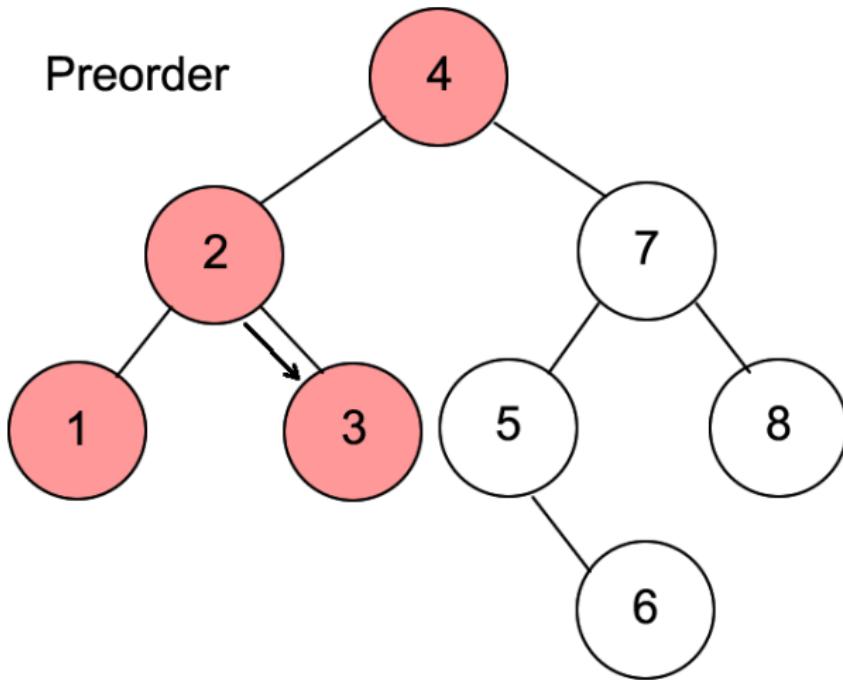
Preorder



4 - 2 - 1

## Trees - PreOrder

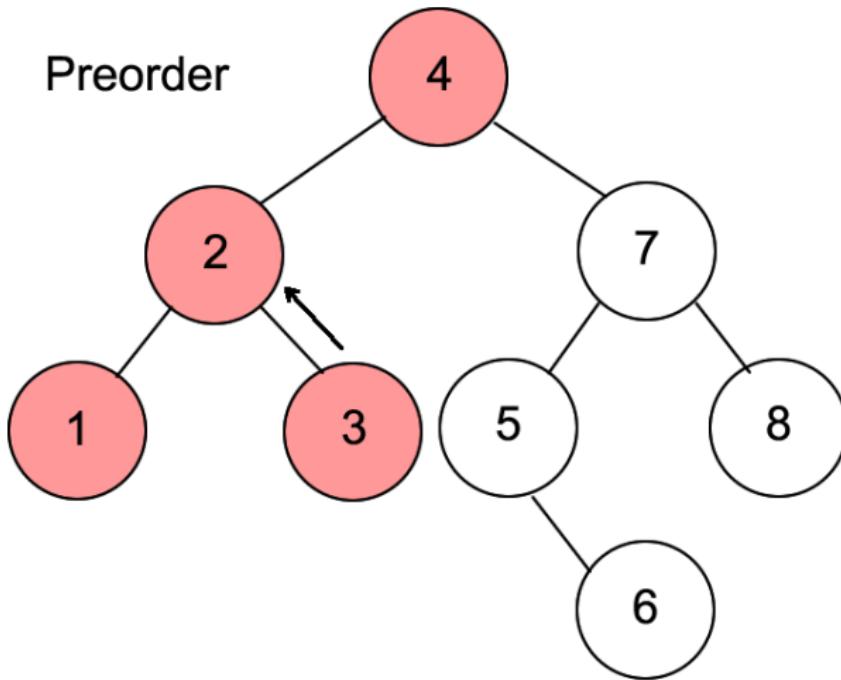
Preorder



4 - 2 - 1 - 3

## Trees - PreOrder

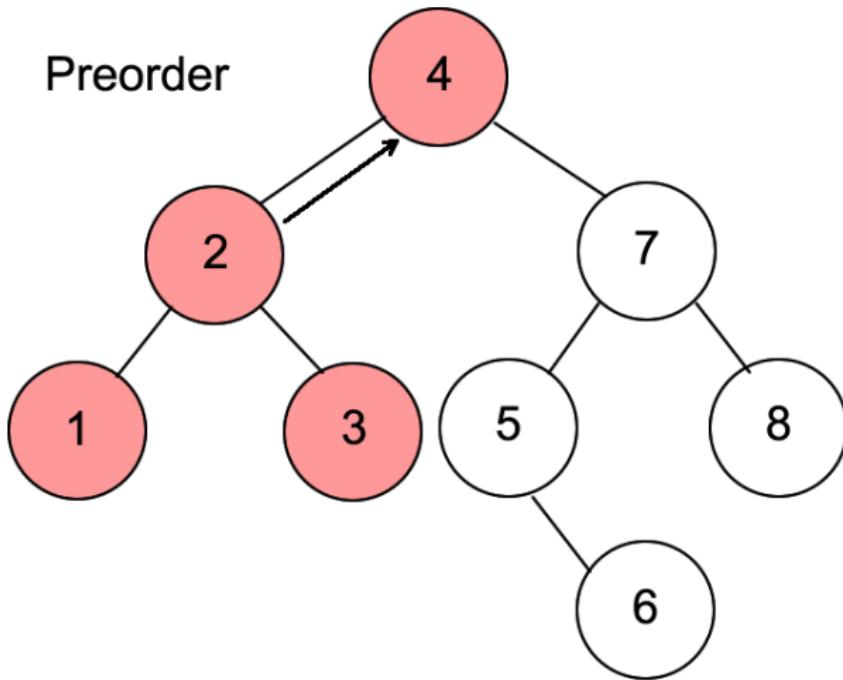
Preorder



4 - 2 - 1 - 3

## Trees - PreOrder

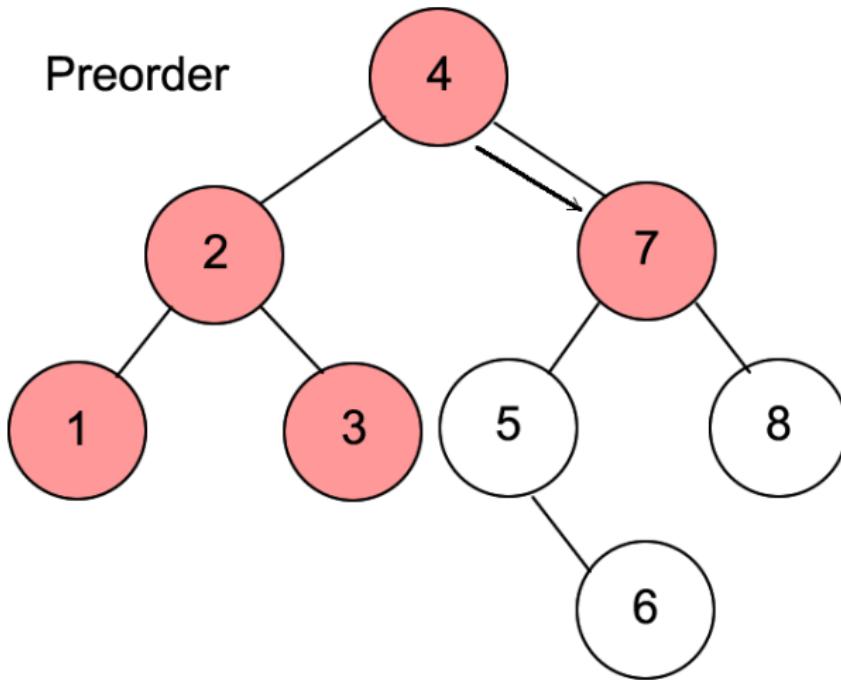
Preorder



4 - 2 - 1 - 3

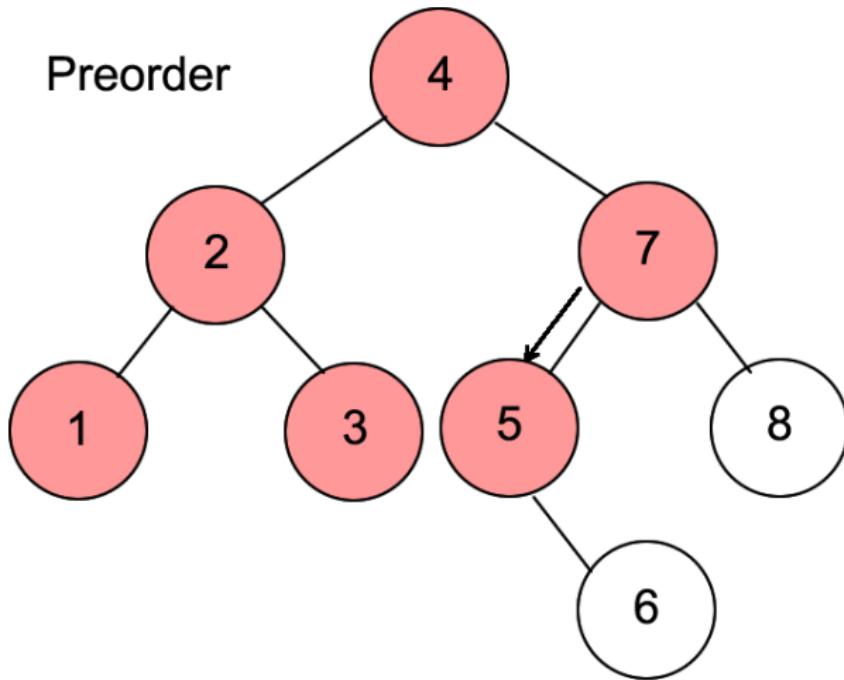
## Trees - PreOrder

Preorder



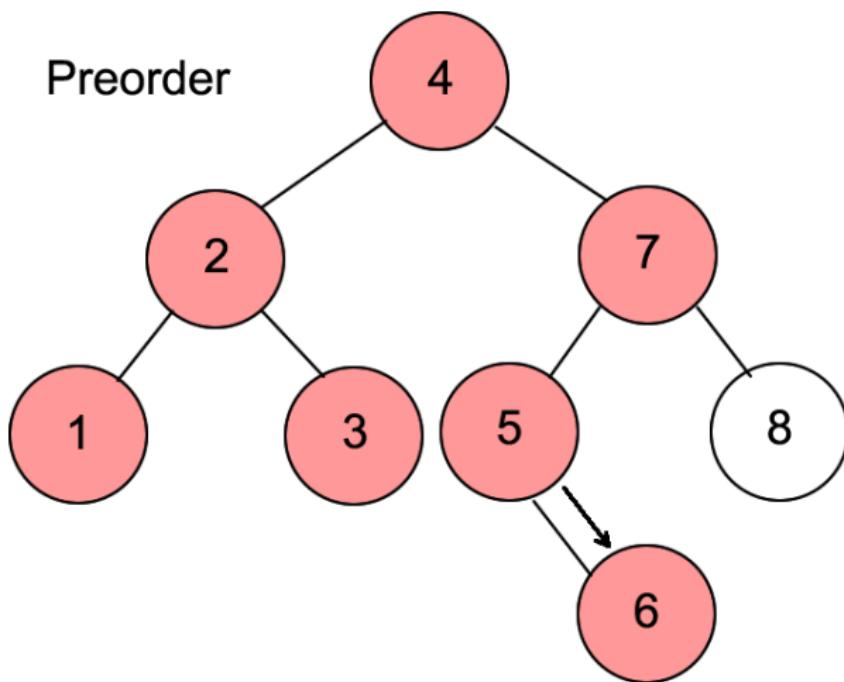
4 - 2 - 1 - 3 - 7

## Trees - PreOrder



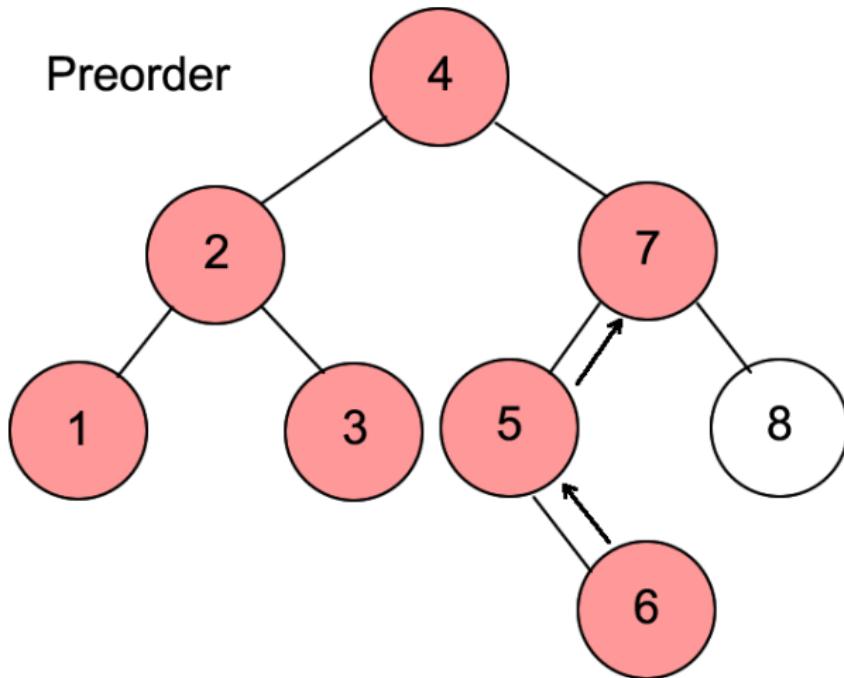
4 - 2 - 1 - 3 - 7 - 5

## Trees - PreOrder



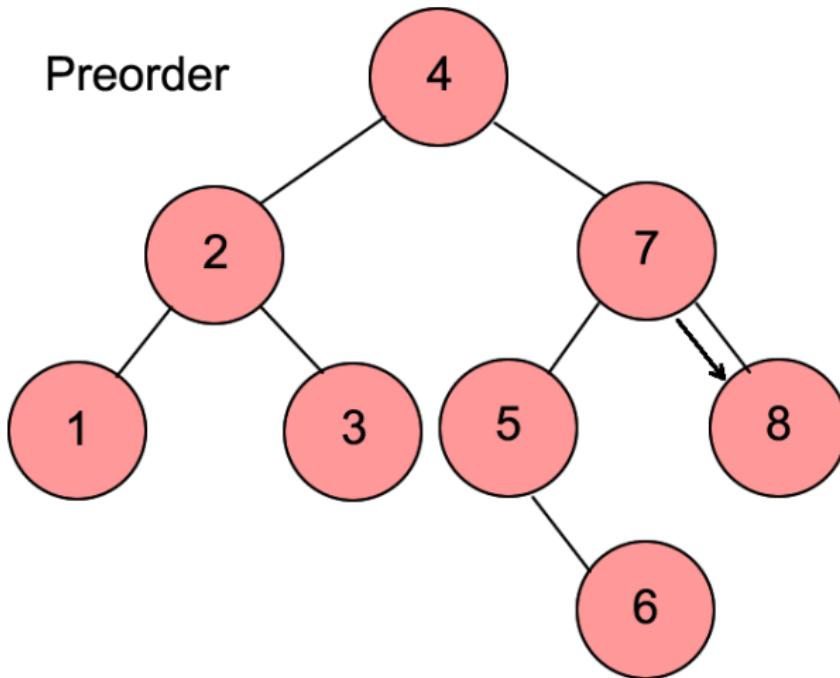
4 - 2 - 1 - 3 - 7 - 5 - 6

## Trees - PreOrder



4 - 2 - 1 - 3 - 7 - 5 - 6

## Trees - PreOrder



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