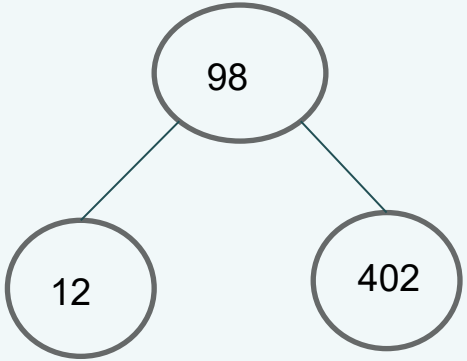
# C- Binary Trees

Leen Alsaleh

# What is Binary Trees

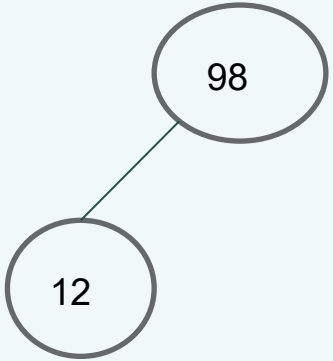
```
typdef struct binary_tree_s
{
Int n ;
struct binary_tree_s *parent;
struct binary_tree_s *left
struct binary_tree_s *right
}
binary_tree_t
```



# New Node

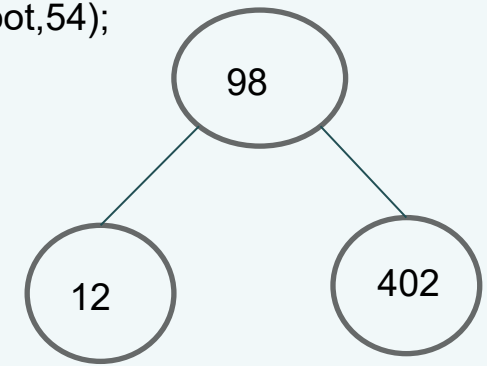
```
binary_tree_t  *binary_tree_node(binary_tree_t *parent, int value);
```

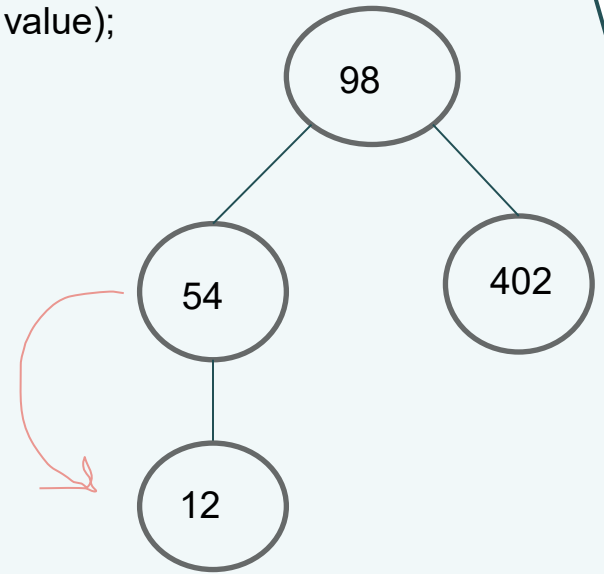
```
binary_tree_t *root ;

root = binary_tree_node (NULL,98);


root ->left = binary_tree_node(root,12);
```



# Insert Left

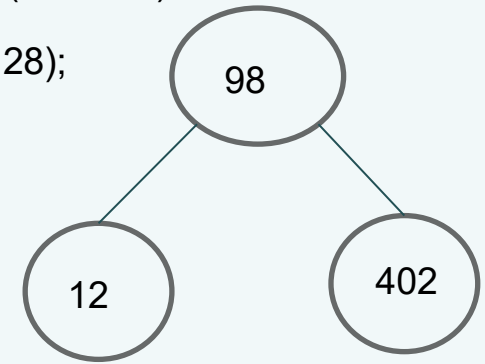
```
binary_tree_t  *binary_tree_insert_left (binary_tree_t *parent, int value);

        binary_tree_t *root ;

        root = binary_tree_node (NULL,98);

       root ->left = binary_tree_node(root,12);

        binary_tree_insert_left (root,54);
```

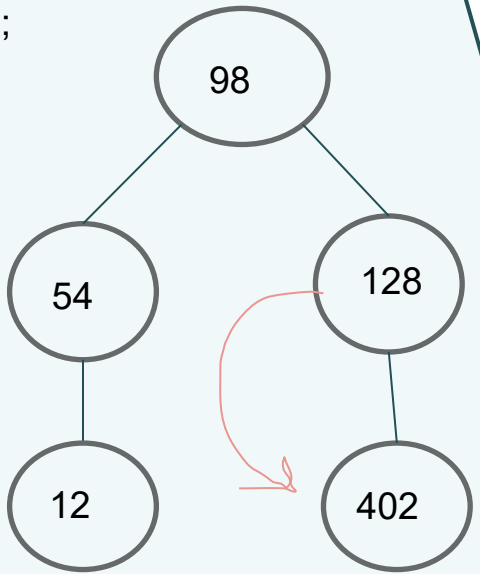


# Insert Right

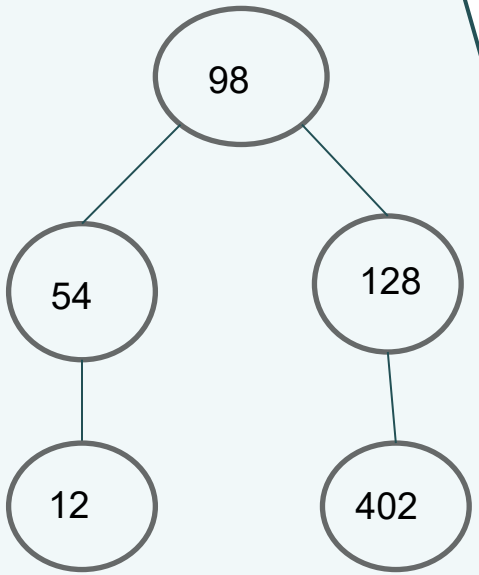
```
binary_tree_t  *binary_tree_insert_right (binary_tree_t *parent, int value);
   binary_tree_t *root ;
   root = binary_tree_node (NULL,98);
root ->left = binary_tree_node(root,12);
root ->right = binary_tree_node(root,402);
binary_tree_insert_right (root,128);
```



# Delete

```
binary_tree_t  *binary_tree_delete(binary_tree_t *parent, int value);

   binary_tree_t *root ;

   root = binary_tree_node (NULL,98);

root ->left = binary_tree_node(root,12);

root ->right = binary_tree_node(root,402);

binary_tree_delete(root);
```



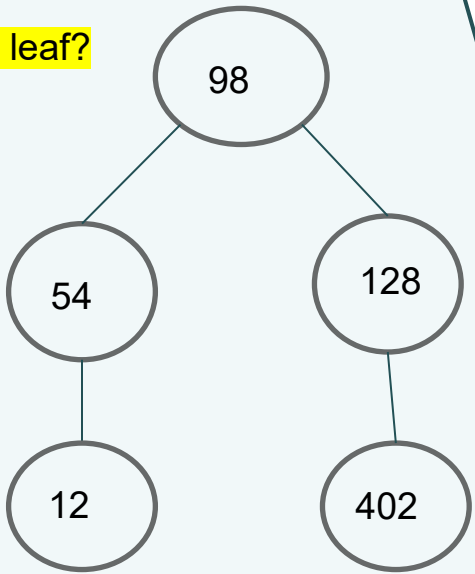

طريقة الحذف تبدا من اليسار الى اليمين الى النود نفسه :
وهذا يسمى :
pre-order-traversal

# CHECK FUNCTION

# Is Leaf

كيف نتحقق اذا نود اللي اعطيناها
Leaf ?

Where is the leaf?




If (node != NULL && node -> left == NULL && node -> right == NULL);

return (1);
else
return (0);

Main.c:

وش راح يرجع لي ؟

ret = binary_tree_is_leaf(root); 0

ret = binary_tree_is_leaf(root->right); 0

ret = binary_tree_is_leaf(root->right->right); 1

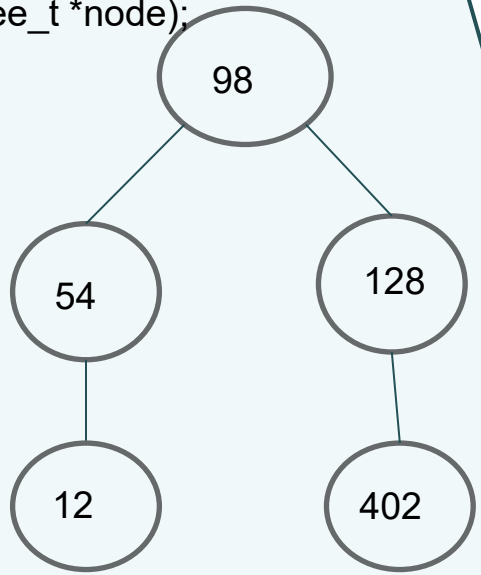
# Is Root

كيف نتحقق اذا نود اللي عطيناها root?

Int binary_tree_is_root(const binary_tree_t *node);

If (node != NULL && node -> parent == NULL)

return (1);
else
return (0);




Main.c:

وش راح يرجع لي ؟

ret = binary_tree_is_root(root); 1

ret = binary_tree_is_root(root->right); 0

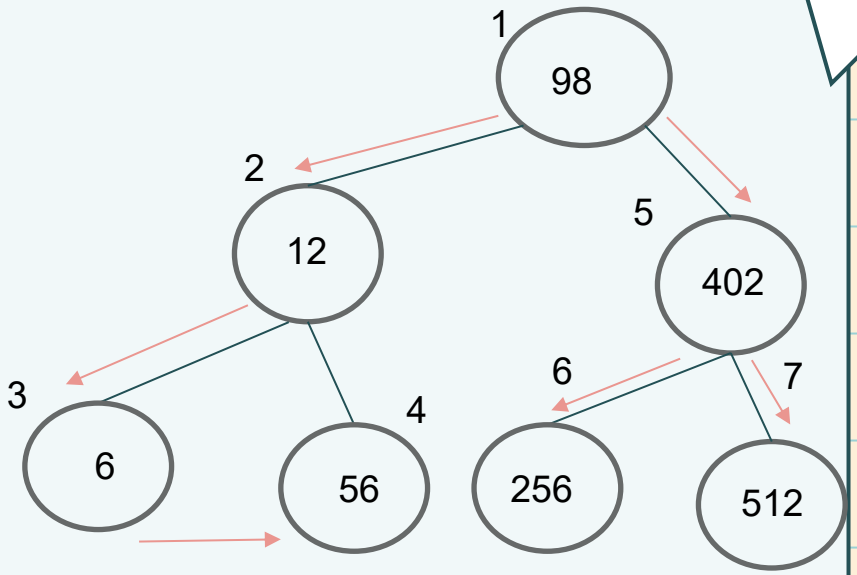
ret = binary_tree_is_root(root->right->right); 0

# Pre-order-traversal

Main.c :

```
Void print_num(int n )
{
pritf(“%d/n”,n);
}
```

98
12
6
56
402
256
512

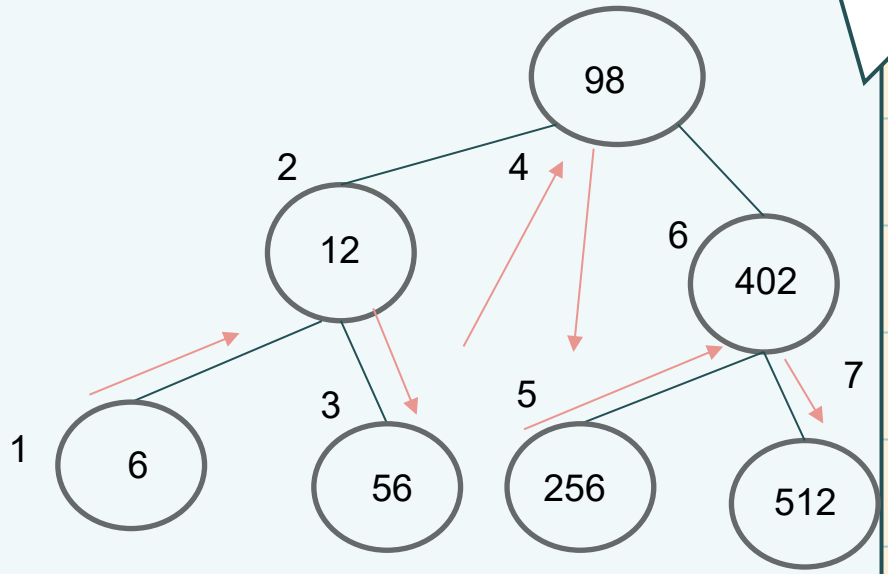


# In-order-traversal

Main.c :

```
Void print_num(int n )
{
pritf(“%d/n”,n);
}
```

6
12
56
98
356
402
512

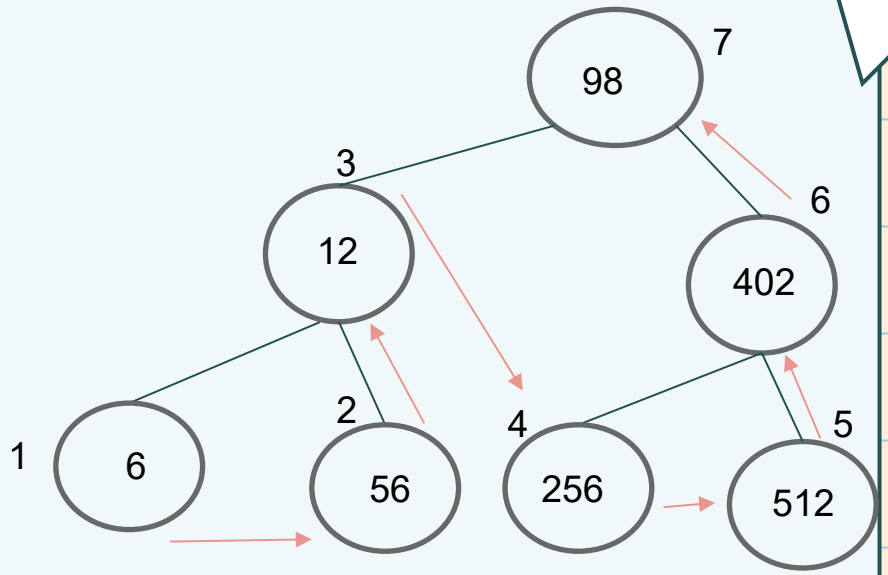


# Post-order-traversal

Main.c :

```
Void print_num(int n )
{
pritf(“%d/n”,n);
}
```

6
56
12
256
512
402
98



# Harry Up

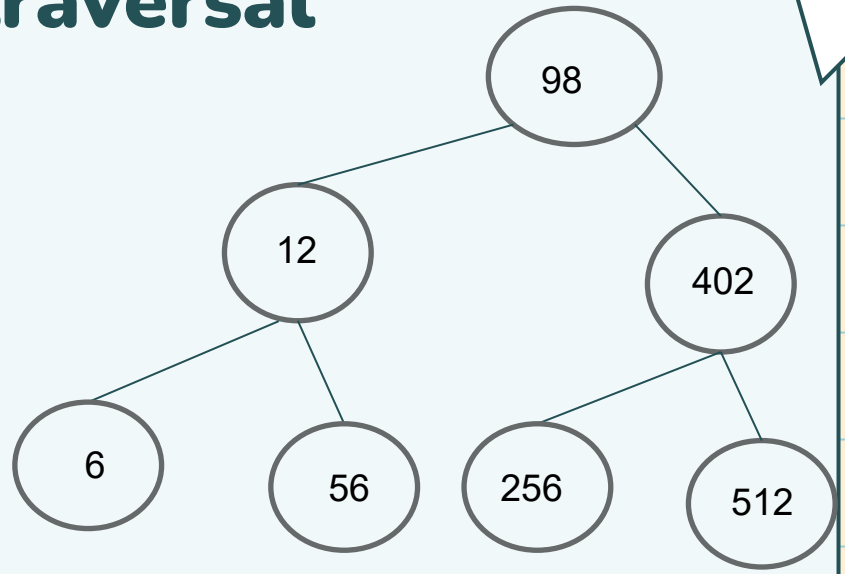
# Speed test

# Draw Pre-order-traversal

Main.c :

```
Void print_num(int n )
{
pritf(“%d/n”,n);
}
```

98
12
6
56
402
256
512



# Harry Up

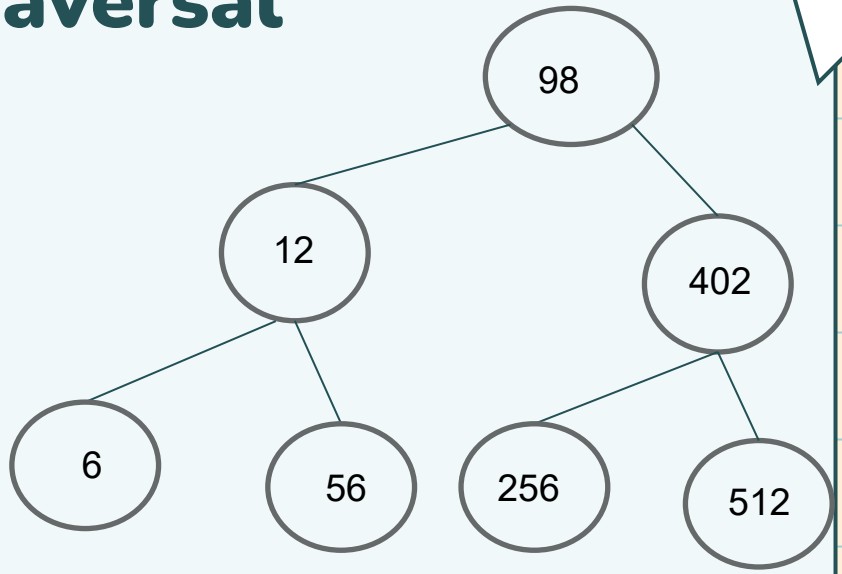
# Speed test

# Draw In-order-traversal

Main.c :

```
Void print_num(int n )
{
pritf(“%d/n”,n);
}
```

6
12
56
98
356
402
512



# Harry Up

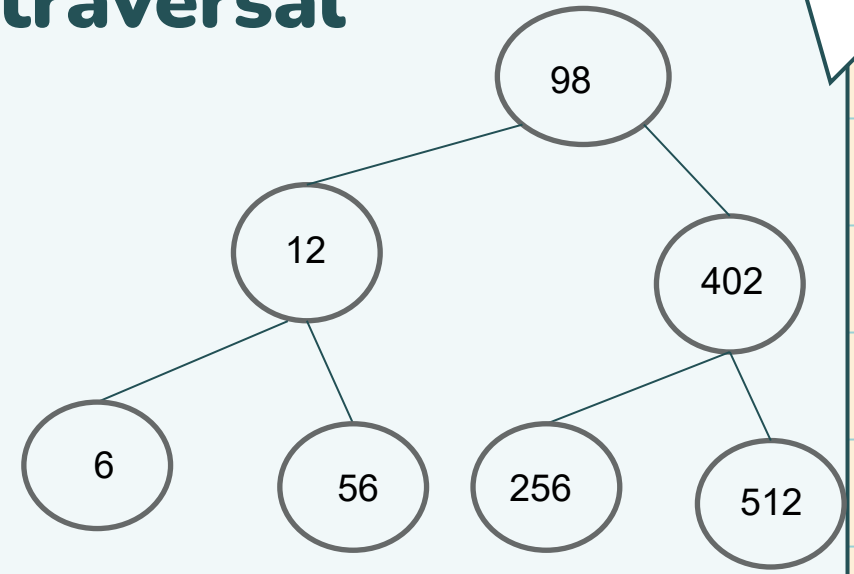
# Speed test

# Draw Post-order-traversal

Main.c :

```
Void print_num(int n )
{
pritf(“%d/n”,n);
}
```

6
56
12
256
512
402
98



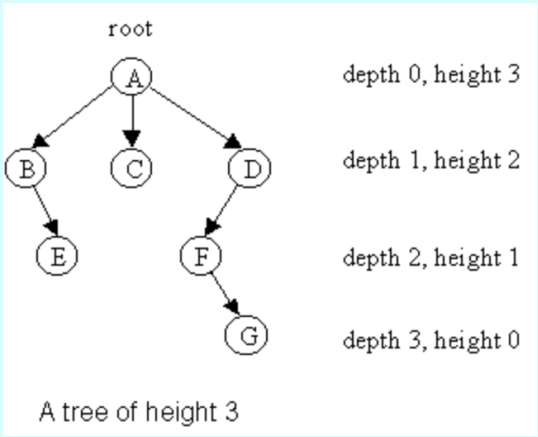
# 18-9

Bushra Alshulail

# Task 9 > height

**What is the height ?**

**Number of** edges .

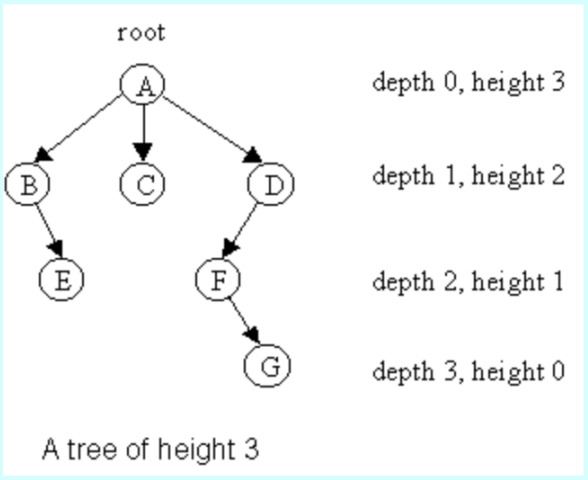



```c
size_t binary_tree_height(const binary_tree_t *tree)
{
size_t left_height, right_height;

if (tree == NULL)
return (0);

if (tree->left == NULL && tree->right == NULL)
return (0);

left_height = binary_tree_height(tree->left);
right_height = binary_tree_height(tree->right);

return ((left_height > right_height ? left_height : right_height)+1);
}
```

# Task 10 > Depth

## What is the Depth ?

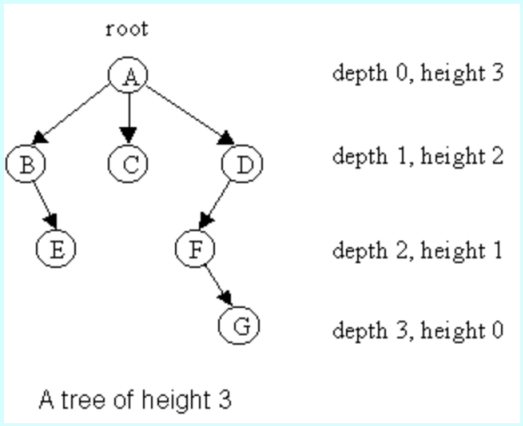
## Number of parents




```c
size_t binary_tree_depth(const binary_tree_t *tree)
{
size_t depth = 0;

if (tree == NULL)
return (0);

while (tree->parent != NULL)
{
depth++;
tree = tree->parent;
}

return (depth);
}
```

# Task 11 > size

## What is the size ?




```c
size_t binary_tree_size(const binary_tree_t *tree)
{
if (tree == NULL)
return (0);

return (1 + binary_tree_size(tree->left) + binary_tree_size(tree->right));
}
```

# Task 12 > Leaves

```c
size_t binary_tree_leaves(const binary_tree_t *tree)
{
if (tree == NULL)
return (0);

if (tree->left == NULL && tree->right == NULL)
return (1);

return (binary_tree_leaves(tree->left) + binary_tree_leaves(tree->right));
}
```

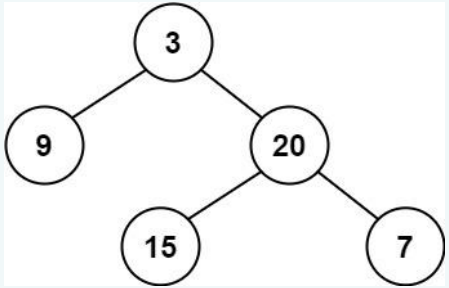
# Task 13 > Nodes

```c
size_t binary_tree_nodes(const binary_tree_t *tree)
{
if (tree == NULL)
return (0);

if (tree->left == NULL && tree->right == NULL)
return (0);

return (1 + binary_tree_nodes(tree->left) + binary_tree_nodes(tree->right));
}
```

# Task 14 > balance binary tree

## What is balance binary tree ?

means that the difference between the heights of the left and right subtrees of any node in the tree does not exceed 1.



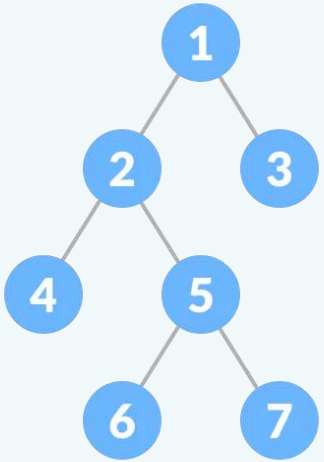

```c
int binary_tree_balance(const binary_tree_t *tree)
{
if (tree == NULL)
return (0);

return (height(tree->left) - height(tree->right));
}
```

Balance if balance factor = zero

# Task 15 > full binary tree

## What is full binary tree ?

every node has either 0 or 2 children.




```c
int binary_tree_is_full(const binary_tree_t *tree)
{
if (tree == NULL)
return (0);

/* If it's a leaf node */
if (tree->left == NULL && tree->right == NULL)
return (1);

/* If both left and right children exist, check recursively */
if (tree->left && tree->right)
return (binary_tree_is_full(tree->left) &&
binary_tree_is_full(tree->right));

/* If one child is missing */
return (0);
}
```

# Task 17 > sibling

## What is sibling?

is a node that shares the same parent with another node.

```c
binary_tree_t *binary_tree_sibling(binary_tree_t *node)
{
if (node == NULL || node->parent == NULL)
return (NULL);

if (node->parent->left == node)
return (node->parent->right);

return (node->parent->left);
}
```

# Task 18 > uncle

## What is uncle?

The **uncle** of a node is the sibling of its parent.

```
binary_tree_t *binary_tree_uncle(binary_tree_t *node)
{
if (node == NULL || node->parent == NULL || node->parent->parent == NULL)
return (NULL);

if (node->parent->parent->left == node->parent)
return (node->parent->parent->right);

return (node->parent->parent->left);
}
```

# Time of Quizizz

# Thanks