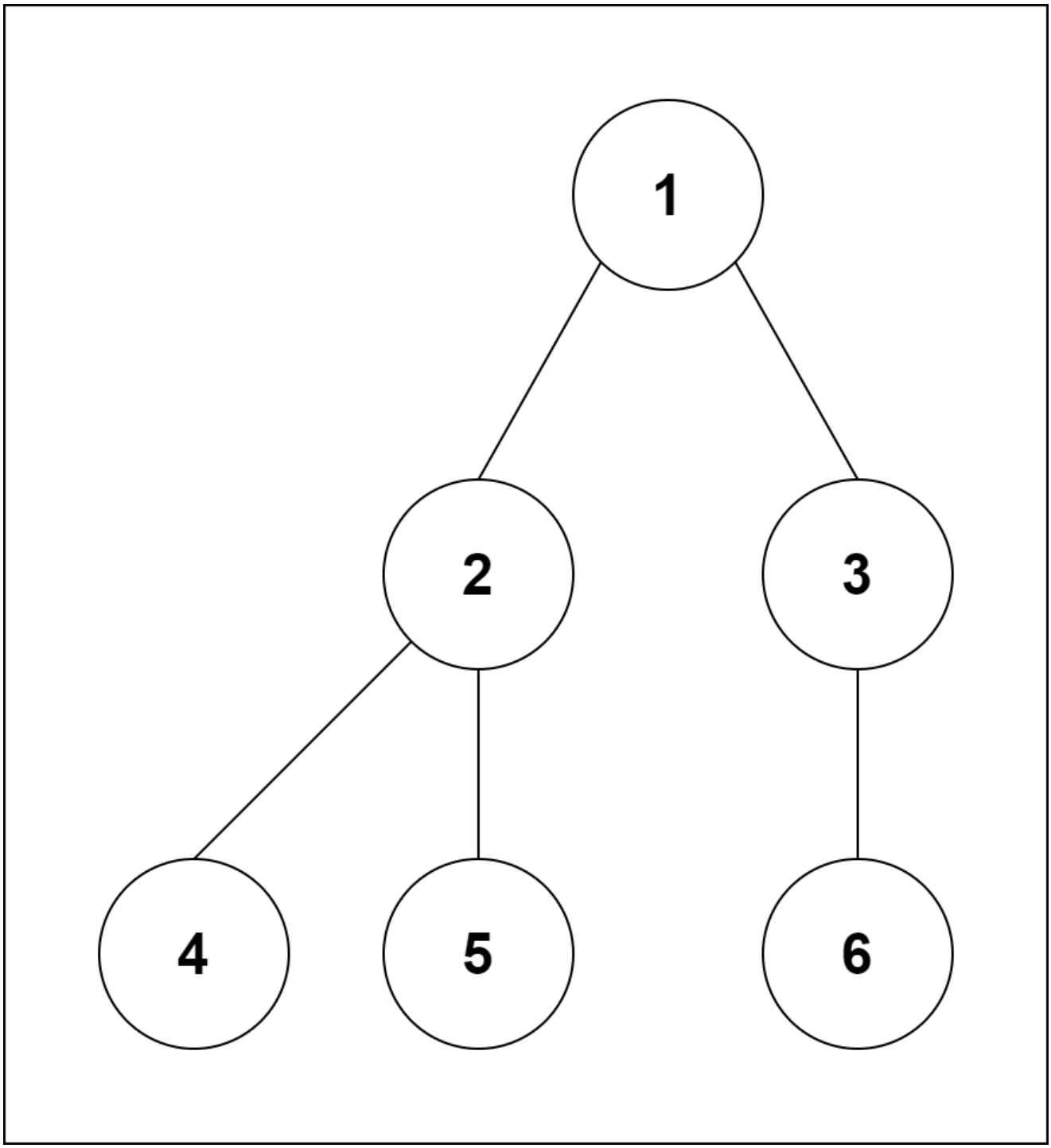
Complete Binary Tree

A complete binary tree is a special kind of binary tree. In this type of tree, all levels are fully filled except for the last one, which is filled from the left side.

Now, what's important about a complete binary tree is that all the leaf nodes, these are the ones without any children, they all appear on the left side. Imagine it like neatly stacking boxes where you start from the left and fill each row before moving on to the next.

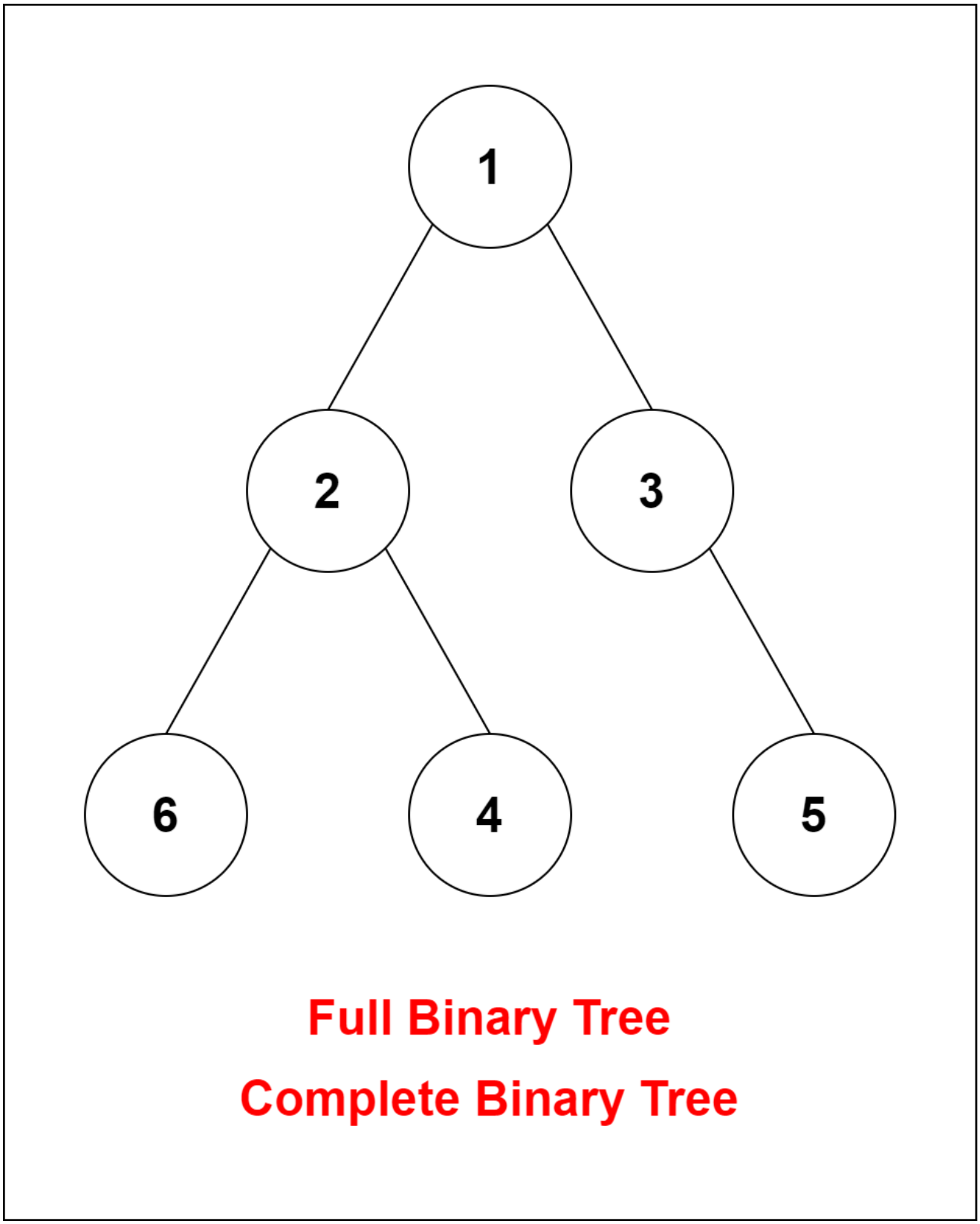
One more thing, unlike a full binary tree where every parent has two children, in a complete binary tree, the last leaf element, the one at the far right, might not have a sibling on its right side. So, it doesn't have to be a full binary tree where every parent has two children; it just needs to be complete with the left-leaning arrangement.



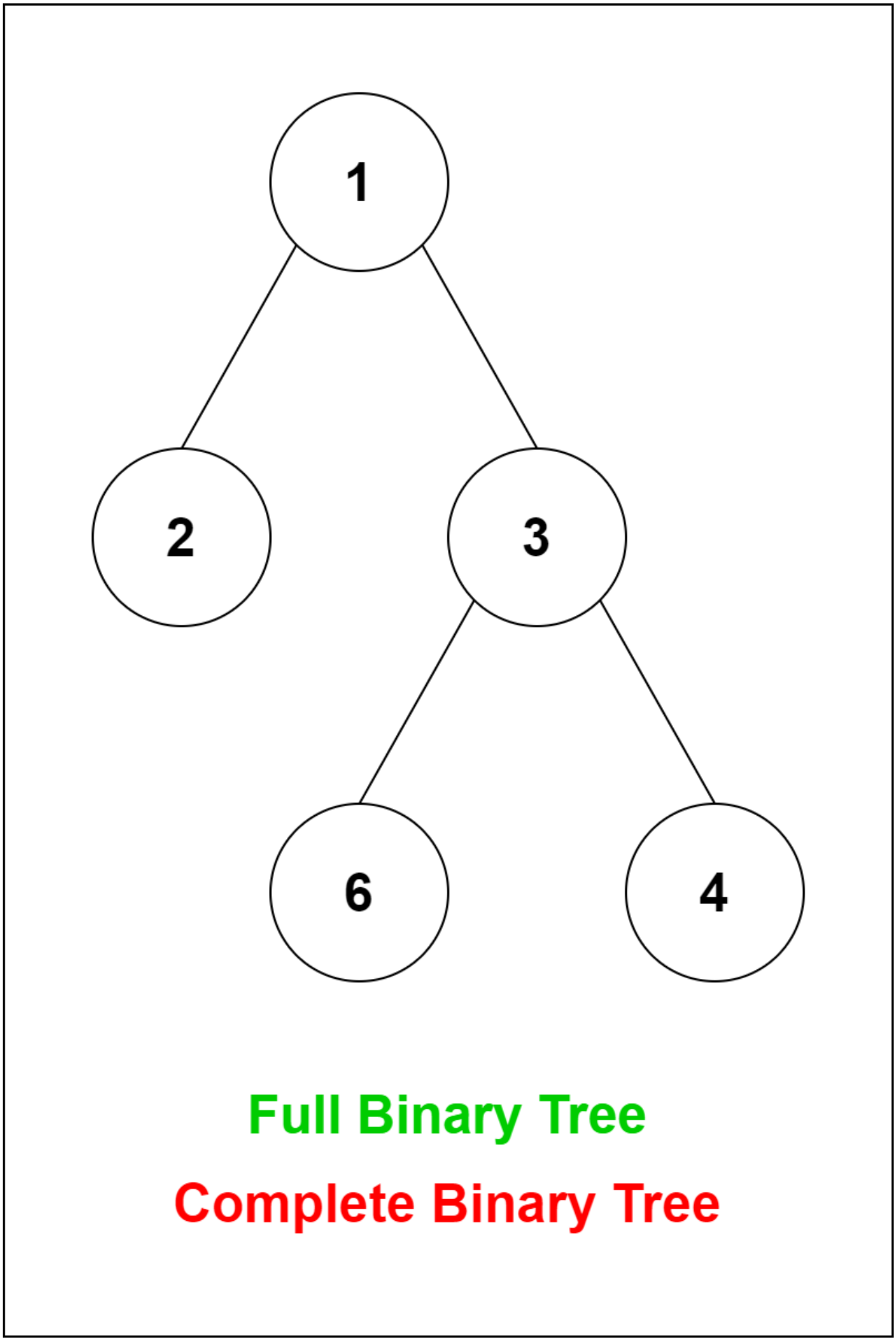
Full Binary Tree vs Complete Binary Tree

Let's break down the concepts of full and complete binary trees:

1. **Not Full and Not Complete Binary Tree:**
   - **Not Full:** If a binary tree has nodes with only one child or no child, it is considered not full. A full tree should have every node with either 0 or 2 children.
   - **Not Complete:** A binary tree is considered complete if all levels, except possibly the last, are completely filled, and all nodes are as left as possible. Gaps in the last level to the left make the tree not complete.

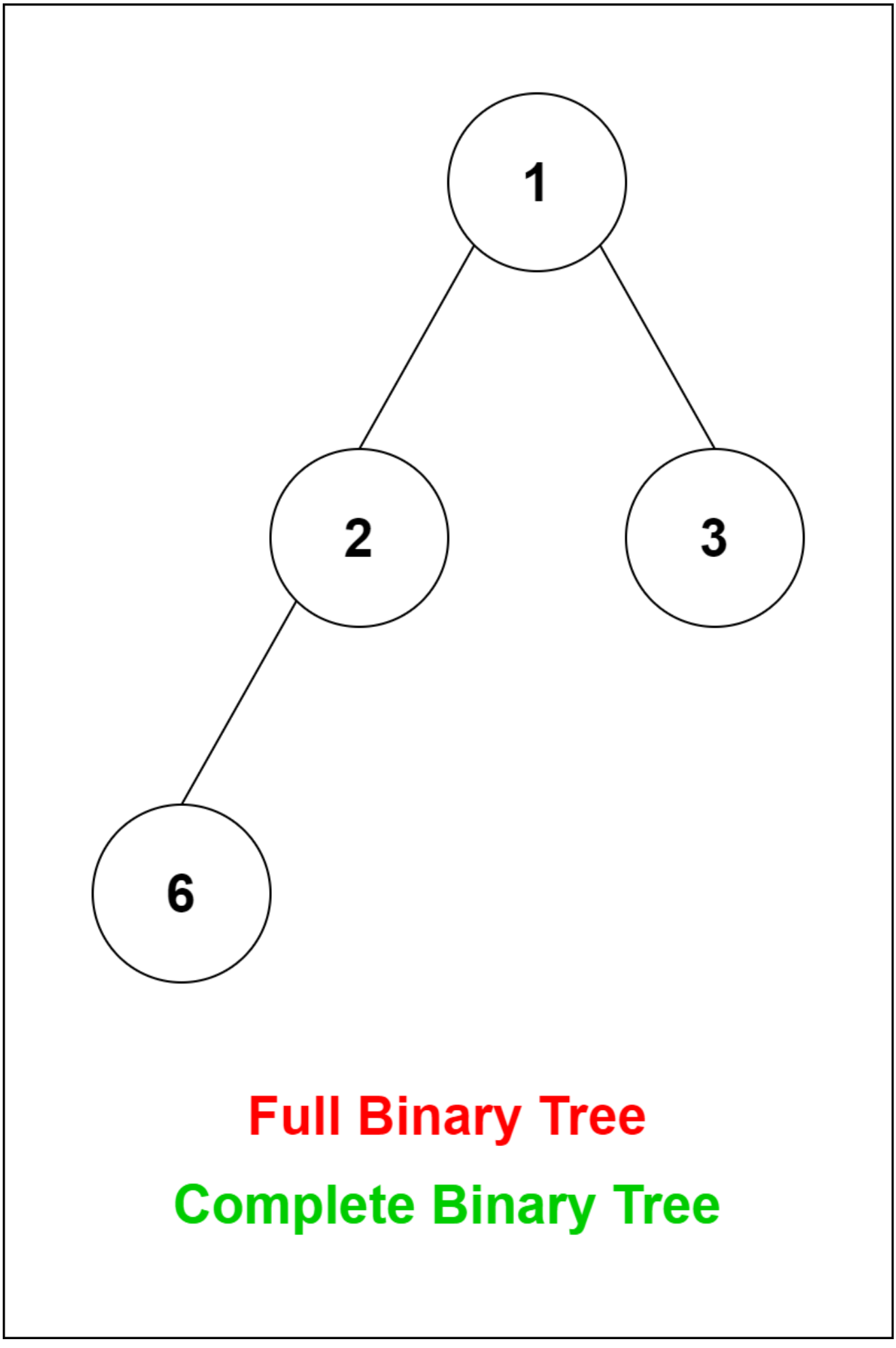


2. **Full and Not Complete Binary Tree:**
    - **Full:** A binary tree is full when every node has either 0 or 2 children. No node has only one child.
    - **Not Complete:** Despite being full, this tree is not complete. It means that there might be gaps in the last level to the left.



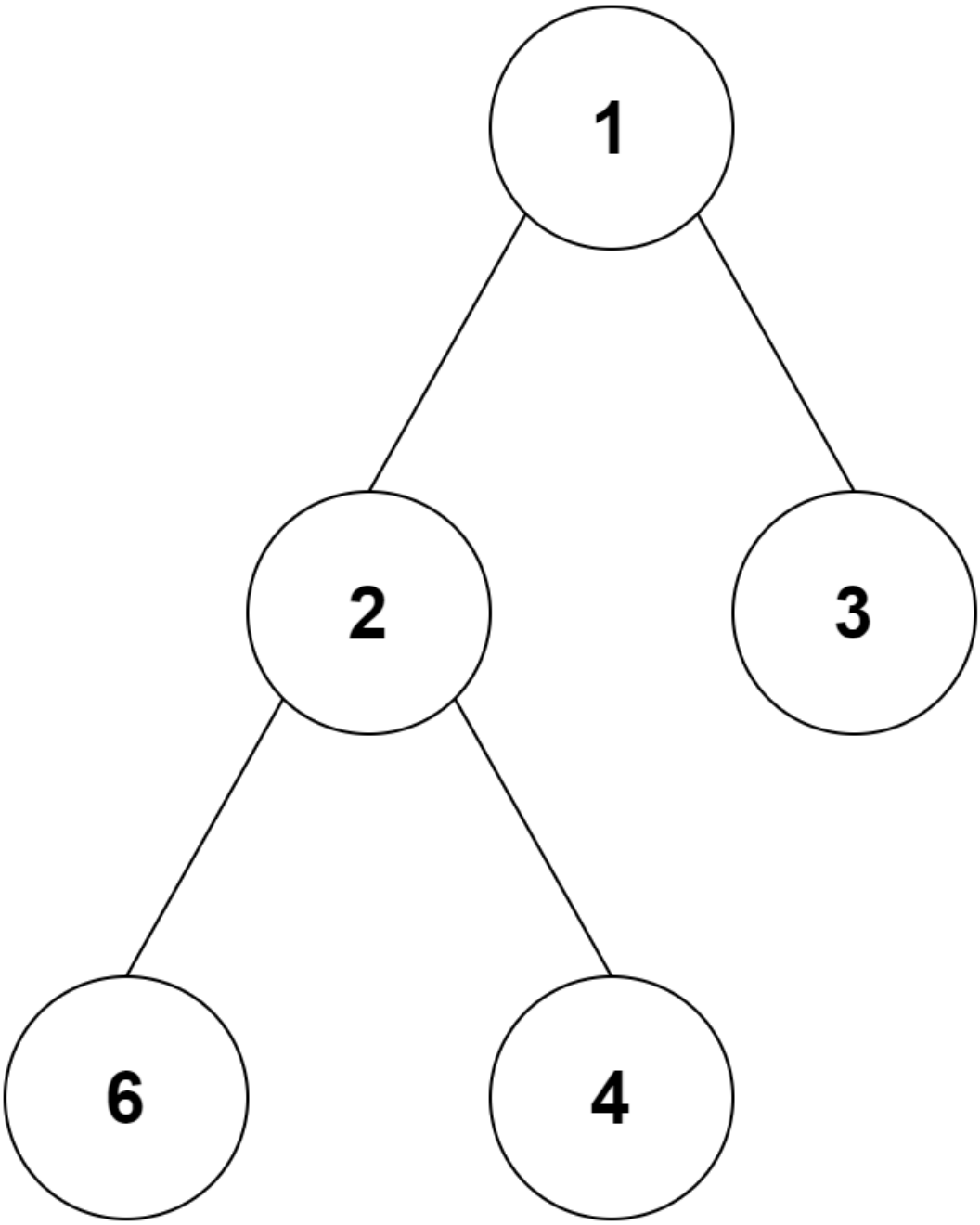
3. **Not Full and Complete Binary Tree:**

- **Not Full:** In this case, some nodes may have only one child.
- **Complete:** Despite not being full, this tree is complete. All levels, except possibly the last, are completely filled, and all nodes are as left as possible. Nodes with one child may exist, but gaps are to the left in the last level.



4. **Full and Complete Binary Tree:**

- **Full:** In a fully binary tree, every node has either 0 or 2 children, and no node has only one child.
- **Complete:** This tree is also complete. All levels, except possibly the last, are completely filled, and all nodes are as left as possible. It satisfies both conditions of being full and complete.



Full Binary Tree

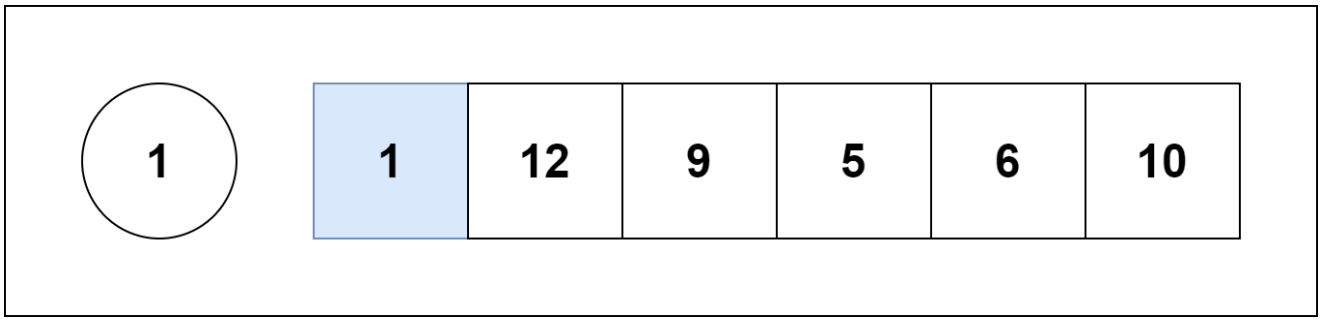
Complete Binary Tree

In summary, "full" refers to the arrangement of children for each node, while "complete" refers to the filling of levels from left to right. A binary tree can be both full and complete, or neither. These terms describe different characteristics of binary trees.

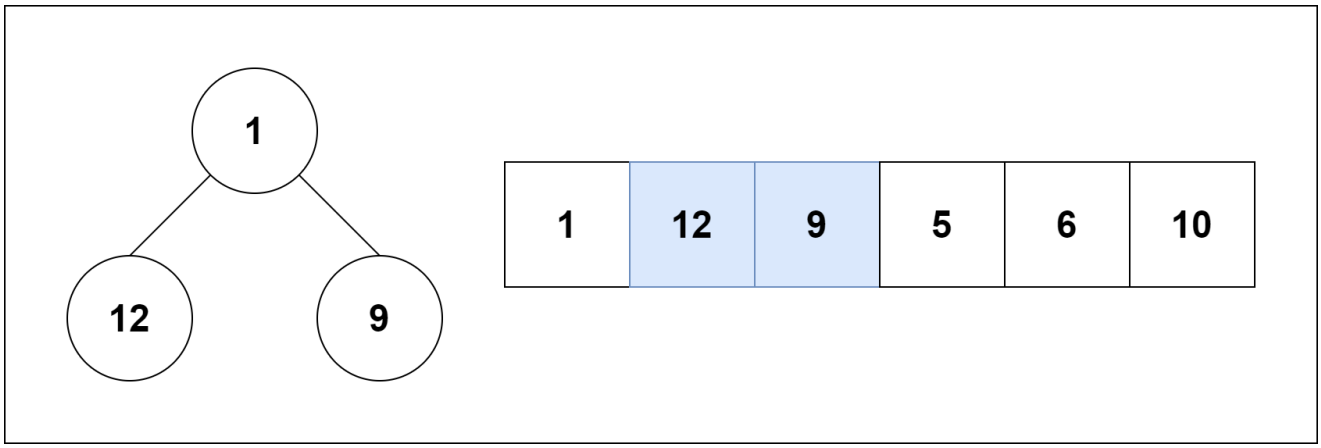
How a Complete Binary Tree is Created?

Creating a complete binary tree is a bit like building a pyramid of boxes. Here's how it works:

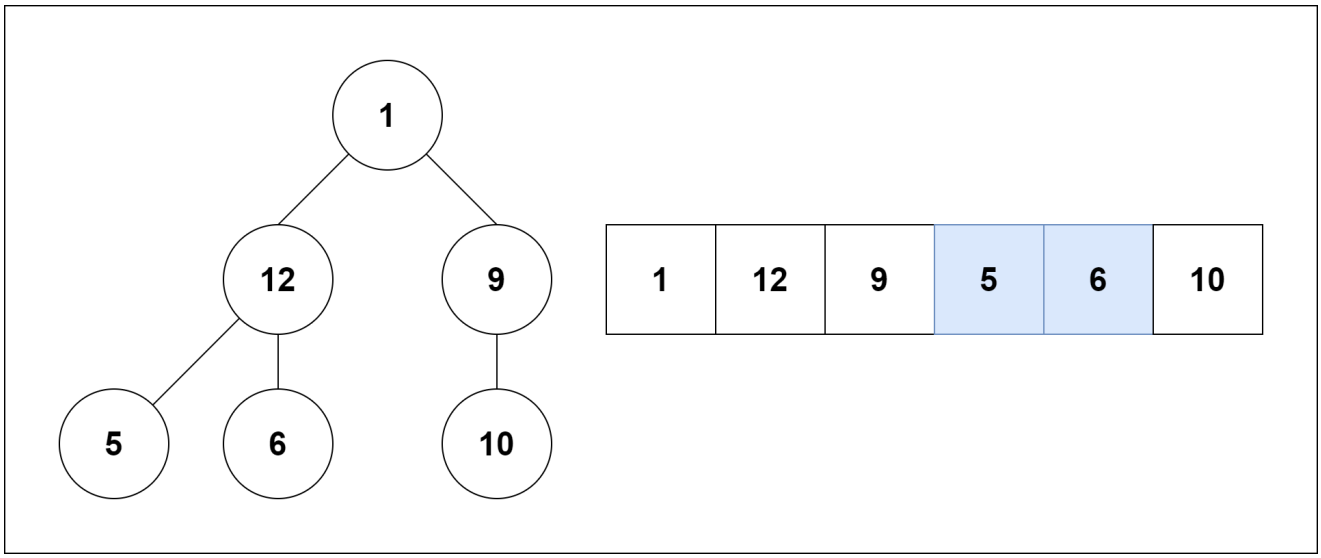
1. Start with the first element in your list. This element becomes the root node of your tree. At this point, you have just one element on the first level of the tree.



2. Take the second element from your list and make it the left child of the root node. Then, take the third element and make it the right child of the root node. Now, you have two elements on the second level of the tree.



3. Continue this pattern. Take the next two elements in your list and make them the children of the left node on the second level. Then, take the next two elements and make them the children of the right node on the second level. This doubles the number of elements on each level.
4. Keep doing this until you've used up all the elements in your list.



By following these steps, you'll create a complete binary tree where every level is filled from left to right, except possibly the last level, and every element finds its place in the tree.

C++ Example

```
// Checking if a binary tree is a complete binary tree in C++

#include <iostream>

using namespace std;

struct Node {
  int key;
  struct Node *left, *right;
};

// Create node
struct Node *newNode(char k) {
  struct Node *node = (struct Node *)malloc(sizeof(struct Node));
  node->key = k;
  node->right = node->left = NULL;
  return node;
}

// Count the number of nodes
int countNumNodes(struct Node *root) {
  if (root == NULL)
    return (0);
  return (1 + countNumNodes(root->left) + countNumNodes(root->right));
}

// Check if the tree is a complete binary tree
```

```
bool checkComplete(struct Node *root, int index, int numberNodes) {

  // Check if the tree is empty
  if (root == NULL)
    return true;

  if (index >= numberNodes)
    return false;

  return (checkComplete(root->left, 2 * index + 1, numberNodes) &&
checkComplete(root->right, 2 * index + 2, numberNodes));
}

int main() {
  struct Node *root = NULL;
  root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);
  root->right->left = newNode(6);

  int node_count = countNumNodes(root);
  int index = 0;

  if (checkComplete(root, index, node_count))
    cout << "The tree is a complete binary tree\n";
  else
    cout << "The tree is not a complete binary tree\n";
}
```

Relationship between array indexes and tree element

Great! Now, let's explore how array indexes relate to the elements in a complete binary tree.

In a complete binary tree, you can use a simple formula to find the children and parents of any node. Here's how it works:

1. If you have the index of any element in your array, let's call it 'i', then the element at index '2i+1' will be its left child, and the element at index '2i+2' will be its right child.

For example, if we want to find the left child and right child of the element at index 0, which is 1, we use this formula:

- Left child of 1 (index 0) = element at (2 * 0 +1) index = element at 1 index = 12
- Right child of 1 = element at (2 * 0 + 2) index = element at 2 index = 9

Similarly, if we want to find the children of the element 12 (index 1):

- Left child of 12 (index 1) = element at (2 * 1 + 1) index = element at 3 index = 5
- Right child of 12 = element at (2 * 1 + 2) index = element at 4 index = 6

2. To find the parent of any element at index 'i', you can use the formula: (i - 1) / 2, rounding down to the nearest integer.

For example, if we want to find the parent of 9 (position 2), we calculate:

- Parent of 9 (position 2) = (2 - 1) / 2 = 1 / 2 = 0.5 (rounded down to 0)
- So, the parent of 9 is the element at index 0, which is 1.

Similarly, to find the parent of 12 (position 1):

- Parent of 12 (position 1) = (1 - 1) / 2 = 0 / 2 = 0
- So, the parent of 12 is also the element at index 0, which is 1.

Understanding this relationship between array indexes and tree elements is essential for working with data structures like the Heap and implementing algorithms like Heap Sort. It allows us to efficiently navigate and manipulate elements in the tree structure.

Complete Binary Tree Applications

Let's discuss the practical applications of a complete binary tree.

1. **Heap-Based Data Structures:** Complete binary trees are fundamental in the implementation of heap-based data structures. Heaps are specialized tree structures that satisfy the heap property, which means that each parent node has a value greater than or equal to (in a max-heap) or less than or equal to (in a min-heap) the values of its children. Complete binary trees are efficient for representing heaps because they maintain a balance that ensures optimal performance for insertion and deletion operations. Heaps are used in various applications, including priority queues, which are commonly employed in algorithms like Dijkstra's algorithm for finding the shortest path and in scheduling processes.
2. **Heap Sort:** Heap sort is a comparison-based sorting algorithm that utilizes the heap data structure. It transforms an array into a max-heap, which is a complete binary tree with the highest element at the root. Then, it repeatedly removes the maximum element from the heap and places it at the end of the array. This process continues until the array is fully sorted. Heap sort has a time complexity of O(n log n), making it efficient for large datasets. It's widely used for tasks like sorting large volumes of data in databases and as a building block in various algorithms and applications.

Understanding complete binary trees is crucial when working with these applications, as it ensures that the underlying tree structure is correctly maintained for efficient operations.