Balanced Binary Tree

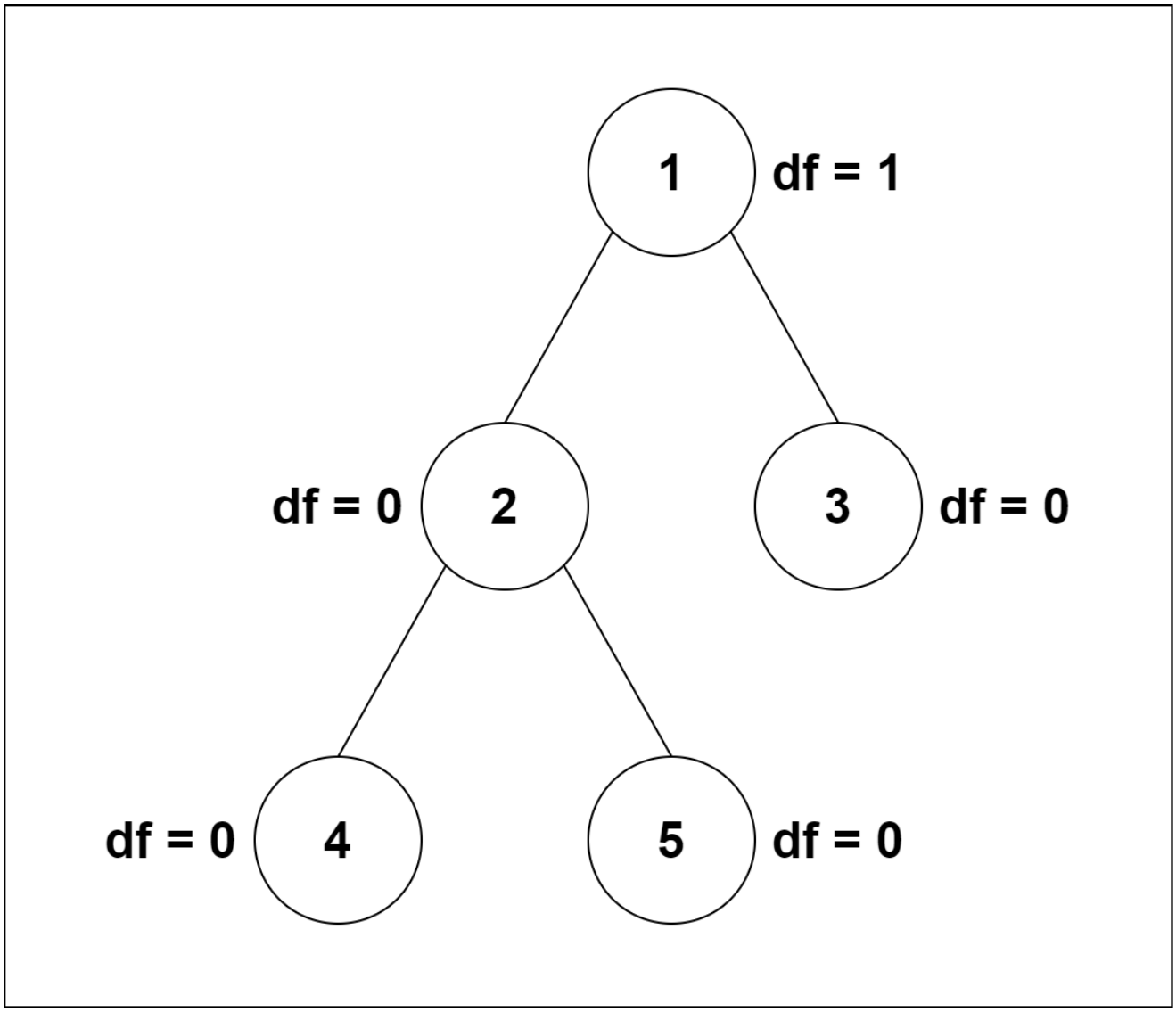
Alright, let's talk about something called a balanced binary tree.

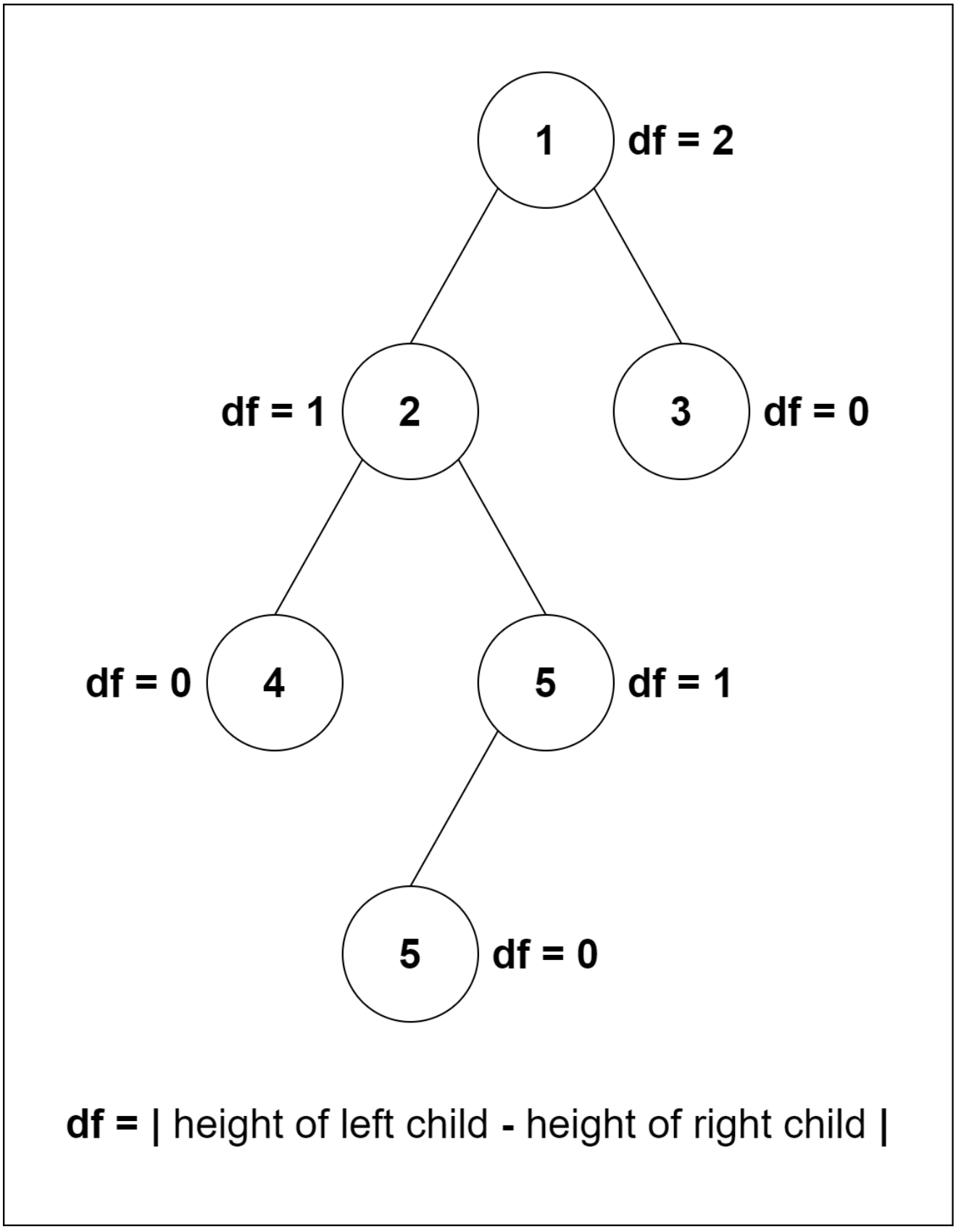
Imagine a binary tree where each node has, at most, two child nodes: one on the left and one on the right. In a balanced binary tree, something interesting is going on.

1. First, at any given node in this tree, the height or distance from that node to its left subtree and its right subtree doesn't differ by more than one level. It's like trying to balance a see-saw; you want both sides to be roughly equal in weight.
2. Second, not only is this balancing act happening at every node, but each of those left and right subtrees is also balanced. So, if you look at any node in the left subtree or the right subtree, they should also satisfy these same conditions.

This balancing act in a binary tree might seem a bit puzzling, but it's quite important in computer science and data structures. It ensures that operations on the tree, like searching for elements or inserting new ones, happen efficiently without taking too much time.

So, remember, in a balanced binary tree, no side of the tree is significantly heavier than the other, and this balance continues throughout the entire tree. This property makes working with these trees really efficient for various tasks.


1
df = 1
df = 0
2
3
df = 0
df = 0
4
5
df = 0



C++ Example

The following code is for checking whether a tree is height-balanced.

```
// Checking if a binary tree is height balanced in C++

#include
using namespace std;

#define bool int

class node {
   public:
  int item;
  node *left;
  node *right;
};

// Create anew node
node *newNode(int item) {
  node *Node = new node();
  Node->item = item;
  Node->left = NULL;
  Node->right = NULL;

  return (Node);
}

// Check height balance
bool checkHeightBalance(node *root, int *height) {
  // Check for emptiness
  int leftHeight = 0, rightHeight = 0;

  int l = 0, r = 0;

  if (root == NULL) {
    *height = 0;
    return 1;
  }

  l = checkHeightBalance(root->left, &leftHeight);
  r = checkHeightBalance(root->right, &rightHeight);

  *height = (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

  if (std::abs(leftHeight - rightHeight >= 2))
    return 0;

  else
    return l && r;
}

int main() {
```

```
  int height = 0;

  node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);

  if (checkHeightBalance(root, &height))
    cout << "The tree is balanced";
  else
    cout << "The tree is not balanced";
}
```

Balanced Binary Tree Applications

Now, let's talk about where we use these balanced binary trees in the real world.

One common use is in something called an AVL tree. AVL stands for the names of its inventors, Adelson-Velsky and Landis. AVL trees are a type of balanced binary search tree. Remember, a binary search tree is a data structure that helps us efficiently organize and search for data.

In an AVL tree, the balancing act we talked about earlier is rigorously maintained. This makes searching for data in AVL trees really fast, and they're often used in computer science for tasks like searching, inserting, and deleting data.

So, when you hear about AVL trees or other balanced binary search trees, remember that they are special kinds of trees designed to keep everything in balance, making data operations fast and efficient.