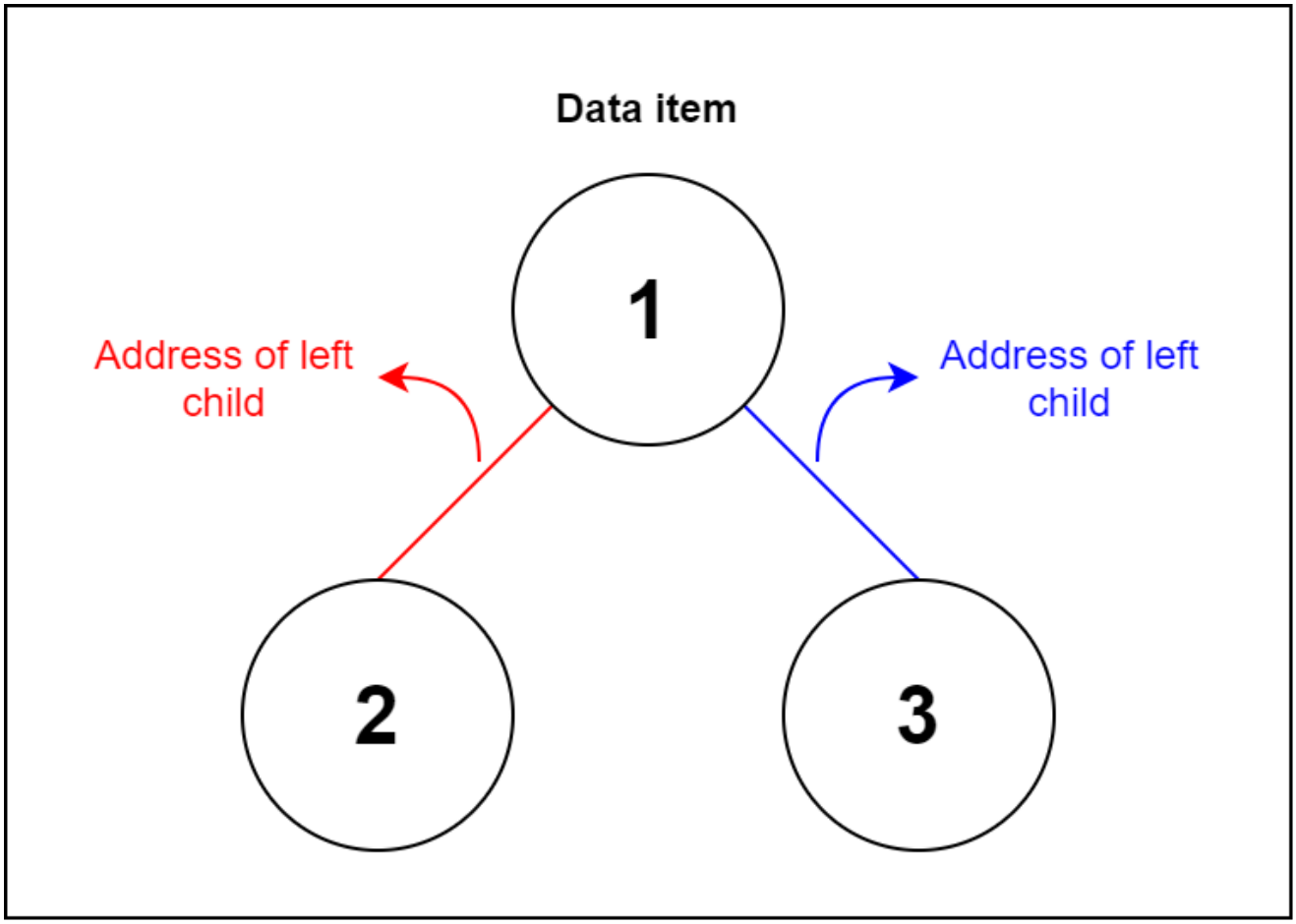
Binary Tree

Let's break down what a binary tree is in simple terms.

Think of a binary tree like a family tree, where every person has at most two children: one on the left and one on the right. Now, in each "person" or node of this tree, you have three things:

1. **Data item**: This is like the name or some information about the person in our family tree.
2. **Address of left child**: It's like saying who the person's left child is, if they have one. In our family tree, it's like saying who their left child is and telling you where to find them in the family tree.
3. **Address of right child**: Similar to the left child, it's telling you who their right child is and where to find them in the family tree.



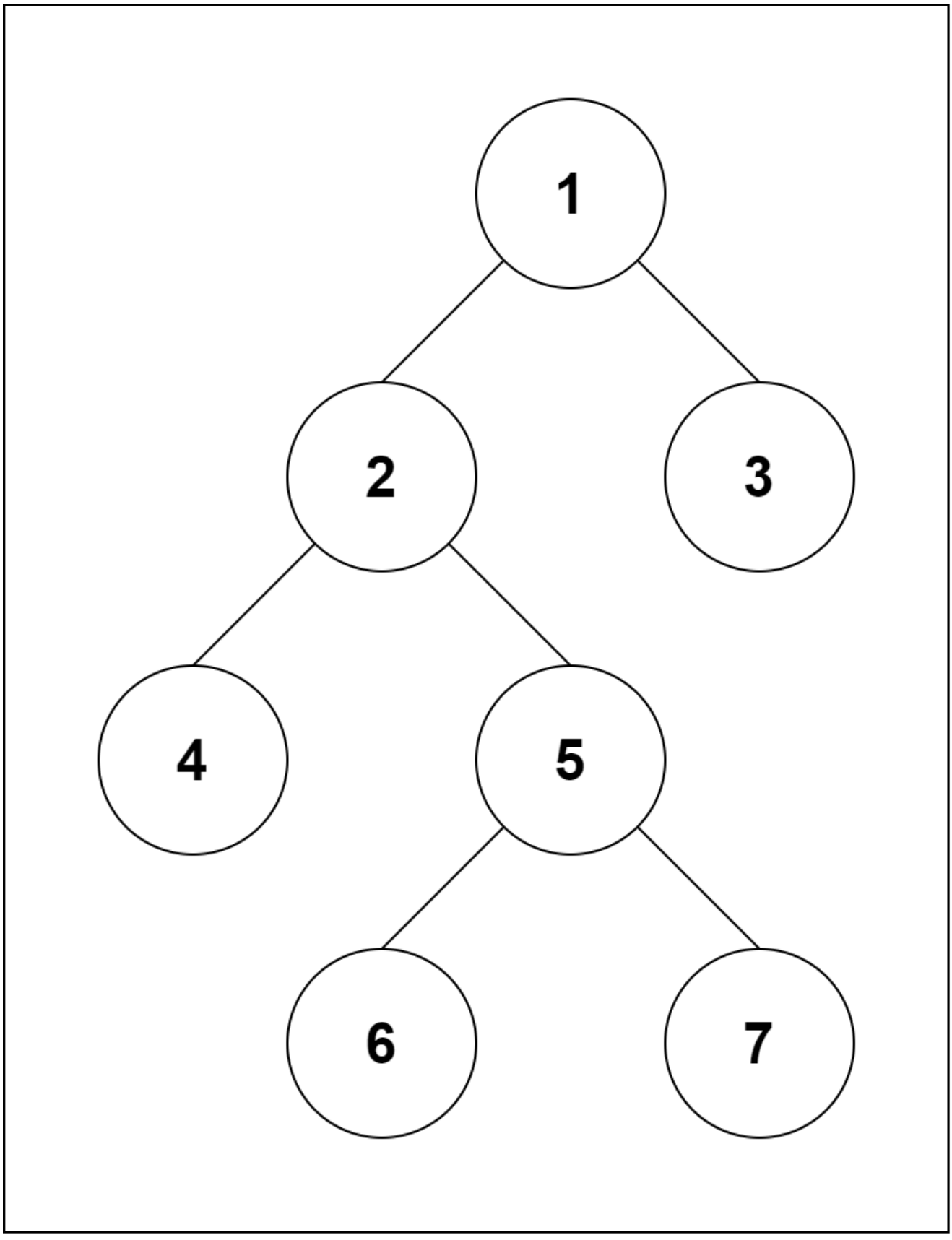
So, in essence, a binary tree is a way of organizing information or data where each piece of data is connected to, at most, two other pieces of data, just like family members in a family tree.

Types of Binary Tree

Let's explore different types of binary trees in a simple way:

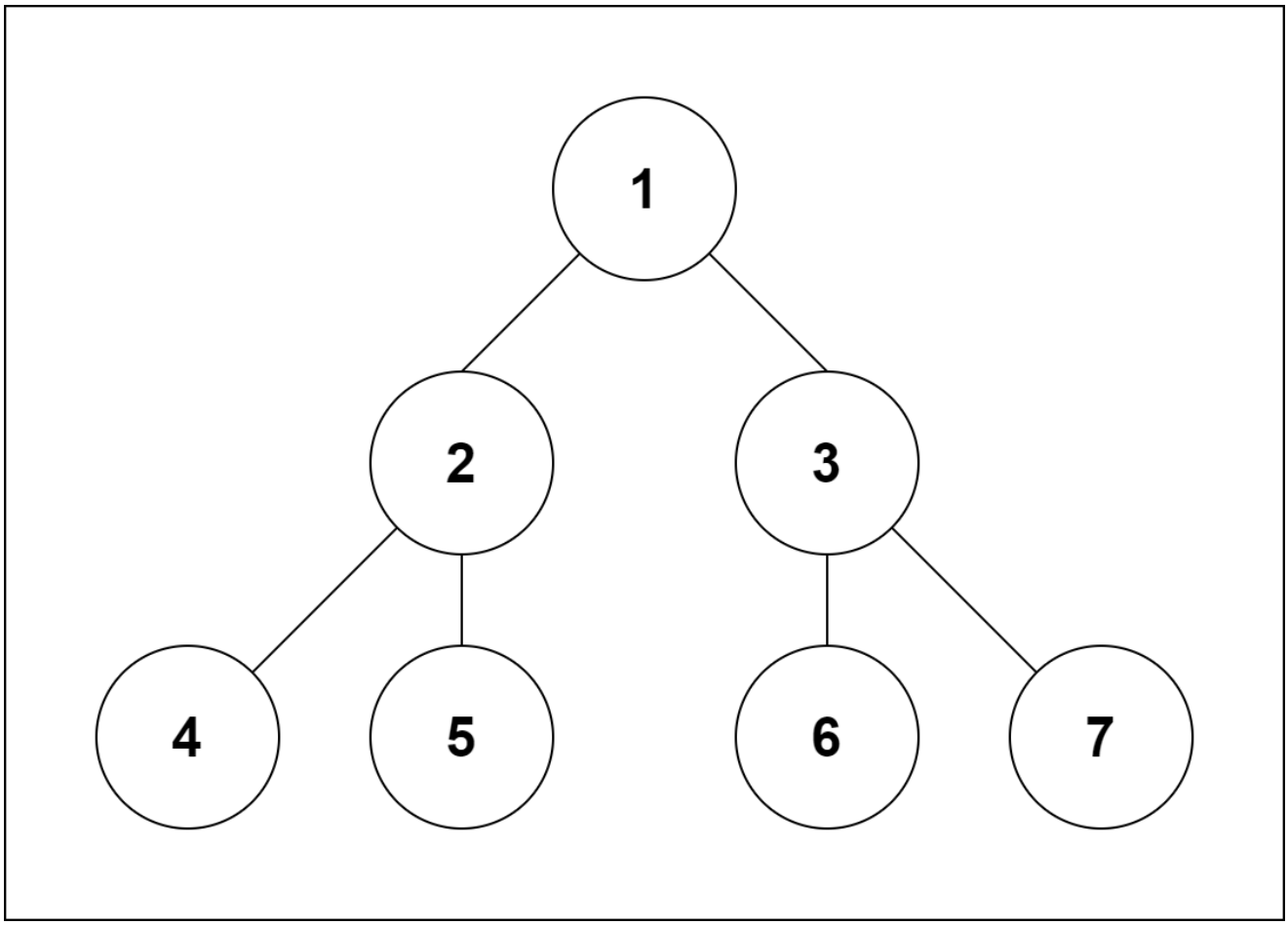
1. Full Binary Tree

Imagine a family where every parent has either two children or none at all. That's what a full binary tree is - every internal family member has two kids.



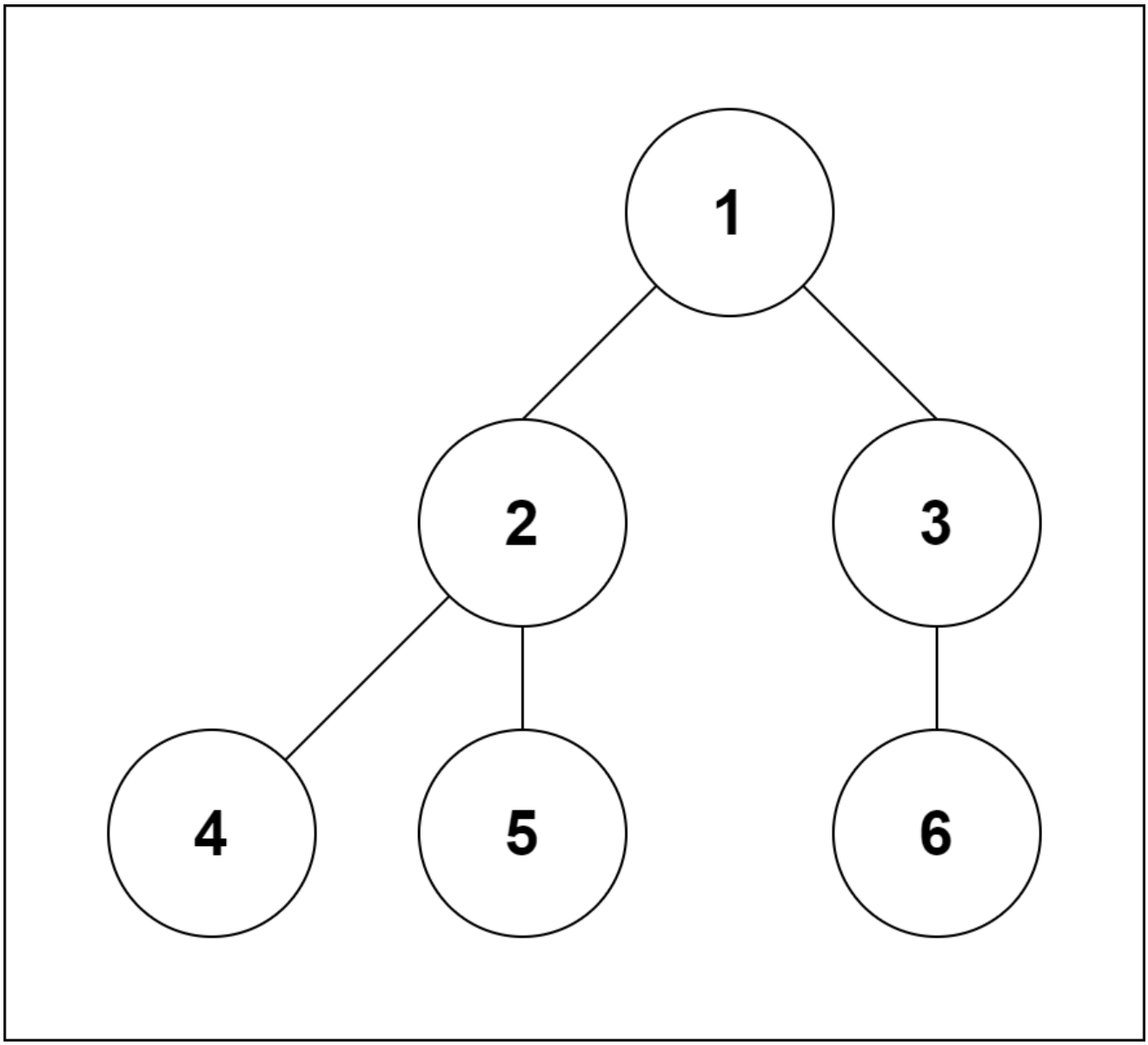
2. Perfect Binary Tree

Now, picture another family, but this time, not only do parents have exactly two children, but all the kids are at the same age level. It's like every generation has the same number of children.



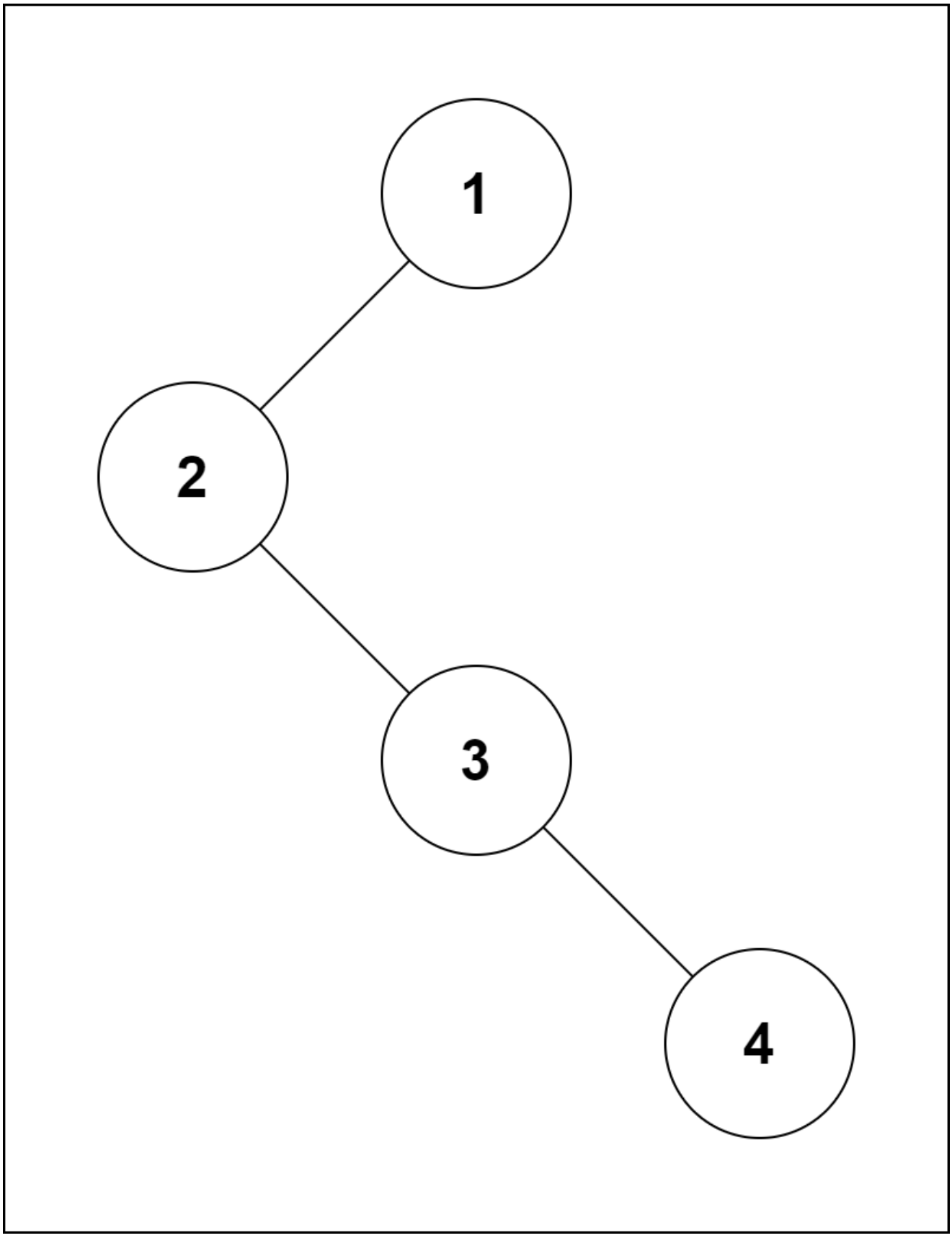
3. Complete Binary Tree

Think of a family gathering where all the tables are set up perfectly. In a complete binary tree, each level of the family gathering is fully occupied, and all the kids sit to the left side, kind of like making sure the dinner tables are filled up as we move from left to right. But the last table might not have all its seats filled.



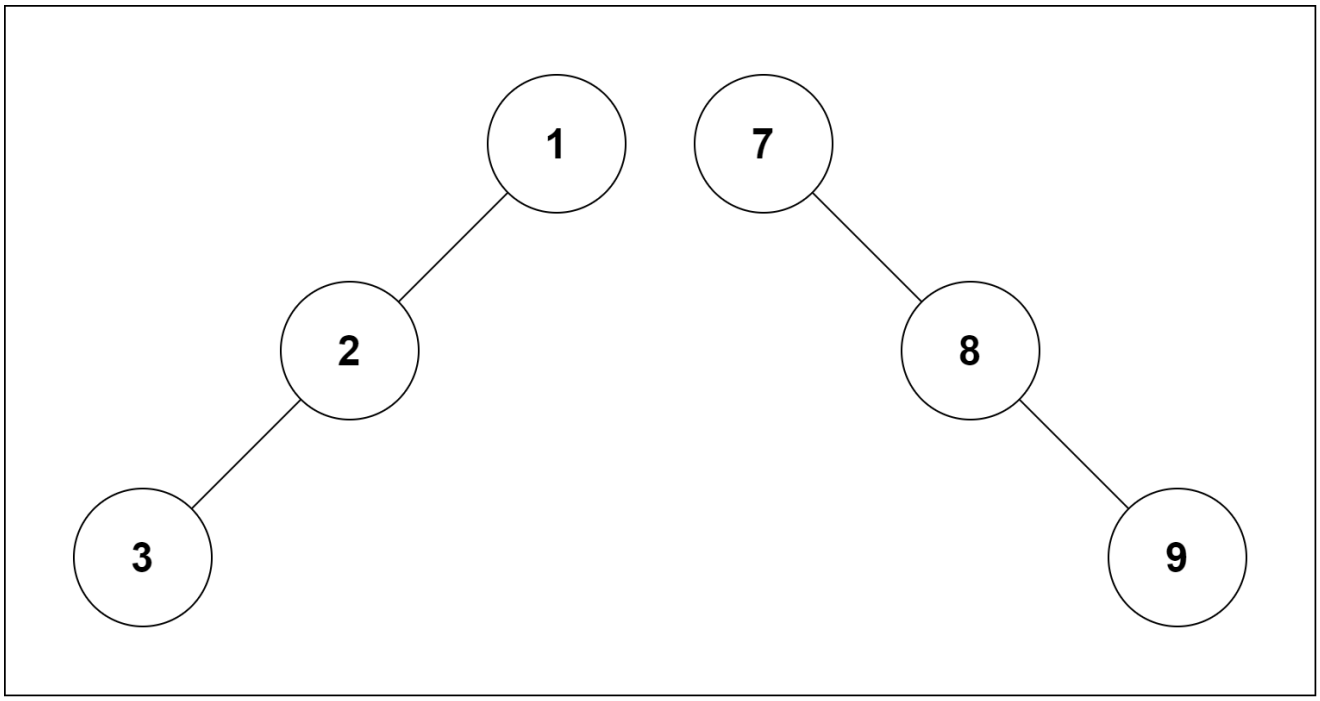
4. Degenerate or Pathological Tree

Sometimes, you might find a family where every parent has just one child, either on the left or the right. It's like a family tree gone in one straight line, not much branching out.



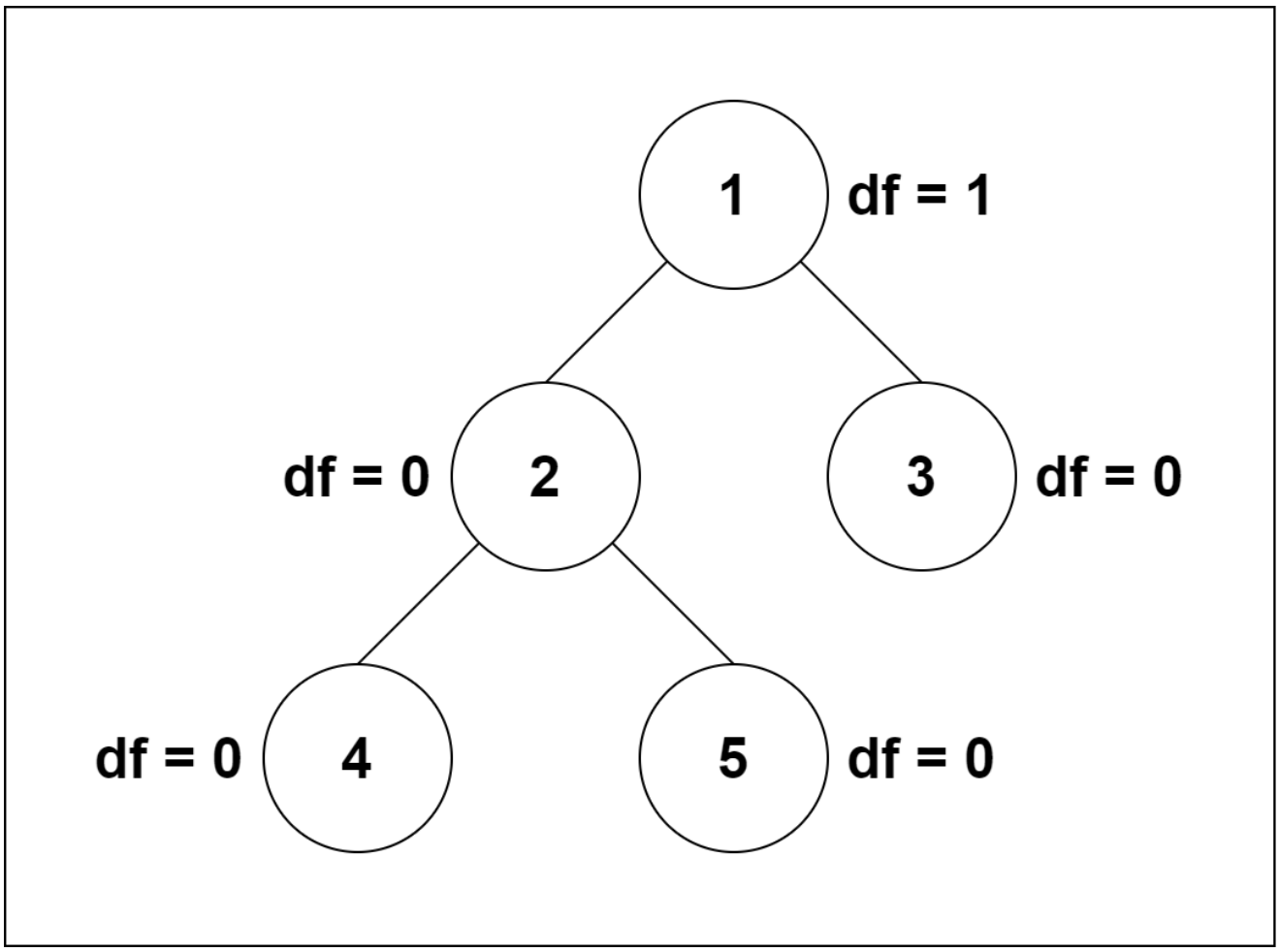
5. Skewed Binary Tree

Imagine a family where either everyone leans to the left or to the right in photos. In a skewed binary tree, it's similar - it's either all about the left kids or all about the right kids.



6. Balanced Binary Tree

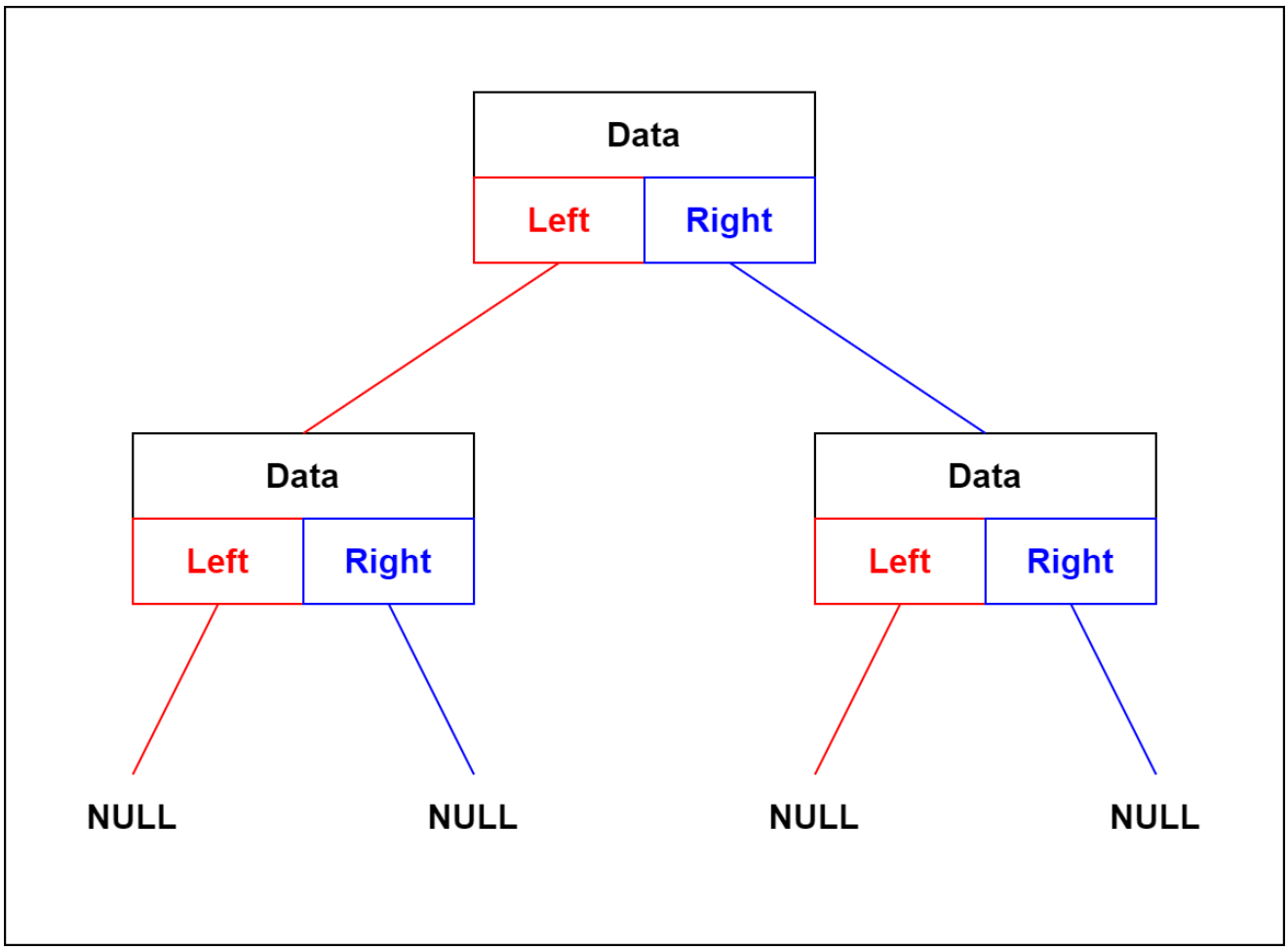
Now, picture a family where everyone is nearly the same height, with no one towering over the others. In a balanced binary tree, the difference in height between the left and right sides of the family tree is very small, either 0 or just 1 level of difference.



Binary Tree Representation

Think of a binary tree like a family tree, where each member has a certain role. In our computer world, a family member is represented as a 'node,' and they have three important attributes:

```
struct node {
 int data;
 struct node *left;
 struct node *right;
};
```



1. **Data**: This is like the name or information about that family member. It could be any kind of data we want to store, like numbers or words.
2. **Left Pointer**: Imagine that each family member can have children. This 'Left Pointer' is like a link to the left child, another family member who is on the left branch of the family tree.
3. **Right Pointer**: Similarly, our family member might also have a child on the right side of the family tree. So, the 'Right Pointer' is like a link to that child.

So, with this structure, we can create a binary tree where each 'node' contains some information ('data') and points to two other nodes - one on the left and one on the right. This

way, we can build a complex tree structure to represent various types of data and relationships in our computer programs.

C++ Example

```
// Binary Tree in C++

#include <stdlib.h>

#include <iostream>

using namespace std;

struct node {
  int data;
  struct node *left;
  struct node *right;
};

// New node creation
struct node *newNode(int data) {
  struct node *node = (struct node *)malloc(sizeof(struct node));

  node->data = data;

  node->left = NULL;
  node->right = NULL;
  return (node);
}

// Traverse Preorder
void traversePreOrder(struct node *temp) {
  if (temp != NULL) {
    cout << " " << temp->data;
    traversePreOrder(temp->left);
    traversePreOrder(temp->right);
  }
}

// Traverse Inorder
void traverseInOrder(struct node *temp) {
  if (temp != NULL) {
    traverseInOrder(temp->left);
    cout << " " << temp->data;
    traverseInOrder(temp->right);
  }
```

```
}

// Traverse Postorder
void traversePostOrder(struct node *temp) {
  if (temp != NULL) {
    traversePostOrder(temp->left);
    traversePostOrder(temp->right);
    cout << " " << temp->data;
  }
}

int main() {
  struct node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);

  cout << "preorder traversal: ";
  traversePreOrder(root);
  cout << "\nInorder traversal: ";
  traverseInOrder(root);
  cout << "\nPostorder traversal: ";
  traversePostOrder(root);
}
```

Binary Tree Applications

Let's talk about where we use binary trees:

1. **Quick Data Access**: Binary trees are like efficient filing systems. They help us quickly find the data we need. Imagine you have a lot of books, and you want to find one specific book on a shelf. Binary trees help us organize and locate data just like finding that book on a shelf without checking every book.
2. **Router Algorithms**: In the world of computer networks, like the internet, routers are like traffic directors. They decide how data packets should move. Binary trees are used in router algorithms to help them make those decisions quickly and efficiently, routing your internet traffic effectively.
3. **Heap Data Structure**: Binary trees are also used to create what we call a "heap." A heap is a data structure that helps us keep track of the largest or smallest item in a collection. It's like always knowing who the tallest or shortest person in a group is without measuring everyone. Binary trees make this happen efficiently.
4. **Syntax Trees**: When you write code, like a program or a webpage, it needs to be checked for errors and understood by computers. Syntax trees, which are essentially binary trees, help in this process. They break down your code into smaller parts,

making it easier for computers to analyze and understand. Think of it as making sure all the sentences and words in a book follow the rules of grammar and make sense.

So, binary trees are like the helpers that make many things in the world of computers run smoothly and efficiently, from finding data quickly to making sure your internet works well and even checking your code for errors.