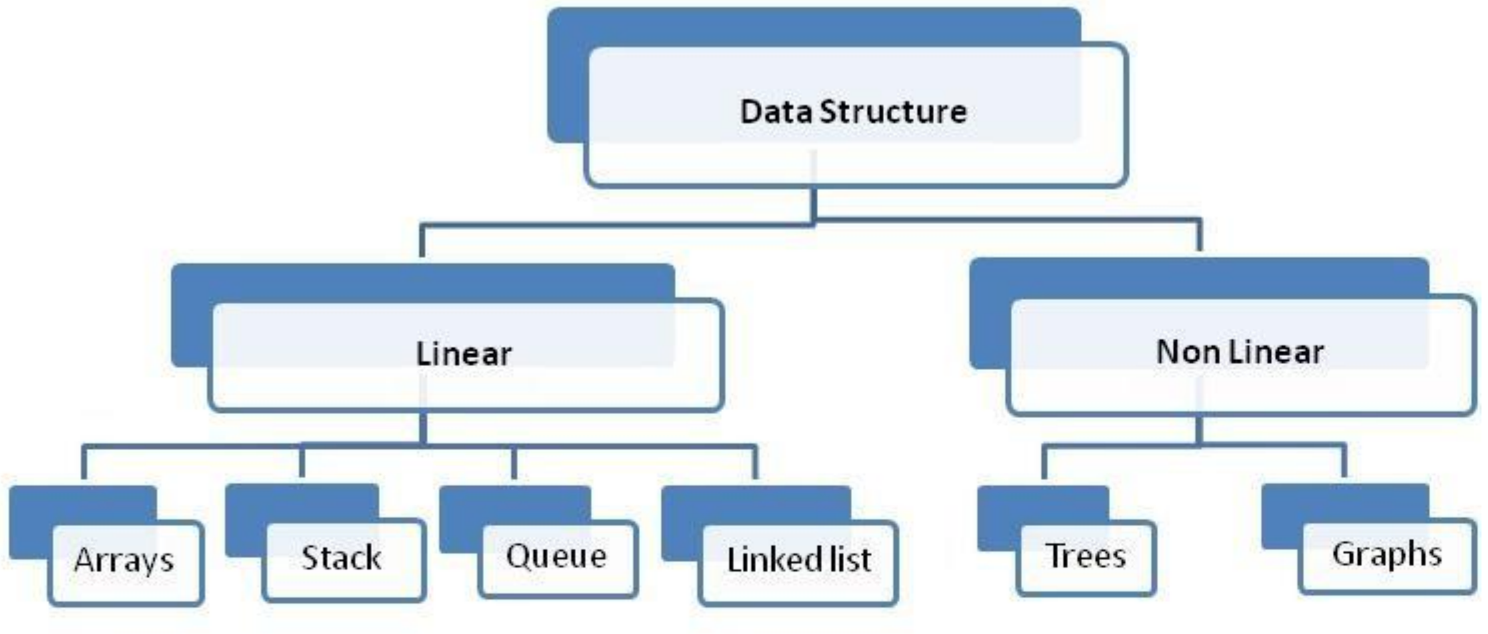
SCS 1308 – FOUNDATION OF ALGORITHMS

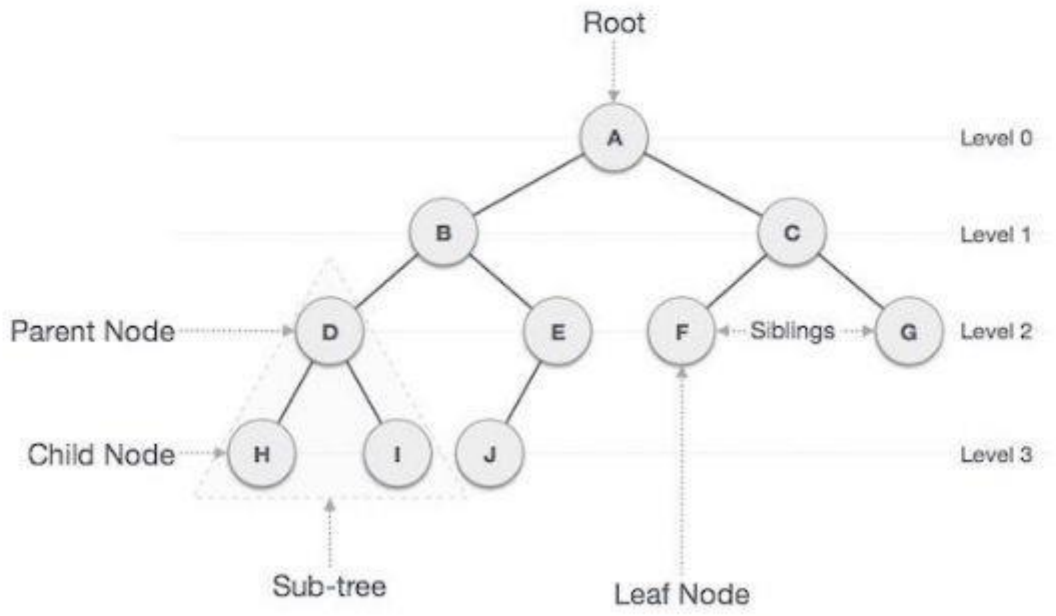
Tutorial - 12

Linear and Non Linear Data Structures



Trees

A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links.



Important Terms in Trees

- **Path** –Path refers to the sequence of nodes along the edges of a tree.
- **Root** –The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** –Any node except the root node has one edge upward to a node called parent.
- **Child** –The node below a given node connected by its edge downward is called its child node.
- **Leaf** –The node which does not have any child node is called the leaf node.
- **Subtree** –Subtree represents the descendants of a node.
- **Levels** –Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

Types of Trees

There are three types of trees –

- **General Trees**
  General trees are unordered tree data structures where the root node has minimum 0 or maximum ‘n’ subtrees.

- **Binary Trees**
  Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree and right subtree.

- **Binary Search Trees**
  In the BST values of left subtree are always less than the values in the root node and the values in the right subtree and values in right subtree are always greater than the values in the root node

Binary Trees

Full Binary Tree

- A full binary tree is a binary tree type where every node has either 0 or 2 child nodes. Leaf nodes are on same level

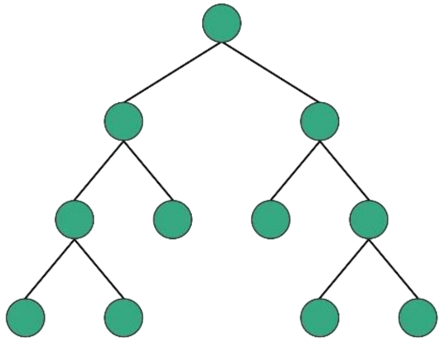
Complete Binary Tree

- A complete binary tree is a binary tree type where all the leaf nodes must be on the same level. However, root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes.

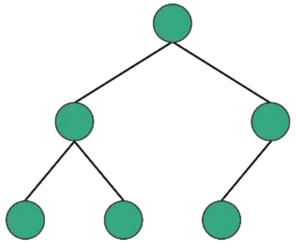
Perfect Binary Tree

- A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.

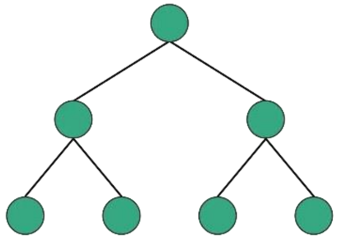
Binary Trees



Full



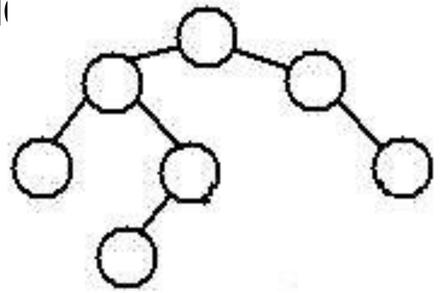
Complete

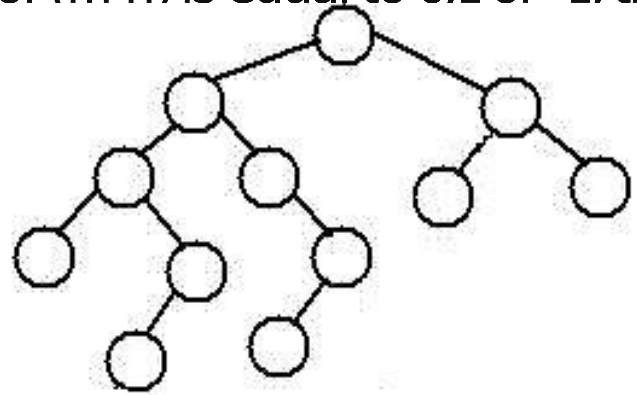


Perfect

Height Balanced Binary Trees

The height of a tree is defined as the number of edges on the longest path from the root node to a leaf node. Alternatively, it is the maximum depth of the tree, where the depth of a node is the distance from the root to that node. Consider a Binary Search Tree with ‘m’ as the height of the left subtree and ‘n’ as the height of the right subtree. If the value of (m-n) is equal to 0,1 or -1, the tree is sai


A height-balanced Tree


Not a height-balanced tree

Local vs Global Balancing

Balancing in trees refers to the process of ensuring that the height difference between the left and right subtrees of every node in the tree remains minimal, typically to optimize search, insertion, and deletion operations.

Global Balancing

Aims to balance the entire tree structure, often achieving an optimal or near-optimal balance. Complete restructuring of the tree is done periodically, rather than after every insertion or deletion.

Eg: DSW trees

Local Balancing

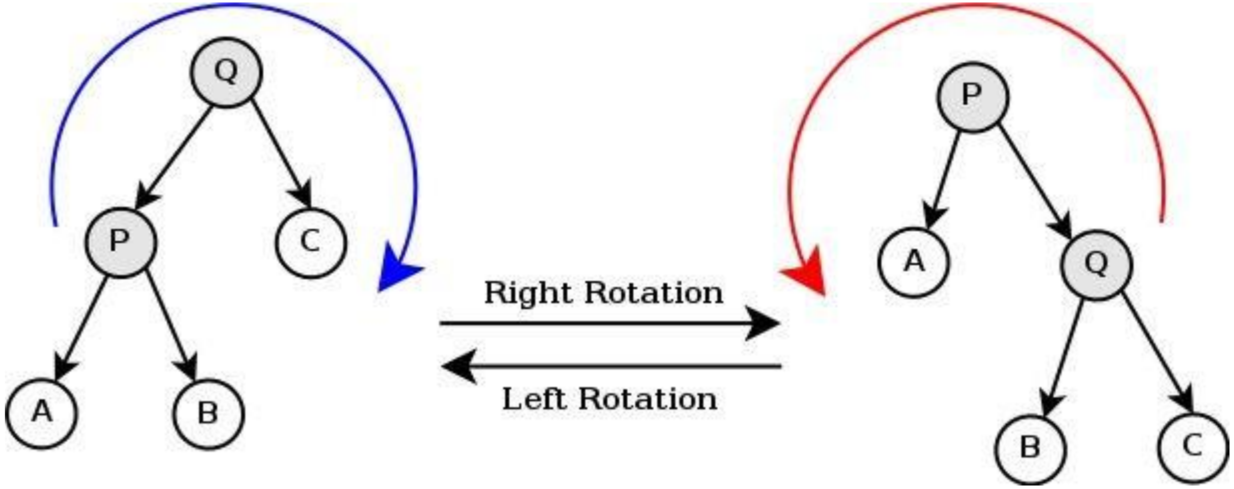
Focuses on maintaining balance around individual nodes or small sub-trees. Balancing operations (like rotations) are performed more frequently, typically after each insertion or deletion.

Eg: AVL trees, Red Black Trees

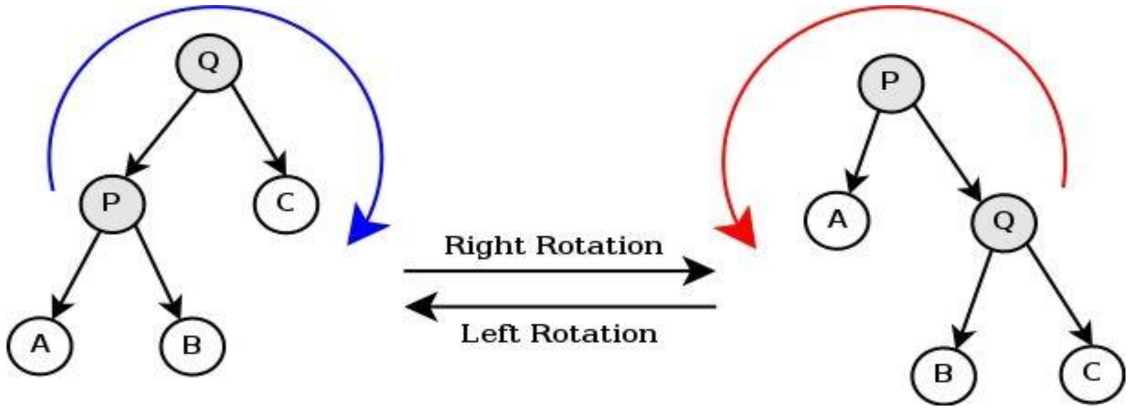
Tree Rotations

The tree balancing algorithms or techniques makes extensive use of rotations (rotating a node about another node). Tree rotations can be done either a left rotation or a right rotation.

Eg: Let's rotate around root node.

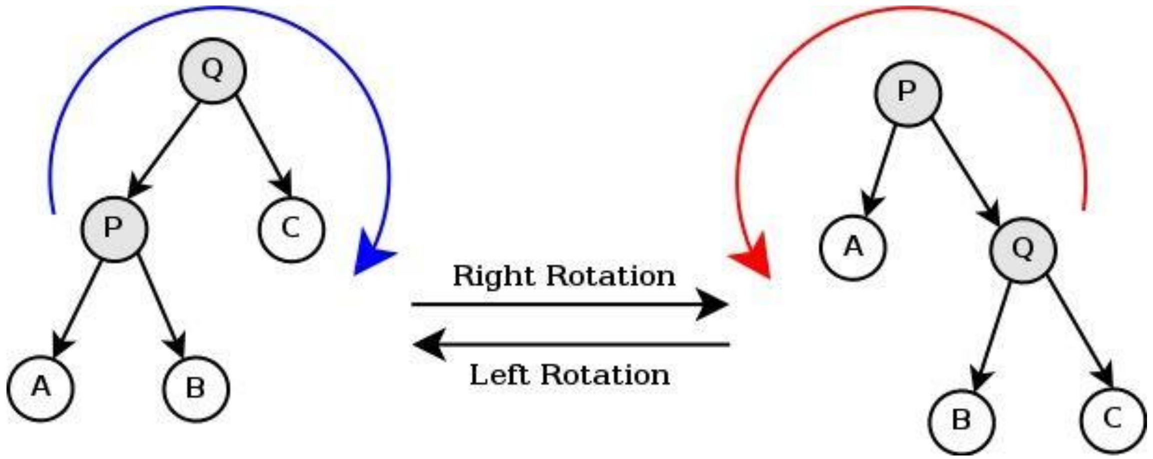


Right Rotation



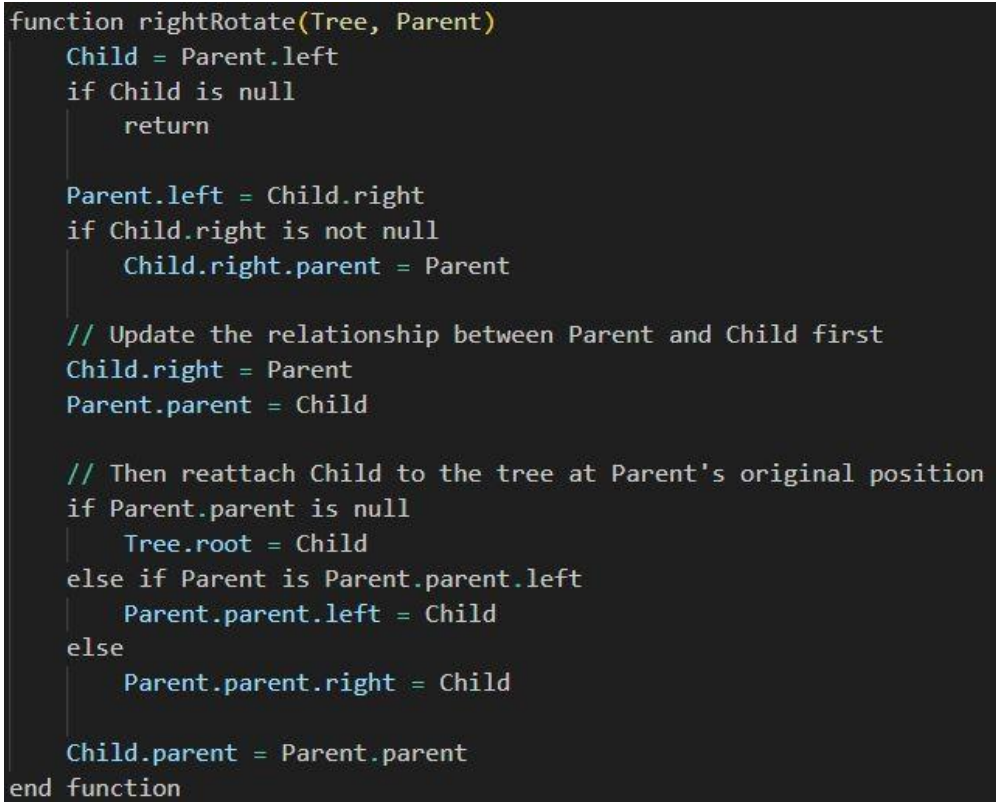
- This is a transformation that moves left nodes to right. In the provided tree, node P is rotated around Q to take the place of node Q, while Q moves down to become P's right child.
- The subtree rooted at A remains unchanged.
- If P had a right child (in this case, B), it becomes the left child of Q.
- Right rotations are used in various self-balancing trees to maintain the tree's balance after insertions and deletions that skew the tree to the right.

Left Rotation



- This is the opposite transformation, where moves right nodes to left. Here, node P is rotated to become the left child of node Q.
- The subtree rooted at C remains unchanged.
- If Q had a left child before the rotation (which would be B in the case of a reverse of the right rotation), it becomes the right child of P.
- Left rotations are used to balance trees that are skewed to the left.

Right Rotation Algorithm



```
function rightRotate(Tree, Parent)
    Child = Parent.left
    if Child is null
        return

    Parent.left = Child.right
    if Child.right is not null
        Child.right.parent = Parent

    // Update the relationship between Parent and Child first
    Child.right = Parent
    Parent.parent = Child

    // Then reattach Child to the tree at Parent's original position
    if Parent.parent is null
        Tree.root = Child
    else if Parent is Parent.parent.left
        Parent.parent.left = Child
    else
        Parent.parent.right = Child

    Child.parent = Parent.parent
end function
```

DSW(Day-Stout-Warren) Trees

The Day-Stout-Warren (DSW) algorithm isn't used to create a specific type of tree called a **"DSW tree**," but rather it's a method for **balancing binary search trees**. This algorithm transforms an unbalanced binary search tree into a balanced one

DSW algorithm is one of the way to balance a tree globally. Global tree balancing happens when all the operations such as insertions and deletions on a tree is completed. This algorithms always creates trees that are perfectly balanced or close to being perfectly balance.

DSW Trees Contd

The Day-Stout-Warren (DSW) algorithm aims to create a complete binary tree means a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

Unbalanced

```
    15
   /  \
  5    20
 /\      \
3  7      30
         / \
        25 35
       /
      1
```

Balanced Complete Binary Tree

```
      15
     /  \
    5    25
   /\    /\
  3  7  20 30
 /           \
1             35
```

DSW Tree

```
       15
      /   \
     5     25
    / \    /\
   3   7  20 30
  / \
 1  35
```

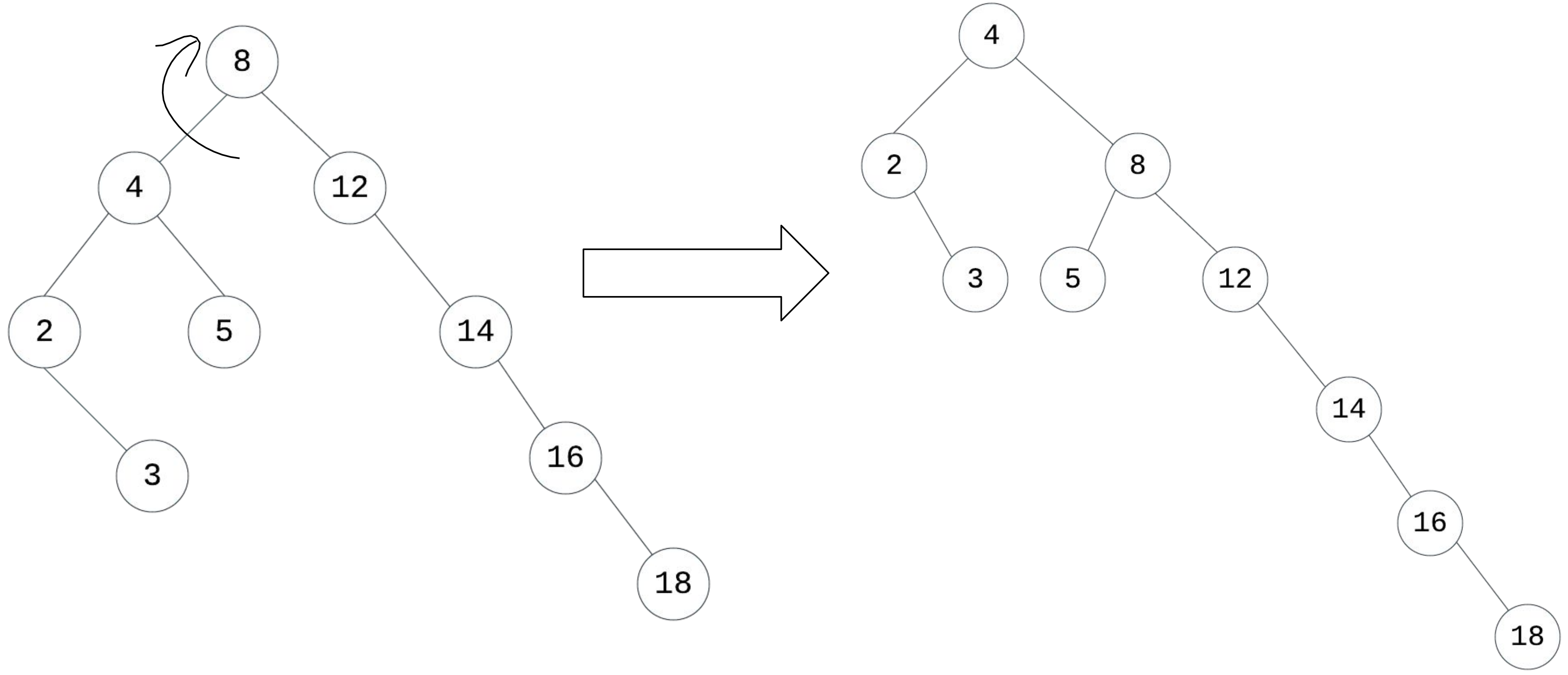
DSW Tree Balancing Approach

The DSW algorithm has two major phases and each phase is dominated by tree rotations.

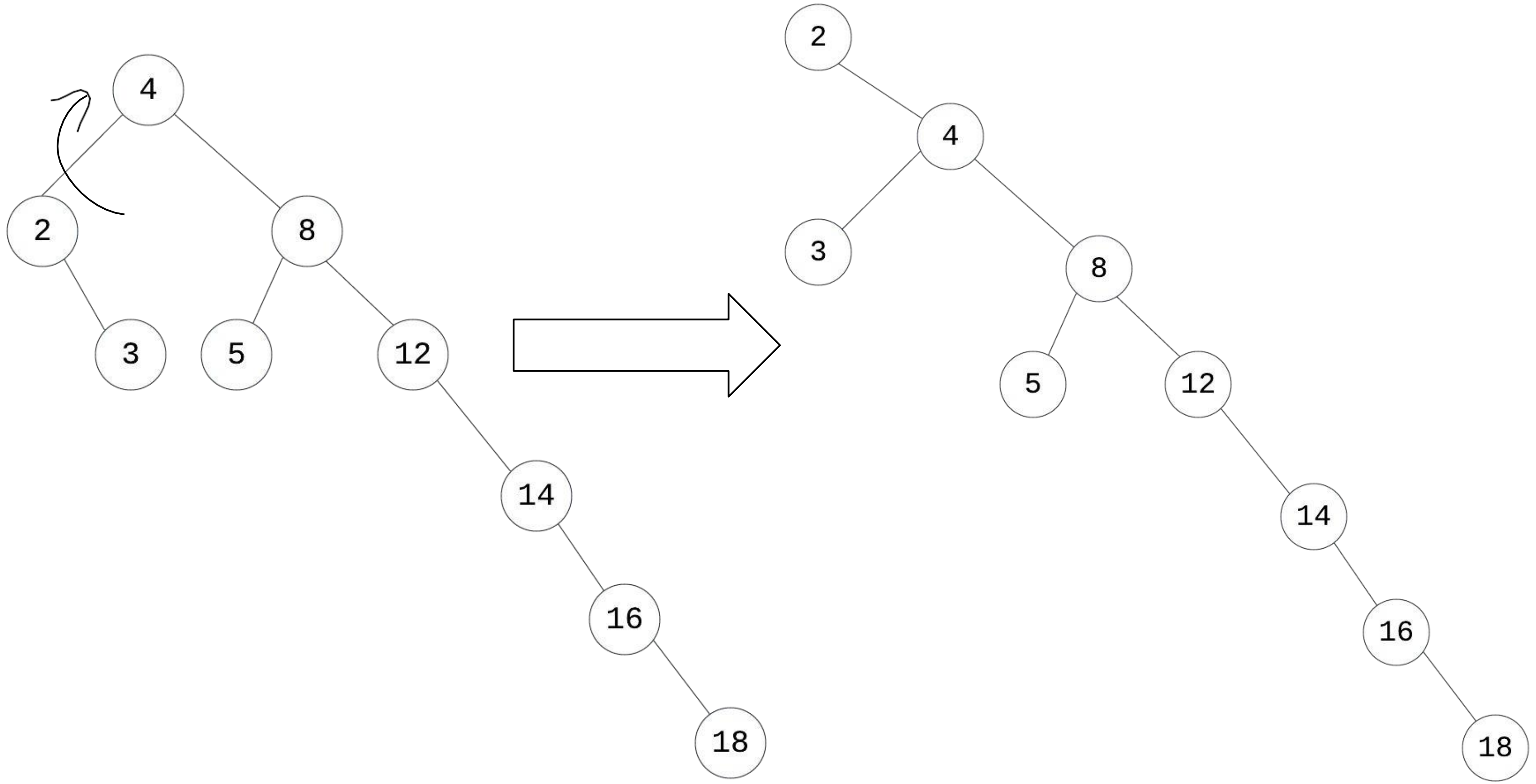
1. **Create a Backbone:**

   This is the initial step of the DSW algorithm. What it does is that it transforms a given tree in to a linked list like tree which is called the backbone or a vine. This is backbone is right vine so it will right rotate left subtree.

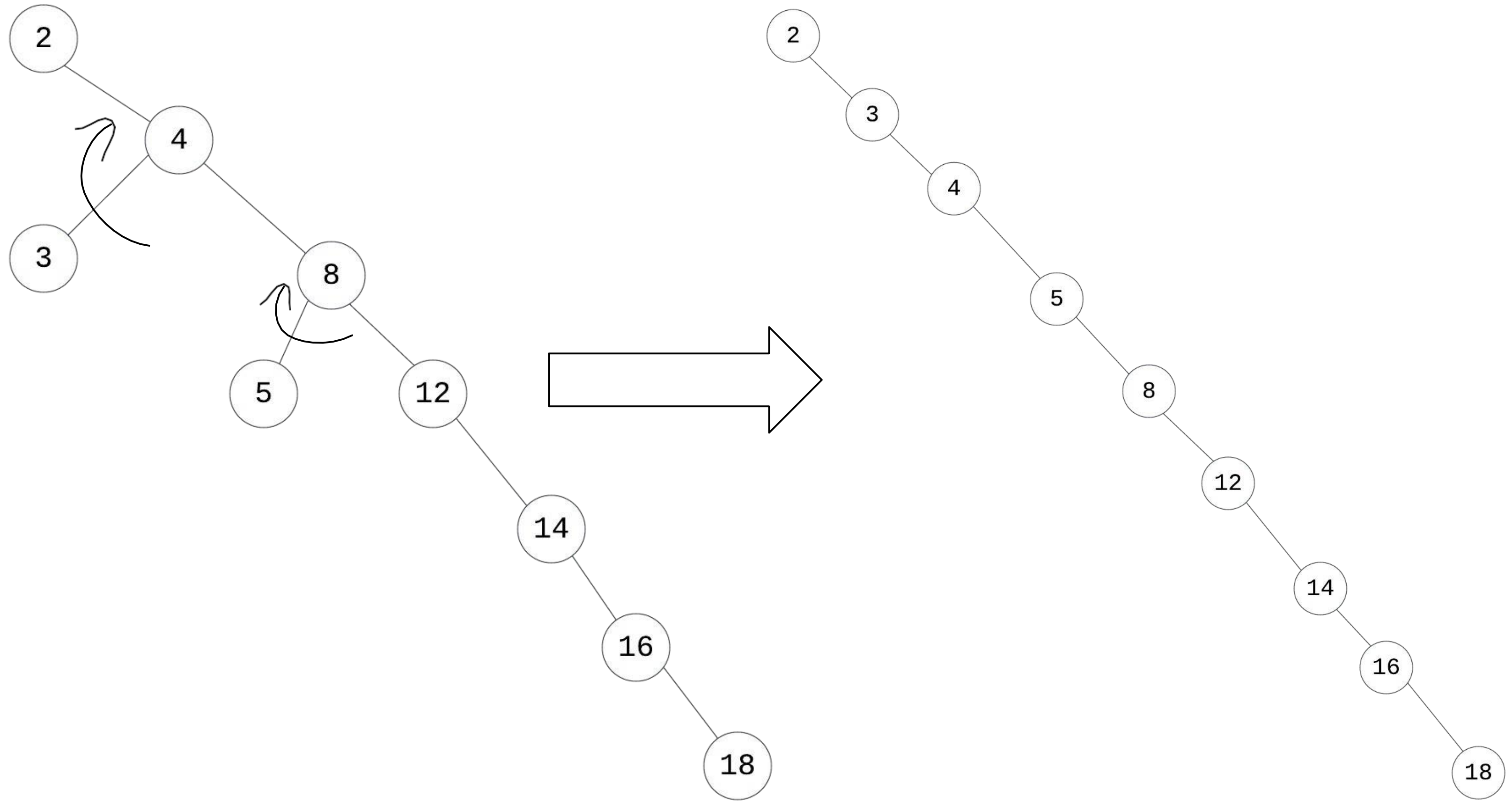
DSW Tree Balancing Approach



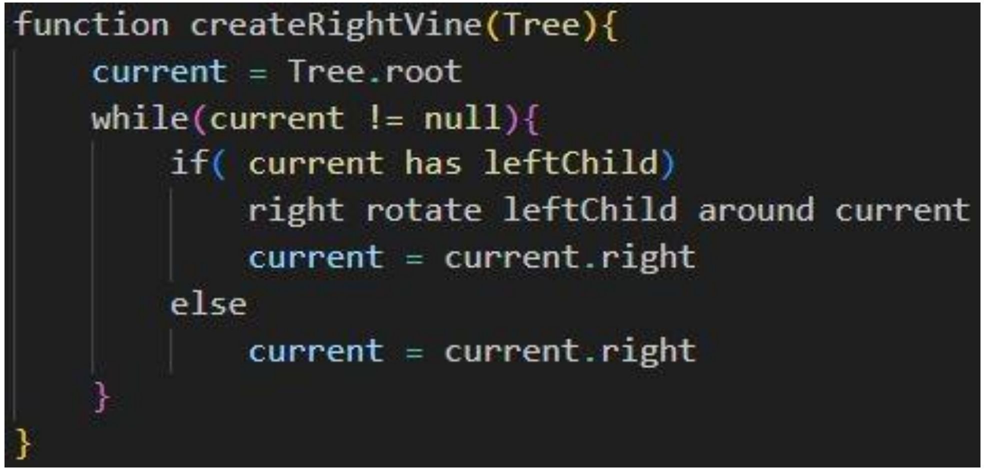
DSW Tree Balancing Approach



DSW Tree Balancing Approach



Creating a Right vine Algorithm



```
function createRightVine(Tree){
    current = Tree.root
    while(current != null){
        if( current has leftChild)
            right rotate leftChild around current
            current = current.right
        else
            current = current.right
    }
}
```

DSW Tree Balancing Approach Contd

2. Balancing the Tree

Now you have created the backbone next as the final phase you have to converts the right vine created in the initial phase as the next step we need to find number of nodes we need to build a perfectly balanced tree.

A perfectly balanced tree represents an exact triangle. Which means a relationship can derived between the height (h) of the perfectly balanced tree and the total number of nodes (n) in the tree.

$$n = 2^h - 1$$
$$h = \lg_2(n+1)$$

DSW Tree Balancing Approach Contd

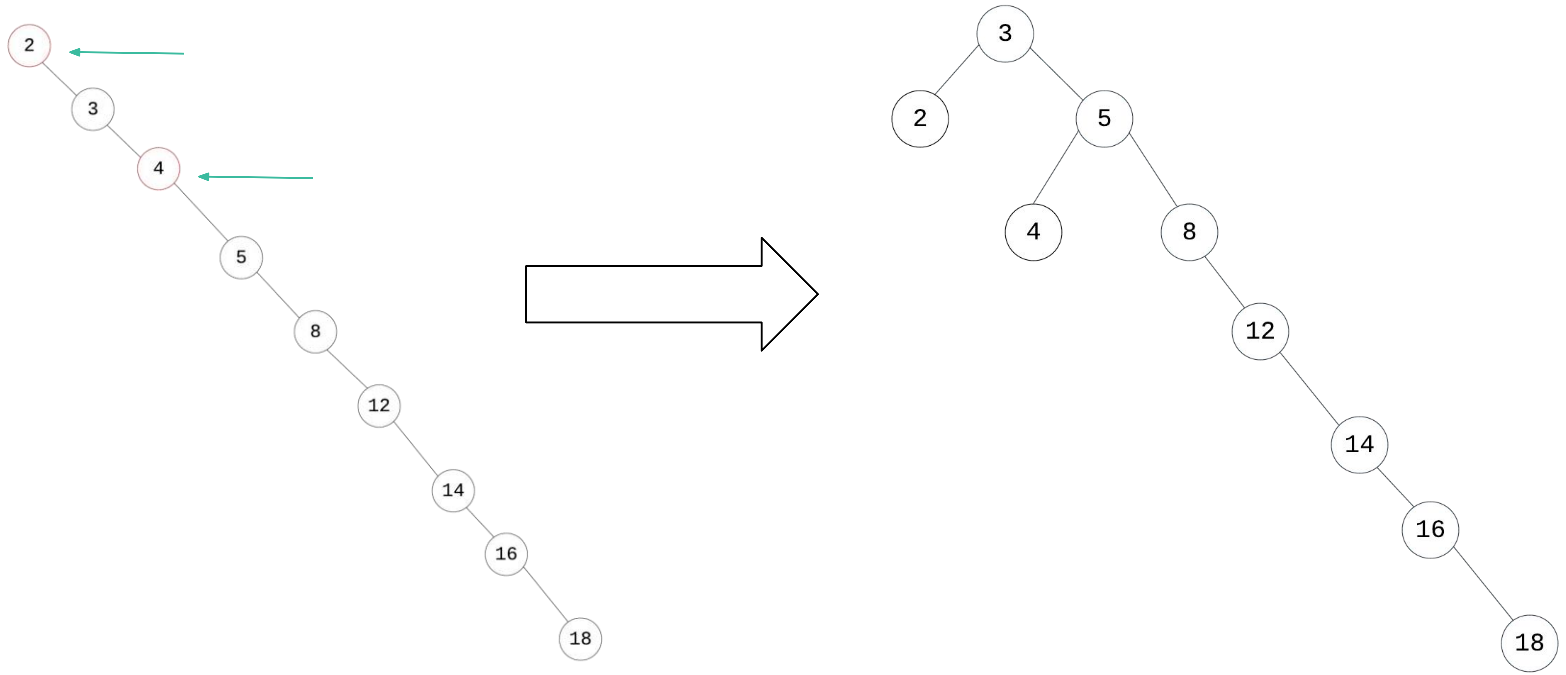
Since the vine in above figure contains 9 nodes the closest height of a perfectly balanced tree is 3 (lg2(10)~3). So the number nodes (m) in a perfectly balanced tree of height 3 as per the 1st equation

n = number of nodes in the tree (6 nodes)
m = number of nodes in the closest perfectly balanced tree (7 nodes)

So the node difference between the perfect tree and the tree to be balanced can be calculated by n-m which results in 2. This is called the **excess of nodes** to the perfectly balanced tree. In order to balance this excess nodes left rotation is performed on **n-m** odd nodes in the vine.

DSW Tree Balancing Approach Contd

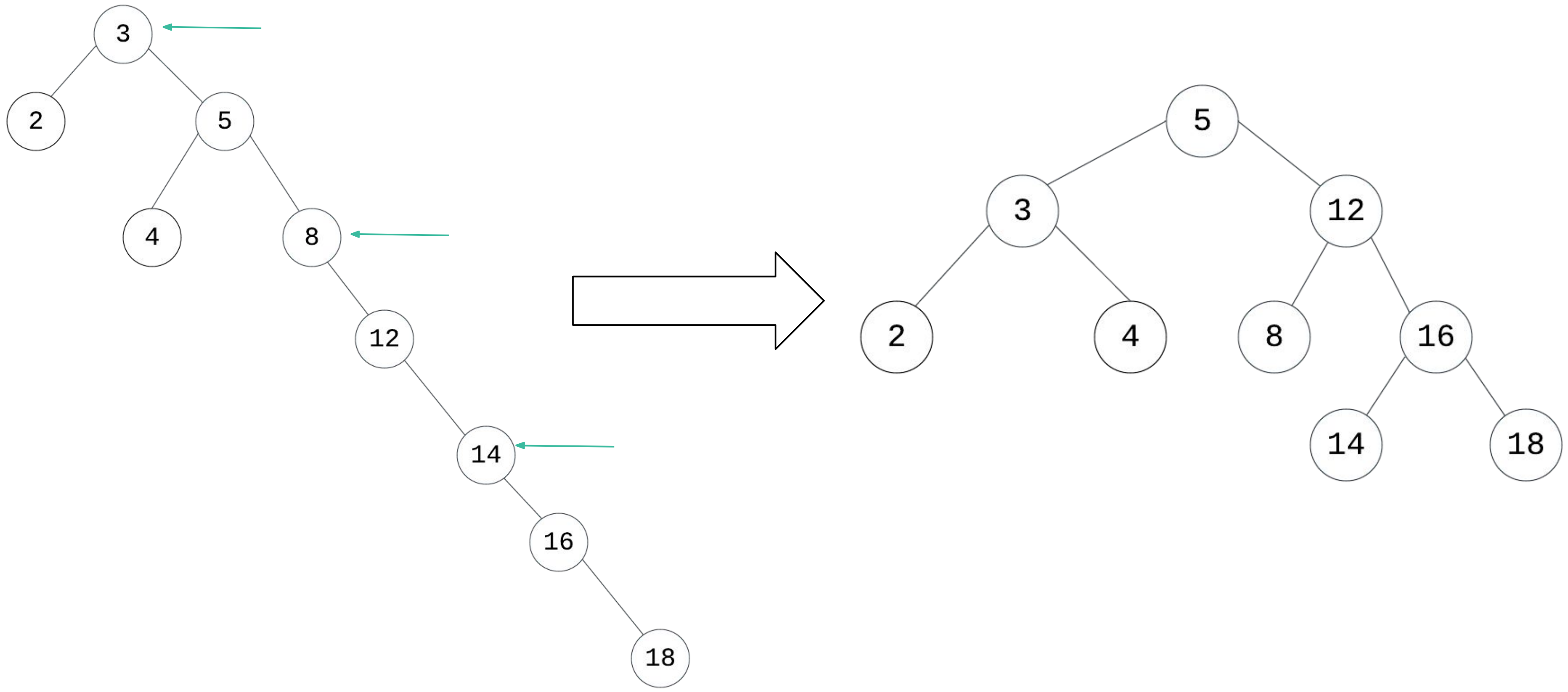
When we consider above example two left rotations**(n-m)** would be done initially. Since we are selecting odd nodes, it will be done on 1st and 3rd node according to above example it is on values 2 and 4



DSW Tree Balancing Approach Contd

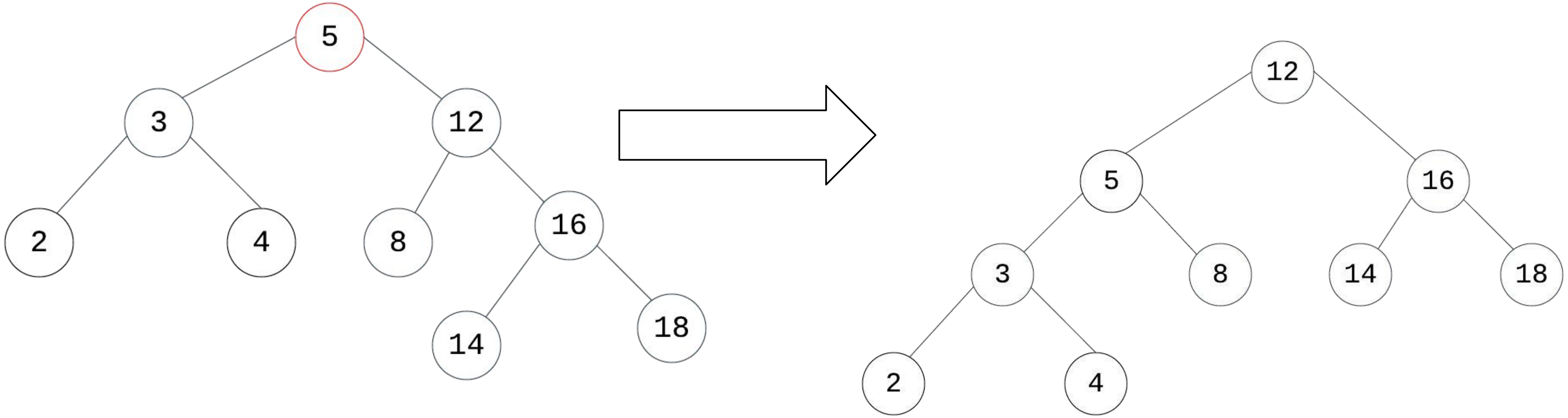
- After left rotation of excess nodes next you have to half the size of the vine through each pass. Number of rotations to be performed is calculated by m which is halved in each iteration of the loop as **m = m/2** (only take the integer value). So to start the final phase we recalculate m which is (7/2 =3). So 3 left rotations are performed to the vine

DSW Tree Balancing Approach Contd



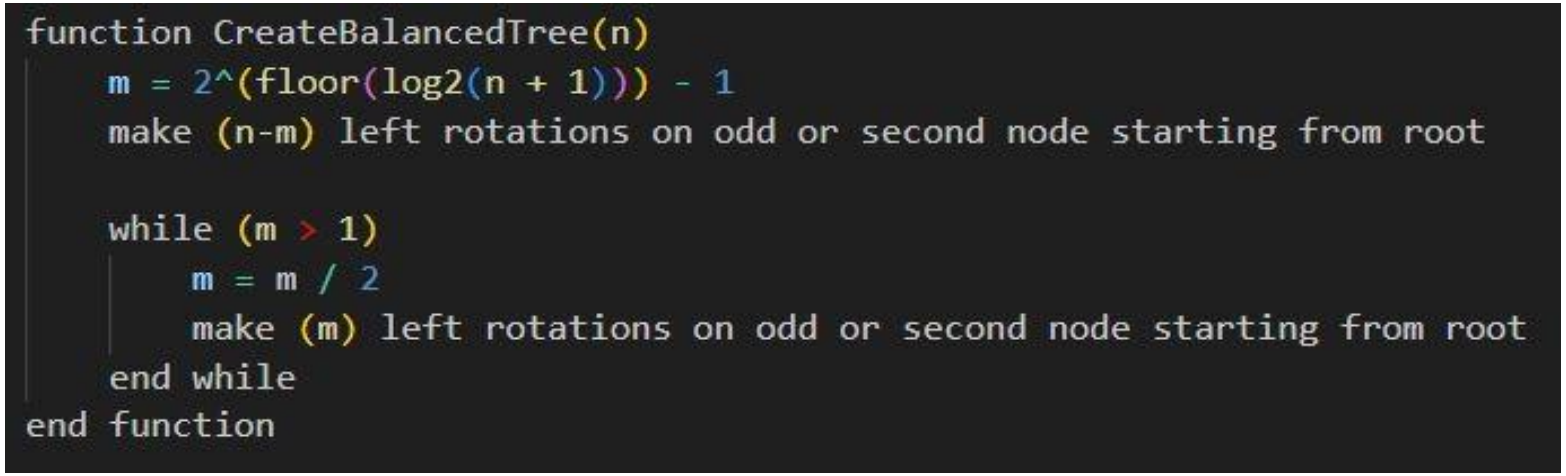
DSW Tree Balancing Approach Contd

In the next iteration m further halved to 1. So one left rotation is performed on the one node in the vine which is value 5 in the above example.



If m was further divided the value will be lesser than 1. Therefore the height differences is now less than one or one. Therefore the tree has attained a balanced state. So the tree in figure 8 is the final globally balanced tree.

Balancing the tree Algorithm



```
function CreateBalancedTree(n)
    m = 2^(floor(log2(n + 1))) - 1
    make (n-m) left rotations on odd or second node starting from root

    while (m > 1)
        m = m / 2
        make (m) left rotations on odd or second node starting from root
    end while
end function
```