Деревья

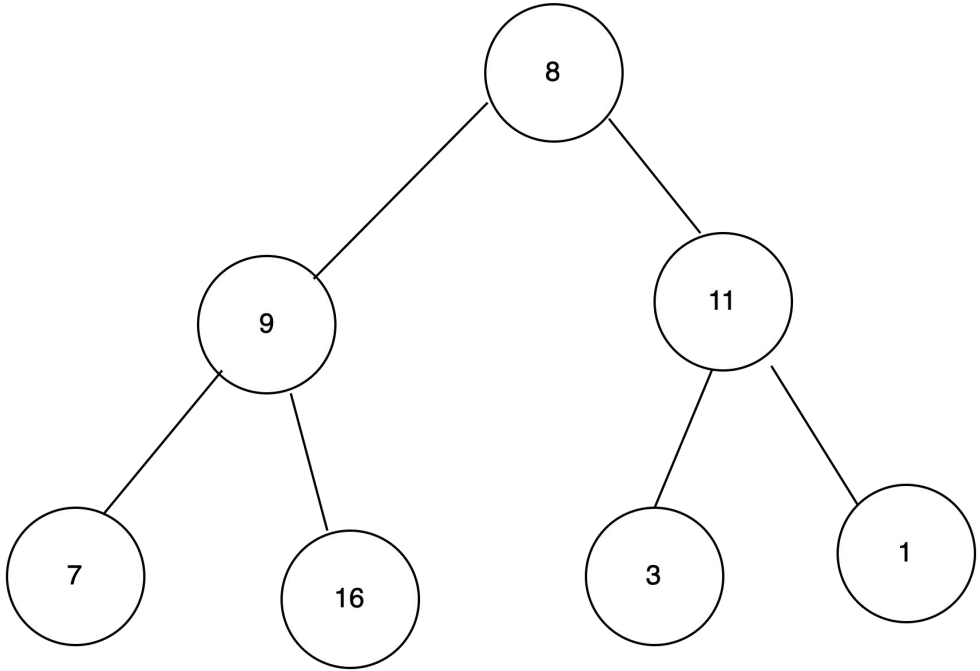
Семинар

Восстановление бинарного дерева из массива
Проверка на симметричность бинарного дерева
Поиск минимальной глубины
Произведение минимального и максимального значения
Сравнение двух деревьев

Восстановить бинарное дерево из массива

8	9	11	7	16	3	1

Необходимо реализовать функцию, которая будет принимать на вход массив и выстраивать из него бинарное дерево.



Восстановить бинарное дерево из массива

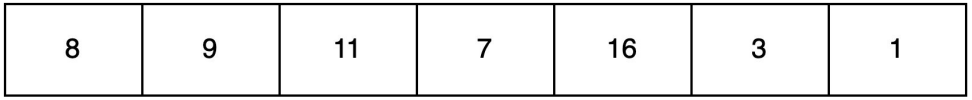
```
class TreeNode:
    function TreeNode(val = 0, left = null, right = null) {
        this.data = val
        this.left = left
        this.right = right
    }

function buildTree(arr) {
    // определяем базовые кейсы
    // создаем узел
    // добавляем ему левого потомка
    // добавляем ему правого потомка

    return root
}
```

Восстановить бинарное дерево из массива

- необходимо рекурсивно для каждого узла достраивать бинарное дерево
- для этого расширим сигнатуру **buildTree** добавив туда индекс, для которого строим поддерево
- Строить дерево начинаем с корня, то есть с нулевого индекса

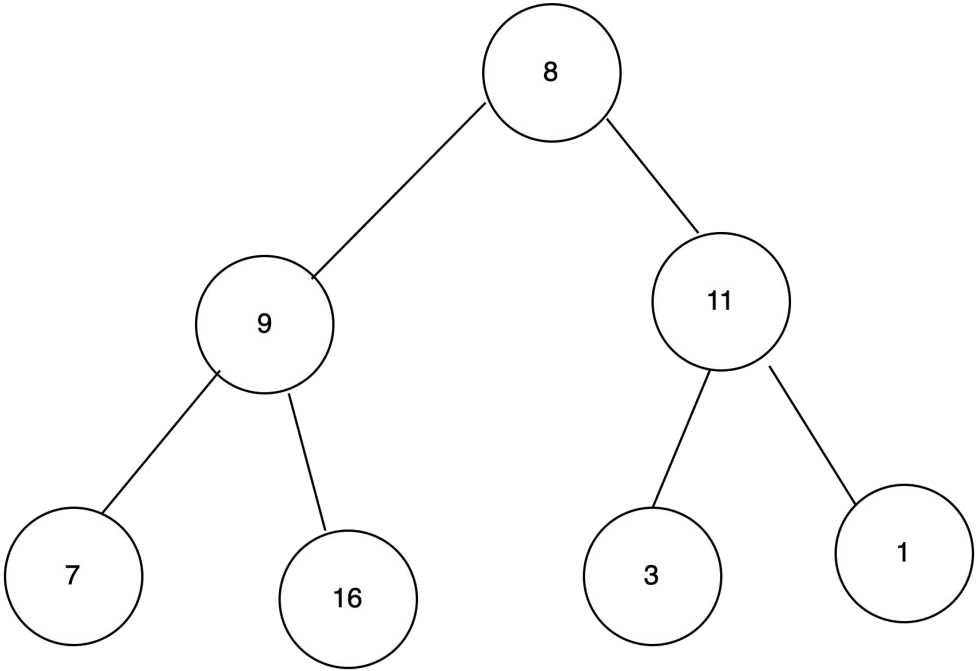
```
function buildTree(arr, i) {
    if i >= len(arr) {
        return null
    }

    root = TreeNode(arr[i])
    // добавляем ему левого потомка 2*i + 1
    // добавляем ему правого потомка 2*i + 2

    return root
}
```

Восстановить бинарное дерево из массива



- Рекурсивно вызываем buildTree для каждого узла

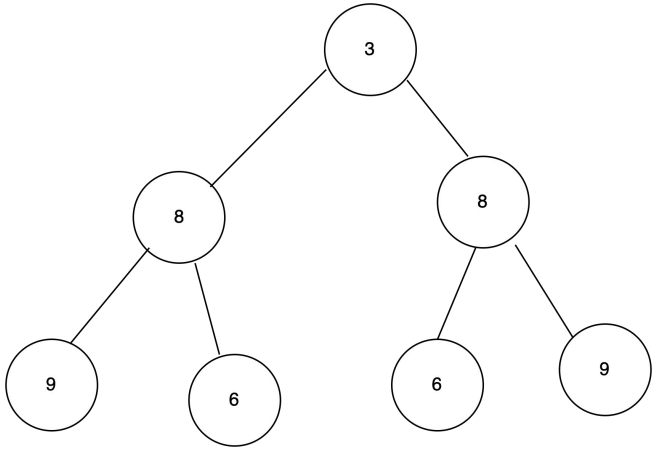
```
function buildTree(arr, i) {
    if i >= len(arr) {
        return null
    }

    root = TreeNode(arr[i])
    root.left = buildTree(arr, 2 * i + 1)
    root.right = buildTree(arr, 2 * i + 2)

    return root
}
```



Симметричное дерево

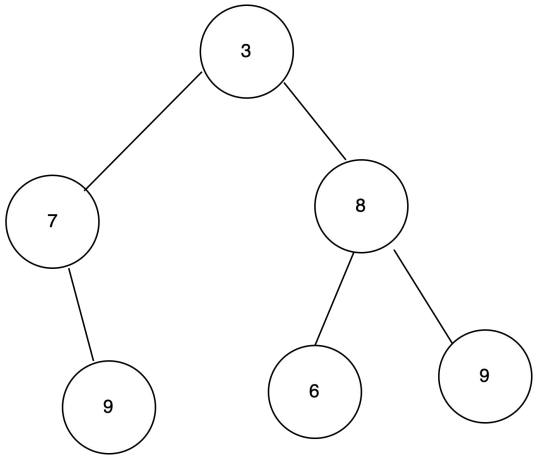
На вход функции подается бинарное дерево.

Необходимо понять, является ли это дерево симметричным



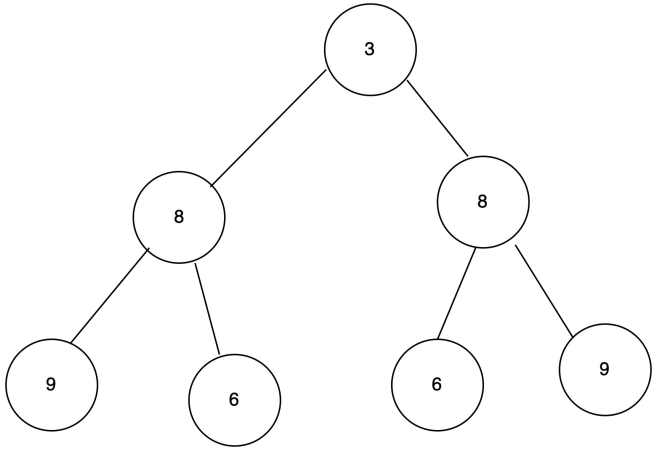
Дерево сверху - симметричное

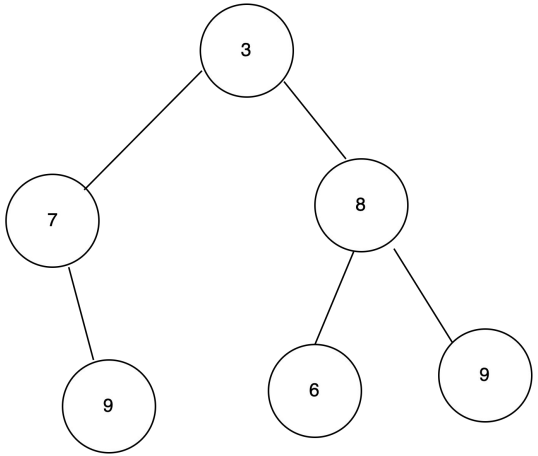
Дерево снизу - нет



Симметричное дерево

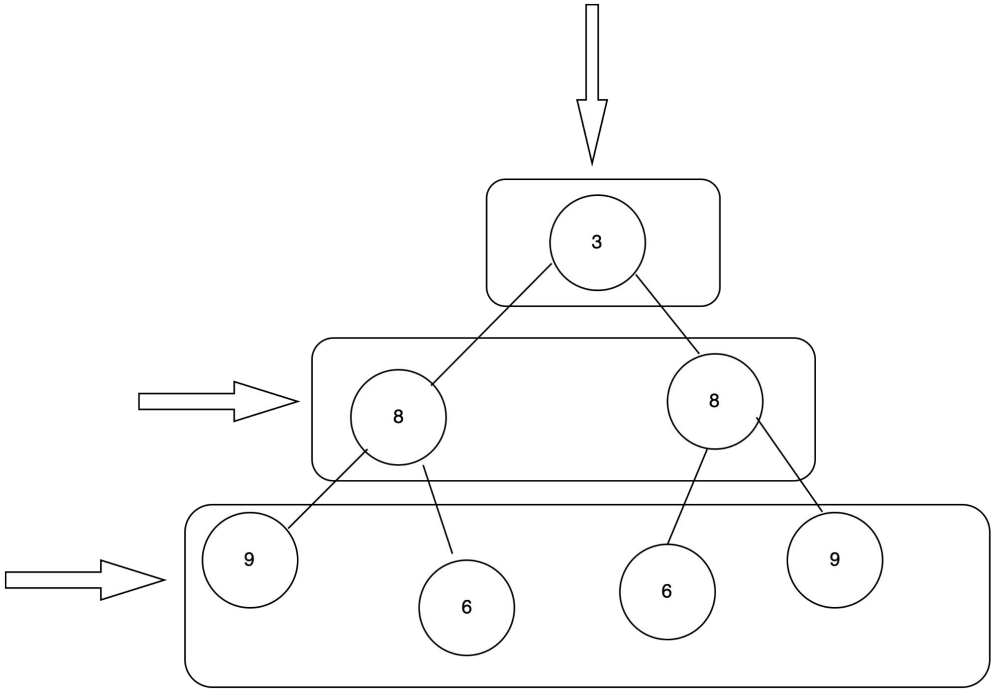
- Обход в ширину
- Можно решить двумя способами
- Итеративно или рекурсивно





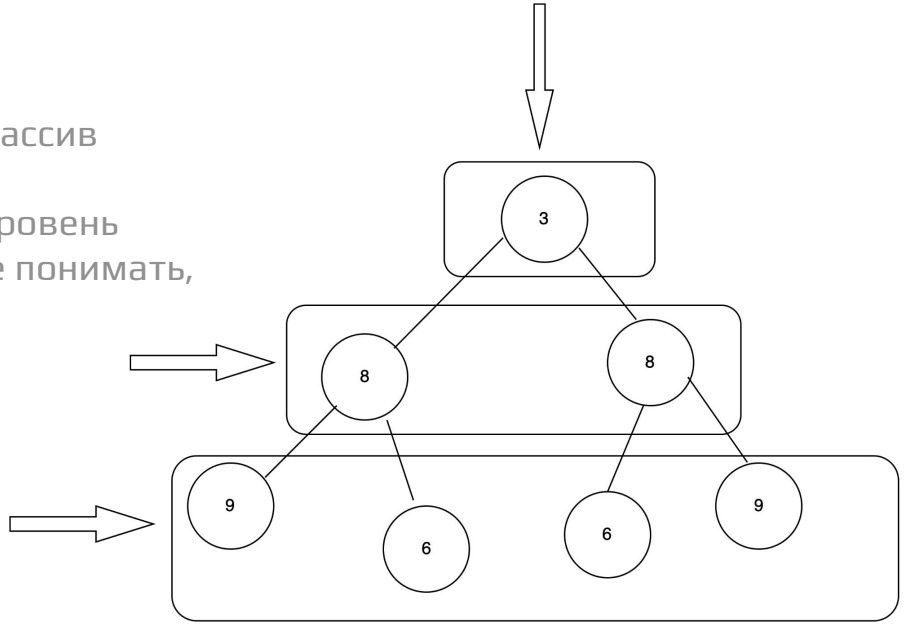
Симметричное дерево BFS

Сверху вниз, слева направо, сверху вниз проходим последовательно все уровни



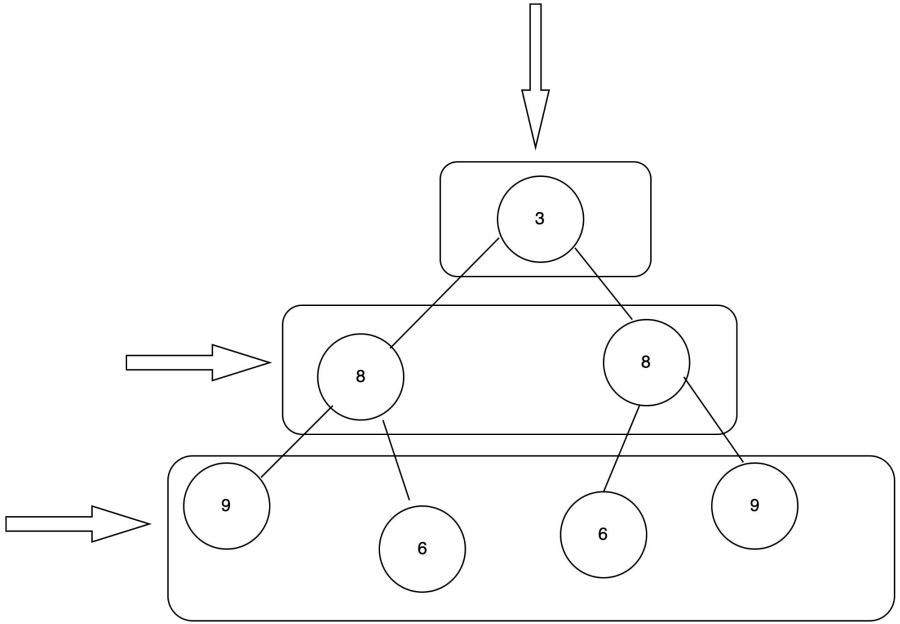
Симметричное дерево
BFS

```
function isSymmetricBFS(bt) {
    // в цикле по каждому узлу собираем потомков в массив
    // сохраняя их последовательность слева направо
    // в результирующем массиве получаем текущий уровень
    // задача сводится к тому, чтобы на каждом уровне понимать,
    // является ли массив симметричным
    ...

    return true
}
```



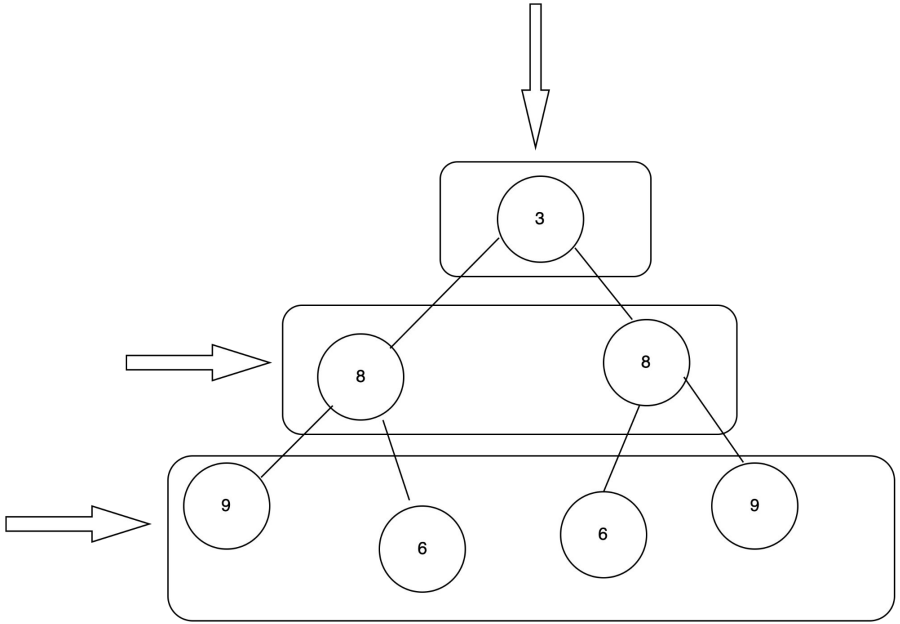
Симметричное дерево BFS

```
function isSymmetricBFS(bt) {
    ...
    // будем решать это посредством two pointers
    ...

    return true
}
```



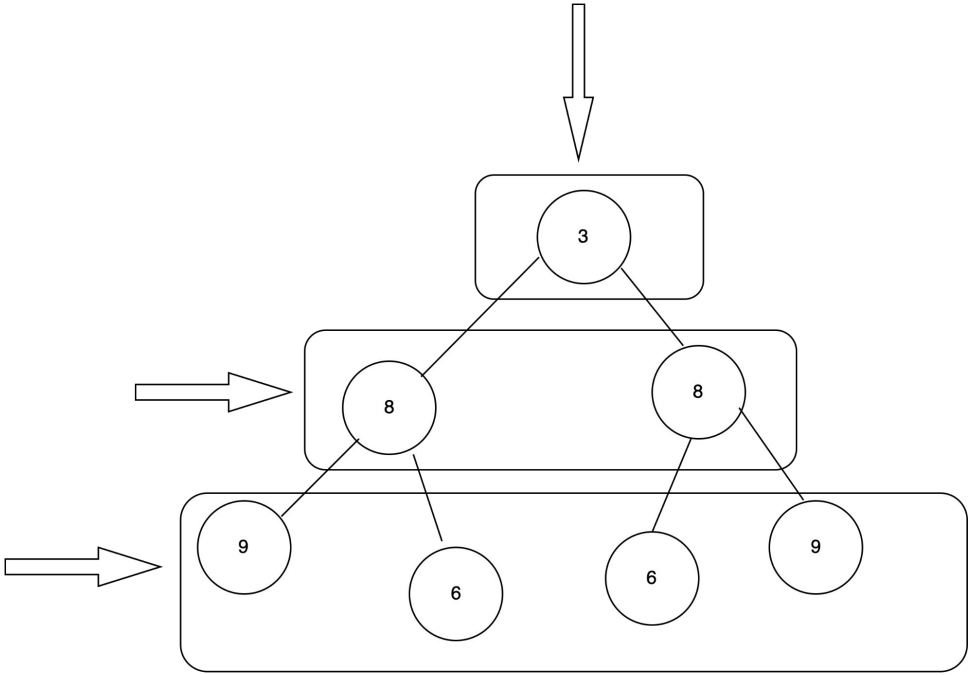
Симметричное дерево
BFS

```
function isSymmetricBFS(bt) {
    // создаем массив узлов дерева
    // чтобы можно было по нему итерироваться
    // на первой итерации в массиве только корень
    // для каждого уровня создаем очередь из узлов
    // в которую будем складывать всех детей
    // текущего уровня. То есть для 2 уровня (8 и 8)
    // в очереди будут хранится узлы 9, 6, 6, 9
    // после того как мы собрали всех детей в очередь
    // проверяем её на симметричность. Если она
    // симметрична - начинаем итерироваться по ней
    ...

    return true
}
```



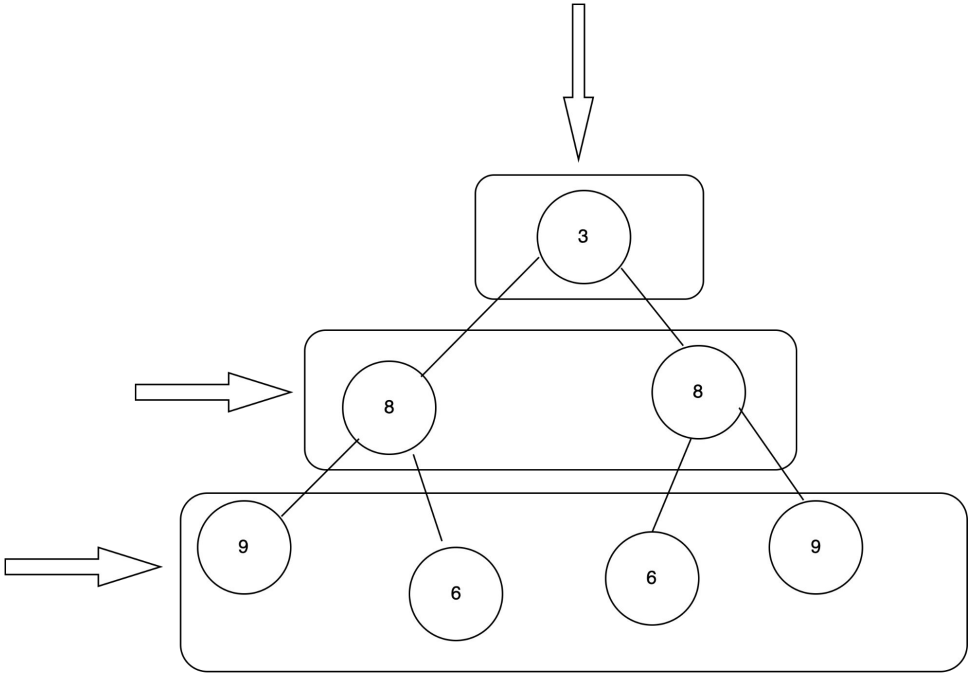
Симметричное дерево BFS

```
function isSymmetricBFS(bt) {
    nodes = [bt]
    while len(nodes) != 0 {
        ...
    }
    ...

    return true
}
```



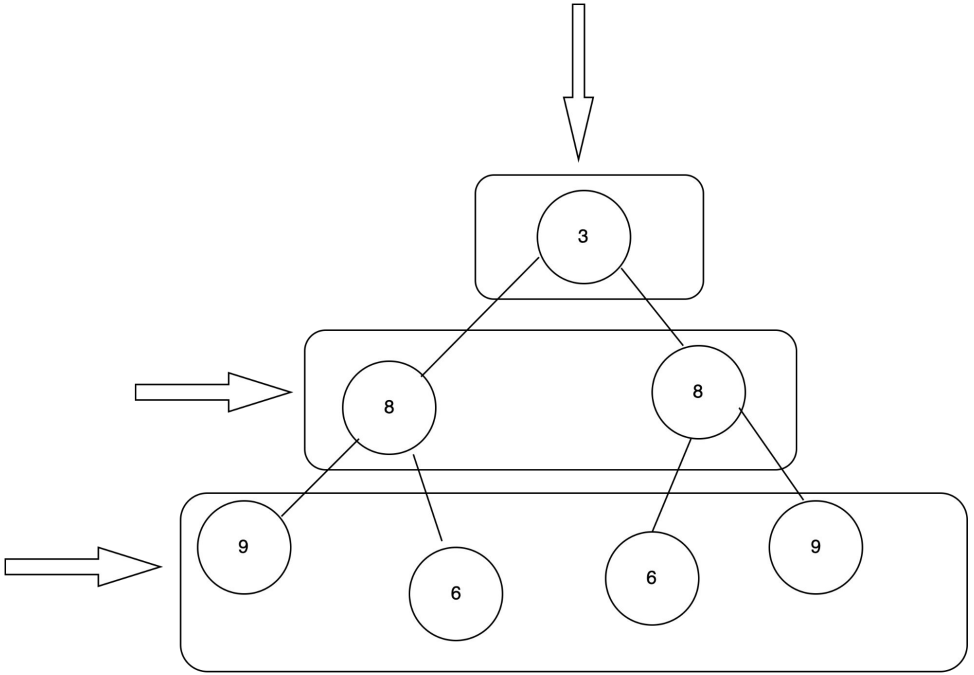
Симметричное дерево BFS

```
function isSymmetricBFS(bt) {
   nodes = [bt]
   while nodes {
      queue = []
      for current in nodes {
         // заполняем очередь
        // слева направо
      }
      nodes = queue
   ...

   return true
}
```



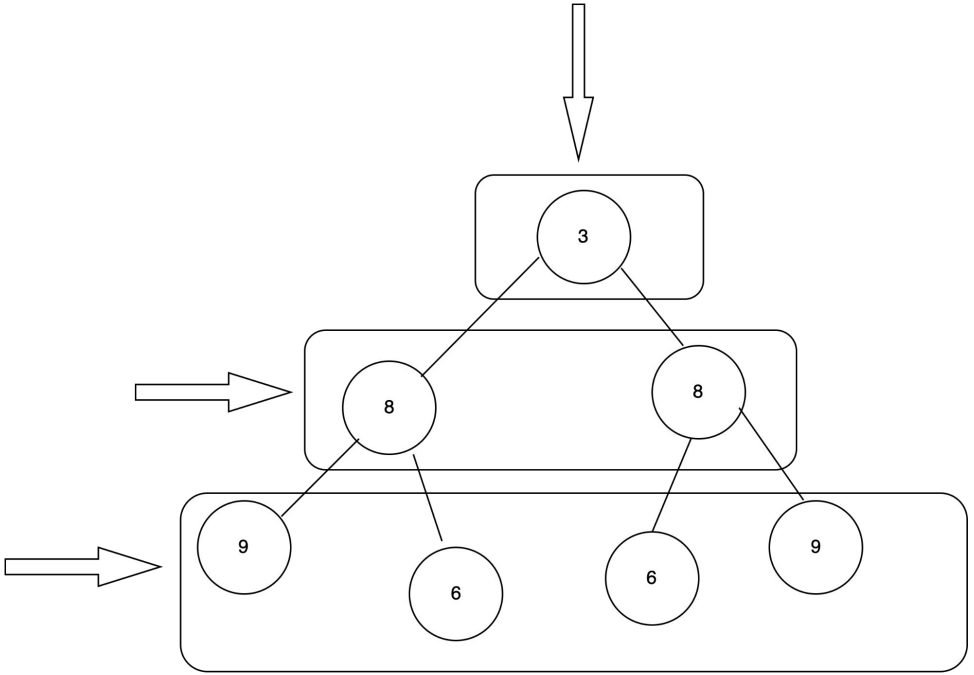
Симметричное дерево BFS

```
function isSymmetricBFS(bt) {
    nodes = [bt]
    while len(nodes) != 0 {
        queue = []
        for current in nodes {
            if current.left {
                queue.append(current.left)
            }
            if current.right {
                queue.append(current.right)
            }
        }
        // проверяем queue на симметричность
        nodes = queue


    return true
}
```



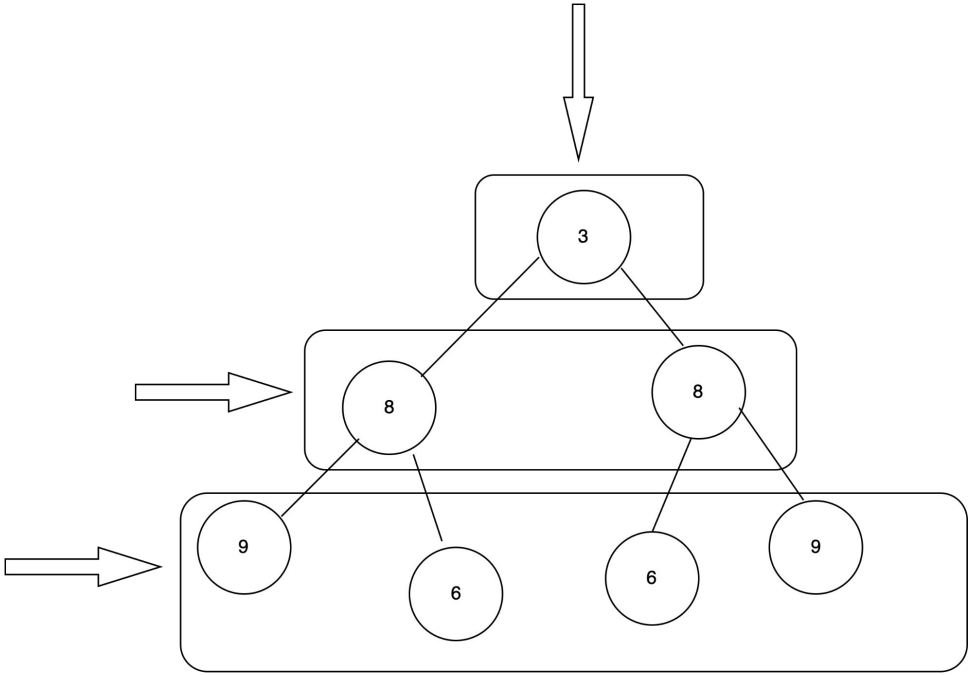
Симметричное дерево BFS

```
function isSymmetricBFS(bt) {
    nodes = [bt]
    while len(nodes) != 0 {
        queue = []
        for current in nodes {
            if current.left {
                queue.append(current.left)
            }
            if current.right {
                queue.append(current.right)
            }
        }
        // в цикле сравниваем нулевой и последний
        // элементы, далее первый и предпоследний
        // пока не дойдем до середины массива

        nodes = queue

    return true
}
```



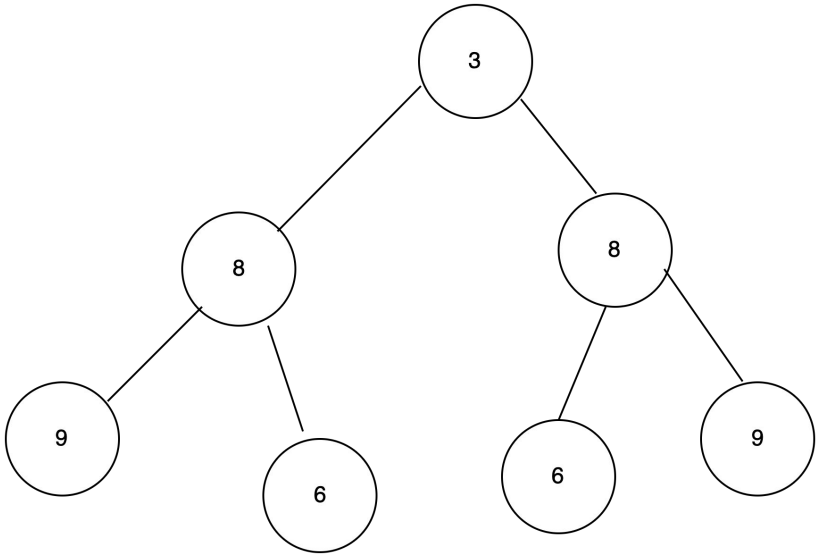
Симметричное дерево BFS

```
function isSymmetricBFS(bt) {
    nodes = [bt]
    while len(nodes) != 0 {
        queue = []
        for current in nodes {
            if current.left {
                queue.append(current.left)
            }
            if current.right {
                queue.append(current.right)
            }
        }
        j = len(queue) - 1
        for i = 0; i < len(queue)/2; i++ {
            if queue[i].data != queue[j].data {
                return false
            }
            j--
        }

        nodes = queue

    return true
}
```



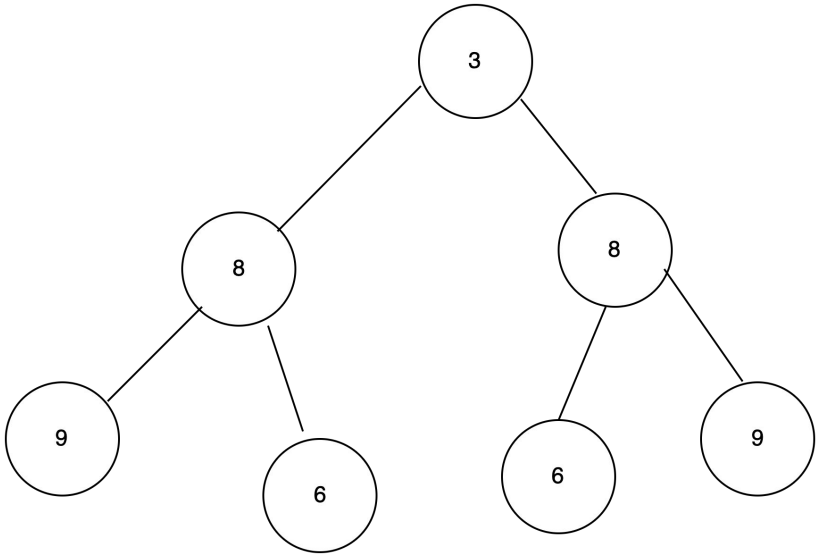
Симметричное дерево DFS

- Какой вариант обхода DFS тут подойдет?
- LNR RNL NLR ...



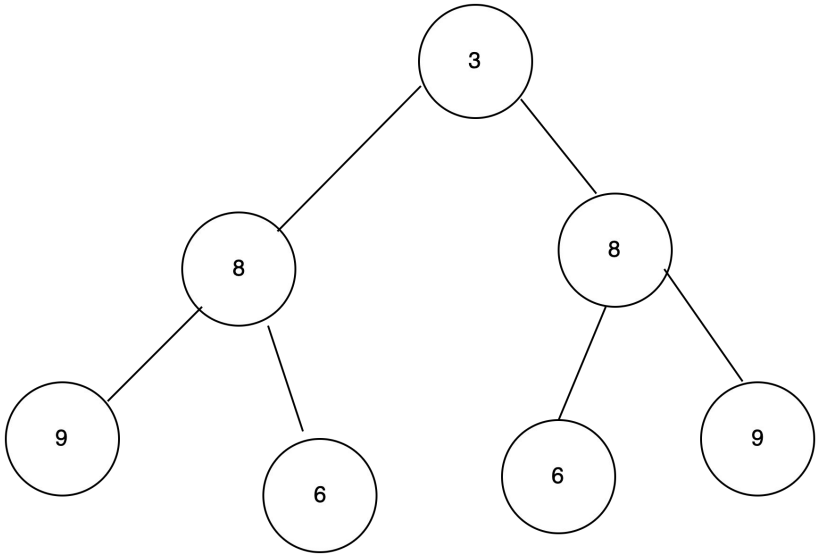
Симметричное дерево DFS

- Самый простой вариант - вариант при котором, если бы проходились по бинарному дереву поиска, то мы бы получили отсортированный массив LNR
- **[9, 8, 6, 3, 6, 8, 9]**



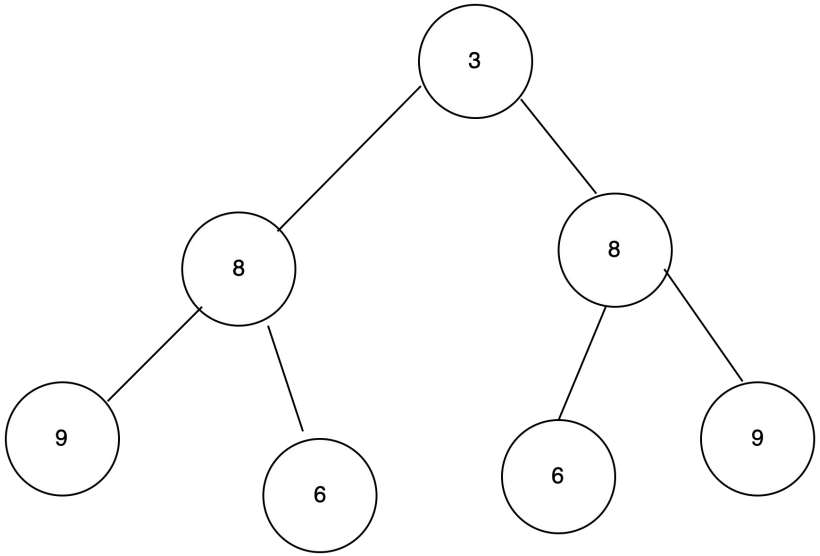
Симметричное дерево DFS

```
function deptSearch(root, res) {
    if root == null:
        return

    // в начале левое поддерево
    deptSearch(root.left, res)
    res.append(root.data)
    // затем правое
    deptSearch(root.right, res)
}
```



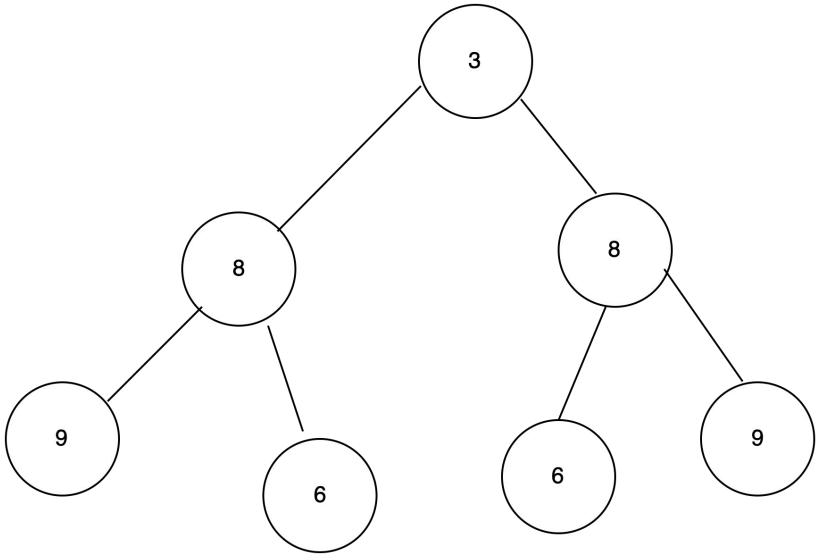
Симметричное дерево
DFS

```
function isSymmetricDFS(root) {
    // если корень пустой, считаем что
    // дерево симметрично.

    // обходим дерево сохраняя
    // в массив результат обхода.

    // как раньше проверяли очередь -
    // проверяем массив на симметричность

    return true
}
```



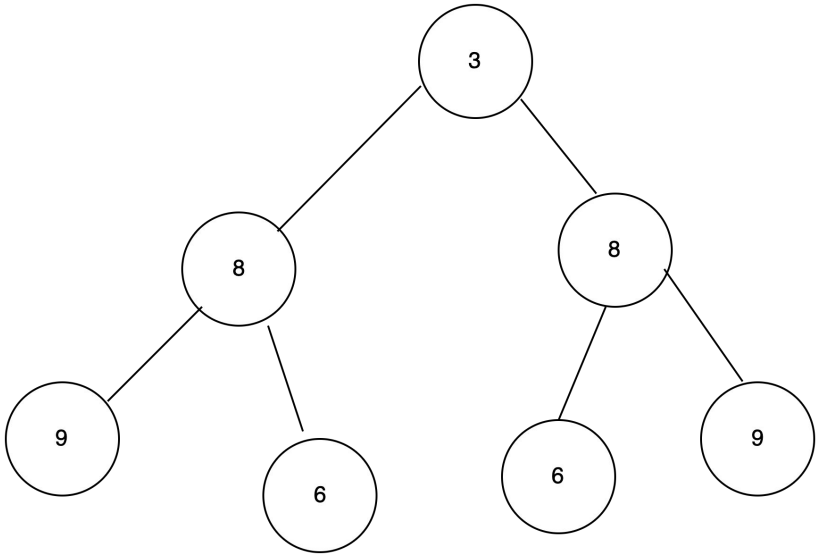
Симметричное дерево
DFS

```
function isSymmetricDFS(root) {
    if root == null {
        return true
    }
    data = []
    deptSearch(root, data)
    // как раньше проверяли очередь
    // проверяем массив на симметричность

    return true
}
```



Симметричное дерево DFS

```
function isSymmetricDFS(root) {
    if root == null {
        return true
    }
    data = []
    deptSearch(root, data)
    j = len(data) - 1
    for i = 0; i < len(data)/2; i++ {
        if data[i] != data[j] {
            return false
        }
        j--
    }

    return true
}
```



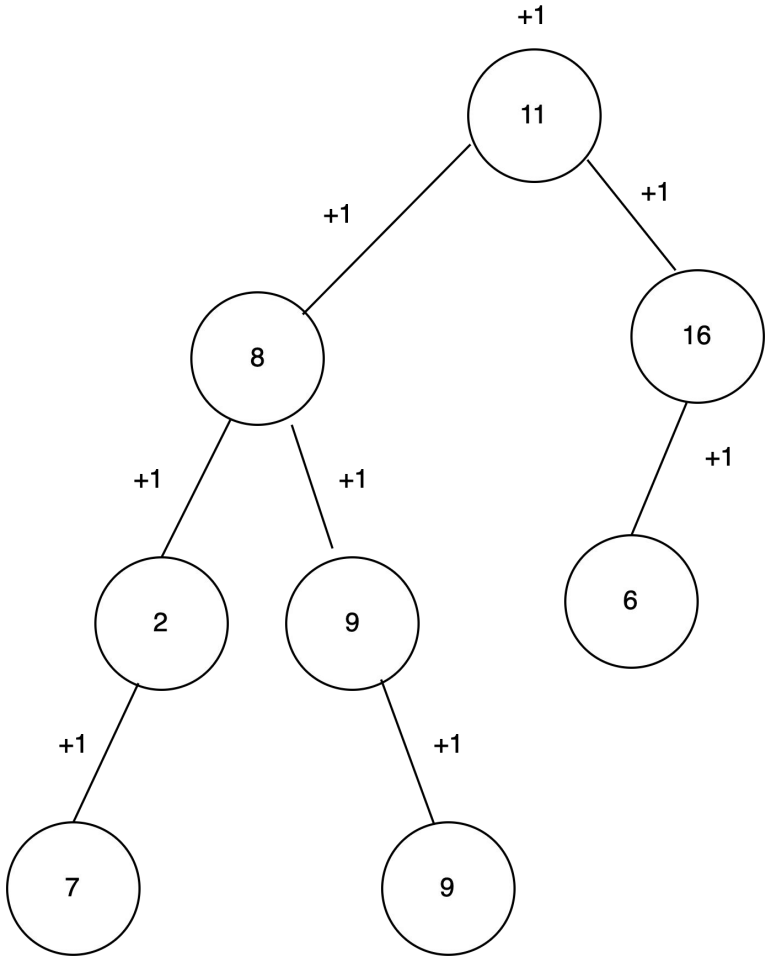
[9, 8, 6, 3, 6, 8, 9]

Поиск минимальной глубины бинарного дерева

На вход функции подается бинарное дерево. Необходимо найти минимальную глубину дерева

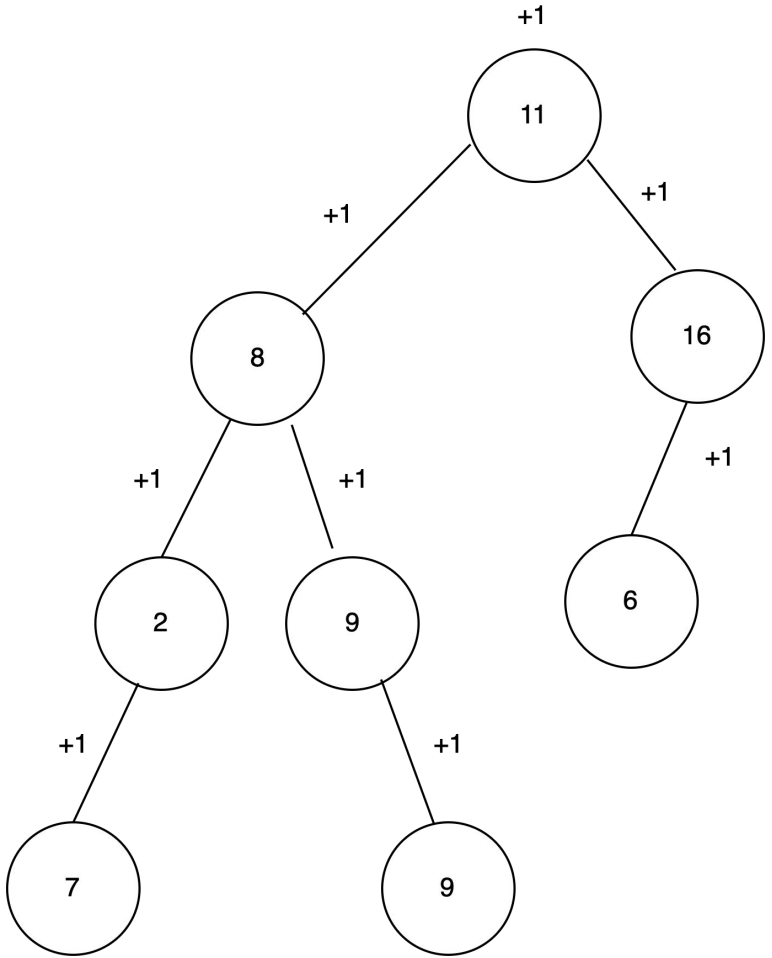
Минимальная глубина — это количество узлов на кратчайшем пути от корневого узла до ближайшего листового узла.

Для данного дерева минимальная глубина 3: узел 11 -> узел 16 -> узел 6



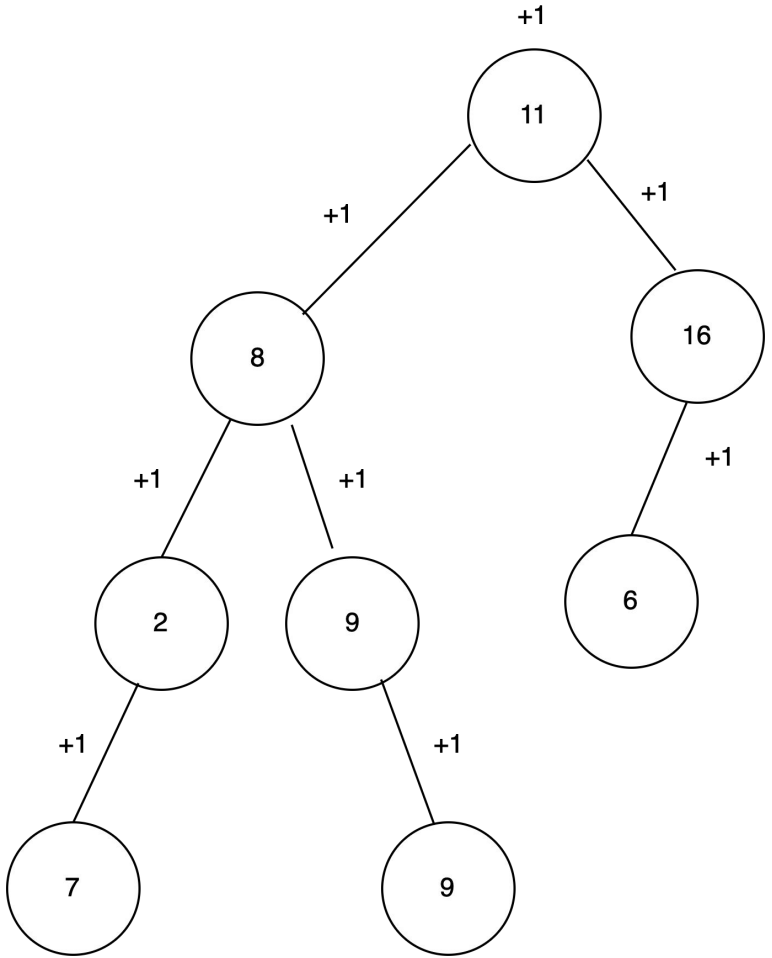
Поиск минимальной глубины бинарного дерева DFS

```
function minDepth(root) {
  // если root не существует, значит на этом
  // уровне глубина равна нулю


  // если у узла есть и левый и правый потомок
  // а значит есть и левое и правое поддерево
  // мы должны вернуть минимальную глубину из
  // этих поддеревьев
  // к результату мы должны прибавить 1
  // чтобы учесть корневой уровень
}
```



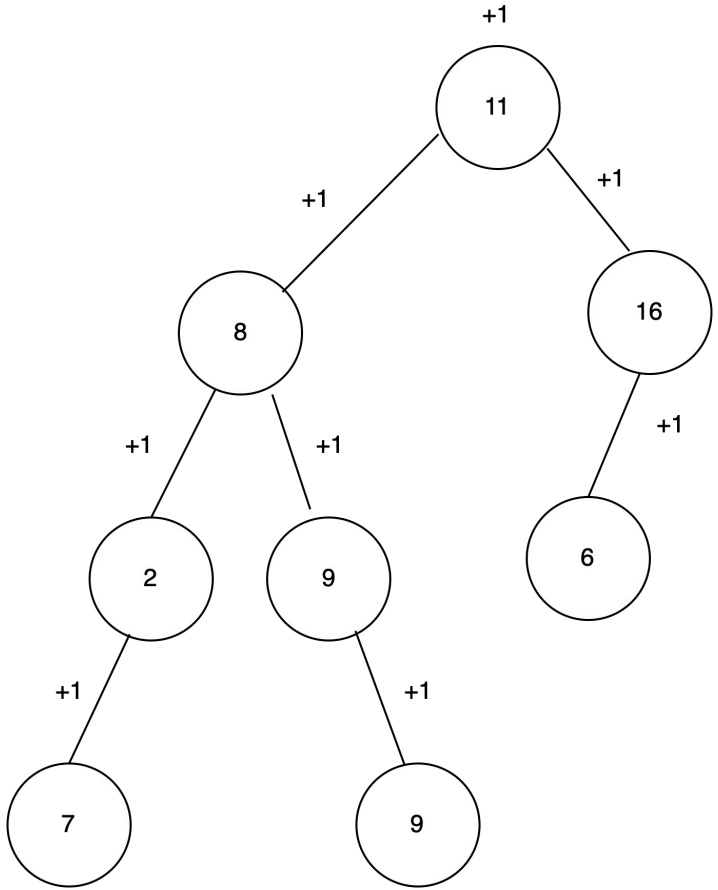
Поиск минимальной глубины бинарного дерева DFS

```
function minDepth(root) {

  ...
  // если есть только левое поддерево
  // продолжаем поиск только по нему
  // плюс корневой уровень

  // если есть только правое поддерево
  // продолжаем поиск только по нему
  // плюс корневой уровень
}
```



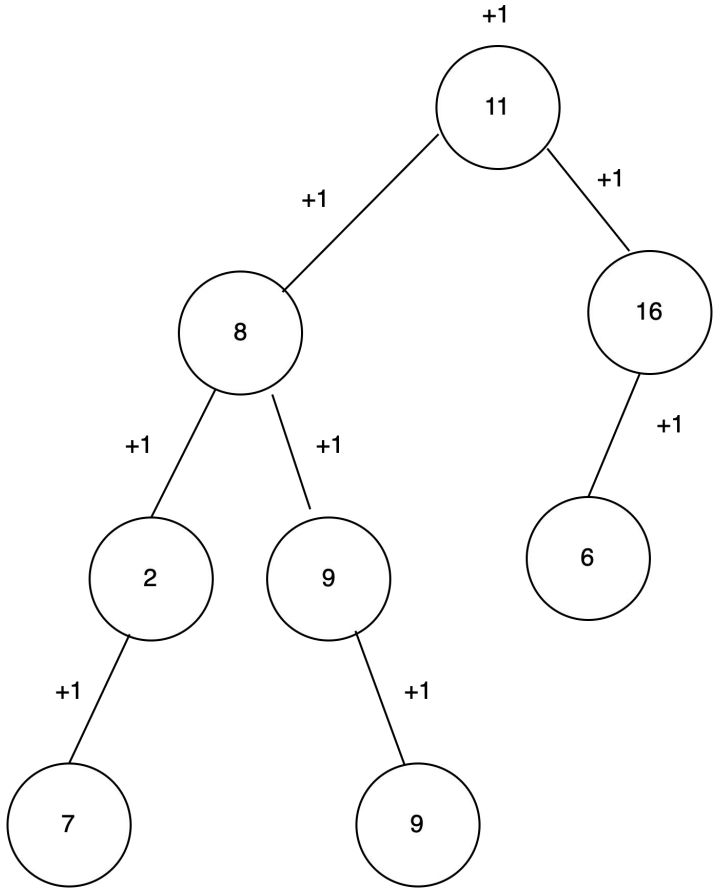
Поиск минимальной глубины бинарного дерева DFS

```
function minDepth(root) {
    if not root {
        return 0
    }

    if root.left != null and root.right != null {
        return 1 + min(minDepth(root.left), minDepth(root.right))
    }

    ...
}
```



Поиск минимальной глубины бинарного дерева DFS

```
function minDepth(root) {
    if not root {
        return 0
    }
    if root.left != null and root.right != null {
        return 1 + min(minDepth(root.left), minDepth(root.right))
    }
    if root.left != null {
        return 1 + minDepth(root.left)
    }
    if root.right != null {
        return 1 + minDepth(root.right)
    }
}
```

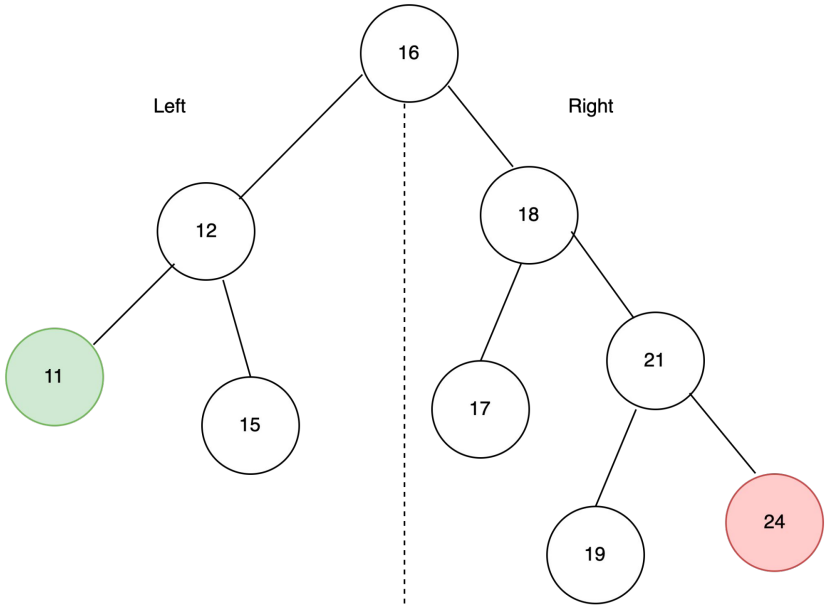


Поиск произведения максимального и минимального элементов

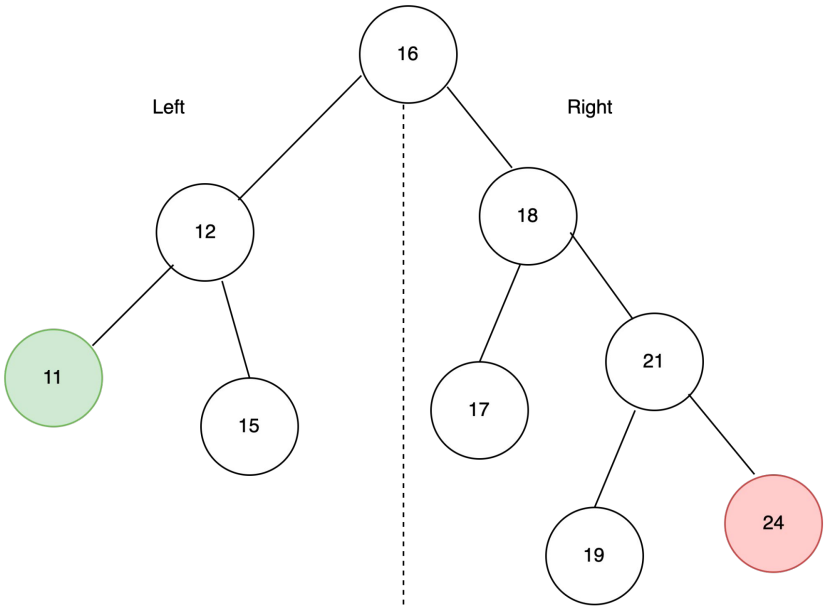
Дано бинарное дерево поиска в виде массива. Необходимо найти произведение минимального и максимального значений.

Поиск максимального произведения

```
function maxMinMultiplication(data) {
    if length(data) == 1 {
        return -1
    }

    // определяем индекс минимального элемента
    // устанавливаем min_index равным 1
    // далее в цикле двигаем его на 2 * i + 1
    // определяем индекс максимального элемента
    // устанавливаем max_index равным 2
    // далее двигаем его на 2 * i + 2

    result = data[min_index] * data[max_index]
    return result
}
```

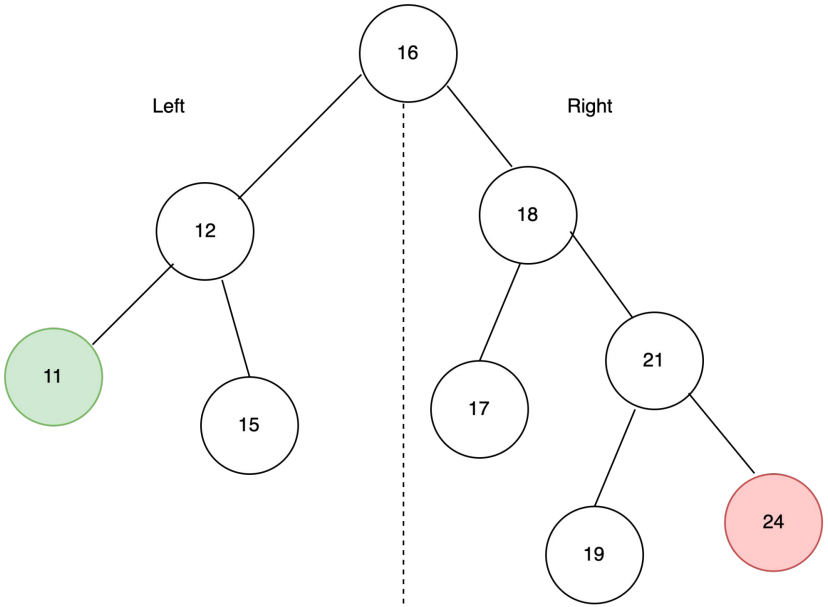


Поиск максимального произведения



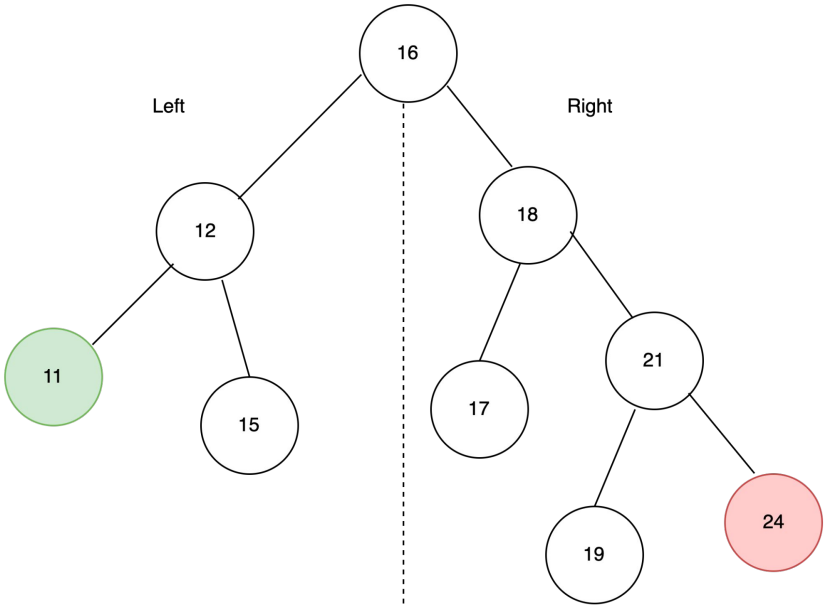

```
function maxMinMultiplication(data) {
    if length(data) < 3 {
        return -1
    }

    min_index = 1
    max_index = 2
    i = 0
    while true {
        min_index_tmp = 2 * i + 1
        if min_index_tmp < length(data) {
            // переносим значение минимального
            // индекса на min_index_tmp
            // также двигаем i на min_index_tmp
            continue
        }
        break
    }

    ...

    result = data[min_index] * data[max_index]
    return result
}
```

Поиск максимального произведения



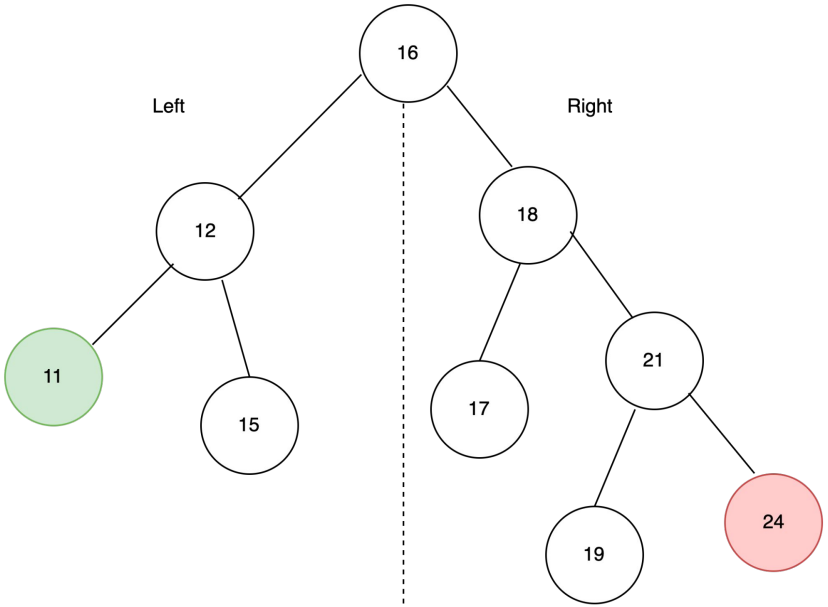

```
function maxMinMultiplication(data) {
    if length(data) < 3 {
        return -1
    }

    min_index = 1
    max_index = 2
    i = 0
    while true {
        min_index_tmp = 2 * i + 1
        if min_index_tmp < length(data) {
            min_index = min_index_tmp
            i = min_index_tmp
            continue
        }
        break
    }

    ...

    result = data[min_index] * data[max_index]
    return result
}
```

Поиск максимального произведения




```
function maxMinMultiplication(data) {
    if length(data) < 3 {
        return -1
    }

    min_index = 1
    max_index = 2
    for i = 1; i < length(data); i = 2 * i + 1 {
        min_index = i
    }

    ...

    result = data[min_index] * data[max_index]
    return result
}
```

Поиск максимального произведения



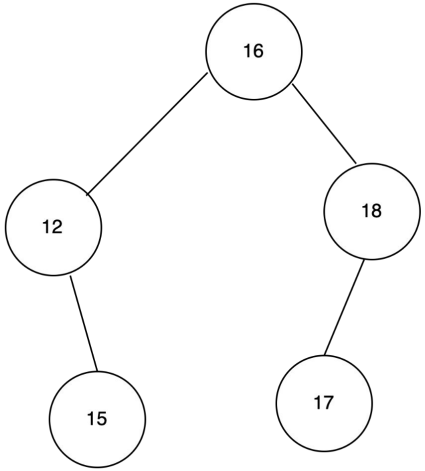

```
function maxMinMultiplication(data) {
    if length(data) < 3 {
        return -1
    }

    min_index = 1
    max_index = 2
    for i = 1; i < length(data); i = 2 * i + 1 {
        min_index = i
    }

    for i = 1; i < length(data); i = 2 * i + 2 {
        max_index = i
    }

    result = data[min_index] * data[max_index]
    return result
}
```

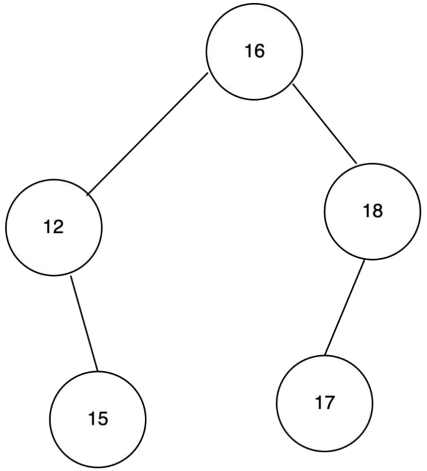
Сравнить два дерева

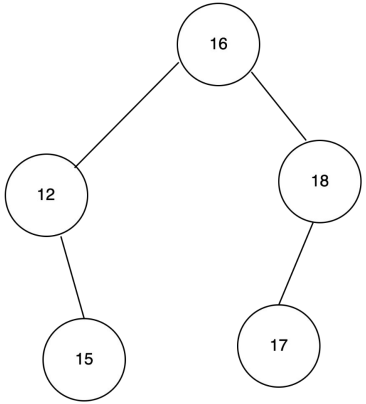
На вход функции подается 2 бинарных дерева. Необходимо понять, являются ли эти два дерева одинаковыми

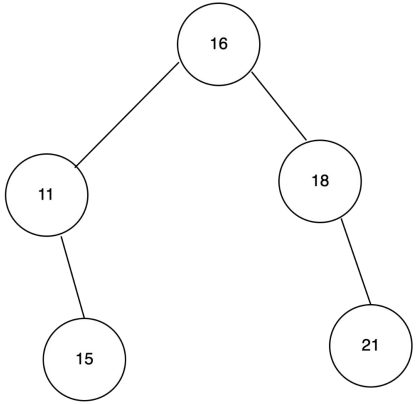
Являются ли два дерева одинаковыми

- BFS
- DFS (NLR NRL)
- Сравниваем два массива



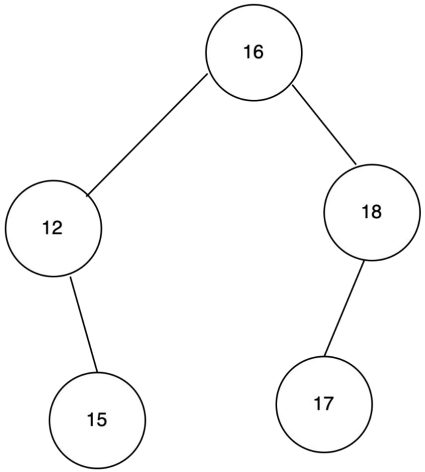


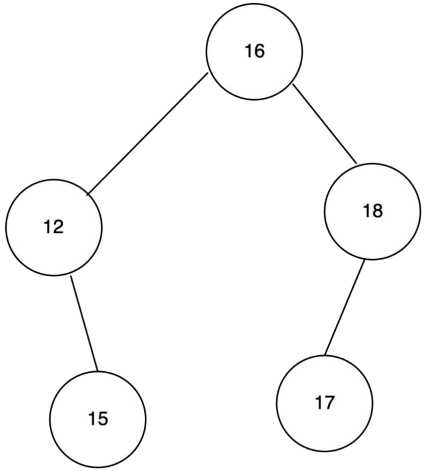




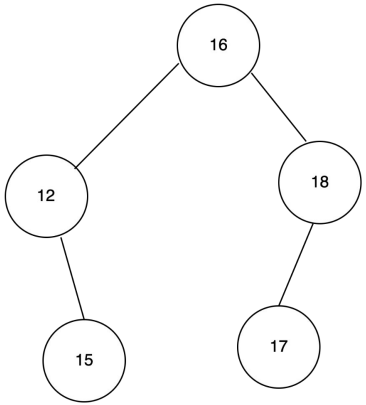
Являются ли два дерева одинаковыми

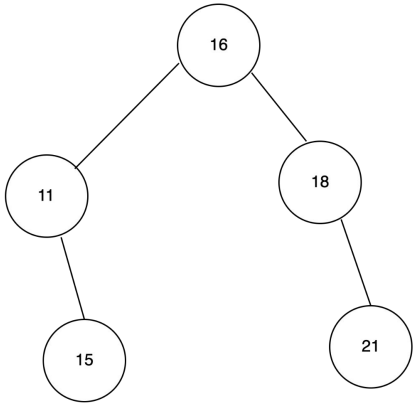
- рекурсивный подход
- на каждом вызове сравниваем соответствующие поддеревья






```
function isSameTree(a, b) {
    if a == null and b == null {
        return true
    }

    ...

    return ?
}
```





Являются ли два дерева одинаковыми

- рекурсивный подход
- на каждом вызове сравниваем соответствующие поддеревья

```
function isSameTree(a, b) {
   if a == null and b == null {
      return true
   }
   if a == null or b == null {
      return false
   }
   ...

   return ?
}
```

Являются ли два дерева одинаковыми

- рекурсивный подход
- на каждом вызове сравниваем соответствующие поддеревья

```
function isSameTree(a, b) {
    if a == null and b == null {
        return true
    }
    if a == null or b == null {
        return false
    }
    if a.data != b.data {
        return false
    }

    return ?
}
```

Являются ли два дерева одинаковыми

- рекурсивный подход
- на каждом вызове сравниваем соответствующие поддеревья

```
function isSameTree(a, b) {
    if a == null and b == null {
        return true
    }
    if a == null or b == null {
        return false
    }
    if a.data != b.data {
        return false
    }

    return isSameTree(a.left, b.left) and
                isSameTree( a.right, b.right)
}
```

Всем спасибо

и хороших выходных:)