// 下面是二叉树的构造函数，

// 三个参数分别是左树、当前节点和右树

function Tree(left, label, right) {

this.left = left;

this.label = label;

this.right = right;

}

// 下面是中序（inorder）遍历函数。

// 由于返回的是一个遍历器，所以要用generator函数。

// 函数体内采用递归算法，所以左树和右树要用yield\*遍历

function\* inorder(t) {

if (t) {

yield\* inorder(t.left);

yield t.label;

yield\* inorder(t.right);

}

}

// 下面生成二叉树

function make(array) {

// 判断是否为叶节点

if (array.length == 1) return new Tree(null, array[0], null);

return new Tree(make(array[0]), array[1], make(array[2]));

}

let tree = make([[['a'], 'b', ['c']], 'd', [['e'], 'f', ['g']]]);

// 遍历二叉树

var result = [];

for (let node of inorder(tree)) {

result.push(node);

}

result

// ['a', 'b', 'c', 'd', 'e', 'f', 'g']

**二叉树遍历方式深度优先遍历和广度优先遍历的递归实现和非递归实现**

定义好一棵二叉树

var nodes = {

node: 6,

left: {

node: 5,

left: {

node: 4

},

right: {

node: 3

}

},

right: {

node: 2,

right: {

node: 1

}

}

}

**深度优先遍历二叉树。**

**1、先序遍历（DLR）的算法**：

递归算法

1. 若二叉树为空，则算法结束，否则：
2. 访问根结点；
3. 前序遍历根结点的左子树；
4. 前序遍历根结点的右子树。

var result = []

var dfs = function(nodes) {

if(nodes.node) {

result.push(nodes.node)

nodes.left && dfs(nodes.left)

nodes.right && dfs(nodes.right)

}

}

dfs(nodes)

console.log(result)

// [6, 5, 4, 3, 2, 1]

非递归算法

* + 1. 初始化一个栈，将根节点压入栈中；
    2. 当栈为非空时，循环执行步骤3到4，否则执行结束；
    3. 出队列取得一个结点，访问该结点；
    4. 若该结点的右子树为非空，则将该结点的右子树入栈，若该结点的左子树为非空，则将该结点的左子树入栈；

var dfs = function(nodes) {

var result = []

var stack = []

stack.push(nodes)

while (stack.length) {

var item = stack.pop()

result.push(item.node)

item.right && stack.push(item.right)

item.left && stack.push(item.left)

}

return result

}

console.log(dfs(nodes))

// [6, 5, 4, 3, 2, 1]

**2、中序遍历（LDR）的算法：**

递归算法

* + 1. 若二叉树为空，则算法结束；否则：
    2. 中序遍历根结点的左子树；
    3. 访问根结点；
    4. 中序遍历根结点的右子树；

var result = []

var dfs = function(nodes) {

if(nodes.node) {

nodes.left && dfs(nodes.left)

result.push(nodes.node)

nodes.right && dfs(nodes.right)

}

}

dfs(nodes)

console.log(result)

// [4, 5, 3, 6, 2, 1]

非递归算法

* + 1. 初始化一个栈，将根节点压入栈中，并标记为当前节点(item)；
    2. 当栈为非空时，执行步骤3，否则执行结束；
    3. 如果当前节点(item)有左子树且没有被 touched，则执行4，否则执行5；
    4. 对当前节点(item)标记 touched，将当前节点的左子树赋值给当前节点(item=item.left) 并将当前节点(item)压入栈中，回到3；
    5. 清理当前节点(item)的 touched 标记，取出栈中的一个节点标记为当前节点(item)，并访问，若当前节点(item)的右子树为非空，则将该结点的右子树入栈，回到3；

var dfs = function(nodes) {

var result = []

var stack = []

var item = nodes

stack.push(nodes)

while (stack.length) {

if(item.left && !item.touched) {

item.touched = true

item = item.left

stack.push(item)

continue

}

item.touched && delete item.touched // 清理标记

item = stack.pop()

result.push(item.node)

item.right && stack.push(item.right)

}

return result

}

console.log(dfs(nodes))

// [4, 5, 3, 6, 2, 1]

**3、后序遍历（LRD）的算法：**

递归算法：

1. 若二叉树为空，则算法结束，否则：
2. 后序遍历根结点的左子树；
3. 后序遍历根结点的右子树；
4. 访问根结点。

var result = []

var dfs = function(nodes) {

if(nodes.node) {

nodes.left && dfs(nodes.left)

nodes.right && dfs(nodes.right)

result.push(nodes.node)

}

}

dfs(nodes)

console.log(result)

// [4, 3, 5, 1, 2, 6]

非递归算法：

1. 初始化一个栈，将根节点压入栈中，并标记为当前节点(item)；
2. 当栈为非空时，执行步骤3，否则执行结束；
3. 如果当前节点(item)有左子树且没有被 touched，则执行4，如果被 touched left 但没有被 touched right 则执行5 否则执行6；
4. 对当前节点(item)标记 touched left，将当前节点的左子树赋值给当前节点(item=item.left) 并将当前节点(item)压入栈中，回到3；
5. 对当前节点(item)标记 touched right，将当前节点的右子树赋值给当前节点(item=item.right) 并将当前节点(item)压入栈中，回到3；
6. 清理当前节点(item)的 touched 标记，弹出栈中的一个节点并访问，然后再将栈顶节点标记为当前节点(item)，回到3；

var dfs = function(nodes) {

var result = []

var stack = []

var item = nodes

stack.push(nodes)

while (stack.length) {

if(item.left && !item.touched) {

item.touched = 'left'

item = item.left

stack.push(item)

continue

}

if(item.right && item.touched !== 'right') {

item.touched = 'right'

item = item.right

stack.push(item)

continue

}

var out = stack.pop()

out.touched && delete out.touched // 清理标记

result.push(out.node)

item = stack.length ? stack[stack.length - 1] : null

}

return result

}

console.log(dfs(nodes))

// [4, 3, 5, 1, 2, 6]

**广度优先遍历二叉树：**

广度优先遍历二叉树(层序遍历)是用队列来实现的，从二叉树的第一层（根结点）开始，自上至下逐层遍历；在同一层中，按照从左到右的顺序对结点逐一访问。

按照从根结点至叶结点、从左子树至右子树的次序访问二叉树的结点。步骤：

1. 初始化一个队列，并把根结点入列队；
2. 当队列为非空时，循环执行步骤3到4，否则执行结束；
3. 出队列取得一个结点，访问该结点；
4. 若该结点的左子树为非空，则将该结点的左子树入队列，若该结点的右子树为非空，则将该结点的右子树入队列；

递归算法：

var result = []

var queue = [nodes]

var bfs = function(count) {

count = count || 0

if(queue[count]) {

result.push(queue[count].node)

var left = queue[count].left

var right = queue[count].right

if(left) {

queue.push(left)

}

if(right) {

queue.push(right)

}

bfs(++count)

}

}

bfs()

console.log(result)

// [6, 5, 2, 4, 3, 1]

非递归算法

var bfs = function(nodes) {

var result = []

var queue = []

var pointer = 0

queue.push(nodes)

while(pointer < queue.length) {

var item = queue[pointer++] // 这里不使用 shift 方法（复杂度高），用一个指针代替

result.push(item.node)

console.log(item.node)

item.left && queue.push(item.left)

item.right && queue.push(item.right)

}

return result

}

console.log(bfs(nodes))

// [6, 5, 2, 4, 3, 1]