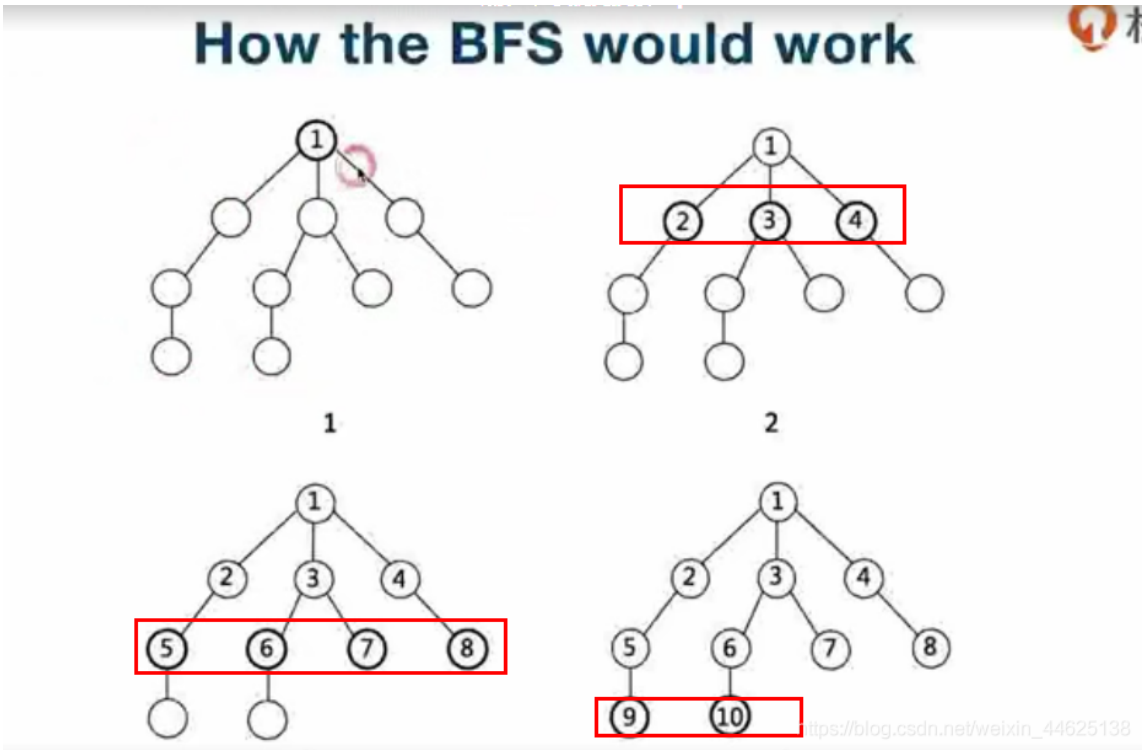
广度优先搜索（BFS ）：

宽度优先搜索算法（又称广度优先搜索）是最简便的图的搜索算法之一，这一算法也是很多重要的图的算法的原型。Dijkstra单源最短路径算法和Prim最小生成树算法都采用了和宽度优先搜索类似的思想。其别名又叫BFS，属于一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。

特点：从算法的观点，所有因为展开节点而得到的子节点都会被加进一个先进先出的队列中。一般的实验里，其邻居节点尚未被检验过的节点会被放置在一个被称为 open 的容器中（例如队列或是链表），而被检验过的节点则被放置在被称为 closed （visited）的容器中。（open-closed表）（如果是遍历图的话就很有可能出现遍历重复的值，因此需要判重）

图解：



广度优先算法的示例代码：（非递归的写法）

```
def BFS(graph, start, end):

    queue = []
    queue.append([start])
    visited.add(start)

    while queue:
        node = queue.pop()
        visited.add(node)

        process(node)
        nodes = generate_related_nodes(node)
        queue.push(nodes)
```

深度优先搜索

深度优先搜索是一种在开发爬虫早期使用较多的方法。它的目的是要达到被搜索结构的叶结点(即那些不包含任何超链的HTML文件) 。在一个HTML文件中，当一个超链被选择后，被链接的HTML文件将执行深度优先搜索，即在搜索其余的超链结果之前必须先完整地搜索单独的一条链。深度优先搜索沿着HTML文件上的超链走到不能再深入为止，然后返回到某一个HTML文件，再继续选择该HTML文件中的其他超链。当不再有其他超链可选择时，说明搜索已经结束。

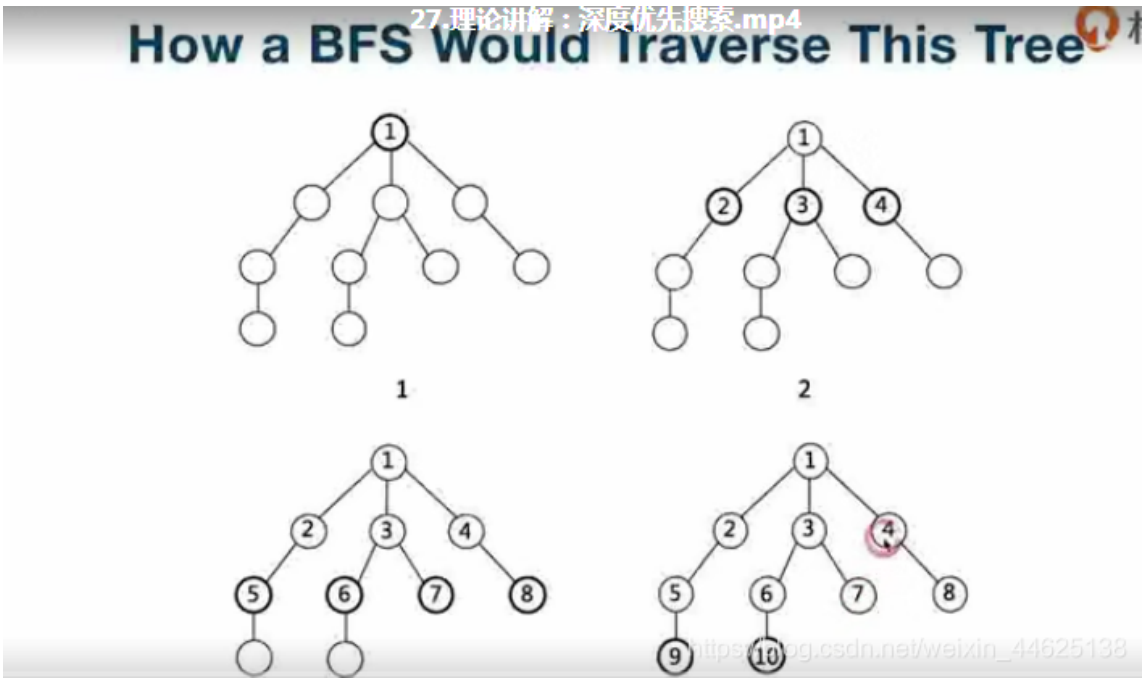
事实上，深度优先搜索属于图算法的一种，英文缩写为DFS即Depth First Search.其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次.

深度优先遍历图的方法是，从图中某顶点v出发：

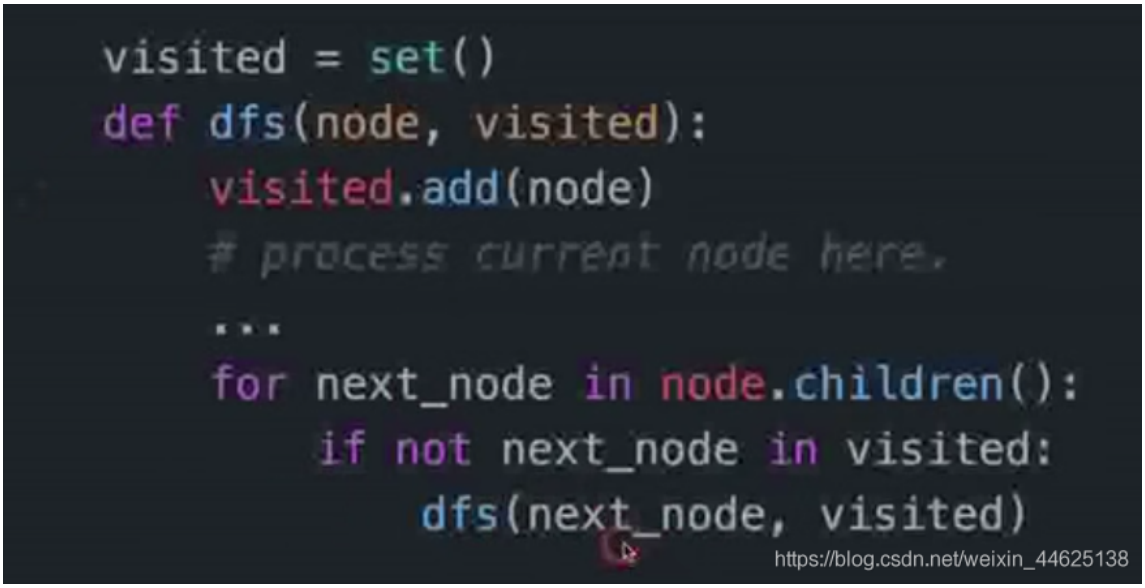
（1）访问顶点v；

（2）依次从v的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和v有路径相通的顶点都被访问；

（3）若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止



深度优先的伪代码：（递归的写法）



经典例题：

1、广度优先搜索层次遍历二叉树

广度优先：

```
public class LevelOrder {
//迭代法：
//时间复杂度：O(N)O(N)，因为每个节点恰好会被运算一次。
// 空间复杂度：O(N)O(N)，保存输出结果的数组包含 N 个节点的值。
public List<List<Integer>> Bfs(TreeNode root) {
List<List<Integer>> result=new ArrayList<>();
Set<TreeNode> Visted=new HashSet<>(); //在图中需要注意是否重复访问
Queue<TreeNode> queue=new LinkedList();
if (root==null) return result;
queue.add(root);
Visted.add(root);
int level=0;
while(!queue.isEmpty()){
result.add(new ArrayList<Integer>());
int size=queue.size();
for(int i=0;i<size;i++){
TreeNode node=queue.poll();
Visted.add(node);
// fulfill the current level
result.get(level).add(node.val);
// add child nodes of the current level
// in the queue for the next level
if (node.left!=null && !Visted.contains(node.left)) {
```

queue.add(node.left);
}
if (node.right!=null&&!Visted.contains(node.right)) {
queue.add(node.right);
}
}
level++; //go to next level
}
return result;
}

```java
//方法二：递归
 List<List<Integer>> levels = new ArrayList<List<Integer>>();
 public List<List<Integer>> Bfs2(TreeNode root){
         if (root==null) return levels;
         helper(root,0);
         return levels;
 }
private void helper(TreeNode node, int level) {
        // TODO Auto-generated method stub
        if (levels.size()==level)
                levels.add(new ArrayList<>());
                levels.get(level).add(node.val);
                if (node.left != null)
            helper(node.left, level + 1);
         if (node.right != null)
            helper(node.right, level + 1);
}

```

}

2、给定一个二叉树，找出其最大深度。二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点（https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/）

package bfsAnddfs;
import java.util.LinkedList;
import java.util.Map;
import java.util.Queue;

public class MaxDepth {
//深度搜索，递归
// 复杂度分析
//
// 时间复杂度：我们每个结点只访问一次，因此时间复杂度为 O(N)，
// 其中 N是结点的数量。
// 空间复杂度：在最糟糕的情况下，树是完全不平衡的，例如每个结点只剩下左子结点，递归将会被调用 N 次（树的高度），因此保持调用栈的存储将是 O(N)O(N)。但在最好的情况下（树是完全平衡的），树的高度将是 \log(N)。因此，在这种情况下的空间复杂度将是 O(\log(N))。

```
public int maxDepth(TreeNode root) {
        if(root==null) return 0;
        int left=maxDepth(root.left);
        int right=maxDepth(root.right);
        return Math.max(left, right)+1;

    }

```

// 方法二：迭代
// 我们还可以在栈的帮助下将上面的递归转换为迭代。
//
// 我们的想法是使用 DFS 策略访问每个结点，同时在每次访问时更新最大深度。

```
public int maxDepth2(TreeNode root) {
        if(root==null) return 0;
        Queue<pair> queue = new LinkedList<>();
        queue.add(new pair(1, root));

        int depth=0;  //the globe variable to mark the depth and update while iterate;
        while(!queue.isEmpty()){

                pair pair=queue.poll();  //return the head of the queue
                int current_depth=pair.height;
                if (pair.node!=null) {
                        depth=Math.max(current_depth, depth);  //
                        queue.add(new pair(depth+1, pair.node.left));
                        queue.add(new pair(depth+1, pair.node.right));
                        }
                return depth;
```

```
        }
    }

```

}

class pair{

int height;

TreeNode node;

public pair(int height, TreeNode node) {

super();

this.height = height;

this.node = node;

}

}

3、给定一个二叉树，找出其最小深度。最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明: 叶子节点是指没有子节点的节点。（https://leetcode-cn.com/problems/minimum-depth-of-binary-tree/）

package bfsAnddfs;

import java.util.LinkedList;

import java.util.Queue;

public class MinDepth {

// 复杂度分析

//

// 时间复杂度：我们访问每个节点一次，时间复杂度为 O(N)O(N)，其中 NN 是节点个数。

// 空间复杂度：最坏情况下，整棵树是非平衡的，例如每个节点都只有一个孩子，递归会调用 NN（树的高度）次，因此栈的空间开销是 O(N)O(N)。但在最好情况下，树是完全平衡的，高度只有 \log(N)log(N)，因此在这种情况下空间复杂度只有 O(\log(N))O(log(N))。

```
public int minDepth1(TreeNode root){
        if(root==null) return 0;
        //If the node is leaf,the depth of the level is 1.
        if (root.left==null &&root.right==null) {
                return 1;
        }

        int depth=Integer.MAX_VALUE;
        //if the left child of node is null ,we don't need to iterate the left child to count.
```

```
        if (root.left!=null) {
                depth=Math.min(minDepth1(root.left), depth);
        }
        //if the right child of node is null ,we don't need to iterate the left child to count.

        if (root.right!=null) {
                depth=Math.min(minDepth1(root.right), depth);
        }

        return depth+1;
}

//方法 2：深度优先搜索迭代

public int minDepth2(TreeNode root){
         LinkedList<pair> stack = new LinkedList<>();
            if (root == null) {
              return 0;
            }
            else {
              stack.add(new pair(1,root));
            }
            int min_depth = Integer.MAX_VALUE;
            while (!stack.isEmpty()) {
              pair current = stack.pollLast();
              root = current.node;
              int current_depth = current.height;
              if ((root.left == null) && (root.right == null)) {
                min_depth = Math.min(min_depth, current_depth);
              }
              if (root.left != null) {
                stack.add(new pair(current_depth + 1,root.left));
              }
              if (root.right != null) {
                stack.add(new pair(current_depth + 1,root.right));
              }
            }
            return min_depth;
}
```

```
49
50 }
51
```

4、给出 n 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合。例如，给出 n = 3，生成结果为：[

"((()))",

"(()())",

"(())()",

"()(())",

"()()()"

] (https://leetcode-cn.com/problems/generate-parentheses/)

package bfsAnddfs;

import java.util.ArrayList;

import java.util.List;

public class GenerateParenthesis {

```
1  //给出 n 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合
2  public List<String> GenerateParenthesis1(int n){
3          List<String> result=new ArrayList();
4          generateOneByOne("",result,n,n);  //n表示左右括号还未用完的个数；
5          return result;
6  }
7
8  private void generateOneByOne(String string, List<String> result, int left, int right) {
9          // TODO Auto-generated method stub
10         if (left==0&&right==0) {  //如果都是剩余的个数都为零
11                 result.add(string);
12         }
13
14         if(left>0){    //如果左括号还有就直接添加
15                 generateOneByOne(string+"(", result, left-1, right);
16         }
17
18         if (right>left) {  //右括号比左括号多的时候才能添加右括号
19                 generateOneByOne(string+")", result, left, right-1);
20         }
21
22 }
```

```
23
```

}
