搜索
遍历

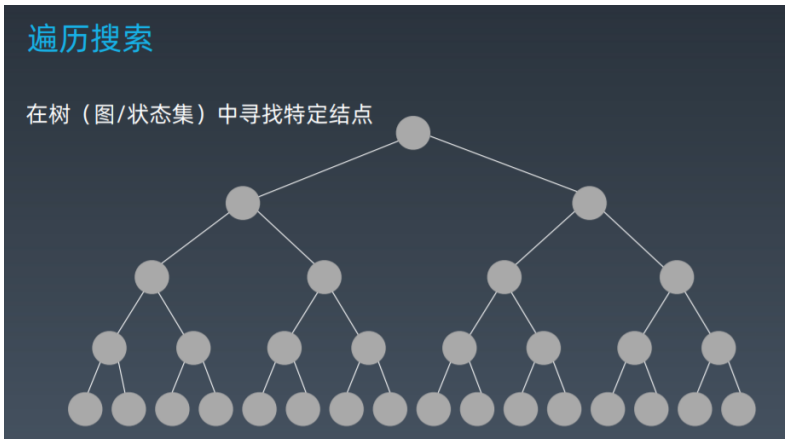
要做的事情 → 无任何智能的情况在里面考虑

把所有结点全部遍历一次，然后找到你要的结果

基于数据结构，如果这个数据结构本身是没有任何特点的，

→ 也就是说是一个很普通的树，或者很普通的图

要做的事情：遍历所有的点，同时保证每个点访问一次，且仅访问一次，最后找到结果




```java
public class TreeNode {
  public int val;
  public TreeNode left, right;
  public TreeNode(int val) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
}
```

**搜索 - 遍历**

- 每个节点都要访问一次
- 每个节点仅仅要访问一次
- 对于节点的访问顺序不限
  - 深度优先：depth first search DFS
  - 广度优先：breadth first search DFS

一优先级优先 (根据业务场景)

→ 启发式搜索

#终止条件　#处理当前层　#下转 → 到子节点

↓

访问了结点 node

把这个节点 node 加到已访问的节点里

DFS

```java
//Java
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> allResults = new ArrayList<>();
        if(root==null){
            return allResults;
        }
        travel(root,0,allResults);
        return allResults;
    }


    private void travel(TreeNode root,int level,List<List<Integer>> results){
        if(results.size()==level){
            results.add(new ArrayList<>());
        }
        results.get(level).add(root.val);
        if(root.left!=null){
            travel(root.left,level+1,results);
        }
        if(root.right!=null){
            travel(root.right,level+1,results);
        }
    }
```

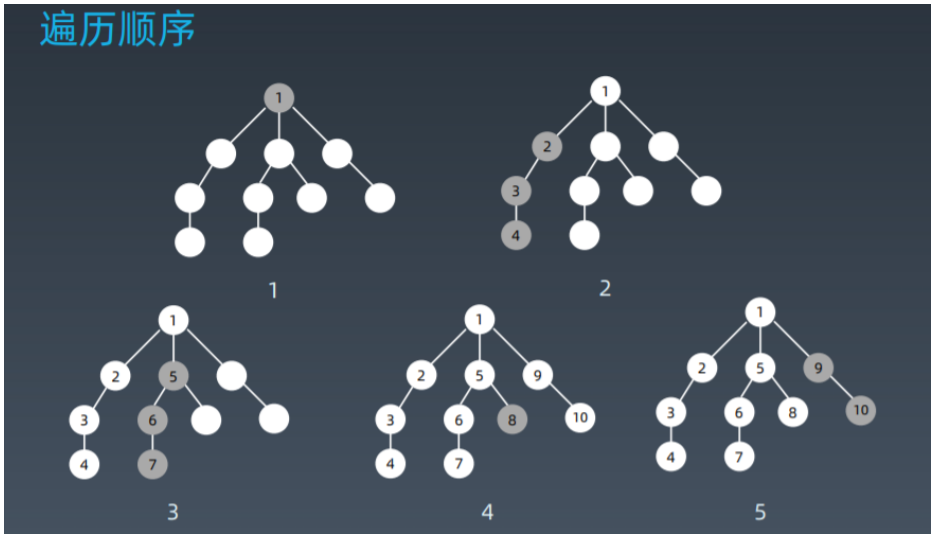
DFS 递归写法


把判断一个节点是否被访问过放在递归的最开始

=

把新的节点加入到要访问的结点再判断

(但要保证新加入的节点是没有访问过的)





返回上一层

看有无其他子节点

root 先放到 visited 里面 (root已经被visit)

被 visited 后就从 root.children 里面找 next_node

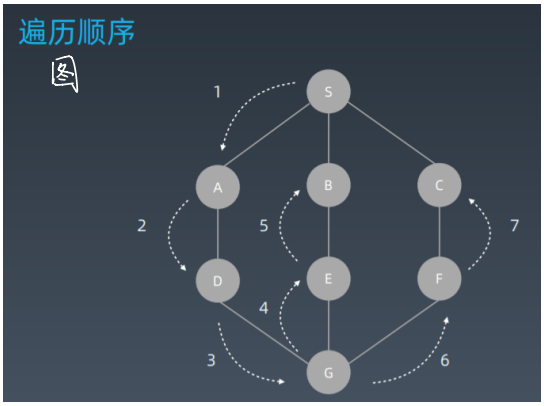
(所有next_node都没有被访问过)

=> 先访问最左边这个节点

dfs

不会等循环结束

会马上下探到下一层.



## DFS 代码 - 递归写法

```
visited = set()

def dfs(node, visited):
    visited.add(node)
    # process current node here.
    ...
    for next_node in node.children():
        if not next_node in visited:
            dfs(next node, visited)
```

## DFS 代码 - 递归写法

```
visited = set()

def dfs(node, visited):
  if node in visited: # terminator
      # already visited
      return

  visited.add(node)

  # process current node here.
  ...
  for next_node in node.children():
      if not next_node in visited:
          dfs(next_node, visited)
```

递归终止条件
处理当前层结果
生成新的下一层要访问的节点
然后再调递归函数
下一层
(下转)
(下转词)再判断一次 next_node 也没有被访问过

## DFS 代码 - 非递归写法

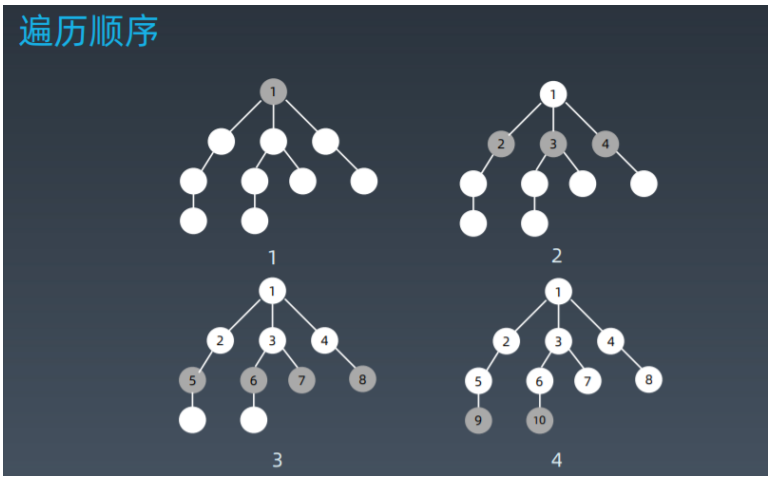
手动维护一个栈

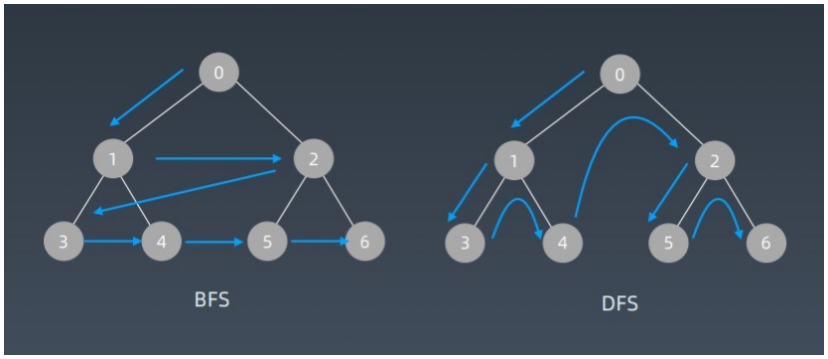
```
def DFS(self, tree):

    if tree.root is None:
        return []

    visited, stack = [], [tree.root]

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

        process (node)
        nodes = generate_related_nodes(node)
        stack.push(nodes)

    # other processing work
    ...
```

# BFS

(不用递归,我们用队列)

## 遍历顺序






## BFS 代码

Python (数组)

```python
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)

    # other processing work
    ...
```

队列为空

把开始节点加入到队列里去

同时的话也要维护 visited 的节点

(接下来) 只要队列不为空，就把这个节点往里面加

然后 process 这个节点，同时从这个节点扩散出它的周围节点，依次加到队列里面去

(队列里的点，一个一个访问，(队列 — 先入先出)

↓ 它会按照成层的这个结点的顺序一个一个从队列里面取

(用链表双端队列)

```java
//Java
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}

public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> allResults = new ArrayList<>();
    if (root == null) {
        return allResults;
    }
    Queue<TreeNode> nodes = new LinkedList<>();
    nodes.add(root);
    while (!nodes.isEmpty()) {
        int size = nodes.size();
        List<Integer> results = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            TreeNode node = nodes.poll();
            results.add(node.val);
            if (node.left != null) {
                nodes.add(node.left);
            }
            if (node.right != null) {
                nodes.add(node.right);
            }
        }
        allResults.add(results);
    }
    return allResults;
}
```