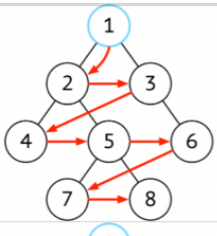
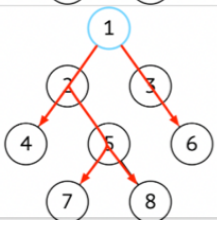
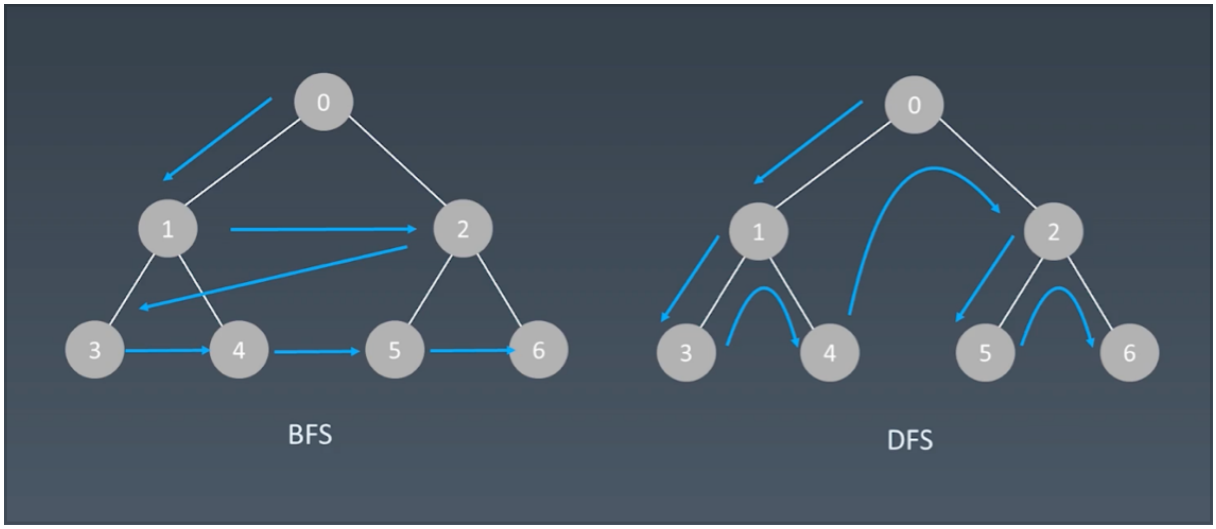
# 10. 深度优先搜索（ DFS ）& 广度优先搜索 (BFS)

## 搜索 - 遍历

- 每个节点都要访问一次；
- 每个节点仅仅访问一次;
- 对于节点的访问顺序不限：
  - 广度优先搜索：Breadth First Search
  - 深度优先搜索：Depth First Search

## BFS & DPS

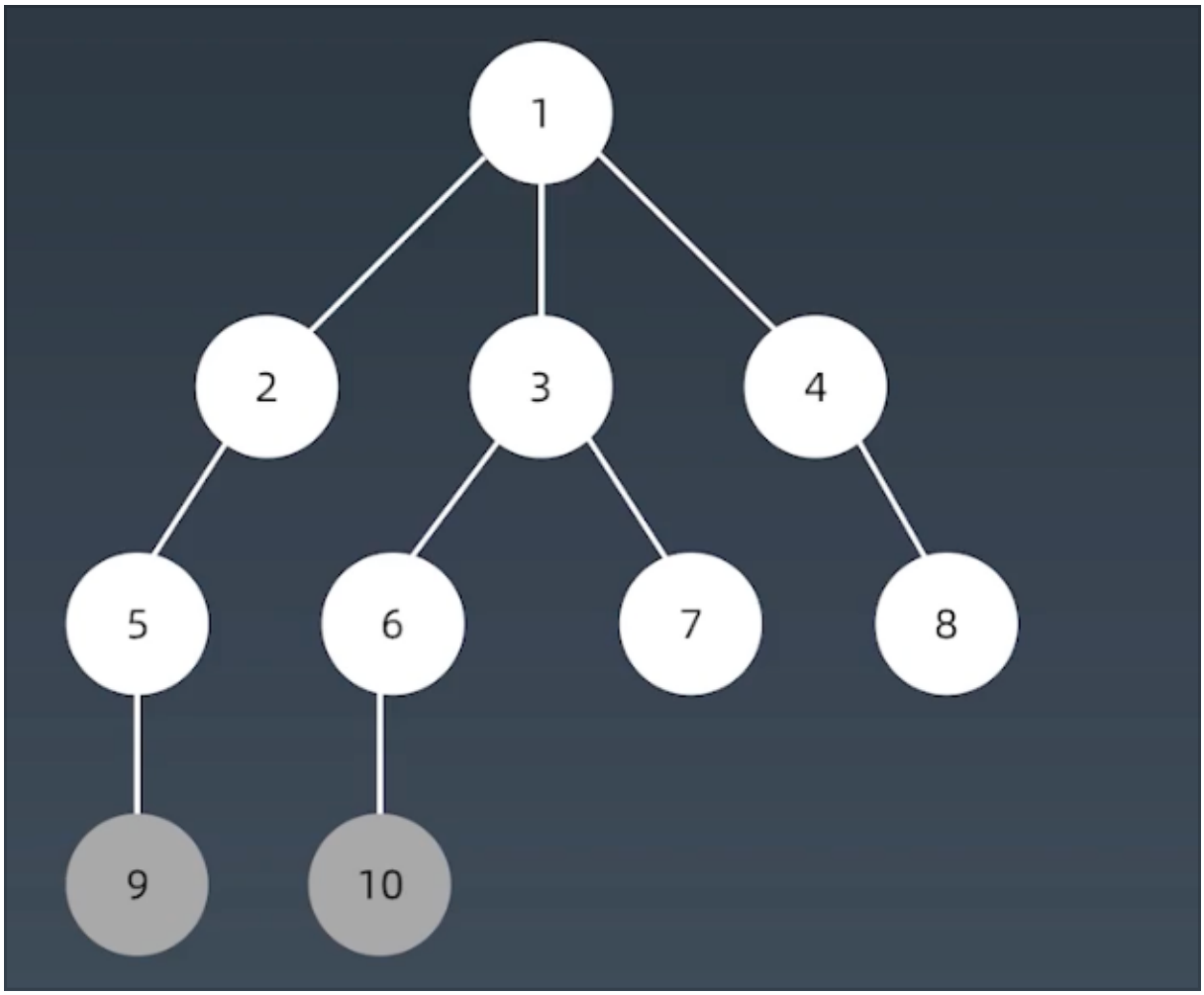
| | | | |
|---|---|---|---|
| 广度优先 | **层序，横向访问** |  | **当树的高度非常高**（非常瘦）<br>使用广度优先剑节省空间 |
| 深度优先 | **纵向，探底到叶子节点** |  | **当每个节点的子节点非常多**（非常胖），使用深度优先遍历节省空间<br>（访问顺序和入栈顺序相关，想当于先序遍历） |




## 示例代码

### BFS 广度优先搜索

- 树 BFS 顺序




**代码示例**

代码结构：多用队列来表示，即 Java 中是 dequeue , Python 中是 list [] 或 connection 库中的高性能 dequeue 数据结构

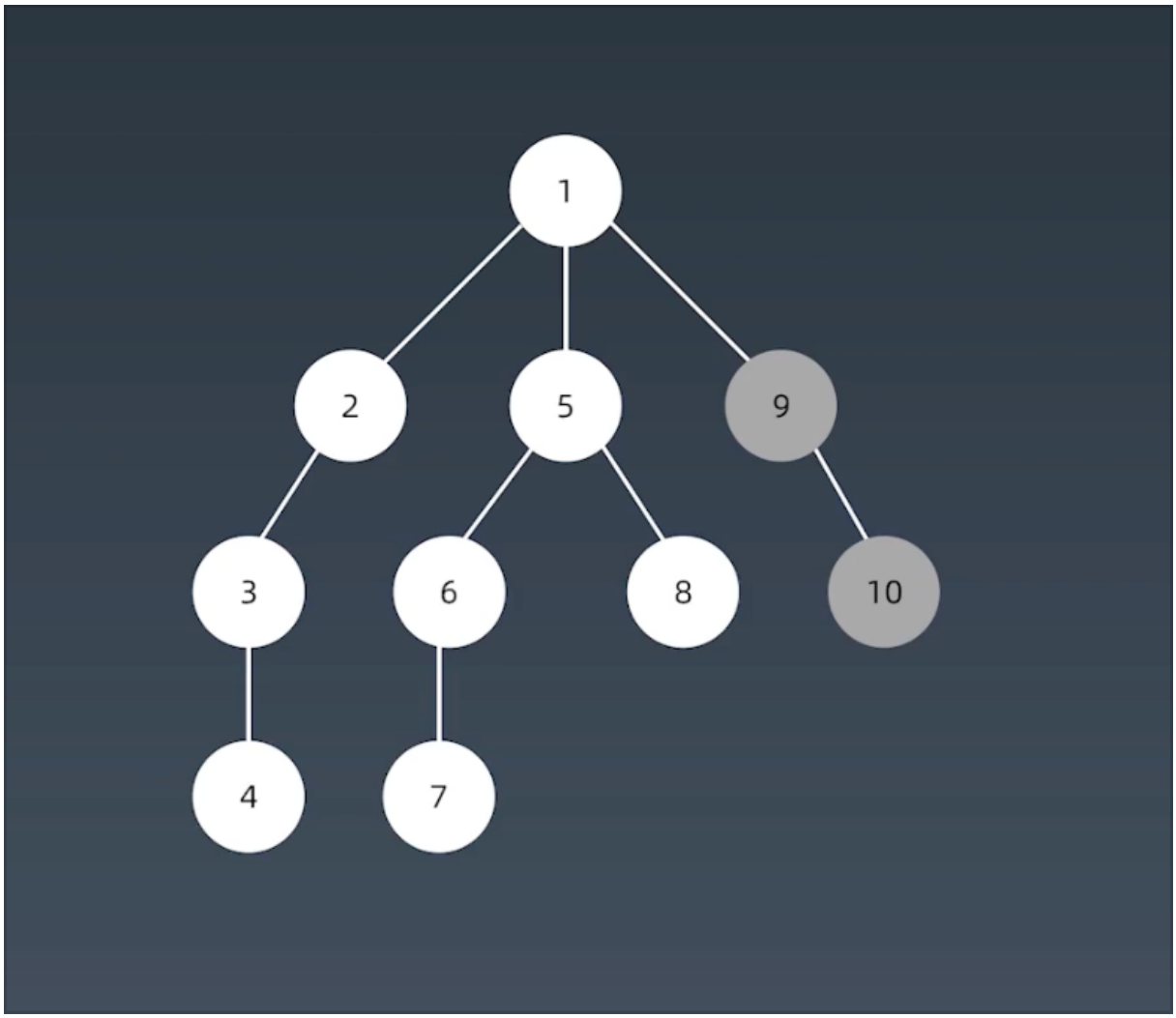
```
def bfs(graph, start, end):

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

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

        process(node)
```

```
        nodes = generate_relaed_nodes(node)
        queue.push(nodes)

    # other processing work
    ....


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;
}
```

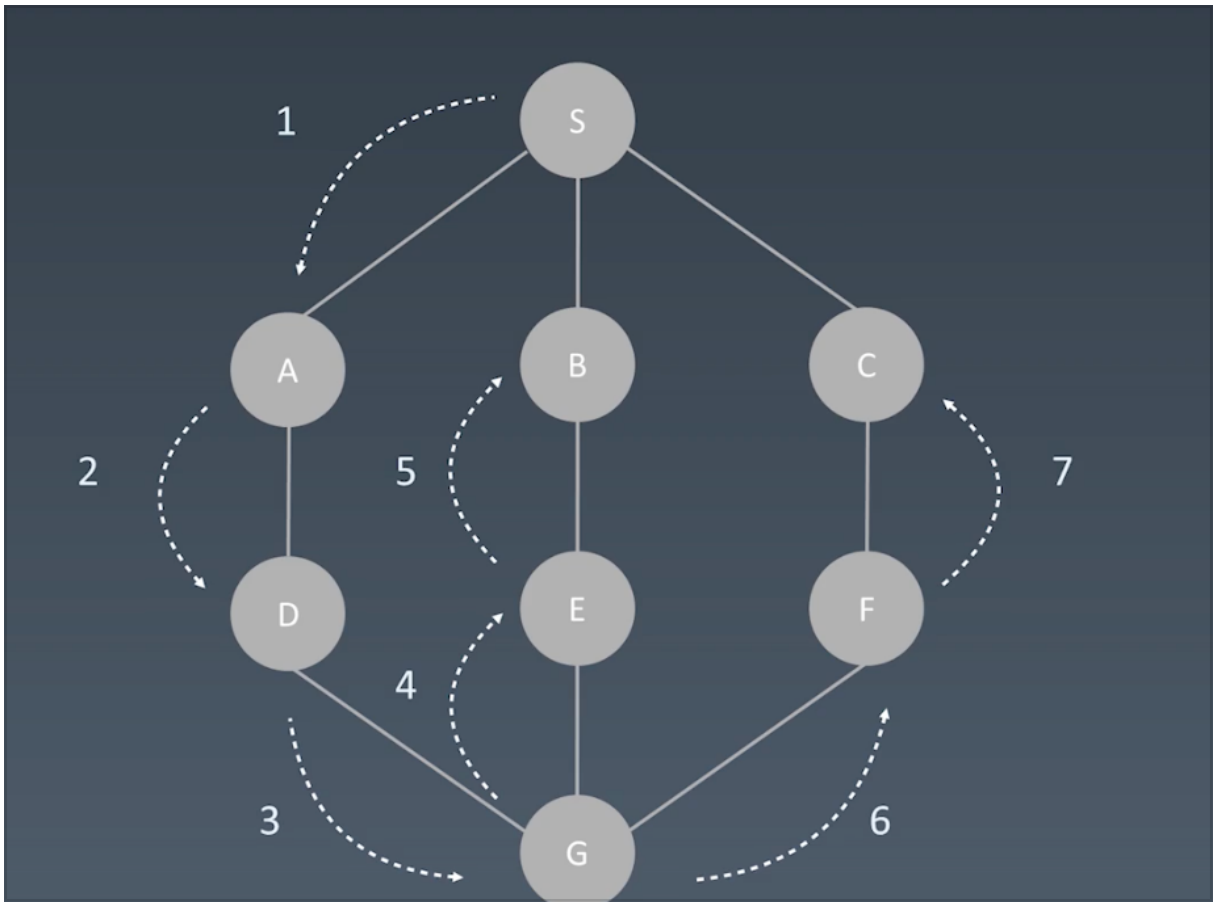
```
void bfs(Node *root)
{
    map<int, int> visited;
    if (!root) return;
    queue<Node *> queueNode;
    queueNode.push(root);
    while (!queueNode.empty())
    {
        Node *node = queueNode.top();
        queueNode.pop();
        if (visited.count(node->val)) continue;
        visited[node->val] = 1;
        for (int i = 0; i < node->children.size(); ++i)
        {
           queueNode.push(node->children[i]);
        }
    }
    return;
}
```

## DFS 深度优先搜索

- 树 DFS 顺序




- 图 DFS 顺序




示例代码

```
def dfs(node):

    if node in visited:
        # already visited
        return

    visited.add(node)

    # process current node
    # ... # logic here
    dfs(node.left)
    dfs(node.right)
```

## DFS 递归写法

树在做 DFS 和 BFS 时，没有环路，顶点不会重复，而图有环路，所以会重复，用 visited = set() 来保证顶点不重复.

```
visited = set() # 防止出现重复顶点

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)
```

```
map<int, int> visited;

void dfs(Node *root)
{
      // terminator
        if (!root) return;
      if (visited.count(root->val))
    {
            // already visited
            return;

    }
      visited[root->val] = 1;
        // process current node here.
        // ...
        for (int i = 0; i < root->children.size(); ++i)
    {
```

```
            dfs(root->children[i]);

    }
      return;
}



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 非递归写法

```
def dfs(self, tree):
```

```
    if tree.root is None:
        reutrn[]

    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
    ...
```

```
void dfs(Node *root)
{
    map<int, int> visited;
    if (!root) return;
    stack<Node *> stackNode;
    stackNode.push(root);
    while (!stackNode.empty())
    {
        Node *node = stackNode.top();
        stackNode.pop();
        if (visited.count(node->val)) continue;
        visited[node->val] = 1;
        for (int i = node->children.size() - 1; i >= 0; --i)
        {
            stackNode.push(node->children[i]);
        }
    }
  return;
}
```

#Algorithm/Part II : Theory/Algorithm#