**Topological sort in a Directed Graph**

**Parameters:**

* int n: (0,…,n-1)
* dependency
  + int[][]: [[1,0], [2,0]] pairs 0->1, 0->2
  + int[][]: [[], [0], [0]] 0->1, 0->2

**Note:**

1. return one of the correct orders.
2. if impossible, return an empty array.
3. pairs can be duplicate

**Algorithm: BFS/DFS**

**Time Complexity: O(m + n)**

**Code:**

use a map to store nodes that a node is pointing to. If the node is removed then all the nodes should have degree decreased by 1.

public int[] findOrder(int n, int[][] dep) {

**// BFS**

int[] degree = new int[n]; int[] result = new int[n];

Queue<Integer> queue = new LinkedList<>();

**Map<Integer, Set<Integer>> map = new HashMap<>();**

// create adjacent linked list

for (int[] i : dep) {

Set<Integer> set = map.get(i[1]);

if (set == null) {

set = new HashSet<>();

map.put(i[1], set);

}

**if (set.add(i[0]))** degree[i[0]]++;

}

// enqueues nodes with 0 degree

for (int i = 0; i < n; i++) {

if (degree[i] == 0) queue.add(i);

}

**int size = 0;**

while (!queue.isEmpty()) {

int val = queue.poll();

result[size++] = val;

if (map.containsKey(val)) {

for (int i : map.get(val)) {

degree[i]--;

if (degree[i] == 0) queue.add(i);

}

}

}

if (size == n) return result;

else return new int[0];

}

public class Solution {

**// DFS**

int[] result;

int cur;

public int[] findOrder(int n, int[][] dep) {

result = new int[n];

cur = n – 1;

int[] visited = new int[n];

// create adjacent linked list

Map<Integer, Set<Integer>> map = new HashMap<>();

for (int[] i : dep) {

Set<Integer> set = map.get(i[1]);

if (set == null) {

set = new HashSet<>();

map.put(int[1], set);

}

set.add(i[0]);

}

for (int i = 0; i < n; i++) {

if (visited[i] == 0) {

if (!findOrderDFS(map, i, visited))

return new int[0];

}

}

return result;

}

public boolean findOrderDFS(Map<Integer,Set<Integer>> map,

int node, int[] visited) {

if (map.containsKey(node)) {

visited[node] = 1; // visiting

for (int i : map.get(node)) {

if (visited[i] == 1) return false;

if (visited[i] == 0) {

if (!findOrderDFS(map, i, visited))

return false;

}

}

}

visited[node] = 2;

result[cur--] = node;

return true;

}

}

**Application:**

1. Course Schedule
2. Alien Dictionary

Derive the order of letters in this language.

[

"wrt",

"wrf",

"er",

"ett",

"rftt"

]

The correct order is: "wertf".

public String alienOrder(String[] words) {

Map<Character, Set<Character>> map = new HashMap<>();

~~Map<Character, Integer> degree = new HashMap<>();~~

// Don’t need to worry null -> 0

// We want to get an ordering of 26 characters

int[] degree = new int[26];

// Create adjacent linked list

for (int i = 1; i < word.length; i++) {

/\* … \*/

}

// topological sort

/\* … \*/

}

**Trie**

class TrieNode {

Map<Character, TrieNode> children

= new HashMap<Character, TrieNode>();

// TrieNode[] children = new TrieNode[26];

boolean isLeaf = false;

}

public class Trie {

private TrieNode root;

public Trie() { root = new TrieNode(); }

// Inserts a word into the trie.

public void insert(String word) {

if (word == null) return;

Map<Character, TrieNode> children = root.children;

for(int i = 0; i < word.length(); i++) {

TrieNode node = children.get(word.charAt(i));

if (node == null) {

node = new TrieNode();

children.put(word.charAt(i), node);

}

children = node.children;

if (i == word.length() - 1) node.isLeaf = true;

}

}

// Returns if the word is in the trie.

public boolean search(String word) {

if (word == null) return false;

Map<Character, TrieNode> children = root.children;

for (int i = 0; i < word.length(); i++) {

TrieNode node = children.get(word.charAt(i));

if (node == null) return false;

children = node.children;

if (i == word.length() - 1 && !node.isLeaf)

return false;

}

return true;

}

}

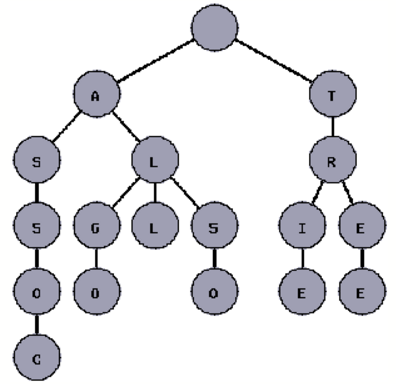
// Your Trie object will be instantiated and called as such:

// Trie trie = new Trie();

// trie.insert("somestring");

// trie.search("key");

Why use **tries** if **set<String>** and hash tables can do the same?

* insert and find strings in O(L) time (L is length of a single word) – much faster than set
* set and hash tables can only find in a dictionary words that match exactly with the single word that we are finding; the trie allows us to find words that have a single character different, a prefix in common, a character missing, etc.

[“tree”, “trie”, “algo”, “assoc”, “all”, “also”]

The root represents an empty string

each vertex represents a single word or a prefix

**Application:**

* web browser auto text completion and showing possibilities of the text
* orthographic check that every word typed is in a dictionary / suggested corrections of the word
* **Search Word**

addWord("bad")

addWord("dad")

addWord("mad")

search("pad") -> false

search("bad") -> true

search(".ad") -> true

search("b..") -> true

* **Search Word 2**

Given **words** = ["oath","pea","eat","rain"] and **board** =

[

['o','a','a','n'],

['e','t','a','e'],

['i','h','k','r'],

['i','f','l','v']

]

Return ["eat","oath"]

**Search word**

class TrieNode {

TrieNode[] children = new TrieNode[26];

boolean isLeaf = false;

}

public boolean search (TrieNode root, String word) {

if (root == null || word == null) return false;

if (word.length() == 0) return root.isLeaf;

TrieNode[] children = root.children;

if (word.charAt(0) == ‘.’) {

for (int i = 0; i < 26; i++) {

if (children[i] != null &&

search(children[i], word.substring(1)))

return true;

}

return false;

} else {

TrieNode trieNode = children[word.charAt(0)-‘a’];

if (trieNode == null) return false;

return search(trieNode, word.substring(1));

}

}

**Search word 2** Given **words** = ["oath","pea","eat","rain"]

**board** =

[

['o','a','a','n'],

['e','t','a','e'],

['i','h','k','r'],

['i','f','l','v']

]

Return ["eat","oath"]

**// DFS/BFS, note that there can be duplicated results**

public class Solution {

Set<String> result;

public List<String> findWords(char[][] board,

String[] words) {

result = new HashSet();

List<String> result = new ArrayList<String>();

if (...) return result;

int m = board.length, n = board[0].length;

TrieNode root = **buildTrie(words);**

// DFS

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++)

**findWordsDFS(root, board, i, j, visited, “”);**

}

result.addAll(result);

return result;

}

public void findWordsDFS(TrieNode root, char[][] board,

int i, int j, boolean[][] visited,

String pre) {

int m = board.length, n = board[0].length;

if (i<0 || i>=m || j<0 || j>=n || visited[i][j])

return;

TreeNode node = root.children[board[i][j]-‘a’];

if (node == null) return;

String str = pre + board[i][j];

if (node.isLeaf) result.add(str);

visited[i][j] = true;

findWordsDFS(node, board, i-1, j, visited, str);

findWordsDFS(node, board, i+1, j, visited, str);

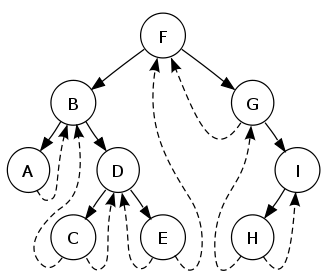
findWordsDFS(node, board, i, j-1, visited, str);

findWordsDFS(node, board, i, j+1, visited, str);

visited[i][j] = false;

}

}

**Binary Tree Traversal**

**In-order Traversal**

**// recursion | time: O(n) | space: stack O(log n)**

**// Iterative | time: O(n) | space: stack O(log n)**

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

if (root == null) return result;

Stack<**TreeNode**> stack = new Stack<>();

TreeNode node = root;

while (node != null || !stack.isEmpty()) {

if (node != null) {

stack.push(root);

node = node.left;

} else {

node = stack.pop();

result.add(node.val);

node = node.right;

}

}

return result;

}

**// Morris Traversal | Threaded Binary Tree**

**// time: O(n) space: O(1)**

如果cur左孩子为空,输出当前节点,cur赋给右孩子

如果cur左孩子不为空,寻找中序前驱节点 (cur左子树的最右子树)

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

TreeNode cur = root, pre = null;

while (cur != null) {

if (cur.left == null) {

result.add(cur.val);

cur = cur.right;

} else {

pre = cur.left;

while (pre.right != null && pre.right != cur)

pre = pre.right;

if (pre.right == null) {

pre.right = cur;

cur = cur.left;

} else {

pre.right = null;

result.add(cur.val);

cur = cur.right;

}

}

}

return result;

}

**Pre-order Traversal**

**// recursion**

**// Iterative**

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

if (root == null) return result;

**Stack<TreeNode>** stack = new Stack<>();

stack.push(root);

while (!stack.isEmpty()) {

TreeNode node = stack.pop();

result.add(node.val);

if (node.right != null) stack.push(node.right);

if (node.left != null) stack.push(node.left);

}

**return result;**

}

**// Morris Traversal | Threaded Binary Tree**

**// time: O(n) space: O(1)**

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

TreeNode cur = root, pre = null;

while (cur != null) {

if (cur.left == null) {

result.add(cur.val);

cur = cur.right;

} else {

pre = cur.left;

while (pre.right != null && pre.right != cur)

pre = pre.right;

if (pre.right == null) {

result.add(cur.val);

pre.right = cur.right;

cur = cur.left;

} else {

pre.right = null;

cur = cur.right;

}

}

}

return result;

}

**Post-order Traversal**

**// recursion**

**// interative**

**// use pre variable to decide whether the right subtree of a node has been visited or not**

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

if (root == null) return result;

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root, **pre = null**;

while (node != null || !stack.isEmpty()) {

if (node != null) {

stack.push(node);

node = node.left;

} else {

TreeNode peekNode = stack.peek();

**if (peekNode.right != null**

**&& pre != peekNode.right)** {

node = peekNode.right;

} else {

stack.pop();

result.add(peekNode.val);

pre = peekNode;

}

}

}

return result;

}

**Level Order Traversal**

**// Graph BFS | time: O(n) | space: O(n)**

**Level Order Traversal bottom-up**

1. **Reverse the result from BFS top-down**
2. **Recursion**
3. **Non-recursion: just like post-order traversal**

public List<List<Integer>> levelOrderBottom(TreeNode root) {

List<List<Integer>> list = new ArrayList<>();

if (root == null) return list;

List<List<Integer>> left = levelOrderBottom(root.left);

List<List<Integer>> right = levelOrderBottom(root.right);

int i = 0;

while (left.size()-i-1 >= 0 && right.size()-i-1 >= 0) {

left.get(left.size()-i-1)

.addAll(right.get(right.size()-i-1));

i++;

}

while (right.size() - i - 1 >= 0) {

left.add(0, right.get(right.size() - i - 1));

i++;

}

List<Integer> newList = new ArrayList<>();

newList.add(root.val);

left.add(newList);

return left;

}

**Zigzag Level Order Traversal**

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

Given, return:

3 [

/ \ [3],

9 20 [20,9],

/ \ [15,7]

15 7 ]

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null) return result;

boolean isLeft = true;

LinkedList <TreeNode> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

int curSize = queue.size();

List<Integer> list = new ArrayList<>();

**for (int i = 0; i < curSize; i++) {**

**TreeNode node = queue.poll();**

**list.add(node.val);**

**if (node.left != null) queue.add(node.left);**

**if (node.right != null) queue.add(node.right);**

**}**

**if (!isLeft) Collections.reverse(list);**

***for (int i = curSize - 1; i >= 0; i--) {***

***TreeNode node = queue.remove(i);***

***list.add(node.val);***

***if (isLeft) {***

***if (node.left != null) queue.add(node.left);***

***if (node.right != null) queue.add(node.right);***

***} else {***

***if (node.right != null) queue.add(node.right);***

***if (node.left != null) queue.add(node.left);***

***}***

***}***

isLeft = !isLeft;

result.add(list);

}

return result;

}