17/10/2018

Course Schedule:

class Solution:

"""

@param: numCourses: a total of n courses

@param: prerequisites: a list of prerequisite pairs

@return: true if can finish all courses or false

"""

def canFinish(self, numCourses, prerequisites):

# write your code here

indegree = [0] \* numCourses

neighborNode = {i:[] for i in range(numCourses)}

for i, j in prerequisites:

neighborNode[j].append(i)

indegree[i] += 1

dq, count = collections.deque([]), 0

for index in range(numCourses):

if indegree[index] == 0:

dq.append(index)

while dq:

course = dq.popleft()

count += 1

for neigh in neighborNode[course]:

indegree[neigh] -= 1

if indegree[neigh] == 0:

dq.append(neigh)

return count == numCourses

Topological Sorting

"""

Definition for a Directed graph node

class DirectedGraphNode:

def \_\_init\_\_(self, x):

self.label = x

self.neighbors = []

"""

class Solution:

"""

@param graph: A list of Directed graph node

@return: A list of integer

"""

def topSort(self, graph):

node\_to\_indegree = self.get\_indegree(graph)

# bfs

order = []

start\_nodes = [n for n in graph if node\_to\_indegree[n] == 0]

queue = collections.deque(start\_nodes)

while queue:

node = queue.popleft()

order.append(node)

for neighbor in node.neighbors:

node\_to\_indegree[neighbor] -= 1

if node\_to\_indegree[neighbor] == 0:

queue.append(neighbor)

return order

def get\_indegree(self, graph):

node\_to\_indegree = {x: 0 for x in graph}

for node in graph:

for neighbor in node.neighbors:

node\_to\_indegree[neighbor] += 1

return node\_to\_indegree

DFS

"""

Definition for a Directed graph node

class DirectedGraphNode:

def \_\_init\_\_(self, x):

self.label = x

self.neighbors = []

"""

class Solution:

"""

@param graph: A list of Directed graph node

@return: A list of integer

"""

def topSort(self, graph):

indegree = {}

for x in graph:

indegree[x] = 0

for i in graph:

for j in i.neighbors:

indegree[j] += 1

ans = []

for i in graph:

if indegree[i] == 0:

self.dfs(i, indegree, ans)

return ans

def dfs(self, i, indegree, ans):

ans.append(i)

indegree[i] -= 1

for j in i.neighbors:

indegree[j] -= 1

if indegree[j] == 0:

self.dfs(j, indegree, ans)

Sequence Reconstruction

public class Solution {

/\*\*

\* @param org a permutation of the integers from 1 to n

\* @param seqs a list of sequences

\* @return true if it can be reconstructed only one or false

\*/

public boolean sequenceReconstruction(int[] org, int[][] seqs) {

// Write your code here

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

Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

for (int num : org) {

map.put(num, new HashSet<Integer>());

indegree.put(num, 0);

}

int n = org.length;

int count = 0;

for (int[] seq : seqs) {

count += seq.length;

if (seq.length >= 1 && (seq[0] <= 0 || seq[0] > n))

return false;

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

if (seq[i] <= 0 || seq[i] > n)

return false;

if (map.get(seq[i - 1]).add(seq[i]))

indegree.put(seq[i], indegree.get(seq[i]) + 1);

}

}

// case: [1], []

if (count < n)

return false;

Queue<Integer> q = new ArrayDeque<Integer>();

for (int key : indegree.keySet())

if (indegree.get(key) == 0)

q.add(key);

int cnt = 0;

while (q.size() == 1) {

int ele = q.poll();

for (int next : map.get(ele)) {

indegree.put(next, indegree.get(next) - 1);

if (indegree.get(next) == 0) q.add(next);

}

if (ele != org[cnt]) {

return false;

}

cnt++; }

return cnt == org.length;} }

class Solution:

# @param {int[]} org a permutation of the integers from 1 to n

# @param {int[][]} seqs a list of sequences

# @return {boolean} true if it can be reconstructed only one or false

def sequenceReconstruction(self, org, seqs):

from collections import defaultdict

edges = defaultdict(list)

indegrees = defaultdict(int)

nodes = set()

for seq in seqs:

nodes |= set(seq)

for i in xrange(len(seq)):

if i == 0:

indegrees[seq[i]] += 0

if i < len(seq) - 1:

edges[seq[i]].append(seq[i + 1])

indegrees[seq[i + 1]] += 1

cur = [k for k in indegrees if indegrees[k] == 0]

res = []

while len(cur) == 1:

cur\_node = cur.pop()

res.append(cur\_node)

for node in edges[cur\_node]:

indegrees[node] -= 1

if indegrees[node] == 0:

cur.append(node)

if len(cur) > 1:

return False

return len(res) == len(nodes) and res == org

20/10/2018

Clone Graph

"""

Definition for a undirected graph node

class UndirectedGraphNode:

def \_\_init\_\_(self, x):

self.label = x

self.neighbors = []

"""

class Solution:

"""

@param: node: A undirected graph node

@return: A undirected graph node

"""

def cloneGraph(self, node):

# write your code here

root = node

if root is None:

return node

nodes = self.getAllNodes(node)

mapping = {}

for node in nodes:

mapping[node] = UndirectedGraphNode(node.label)

for node in nodes:

new\_node = mapping[node]

for neighbor in node.neighbors:

new\_neighbor = mapping[neighbor]

new\_node.neighbors.append(new\_neighbor)

return mapping[root]

def getAllNodes(self, node):

dq = collections.deque([node])

result = set([node])

while dq:

head = dq.popleft()

for neighbor in head.neighbors:

if neighbor not in result:

result.add(neighbor)

dq.append(neighbor)

return result

DFS

class Solution:

def \_\_init\_\_(self):

self.dict = {}

"""

@param: node: A undirected graph node

@return: A undirected graph node

"""

def cloneGraph(self, node):

if node is None:

return None

if node.label in self.dict:

return self.dict[node.label]

root = UndirectedGraphNode(node.label)

self.dict[node.label] = root

for item in node.neighbors:

root.neighbors.append(self.cloneGraph(item))

return root

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* ArrayList<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) {

return node;

}

// use bfs algorithm to traverse the graph and get all nodes.

ArrayList<UndirectedGraphNode> nodes = getNodes(node);

// copy nodes, store the old->new mapping information in a hash map

HashMap<UndirectedGraphNode, UndirectedGraphNode> mapping = new HashMap<>();

for (UndirectedGraphNode n : nodes) {

mapping.put(n, new UndirectedGraphNode(n.label));

}

// copy neighbors(edges)

for (UndirectedGraphNode n : nodes) {

UndirectedGraphNode newNode = mapping.get(n);

for (UndirectedGraphNode neighbor : n.neighbors) {

UndirectedGraphNode newNeighbor = mapping.get(neighbor);

newNode.neighbors.add(newNeighbor);

}

}

return mapping.get(node);

}

private ArrayList<UndirectedGraphNode> getNodes(UndirectedGraphNode node) {

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

HashSet<UndirectedGraphNode> set = new HashSet<>();

queue.offer(node);

set.add(node);

while (!queue.isEmpty()) {

UndirectedGraphNode head = queue.poll();

for (UndirectedGraphNode neighbor : head.neighbors) {

if(!set.contains(neighbor)){

set.add(neighbor);

queue.offer(neighbor);

}

}

}

return new ArrayList<UndirectedGraphNode>(set);

}

}

10/21/2018

Serialize and Deserialize Binary Tree

"""

Definition of TreeNode:

class TreeNode:

def \_\_init\_\_(self, val):

self.val = val

self.left, self.right = None, None

"""

class Solution:

"""

@param root: An object of TreeNode, denote the root of the binary tree.

This method will be invoked first, you should design your own algorithm

to serialize a binary tree which denote by a root node to a string which

can be easily deserialized by your own "deserialize" method later.

"""

def serialize(self, root):

# write your code here

if root is None:

return '#'

dq = collections.deque([root])

res = []

while dq:

node = dq.popleft()

if node is None:

res.append('#')

else:

res.append(str(node.val))

dq.append(node.left)

dq.append(node.right)

return res

"""

@param data: A string serialized by your serialize method.

This method will be invoked second, the argument data is what exactly

you serialized at method "serialize", that means the data is not given by

system, it's given by your own serialize method. So the format of data is

designed by yourself, and deserialize it here as you serialize it in

"serialize" method.

"""

def deserialize(self, data):

# write your code here

if data[0] == '#':

return None

root = TreeNode(int(data.pop(0)))

dq = collections.deque([root])

isLeft = True

while data:

s = data.pop(0)

if s != '#':

node = TreeNode(int(s))

dq.append(node)

if isLeft:

dq[0].left = node

else:

dq[0].right = node

if not isLeft:

dq.popleft()

isLeft = not isLeft

return root

#DFS solution

class Solution:

def serialize(self, root):

# write your code here

if not root:

return ['#']

ans = []

ans.append(str(root.val))

ans += self.serialize(root.left)

ans += self.serialize(root.right)

return ans

def deserialize(self, data):

# write your code here

ch = data.pop(0)

if ch == '#':

return None

else:

root = TreeNode(int(ch))

root.left = self.deserialize(data)

root.right = self.deserialize(data)

return root

public class Solution {

/\*

\* @param nums: A set of numbers

\* @return: A list of lists

\*/

public List<List<Integer>> subsets(int[] nums) {

// List vs ArrayList （google）

List<List<Integer>> results = new LinkedList<>();

if (nums == null) {

return results; // 空列表

}

Arrays.sort(nums);

// BFS

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

queue.offer(new ArrayList<Integer>());

while (!queue.isEmpty()) {

List<Integer> subset = queue.poll();

results.add(subset);

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

if (subset.size() == 0 || subset.get(subset.size() - 1) < nums[i]) {

List<Integer> nextSubset = new ArrayList<Integer>(subset);

nextSubset.add(nums[i]);

queue.offer(nextSubset);

}

}

}

return results;

}

}