**Challenge: Maintain and Traverse a Graph**

**Coding Challenge**

**import java.io.BufferedReader;**

**import java.io.BufferedWriter;**

**import java.io.File;**

**import java.io.FileReader;**

**import java.io.FileWriter;**

**import java.io.IOException;**

**import java.util.HashMap;**

**import java.util.Scanner;**

**import java.util.Set;**

**import java.util.HashSet;**

**import java.util.List;**

**import java.util.ArrayList;**

**public class Solution {**

**List<String> input;**

**HashMap<Integer,Set<Integer>> hm;**

**List<String> output;**

**public Solution(){**

**input=new ArrayList<String>();**

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

**output=new ArrayList<String>();**

**}**

**public void MaintainTraverseGraph(List<String> inputs){**

**setInput(inputs);**

**for(String command:getInput()){**

**//parse command**

**String[] strarray=command.split(" ",0);**

**if(strarray.length<3 || strarray.length>4){**

**System.out.println("Invalid input:"+command+", continue....");**

**continue;**

**}**

**HashMap<Integer,Set<Integer>> graph=getGraph();**

**if(strarray.length==3){**

**int value1=Integer.valueOf(strarray[1]);**

**int value2=Integer.valueOf(strarray[2]);**

**//Add command**

**if(strarray[0].equalsIgnoreCase("add")&&**

**strarray.length==3 &&**

**strarray[1].matches("\\d+")&&**

**strarray[2].matches("\\d+")){**

**//Both nodes exist**

**if(graph.containsKey(value1) && graph.containsKey(value2)){**

**//If they have not linked to each other yet, then link them**

**if(!isLinked(value1,value2,graph)){**

**graph.get(value1).add(value2);**

**graph.get(value2).add(value1);**

**}**

**//If they are already linked to each other, then do nothing**

**}**

**//Only node1 exists**

**else if(graph.containsKey(value1)){**

**graph.get(value1).add(value2);**

**Set<Integer> set=new HashSet<Integer>();**

**set.add(value1);**

**graph.put(value2, set);**

**}**

**//Only node2 exists**

**else if(graph.containsKey(value2)){**

**graph.get(value2).add(value1);**

**Set<Integer> set=new HashSet<Integer>();**

**set.add(value2);**

**graph.put(value1, set);**

**}**

**//Both nodes don't exist**

**else{**

**Set<Integer> set1=new HashSet<Integer>();**

**set1.add(value2);**

**graph.put(value1, set1);**

**Set<Integer> set2=new HashSet<Integer>();**

**set2.add(value1);**

**graph.put(value2, set2);**

**}**

**}**

**//Remove Command**

**else if(strarray[0].equalsIgnoreCase("remove")&&**

**strarray.length==3 &&**

**strarray[1].matches("\\d+")&&**

**strarray[2].matches("\\d+")){**

**//Both nodes exist**

**if(graph.containsKey(value1) && graph.containsKey(value2)){**

**//If they are already linked to each other, then remove the link**

**if(isLinked(value1,value2,graph)){**

**graph.get(value1).remove(value2);**

**graph.get(value2).remove(value1);**

**}**

**//If they have not linked to each other yet, then do nothing**

**}**

**//If only one node or both nodes exist(s), then do nothing**

**}**

**//Other invalid command, just continue**

**else{**

**System.out.println("Invalid input:"+command+", continue....");**

**continue;**

**}**

**}**

**if(strarray.length==4){**

**int value1=Integer.valueOf(strarray[2]);**

**int value2=Integer.valueOf(strarray[3]);**

**//Is Linked Command**

**if(strarray[0].equalsIgnoreCase("is")&&**

**strarray[1].equalsIgnoreCase("linked")&&**

**strarray.length==4&&**

**strarray[2].matches("\\d+")&&**

**strarray[3].matches("\\d+")){**

**getOutput().add(String.valueOf(isLinked(value1, value2, graph)));**

**}**

**//Other invalid command, just continue**

**else{**

**System.out.println("Invalid input:"+command+", continue....");**

**continue;**

**}**

**}**

**}**

**}**

**public boolean isLinked(int value1,int value2, HashMap<Integer,Set<Integer>> graph){**

**if(!graph.containsKey(value1) || !graph.containsKey(value2)) return false;**

**HashMap<Integer,Boolean> visited=new HashMap<Integer,Boolean>(graph.size());**

**for(Integer key:graph.keySet()){**

**visited.put(key, false);**

**}**

**return isLinkedhelper(value1,value2,graph,visited);**

**}**

**public boolean isLinkedhelper(int value1,int value2, HashMap<Integer,Set<Integer>> graph, HashMap<Integer,Boolean> visited){**

**visited.replace(value1,true);**

**boolean result=false;**

**//BFS search**

**for(Integer integer2:graph.get(value1)){**

**if(integer2==value2){**

**result=true;**

**return result;**

**}**

**}**

**if(!result){**

**for(Integer integer2:graph.get(value1)){**

**if(visited.get(integer2)==false){**

**return isLinkedhelper(integer2, value2,graph,visited);**

**}**

**}**

**}**

**// Reverse Check**

**// for(Integer integer1:graph.get(value2)){**

**// if(integer1==value1) result&=true;**

**// }**

**//**

**// if(!result){**

**// for(Integer integer1:graph.get(value2)){**

**// isLinked(integer1, value1,graph);**

**// }**

**// }**

**return result;**

**}**

**public void printInput(){**

**System.out.println("-----Input-----");**

**for(String command:getInput()){**

**System.out.println(command);**

**}**

**}**

**public void printOutput(){**

**System.out.println("-----Output-----");**

**for(String command:getOutput()){**

**System.out.println(command);**

**}**

**}**

**public void setInput(List<String> inputs){**

**this.input=inputs;**

**}**

**public List<String> getInput(){**

**return this.input;**

**}**

**public void setOutput(boolean output){**

**getOutput().add(String.valueOf(output));**

**}**

**public List<String> getOutput(){**

**return this.output;**

**}**

**public HashMap<Integer,Set<Integer>> getGraph(){**

**return this.hm;**

**}**

**public void printGraph(){**

**System.out.println("----Graph----");**

**for(Integer key:getGraph().keySet()){**

**System.out.print("Node: "+key+" ");**

**System.out.print("Adjacent Nodes: ");**

**for(Integer value:getGraph().get(key)){**

**System.out.print(value+" ");**

**}**

**System.out.println();**

**}**

**}**

**public static void main(String[] args) throws IOException {**

**FileReader fr=new FileReader("input.txt");**

**BufferedReader br = new BufferedReader(fr);**

**String thisline;**

**List<String> input1=new ArrayList<String>();**

**while((thisline=br.readLine())!=null){**

**input1.add(thisline);**

**}**

**br.close();**

**Solution sol1=new Solution();**

**sol1.MaintainTraverseGraph(input1);**

**sol1.printInput();**

**sol1.printOutput();**

**sol1.printGraph();**

**File outputfile=new File("output.txt");**

**if(!outputfile.exists()){**

**outputfile.createNewFile();**

**}**

**//Here false is to not append the content to file**

**FileWriter fw=new FileWriter(outputfile);**

**BufferedWriter bw=new BufferedWriter(fw);**

**for(String result:sol1.getOutput()){**

**bw.write(result+"\n");**

**}**

**bw.flush();**

**bw.close();**

**//Test Case 2:**

**List<String> input2=new ArrayList<String>();**

**input2.add("is linked 1 4");**

**Solution sol2=new Solution();**

**sol2.MaintainTraverseGraph(input2);**

**sol2.printInput();**

**sol2.printOutput();**

**sol2.printGraph();**

**//Test Case 3:**

**List<String> input3=new ArrayList<String>();**

**input3.add("remove 1 2");**

**input3.add("add 1 2");**

**input3.add("add 2 3");**

**input3.add("add 1 3");**

**input3.add("add 3 4");**

**input3.add("add 5 6");**

**input3.add("is linked 1 1");**

**input3.add("is linked 1 4");**

**input3.add("is linked 5 6");**

**input3.add("is linked 1 6");**

**input3.add("remove 1 3");**

**input3.add("is linked 4 1");**

**input3.add("remove 5 6");**

**input3.add("is linked 5 6");**

**Solution sol3=new Solution();**

**sol3.MaintainTraverseGraph(input3);**

**sol3.printInput();**

**sol3.printOutput();**

**sol3.printGraph();**

**}**

**}**

**Write Up:**

**(1) What is the memory footprint for your approach? What is the running time for each add link, remove link, and is linked?**

1. **Space Complexity (SC):**

In each test case, there are 3 major data structures used:

List that stores input commands (SC=O(#commands)),

List that stores output result (SC=O(#isLinkedcommands)),

HashMap<key, value> for nodes and node-relation (SC=O(#nodes\*#adjacent nodes))

e.g.

(Node is key and Adjacent Nodes is value that stores a list of adjacent nodes)

Node: 1 Adjacent Nodes: 2

Node: 2 Adjacent Nodes: 1 3

Node: 3 Adjacent Nodes: 2 4

Node: 4 Adjacent Nodes: 3

If the commands include is linked command, there is an extra space for visited-nodes status hashmap (SC=O(#nodes))

So, the total SC=O(#commands+(#nodes^2)

The reason to use HashMap is because the number of nodes >> the number of links, we need a faster way to search/access a particular node and a less fast way to search/access adjacent nodes for in linked command.

**(b) Time Complexity (TC):**

**(B.1) For add link running time,**

1st-step:

find whether the two node exist by using hashmap (TC=O(2))

2nd –step:

condition1: nodes both exist, find whether they have a link (TC=is-linked TC= O(#nodes))

(explain later)

condition2: one node exists, this node adds adjacent node (TC=O(1))

add a new node and it adjacent node (TC=O(1))

condition3: nodes both don’t exist, add new 2 nodes and their adjacent nodes (TC=O(2))

So, the critical running time is at condition1: is-linked TC= O(#nodes) (explain later)

**(B.2) For is linked running time,**

1ts-step:

initialize the visited-nodes status hashmap (TC=O(#nodes))

2nd-step:

use Recursion and Breadth-First-Search methods to trace the link between the two nodes

e.g. is linked 1 4

Node: 1 Adjacent Nodes: 2

Node: 2 Adjacent Nodes: 1 3

Node: 3 Adjacent Nodes: 2

Node: 4 Adjacent Nodes:

Fine Node 1 and traverse its Adjacent Nodes 2, if the adjacent nodes != 4,

then find Node 2 and traverse its Adjacent Nodes 1 & 3, if the adjacent nodes != 4,

then find Node 1 ……

and so on until fine the adjacent nodes = 4 or the nodes are visited.

Find Nodes (TC=O(#nodes)) and traverse Adjacent Nodes (TC=O(#links)=O(at most #nodes\*2)). So, the total TC of is-linked =O(#nodes)

**(B.3) For remove running time,**

1ts-step:

find whether the two node exist (TC=O(2))

2nd –step:

condition1: nodes both exist, judge whether they has a link(TC=is-linked TC= O(#nodes))

condition2: one node exists, don’t need to remove (TC=O(1))

condition3: nodes both don’t exist, don’t need to remove (TC=O(1))

So, the critical running time is at condition1: is-linked TC= O(#nodes)

**(2) Our application has too many nodes to keep them all in memory on one machine. How would you handle this in a real world application? There are many ways to approach this issue, but if you choose to go the database route please describe which database, your tables, indexes, queries, and performance. If you have another approach, please describe it in detail.**

Because there are too many nodes, NoSQL database can be used, which concerns about 3V (Velocity, Volume, Variety). NoSQL database usually has features like key-value pair, column-based table, graph relation diagram, in-memory processing, MapReduce processing framework, partitioning, parallel process, and so on.

**(a) For database table storage concern:**

In the coding example above, I can create a new class, called Node that stores different information in it. The tables are: (1) Node\_list (int node\_id, String node\_info), (2) Adjacent\_Nodes (int node\_id primary key, int adjacent\_node\_id). In (2) Adjacent\_Nodes Table, we can modified it to Linked\_Nodes(int node\_id foreign key, int link\_node\_id foreign key). Although the modification would increase the search speed, it would take more storage space (more rows) and more time of adding and removing link (Every time we add/remove a link, we not just only add/remove two nodes and the link between them, but also the other nodes which can link to these two nodes too.). So, based on the storage space concern in the question, it would be better to use Adjacent\_Nodes Table than Linked\_Nodes Table.

**(b) For read/write concern:**

We can store the frequent queried nodes in memory so that users can read nodes fast, even do some simple calculation in the memory and store the result in memory. When the memory is going to be full, the result can then be stored back into database.

**(c) For distributed processing concern:**

The NoSQL database can store the data/rows into different workers and process them in parallel, which increases the processing speed.