WIA1002/WIB1002 Data Structure
Semester 2 2018/2019
Assignment

Read the instruction carefully, form your group, and complete the given task.

Group formation

1. Form a group of not more than 4 members. All the members must be from the same tutorial group.
2. Each of the groups will be given one of the listed projects randomly.
3. Every group member must contribute to the project, including certain amount of coding.
4. The role played by and contribution of each member to the project must be included in the managerial report.

Submission

Your group has to submit the softcopy of the items listed below:

1. A technical report explaining the assigned task, the requirements of the task, the approach taken to solve the task, a detail description of your solution (including the flowchart, modules, etc.), sample snapshot of your program output.
2. A managerial report explaining the formation of the group, role and assigned work for each of the members, the project timeline, the problems faced in accomplishing this assignment and your solutions, and other issues encountered.
3. The complete source code.

For softcopy, the group leader will submit them in one .zip file to Spectrum. The name of the zip file should be *TutorialGroupNumber-GroupLeaderName.zip.*

Important dates

1. Submission: (week 13th) 12:00pm on Friday, 24th May 2018
2. Viva/demo: (week 14th) during tutorial & lab

Marking scheme: (20 marks)

1. Source code/Program. (18 marks)
   a. Meeting the basic task requirements, i.e. identify the requirements of the task and produce a workable solution - *10 marks*
   b. Extra features/functionalities that are not included in the task specification - *6 marks*.

2. Reports (4 marks)

   In your technical report and during viva, justify the design of your solution and the choice of **data structures** being used, it may include **technical difficulties*** that are impossible to achieve or any other valid reasons.

   * *Subjected to your lecturer's judgement based on your justifications*

Warning: **Plagiarism** is prohibited, should there be any evidence found, it will result in a **heavy penalty** on your assignment grades!

Good luck and have fun!

Topic 1: CrabFood

Introduction

Citizens in Crabby Island (known as the Crabbians) do have a crabby culture, you can never imagine how large are their loves towards crab dishes. The most recent statistics showed that every citizen in Crabby Island orders at least 3 crab dishes every day. Your company, Crab has seen this as a golden business opportunity and decides to come up with an evolutionary CrabFood to provide crab delivery service for the Crabbians. Now, you are the selected programmers to develop this CrabFood system. Make sure that you come up with an amazing product that can help improve the Crabbians' lives.

Problem Statement

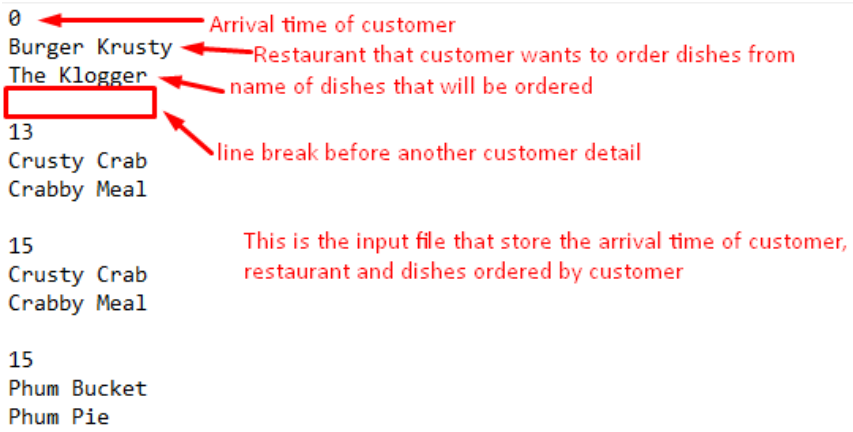
Your team is given this CrabFood project for 10,000 CrabCoins (currency of Crabby Island, CC). After analyzing the Crabbians' feedback, as well as having some serious meetings with the top management, your team finally finalized the requirements for CrabFood. CrabFood is defined as a Desktop application that manages and keeps track of daily delivery order for crabs to ensure the efficiency of delivery services, just think about GrabFood.

To shorten your work, your team has come up with a checklist for the requirements:

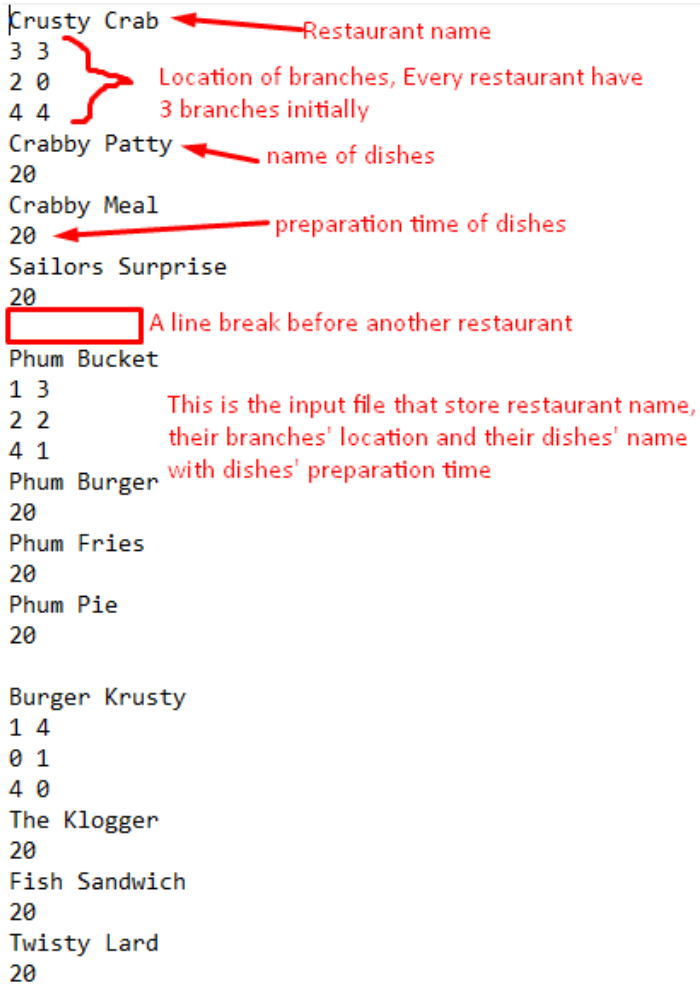
1. Make a Java console application that shows the delivery process.
2. Make a logging system that shows the entire process when CrabFood is up.
3. Make a reporting system that displays daily order information for every restaurant. This is explicitly mentioned by the restaurants that partner with CrabFood for them to improve their services.
4. You will be given sample input files to help in your development, utilize them wisely.

Sample Input

Customer.txt



Input.txt



Sample Output

```
0B000
000PB
C0P00
000C0
BP00C
0: a new day has start!
0: Customer 1 wants to order The Klogger from Burger Krusty.
0: Branch of Burger Krusty at (0, 1) take the order.
13: Customer 2 wants to order Crabby Meal from Crusty Crab.
13: Branch of Crusty Crab at (2, 0) take the order.
15: Customer 3 wants to order Crabby Meal from Crusty Crab.
15: Branch of Crusty Crab at (3, 3) take the order.
15: Customer 4 wants to order Phum Pie from Phum Bucket.
15: Branch of Phum Bucket at (2, 2) take the order.
20: Branch of Burger Krusty at (0, 1) finish the order and delivery the food to customer 1.
21: The food have been delivered to customer 1
33: Branch of Crusty Crab at (2, 0) finish the order and delivery the food to customer 2.
35: The food have been delivered to customer 2
35: Branch of Crusty Crab at (3, 3) finish the order and delivery the food to customer 3.
35: Branch of Phum Bucket at (2, 2) finish the order and delivery the food to customer 4.
39: The food have been delivered to customer 4
41: The food have been delivered to customer 3
41: All Customer served and shops are closed.
```

Print out the map with the first char of the restaurant name represent the restaurant's branch

Print out all info with the timestamp

Print out order of customer with customer ID

Print out the branch that take the order

Print when a order is finished and delivery is started

Print when the food have been delivered to customer

Print the end of the day message when all customers sucessfully served

Example log file

Customer	Arrival	Order Time	Finished Cooking Time	Delivery Time	Total Time
1	0	0	20	1	21
2	13	13	33	2	22
3	15	15	35	6	26
4	15	15	35	4	24

Customer	Arrival	Order Time	Finished Cooking Time	Delivery Time	Total Time
1	0	0	20	1	21
2	13	13	33	2	22
3	15	15	35	6	26
4	15	15	35	4	24

Example of log file

Assumption you can make

1. All the users are at the same position (0, 0).
2. All the restaurants are scattered around on the map.
3. Each user will order one dishes per time.
4. All dishes have the same preparation time. (You may set it yourselves or follow the input file)
5. User will specify which restaurant he/she wants, but not the specific branch.
6. After a user places an order, the order will be processed by CrabFood, and sent to the branch that can complete the order in shortest time. (total duration = time to complete previous order if exists + time to prepare the order + time to deliver to the customer)
7. Time to deliver to the customer is calculated based on the distance from the branch to customer. (e.g. time from A to customer based on above map is 0 + 3 = 3)
8. All restaurants must not reject an order assigned.
9. Logging system must show the timestamp of actions.
10. Reporting system must show the start time, end time and duration of delivery in daily basis.

Some crazy idea

1. There are only a limited amount of the food deliveryman. So the customers need to wait until their turn to get their food delivered.
2. Different starting point of customer? And the customer will choose the nearest shop available first to order dishes.
3. GUI?
4. The customers are able to order multiple dishes?
5. The shops can prepare multiple dishes at the same time, but they cannot prepare 2 same type of dishes at the same time.
6. Larger Map? More shops? More branches? More dishes?
7. Customer can cancel the order and the shop can keep the dishes they order for next customer (if any customer want that dishes), but if the dish wait for too long it is no longer fresh and need to be dumped
8. Customer can have special requirement
9. The delivery point is not necessarily same location with where the customer is.
10. Traffic jam issue occur, so shorter distance doesn't mean shorter time.

Topic 2: not-git

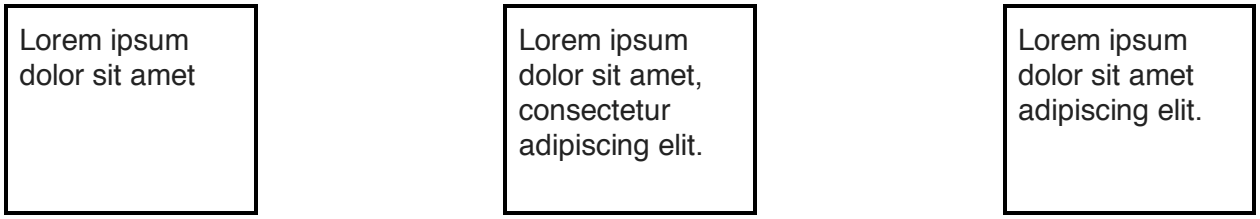
Introduction

According to wikipedia, **git** is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

A version-control system (VCS) allows you to revert project files back to a previous state, review changes made over time and more. Using a VCS also means that if you screw things up or lose files, you can generally recover easily. And sometimes you just want to know **"who wrote this crap"**, and having access to that information is worthwhile.

But you don't really need to know all that because you will be developing **not-git**, which requires no knowledge of git or whatsoever.

Basically, **not-git** is a file tracker. It helps you keep track of the changes made to your file. Each time you save your files into **not-git**, it stores it permanently in a timeline, where you can retrieve any of them later. Example below shows the timeline (version history) of the *file.txt*:



Version 1 Version 2 Version 3

Why not just save the files into multiple copies? Well, it all gets messy quickly isn't it? With **not-git** you only need one copy of the file, and the old copies are stored in the **not-git** itself. Save you from the headache of going through all the mess.

With **not-git** you can easily go back to version 2 or version 1 of *file.txt*. You can jump to and from to any version of the file in the **not-git** time continuum.

Problem Statement

How to **not-git**?

Since **not-git** will be your own project, most of the implementation details are up to you to decide. However, you must satisfy certain characteristics and requirements. Below are some terms and logic you should use and implement:

Repository

Since we will be tracking files changes, there must be some places where you are storing all the files. We refer to this location (usually a folder) as a repository.

Structure

not-git stores the file history in a **list**. A doubly linked-list is recommended, however for advanced usage you should consider implementing a tree. Each node in the list is a point in time, user should be able to move to any of the node, and restore the files to the states they are in during the point.

Add

Each individual files should be added to **not-git** to request the program to start tracking it. Files that are not added are ignored.

Commit

Commit is the term used when you store the files into **not-git**. When you commit the files, you save the files and their current states into **not-git**'s structure. User should be able to retrieve this commit using "revert" feature. Each commit should has its own commit id.

Revert

Revert is one of the feature that need to be implemented. Revert is used to restore your files back to the state they are in a particular previous commit. When revert is used, a commit is made as well to keep the history clear.

Status

Status is one of the feature that need to be implemented. Status should shows the files that are being tracked by not-git, as well as if any of the files were different (new, modified or deleted).

not-git Commands

not-git

When you run the program, a new empty list is created.

not-git //run the program at the repository folder
Welcome to not-git!

add

Add files to-be-tracked to start tracking the files.

add “file.txt”
“file.txt” is now being tracked

commit

Create a version history of the current state of the working directory. A commit-id should be assigned to each commit made.

commit
files commited with commit id “1”

revert

In simple term, revert is undo. You can revert to any previous commit, but each time revert is used, a new commit is created and push to the end of the list, instead of removing latest commits. This is to ensure the commits are still accessible.

revert 1 // commit-id “1”
revert to commit-id “1” and committed with commit-id “3”

status

not-git-status should check for modified files and untracked files.

status
No file is being tracked.
status
tracked files:
file.txt (modified)

Sample Output

Below is the example of how you can run **not-git** and use it in your folder. **Bold** text are the commands while normal text is the output.

not-git //run the program at the repository folder
Welcome to not-git!

status
No file is being tracked.

add "file.txt"
"file.txt" is now being tracked

status
tracked files:
file.txt (new)

commit
files commited with commit id "1"

status
tracked files:
file.txt

// edit file.txt

status
tracked files:
file.txt (modified)

commit
files commited with commit-id "2"

status
tracked files:
file.txt

revert 1 // commit-id "1"
revert to commit-id "1" and committed with commit-id "3"

Some crazy idea

1. A Better **not-git** (*Extra Requirements*)

Implementing a list and saving all text into the list is easy. Complete them and you get the basic marks! But to achieve higher grades, you will need to implement more features:

A Better status

not-git should scan the whole folder and list out all the files including untrack ones, and output the files status.

status
new files:
    just_added.txt
modified files:
    i_was_modified.txt
tracked files:
    file.txt
untracked files:
    track_me_please.txt

A Better **not-git**

Previously, each time you run not-git, a new list is created, the list is populated as you insert new commands into the program. When the program is closed, the list is destroyed together with all the old copies of files. Now, you need to figure out a way to save the created list along with all the data in it. Each time you run not-git, it first determine if any history is present, if not, it create a new empty list; if file history is already present, it will use it and continue from there instead. Be it ObjectOutputStream or even your own SQL database, anything that can provide a persistent storage will satisfy this requirement.

not-git
Welcome to not-git!
status
tracked files:
    file.txt
    just_added.txt
    i_was_modified.txt
untracked files:
    track_me_please.txt

A Better commit

commit should requires a description message to better identify each commit.

commit
description: **added just_added.txt and edited i_was_modified.txt**
files commited with commit id “18”

New command - log

log command will output the list of commits, along with the respective commit-id, timestamp and description.

log
commit “1” - 10:02 15-02-2019
“initial commit”
commit “2” - 18:27 16-02-2019
“edited file.txt”
.
.
.
commit “18” - 11:35 11-03-2019
“added just_added.txt and edited i_was_modified.txt”

New command - diff

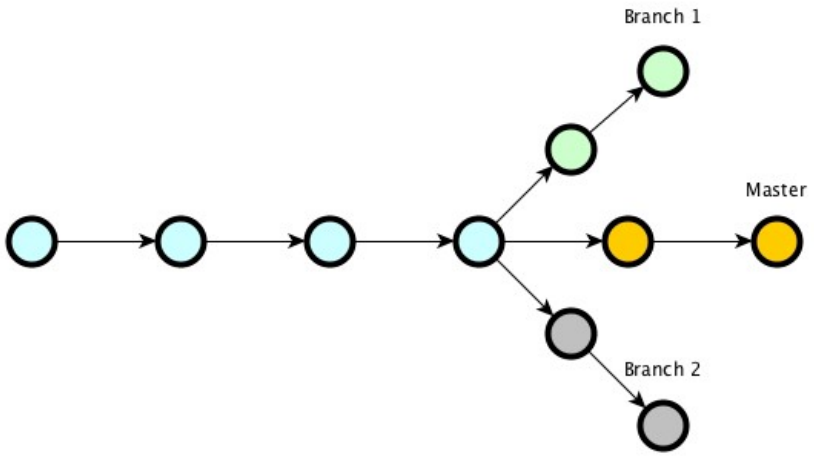
diff command should output the changes made to a file since the latest commit.

diff i_was_modified.txt

```
+ a line was added, notice the “+” sign
+ second line added here
- This line was deleted, notice the “-” sign
```

2. The Ultimate not-git (Bonus)

So you want an A+? You have to prove that you deserve it. Having a boring list of tracked files is not really useful. A good version control system does not just let you keep track of file changes, it allows you to branch into different versions of files, allowing you to focus on different modifications in each branch.



When you have edited a file in branch1, the changes are not reflected in master or branch2. Branches are isolated, changes made at any branch will not affected files on other branches, just as the file change is only recorded in branch1.

New command - branch

branch is used to, as the name suggest, create a new branch. Branches has the exact same history and files at first, but as changes are made to the files, commit are only append to the branch you are working on. Running branch without any branch name list all the branches.

branch any-name-for-a-branch
branch “any-name-for-a-branch” created.

branch
master
*any-name-for-a-branch

New command - checkout

checkout is use to switch between branches. When you switch to a particular branch, you need to ensure all the files are up to date with the latest commit, else all the changes made will be lost. log command will only output the history of the branch you are working on.

checkout any-name-for-a-branch
There are modified files! Please commit or remove the changes before switching to another branch.
checkout any-name-for-a-branch
checkout to branch “any-name-for-a-branch”

New command - tree

tree command is used to output a pretty tree of the entire history.

tree

```
* 1: initial commit
* 2: edited file.txt
.
.
* 18: added just_added.txt and edited i_was_modified.txt
|\
| * 19: first commit in branch any-name-for-a-branch
* | 20: a commit in the original branch (aka master)
* | 21: another commit in the branch master
| * 22: second commit in branch any-name-for-a-branch
| |\
| | * 23: I love chaos so I created a third branch
```

Topic 3: Brain Game

Problem Statement

You have a friend called Sheldon who is a neurobiologist interested in how human brain works. Recently, Sheldon had a breakthrough in his research and proposed that human brains consist of numerous neurons connected to each other through something he called ‘synapse’. He came up with the idea that each thought in our brain are similar to messages passing through each neuron until it reaches its destination. You as a great friend decided to help him by creating a simulation program to visualize his findings.
Your simulation is very simple. Imagine each neuron as a messenger. Message will start at a selected neuron and has to reach its destination neuron through several interconnected neurons. However, Sheldon mentioned that each connection (synapse) has a different length, and time needed to pass message. You are required to calculate the distance travelled and time needed for each message.

Background information

Neuron Network Simulation Program
The network will definitely have more than 2 neurons. There is a chance that one neuron will not be connected to another neuron. This means that some message task cannot be completed.

Neuron
Each neuron serves as a node in the network. It can have zero or more connections with other neurons. However, two neurons can only have one and only one connection.

Synapse/Connection
Each synapse is similar to edges connecting two neurons. The connection has two properties, one being the length of synapse, the other is time needed for a message to travel through it.

Sample Input

Sheldon will provide you with inputs containing information about how he wants his neuron network to be and also a list of messages to be pass through the network.

The first line of the first input will be an integer n, the number of neurons that is in the network. The following lines describe n instances of neuron, each separated by a blank line. In each instance, the first line contains two integers, a and m, with a representing the ID of the neuron, while m is the number of edges connecting to the neuron. The next m lines will contain 3 numbers, A, d, and t. A is the ID of the other neuron which is connected to the current neuron, d is the distance of the edge while t is time needed for a message to pass through it.

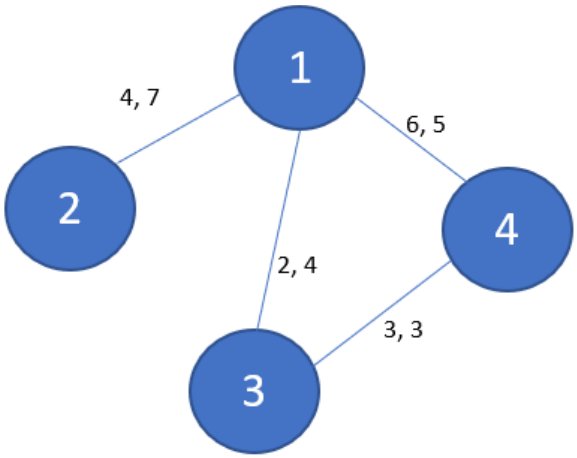
The second input contains information of the messages to be pass through. First line of the input will be an integer x, the number of messages to be pass through. The following x lines will contain 2 integers each, which represents the start point and end point of the message.

```
4
1 3
2 4 7
3 2 4
4 6 5

2 0

3 1
4 3 3

4 0

5
1 4
2 3
4 2
1 2
3 1
```

Note: *Feel free to modify the input format as you see fit. You may change to console prompt but do not leave out any information.*

Sample Output

The output should consist of x lines. If the message can reach its destination, print out both the distance travelled and time needed for the message, separated by a whitespace. Otherwise, print 'No path available'. The sequence of each line should follow the sequence in the input.




```
6 5
6 11
10 12
4 7
2 4
```

Explanation: - Most of the message are straight forward to pass through (only requires one synapse). Only the second and third message have to pass through multiple synapses. The distance travelled and the time needed is the summation of each synapse.

Assumption you can make

1. Each synapse is bi-directional, meaning that each synapse can pass message forward and backward.
2. The number of neurons will always be less than 20.
3. The message can pass through any neuron as long as it finally reaches its destination.
4. There will only be at most one synapse between any two neurons.

Some crazy idea

1. GUI
   - Visualize the network. Your program may show the whole network with all the interconnected neurons. Your program may also animate how the message is passed through from start to destination.

2. Unidirectional Synapse
   - Each synapse can only pass message one way. This now means that there can be at most two synapses between any two neurons, each representing forward or backward (depending on how you see it).
   - For example, the third line in the sample output '2 4 7' now means that the synapse can pass message from neuron **1** to neuron **2** only. This means that certain message that could be passed originally cannot be completed now.

3. Optimization
   - Sheldon once said that the neuron will magically pick a synapse that always result in the shortest distance and/or time. Now, for each message, find out the path that follows the rule mentioned.
   - Prioritize the shortest time first. If multiple paths have the same amount of time, choose the one with the shortest distance. If there are multiple path with same amount of time and distance, then you may choose any one of it.

4. Synapse Lifetime
   - Each synapse can only pass message for a limited amount of times only before it dies and disconnect. The input of each synapse will now contain an extra integer, *i*, which range between 1 and 10.
   - Once the synapse dies, it cannot pass any message between the neurons anymore.

5. ??
   - Any idea that comes from your creative mind that might spark Sheldon's interest.

Topic 4: DreamCorporation

Introduction

Figure 1: The graphical illustrations of a multi-level marketing scheme

DreamCorporation (With slogan "**Do You Have a Dream?**") is a newly found startup that is based on multi-level marketing as the organisation's primary source of income. DreamCorporation has nothing to sell except its exclusive membership which it has spent a lot of money marketing to target people who would like to earn money without doing much work.

Multi-level marketing, also called pyramid selling is a marketing strategy for the sale of products or services where the revenue of the MLM company is derived from a non-salaried workforce selling company's products/services, while the earnings of the participants are derived from a pyramid-shaped or binary compensation commission system. (Sourced from Wikipedia: https://en.wikipedia.org/wiki/Multi-level_marketing)

Problem Statement

DreamCorporation has contacted University of Malaya to hire talented students like you to build the system. Depending on the capability of your system, the equivalent reward will be added to your Data Structure assignment marks.

DreamCorporation want to test you out to see if you're fit for the job, here's the skills you need:

1. Traversing in Tree using Reference - Make sure you understand parent-child relationship
2. Encryption and Decryption - To protect information leaking to interested parties
3. Primary school maths - Percentage, Addition and Multiplication... Seriously??

Project Requirements

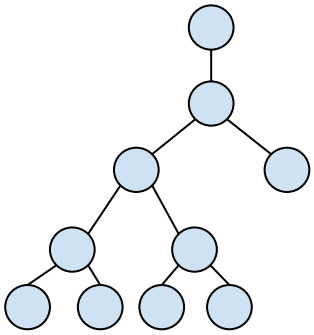


Figure 2: Implementation of Multi-level marketing scheme as a tree

Definition with examples (Referring to Figure 2)

- Direct Downline : C's direct downlines are E and F
- All Downlines : C's all downlines are E, F, G, H, I, J
- Direct Upline : C's direct upline is B
- All Uplines : C's all uplines are A, B

How does MLM work - by example?

1. User A is the root node owned by DreamCorporation, its commission will be added to the company revenue.
2. User J just paid RM 50 and signed up.
    a. User F will get RM 25 (50%)
    b. User C will get RM 6 (12%)
    c. User B will get RM 4.5 (9%)
    d. User A will get RM 3 (6%)
    e. Since root node (User A) is reached, whatever's that left RM 11.5 (3% + 20%) will be added to company revenue.

Encryption & Decryption

1. Every user will need to have an encrypted_name to protect their identity.
2. Only the company has the KEY to decrypt the encrypted_name to real_name.
    a. encrypt(NAME, KEY) → ENCRYPTED_NAME
    b. decrypt(ENCRYPTED_NAME, KEY) → NAME
3. When signing up the new user, you must only save the encrypted name.
4. Every time you want to see the user original NAME, you should type in the KEY.

Expected Output

1. Show that when adding a new user, RM50 registration fee be added into the new user's uplines.

2. Show that your encryption and decryption works as expected.

3. Print the tree of users
4. Requirements as identified in the table 1 below:

No.	Requirements by Role

1	**As a USER,**
	- I must have ID, ENCRYPTED_NAME, REVENUE
	- I must pay RM50 buy a member card to join DreamCorporation
	- I can only have one direct upline but any number of direct downlines
	- I can receive payments from 5 generation of my downlines when they sign up and pay RM50, with the following commision. The commission are added to my REVENUE.
	  - 1st gen: 50%
	  - 2nd gen: 12%
	  - 3rd gen: 9%
	  - 4th gen: 6%
	  - 5th gen: 3%

2	**As a COMPANY_ADMIN,** - I have COMPANY_REVENUE - I can CRUD (Create/Retrieve/Update/Delete) users - I can encrypt and decrypt - I can save and load all users and data in a file - I can see the entire tree of users - I collect all the remaining profit after commissions are paid (in other words, i will earn at least 20% of every membership signup) - I can see the company's revenue - I can see the company's revenue of each generation - I can check every user's revenue - I can change the registration fee and commission from time to time to motivate people to ask more people to sign up for membership

Table 1: Requirements by Role

Some crazy idea

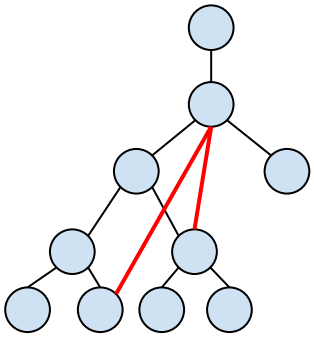


Figure 3: Implementation of "Why Only One Direct Upline?"
Allows relationships like the red line.

1. WHY ONLY ONE DIRECT UPLINE? [A+ guaranteed for this feature]
- All the features above + the system now allow user to have more than 1 direct upline - Direct uplines will share the commission (commission / number_of_direct_upline)
2. GRAPHICS
- GUI Interface
- Animation - When a user signed up, do an animation on the RM50 as it splits into parts and is given to respective uplines
3. DO IT YOUR WAY
- Or any modification that makes your system more user-friendly and efficient