Starfleet - Day 03

Trees and Graphs

Staff 42 pedago@42.fr

Summary: This document is the day03's subject for the Starfleet Piscine.

Contents

Chapter I

General rules

- Every instructions goes here regarding your piscine
- Turn-in directories are `ex00/`, `ex01/`, . . . , `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- The exercises must be done in order. The evaluation will stop at the first failed exercise. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The subject can be modified up to 4 hours before the final turn-in time.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.
- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter II

Day-specific rules

- If asked, you must turn-in a file named `bigo` describing the time and space complexity of your algorithm as below. You can add to it any additional explanations that you will find useful. When your algorithm involves going through a tree or a graph, specify if it is breadth or depth first search, iterative or recursive.

```
$> cat bigo
O(n) time
O(1) space
iterative BFS
$>
```

- Your work must be written in `C`. You are allowed to use all functions from standard libraries.

- For each exercise, you must provide a file named main.c with all the tests required to attest that your functions are working as expected.

Chapter III

Exercise 00: Toy factory

	Exercise 00
Exercise 00: Toy factory	
Turn-in directory : *ex*00/	
Files to turn in : `info.c header.h main.c`	
Allowed functions : `all`	
Notes : `n/a`	

Thanks to your `Poker` addiction, you find yourself in the middle of nowhere working in a strange toy factory for a few days.

One day, you notice a toy which is a `plastic tree` for children. Apparently, it's a normal toy, but when you get closer, you notice some `numbers on the branches`.

You know what it is and you shout it at loud: `It's a binary tree!`

At the same time, the boss of the factory notices you during this `euphoric` moment and immediatly comes to you, not for reprimand, but to thank you. You were the first one to notice the computer science elements in his toys, it's in fact his strange little `fantasy`.

Now he ask you to go further: Is it a `binary search tree`? What is its `depth`? Is it `balanced`? You answer correctly. Then, with a `smile`, he brings `100 different tree toys`...

You know that the creation of a `little software` is necessary to answer all his questions!

The boss is asking you the following informations about a tree :

- The `minimum` value in the tree.
- The `maximum` value in the tree.
- The `number of nodes` in the tree.

- The `depth` of the tree, the number of depth level a tree can have.
- Is it a `Binary Search Tree` (BST)?
- Is it `balanced` ?

Given the following structure:

```c
struct s_info {
        int min;        // the minimum value in the tree
        int max;        // the max value in the tree
        int elements;   // the number of nodes in the tree
        int depth;      // the depth of the tree
        int isBST;      // 0 = TRUE, 1 = FALSE
        int isBalanced; // 0 = TRUE, 1 = FALSE
};
```

Create a function that returns the informations related to a tree, with his root node passed as parameter:

```c
struct s_info getInfo(struct s_node *root);
```

`If you're beginning with the tree, we advise you to do one function per information.`

`A binary search tree (BST) is a binary tree in which every node fits a specific ordering property : all left descendants <= n < all right descendants. This must be true for each node n.`

Chapter IV

Exercise 01: Tree of life

	Exercise 01
Exercise 01: Tree of life	
Turn-in directory : *ex*01/	
Files to turn in : `findParent.c main.c bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Back to the `civilization`, you decide to take a trip to `Sequoia Park`, on the road to the `Giant Sequoia`... All of a sudden, you find an `old deserted house`. You decide to go inside and then, you discover an `old dusty computer`...

Intrigued, you turn it on and discover something amazing! It contains the `life work of a searcher`. One of his work is a `family tree` of all the `living species` since the creation of the earth ! An idea comes to your mind: what is the common ancestor of the `man` and the `dinausor`?

The tree of life is an `n-ary tree` using the following structure :

```c
struct s_node {
    char           *name;
    struct s_node **children;
};
```

Create a function wich returns the `first common ancestor` between two species, by taking as parameters the `root` node of a tree and 2 strings which are the names of `two species`:

```c
struct s_node *findParent(struct s_node *root, char *firstSpecies, char *secondSpecies);
```

Where children is a null-terminated array containing pointers to child nodes.

If one the the two species cannot be found in the tree, the function returns `NULL`.

Here are some valid examples with the 'tree of life' file :

```
findParent(root, "Dinosauria", "Homo sapiens"); // returns the "Amniota" node

findParent(root, "Lynx", "Marsupialia"); // returns the "Mammalia" node

findParent(root, "Dinosauria", "I do not exist !"); // returns NULL
```

Chapter V

Exercise 02: Planet of the Apes

	Exercise 02
Exercise 02: Planet of the Apes	
Turn-in directory : *ex*02/	
Files to turn in : `saveTheSequoia.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Once you reach the giant Sequoia, the stupor!

`Lots of monkeys escaped from a zoo nearby`! These monkeys are now `hiding in the Giant Sequoia`. It could be ok, but currently the `bigger` monkeys are on the `lower branches` (closer to the ground) and the most `lightweighted` monkeys are at the `top of the tree`. The `firemen` are at the `ball`, the Sequoia will `break off` soon... What a `calamity`!

You have to save the day, you pull out your emergency drone, your drone can make 2 monkeys `exchange` their `places` (by frightening them). It's now up to you to reorganize the tree to prevent it from `breaking off`.

Each node contains the `weight` of the `monkey` who is on it, you must create a function able to switch a `min-heap` to a `max-heap`.

Given the following structure:

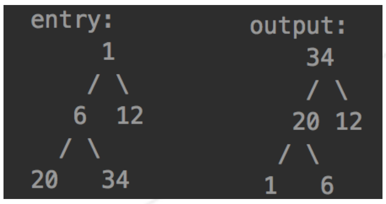
```
//binary tree node
struct s_node {
    int           value;
    struct s_node *right;
    struct s_node *left;
};
```

Given a pointer to the root node of a `min-heap`, implement a function that transform the `min-heap` into a `max-heap` :

```c
void saveTheSequoia(struct s_node **root);
```

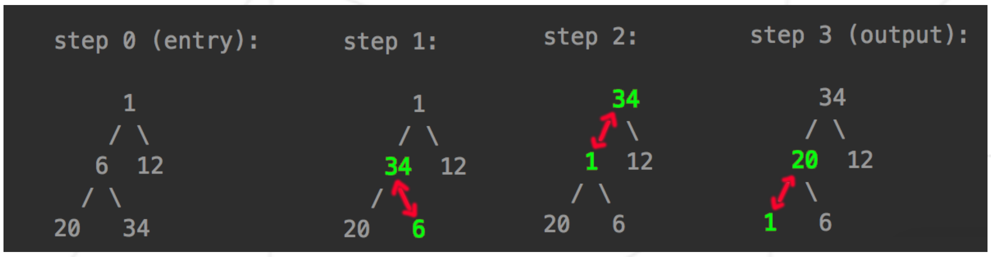
`Be careful`, you can't recreate a new tree, you must only `swap` the monkeys.

Example:



```
entry:                 output:
         1                  34
        / \                 / \
       6   12              20  12
      / \                 / \
    20   34              1   6
```

Same example showing the steps:



```
step 0 (entry):      step 1:        step 2:        step 3 (output):

       1                1             34              34
      / \              / \              \             / \
     6   12          34   12          1   12        20   12
    / \              /                / \               \
  20   34          20    6          20   6          1    6
```

By swapping the monkeys we mean swapping the nodes, not just the weights!

Chapter VI

Exercise 03: Planet of the Apes 2

	Exercise 03
Exercise 03: Planet of the Apes 2	
Turn-in directory : *ex*03/	
Files to turn in : `insertMonkey.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Oh no, a new `wave` of monkeys is `approaching`, use your `drone` to make sure that the `tree stays in max-heap` with the `newcomers`!

Given the following structure:

```
//binary tree node
struct s_node {
    int            value;
    struct s_node *right;
    struct s_node *left;
};
```

Create a function that inserts a new ape in the tree, given as parameters the root node of a tree, and the node of the new incoming monkey :

```
void insertMonkey(struct s_node *root, struct s_node *monkey);
```

The tree must `stay a max-heap` !

`Be careful, having a max-heap means also having a complete tree !`

Chapter VII

Exercise 04: The jo?e of the y?a? !

	Exercise 04
Exercise 04: The jo?e of the y?a? !	
Turn-in directory : *ex*04/	
Files to turn in : `understand.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

After your return from vacation, you receive a letter from one of your friends, he says he needs you immediately.

Here you go again for an adventure in ... `Paris`!

In the `plane`, you are so much bored. The power cut that `turned off all the tablets` to watch movies does not help! You have no choice but to `talk to the person next to you`. However `this one has difficulty articulating`, some words are not `understandable`.

Argh! He just told you a `joke` that seemed very funny ...

You decide to create a `program to understand the joke`, which is the perfect time to create a `Prefix Tree` (also called `Trie`).

A `Prefix Tree` is a n-ary tree where each node stores :

- A character '`c`'.
- If the sequence of characters from the root to the node is a valid word, in a variable '`isWord`'.
- A null-terminated array of pointers to `child` nodes.

Here are the structures for the `Prefix Tree`:

```
struct s_node {
    char          c;
    unsigned int  isWord:1; // 0 = FALSE, 1 = TRUE
    struct s_node **child;
};

struct s_trie {
    struct s_node *node;     // root of the trie
};
```

Implement 2 functions:

- `createTrie(dictionary)` creates a `Prefix Tree` with the dictionary of english words passed as parameter (null-terminated array).
- `understand(word, trie)` returns the correct word from the `Prefix Tree` by substituting the characters `'?'` by other characters. If no match is found the function returns the incomplete word. If multiple matches are possible, the function returns the first one found.

These functions must be declared as follows :

```
struct s_trie *createTrie(char **dictionary);

char *understand(char *word, struct s_trie *trie);
```

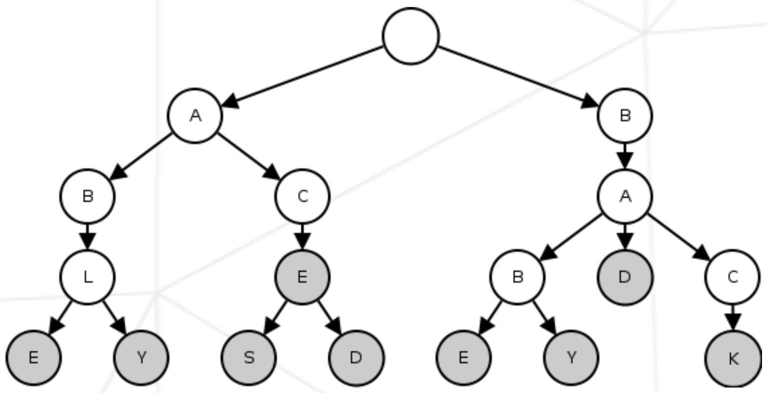
The `word` passed as parameter will always be `lowercased` (characters from 'a' to 'z' and of course '?').

Examples :

For the following dictionary :

```
char *dictionary[] = {"able", "ably", "ace", "aces", "aced", "babe", "baby", "bad", "back", NULL};
```

The `Prefix Tree` created by the function `createTrie` would be :



```
$> compile understand.c
$> ./understand 'what is the differ???? betw??? a figh??? pilo? and god'
what is the difference between a fighter pilot and god
$> ./understand 'i am goi?? to paris to eat some che?s?, yum?? !'
i am going to paris to eat some cheese, yummy !
$>
```

Chapter VIII

Exercise 05: Junk Food Search

	Exercise 05
Exercise 05: Junk Food Search	
Turn-in directory : *ex*05/	
Files to turn in : `junkFood.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Once you arrived in `Paris`, you meet with your friend. He looks troubled ... He explains his problem :

Three days ago he was walking quietly in the `Tuileries Garden`, taking advantage of the Parisian life. The weather was good, perfect to eat one of his favorite `cereal bars`! When all of a sudden, while he was enjoying his `cereal bar`, a thief pops up and steals it from him.

Absolutely furious, our friend is about to set up a giant espionage network in all the city and counts on your help to find the culprit ...

You first tell yourself that it is not the end of the world and that instead of having a `"Big Brother"` in Paris it might be wiser to buy a new cereal bar, offer it to your friend and get this over with !

You have an undirected graph that contains the squares of Paris (or Places), some squares contain sellers of cereal bars.

Your goal is to find the closest sellers.

The undirected graph uses the following structures :

```
struct s_node {
  char          *name;
  int           hasCerealBar; //0 = FALSE, 1 = TRUE
  int           visited;      //0 = FALSE, 1 = TRUE
  struct s_node **connectedPlaces;
};

struct s_graph {
  struct s_node **places; //places of Paris
};
```

Where each node represent a Place, and connectedPlaces is a null-terminated array containing pointers to the places it is connected to.

Given the following structure:

```
struct s_sellers {
    int           distance;
    char          **places;
};
```

Implement a function to find the closest squares with a seller of cereal bars from where you are (Place de la Concorde) :

```
struct s_sellers *closestSellers(struct s_graph *parisPlaces, char *youAreHere);
```

The function will return a pointer to a structure with :

- A null-terminated array containing the names of the closest squares (of equal distance from where you are).
- The distance from where you are to the closest squares with a cereal bar seller. The distance between 2 connected squares is 1.

If your current square or a seller could not be found in the graph, the function returns NULL.

Examples, using the file 'main.c', and searching the closest sellers for a place given in the argument:

```
$> compile junkFood.c
$> ./junkFood "Place de la Concorde"
At a distance of 1, these places have cereal bars:
Place Maurice-Chevalier
Place Joachim-du-Bellay
$> ./junkFood "Place El Salvador"
At a distance of 3, these places have cereal bars:
```

```
Place Constantin-Pecqueur
Place Violet
$> ./junkFood "Place de la Fontaine-aux-Lions"
At a distance of 5, these places have cereal bars:
Place Violet
```

You can use the visited field to search through the graph. Don't forget to reset the field to 0 after.

For this exercise, we provide an implementation of a queue that you can use with the following functions :

- `queueInit()` : Initialize the queue. The first and last pointers are set to `NULL`.
- `enqueue(queue, node)` : Add a node to the end of the queue.
- `dequeue(queue)` : Remove the first item from the queue and return it.

And the following structures:

```c
struct s_queueItem {
  struct s_node *place;
  struct s_queueItem *next;
};

struct s_queue {
  struct s_queueItem *first;
  struct s_queueItem *last;
};
```

The functions defined above are declared as follows:

```c
struct s_queue *queueInit(void);

void enqueue(struct s_queue *queue, struct s_node *place);

struct s_node *dequeue(struct s_queue *queue);
```

Here is an example of code on how to use it:

```c
struct s_graph *graph = getSquares("squares.txt"); // get some Paris places

struct s_queue *queue = queueInit(); // queue content: NULL

enqueue(queue, graph->places[0]);     // queue content: 'Place de la Contrescarpe'
enqueue(queue, graph->places[1]);     // queue content:
                                      // 'Place de la Contrescarpe' => 'Place de la Porte-Molitor'

dequeue(queue);                       //queue content: 'Place de la Porte-Molitor'
```

Chapter IX

Exercise 06: Big Brother

	Exercise 06
Exercise 06: Big Brother	
Turn-in directory : *ex*06/	
Files to turn in : `bigBrother.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Now that you have finally bought the `cereal bar`, you gladly offer it to your friend, however he refuses violently.

He then tells you that the stolen cereal bar was very special to him. Because he ate it in one of those memorable moments when all his problems were lifted, like flying in the sky, letting him appreciate life and its beauty.

The fact that the cereal bar was stolen at this specific moment is as if this precious moment was stolen from him.

You don't know what to think, but one thing is for sure : more work for you !

Your friend has launch his great `espionage program` in Paris.

The program is `community based`. It uses a `calling system` which can be reach by any inhabitant willing to contribute (i.e. report a crime).

This system uses a limited `set of actions` organized in a `Binary Search Tree` (BST)! Each `action` of the set is a node of the `BST` with a designated `name` and `id` (the id is a unique positive integer).

Here is the structure of a node :

```
struct s_node {
    char            *actionName;
    int             actionId;

    struct s_node   *right;
    struct s_node   *left;
};
```

Example of BST :

```
    "Happy slapping"(11)
           /   \
          /     \
"Prank call"(4)  "Cereal bar robbery"(13)
        \                  \
    "Blackmail"(6)          ...
        /   \
      ...   ...
```

To find the culprit, the software created an `undirected graph of relationship` between the `inhabitants of Paris`. Each inhabitant has a `calling list`.

The algorithm of the program has retained just the most important information of each call and organizes them as follow :

- `Subject` : a pointer to the `person` who committed the action (according to the caller).
- `Action` : the name of the `crime` among the diverse set of actions.

One of the actions is `"Cereal bar robbery"`. These actions are stored are stored in the `Binary Search Tree`.

You have to browse the `graph` to output a list of `cereal bars thieves`.

You have to pay attention to `false gossip`, so if a person reporting a crime is too far away from the subject of the call, the `testimony` is considered `false` ! Two people are too far away if they are not connected to the third degree or less.

Example :

```
p1 <-> p2 <-> p3 <-> p4 <-> p5
```

p1 is too far away from p5 but not from p4

Given the following structures for the `graph of people`:

```c
struct s_call {
    int             action;
    struct s_person *subject;
};

struct s_person {
    char            *name;
    struct s_call   **calls;
    int             visited; //0 = FALSE, 1 = TRUE
    struct s_person **connectedPeople;
};

struct s_graph {
    struct s_person **people;
};
```

Create a function that takes as parameter the `graph of the people`, the root node of the `action BST`, and the `action name` that you have to look for.

This function has to returns a `null-terminated array` containing the names of the potential culprits:

```c
char **findCulprit(struct s_graph *graphPeople, struct s_node *actionRoot, int idAction);
```

Examples:

- `"Cereal bar robbery"` : `findCulprit(graphPeople, actionRoot, 612)` will return:

```
Angella Gaudry
Arianne Ber
Brittanie Ilic
Eugenio Oceane
```

- `"Pickpocketing"` : `findCulprit(graphPeople, actionRoot, 1586)` will return:

```
Ambrose Julliard
Annabel Perrault
Branden Erdmann
Gwyneth Vernhes
Larue Wirtz
Sidney Lesueur
```

For this exercise, we also provide an implementation of a queue, which is same as the previous exercise, except that now it doesn't store a `s_node` structure but a `s_person` structure.

Chapter X

Exercise 07: Bubble path finder

	Exercise 07
Exercise 07: Bubble path finder	
Turn-in directory : *ex*07/	
Files to turn in : `maxTraffic.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Now that you have found the `thief`, you can finally enjoy your well deserved vacation.

Unfortunately, your friend has another request before you leave France. A huge `outdoor reggea concert` will be organized in Paris, he sees an opportunity to make some money.

It is well know, during `reggea concert`, reggea fan love to ... chew `bubble gum` (of course !). He wants to set a little bubble gum shop (more of a stand in fact) near the event. In order to sell a maximum number of bubble gums, he wants to know which street it should set it on.

Your mission will be to create a function which will return the `maximum traffic` on one street, depending on the location of the concert.

You have an `undirected graph` which represents the city. Each node is a `square` of this city, each node has an associated `population` (people living in this neighborhood). Each `edge` (arc between two nodes) is considered as a street.

The following structures are used for the graph of `Paris places`:

```c
struct s_node {
    char           *name;
    int            population;
    int            visited; //0 = FALSE, 1 = TRUE
    struct s_node **connectedPlaces;
};

struct s_graph {
    struct s_node **places;
};
```

Implement a function that finds the street with the most important traffic and returns that traffic, given as parameters :

- The undirected `graph` of places of Paris
- The name of the `place` where the concert will take place

This function will be declared as follows :

```c
float maxTraffic(struct s_graph *parisPlaces, char *eventHere);
```

`Note`: If the `place` passed as parameter could not be found in the `graph`, the function returns -1.

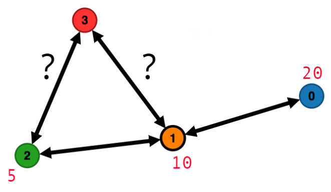
For this exercise you can assume that:

- All the inhabitants of Paris will attend the concert.
- They will take the shortest path.
- If there is more than one shortest path, the traffic will be evenly distributed between the different shortest paths.

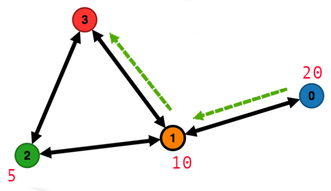
For this exercise, we also provide an implementation of a queue in the file '`main.c`'.

`Example 1 :`

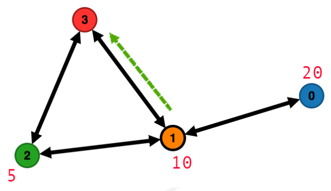
You know that the concert will be organized at the square `3`, this square has `2 streets` around it. You want to know which one of these streets will have the most traffic.



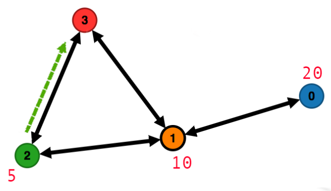
You first find the shortest path from `node '0'` to `'3'`, which is going through `node '1'`, then `node '3'`:



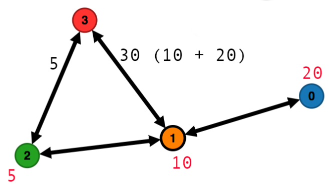
Then you find the shortest path from `node '1'` to `'3'`:



And then for `node '2'`:



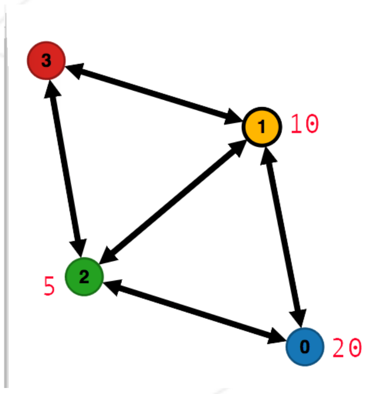
Now, you know which street the population of each node will use:



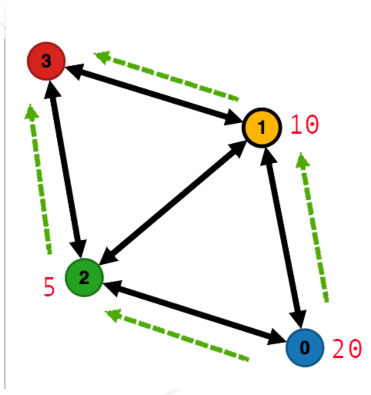
In this case, the function returns `30`.

`Example 2 :`

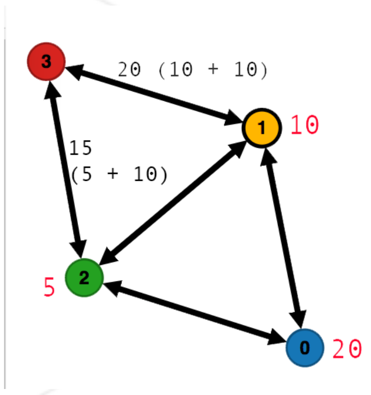
The concert will be organized at node `'3'`:



This time, node `'0'` has `2 equal paths` to go to the node `'3'`.



So, its population will be evenly distributed (half go to the first path, the second half go to the second path):



In this case, the function return `20`.

Expected output:

```
$> compile maxTraffic.c
$> ./maxTraffic
Place du Louvre : 497301.2
Place Pigalle : 449797.3
Place des Invalides : 512979.1
I do not exist : -1.0
$>
```