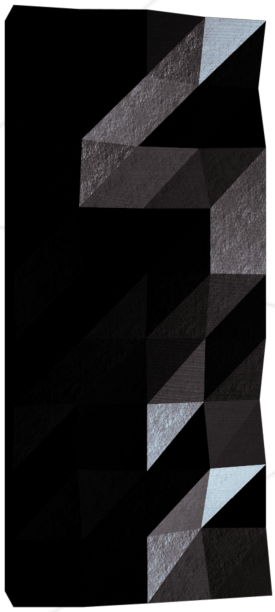


Starfleet - Day 01

Lists, Stacks and Queues

Staff 42 pedago@42.fr

Summary: This document is the day01'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.

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

- 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: We wants it

	Exercise 00
Exercise 00: We wants it	
Turn-in directory : $ex00/$	
Files to turn in : `precious.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

One day you find yourself in a `strange machine museum`. The visit was pretty cool. As you are heading towards the exit, all of a sudden, you notice a strange machine. The size of a cat, the shape of a ring...

This machine is beautiful, shiny like silver, bewitching... We wants it. We needs It. We must have precious...

Mesmerized, you tell yourself, that you have to recreate the machine on your own computer, to keep it with you always.

Here are the notes from the creator of the machine, Blaise Pascal, his notes are in french from the 16th century:

" Cette machine est une sorte de roue avec une soixantaine de petits points autour. Chaque point dispose d'une lettre de l'alphabet (minuscule, majuscule, ou charactère spécial). La machine prend en paramètre une phrase et va appliquer des rotations à chaque charactère, puis va, feuille de papier présente, écrire la distance entre les 2 charactères.

Ma foi, je l'ai créé pour pouvoir crypter mes messages afin de pouvoir envoyer plus sereinement mes messages. "

Heu.. OK... given the following define:

```
# define CS "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890 !?:;,.$@<>()+*-_="
```

Implement a function that reverse engineers a text based on the description of the machine above.

`You must use a circular doubly linked list.`

The function will take an array of `integers` and the `size` of this array as parameters.

```
char *precious(int *text, int size);
```

Examples :

```
$> compile precious.c
$> ./precious -32 36 -22 40 -22 13 6 -1 -36 26 11 -31
We wants it.
$> ./precious -32 36 -22 31 -9 0 -1 15 -36 -28 -15 -31
We needs It.
$> ./precious -32 36 -22 30 8 -2 1 -37 25 -7 21 -17 -22 37 -12 -3 -22 33 2 -13 -2 6 6 6 -2 -30 0 0
We must have the precious...
$>
```

Chapter IV

Exercise 01: The Mouth of Truth

	Exercise 01
Exercise 01: The Mouth of Truth	
Turn-in directory : $ex01/$	
Files to turn in : `reverse.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

While at the `strange machine museum`, the curator notices your work on Pascal's machine.

He is pretty amazed and asks for your assistance on another matter.

He shows you another wonder owned by the museum : an ancient lie detector called `The Mouth of Truth`.

Back in the time, it was used to reveal liars and trickery. Sadly, it has suffered from several damages through the times.

Some engineers worked on it for a while, and were able to put it back in order. However, they made a mistake :

The machine can only work if the words in the tested sentence are in reverse order (strange isn't it?).

Given the following structure which represents a node of a linked list:

```
struct s_node {
        char          *word;
        struct s_node *next;
};
```

Using `recursion`, implement a function that prints all the elements of a linked list in reverse order, with a space (' ') between 2 words :

```
void printReverse(struct s_node *lst);
```

Examples:

```
$> compile reverse.c
$> ./reverse inspectors! nothing did I
I did nothing inspectors!
$> ./reverse elegant more much so is Recursion
Recursion is so much more elegant
$>
```

Chapter V

Exercise 02: The Mouth of Truth 2

	Exercise 02
Exercise 02: The Mouth of Truth 2	
Turn-in directory : $ex02/$	
Files to turn in : `reverseV2.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Now the machine works, that's cool.

But unfortunately, they plan to use your function on an old machine, which doesn't have enough memory to support deep recursive calls.

You have to `recreate` the function `printReverse` but `iteratively` this time.

To achieve this, you must implement a `stack` with the following methods :

- `stackInit()` : Initialize the stack.

The top pointer is set to `NULL`.

- `pop(stack)` : Remove the top item from the stack and return it.

If the stack is empty, the function returns `NULL`.

- `push(stack, word)` : Add a word to the top of the stack.

given the following structure:

```
struct s_item {
  char          *word;
  struct s_item *next;
};

struct s_stack {
        struct s_item *item;
};
```

The functions defined above is declared as follows:

```
struct s_stack *stackInit(void);

void *pop(struct s_stack *stack);

void push(struct s_stack *stack, char *word);
```

The function `printReverseV2` has the same declaration as the previous exercise:

```
void printReverseV2(struct s_node *lst);
```

Examples:

```
$> compile reverseV2.c
$> ./reverseV2 inspectors! nothing did I
I did nothing inspectors!
$> ./reverseV2 powerful. very is Iterative
Iterative is very powerful.
```

Chapter VI

Exercise 03: Houston

	Exercise 03
Exercise 03: Houston	
Turn-in directory : $ex03/$	
Files to turn in : `console.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

One day, a man comes to you and invites you to go to a distant country:

`Syldavia`.

This country is, so to speak, ahead of its time concerning space conquest. Their objective would be to launch into orbit a fully automated vessel equipped with passengers. That's how you end up working for the `Space Cheap` Company.

In order to communicate with the exterior, the team in charge of the project asks for your help.

They want you to create a small program for the passengers allowing them to send messages during the travel.

Here are the specifications of the company:

- The program waits for an input, you can use the standard function `scanf` (or getline):

```
scanf("%s\n", &input); //will read an iput
```

- The program must have a prompt (in our example we choose `"?:"`, but `it's up to you`).

- The length of the message won't be over `255`.
- You have 2 specific commands to implement: `"UNDO"` and `"SEND"`. The `"UNDO"` command will delete the last chunk the user has added. The `"SEND"` command will send the message by returning the string added.
- You have to implement a `stack` (defined below), but for this time, the `"pop"` and `"push"` function must be in `O(1) time and O(1) space`.

`Example`:

```
$> compile console
$> ./console
?: Houston,
Houston,

?: we have a problem.
Houston, we have a problem.

?: UNDO
Houston,

?: we have lift off.
Houston, we have lift off.

?: SEND
this message was sent: Houston, we have lift off.
$>
```

The stack implementation is almost the same as the previous exercises, but the `'item'` structure look like this:

```
struct s_item {
    int           idx;
    struct s_item *next;
};
```

With `idx` the position where the last chunk was added.

Implement a function which waits for a user input and then returns the string that the user has written:

```
char *console(void);
```

Chapter VII

Exercise 04: Lost messages

	Exercise 04
Exercise 04: Lost messages	
Turn-in directory : $ex04/$	
Files to turn in : `queue.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

The team has some concerns about the messages sent to the ground :

Sending a message takes several seconds and if during the sending another person sends a new message, the return will be forgotten for the first message.

The team asks you to kindly implement a waiting system that will put the messages on hold.

To create this waiting system, you have to `implement a new data structure` called `'queue'`.

Implement 5 functions for the following `queue` operations :

- `queueInit()` : Initialize the queue.

The first and last pointers are set to `NULL`.

- `enqueue(queue, message)` : Add an message to the end of the queue.
- `dequeue(queue)` : Remove the first item from the queue and return it.

If the queue is empty, the function returns `NULL`.

- `peek(queue)` : Return the first item of the queue.

If the queue is empty, the function returns `NULL`.

- `isEmpty(queue)` : Return 1 if the queue is empty, 0 otherwise.

Given the following structure:

```
struct s_item {
    char          *message;
    struct s_item *next;
};

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

The functions defined above must be declared as follows:

```
struct s_queue *queueInit(void);

char *dequeue(struct s_queue *queue);

void enqueue(struct s_queue *queue, char *message);

char *peek(struct s_queue *queue);

int isEmpty(struct s_queue *queue);
```

All of these operations have to perform in O(1) time and O(1) space.

Chapter VIII

Exercise 05: Tank stack

	Exercise 05
Exercise 05: Tank stack	
Turn-in directory : $ex05/$	
Files to turn in : `tank.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

The ship has a rather special energy system:

The vessel has a first reservoir, this reservoir is filled using energy bars which have a variable power of 10 to 100.

The reservoir has a maximum power of 1000.

When a new bar needs to be inserted, if the power of the bar plus the total capacity of the tank is above 1000: the bar will be inserted into a new reservoir (nitro).

For testing purposes, the team would like you to simulate this tank on a computer.

Example of how it works :

```
 normal
 ------
 |990 |
 |    |

push(100):

 normal   nitro1
 ------   ------
 |990 |   |100 |
 |    |   |    |

push(10):

 normal   nitro1
 ------   ------
 |990 |   |110 |
 |    |   |    |

pop():

 normal   nitro1
 ------   ------
 |990 |   |100 |
 |    |   |    |
```

Given the following structures:

```
struct s_item {
    int            energy;
    struct s_item  *next;
};

struct s_stack {
    struct s_item  *item;
    int            total;
};

struct s_tank {
    struct s_stack **stacks;
    int            n;
};
```

You must implement a `tank stack`:

```
struct s_tank *initTank(void);

void tankPush(struct s_tank *tank, int energy);

int tankPop(struct s_tank *tank);
```

Implement a `stack` consisting of `n stacks`. (a 'stable' stack and then 'nitro' stacks).

Each element has a power of 10 to 100. When a stack has a power of 1000, you must switch to the next stack.

Chapter IX

Exercise 06: I visited all the cities

	Exercise 06
Exercise 06: I visited all the cities	
Turn-in directory : $ex06/$	
Files to turn in : `NthLastCity.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

After several months of work, the ship flies away and is out of the stratosphere.

It's a great success!

After your return from Syldavia, you relay your perilous trip on a Sunday afternoon at the table to the whole family, then after a while you say:

`-I have traveled the country so much` that I know every town and its inhabitants!

Your `nephew`, this little brat, intercepts you and asks:

```
-Ho really?  What is the 17th least populated city?

-Hummm ...
```

You have a `linked list` of which each `node represents a city` with its number of inhabitants.

This list is sorted from the cities that have the most inhabitants to the ones that count the least.

Given the following structure:

```
struct s_city {
    char *name;
    struct s_city *next;
};
```

Implement a function that returns the n-th elements from the end of a linked list.

`Your function must go through the list only one time!!`

```
char *NthLastCity(struct s_city *city, int n);
```

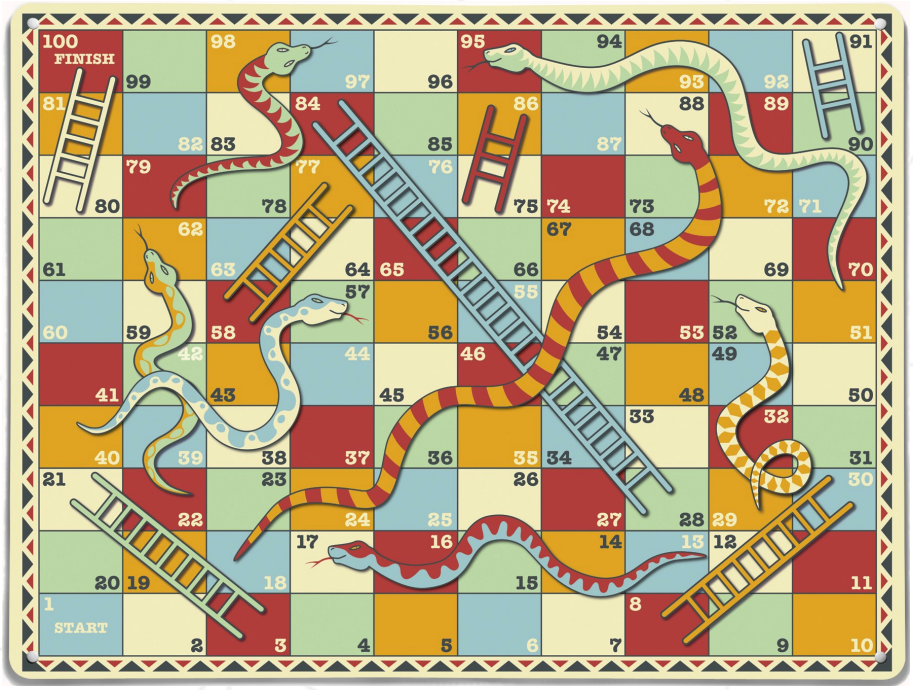
Chapter X

Exercise 07: Snakes and Ladders

	Exercise 07
Exercise 07: Snakes and Ladders	
Turn-in directory : $ex07/$	
Files to turn in : `copyGameBoard.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

In order to calm your adorable nephew, you decide to play a board game with him.

You take out your old board game: `Snakes and Ladders`.



You like this game, in fact, it reminds you your childhood!

You wonder now how to implement it on a computer...

hmmm... Ok maybe by using a linked list, where each node represents a cell of the board game.

Each node may have the famous `next` pointer which points to the next cell.

Also, in order to represent a snake or a ladder, we can add another pointer to our node which points to a random cell of the board, we can call it `'random'`!

You think that this game will look pretty good on your computer, but now you wonder: what if I want to `clone` the linked list representing the board?

You leave your little nephew alone, and turn on your computer.

Given the following structure:

```
//node of Linked List
struct s_node {
        int value; //the value of the cell
        struct s_node *random;
        struct s_node *next;
};
```

Implement a function that returns a copy of the board, with the linked list passed as a parameter.

```
struct s_node *cloneGameBoard(struct s_node *node);
```

Watch out! The random pointer can point to any other cell of the board, even its own cell!

Chapter XI

Exercise 08: The SML game

	Exercise 08
Exercise 08: The SML game	
Turn-in directory : *ex*08/	
Files to turn in : `minimumMoves.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

After a few hours, you have created the `Snakes and Ladders` game on your computer. You come back to see your nephew, and ask him to play your `Snakes And Ladders` game!

However, he laughs at you, your game is out of date.

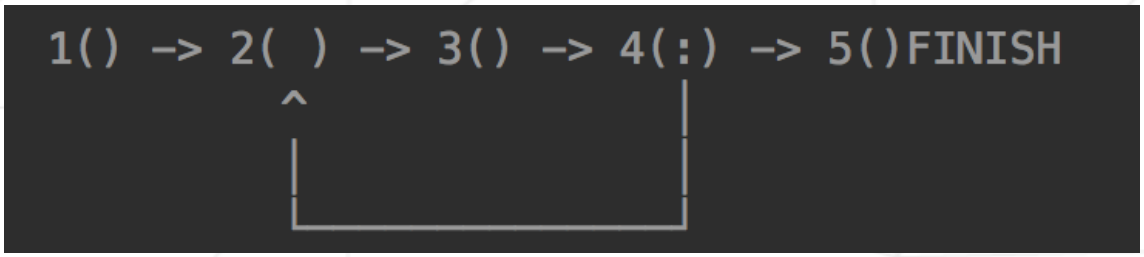
In fact, a new board game has just been released, it's called `"Snakes and Ladders and Mirages"`.

It's the same as `Snake and Ladders`, but it has a special feature: `Mirages`.

A `Mirage` is in fact another `board`, close to the current `Board`, which can be accessed by `snakes` or `ladders`.

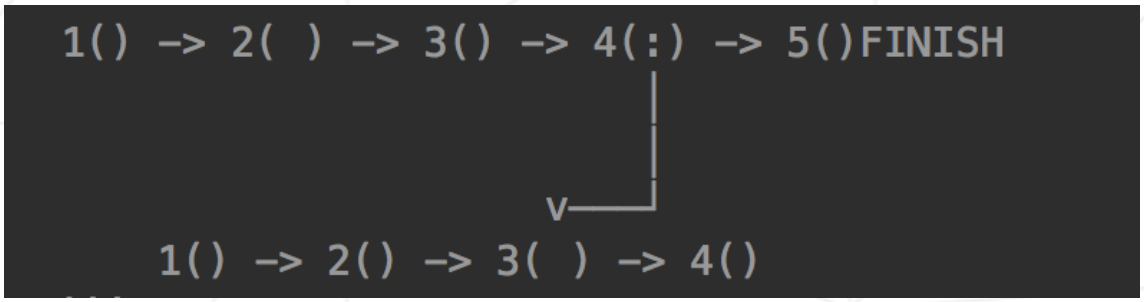
When you have a simple `Snakes and Ladders` game, the `'random'` pointer can bring you to any cell on the board.

Here is a simple example, a snake goes from node `'4'` to node `'2'`:



But with `SLM`, a snake can now go to a cell on one of the `Mirages`!

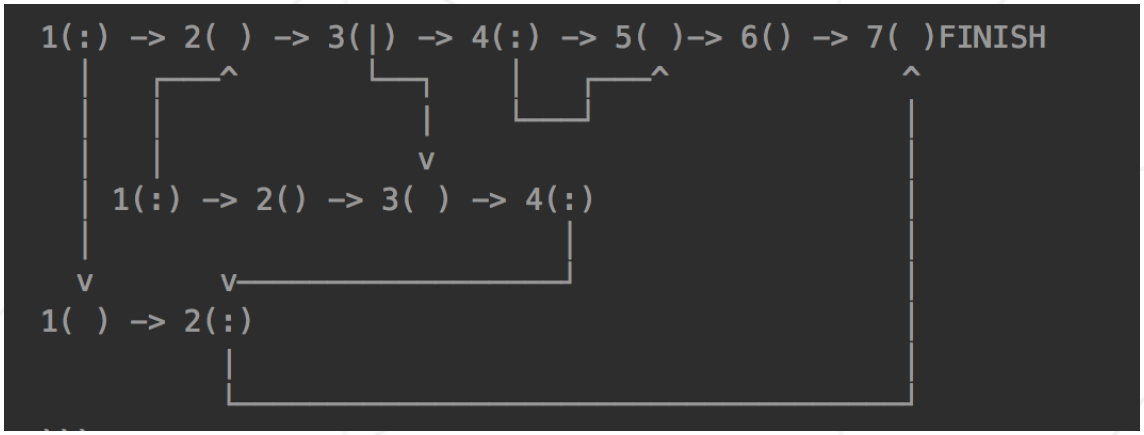
Here, a snake goes from node '4' of the current board to node '3' of a mirage:



On a `SLM` game, there are multiple `Mirages` and each cell of a `Mirage` can also have `Snakes` and `Ladders`!

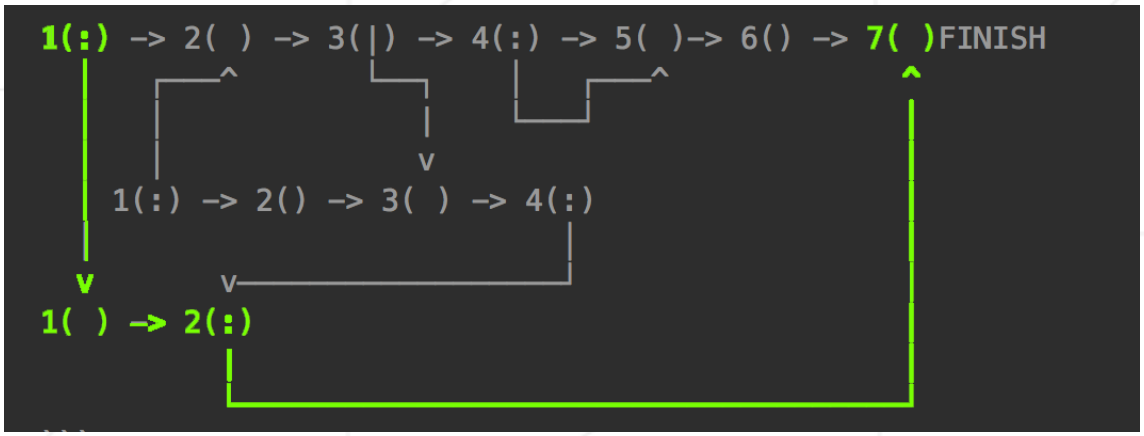
The goal of this game is to get as fast as possible to the end of the first board.

An example that represents a `Snakes and Ladders and Mirages` board:



On the example above, the shortest path, starting from the first cell of the main Board, will be:

```
1() -> Mirage 2 : cell 1() -> 2() -> Main Board : 7() FINISH.
```



You are convinced that an AI can defeat all humans at this game, so you decided to implement it!

Given the following structure:

```
struct s_node {
        int value; //the value of the cell
        int isFinal; //tell if this node is the last node of the first board, 0 = FALSE, 1 = TRUE
        struct s_node *random;
        struct s_node *next;
};
```

Given as parameter only the first cell of the main board, implement a function that returns the minimum number of moves to get to the to `finish cell`!

```
int minimumMoves(struct s_node *node);
```

We also provide a `parse()` function in the file `main.c`, which reads, parses and transforms a file into a `SLM Board game`!

The extension of the file for this game is `'.slm'` Here is a sample of a SLM file:

```
b
1() -> 2() -> 3() -> 4(b.1) -> 5(m1.4) -> 6(m2.1) -> 7() -> 8() -> 9()

m1
1() -> 2(m2.1) -> 3() -> 4(b.9)

m2
1() -> 2() -> 3()
```

With 'b' the main board, 'm1' as mirage 1, 'm2' as mirage 2 etc...

Inside the parenthesis '()', you can tell where the random pointer should point.

Example: if you have '(m1.2)', the random pointer will point to the `Mirage 1` on the node `2`.

`There can be a large number of mirages, so be careful!`