TP C

SET UP:

You can find the subject here:
https://github.com/MathieuMerienne/TP-C

```
TPC
|
|---alloc.c
|---alloc.h
|
|---basics
|       |---basics.c
|       |---basics.h
|
|---graphs
|       |---graph.c
|       |---graph.h
|
|---lists
|       |---list.c
|       |---list.h
|
|---Makefile
|---MyMain.c
|
|---queues
|       |---queue.c
|       |---queue.h
|
|---structs.h
|---sujet.md
|---sujet.pdf
```

Makefile:

There is a Makefile for this project. The command are simple

```
make test -> will compile all the code
make clean -> will clean all the compilation files and the executable.

make my_tests -> will compile the MyMain.c were you can write whatever you
want.

make my_clean -> will clean all the compilation files from my_tests.
```

Moulinette:

when you have compile the project you can use the moulinette.

```
./test part_you_want_to_test nb_tests //this is the pattern
```

```
by example:
./tests lists 10
./tests basiscs 10
./tests graphs 10
./tests queues 10
```

Basics:

```
unsigned long power_of_two(int n);
```

Behavior:
return 2^n

```
examples:
parameter -> return

0 -> 1
1 -> 2
3 -> 8
7 -> 128
```

```
unsigned long binary_to_int(int bin);
```

Behavior:
Take a "binary" number and return the conversion in int.

```
examples:
parameter -> return

1 -> 1
10 -> 2
101 -> 5
1111 -> 15
```

```
```

```
void decompose(int n, int tab[]);
```

Behavior:
Add all the elements of the the decomposition in prime numbers of n in array tab. We supposed that tab is long enough.

```
examples:
parameter -> tab

5 -> [5]
14 -> [2,7]
172 -> [2,2,43]
```

```
void reverse_arr(int* arr,size_t length);
```

Behavior:
Reverse the order of the elements in an array of int.

```
examples:
before -> after

[1,2,3,4] -> [4,3,2,1]
[6,1,8,4,9] -> [9,4,8,1,6]
[1] -> [1]
```

```
void n_shift_arr(int* arr,size_t length, int n);
```

Behavior: Shift all elements in the array arr of n places.

```
examples:
arr, n  -> arr

[1,2,3,4], 1 -> [4,1,2,3]
[1,2,3,4], -1 -> [2,3,4,1]
[8,4,5,9,1,7], 3 -> [9,1,7,8,4,5]
[8,4,5,9,1], -3 -> [9,1,8,4,5]
```

```c
void replace(char* s1, char* s2);
```

Behavior: Replace as much as possible the characters of s1 by those of s2.

```
examples:
s1, s2 -> s1

"co", "a" -> "ao"
"sa", "cou" -> "co"
"salut", "allo" -> "allot"
"", "a" -> ""
```

```c
void cesar(char* s, int dec);
```

Behavior: Implement the cesar cipher, it has to work with possitives and negatives shifts, it has to shift the letters and only the letters.

```
examples:
s, dec -> s

"abc", 1 -> "bcd"
"a2b", 2 -> "c2d"
"ea4(", -1 -> "dz4"
"coucou toi", 52 -> "coucou toi"
```

```c
unsigned long basic_calculator(char* s);
```

Behavior: Implement a simple caculator, it has to execute the mathematical expression from the left to the right without respecting the priorities of calculus.

```
examples:
s -> return

"2+2" -> 4
"2+2*2" -> 8
"2*2+2" -> 6
"3-1*2" -> 4
```

Lists:

Here is the struct:

```
typedef struct list list;
struct list {
  int val;
  list* next;
};
```

In this part you always have to make NULL list consideration, a NULL could be give as a parameter of a function and that could lead to segfaults if not handled.

```
void list_init(list* l);
```

Behavior:
Initialise a list pointer with -42 as value and a Null pointer for the next element.

```
example:
before -> after
val = xxxxxx -> val = -42
next = yyyyyy -> next = NULL
```

```
list* create_list(int nb);
```

Behavior:
Create an element of "type" list with nb as value and Null as next element. It has to be allocated !

```
int list_is_empty(list *l);
```

Behavior:
Return true (1) if the list is empty (if there is only the sentinel) else return false (0).

```
examples:
list -> return
```

```
NULL -> 1
-42: -> 1
-42: 10 -> 0
-42: 1: 84 ->0
```

```c
size_t list_len(list *l);
```

Behavior: Return the length of the list, the sentinel must be excluded.

```
examples:
list -> len

NULL -> -1
-42: -> 0
-42: 1 -> 1
-42: 8 :45: 96 -> 3
```

```c
list* list_get(list* l, size_t n);
```

Behavior: return the n-ieme element of the list l. If it does not exist return NULL.

```
examples:
list, n -> element

NULL, x -> NULL
-42: 1: 8: 42, 3 -> List(val = 42, next = NULL)
-42: 1: 8: 42, -1 -> NULL
-42: 1: 8: 42, 2 -> List(val = 8, next -> 42)
```

```c
void list_push_front(list *l, list *elm);
```

Behavior: Insert a position 0 of the list the element elm.

```
examples:
list before, elm -> list after

NULL, List(2,NULL) -> NULL
```

```
-42:, List(2,NULL) -> -42:2
-42: 48: 12: 15, List(2,NULL) -> -42:2:48:12:15
```

```
void list_push_end(list *l, list *elm);
```

Behavior: Insert at the end of the list the element elm.

```
examples:
list before, elm -> list after

NULL, List(2,NULL) -> NULL
-42:, List(2,NULL) -> -42: 2
-42: 1: 2 : 3: 4, List(5,NULL) -> -42: 1: 2: 3: 4: 5
```

```
void list_insert(list *l, list *elm, size_t n);
```

Behavior:
Insert at the n-ieme place of the list l the element elm.

```
examples:
list, elm, n -> list after

NULL, List(2,NULL), n -> NULL
-42: , List(2,NULL), n>=0 -> -42:2
-42: 1: 3: 4, List(2,NULL), 1 -> -42: 1: 2: 3: 4
```

```
void insert_in_sorted_list(list* l, list* elm);
```

Behavior: Insert element elm in the sorted list l in order to keep l sorted.

```
examples:
list, elm -> list after

NULL, List(2,NULL) -> NULL
-42:, List(2,NULL) -> -42: 2
-42: 1: 3: 4: 5, List(2,NULL) -> -42: 1: 2: 3: 4: 5
```

```
list *list_pop_front(list *l);
```

Behavior:
Remove and return the first element of the list l.

```
examples:
list before -> list after, elm

NULL -> NULL, NULL
-42: -> -42:, NULL
-42: 2 -> -42:, List(2,NULL)
-42: 1: 2: 3 -> -42: 2: 3, List(val=1,next->2)
```

```
list* list_remove(list* l, size_t n);
```

Behavior:
Remove and return the n-ieme element of the list l. If it does not exist do nothing.

```
examples:
list, n -> list after

-42: 1: 2: 3: 4, 1 -> -42: 2: 3: 4
-42: 1: 2: 3: 4, -1 -> -42: 1: 2: 3: 4
-42: 1: 2: 3: 4, 3 -> -42: 1: 2: 3:
```

```
list* list_search(list* l, int val);
```

Behavior:
Search in the list l if there is an element of the list with as value the in val. If there is return it else return NULL.

```
examples:
l, val -> return

-42:12:98, -42 -> NULL
-42:12:98, 12 -> List(12,-> 98)
-42:12:98, 100 -> NULL
```

```
void FreeList(list* l);
```

Behavior:
Free all the elements of the list l.

QUEUES:

The struct of queue:

```
typedef struct queue queue;
struct queue{
  list* Queue;
};
```

We assume for this part that queues won't be NULL as parameters

---

```
queue* init_queue();
```

Behavior: Have to create and initialise an empty queue. Do not forget to initialse the list of the queue with a sentinelle (-42).

```
void enqueue(queue* q, int elt);
```

Behavior:
Enqueue an element to the list Queue of the queue q. WARNING: elt is not a list element, you have to malloc it.

```
examples:
q, elt -> q after

-42:, 1 -> -42:1
-42: 1: 2: 9, 4 -> -42: 1: 2: 9: 4
```

```
list* dequeue(queue* q);
```

Behavior:
Dequeue an element of the queue q and return it. If there is no element to dequeue return NULL.

```
examples
q -> return, q after

-42: 1: 7: 10 -> List(1,->7), -42: 7: 10
-42: -> NULL, -42:
```

```
int isempty(queue* q);
```

Behavior:
Return true (1) if the queue q is empty, else return false (0).

```
examples:
q -> return

-42: 1: 2 -> 0
-42: -> 1
```

```
void FreeQueue(queue* q);
```

Behavior:
Free all the element of the queue q and free the queue q.

GRAPHS:

The graph struct:

```
typedef struct graph graph;
struct graph{
  int is_directed;
  int order;
  list** adjlists;
};
```

```
graph* init_graph(int order, int is_directed);
```

Behavior:
create and Initialise a graph without links, juste it initialise each list of the array with a sentinelle (-42) and declare if the graph is directed or not.

```
void add_edge(graph* G, int src, int dst);
```

Behavior:
Add an edge between vertice src and vertice dst. Do not forget the case of an non directed graph.
Warning if the src and/or the dst do not exist(s) then do nothing

---

```
void add_vertice(graph* G);
```

Behavior:
Add a vertices to the array of adjlists. You have to realocate data.

```
void backedges_dfs(graph* G);
```

Behavior:
Display backedges (you can let repetitions) of the graph G. It is the same function as saw in algorithm.

```
int components_dfs(graph* G);
```

Behavior:
Return the number of connective components with a dfs algorithm.

```
int components_bfs(graph* G);
```

Behavior:
Return the number of connective components with a bfs algorithm.

```
list* path(graph* G, int src, int dst);
```

Behavior:
Return a list representing the sortest path to go from src to dst.

```
int eccentricity(graph* G, int src);
```

Behavior: Return the eccentricity of the the Vertice src of graph G.

```
list* center(graph* G);
```

Behavior:
Return the list of all element of the center of the graph G.

```
void FreeGraph(graph* g);
```

Free everything in the graph G.

END:

send me email if there is a problem or question: mathieu.merienne@epita.fr

Good Luck !