INFO0947: Complément de programmation
Projet 2: TAD & Récursivité.

Groupe 34: Timothy SMEERS, Soline LÈBRE

10 mai 2021

Table des matières

1 Spécifications des TAD

1.1 Escale_t

Types :

Escale

Utilise :

float

String

Opérations [1] **:**

create_stopover :float × float × String ⇀ Escale

log_time : Escale × float ⇀ Escale

free_stopover : Escale ⇀ void

calculate_range : Escale × Escale ⇀ float

get_name : Escale ⇀ String

get_latitude : Escale ⇀ float

get_longitude : Escale ⇀ float

get_best_time : Escale ⇀ float

Préconditions :

- $\forall\, x \in float,\ x \in [-90, 90]$ && $\forall\, y \in float,\ y \in [-180, 180]$ && $\forall\, name \neq \oslash \Rightarrow create\_stopover(x\ ,\ y\ ,\ *name)$
- $\forall\, x \in Escale \neq \oslash$ && $\forall\, time \geq 0 \Rightarrow log\_time(*x\ ,\ time)$
- $\forall\, x \in Escale \neq \oslash \Rightarrow free\_stopover(*x)$
- $\forall\, x,\ y \in Escale \neq \oslash \Rightarrow calculate\_range(*x\ ,\ *y)$
- $\forall\, x \in Escale \neq \oslash \Rightarrow get\_name(*x)$
- $\forall\, x \in Escale \neq \oslash \Rightarrow get\_latitude(*x)$
- $\forall\, x \in Escale \neq \oslash \Rightarrow get\_longitude(*x)$
- $\forall\, x \in Escale \neq \oslash \Rightarrow get\_best\_time(*x)$

Axiomes :

-

1. Nom des opérations interne
1. Arguments
1. Types de retour
1. Nom des opérations d'observation

1.2 Course_t

1.2.1 Tableau

Types :

Course

Utilise :

Escale
float
unsigned int

Opérations [2] **:**

memory_allocation :Course ⇀ Course
right_shift : Course × unsigned int ⇀ Course
left_shift : Course × unsigned int ⇀ Course
create_table_race : Escale × Escale ⇀ Course
add_table_stopover : Course × Escale × int ⇀ Course
remove_table_stopover : Course × int ⇀ Course
obtain_table_stopover : Course × int ⇀ Escale
free_table_race : Course ⇀ void
is_table_circuit : Course ⇀ unsigned int
race_table_time : Course ⇀ float
get_table_stopover : Course ⇀ unsigned int
get_step : Course ⇀ unsigned int
get_time : Course × int ⇀ float

Préconditions :

- $\forall\, x \in Course, \Rightarrow memory\_allocation(*x)$
- $\forall\, x \in Course, \Rightarrow right\_shift(*x\ ,\ start)$
- $\forall\, x \in Course, \Rightarrow left\_shift(*x\ ,\ start)$
- $\forall\, x,\ y \in Escale \neq \oslash \Rightarrow create\_table\_race(*x\ ,\ *y)$
- $\forall\, x \in Course\ ,\ y \in Escale\ ,\ x\ ,\ y \neq \oslash \Rightarrow add\_table\_stopover(*x\ ,\ *y\ ,\ position)$
- $\forall\, x \in Course\ ,\ x \neq \oslash \Rightarrow remove\_table\_stopover(*x)$
- $\forall\, x \in Course\ ,\ x \neq \oslash \Rightarrow obtain\_table\_stopover(*x\ ,\ position)$
- $\forall\, x \in Course\ ,\ x \neq \oslash \Rightarrow free\_table\_race(*x)$
- $\forall\, x \in Course\ ,\ x \neq \oslash \Rightarrow is\_table\_circuit(*x)$
- $\forall\, x \in Course\ ,\ x \neq \oslash \Rightarrow race\_table\_time(*x)$
- $\forall\, x \in Course\ ,\ x \neq \oslash \Rightarrow get\_table\_stopover(*x)$
- $\forall\, x \in Course\ ,\ x \neq \oslash \Rightarrow get\_step(*x)$
- $\forall\, x \in Course\ ,\ x \neq \oslash\ \&\&\ \forall\, y \in int\ ,\ y\ <\ nbrStopover \Rightarrow get_time(*x, y)$

Axiomes :

-

2. Nom des opérations interne
2. Arguments
2. Types de retour
2. Nom des opérations d'observation

1.2.2 Liste chaînée

Types :

Cell

Utilise :

Escale
Cell
unsigned int

Opérations [3] **:**

create_liste_race : Escale × Escale ⇀ Course
obtain_list_stopover : Course × Escale ⇀ Escale
add_start : Course × Escale ⇀ void*
add_end : Course × Escale ⇀ void*
remove_list_stopover : Course × Escale ⇀ void
print_race : Course ⇀ void
free_list_race : Course ⇀ void
add_list_stopover : Course × Escale × int ⇀ void*
stopover_race_time : Course × Escale ⇀ float
race_list_time : Course⇀ float
is_list_circuit : Course⇀ unsigned int
get_list_stopover : Course⇀ unsigned int

Préconditions :

- $\forall\, x\ ,\ y \in Escale,\ x\ ,\ y \neq \oslash \Rightarrow create\_list\_race(*x\ ,\ *y)$
- $\forall\, x \in Course\ ,\ \forall y \in Escale,\ x\ ,\ y \neq \oslash \Rightarrow obtain\_list\_stopover(*x\ ,\ *y)$
- $\forall\, x \in Course\ ,\ \forall y \in Escale,\ x\ ,\ y \neq \oslash \Rightarrow add\_start(**x\ ,\ *y)$
- $\forall\, x \in Course\ ,\ \forall y \in Escale,\ x\ ,\ y \neq \oslash \Rightarrow add\_end(**x\ ,\ *y)$
- $\forall\, x \in Course\ ,\ \forall y \in Escale,\ x\ ,\ y \neq \oslash \Rightarrow remove\_list\_stopover(**x\ ,\ *y)$
- $\forall\, x \in Course,\ x \neq \oslash \Rightarrow print_race(*x)$
- $\forall\, x \in Course,\ x \neq \oslash \Rightarrow free\_list\_race(*x)$
- $\forall\, x \in Course\ ,\ \forall y \in Escale,\ x\ ,\ y \neq \oslash \Rightarrow add\_list\_stopover(**x\ ,\ *y\ ,\ position)$
- $\forall\, x \in Course\ ,\ \forall y \in Escale,\ x\ ,\ y \neq \oslash \Rightarrow stopover\_race\_time(*x\ ,\ *y)$
- $\forall\, x \in Course,\ x \neq \oslash \Rightarrow race\_list\_time(*x)$
- $\forall\, x \in Course,\ x \neq \oslash \Rightarrow is\_list\_circuit(*x)$
- $\forall\, x \in Course,\ x \neq \oslash \Rightarrow get\_list\_stopover(*x)$

Axiomes :

-

3. Nom des opérations interne
3. Arguments
3. Types de retour
3. Nom des opérations d'observation

2 Déscriptions des TAD

2.1 Escale

Cette structure permettra d'implémenté le TAD Escale permettant de créer une escale avec les position géographique de chaque étapes, et de stocker chaque étapes avec un nom. L'escale comportera le meilleur temps pour la parcourir.

```
struct Escale_t {
    float s_latitude;
    float s_longitude;
    float s_bestTime;
    char *s_name;
};
```

- float *s_latitude*;
  Permet de stocker la latitude d'une étape.
- float *s_longitude*;
  Permet de stocker la longitude d'une étape.
- float *s_bestTime*;
  Permet de stocker le meilleur temps de l'escale.
- char $*$*s_name*;
  Permet de stocker le nom de chaque étapes.

2.2 Course

Cette structure permettra d'implémenté le TAD Course basée sur un tableau. Cette structure comprendra un pointeur sur le TAD escale permettant d'obtenir ses variables. Une taille du tableau qui dépendra du nombre d'escale. Ces variables me permettrons de trouvé le meilleur temps d'une course.

```
struct Course_t {
    Escale **s_stopover;
    float s_bestTime;
    unsigned int s_sizeBoard;
    unsigned int s_nbrStopover;
};
```

- Escale $**$*s_stopover*;
  Un pointeur sur le TAD Escale.
- float *s_bestTime*;
  Permet de stocker le meilleur temps de la course.
- unsigned int *s_sizeBoard*;
  Permet de stocker la taille du tableau.
- unsigned int *s_nbrStopover*;
  Permet de stocker le nombre d'escale présente dans la course.

2.3 Cell

Cette structure permettra d'implémenté le TAD Course basée sur un tableau. Cette structure comprendra un pointeur sur le TAD escale permettant d'obtenir ses variables. Elle contient aussi un pointeur sur la structure Cell. Une variable permettra de trouver le nombre d'escale.

```
typedef Course Cell;

struct Course_t {
   Escale *s_stopover;
   Cell *s_next_stopover;
   unsigned int s_nbrStopover;
};
```

- Escale $*s\_stopover$;
  Un pointeur sur le TAD Escale.
- Cell $*s\_next\_stopover$;
  Un pointeur sur une cellule de la liste.
- unsigned int $s\_nbrStopover$;
  Permet de stocker le nombre d'escale dans la course.

3 Avantages et inconvénients

3.1 Escale

Avantages

Inconvénients

4 Sous-problèmes

SP_1 :
- Créé le TAD Escale permettant de créé une escale et enregistrer un meilleur temps.

SP_2 :
- Créé le TAD course sous forme de tableau.

SP_3 :
- Créé les tests pour le TAD sous forme de tableau.

SP_4 :
- Créé le TAD course sous forme de Liste.

SP_5 :
- Créé les tests pour le TAD sous forme de liste.

$SP_1 \subset SP_2 \subset SP_4 \rightarrow SP_5$
$SP_2 \rightarrow SP_3$

5 Spécification de fonctions

5.1 course_liste.c

```
/**
 * @fn Course create_list_race*(Escale*, Escale*)
 * @brief Allows you to initialize the race structure
 *
 * @pre p_stopover != NULL && p_secondStopover != NULL
 * @post a race was create with connection cell
 * @param p_stopover the first stopover
 * @param p_secondStopover the second stopover
 * @return Course * a memory allowed pointer to the header of the course structure
 *          NULL on error
 */
Course* create_list_race(Escale *p_stopover, Escale *p_secondStopover);
```

```
/**
 * @fn Escale obtain_list_stopover*(Course*, Escale*)
 * @brief getter of the data of a cell
 *
 * @pre p_race != NULL && p_stopover != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 * @param p_stopover a pointer to the Escale structure
 * @return Escale *a pointer to the data Escale structure
 *          NULL on error
 */
Escale* obtain_list_stopover(Course *p_race, Escale *p_stopover);
```

```
/**
 * @fn void add_start*(Course**, Escale*)
 * @brief Allows you to add a stopover at the start
 *
 * @pre p_race != NULL && p_stopover != NULL
 * @post p_stopover0 < p_stopover
 * @param p_race a pointer to the Course structure
 * @param p_stopover a pointer to the Escale structure
 */
void* add_start(Course **p_race, Escale *p_stopover);
```

```
/**
 * @fn void add_end*(Course**, Escale*)
 * @brief Allows you to add a stopover at the end
 *
 * @pre p_race != NULL && p_stopover != NULL
 * @post p_stopover0 < p_stopover
 * @param p_race a pointer to the Course structure
 * @param p_stopover a pointer to the Escale structure
 */
void* add_end(Course **p_race, Escale *p_stopover);
```

```
/**
 * @fn void remove_list_stopover(Course**, Escale*)
 * @brief Allows you to remove a stopover to the structure
 *
 * @pre p_race != NULL && p_stopover != NULL
 * @post a stopover has been removed
```

```
 * @param p_race a pointer to the Course structure
 * @param p_stopover a pointer to the Escale structure
 */
void remove_list_stopover(Course **p_race, Escale *p_stopover);
```

```
/**
 * @fn void print_race(Course*)
 * @brief Allows you to display the race in the console
 *
 * @pre p_race != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 */
void print_race(Course *p_race);
```

```
/**
 * @fn void free_list_race(Course*)
 * @brief Used to free the memory of the Course structure
 *
 * @pre p_race != NULL
 * @post Course *p_race is released
 * @param p_race a pointer to the Course structure
 */
void free_list_race(Course *p_race);
```

```
/**
 * @fn void add_list_stopover*(Course**, Escale*, int)
 * @brief Allows you to add a stopover to the structure
 *
 * @pre p_race != NULL && p_stopover != NULL
 * @post a stopover has been added
 * @param p_race a pointer to the Course structure
 * @param p_stopover a pointer to the Escale structure
 * @param p_position the position in the list
 */
void *add_list_stopover(Course **p_race, Escale *p_stopover, int p_position);
```

```
/**
 * @fn float stopover_race_time(Course*, Escale*)
 * @brief allows you to find the best time of the stage
 *
 * @pre p_race != NULL && p_stopover != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 * @param p_stopover  pointer to the Escale structure
 * @return float l_time
 *          NULL on error
 */
float stopover_race_time(Course *p_race, Escale *p_stopover);
```

```c
/**
 * @fn float race_table_time(Course*)
 * @brief get time to race
 *
 * @pre p_race != NULL
 * @post p_race->s_bestTime0 != p_race->s_bestTime
 * @param p_race a pointer to the Course structure
 * @return float p_race->s_bestTime
 */
float race_list_time(Course *p_race);
```

```c
/**
 * @fn unsigned int is_list_circuit(Course*)
 * @brief Used to determine if the race is a circuit
 *
 * @pre p_race != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 * @return 0 if the race is not a circuit
 *          1 else
 */
unsigned int is_list_circuit(Course *p_race);
```

```c
/**
 * @fn unsigned int get_list_stopover(Course*)
 * @brief getter of the number of stopover
 *
 * @pre p_race != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 * @return unsigned int p_race->s_nbrStopover
 */
unsigned int get_list_stopover(Course *p_race);
```

5.2 course_tableau.c

```c
/**
 * @fn Course memory_allocation*(Course*)
 * @brief Allows memory to be reallocated for a race
 *
 * @pre p_race != NULL
 * @post p_race memory0 < p_race memory
 * @param p_race a pointer to the Course structure
 * @return Course * a pointer to the header of the course structure
 */
static Course* memory_allocation(Course *p_race);
```

```c
/**
 * @fn Course right_shift*(Course*, unsigned int)
 * @brief Allows you to shift a box in the table to the right
 *
 * @pre p_race != NULL
 * @post p_race->s_sizeBoard < p_race->s_nbrStopover
 * @param p_race a pointer to the Course structure
 * @param p_start the nth cell of the table to shift
 * @return Course * a pointer to the header of the course structure
 */
static Course* right_shift(Course *p_race, unsigned int p_start);
```

```c
/**
 * @fn Course left_shift*(Course*, unsigned int)
 * @brief Allows you to shift a box in the table to the left
 *
 * @pre p_race != NULL
 * @post p_race->s_sizeBoard < p_race->s_nbrStopover
 * @param p_race a pointer to the Course structure
 * @param p_start the nth cell of the table to shift
 * @return Course * a pointer to the header of the course structure
 */
static Course* left_shift(Course *p_race, unsigned int p_start);
```

```c
/**
 * @fn Course create_table_race*(Escale*, Escale*)
 * @brief Allows you to initialize the race structure
 *
 * @pre p_stopover != NULL && p_secondStopover != NULL
 * @post a race was create with connection cell
 * @param p_stopover the first stopover
 * @param p_secondStopover the second stopover
 * @return Course * a memory allowed pointer to the header of the course structure
 *          NULL on error
 */
Course* create_table_race(Escale *p_stopover, Escale *p_secondStopover);
```

```c
/**
 * @fn Course add_table_stopover*(Course*, Escale*, int)
 * @brief Allows you to add a stopover to the structure
 *
 * @pre p_race != NULL && p_stopover != NULL && p_position > 0
 * @post a stopover has been added
 * @param p_race a pointer to the Course structure
 * @param p_stopover a pointer to the Escale structure
 * @param p_position the position in the list
 * @return Course * a pointer to the header of the course structure
 *          NULL on error
 */
Course* add_table_stopover(Course *p_race, Escale *p_stopover, int p_position);
```

```c
/**
 * @fn Course remove_table_stopover*(Course*, int)
 * @brief Allows you to remove a stopover to the structure
 *
 * @pre p_race != NULL && p_position > 0
 * @post a stopover has been removed
 * @param p_race a pointer to the Course structure
 * @param p_position the position in the list
 * @return Course * a pointer to the header of the course structure
 *          NULL on error
 */
Course* remove_table_stopover(Course *p_race, int p_position);
```

```c
/**
 * @fn Escale obtain_table_stopover*(Course*, int)
 * @brief getter of the data of a cell
 *
 * @pre p_race != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 * @param p_position the position in the list
 * @return Escale *a pointer to the data Escale structure
 *          NULL on error
 */
Escale* obtain_table_stopover(Course *p_race, int p_position);
```

```c
/**
 * @fn void free_table_race(Course*)
 * @brief Used to free the memory of the Course structure
 *
 * @pre p_race != NULL
 * @post Course *p_race is released
 * @param p_race a pointer to the Course structure
 */
void free_table_race(Course *p_race);
```

```c
/**
 * @fn unsigned int is_table_circuit(Course*)
 * @brief Used to determine if the race is a circuit
 *
 * @pre p_race != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 * @return 0 if the race is not a circuit
 *          1 else
 */
unsigned int is_table_circuit(Course *p_race);
```

```c
/**
 * @fn float race_table_time(Course*)
 * @brief get time to race
 *
 * @pre p_race != NULL
 * @post p_race->s_bestTime0 != p_race->s_bestTime
 * @param p_race a pointer to the Course structure
 * @return float p_race->s_bestTime
 */
float race_table_time(Course *p_race);
```

```c
/**
 * @fn unsigned int get_table_stopover(Course*)
 * @brief getter of the number of stopover
 *
 * @pre p_race != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 * @return unsigned int p_race->s_nbrStopover
 */
unsigned int get_table_stopover(Course *p_race);
```

```c
/**
 * @fn unsigned int get_step(Course*)
 * @brief getter of the number of step
 *
 * @pre p_race != NULL
 * @post /
 * @param p_race a pointer to the Course structure
 * @return unsigned int p_race->s_nbrStopover -1
 */
unsigned int get_step(Course *p_race);
```

```c
/**
 * @fn float get_time(Course*, int)
 * @brief
 *
 * @pre p_race != NULL && p_position < p_race->s_nbrStopover
 * @post /
 * @param p_race a pointer to the Course structure
 * @param p_position the position in the list
 * @return float l_time
 */
float get_time(Course *p_race, int p_position);
```

6 Documentation

Pour plus d'informations sur le code vous pouvez consulter le site internet contenant la documentation doxygen.