INFO0947: FLAMME OLYMPIQUE

Groupe 26: Franck Duval HEUBA BATOMEN, Bilali ASSALNI

Contents

1 Introduction

la flamme Olympique arrive dans le pays hôte c'est l'esprit des Jeux qui débarque. Avant la cérémonie d'ouverture, la flamme, portée par une multitude de relayeurs, réalise un parcours jusqu'à la ville hôte des Jeux. Ainsi, elle devra parcourir un ensemble de villes, constituant ainsi un itinéraire jusqu'à la ville pour la cérémonie d'ouverture. Ainsi, le travail que nous avons réalisé à consisté à numériser ce parcours de ville en ville, region en region.

2 Spécifications Abstraites

Nous avons principalement deux types abstraits de données:

- Region
- ItineraireFlame

2.1 TAD Region

2.1.1 Syntaxe

Type: Region

Utilise:

- Integer
- String
- Double

Opérations:

- create: Double × Double × String → Region
- get_x: Region → Double
- get_y: Region → Double
- get_nb_people: Region → Integer
- get_headquater: Region → String
- get_name: Region → String
- get_speciality: Region → String
- distance: Region × Region → Double
- set_x: Region × Double → Region
- set_y: Region × Double → Region
- set_headquater: Region × String → Region
- set_speciality: Region × String → Region
- set_nb_people: Region × Integer → Region
- destroy: Region → ∅

2.1.2 Sémantique

Préconditions:

$\forall j \in$ Integer, $\forall k \in$ Region

$\forall j \geq 0$, set_nb_people(k, j)

Axiomes:

$\forall r \in$ Region, $\forall i \in$ Double, $\forall j \in$ Integer, $\forall s \in$ String

$get\_x(set\_x(r, i)) = i$

$get\_y(set\_y(r, i)) = i$

$get\_speciality(set\_speciality(r, s)) = s$

$get\_headquater(set\_headquater(r, s)) = s$

		Opérations Internes			
		$create(\cdot)$	$set\_x(\cdot)$	$set\_y(\cdot)$	$set\_headquater(\cdot)$
Observateurs	$get\_x(\cdot)$	✓	✓	✓	✓
	$get\_y(\cdot)$	✓	✓	✓	✓
	$get\_headquater(\cdot)$	✓	✓	✓	✓
	$get\_name(\cdot)$	✓	✓	✓	✓
	$get\_speciality(\cdot)$	✓	✓	✓	✓

		Opérations Internes		
		$set\_nb\_people(\cdot)$	$set\_speciality(\cdot)$	$destroy(\cdot)$
Observateurs	$get\_x(\cdot)$	✓	✓	$\emptyset$
	$get\_y(\cdot)$	✓	✓	$\emptyset$
	$get\_headquater(\cdot)$	✓	✓	$\emptyset$
	$get\_name(\cdot)$	✓	✓	$\emptyset$
	$get\_speciality(\cdot)$	✓	✓	$\emptyset$

2.2 TAD ItineraireFlame

2.2.1 Syntaxe

Type: ItineraireFlame

Utilise:

- Region
- Boolean
- Integer

Opérations:

- create: Region × Region → ItineraireFlame
- is_circuit: ItineraireFlame → Boolean
- count_region: ItineraireFlame → Integer
- count_people: ItineraireFlame → Integer
- add_region: ItineraireFlame × Region → ItineraireFlame
- remove_region: ItineraireFlame × Region → ItineraireFlame
- destroy: ItineraireFlame → ∅

2.2.2 Sémantique

Préconditions:

$\forall i, j \in$ Region

$\forall i, j$, create(i, j)

Axiomes:

$\forall r_0, r \in$ Region, $\forall j \in$ Integer, $\forall k \in$ ItineraireFlame

$count\_region(add\_region(k, r)) = count\_region(k) + 1$

$is\_circuit(add\_region(create(r_0, r), r)) = True$

$is\_circuit(add\_region(create(r_0, r), r_0)) = True$

$is\_circuit(create(r_0, r)) = False$

		Opérations Internes			
		$create(\cdot)$	$add\_region(\cdot)$	$remove\_region(\cdot)$	$destroy(\cdot)$
Observateurs	$is\_circuit(\cdot)$	✓	✓	✓	∅
	$count\_region(\cdot)$	✓	✓	✓	∅
	$count\_people(\cdot)$	✓	✓	✓	∅

3 Specifications

3.1 Specifications Region:

3.1.1 Création d'un objet Region

- @Pre: $name \in String \wedge name \neq \emptyset$
- @Post: $new\_region = x \wedge x \in Region$

```c
/**
 * @brief
 * Cette fonction va creer et retourner un nouvel
 * objet Region
*/
struct Region_t *new_region(double x, double y, char *name);
```

3.1.2 Coordonné X

- @Pre: $region \neq NULL$
- @Post: $get\_coord\_x \in \Re$

```c
/**
    * @brief
    * retourne la coordonné x de l'objet Region
*/
double get_coord_x(struct Region_t *region);
```

3.1.3 Coordonné Y

- @Pre: $region \neq NULL$
- @Post: $get\_coord\_y \in \Re$

```c
/**
    * @brief
    * retourne la coordonné y de l'objet Region
*/
double get_coord_y(struct Region_t *region);
```

3.1.4 Nom de la Region

- @Pre: $region \neq NULL$
- @Post: $get\_region\_name \in String$

```c
/**
    * @brief
    * retourne le nom de la Region
*/
char *get_region_name(struct Region_t *region);
```

3.1.5 Chef Lieu de la region

3.1.6 Accésseur

- @Pre: $region \neq NULL$
- @Post: $get\_region\_headquater \in String$

```
/**
    * @brief
    * retourne le chef lieu de la Region
*/
char *get_region_headquater(struct Region_t *region);
```

3.1.7 Mutateur

- @Pre: $region \neq NULL \wedge headquater \neq NULL$
- @Post: /

```
/**
    * @brief
    * modifie le chef lieu d'une region
*/
void set_region_headquater(struct Region_t *region, char *headquater);
```

3.1.8 Spécialité de la region

3.1.9 Accésseur

- @Pre: $region \neq NULL$
- @Post: $get\_region\_speciality \in String$

```
/**
    * @brief
    * retourne la spécialité de la Region
*/
char *get_region_speciality(struct Region_t *region);
```

3.1.10 Mutateur

- @Pre: $region \neq NULL \wedge speciality \neq NULL$
- @Post: /

```
/**
    * @brief
    * Modifie la spécialité de la region
*/
void set_region_speciality(struct Region_t *region, char *speciality);
```

3.1.11 Nombre d'habitant de la region

3.1.12 Accésseur

- @Pre: $region \neq NULL$
- @Post: $get\_nb\_people \geq 0$

```
/**
    * @brief
    * retourne le nombre d'habitant d'un region
*/
unsigned int get_nb_people(struct Region_t *region);
```

3.1.13 Mutateur

- @Pre: $region \neq NULL \wedge nb\_people \geq 0$
- @Post: /

```
/**
    * @brief
    * Modifie le nombre d'habitant de la region
*/
void set_nb_people(struct Region_t *region, unsigned int nb_people);
```

3.1.14 Distance entre deux region

- @Pre: $region\_1 \neq NULL \wedge region\_2 \neq NULL$
- @Post: $distance\_between\_region \in \Re$

```
/**
    * @brief
    * retourne la distance entre region_1 et region_2
*/
double distance_between_region(Region *region_1, Region *region_2);
```

3.1.15 Destruction de l'objet region

- @Pre: $region \neq NULL$
- @Post: /

```
/**
    * @brief
    * libère l'espace mémoire prévu pour region
*/
void destroy_region(struct Region_t *region);
```

3.2 Specifications ItineraireFlame:

3.2.1 Création de l'objet ItineraireFlame_t

- @Pre: $start \neq NULL \wedge end \neq NULL$
- @Post: $new\_itineraireflame \in structItineraireFlame\_t$

```
/**
    * @brief
    * Crée un objet de type ItineraireFlame_t
*/
struct ItineraireFlame_t *new_itineraireflame(Region *start, Region *end);
```

3.2.2 Vérifier si un itinéraire est un circuit

- @Pre: $way \neq NULL$
- @Post: $is\_circuit \in Boolean$

```
/**
    * @brief
    * Vérifie si l'itinéraire est un circuit
*/
unsigned int is_circuit(struct ItineraireFlame_t *way);
```

3.2.3 Compter le nombre de region

- @Pre: $way \neq NULL$
- @Post: $count\_region \geq 0$

```
/**
    * @brief
    * Retourne le nombre de regions différentes de l'itinéraire
*/
unsigned int count_region(struct ItineraireFlame_t *way);
```

3.2.4 Compter le nombre d'habitant de l'itinéraire

- @Pre: $way \neq NULL$
- @Post: $count\_resident \geq 0$

```
/**
    * @brief
    * Retourne le nombre d'habitant de l'itinéraire
*/
unsigned int count_resident(struct ItineraireFlame_t *way);
```

3.2.5 Ajouter une region sur un itinéraire

- @Pre: $way \neq NULL \wedge region \neq NULL$
- @Post: $add\_region == 1 \wedge region \in way$

```
/**
    * @brief
    * Ajoute une region sur un itinéraire
*/
unsigned int add_region(struct ItineraireFlame_t *way, Region *region);
```

3.2.6 Supprime une region d'un itinéraire

- @Pre: $way \neq NULL \wedge region \neq NULL$
- @Post: $remove\_region == 1 \wedge region \notin way$

```
/**
    * @brief
    * Supprime une region d'un itinéraire
*/
void remove_region(struct ItineraireFlame_t *way, Region *region);
```

3.2.7 Supprime une region d'un itinéraire

- @Pre: $way \neq NULL \wedge 0 \leq delete\_regions \leq 1$
- @Post: /

```
/**
    * @brief
    * L'ibère l'espace mémoire reservé pour un itinéraire
*/
void destroy_itineraireflame(ItineraireFlame *way, int delete_regions);
```

4 Invariants

$In(a, b)$: Permet de vérifier si b est dans a

a: Est un tableau de Region de taille ARRAY_SIZE

b: b est de type Region

$CountRegion(map)$: Compte le nombre de region différentes dans map

map: Est un tableau de Region de taille ARRAY_SIZE

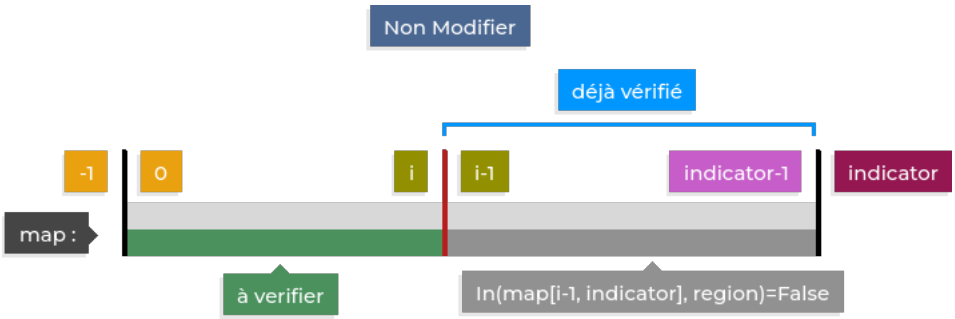
$CountHabitant(map)$: Compte le nombre d'habitant des regions différentes de map

map: Est un tableau de Region de taille ARRAY_SIZE

4.1 Recherche d'une region sur l'itinéraire: $search(map, region)$

- region: de type Region
- map[0, indicator]: est un tableau de region

4.1.1 Invariant Graphique



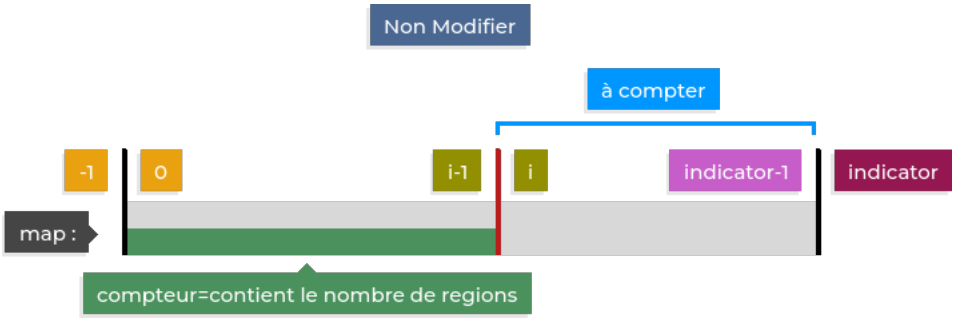
4.1.2 Invariant Formel

INV $\equiv map = map_0 \wedge -1 \leq i < indicator$

4.2 Compter le nombre de region sur l'itinéraire: $count\_region(map)$

- map[0, indicator]: est un tableau de region

4.2.1 Invariant Graphique

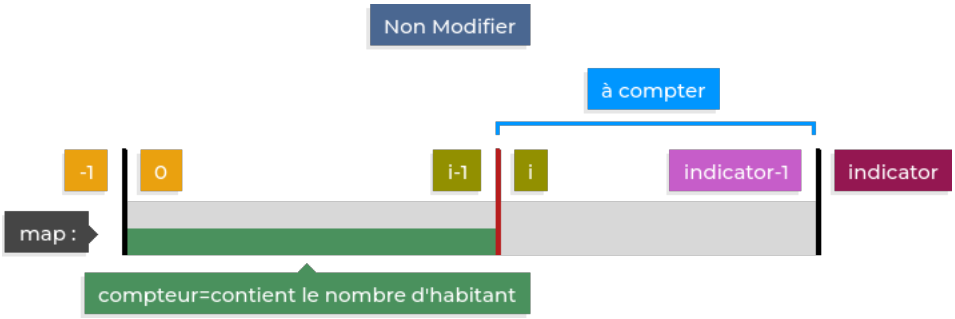


4.2.2 Invariant Formel

$$\text{INV} \equiv map = map_0 \wedge 0 \leq i \leq indicator \wedge compteur = CountRegion(map[0, i])$$

4.3 Compter le nombre d'habitant de l'itinéraire: $count\_resident(map)$

4.3.1 Invariant Graphique

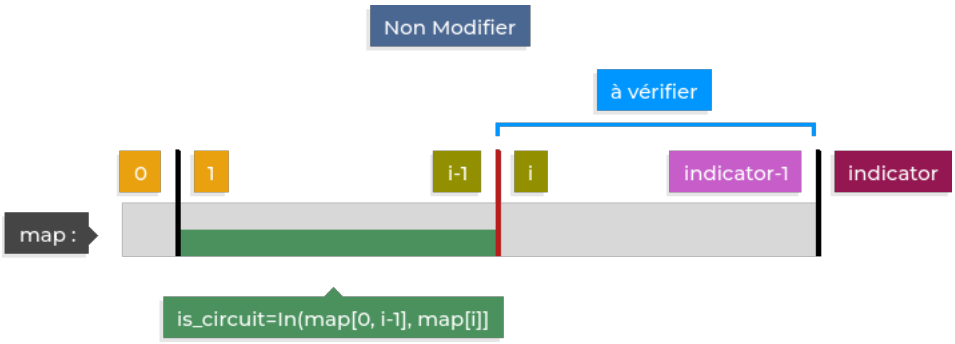


4.3.2 Invariant Formel

$$\text{INV} \equiv map = map_0 \wedge 0 \leq i \leq indicator \wedge compteur = CountHabitant(map[0, i])$$

4.4 Vérifier si un itinéraire est un circuit: $is\_circuit(map)$

4.4.1 Invariant Graphique



4.4.2 Invariant Formel

$$\text{INV} \equiv map = map_0 \wedge 0 \leq i \leq indicator \wedge is\_circuit = In(map[0, i-1], map[i])$$

5 Implémentations Récursives

Nous avons implémenté recursivement la fonction $in\_list(head, value)$, qui permet de vérifier si une value se trouve parmis les élements précédents d'un Noeud, et va retourner 1 dans le cas favorable sinon 0.

- head: est le dernier Noeud d'une liste d'oublement chainée
- value: est la valeur recherchée dans la liste

5.1 Conception:

- Cas de base: ici deux cas sont possibles
  - $head == NULL$ dans ce cas l'élément n'est pas dans la liste on retourne 0
  - $head \neq NULL \wedge head \rightarrow value == value$ dans ce cas on retourne 1
- Cas non récursif: $head \neq NULL \wedge head \rightarrow value \neq value$

5.2 Implémentation:

```
1 static unsigned int in_list(struct Node_t *last, Region *value)
2 {
3     struct Node_t *head = last;
4     if (head != NULL)
5     {
6         if (head->value == value)
7             return 1;
8         return in_list(head->prev, value);
9     }
```

```
10
11      return 0;
12  }
```

6 Complexité

Le projet est divisé en deux partie, Region et ItineraireFlame.

6.1 Partie Region:

Dans cette partie nous n'avons que des accésseurs et des mutateurs, ainsi, ils fonctionnent en $O(1)$ sur les champs de la structure car aucun tableau ou liste est utilisé.

6.2 Partie ItineraireFlame:

6.2.1 Create

```
1  struct ItineraireFlame_t *new_itineraireflame(Region *start, Region *end)
2  {
3      assert(start != NULL && end != NULL);
4
5      //T1
6      if (start == end)
7          return NULL;
8
9      //T2
10     struct ItineraireFlame_t *way = malloc(sizeof(struct ItineraireFlame_t));
11
12     //T3
13     if (way == NULL)
14         return NULL;
15
16     //T4
17     way->map = malloc(sizeof(Region *) * ARRAY_SIZE);
18
19     //T5
20     if (way->map == NULL)
21     {
22         free(way);
23         return NULL;
24     }
25
26     //T6
27     way->departure = start;
28     way->arrival = end;
29     way->indicator = 0;
30
31     return way;
32 }
```

La complexité ici est égale à la somme de T_1 à T_6
$T(.) = \sum_{i=1}^{6} T_i$

- $T_1 = 1$
- $T_2 = 1$

- $T_3 = 1$
- $T_5 = 1$
- $T_6 = 1$

Ainsi $T(.) = 6 \rightarrow T(.) \in O(1)$

6.2.2 Vérifier si un itineraireflame est un circuit

```
unsigned int is_circuit(struct ItineraireFlame_t *way)
{
    assert(way != NULL);

    //T1
    if (way->indicator == 0)
        return 0;

    //T2
    for (int i = 0; i < way->indicator; i++)
    {
        //T2.1
        if(way->map[i] == way->departure || way->map[i] == way->arrival)
            return 1;

        //T2.2
        for(int y=i-1; y>=0; y--)
        {
            //T2.2.1
            if(way->map[y] == way->map[i])
                return 1;
        }
    }

    //T3
    return 0;
}
```

$T(.) = T_1 + T_2 + T_3$

$T_1 = 1$

$T_2 = (T_{2.1} + T_{2.2}) * indicator$

$T_{2.1} = 1$

$T_{2.2} = y * T_{2.2.1} = indicator$

$T_3 = 1$

Ainsi: $T_2 = (1 + indicator) * indicator \wedge 0 \leq indicator \leq ARRAY\_SIZE$
$T_2 = ARRAY\_SIZE^2 + ARRAY\_SIZE$
$T(.) = ARRAY\_SIZE^2 + ARRAY\_SIZE + 2$
Conclusion: $T(.) \in O(ARRAY\_SIZE^2)$

6.2.3 Compter le nombre de region

```
unsigned int count_region(struct ItineraireFlame_t *way)
{
    assert(way != NULL);

    Region *tmp[ARRAY_SIZE];
    unsigned int is_counted, counter = 2;

    tmp[0] = way->departure;
    tmp[1] = way->arrival;

    for (int i = 0; i < way->indicator; i++)
    {
        is_counted = 0;
        for(int y = counter-1; y >= 0 && !is_counted; y--)
            is_counted = (tmp[y] == way->map[i]);

        if(!is_counted)
            tmp[counter++] = way->map[i];
    }

    return counter;
}
```

Par le même processus que celui de précédent on obtient: $T(.) \in O(ARRAY\_SIZE^2)$

6.2.4 Ajouter une région

```
unsigned int add_region(struct ItineraireFlame_t *way, Region *region)
{
    assert(way != NULL && region != NULL);

    //T1
    if (way->indicator >= ARRAY_SIZE)
        return 0;

    //T2
    way->map[way->indicator++] = region;

    return 1;
}
```

$T(.) = T_1 + T_2 = 2$
Conclusion: $T(.) \in O(1)$

6.2.5 Supprimer une région de l'itineraireflame

```
void remove_region(struct ItineraireFlame_t *way, Region *region)
{
    assert(way != NULL && region != NULL);

    //T1
    int i = way->indicator;

    //T2
    while (i >= 0)
    {
```

```
11          //T2.1
12          if (way->map[i] == region)
13          {
14              //T2.1.1
15              for (int y = i+1; i < way->indicator; i++)
16                  way->map[y-1] = way->map[y];
17
18              //T2.1.2
19              way->indicator--;
20          }
21
22          //T2.2
23          i--;
24      }
25  }
```

$T(.) = T_1 + T_2$

$T_1 = 1$

$T_2 = indicator * (T_{2.1} + T_{2.2})$

$T_{2.1} = T_{2.1.1} + T_{2.1.2}$

$T_{2.1.1} = indicator$

$T_{2.1.2} = 1$

$T_{2.2} = 1$

Ainsi:
$\rightarrow T_{2.1} = indicator + 1 \wedge T_{2.2} = 1$
$\rightarrow T_2 = indicator * (indicator + 2)$
$\rightarrow T(.) = indicator^2 + indicator + 1 \wedge 0 \leq indicator \leq ARRAY\_SIZE$
$\rightarrow T(.) = ARRAY\_SIZE^2 + ARRAY\_SIZE + 1$
Conclusion: $T(.) \in O(ARRAY\_SIZE^2)$

7 Tests Unitaires

Comme il y'a deux implémentations du module "itineraireflame.h", les tests unitaires ont étés divisés en deux parties:

- Les tests de l'implémentations avec les listes
- Les tests de l'implémentations avec les tableaux

Chacun de ces tests commencent par des tests sur le module "region.h", notaments:

- La creation d'une region
- Les tests des Accésseurs
- Les tests des Mutateurs

Puis on fait des tests de calculs de distance entre les régions.
Une fois, les tests sur le module "region.h" terminés, on commencent les tests sur accésseurs basics du type ItineraireFlame, c'est à dire:

- get_coord_x
- get_coord_y
- get_region_name

Ces tests passés, on test l'ajout et la suppréssion des regions et le comptage des regions.

8 Conclusion

En conclusion, il a été question tout au long de ce projet:

D'abord de faire la conception des types abstraits utilisés

- Region
- ItineraireFlame

Ensuite d'implémenter les types abstraits

Le type ItineraireFlame devant utiliser une structure de données itérative, nous avons dû faire deux implémentations, donc une en utilisans des tableaux et l'autre en utilisans une liste doublement chainées.
Ainsi, nous avons procédé à l'établissement des invariants graphiques et formels pour chaque sous-problèmes.
Enfin, nous avons dû procéder à la rédaction du rapport.