INFO0947: Projet 2
Types Abstraits de Données

Groupe 10: Cyril RUSSE

Table des matières

1 Introduction

1.1 Contexte

En référence à la BD Astérix et Obelix, "Le tour de Gaule d'Astérix", ce travail a pour but d'implémenter ce fameux tour sous la forme d'un type abstrait de données.

1.2 Enoncé

1.2.1 Types abstraits

Dans le cadre de ce projet, nous avions pour consigne de spécifier deux types abstraits :

1. **Ville**
   Ce type abstrait doit permettre de :
   — créer une ville à partir de ses deux coordonnées X et Y et de son nom ;
   — obtenir la coordonnée X d'une escale ;
   — obtenir la coordonnée Y d'une escale ;
   — obtenir le nom de la ville ;
   — calculer la distance géographique entre deux ville ;
   — enregistrer la spécialité gastronomique de la ville ;
   — obtenir la spécialité gastronomique de la ville ;
2. **Gaule**
   Ce type abstrait doit permettre de :
   — créer un tour de Gaule sur base de deux villes. Par définition, un tour nouvellement créé ne peut pas constituer un circuit ;
   — déterminer si un tour de Gaule constitue un circuit ;
   — déterminer le nombre de villes visitées durant le tour ;
   — déterminer le nombre totale de spécialités gastronimiques dans le tour ;
   — déterminer la spécialité gastronomique d'une ville du tour ;
   — ajouter une ville à un tour ;
   — supprimer une ville d'un tour ;

1.2.2 Représentation Concrète

Ces types abstraits doivent être représentés de 2 manières :
— Un tableau
— Une liste chainée

2 Signature des TAD

2.1 Ville

Type :
— Ville
Utilise :
— $String$
— $\mathbb{R}$
Opérations :
— creer_ville : $String \times \mathbb{R} \times \mathbb{R} \rightarrow Ville$
— get_x_ville : $Ville \rightarrow \mathbb{R}$
— get_y_ville : $Ville \rightarrow \mathbb{R}$

— get_nom_ville : $Ville \rightarrow String$
— get_specialite_ville : $Ville \rightarrow String$
— set_specialite_ville : $Ville \times String \rightarrow Ville$
— distance_entre_2_villes : $Ville \times Ville \rightarrow \mathbb{R}$

Préconditions :
— $\emptyset$

Axiomes : $\forall x, y \in \mathbb{R} \wedge \forall V_1, V_2 \in Ville \wedge \forall s, t \in String$
— distance_entre_2_villes(V_1, V_2) = $\sqrt{(get\_x\_ville(V_2) - get\_x\_ville(V_1))^2 + (get\_y\_ville(V_2) - get\_y\_ville(V_1))^2}$
— distance_entre_2_villes(set_specialite_ville(V_1, s), set_specialite_ville(V_2, t)) = distance_entre_2_villes(V_1, V_2)
— get_x_ville(creer_ville(s, x, y)) = x
— get_y_ville(creer_ville(s, x, y)) = y
— get_x_ville(set_specialite_ville(V_1, s)) = get_x_ville(V_1)
— get_y_ville(set_specialite_ville(V_1, s)) = get_y_ville(V_1)
— get_nom_ville(creer_ville(s, x, y)) = s
— get_nom_ville(set_specialite_ville(V_1, s)) = get_nom_ville(V_1)
— get_specialite_ville(set_specialite_ville(V_1, s)) = s
— get_specialite_ville(creer_ville(s, x, y)) = NULL

2.2 Gaule

Type :
— Gaule

Utilise :
— $Ville$
— $Entiers$
— $String$
— $Booleen$

Opérations :
— cree_nouveau_tour : $Ville \times Ville \rightarrow Gaule$
— get_nombre_villes : $Gaule \rightarrow Entiers$
— ajoute_ville : $Gaule \times Ville \rightarrow Gaule$
— supprime_ville : $Gaule \rightarrow Gaule$
— get_est_circuit : $Gaule \rightarrow Booleen$
— get_nombre_specialite : $Gaule \rightarrow Entiers$
— get_specialite : $Gaule \times String \rightarrow String$
— ville_en_double : $Gaule \times Ville \rightarrow Booleen$

Préconditions :

Axiomes :$\forall G \in Gaule, \forall V_1, V_2, \in Ville$
— get_nombre_villes(ajoute_ville(G, V_1)) = get_nombre_villes(G)+1
— get_nombre_villes(supprime_ville(G)) = get_nombre_villes(G)-1
si get_nombre_villes(G)> 0
sinon get_nombre_villes(supprime_ville(G)) = get_nombre_villes(G)
— get_nombre_villes(cree_nouveau_tour(V_1, V_2)) = 2
— get_nombre_specialite(ajoute_ville(G, V_1)) = get_nombre_specialite(G)+1
si ville_en_double(G, V_1)=False $\wedge$ get_specialite$(G, V_1)\neq$NULL
sinon get_nombre_specialite(ajoute_ville(G, V_1)) = get_nombre_specialite(G)

— get_nombre_specialite(cree_nouveau_tour(V_1, V_2)) = 2
si get_specialite(G, V_1)≠NULL ∧ get_specialite(G, V_2)≠NULL
— get_nombre_specialite(cree_nouveau_tour(V_1, V_2)) = 1
si get_specialite(G, V_1)≠NULL ∧ get_specialite(G, V_2)=NULL
∨get_specialite(G, V_2)≠NULL ∧ get_specialite(G, V_1)=NULL
— get_nombre_specialite(cree_nouveau_tour(V_1, V_2)) = 0
si get_specialite(G, V_1)=NULL ∧ get_specialite(G, V_2)=NULL
— get_est_circuit(cree_nouveau_tour(V_1, V_2)) = False
— get_specialite(cree_nouveau_tour(V_1, V_2), V_1) = get_specialite_ville(V_1)
— get_specialite(cree_nouveau_tour(V_1, V_2), V_2) = get_specialite_ville(V_2)
— get_specialite(ajoute_ville(G, V_1), V_1) = get_specialite_ville(V_1)

3 Spécifications des TAD

3.1 Ville

3.1.1 Structure

```c
struct Ville_t{
char *nom;
float x;
float y;
char *specialite;
};
```

Extrait de Code 1 – Structure "Ville"

3.1.2 Spécification des fonctions et procédures de ville.h

```c
/*
*@pre : nom ≠ NULL ∧ x = x0 ∧ y = y0 ∧ nom = nom0
*@post : ville_init ∧ x = x0 ∧ y = y0 ∧ nom = nom0 ∧ get_x_ville(ville) = ville− > x∧
*get_y_ville(ville) = ville− > y ∧ get_nom_ville(ville) = ville− > nom
*/
Ville *creer_ville(char *nom, float x, float y);

/*
*@pre : ∅
*@post : ville = NULL
*/
void detruit_ville(Ville *ville);

/*
*@pre : ville ≠ NULL
*@post : ville = ville0 ∧ get_x_ville(ville) = ville− > x
*/
float get_x_ville(Ville *ville);

/*
*@pre : ville ≠ NULL
*@post : ville = ville0 ∧ get_y_ville(ville) = ville− > y
*/
float get_y_ville(Ville *ville);

/*
```

```
27      *@pre : ville ≠ NULL
28      *@post : ville = ville0 ∧ get_nom_ville(ville) = ville− > nom
29      */
30      char *get_nom_ville(Ville *ville);
31
32      /*
33      *@pre : ville ≠ NULL
34      *@post : ville = ville0 ∧ get_specialite_ville(ville) = ville− > specialite
35      */
36      char *get_specialite_ville(Ville *ville);
37
38      /*
39      *@pre : ville ≠ NULL ∧ specialite ≠ NULL ∧ specialite = specialite0
40      *@post : ville = ville0 ∧ specialite = specialite0 ∧ get_specialite_ville(ville) = specialite
41      */
42      void set_specialite_ville(Ville *ville, char *specialite);
43
44      /*
45      *@pre : ville1 = ville10 ≠ NULL ∧ ville2 = ville20 ≠ NULL
46      *@post : ville1 = ville10 ∧ ville2 = ville20∧
47      *distance_entre_2_villes(ville1, ville2) =
48      *√((get_x_ville(ville2) − get_x_ville(ville1))² + (get_y_ville(ville2) − get_y_ville(ville1))²
49      */
50      float distance_entre_2_villes(Ville *ville1, Ville *ville2);
51
52      /*
53      *@pre : ∅
54      *@post : retourne la taille mémoire de la struct Ville
55      */
56      int size_ville(void);
```

Extrait de Code 2 – Spécification des fonctions et procédures du header "ville.h"

3.2 Gaule

3.2.1 Structure en Tableau

```
1       struct Gaule_t{
2       Ville **tableau_ville;
3       int nombre_villes;
4       int est_circuit;
5       int nombre_specialites;
6       };
```

Extrait de Code 3 – Structure "Gaule" dans l'implémentation en tableau

3.2.2 Structure en Liste chainée

Pour la liste chainée, une deuxième structure vient s'ajouter. La première, comme pour les tableaux, garde les informations sur la liste et la deuxième sont les structures qui correspondront chacune à une des villes avec un pointeur sur l'élément suivant et précédent de la liste.

```
1       struct Gaule_t{
2       Cellule_Gaule *premiere_cellule;
3       Cellule_Gaule *derniere_cellule;
4       int nombre_villes;
5       int est_circuit;
6       int nombre_specialites;
```

```
};

struct Cellule_Gaule_t{
Cellule_Gaule *cellule_suivante;
Cellule_Gaule *cellule_precedente;
Ville *ville;
};
```

Extrait de Code 4 – Structure "Gaule" dans l'implémentation en tableau

3.2.3 Spécification des fonctions et procédures gaule.h

```
/*
*@pre : ville1 = ville1_0 ≠ NULL ∧ ville2 = ville2_0 ≠ NULL
*@post : ville1 = ville1_0 ∧ ville2 = ville2_0 ∧ tour_init
*/
Gaule *cree_nouveau_tour(Ville *ville1, Ville *ville2);

/*
*@pre : ∅
*@post : tour = NULL
*/
void detruit_tour(Gaule *tour);

/*
*@pre : tour ≠ NULL ∧ nombre_villes > 0
*@post : get_nombre_villes(tour) = nombre_villes
*/
void set_nombre_villes(Gaule *tour, int nombre_villes);

/*
*@pre : tour ≠ NULL
*@post : get_nombre_villes = get_nombre_villes(tour)
*/
int get_nombre_villes(Gaule *tour);

/*
*@pre : tour ≠ NULL ∧ ville ≠ NULL
*@post : get_nombre_villes(ajoute_ville(tour, ville)) = get_nombre_villes(tour) + 1∧
*ville ajoutée à tour
*/
int ajoute_ville(Gaule *tour, Ville *ville);

/*
*@pre : tour ≠ NULL ∧ get_nombre_villes(tour) > 0
*@post : dernière ville retirée de tour
*∧get_nombre_villes(tour) = get_nombre_villes(tour_0) − 1
*/
void supprime_ville(Gaule *tour);

/*
*@pre : ∅
*@post : compare_string(chaine1, chaine2) = 0 si chaine1 = chaine2, −1 sinon
*/
int compare_string(char *chaine1, char *chaine2);

/*
*@pre : tour ≠ NULL
*@post : get_est_circuit(tour) = 1
```

```
48      *si premiere ville et derniere ville de tour sont les mêmes
49      */
50      void maj_est_circuit(Gaule *tour);
51
52      /*
53      *@pre : tour ≠ NULL
54      *@post : retourne tour->est\_circuit
55      */
56      int get_est_circuit(Gaule *tour);
57
58      /*
59      *@pre : tour ≠ NULL
60      *@post : retourne tour_>nombre\_specialites
61      */
62      int get_nombre_specialites(Gaule *tour);
63
64      /*
65      *@pre : tour ≠ NULL ∧ nom_ville ≠ NULL
66      *@post : ∃V1 ∈ Ville, get_nom_ville(Ville) = nom_ville ⟹ get_specialite(tour, nom_ville) =
67      *get_specialite_ville(V1)
68      */
69      char *get_specialite(Gaule *tour, char *nom_ville);
70
71      /*
72      *@pre : tour ≠ NULL ∧ nom_ville ≠ NULL
73      *@post : ville_en_double(tour, nom_ville) = 1,
74      *∃V1, V2 ∈ Ville, V1, V2 ⊂ tour ∧ get_nom_ville(V1) = get_nom_ville(V2)
75      *sinon ville_en_double(tour, nom_ville) = 0
76      */
77      int ville_en_double(Gaule *tour, char *nom_ville);
```

Extrait de Code 5 – Spécification des fonctions et procédures du header "gaule.h"

3.2.4 Schématisation

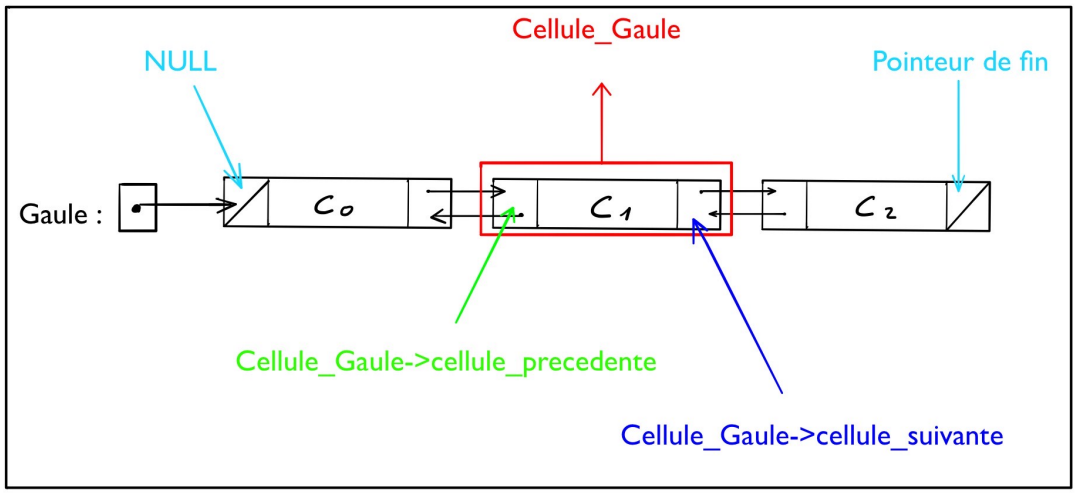

FIGURE 1 – Schéma de la strucutre sous forme de liste chainée

4 Implémentation des structures

4.1 Ville

Le header ville.h est implémenté par le ficher ville.c. Toutes les implémentations des fonctions/procédures sont très simples. En effet, la majorité d'entre elles ne sont que des accesseurs/-setteurs et pour les autres il s'agit d'opérations mathématique ou d'allocation de mémoire très simple ayant toutes une complexité d'ordre O(1).

4.2 Gaule

4.2.1 Tableau

1. accesseurs
   Comme pour ville, certaines fonctions sont uniquement des accesseurs/setteurs et donc retourne une valeur d'un champ de Gaule. C'est notamment le cas des fonctions/procédures :
   — **get_nombre_specialites**
   — **get_est_circuit**
   — **get_nombre_villes**
   — **set_nombre_villes**
2. **ajoute_ville**
   Cette fonction va s'occuper d'ajouter au tableau de villes, la ville donnée en argument en s'assurant que celle-ci n'est pas la même que la dernière du tableau. Elle va également s'occuper de mettre à jour certains champs de la structure Gaule(est_circuit, nombre_villes et nombre_specialites)
   — SP1 : Vérification que l'on ne rajoute pas la même ville que la dernière de la liste.
   — SP2 : Mise à jour du nombre de ville du tour
   — SP3 : Aggrandissement de l'espace mémoire de tableau ville et ajout de la nouvelle ville
   — SP4 : Mise à jour du nombre de spécialités et du champ de Gaule permettant de savoir si le tour est un circuit ou non
3. **supprime_ville**
   A l'inverse de ajoute_ville, cette procédure retire la dernière ville du tableau à qu'il ne soit pas vide.
   — SP1 : Mise à jour du nombre de spécialités et du nombre de villes
   — SP2 : Realloue la mémoire du tableau afin d'en retirer la dernière ville
   — SP3 : Mise à jour du champ de Gaule permettant de savoir si le tour est un circuit ou non
4. **compare_string**
   Cette fonction permet de savoir si les 2 chaines de caractères données en argument sont les mêmes. Pour ce faire, on compare tous caractères un à un de la première chaine avec ceux respectifs de la deuxième. Ce problème va donc parcourir au Maximum l'entièreté de la première chaine de caractères.
   Voici le code étant solution de ce problème, agrémenté de son invariant de boucle. Spécifions pour ce problème $\forall s \in string, \exists i \in Entier, taille(s) = i, s[i-1] = 0$

```
int i=0;
{Inv : chaine1 = chaine1_0 ∧ chaine2 = chaine2_0 ∧ 0 ≤ i ≤ length(chaine1)}
while(chaine1[i]!=0){
    {Inv ∧ B : chaine1 = chaine1_0 ∧ chaine2 = chaine2_0 ∧ 0 ≤ i < length(chaine1)}
    if (chaine2[i]==0 || (chaine1[i]!=chaine2[i]))
        return -1;
    i++;
}
{Inv ∧ ¬B ⟹ chaine1 = chaine1_0 ∧ chaine2 = chaine2_0 ∧ i = length(chaine1)}
if(chaine2[i-1]!=0)
    return -1;
return 0;
```

Extrait de Code 6 – code comparaison de 2 chaines de caractères

5. **maj_est_circuit**
   Cette procédure redéfinit le booléen du champ est_circuit de Gaule. Dans le cas où le tour contient au moins 3 villes, elle compare le nom de la première et dernière ville. Le tour est un circuit s'ils correspondent.

6. **get_specialite**
   Cette fonction retourne la specialite de la ville dont le nom a été donné en argument. Pour se faire, la fonction parcours le tableau de villes et compare leur nom un à un avec le nom donné. Si la ville est trouvée dans le tour, on utilise l'accesseur créé à cet effet pour rendre la spécialité.
   Voici le code étant solution de ce problème, agrémenté de son invariant de boucle.

```
int i = 0;
{Inv : nom_ville = nom_ville_0 ∧ 0 ≤ i ≤ get_nombre_villes(tour)}
for(; i<get_nombre_villes(tour) && ville_appartient_tour==0; i++){
    {Inv ∧ B : nom_ville = nom_ville_0∧
    0 ≤ i < get_nombre_villes(tour) ∧ ville_appartient_tour = 0}
    nom2 = get_nom_ville(tour->tableau_ville[i]);
    if(!compare_string(nom_ville, nom2)){
        ville_appartient_tour = 1;
        /*décrémente i de 1 car sera à nouveau incrémenter en fin de
        boucle malgré que la boucle s'arrête dû à
        ville_appartient_tour=1, ainsi i sera bien l'indice du tableau
        où trouver la ville après la boucle*/
        i--;
    }
}
{Inv ∧ ¬B ⟹ nom_ville = nom_ville_0∧
(i = get_nombre_villes(tour) ∨ (ville_appartient_tour = 1 ∧ 0 ≤ i < get_nombre_villes(tour)))}
```

Extrait de Code 7 – code boucle trouver ville correspondante

7. **ville_en_double**
   Cette fonction à pour objectif de définir si la ville donnée en argument apparait plus d'une fois ou non dans le tour. On parcourt alors le tableau de villes afin d'y dénombrer le nombre d'occurance de la ville en question et enfin retourner un booléen correspondant.
   Voici le code étant solution de ce problème, agrémenté de son invariant de boucle.

```
int n_apparition = 0;
{Inv : nom_ville = nom_ville_0 ∧ 0 ≤ i ≤ get_nombre_villes(tour)∧
n_apparition ≤ 2}
for(int i = 0; i<get_nombre_villes(tour) && n_apparition<2 ; i++){
{Inv ∧ B : nom_ville = nom_ville_0 ∧ 0 ≤ i < get_nombre_villes(tour)∧
```

```
6       n_apparition < 2}
7       if(compare_string(get_nom_ville(tour->tableau_ville[i]),nom_ville)==0)
8           n_apparition++;
9       }
10      {Inv ∧ ¬B ⟹ nom_ville = nom_ville_0 ∧ (i = get_nombre_villes(tour)∨
11      n_apparition = 2}
```

Extrait de Code 8 – code boucle nombre d'apparition d'une ville

4.2.2 Liste chainée

1. Accesseurs
   De même que pour l'implémentation sous forme de tableau, certaines fonctions/procédures (précisées ci-dessous) ne requierent pas d'explications supplémentaires dû à leur simplicité. Elles sont d'ailleurs également toutes de complexité O(1).
   — **set_nombre_villes**
   — **get_nombre_villes**
   — **get_est_circuit**
   — **get_nombre_specialites**
2. Allocation de mémoire
   Quatre fonctions, s'occupe de l'allocation de mémoire des structures Gaule et Cellule_Gaule.
   — **cree_Cellule_Gaule** : alloue de la mémoire pour une nouvelle cellule Cellule_Gaule. Si l'opération n'a pas échouée alors elle y intègre également les informations données en argument.
   — **detruit_liste** : Cette fonction a pour but de libérer la mémoire des cellules de la liste. Dû à son fonctionnnement récurssif, elle permet de libérer toutes les cellules depuis celle donnnée en argument jusqu'à la dernière de la liste.
   En effet : $\forall C \in Cellule\_Gaule$
   detruit_liste(C)=detruit_liste(C→cellule_suivante)+free(C)
   si C→cellule_suivante≠NULL
   sinon detruit_liste(C)=free(C)
   — **cree_nouveau_tour** : alloue de la mémoire pour un nouveau_tour. Cette fonction fait également appel à cree_Cellule_Gaule avant d'intégrer les 2 premières villes du tour données en argument à la liste nouvellement créer et donc permettre d'initialiser les champs du nouveau tour lié à cette liste.
   — **detruit_tour** : Libère la mémoire allouer à un struct Gaule donnée en argument. Elle y libère également la liste incluse à l'aide de la fonction créée à cet effet.
3. **ajoute_ville**