

INFO0947: Projet 2
Types Abstraits de Données

Groupe 10: Cyril RUSSE

Table des matières

1	Introduction	3
1.1	Contexte	3
1.2	Enoncé	3
1.2.1	Types abstraits	3
1.2.2	Représentation Concrète	3
2	Signature des TAD	3
2.1	Ville	3
2.2	Gaule	4
3	Spécifications des TAD	5
3.1	Ville	5
3.1.1	Structure	5
3.1.2	Spécification des fonctions et procédures de ville.h	5
3.2	Gaule	6
3.2.1	Structure en Tableau	6
3.2.2	Structure en Liste chaînée	6
3.2.3	Spécification des fonctions et procédures gaule.h	7
3.2.4	Schématisation	8
4	Implémentation des structures	9
4.1	Ville	9
4.2	Gaule	9
4.2.1	Tableau	9
4.2.2	Liste chaînée	11
5	Complexité	12
5.1	Gaule - Tableau	12
5.2	Gaule - Liste	13
6	Seatest	13
6.1	ajoute_ville	13
6.2	get_nombre_villes	13
7	Avantage et Inconvénients entre liste et tableau	14

1 Introduction

1.1 Contexte

En référence à la BD Astérix et Obélix, "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 avons 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 gastronomiques 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 chaînée

2 Signature des TAD

2.1 Ville

Type :

- Ville

Utilise :

- *String*
- \mathbb{R}

Opérations :

- $\text{creer_ville} : \text{String} \times \mathbb{R} \times \mathbb{R} \rightarrow \text{Ville}$
- $\text{get_x_ville} : \text{Ville} \rightarrow \mathbb{R}$
- $\text{get_y_ville} : \text{Ville} \rightarrow \mathbb{R}$

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

Préconditions :

- \emptyset

Axiomes : $\forall x, y \in \mathbb{R} \wedge \forall V_1, V_2 \in \text{Ville} \wedge \forall s, t \in \text{String}$

- $\text{distance_entre_2_villes}(V_1, V_2) = \frac{\sqrt{(\text{get_x_ville}(V_2) - \text{get_x_ville}(V_1))^2 + (\text{get_y_ville}(V_2) - \text{get_y_ville}(V_1))^2}}{2}$
- $\text{distance_entre_2_villes}(\text{set_specialite_ville}(V_1, s), \text{set_specialite_ville}(V_2, t)) = \text{distance_entre_2_villes}(V_1, V_2)$
- $\text{get_x_ville}(\text{creer_ville}(s, x, y)) = x$
- $\text{get_y_ville}(\text{creer_ville}(s, x, y)) = y$
- $\text{get_x_ville}(\text{set_specialite_ville}(V_1, s)) = \text{get_x_ville}(V_1)$
- $\text{get_y_ville}(\text{set_specialite_ville}(V_1, s)) = \text{get_y_ville}(V_1)$
- $\text{get_nom_ville}(\text{creer_ville}(s, x, y)) = s$
- $\text{get_nom_ville}(\text{set_specialite_ville}(V_1, s)) = \text{get_nom_ville}(V_1)$
- $\text{get_specialite_ville}(\text{set_specialite_ville}(V_1, s)) = s$
- $\text{get_specialite_ville}(\text{creer_ville}(s, x, y)) = \text{NULL}$

2.2 Gaule

Type :

- *Gaule*

Utilise :

- *Ville*
- *Entiers*
- *String*
- *Booleen*

Opérations :

- $\text{cree_nouveau_tour} : \text{Ville} \times \text{Ville} \rightarrow \text{Gaule}$
- $\text{get_nombre_villes} : \text{Gaule} \rightarrow \text{Entiers}$
- $\text{ajoute_ville} : \text{Gaule} \times \text{Ville} \rightarrow \text{Gaule}$
- $\text{supprime_ville} : \text{Gaule} \rightarrow \text{Gaule}$
- $\text{get_est_circuit} : \text{Gaule} \rightarrow \text{Booleen}$
- $\text{get_nombre_specialite} : \text{Gaule} \rightarrow \text{Entiers}$
- $\text{get_specialite} : \text{Gaule} \times \text{String} \rightarrow \text{String}$
- $\text{ville_en_double} : \text{Gaule} \times \text{Ville} \rightarrow \text{Booleen}$

Préconditions :

Axiomes : $\forall G \in \text{Gaule}, \forall V_1, V_2 \in \text{Ville}$

- $\text{get_nombre_villes}(\text{ajoute_ville}(G, V_1)) = \text{get_nombre_villes}(G) + 1$
- $\text{get_nombre_villes}(\text{supprime_ville}(G)) = \text{get_nombre_villes}(G) - 1$
si $\text{get_nombre_villes}(G) > 0$
sinon $\text{get_nombre_villes}(\text{supprime_ville}(G)) = \text{get_nombre_villes}(G)$
- $\text{get_nombre_villes}(\text{cree_nouveau_tour}(V_1, V_2)) = 2$
- $\text{get_nombre_specialite}(\text{ajoute_ville}(G, V_1)) = \text{get_nombre_specialite}(G) + 1$
si $\text{ville_en_double}(G, V_1) = \text{False} \wedge \text{get_specialite}(G, V_1) \neq \text{NULL}$
sinon $\text{get_nombre_specialite}(\text{ajoute_ville}(G, V_1)) = \text{get_nombre_specialite}(G)$

- $\text{get_nombre_specialite}(\text{cree_nouveau_tour}(V_1, V_2)) = 2$
 si $\text{get_specialite}(G, V_1) \neq \text{NULL} \wedge \text{get_specialite}(G, V_2) \neq \text{NULL}$
- $\text{get_nombre_specialite}(\text{cree_nouveau_tour}(V_1, V_2)) = 1$
 si $\text{get_specialite}(G, V_1) \neq \text{NULL} \wedge \text{get_specialite}(G, V_2) = \text{NULL}$
 $\vee \text{get_specialite}(G, V_2) \neq \text{NULL} \wedge \text{get_specialite}(G, V_1) = \text{NULL}$
- $\text{get_nombre_specialite}(\text{cree_nouveau_tour}(V_1, V_2)) = 0$
 si $\text{get_specialite}(G, V_1) = \text{NULL} \wedge \text{get_specialite}(G, V_2) = \text{NULL}$
- $\text{get_est_circuit}(\text{cree_nouveau_tour}(V_1, V_2)) = \text{False}$
- $\text{get_specialite}(\text{cree_nouveau_tour}(V_1, V_2), V_1) = \text{get_specialite_ville}(V_1)$
- $\text{get_specialite}(\text{cree_nouveau_tour}(V_1, V_2), V_2) = \text{get_specialite_ville}(V_2)$
- $\text{get_specialite}(\text{ajoute_ville}(G, V_1), V_1) = \text{get_specialite_ville}(V_1)$

3 Spécifications des TAD

3.1 Ville

3.1.1 Structure

```

1  struct Ville_t{
2  char *nom;
3  float x;
4  float y;
5  char *specialite;
6  };

```

Extrait de Code 1 – Structure "Ville"

3.1.2 Spécification des fonctions et procédures de ville.h

```

1  /*
2  *@pre : nom  $\neq \text{NULL}$   $\wedge x = x_0 \wedge y = y_0 \wedge \text{nom} = \text{nom}_0$ 
3  *@post : villeinit  $\wedge x = x_0 \wedge y = y_0 \wedge \text{nom} = \text{nom}_0 \wedge \text{get\_x\_ville}(\text{ville}) = \text{ville} \rightarrow x \wedge$ 
4  *get_y_ville(ville) = ville  $\rightarrow y \wedge \text{get\_nom\_ville}(\text{ville}) = \text{ville} \rightarrow \text{nom}$ 
5  */
6  Ville *creer_ville(char *nom, float x, float y);
7
8  /*
9  *@pre :  $\emptyset$ 
10 *@post : ville = NULL
11 */
12 void detruit_ville(Ville *ville);
13
14 /*
15 *@pre : ville  $\neq \text{NULL}$ 
16 *@post : ville = ville0  $\wedge \text{get\_x\_ville}(\text{ville}) = \text{ville} \rightarrow x$ 
17 */
18 float get_x_ville(Ville *ville);
19
20 /*
21 *@pre : ville  $\neq \text{NULL}$ 
22 *@post : ville = ville0  $\wedge \text{get\_y\_ville}(\text{ville}) = \text{ville} \rightarrow y$ 
23 */
24 float get_y_ville(Ville *ville);
25
26 /*

```

```

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 chaînée

Pour la liste chaînée, 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;

```

```

7   };
8
9   struct Cellule_Gaule_t{
10  Cellule_Gaule *cellule_suivante;
11  Cellule_Gaule *cellule_precedente;
12  Ville *ville;
13  };

```

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

3.2.3 Spécification des fonctions et procédures gaule.h

```

1   /*
2   *@pre : ville1 = ville1_0 ≠ NULL ∧ ville2 = ville2_0 ≠ NULL
3   *@post : ville1 = ville1_0 ∧ ville2 = ville2_0 ∧ tour_init
4   */
5   Gaule *cree_nouveau_tour(Ville *ville1, Ville *ville2);
6
7   /*
8   *@pre : ∅
9   *@post : tour = NULL
10  */
11  void detruit_tour(Gaule *tour);
12
13  /*
14  *@pre : tour ≠ NULL ∧ nombre_villes > 0
15  *@post : get_nombre_villes(tour) = nombre_villes
16  */
17  void set_nombre_villes(Gaule *tour, int nombre_villes);
18
19  /*
20  *@pre : tour ≠ NULL
21  *@post : get_nombre_villes = get_nombre_villes(tour)
22  */
23  int get_nombre_villes(Gaule *tour);
24
25  /*
26  *@pre : tour ≠ NULL ∧ ville ≠ NULL
27  *@post : get_nombre_villes(ajoute_ville(tour, ville)) = get_nombre_villes(tour) + 1 ∧
28  *ville ajoutée à tour
29  */
30  int ajoute_ville(Gaule *tour, Ville *ville);
31
32  /*
33  *@pre : tour ≠ NULL ∧ get_nombre_villes(tour) > 0
34  *@post : dernière ville retirée de tour
35  *^get_nombre_villes(tour) = get_nombre_villes(tour_0) - 1
36  */
37  void supprime_ville(Gaule *tour);
38
39  /*
40  *@pre : ∅
41  *@post : compare_string(chaine1, chaine2) = 0 si chaine1 = chaine2, -1 sinon
42  */
43  int compare_string(char *chaine1, char *chaine2);
44
45  /*
46  *@pre : tour ≠ NULL
47  *@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

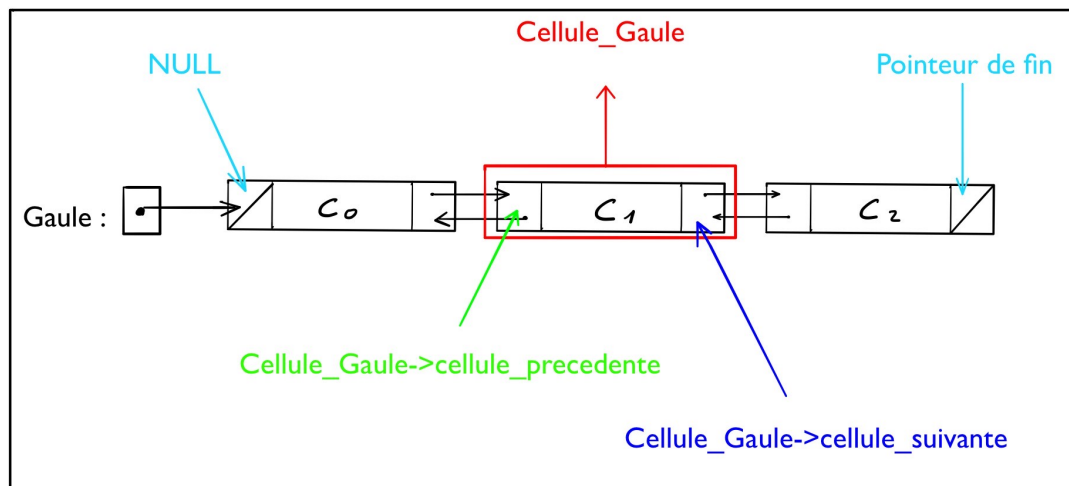


FIGURE 1 – Schéma de la strucutre sous forme de liste chaînée

4 Implémentation des structures

4.1 Ville

Le header ville.h est implémenté par le fichier 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. Allocation de mémoire

Une fonction et une procédure permettent l'allocation/libération de mémoire nécessaire à la création/destruction des structures Gaule :

- `cree_nouveau_tour` : alloue de la mémoire pour un nouveau tour. Cette fonction y crée un pointeur vers un tableau pouvant contenir deux structures Ville, les deux villes étant données en argument y sont intégrées si aucun échec d'allocation de mémoire n'a eu lieu. Le nombre de ville est initialisé à 2. Par définition, le tour ne peut pas être un circuit lorsqu'il est nouvellement créé comme le spécifie l'énoncé. Le champ `est_circuit` est donc initialement nul.
- `detruit_tour` : libère la mémoire allouée à une structure Gaule.

3. `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

4. `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

5. `compare_string`

Cette fonction permet de savoir si les 2 chaînes de caractères données en argument sont

les mêmes. Pour ce faire, on compare tous caractères un à un de la première chaîne avec ceux respectifs de la deuxième. Ce problème va donc parcourir au Maximum l'entièreté de la première chaîne de caractères.

Voici le code étant solution de ce problème, agrémenté de son invariant de boucle. Spécifions pour ce problème $\forall s \in \text{string}, \exists i \in \text{Entier}, \text{taille}(s) = i, s[i - 1] = 0$

```

1      int i=0;
2      {Inv : chaine1 = chaine10 ∧ chaine2 = chaine20 ∧ 0 ≤ i ≤ length(chaine1)}
3      while (chaine1[i] != '\0') {
4          {Inv ∧ B : chaine1 = chaine10 ∧ chaine2 = chaine20 ∧ 0 ≤ i < length(chaine1)}
5          if (chaine2[i] == '\0' || (chaine1[i] != chaine2[i]))
6              return -1;
7          i++;
8      }
9      {Inv ∧ ¬B ⇒ chaine1 = chaine10 ∧ chaine2 = chaine20 ∧ i = length(chaine1)}
10     if (chaine2[i] != '\0')
11         return -1;
12     return 0;

```

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

6. 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.

7. get_specialite

Cette fonction retourne la specialite de la ville dont le nom a été donné en argument. Pour se faire, la fonction parcourt 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'accessor créé à cet effet pour rendre la spécialité.

Voici le code étant solution de ce problème, agrémenté de son invariant de boucle.

```

1      int i = 0;
2      {Inv : nom_ville = nom_ville0 ∧ 0 ≤ i ≤ get_nombre_villes(tour)}
3      for (; i < get_nombre_villes(tour) && ville_appartient_tour == 0; i++){
4          {Inv ∧ B : nom_ville = nom_ville0 ∧
5          0 ≤ i < get_nombre_villes(tour) ∧ ville_appartient_tour = 0}
6          nom2 = get_nom_ville(tour->tableau_ville[i]);
7          if (!compare_string(nom_ville, nom2)){
8              ville_appartient_tour = 1;
9              /*décrémente i de 1 car sera à nouveau incrémenter en fin de
10             boucle malgré que la boucle s'arrête dû à
11             ville_appartient_tour=1, ainsi i sera bien l'indice du tableau
12             où trouver la ville après la boucle*/
13             i--;
14         }
15     }
16     {Inv ∧ ¬B ⇒ nom_ville = nom_ville0 ∧
17     (i = get_nombre_villes(tour) ∨ (ville_appartient_tour = 1 ∧ 0 ≤ i < get_nombre_villes(tour)))}

```

Extrait de Code 7 – code boucle trouver ville correspondante

8. ville_en_double

Cette fonction a pour objectif de définir si la ville donnée en argument apparaît plus d'une fois ou non dans le tour. On parcourt alors le tableau de villes afin d'y dénombrer le nombre d'occurrence 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.

```

1      int n_apparition = 0;
2      {Inv : nom_ville = nom_ville0 ∧ 0 ≤ i ≤ get_nombre_villes(tour) ∧
3      n_apparition ≤ 2}
4      for(int i = 0; i < get_nombre_villes(tour) && n_apparition < 2 ; i++){
5      {Inv ∧ B : nom_ville = nom_ville0 ∧ 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_ville0 ∧ (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 chaînée

1. Accesseurs

De même que pour l'implémentation sous forme de tableau, certaines fonctions/procédures (précisées ci-dessous) ne requièrent 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.
- **detrui_liste** : Cette fonction a pour but de libérer la mémoire des cellules de la liste. Dû à son fonctionnement récursif, elle permet de libérer toutes les cellules depuis celle donnée en argument jusqu'à la dernière de la liste.
En effet : $\forall C \in \text{Cellule_Gaule}$
 $\text{detrui_liste}(C) = \text{detrui_liste}(C \rightarrow \text{cellule_suivante}) + \text{free}(C)$
si $C \rightarrow \text{cellule_suivante} \neq \text{NULL}$
sinon $\text{detrui_liste}(C) = \text{free}(C)$
- **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.
- **detrui_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

Cette fonction va ajouter une nouvelle cellule à la fin de la liste de cellule contenue dans le tour et y mettre à jour certaines champ de tour.

- SP1 : Vérification que l'on ne rajoute pas la même ville que la dernière de la liste.
- SP2 : Incrémentation de 1 le champ contenant le nombre de villes du tour
- SP3 : Création d'une nouvelle cellule contenant la nouvelle ville
- SP4 : pointeur de cellule suivante de la cellule précédant la nouvelle cellule et pointeur de dernière cellule du tour mis à jour

- SP5 : Mise à jour du nombre de spécialités et du champ de Gaule permettant de savoir si le tour est un circuit ou non
4. **supprime_ville**
 La dernière cellule de la liste chaînée est retirée et libérée à l'aide de cette procédure. Elle met également à jour les champs du tour nécessitant un changement.
- SP1 : Mise à jour du nombre de spécialités et du nombre de villes
 - SP2 : Libère la dernière cellule et met à jour le pointeur de dernière cellule du tour sur la cellule précédant l'ancienne dernière.
 - SP3 : Mise à jour du champ de Gaule permettant de savoir si le tour est un circuit ou non
5. **maj_est_circuit**
 Cette procédure vérifie, dans le cas où le tour est constitué d'au moins 3 villes, la correspondance des noms de la première et dernière ville afin de renvoyer un booléen correspondant à cette vérification.
6. **get_specialite**
 Cette fonction renvoie la spécialité de la ville dont le nom est donné en argument.
- SP1 : Chercher la ville parmi la liste contenue dans le tour
 - SP2 : Renvoyer le nom de la specialite de la ville si elle est trouvée
- SP1 \implies SP2 Le SP1 implique l'utilisation d'une boucle où l'on compare le nom de la ville de la cellule actuelle avec le nom donné. Tant qu'ils ne correspondent pas, notre cellule devient la cellule suivante. Notre invariant se traduirait alors comme ceci
Inv : $1 \leq i \leq \text{get_nombre_villes}(\text{tour}) + 1$ avec *i* correspondant à la *i*ème cellule de la liste.
7. **ville_en_double**
 Cette fonction a pour objectif de définir si la ville donnée en argument apparaît plus d'une fois ou non dans le tour. On parcourt alors le tableau de villes afin d'y dénombrer le nombre d'occurrence de la ville en question et enfin retourner un booléen correspondant. Pour se faire, on parcourt la liste par chaque cellule en partant de la première jusqu'à la fin ou jusqu'à avoir au moins trouver deux cellules dont le nom de la ville correspond au nom donné en argument. Elle renvoie alors le booléen correspondant au fait que la ville s'y trouve au moins deux fois ou non.
8. **compare_string** Cette fonction correspond parfaitement à la fonction de l'implémentation sous forme de tableau. Son implémentation est donc détaillée dans la section 4.2.1 point 4 et également dans l'Extrait de Code 6 s'y trouvant.

5 Complexité

Comme il l'a été précisé dans la section sur l'implémentation de Ville, la complexité des fonctions/procédures de celles-ci sont toutes majorées par $O(1)$. Nous allons dès lors nous intéresser uniquement à celle de Gaule.

5.1 Gaule - Tableau

- Les fonctions/procédures `set_nombre_villes`, `get_nombre_villes`, `get_est_circuit` et `get_nombre_specialites` sont toutes composées d'accesseurs/setteurs donc leurs complexités sont majorées par $O(1)$. Il en va de même pour `cree_nouveau_tour`, `detrui_tour` et `supprime_ville` qui n'appliquent que des opérations simples.

- `ajoute_ville` : est composée d'opérations simples, appelle des fonctions de complexité majorée par $O(1)$ et fait appel à une fonction pour comparer deux chaînes de caractères. Elle parcourt donc au plus la taille, de la chaîne de caractère. Fonction donc majorée par $O(N)$ L'addition de ces complexités nous amène à en conclure une complexité linéaire.
- `compare_string` : si la première chaîne donnée en argument fait une taille i alors la fonction fera au maximum i opérations pour ce qui est du parcourt de la chaîne de caractères, incrémenter des quelques opérations simples nous pouvons en conclure une complexité majorée par $O(N)$.
- `maj_est_circuit` : est constituée d'opérations simples et fait appelle à une fonction de complexité majorée par $O(N)$, l'addition des différentes complexités amène à une complexité majorée par $O(N)$.
- `get_specialite` : nous avons à nouveau notre fonction dont la complexité est majorée par $O(N)$ mais elle est cette fois-ci intégrée dans un autre sous problème qui va effectuer au plus `get_nombre_villes` itérations. Ce sous-problème est donc lui aussi de complexité linéaire ce qui est implique une complexité linéaire intégré dans une autre, ce qui implique une complexité majorée par $O(N^2)$. L'addition des quelques opérations simples n'ont pas d'impact sur la complexité qui reste quadratique.
- `ville_en_double` : idem que pour `get_specialite`

5.2 Gaule - Liste

Les seules fonctions pour lesquelles la complexité aurait pu être affectée par l'implémentation sous forme de liste aurait été dans le cas d'ajout/retrait de ville pouvant se faire au milieu du tour, hors comme mes fonctions ne permettent que de le faire pour la dernière ville du tour, la totalité des fonctions/procédures ont la même complexité à l'exception des fonctions propres à la version liste analysée ci-dessous.

- `cree_Cellule_Gaule` : est aussi une fonction composée d'opérations simples donc de complexité majorée par $O(1)$.
- `detrui_liste` : est une fonction récursive pouvant dans le pire des cas faire appel à elle même au maximum le nombre de cellule de la liste, donc une complexité linéaire.

6 Seatest

6.1 ajoute_ville

Afin de vérifier le bon fonctionnement de la fonction "`ajoute_ville`" étant donné qu'il n'était pas demandé de faire une fonction permettant d'accéder à une ville du tour, j'ai utilisé la fonction `get_specialite`. En effet, elle permet d'accéder au champ spécialité d'une ville du tour donnée en argument. Ainsi, ce test permet donc de vérifier que lorsque l'on a ajouté une ville, si nous avons accès à un des champs, ici spécialité, alors la ville a bien été ajoutée au tour. Nous vérifions également que si la même ville que la première est ajoutée alors le tour est considéré comme un circuit, prouvant une nouvelle fois le bon fonctionnement de `ajoute_ville`.

6.2 get_nombre_villes

Ce test va vérifier différents cas de nombre de ville du tour et vérifier que la fonction renvoie le nombre exact. Nous allons d'abord vérifier que la fonction renvoie bien la valeur 2 dans le cas où nous venons de créer un tour. Enfin, nous ajoutons une ville puis la supprimons et des tests sont faits vérifiant successivement que le nombre de villes est bien incrémenté puis décrémenté d'une unité.

7 **Avantage et Inconvénients entre liste et tableau**

Nous pouvons constater des différences notables entre les deux implémentations.

Premièrement, il est nettement plus facile d'accéder à un élément du tableau, qui se fait de manière directe, comparé à la liste qui nécessite dans tous les cas de parcourir toute la liste jusqu'à l'élément recherché.

Par ailleurs, la liste a pour net avantage la gestion de ses cellules qui permet l'ajout et le retrait d'une ville très facilement comparé au tableau qui, pour se faire, devra impliquer une réallocation de mémoire, telle est la technique que j'ai décidé d'adopter, ou la recréation complète d'un nouveau tableau aux nouvelles dimensions. Dans mon implémentation, j'ai fixé comme règle que l'ajout ou le retrait ne se fait uniquement à la dernière position dans les deux cas, ce qui rend la liste d'autant plus pratique dès lors que je garde en mémoire l'adresse de sa dernière cellule. Si j'avais accordé à l'utilisateur de donner l'indice de l'endroit où ajouter/retirer un cellule, alors il serait paru comme inconvénients pour la liste de devoir parcourir cette dernière jusqu'à la cellule se trouvant à cet indice.

Enfin, notons une très légère augmentation de l'espace mémoire nécessaire à une liste dont chacune de ses cellules contient deux pointeurs supplémentaires. Cette différence ne serait qu'un véritable inconvénients qu'en cas de liste d'une taille très importante.