INFO0947: Récursivité et ypes abstraits de données

Groupe s190632: Luca Matagne,

Table des matières

1 Signature et sémantique des TAD

1.1 Escale

Type :
- Escale

Utilise :
- float
- char

Opérations :
- creation : $float \times float \times char \rightarrow Escale$
- set_time : $Escale \times float \rightarrow Escale$
- get_time : $Escale \rightarrow float$
- get : $Escale \rightarrow Escale$
- calc_distance : $Escale \times Escale \rightarrow float$

Précondtions :
- str $\neq$ NULL

Axiomes : $\forall A, B \in Escale \wedge \forall x, y \in Real^{+} \wedge str \in char$
- get_time(set_time(A, x)) $= x$
- get(creation(x, y, str)) $= x, y, str$
- calc_distance(A, B) $= \arccos sin(B \rightarrow x) * sin(A \rightarrow x) + cos(B \rightarrow x) * cos(A \rightarrow x)$

1.2 Course

Type :
- Course

Utilise :
- Escale
- Course
- int
- char
- float

Opérations :
- create : $Escale \times Escale \rightarrow Course$
- is_a_loop : $Course \rightarrow char$
- how_many_escale : $Course \rightarrow int$
- how_many_step : $Course \rightarrow int$
- best_time_race : $Course \rightarrow float$
- best_time_step : $Course \times Escalee \rightarrow float$
- add_step : $Course \times int \times Escale \rightarrow Course$
- remove_step : $Course \times int \rightarrow Course$

Précondtions : $\forall A, B \in Escale \wedge \forall i \in Real$
- create(A, B) is defined iff $A \neq NULL \wedge B \neq NULL$
- add_step(A, i) is defined iff $0 \leq i$
- remove_step(A, i) is defined iff $0 \leq i$

Axiomes : $\forall A, B, C \in Escale \wedge \forall i \in Real \wedge \forall X \in Course$
- best_time_race(create(A, B)) = best_time_step(A) + best_time_step(B)
- remove_step(add_step(X, i, A), A) = X
- add_step(remove_step(X, A), i, A) = X
- how_many_escale(create(A, B)) = 2

- how_many_escale(add_step(X, i, A)) = how_many_escale(X) + 1
- how_many_escale(remove_step(X, i, A)) = how_many_escale(X) - 1
- how_many_step(create(A, B)) = how_many_escale(create(A, B)) - 1
- is_a_loop(add_step(create(A, B), 2, A)) = 'oui'

2 TAD Escale

2.1 Spécifications des fonctions et procédures

```
/*
 * @pre: x != NULL,  y != NULL, name != NULL
 * @post: get(creation) = step=(name, x, y)
 *
 */
Escale *creation(float x, float y, char *name);
/*
 * @pre: step != NULL, time != NULL}
 * @post: step=(x, y, name, time)
 */
Escale *set_time(Escale *step, float time);
/*
 * @pre: step != NULL
 * @post: get_time=time
 */
float get_time(Escale *step);
/*
 * @pre: step != NULL
 * @post: get(step) = step
 */
Escale *get(Escale *step);
/*
 * @pre: step_depart != NULL, step != NULL
 * @post: distance = acos((sin(step_depart->x)*sin(step->x))+(cos(step_depart->y)*cos(step->y
    step->distance =arccos * 6371
 */
float calc_distance(Escale *step_depart, Escale *step);
```

2.2 Structure

Une escale est composée de ses coordonnées et de son nom (donnés à la création) mais elle peut aussi contenir, par ajour ultérieur, un meilleur temps et la distance avec l'étape précdente :

```
struct Escale_t {

    float x;
    float y;
    float time;
    char *name;
    float distance;
};
```

2.3 Gestion de la structure

L'implémentation des fonctions gérant ce TAD consiste à appliquer des opérations mathématiques ou effectuer un simple retour de champ. Elles ne seront donc pas plus détailées.

3 TAD Course

3.1 Spécifications des fonctions et procédures

```
/*
 * @pre: step1 != NULL, step2 != NULL
 * @post: (get(create) = race = (step1, step2)
 *
 */
Course *create(Escale *step1, Escale *step2);
/*
 * @pre: race != NULL
 * @post: is_a_loop = 'oui' OU 'non'
 */
char *is_a_loop(Course *race);
/*
 * @pre: race != NULL
 * @post: how_many_escale = 2 + as much time as there is a call of add_step
 */
int how_many_escale(Course *race);
/*
 * @pre: race != NULL
 * @post: how_many_escale = 2 + as much time as there is a call of add_step - 1
 */
int how_many_step(Course *race);
/*
 * @pre: race != NULL
 * @post: best_time_race = the sum of every best time step by step
 */
float best_time_race(Course *race);
/*
 * @pre: race != NULL, step != NULL
 * @post: best_time_step = step<-time
 */
float best_time_step(Course *race, Escale *step);
/*
 * @pre: race != NULL, newStep != NULL, 0 <= i <= how_many_step
 * @post: race[i] = newStep, how_many_escale += 1
 */
Course *add_step(Course *race, int i, Escale *newStep);
/*
 * @pre: race != NULL, 0 <= i <= how_many_step
 * @post: how_many_escale -= 1, race[i] = race(before function)[i+1]
 */
 Course *remove_step(Course *race, int i);
```

3.2 Structure en tableau

3.2.1 Présentation

Cette structure représente une course sous le format d'un tableau d'escales de taile array_size et de trois informations supplémentaires à savoir : le nombre d'étapes de la course, le meilleur temps pour parcourir l'intégralité de la course et le fait (ou non) que la course soit un circuit (cf. énoncé). Voici donc la structure d'un e course :

```
struct Course_t {

int array_size;// current size (i.e., number of squares in the array)
int length;// number of element recorded in the array
```

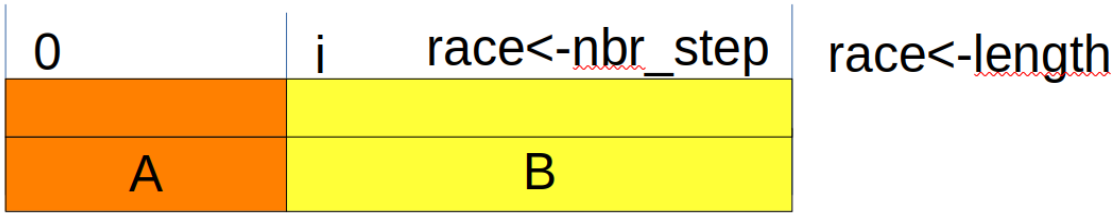
```
Escale **step; //the array itself
char *circuit;
int nbr_step;
float best_time_race;

};
```

3.2.2 Gestion de la structure en tableau

Etant donné la simplicité d'accès aux information contenues dans un tableau et grâce à la présence d'informations comme 'length' ou encore 'nbr_step', les observateurs de cette structure ont une implémentation assez triviale. Il faut dans la majorité des cas vérifier les préconditions et retourner le champ de la structure visé(avec parfois une opération arithmétique de base comme '-1'par exemple). Il est cependant important de noter que pour les deux fonctions permettant de retourner un temps (d'une étape ou de la course entière), nous avons une boucle. Ces deux boucles étant sensiblement les mêmes voici l'invariant unique qui les fait tourner toutes les deux :

3.2.3 Schématisation



Pour ce qui est de la fonction 'best_time_race', la zone A (orange) correspond aux étapes de la course pour lesquelles le temps à déjà été additionné et stocké. La partie B (jaune), quant à elle, contient le reste des étapes de la course pour lesquelles nous devons encore récupérer le temps stocker.

La fonction 'best_time_step' est bien plus simple et ne consiste qu'à parcourir le tableau afin de trouver l'étape voulue et d'en retourner le champ 'time'. La partie A (orange) de l'invariant contient alors la portion de la course déjà parcourue et la zone B (jaune), la partie de la course encore à parcourir.

3.3 Structure basée sur une liste chaînée

3.3.1 Présentation

La structure qui suit présente une course stockée dans une liste. Chaque cellule de la liste contient une escale et un pointeur vers la cellule suivante. contrairement à l'implémentation par tableau, les données supplémentaires (de type autre que 'Escale') ne sont pas stockées dans la liste mais sont calculées lorsque cela est nécessaire dans leur fonction respective. Voici à quoi ressemble notre structure :

```
struct Course_t{
    Escale *step;
    struct Course_t *next;
};
```

3.3.2 Gestion de la structure en liste

Il est intéressant de voir que le principe des observateurs de temps reste globalement le même. Les observateurs servant à compter escales et étapes, eux changent un peu étant donné que ces informations ne sont plus stockées dans la structure même. En effet, pour ces deux fonctions ci, nous devons maintenant parcours la liste entièrement afin d'en calculer la longueur ce qui nous permettra de déduire le nombre d'étapes et d'escales.

Les opérations internes, elles, feront l'objet d'une analyse plus approfondie.

3.3.3 Implémentation

create La fonction de création de notre fonction consiste à invoquer deux fois (car notre course est créée aces deux escales à la base) la fonction static 'create_cell'. Cette fonction statique nous permet d'allouer dynamiquement de la mémoire cellule par cellule et de placer l'étape choisie(en argument) dans la partie "riche" [1] de la cellule (la partie "pauvre" renfermant un pointeur sur NULL pour l'instant). Lors du deuxième appel de cette fonction, il faut noter que la partie "riche" de la cellule en création va être pointée par la partie "pauvre" de la cellule qui la précède dans le but de respecter le fonctionnement de la liste.

add_step En ce qui concerne l'ajout d'une étape à la course, il faut distinguer deux cas. Soit on ajoute une étape en début de course, soit n'importe où ailleurs dans la liste.

Dans le premier cas, le travail est simple. Il nous suffit de créeer une nouvelle cellule (comme vu ci-dessus) et d'attribuer à la partie "pauvre" de cette cellule, un pointeur vers le reste de la course qui de ce fait se retrouvera après l'étape nouvellement créée.

Dans le cas plus général, malheureusement, il va falloir jouer avec les pointeurs de la liste. Il nous faut un deux pointeurs initialisés sur deux cellules consécutives. Nous allons faire avancer ces deux pointeurs le long de la course. Une fois que les pointeurs pointeront vers les deux cellules qui entoureront tout prochainement notre nouvelle cellule, nous pouvons appeler la fonction 'create_cell'. La nouvelle cellule étant maintenant créée, il faut la connecter à la course. Pour cela nous allons mettre le premier de nos deux pointeurs (celui pointant vers la cellule suivante) dans la partie "pauvre de notre nouvelles cellule) et nous allons placer dans la partie "pauvre" de la cellule précédente (grâce au second pointeur créé au début de la démarche) vers notre nouvelle cellule. Nous voila avec une course bien plus complète !

4 Tests unitaires

4.1 best_time_race

Soient 'race' une course, 'step1' et 'step2' deux escales.

Nous devons vérifier que l'appel de la fonction best_time_race applique bien les 2 propriètes suivantes :

1. Le temps de la course est supérieur à 0.
2. Le temps de la course correspond à la somme du temps de chacune des étapes.

4.2 add_step

Soient 'race1', 'race2' deux courses, 'step1', 'step2' et 'step3' trois escales.

Il faut vérifier que la course renvoyer après ajout d'une nouvelle étape contient bien cette nouvelle étape.

1. La partie "riche" contient les données utiles et la partie "pauvre" contient le pointeur vers la cellule suivante

5 Comparaison entre le tableau et la liste

On peut remarquer assez vite que, trois différences :

1. Dans la liste, le fait d'aller rechercher les données telles que le temps et la longueur nous font perdre en efficacité.
2. La manipulation de pointeurs de la liste nous permet de ne pas jouer avec l'allocation dynamique et donc d'éviter tous problème de décalage à gauche ou à droite.
3. Le tableau peut parfois occuper plus de place qu'il n'en faut en mémoire.