INFO0947 : TAD

Groupe 02 : Simon Lorent, Corentin Jemine

Avril/Mai 2015

1 Introduction

Dans ce troisième projet nous avons définit deux structures de données, List et Array, pour lesquelles nous avons implémenté des fonctions de base. Ces fonctions sont des constructeurs, des observateurs ou des transformateurs. Nous nous pencherons sur une définition théorique de ces structures ainsi que leurs avantages et inconvénients respectifs.

2 Définition du type abstrait

2.1 Signature

Type :
Multi [1]
Utilise :
Natural, Boolean, Element [2]
Opérations :
create_empty : → Multi
is_empty : Multi → Boolean
count : Multi → Natural
occurrences : Element x Multi → Natural
part_of : Element x Multi → Boolean
equals : Multi x Multi → Boolean
join : Multi x Multi → Multi
add_to : Element x Multi → Multi
remove_from : Element x Multi → Multi

2.2 Sémantique

Préconditions :
Aucune [3]
Axiomes :
Notations : #m désigne le nombre d'Elements dans m
m[i] désigne le i-ème Element de m
Remarque : le signe d'égalité entre 2 Multis suit la définition de la fonction equals()

∀ m, m' ∈ Multi, ∀ e ∈ Element :

is_empty(create_empty()) = True

count(m) = #m

$$\text{occurrences(e, m)} = \sum_{i=1}^{count(m)} (\text{m[i]} == \text{e})$$

part_of(e, m) = (occurrences(e, m) > 0)

$$\text{equals(m, m')} = (\text{count(m)} == \text{count(m')}) \;\&\& \prod_{i=1}^{count(m)} ((\text{occurrences(m[i], m)} == \text{occurrences(m[i], m')})$$

1. Multi désigne soit le type List, soit le type Array
2. Element désigne une type générique
3. remove_from est définit sur un Multi vide mais n'aura aucun effet

occurrences(e, join(m, m')) = occurrences(e, m) + occurrences(e, m')

add_to(e, m) = join(m, add_to(e, create_empty()))

count(add_to(e, m)) = count(m) + 1

is_empty(add_to(e, m)) = False

Si part_of(e, m) **alors** m = remove_from(e, add_to(e, m))

Si ¬ part_of(e, m) **alors** equals(m, remove_from(e, m))

2.3 Jusitification des axiomes

TODO

3 Description des structures

Multi est une structure de donnée de type multi-ensemble générique : elle peut contenir un ensemble de n'importe quelle données et peut contenir plusieurs fois une même donnée. Dans notre projet nous avons utilisé des pointeurs sur `void` pour être conformes à la généricité. Notre structure Multi ne tient pas compte de l'ordre des éléments : des ensembles sont considérés égaux s'ils présentent tous deux les mêmes éléments le même nombre de fois. Elle est aussi implémentée afin de ne pas retourner d'erreur, si par exemple un utilisateur tente d'utiliser `remove_from()` sur un ensemble qui ne contient pas l'élément spécifié, alors rien ne se passe.

3.1 Array

La structure Array contient deux champs : un tableau de type `void*` qui va contenir l'entièrté des éléments ajoutés à l'Array correspondant ainsi que la taille de ce tableau sous forme d'un entier. Ces champs sont étroitement liés car la taille réelle du tableau doit toujours correspondre à la taille indiquée dans la structure.

3.2 List

indique une description rapide de ta structure

4 Avantages et inconvénients

Comparons les deux types. La taille d'un Array est accessible directement dans la structure alors que la taille d'une List nécéssite un parcours complet de la List afin d'être obtenue. Par contre, redimmensionner une List est rapide car il suffit de créer ou de supprimmer une seule cellule alors que pour chaque ajout ou retrait d'un élément dans un Array, un nouveau tableau est alloué afin de convenir à la nouvelle taille. C'est un processus coûteux aussi bien en termes de temps d'exécution qu'en termes de mémoire car lors de l'allocation de ce nouveau tableau, le tableau courant reste présent en mémoire afin de pouvoir être recopié. C'est pendant ce procédé uniquement qu'un Array va occuper la même taille [4] qu'une List contenant des éléments identiques, le reste du temps un Array occupera la moitié de l'espace mémoire de cette List. En effet, chaque cellule d'une liste chaînée va contenir un pointeur vers la cellule suivante en plus de la donnée fournie par l'utilisateur soit le double de mémoire nécessaire pour stocker un seul élément par rapport à un tableau.

4. Si on néglique le champ indiquant la taille du tableau

5 Fonctions

Lors du calcul de complexité algorithmique, nous avons omis tous les `assert()` présents dans le code pour deux raisons : d'abord parce que les assertions ne sont souvent que la traduction en code des préconditions (redondance) et ensuite parce que la complexité de `assert()` est de complexité $O(1)$ donc négligeable en présence d'autres lignes de code.

5.1 Array

5.1.1 create_empty

```
/*
* @pre: \
* @post: array, #array = 0
*/
Array *create_empty(void) {
    Array *array = malloc(sizeof(Array));        O(1)
    array->count = 0;                            O(1)
    return array;                                O(1)
}
```

Complexité : $O(1)$ - constante

5.1.2 is_empty

```
/*
* @pre: array initialisé
* @post: (#array == 0), array = array_0
*/
bool is_empty(Array *array) {
    return !array->count;                        O(1)
}
```

Complexité : $O(1)$ - constante

5.1.3 count

```
/*
* @pre: array initialisé
* @post: #array, array = array_0
*/
int count(Array *array) {
    return array->count;                         O(1)
}
```

Complexité : $O(1)$ - constante

5.1.4 occurrences

```
/*
* @pre: array initialisé
* @post: \sum_{i=1}^{#array} (array[i] == element), array = array_0,
*       element = element_0
*/
int occurrences(void* element, Array *array, bool(*compare)
(const void *, const void *)) {
    int occurrences = 0;                                    O(1)
    for (int i = 0; i < array->count; i++)                  O(n)
        if (compare(element, array->elements[i]))           O(1)
            occurrences++;                                  O(1)
    return occurrences;                                     O(1)
}
```

Complexité : $O(n)$ - linéaire

5.1.5 part_of

```
/*
* @pre: array initialisé
* @post: (∃i, 0 ≤ i ≤ #m:(array[i] == element)), array = array_0,
*       element = element_0
*/
bool part_of(void* element, Array *array, bool(*compare)
(const void *, const void *)) {
    for (int i = 0; i < array->count; i++)                  O(n)
        if (compare(element, array->elements[i]))           O(1)
            return TRUE;                                    O(1)
    return FALSE;                                           O(1)
}
```

Complexité : $O(n)$ - linéaire

5.2 List

bonne merde