INFO0947 : TAD

Groupe 02 : Simon Lorent, Corentin Jemine

Avril/Mai 2015

1 Introduction

Dans ce troisième projet nous avons défini 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 sur 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 défini sur un Multi vide mais n'aura aucun effet dans ce cas

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 donnée 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'entièreté des éléments ajoutés à l'Array 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

La structure list contient deux champs :
- un pointeur de type `void*` qui permet de retenir les données voulues,
- un pointeur de type list qui permet d'obtenir l'adresse de la cellule suivante lors du parcours de la liste.

3.2.1 Ajout d'une notation

Si on a la liste $L = (e_0, e_1, ..., e_{n-1})$ alors on écrit $e_i(data)$ pour représenter les données présentes dans la cellule i de la liste.

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, redimensionner une List est rapide car il suffit de créer ou de supprimer 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 même 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 `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=0}^{#array-1} (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 < #array:(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

5.1.6 equals

```
/*
* @pre: array1 initialisé, array2 initialisé
* @post: ((#array1 == #array2) &&
*      (∀i, 0 ≤ i < #m : (∑_{j=0}^{#array1-1} (array1[j] == array1[i]) ==
*      ∑_{j=0}^{#array2-1} (array2[j] == array2[i]))),
*      array1 = array1_0, array2 = array2_0
*/
bool equals(Array *array1, Array *array2, bool(*compare)
(const void *, const void *)) {
   if (array1->count != array2->count)          O(1)
      return FALSE;                             O(1)
   for (int i = 0; i < array1->count; i++)      O(n)
      if (occurrences(array1->elements[i], array1, compare) != \
      occurrences(array1->elements[i], array2, compare)) O(n)
         return FALSE;                          O(1)
   return TRUE;                                 O(1)
}
```

Complexité : $O(n^2)$ - quadratique

5.1.7 join

```
/*
* @pre: array1 initialisé, array2 initialisé
* @post: r_array = array1 ∪ array2, #r_array = #array1 + #array2,
*      array1 = array1_0, array2 = array2_0
*/
Array *join(Array *array1, Array *array2) {
   Array* r_array = create_empty();             O(1)
   r_array->count = array1->count + array2->count; O(1)
   r_array->elements = malloc(sizeof(void*)*(r_array->count)); O(1)
   for (int i = 0; i < array1->count; i++)      O(n)
      r_array->elements[i] = array1->elements[i]; O(1)
   for (int i = 0; i < array2->count; i++)      O(n)
      r_array->elements[array1->count+i] = array2->elements[i];O(1)
   return r_array;                              O(1)
}
```

Complexité : $O(n)$ - linéaire

5.1.8 add_to

```
/*
* @pre: array initialisé
* @post: r_array = array ∪ element,
*       array = array_0, element = element_0
*/
Array *add_to(void* element, Array *array) {
    Array *r_array = create_empty();                    O(1)
    r_array->count = array->count + 1;                  O(1)
    r_array->elements = malloc(sizeof(void*)*(array->count+1));O(1)
    for (int i = 0; i < array->count; i++)              O(n)
        r_array->elements[i] = array->elements[i];O(1)
    r_array->elements[array->count] = element;          O(1)
    if (array->count)                                   O(1)
        free(array->elements);                          O(1)
    return r_array;                                     O(1)
}
```

Complexité : $O(n)$ - linéaire

5.1.9 remove_from

```
/*
* @pre: array initialisé
* @post: array = r_array ∪
*       (element * (∃i, 0 ≤ i < #array:(array[i] == element)),
*       array = array_0, element = element_0
*/
Array *remove_from(void* element, Array *array, bool(*compare)
(const void *, const void *)) {
    if (!array->count)                                  O(1)
        return array;                                   O(1)
    Array *r_array = create_empty();                    O(1)
    if (occurrences(element, array, compare)) {         O(n)
        r_array->count = array->count - 1;              O(1)
        r_array->elements=malloc(sizeof(void*)*(array->count-1));O(1)
        bool removed = FALSE;                           O(1)
        for (int i = 0; i < r_array->count; i++){       O(n)
            if (!compare(array->elements[i], element) || removed)O(1)
                r_array->elements[i] = array->elements[i];O(1)
            else
                removed = TRUE;                         O(1)
        }
    }
    else return array;                                  O(1)
    return r_array;                                     O(1)
}
```

Complexité : $O(n)$ - linéaire

5.2 List

5.2.1 create_empty

```
/*
* @pre: \
* @post: list , L = ()
*/

list *create_empty(void)
{
      return NULL;                                    O(1)
}
```

Complexité constante

5.2.2 is_empty

```
/*
* @pre: \
* @post: return TRUE if L = () else return FALSE
*/

bool is_empty(list *L)
{
      if(!L)                                          O(1)
         {
            return TRUE;                              O(1)
         }
         else                                         O(1)
         {
            return FALSE;                             O(1)
         }
}
```

Complexité constante

5.2.3 count

```
/*
* @pre: \
* @post: return long(L)
*/

int count(list *L)
   {
         if(is_empty(L))                             O(1)
```

```
        {
            return 0;                          O(1)
        }
        cell *current = L;                     O(1)
        int element_count = 0;                 O(1)
        while(current)                         O(length(L))
        {
            ++element_count;                   O(1)
            current = current->next;           O(1)
        }
        return element_count;                  O(1)
    }
```

Complexité linéaire.

5.2.4 occurrences

```
/*
* @pre: element != NULL  compare is
a fonction to compare element in the list
* @post: occurrences = #{data|data = element}
*/

int occurrences(list *L,
                void *element,
                bool (*compare)(const void *,
                const void *))
{
    if(is_empty(L))                            O(1)
        return 0;                              O(1)
    cell *current = L;                         O(1)
    int occurrences = 0;                       O(1)
    while(current)                             O(lenght(L))
        {
            if(compare(current->data, element)) O(1)
                ++occurrences;                 O(1)
            current = current->next;           O(1)
        }
    return occurrences;                        O(1)
}
```

Complexité linéaire.

5.2.5 part_of

```
/*
* @pre: L = {e_0, ..., e_n} ∧ compare is a fonction to compare element
* @post: Return 1 if ∃i | e_i(data) = element else return 0
```

```
*/

bool part_of(list *L,
             void *element,
             bool (*compare)(const void *, const void *))
{
       assert(element);                              O(1)
       if(is_empty(L))                               O(1)
          return FALSE;                              O(1)
       cell *current = L;                            O(1)
       while(current)                                O(lenght(L))
       {
          if(compare(current->data, element))        O(1)
             return TRUE;                            O(1)
          current = current->next;                   O(1)
       }
   return FALSE;                                     O(1)
}
```

Complexité linéaire.

5.2.6 equals

```
/*
* @pre: compare is a fonction to compare element
∧L1 = (e0, ..., ei−1) ∧ L2 = (a0, ..., aj−1)
* @post: return 1 if i = j ∧ ∀i, occurences(ei, L1) = occurences(ei, L2)
else return 0
*/

bool equals(list *L1, list *L2,
             bool (*compare)(const void *, const void *))
{
       cell *current = L1;                           O(1)
       if(count(L1) != count(L2))                    O(lenght(L1) + length(L2))
       {
          return FALSE;                              O(1)
       }
       else
       {
        while(current)                               O(lenght(L1))
          {
             if(occurrences(L1, current->data, compare) !=
             occurrences(L2, current->data, compare))
                                                     O(length(L1) + lenght(L2))
                return FALSE;                        O(1)
             current = current->next;                O(1)
          }
```

```
        }
    return TRUE;                                    O(1)
}
```

Complexité quadratique.

5.2.7 join

```
/*
* @pre : L_1 = (e_0, ...e_{i-1}), i ≥ 0 ∧ L_2 = (a_0, ..., a_{j-1}), j ≥ 0
* @post : joined_list = L_1||L_2
*/

list *join(list *L1, list *L2)
{
    if(is_empty(L1))                                O(1)
        return L2;                                  O(1)
    else if(is_empty(L2))                           O(1)
        return L1;                                  O(1)

    list *joined_list = create_empty();             O(1)
    cell *current = L1;                             O(1)
    while(current)                                  O(lenght(L1)
    {
        joined_list = add_to(joined_list, current->data); O(1)
        current = current->next;                    O(1)
    }
    current = L2;                                   O(1)
    while(current)                                  O(lenght(L2))
    {
        joined_list = add_to(joined_list, current->data); O(1)
        current = current->next;                    O(1)
    }
    return joined_list;                             O(1)
}
```

Complexité O(lenght(L1)+lenght(L2)), complexité linéaire.

5.2.8 add_to

```
/*
* @pre : element = e_k ∧L = (e_0, ..., e_{i-1})
* @post : L = (e_0, ..., ei - 1, e_k)
*/

list *add_to(list *L, void *element)
{
    assert(element);                                O(1)
```

```
        cell *current = L;                          O(1)
        if(is_empty(L))                             O(1)
        {
            L = malloc(sizeof(cell));               O(1)
            assert(L);                              O(1)
            L->next = NULL;                         O(1)
            L->data = element;                      O(1)
            return L;                               O(1)
        }
        else
        {
            while(current->next)                    O(length(L)
                {
                    current = current->next;        O(1)
                }
            current->next = malloc(sizeof(cell));   O(1)
            assert(L->next);                        O(1)
            current = current->next;                O(1)
            current->next = NULL;                   O(1)
            current->data = element;                O(1)
            return L;                               O(1)
        }
    }
```

5.2.9 remove_from

```
    /*
    * @pre: element = d_0 ∧ L = (e_0, ..., e_{i-1})
    * @post: ∀0 ≤ i ≤ length(L) − 1, L[i] ≠ d_0
    */

    list *remove_from(list *L,
                      void *element,
                      bool (*compare)(const void *, const void *))
    {
        assert(element);                            O(1)
        cell *current = L->next, *tmp = L;          O(1)
        if(is_empty(L))                             O(1)
            return L;                               O(1)
        else if(!tmp->next && compare(tmp->data, element)) O(1)
        {
            free(tmp);                              O(1)
            return NULL;                            O(1)
        }
        else
        {
            while(current)                          O(length(L))
```

```
            {
                if(compare(current->data, element))  O(1)
                {
                    tmp->next = current->next;       O(1)
                    free(current);                   O(1)
                }
            tmp = tmp->next;                         O(1)
            current = tmp->next;                     O(1)
            }
            return L;                                O(1)
        }
    }
```

Complexité linéaire.