Chapitre 4 - Ensembles dynamiques, pointeurs et listes

Georges-Pierre BONNEAU (cours) - Mica MURPHY (note) - Antoine SAGET (note)

Lundi 19 Novembre 2018

Introduction

Opération sur un ensemble dynamique :

- **Énumération** des valeurs de l'ensemble
- **Insertion** de valeur
- **Supression** de valeur
- **Requête** : est-ce que la valeur k est dans l'ensemble E ?
- Si une relation d'ordre existe : valeurs Minimum, Maximum, Suivante et Précédente

Pointeurs

- On appelle "Objet" les éléments des ensembles
- Un objet peut comperter un ou plusieurs "Attributs"
- Pointeur vers un objet : repère, adresse de l'objet

Analogie avec les tableaux :

- Soit T un tableau de comptes bancaires
- $T[i]$ un compte bancaire d'indice i
- p : un **pointeur** vers un compte bancaire
- $p\hat{}$: **objet** = compte bancaire pointé par p
- **Attributs** : Solde de $p\hat{}$, découvert de $p\hat{}$, etc.
- Ne pas confondre p le pointeur et $p\hat{}$ l'objet pointé par p

Deux opéraitons fondamentales sur les pointeurs

`Allouer(p)`	`Libérer(p)`
Recherche d'un espace mémoire suffisamment grand pour stocker un objet. L'adresse de l'espace mémoire est écrite dans p.	Indique au système d'exploitation que l'espace mémoire pointé par p peut être utilisé.

- `Nil` : une valeur spéciale du pointeur, signfiant qu'il ne pointe vers aucun objet

Types :

- `Objet` : un type quelconque
- `AdObjet` : un type pointeur vers un type Objet

Pièges classiques des pointeurs

Soit `type Objet : {`$\triangle, \square, \bigcirc$`}` et `p1`, `p2` pointeurs vers des `Objet` :

- Piège 1 : Accéder à l'objet $p_1\hat{}$ alors que le pointeur p_1 a été libéré
  - `Libérer(p1) ; somme <- somme + solde de p1↑`
- Piège 2 : Accéder à l'objet $p_1\hat{}$ alors que le pointeur p_2 a été libéré (et que $p_1\hat{} == p_2\hat{}$)
  - `p2 <- p1 ; libérer(p1)`
- Piège 3 : Modifier l'objet $p_1\hat{}$ en oubliant que cela modifie aussi $p_2\hat{}$ (et que $p_1\hat{} == p_2\hat{}$)
  - `p1↑ <-`$\bigcirc$
- Piège 4 : Libérer p_1 alors qu'il est le dernier pointeur un certain objet, donc cet objet est "perdu" *(pour éviter de le perdre si on a encore besoin, on peut sauver sa valeur avant de libérer le pointeur ou créer un autre pointeur vers cet objet)* (**Explication confuse**)
  - `Libérer(p1)`
- Piège 5 : Fuite de mémoire = allouer en boucle sans penser à libérer (ré-allouer un pointeur sans avoir libéré son contenu auparavant)
  - `Tant que Condition : (Allouer(p1) ; utilisation de p1↑)`

Listes

Définition. Une structure de donnée dans laquelle les objets sont arrangés linéairement, et permettant de représenter des ensembles dynamiques.

Dans les listes, l'ordre des objets est determiné par l'ajout d'un ou deux attributs **pointeurs**.

- p : pointeur vers un objet
- $p\hat{}$: objet dont les attributs sont
  - succ/suiv/next de $p\hat{}$: attribut de l'objet $p\hat{}$, qui est un pointeur vers l'objet suivant dans la liste
  - pred/prec/prev de $p\hat{}$: un pointeur de l'objet précédent dans la liste. *(facultatif)*
- tête de liste : pointeur vers le 1^{er} élement de la liste

Si on ne dispose que de succ, c'est une **liste simplement chaînée**, ; si on connaît succ et pred, on parle de **liste doublement chaînée**.

Représentation graphique des deux types

Représentation Non Contigue des objets en mémoire. Avec les tableaux, les objets sont représentés de manière contigue en mémoire.

Définition de types pour les listes

Simplement chaînée :

- **Objet** : type quelconque
- **Doublet** : le type <**O** : un Objet; **Succ** : un AdDoublet>
- **AdDoublet** : le type pointeur vers un **Doublet**

Doublement chaînées :

- **Objet** : type quelconque
- **Triplet** : le type <**O** : un Objet; **Succ** : un AdTriplet; **Pred** : un AdTriplet>
- **AdTriplet** : le type pointeur vers un **Triplet**

Recherche, Insertion, Suppression

Recherche : action

```
  (la donnée L : un AdTriplet ; la donnée O : un objet ;
  le résultat x : un AdTriplet)

{État initial : L pointe vers la tête de liste}
{État final :
  - X vaut Nil et O n'est pas contenu dans la liste
  OU ALORS
  - X != Nil et X est le pointeur vers le premier élemént de la liste contenant O}

{Algorithme :
  X <- L
  Tant que (X != Nil) et puis (O de x↑ != O) faire :
    x <- succ de x↑
}
```

Suppression : action

```
  (la donnée-résultat L : un AdTriplet ; la donnée X : un AdTriplet)

{État initial :
- L pointe vers la tête de liste
- X pointe vers un élément de la liste (qui n'est pas vide)}
{État final : L pointe vers la tête de liste de liste dans laquelle X a été supprimé}

{Algorithme :
  Si pred de X↑ != Nil alors :
    succ de (pred de X↑)↑ <- succ de X↑
  Sinon
    L <- succ de X↑

  Si succ de X↑ != Nil alors :
    pred de (succ de X↑) <- pred de X↑
}
```

Insertion-tête : action

```
  (la donnée-résultat L : un AdTriplet ; la donnée X : un AdTriplet)

{État initial :
- L pointe vers la tête de liste
- X pointe vers un objet (donc x!= Nil)}
{État final : L pointe vers la tête de liste de liste avec x inséré en tête}

{Algorithme :
  succ de X↑ <- L
  Si L != Nil alors : pred de L↑ <- X
  pred de X↑ <- Nil
  L <- X
}
```

Complexité asyptotique dans des listes ou tableaux de N éléments :

Opération	Complexité asymptotique : liste	Complexité asymptotique : tableau
Recherche	$O(N)$	$O(i) = O(N)$
Suppression	$O(1)$	$O(N - i) = O(N)$
Insertion	$O(1)$	$O(1)$

Rajout d'élément fictif dans une liste

Fictif : c'est un vrai objet, mais dont la valeur sera ignorée (MonNil).

Le successeur du dernier élément devient l'élément fictif. Le prédecesseur du premier élément devient l'élément fictif.

Le successeur de l'élément fictif est le premier élément. Le prédecesseur de l'élément fictif est le dernier élément.

```
Insertion-tête : action
  (la donnée-résultat MonNil : un AdTriplet ; la donnée X : un AdTriplet)

{Algorithme :
  succ de X↑ <- succ de MonNil↑
  pred de X↑ <- MonNil
  pred (succ MonNil↑)↑ <- X
  succ de MonNil↑ <- X
}
```

```
Supprimer : action
  (la donnée-résultat MonNil : un AdTriplet ; la donnée X : un AdTriplet)

{Algorithme :
  succ de (pred de X↑)↑ <- succ de X↑
  pred de (succ de X↑)↑ <- pred de X↑
}

{Remarque : pred/succ de X↑ != Nil même si X est en début/fin de liste.}
```

Piège

```
DernierElement : action
  (dernier L : un AdDoublet ; résultat X : un AdDoublet)

{État initial : L pointe vers la tête de liste}
{État final : (X vaut Nil) ou alors (X pointe vers le dernier élément de la liste)}

{Algorithme FAUX (renvoie Nil) :
  X <- L
  Tant que X != Nil faire :
    X <- succ de X↑
}

{Algorithme FAUX (ne marche pas quand la liste est vide) :
  X <- L
  Tant que succ de X↑ != Nil faire :
     X <- succ de X↑
}

{Algorithme CORRECT :
  Si L = Nil alors X <- Nil
  Sinon :
    X <- L
    Tant que succ de X↑ != Nil faire :
      X <- succ de X↑
}
```