DS
N° 2

Licence Sciences et Technologies (S3)
2009-2010
Algotithmes et Programmation Impérative 2

8 janvier 2010

Documents de cours autorisés

1 Représentations des ensembles

Un ensemble est une collection d'objets distinguables appelés éléments. Ces éléments sont deux à deux distincts. Cette propriété doit être maintenue quand on manipule les ensembles. On doit donc pouvoir tester l'égalité de deux éléments. On suppose que l'ensemble des éléments est représenté par le type `ELEMENT` et que celui-ci est muni d'une relation d'ordre total. On suppose également que l'on dispose d'une fonction

```
// compareElements(e1,e2) = -1 si e1 < e2
//                        =  0 si e1 = e2
//                        =  1 si e1 > e2
function compareElements(e1,e2: ELEMENT): INTEGER ;
```

Tous les programmes nécessitant la comparaison de deux éléments doivent utiliser cette fonction. On va examiner différentes façons de représenter les ensembles et les opérations sur ces ensembles.

1.1 Représentation par listes

Exercice 1.

Dans cette première partie, on représente les ensembles par des listes chaînées. Ainsi, on suppose que l'on dispose de l'unité `U_Liste` et que l'on a fait les déclarations suivantes :

```
Type
   ENSEMBLE = LISTE ;
```

Ainsi, toutes les primitives sur les listes seront disponibles pour les ensembles.

Question 1.1. *Écrire un prédicat (fonction à résultat booléen) nommé `appartient` qui teste l'appartenance d'un élément `x:ELEMENT` à un ensemble `E:ENSEMBLE`. Donner une version récursive et une version itérative.*

Solution 1.1.

```
// appartient(x,E) teste si x appartient à E (version récursive)
function appartient(const x: ELEMENT; const E: ENSEMBLE): BOOLEAN ;
begin
   if estListeVide(E) then appartient := false
   else
      appartient := (compareElements(x,tete(E))=0) or appartient(x,reste(E))
end; {appartient}

// appartient_iter(x,E) teste si x appartient à E (version itérative)
function appartient_iter(const x: ELEMENT; const E: ENSEMBLE): BOOLEAN ;
var
   F: ENSEMBLE;
   B: BOOLEAN ;
begin
   B := false ;
   F := E ;
   while not B and not estListeVide(F) do
```

```
      begin
         if compareElements(x,tete(F))=0 then B := true
         else F := reste(F)
      end; {while}
      appartient_iter := B
end; {appartient_iter}
```

Question 1.2. *Avant d'ajouter un élément à un ensemble, on doit s'assurer qu'il n'y est pas déjà présent. Écrire une fonction nommée `ajouteElement` qui ajoute un élément `x:ELEMENT` à un ensemble `E:ENSEMBLE`.*

Solution 1.2. On utilise le prédicat `appartient` et la fonction `ajouteEnTete` :

```
// ajouteElement(x,E) ajoute l'élément x à l'ensemble E s'il n'y est pas déjà
function ajouteElement(const x: ELEMENT; const E: ENSEMBLE): ENSEMBLE ;
begin
   if appartient(x,E) then ajouteElement := E
   else ajouteElement := ajouteEnTete(x,E)
end; {ajouteElement}
```

Question 1.3. *Rappelons que si E et F sont deux ensembles, alors la réunion de E et de F, notée $E \cup F$, est l'ensemble des éléments qui appartiennent à E ou à F. Écrire une fonction nommée `union` qui calcule la réunion de deux ensembles (cette fonction va créer une nouvelle liste). Donner une version récursive et une version itérative.*

Solution 1.3.

```
// union(E,F) construit E U F
function union(const E,F: ENSEMBLE): ENSEMBLE ;
begin
   if estListeVide(E) then union := F
   else union := ajouteElement(tete(E),union(reste(E),F))
end; {union}

// union_iter(E,F) construit E U F (version itérative)
function union_iter(E,F: ENSEMBLE): ENSEMBLE ;
var
   G: ENSEMBLE ;
begin
   G := F ;
   while not estListeVide(E) do
      begin
         G := ajouteElement(tete(E),G) ;
         E := reste(E)
      end; {while}
   union_iter := G
end; {union_iter}
```

Pour évaluer le coût de ces opérations (versions itératives **ou** récursives), on prend en compte uniquement le nombre de comparaisons entre objets de type `ELEMENT`. Soit E un ensemble de cardinal n et F un ensemble de cardinal m.

Question 1.4. *Donner dans le pire cas*

1. *le coût de l'appel à `appartient(x,E)` en fonction de n*
2. *le coût de l'appel à `ajouteElement(x,E)` en fonction de n*
3. *le coût de l'appel à `union(E,F)` en fonction de n et de m*

Solution 1.4.

1. Dans le pire des cas, l'élément n'appartient pas à E ou bien se trouve en fin de liste. On doit donc parcourir tout l'ensemble pour s'en rendre compte. On fait donc n comparaisons. Ainsi, le coût du test d'appartenance est en $\mathcal{O}(n)$.
2. Avant d'ajouter un élément à un ensemble, on teste d'abord s'il n'y est pas déjà. Donc, dans le pire cas, le coût de l'ajout est en $\mathcal{O}(n)$.
3. le coût de la fonction qui calcule la réunion est en $\mathcal{O}(n \times (n+m))$.

1.2 Représentation par des listes ordonnées

Exercice 2.

Dans cette partie, on suppose que les ensembles sont représentés par des listes chaînées comme dans la première partie. De plus, on suppose que celles-ci sont ordonnées par ordre croissant et on doit maintenir cette propriété d'ordre.

Question 2.1. *Récrire la version itérative du prédicat* `appartient` *en tenant compte de la propriété d'ordre.*

Solution 2.1.

```
// member(x,E) teste si x appartient à E trié (version récursive)
function member(const x: ELEMENT; const E: ENSEMBLE): BOOLEAN ;
begin
   if estListeVide(E) or (compareElements(x,tete(E))=-1) then member := false
   else
      member := (compareElements(x,tete(E))=0) or member(x,reste(E))
end; {member}

// member_iter(x,E) teste si x appartient à E trié (version itérative)
function member_iter(const x: ELEMENT; const E: ENSEMBLE): BOOLEAN ;
var
   F: ENSEMBLE;
begin
   F := E ;
   while not estListeVide(F) and (compareElements(x,tete(F))<>0) do F := reste
       (F) ;
   member_iter := not estListeVide(F) and (compareElements(x,tete(F))=0)
end; {member_iter}
```

Question 2.2. *Récrire la fonction* `ajouteElement` *en maintenant la propriété d'ordre.*

Solution 2.2. C'est l'algorithme de l'insertion dans une liste triée (vu en TD) !

```
// ajouteElementTriee(x,E) ajoute l'élément x à l'ensemble E s'il n'y est pas
    déjà
// dans le cas trié
function ajouteElementTriee(const x: ELEMENT; const E: ENSEMBLE): ENSEMBLE ;
begin
   if estListeVide(E) or (compareElements(x,tete(E))=-1) then
      ajouteElementTriee := ajouteEntete(x,E)
   else
      ajouteElementTriee := ajouteEnTete(tete(E),
                                         ajouteElementTriee(x,reste(E)))
end; {ajouteElementTriee}
```

Question 2.3. *Récrire la version itérative de la fonction* `union` *en maintenant la propriété d'ordre.*

Solution 2.3. C'est l'algorithme de la fusion de deux listes triées !

```
// union_triee_iter(E,F) construit E U F
// cas des ensembles triés (version itérative)
function union_triee_iter(E,F: ENSEMBLE): ENSEMBLE ;
var
   G: ENSEMBLE;
begin
   G := LISTEVIDE;
   while not estListeVide(E) and not estListeVide(F) do
      begin
         if compareElements(tete(E),tete(F))=-1 then
            begin
               ajouterEnFin(tete(E),G);
               E := reste(E)
            end
         else
```

```
            if compareElements(tete(E),tete(F))=1 then
               begin
                  ajouterEnFin(tete(F),G);
                  F := reste(F)
               end
            else
               begin
                  ajouterEnFin(tete(E),G);
                  E := reste(E);
                  F := reste(F);
               end;
      end; {while}
   while not estListeVide(E) do
      begin
         ajouterEnFin(tete(E),G);
         E := reste(E)
      end; {while}
   while not estListeVide(F) do
      begin
         ajouterEnFin(tete(F),G);
         F := reste(F)
      end; {while}
   union_triee_iter := G;
end; {union_triee_iter}
```

Question 2.4. *Comparer les coûts des nouvelles versions avec ceux de la première partie.*

Solution 2.4.

Représentation	appartenance	ajout	union
Liste	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Liste triée	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

1.3 Représentation par des ABO

Exercice 3.

On peut enfin représenter les ensembles par des arbres binaires ordonnés (ABO) dont les noeuds sont deux à deux distincts. On suppose donc que l'on dispose des unités `U_Element` et `U_Arbre` et de toutes les primitives qu'elles contiennent. On suppose également que l'on a fait les déclarations suivantes :

```
type
   ENSEMBLE = ARBRE ;
const
   ENSEMBLEVIDE = ARBREVIDE ;
```

Question 3.1. *Récrire le prédicat `appartient` pour cette nouvelle représentation.*

Solution 3.1.

```
// appartient(x,E) teste si x appartient à E (version récursive)
function appartient(const x: ELEMENT; const E: ENSEMBLE): BOOLEAN ;
begin
   if estArbreVide(E) then appartient := false
   else
      if compareElements(x,racine(E)) = 0 then appartient := true
      else
         if compareElements(x,racine(E)) = -1 then
            appartient := appartient(x,gauche(E))
         else
            appartient := appartient(x,droit(E))
end; {appartient}

// appartient_iter(x,E) teste si x appartient à E (version itérative)
```

```
function appartient_iter(const x: ELEMENT; const E: ENSEMBLE): BOOLEAN ;
var
   F: ENSEMBLE;
   B: BOOLEAN ;
begin
   B := false ;
   F := E ;
   while not B and not estArbreVide(F) do
      begin
         if compareElements(x,racine(F)) = 0 then
            B := true
         else
            if compareElements(x,racine(F)) = -1 then
               F := gauche(F)
            else
               F := droit(F)
      end; {while}
      appartient_iter := B
end; {appartient_iter}
```

Question 3.2. *Écrire une procédure récursive nommée* `ajouterElement` *qui ajoute un élément à un ensemble.*

Solution 3.2. C'est une adaptation de la procédure `insererABO` faite en cours.

```
// ajouteElement(x,E) ajoute l'élément x à l'ensemble E s'il n'y est pas déjà
procedure ajouteElement(const x: ELEMENT; var E: ENSEMBLE) ;
var
   F: ENSEMBLE;
begin
   if estArbreVide(E) then
      E := creerArbre(x,ARBREVIDE,ARBREVIDE)
   else
      if compareElements(x,racine(E)) = -1 then
         begin
            F := gauche(E);
            ajouteElement(x,F);
            modifierGauche(E,F)
         end
      else
         if compareElements(x,racine(E)) = 1 then
            begin
               F := droit(E);
               ajouteElement(x,F);
               modifierDroit(E,F)
            end
end; {ajouteElement}
```

Question 3.3. *Utiliser la procédure* `ajouterElement` *pour récrire une fonction récursive nommée* `union` *qui construit la réunion de deux ensembles.*

Solution 3.3.

```
// union(E,F) construit E U F
function union(const E,F: ENSEMBLE): ENSEMBLE ;
var
   G: ENSEMBLE;
begin
   if estArbreVide(E) then union := F
   else
      begin
         G := union(droit(E),union(gauche(E),F)) ;
         ajouteElement(racine(E),G) ;
         union := G
      end
```

```
end; {union}
```

Question 3.4. *Analyser les coûts de ces nouvelles versions et les comparer aux coûts des versions précédentes.*

Solution 3.4. Il est facile de voir que les coûts de `appartient` et `ajouteElement` sont en $\mathcal{O}(h)$ où h est la hauteur de l'arbre. On sait par ailleurs que la hauteur de l'arbre dépend de sa forme.

Le calcul de la réunion revient à l'ajout des éléments de E à ceux de F. Si E est de taille n, alors le coût de la réunion est en $\mathcal{O}(n \cdot h)$ où h est la hauteur de l'arbre représentant F.

2 Sur les arbres binaires

Exercice 4.

Question 4.1. *On définit la hauteur minimale d'un arbre comme étant la longueur (en nombre d'arêtes) du plus court chemin de la racine à une feuille. On suppose que la hauteur minimale de l'arbre vide est -1. Dessiner les arbres binaires suivants et donner leur hauteur minimale :*

```
a = <1;Δ;Δ>
b = <1;<2;<3;<5;Δ;Δ>;Δ>;<4;Δ;Δ>>;Δ>
c = <1;Δ;<2;<3;Δ;Δ>;<4;<5;Δ;Δ>;<6;Δ;Δ>>>>
d = <1;<2;<4;Δ;Δ>;Δ>;<3;<5;<7;Δ;Δ>;<8;Δ;Δ>>;<6;<9;Δ;Δ>;Δ>>>
```

Solution 4.1.

a	b	c	d
0	2	2	2

Question 4.2. *Écrire une fonction nommée `hauteur_min` qui calcule la hauteur minimale d'un arbre binaire.*

Solution 4.2.

```
// hauteur_min(a) renvoie la hauteur minimale de l'arbre a soit la
// longueur du plus court chemin de la racine à une feuille
function hauteur_min(const a: ARBRE): INTEGER ;
begin
   if estArbreVide(a) then hauteur_min := -1
   else
      if estArbreVide(Gauche(a)) then
         hauteur_min := 1 + hauteur_min(Droit(a))
      else
         if estArbreVide(Droit(a)) then
            hauteur_min := 1 + hauteur_min(Gauche(a))
         else
            hauteur_min := 1 + min(hauteur_min(gauche(a)), hauteur_min(droit(a
               )))
end; {hauteur_min}
```

Un arbre binaire est dit *complet* si tous ses noeuds ont zéro ou deux fils et que toutes ses feuilles sont à la même profondeur. On suppose que l'arbre vide est complet.

Question 4.3. *Écrire un prédicat nommé `estComplet` qui teste si un arbre binaire est complet.*

Solution 4.3.

```
// estComplet(a) vaut true si a est vide ou si Gauche(a) et Droit(a) ont même
// hauteur et sont complets
function estComplet(const a: ARBRE): BOOLEAN ;
begin
   estComplet := estArbreVide(a) or
                  ((hauteur(Gauche(a))=hauteur(Droit(a))) and
                    estComplet(Gauche(a)) and
                    estComplet(Droit(a))
                  )
end; {estComplet}
```

Un arbre binaire a de hauteur $h(a)$ est *quasi-complet* si à chaque profondeur $0 \leq p \leq h(a) - 1$ on trouve 2^p noeuds, et que les noeuds de profondeur maximale soient tous situés le plus à gauche possible. En particulier tout arbre complet est quasi-complet.

Question 4.4. *Dessiner des arbres quasi-complets de taille 4 à 7.*

Solution 4.4.

Question 4.5. *Dessiner deux arbres non vides, non quasi-complets dont les deux sous-arbres sont quasi-complets.*

Solution 4.5.

Question 4.6. *Donner une condition nécessaire et suffisante pour qu'un arbre soit quasi-complet.*

Solution 4.6. Un arbre binaire est quasi-complet s'il vérifie l'une des 3 propriétés suivantes :

- il est vide
- son fils gauche est complet de hauteur h, et son fils droit est quasi-complet de hauteur h
- son fils gauche est quasi-complet de hauteur h, et son fils droit est complet de hauteur $h - 1$

Question 4.7. *Écrire un prédicat nommé* `estQuasiComplet` *qui teste si un arbre est quasi-complet.*

Solution 4.7.

```
// estQuasiComplet(a) est un prédicat qui renvoie true si a est quasi-complet
// autrement-dit a est complet où manquent éventuellement quelques feuilles
// sur le dernier rang.
function estQuasiComplet(const a: ARBRE): BOOLEAN ;
begin
   estQuasiComplet := estArbreVide(a) or
                        ((hauteur(Gauche(a))=hauteur(Droit(a))) and
                          estComplet(Gauche(a)) and
                          estQuasiComplet(Droit(a))) or
                           ((hauteur(Gauche(a))-hauteur(Droit(a))=1) and
                             estComplet(Droit(a)) and
                             estQuasiComplet(Gauche(a)))
end; {estQuasiComplet}
```