M2 CCI – Algorithmique – Devoir surveillé

Durée 2h, sans documents **7 décembre 2023**

NE PAS RECOPIER les énoncés des questions. Ne pas perdre de temps à un soin excessif de la présentation. Les questions sont indépendantes. Le barème est indicatif.

1. Suppression des éléments redondants (représentations contiguë et chainée)

Dans une séquence **S**, un élément est *redondant* s'il existe dans la séquence un autre élément de même valeur (une séquence dont les éléments sont distincts deux à deux, n'a pas d'élément redondant).

Q1 [6 points]

(i) Représentation contiguë (ordre de liste non important)

On étudie un algorithme de suppression des éléments redondants d'une séquence d'entiers **S** donnée sous forme contiguë dans un tableau avec longueur explicite. Il n'y a aucune hypothèse d'ordre sur la séquence. Par exemple, si à l'état initial **S = [4, 19, 4, 8, 11, 11, 3, 4, 19]**, à l'état final la séquence sans ses éléments redondants est **S = [4, 19, 8, 11, 3]**.

On spécifie l'action suivante :

n : constante de type entier > 0 *{taille maximum des séquences}*

SupRedT : action (donnée-résultat T : tableau sur [1…n] d'entier ; L : entier sur [0…n])

{SupRedT(T,L) : supprime les éléments redondants d'une séquence d'entiers de longueur L donnée dans le tableau T entre les positions 1 et L. À l'état final, la séquence privée de ses éléments redondants est implantée dans T entre les positions 1 et L.}

— Donner une **réalisation itérative** de l'action **SupRedT**. **L'algorithme doit nécessairement procéder selon le principe suivant :** un élément de la séquence est supprimé s'il existe avant lui un élément de même valeur.

(ii) Représentation chaînée (ordre important, liste triée)

On traite ici la suppression des éléments redondants d'une **séquence d'entiers triée** (ordre croissant). Par exemple, si à l'état initial **S = [3, 4, 4, 4, 8, 11, 11, 19, 19]**, à l'état final **S = [3, 4, 8, 11, 19]**. De plus, **S** est représentée sous forme chaînée standard avec pointeurs.

On spécifie l'action suivante :

adCelE : type pointeur de CelE

CelE : type <E : entier ; Suc : adCelE>

SupRedL : action (donnée-résultat T : adCelE)

*{SupRedL(T) : supprime les éléments redondants d'une séquence d'entiers triée en ordre croissant, donnée par la liste de tête T. À l'état final, la liste de tête T est en ordre croissant et représente la séquence privée de ses éléments redondants. L'algorithme procède par **modification des liens de chaînage**. Les cellules contenant les éléments redondants ont été libérées. }*

— Donner une **réalisation itérative** de l'action **SupRedL**, **utilisant** le fait que la séquence est triée.

2. Échange des extrémités d'une liste (représentation chaînée)

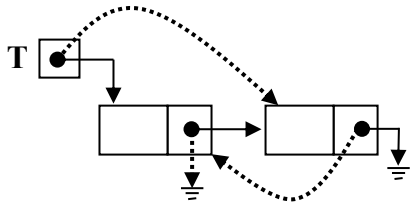
Étant donnée une liste chaînée, on étudie un algorithme d'échange du premier et du dernier élément de cette liste. **Cet échange doit être fait uniquement par modification du chaînage et sans déplacer les valeurs d'éléments. On sait a priori que la liste traitée a au moins deux éléments**.

On utilise le lexique suivant :
Élément : type
adCel : type pointeur de Cel
Cel : type <E : Élément ; Suc : adCel>
T : adCel *{ adresse de tête de la liste traitée. Précondition : T≠Nil et T↑.Suc ≠Nil}*

Q2 [4 points]

(i) Étude du cas simple : liste de deux éléments

Le dessin ci-dessous représente le principe de modification des liens de chaînage :



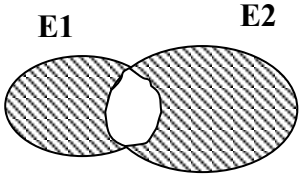
— Donner la **réalisation de l'algorithme** dans ce cas.

(ii) Étude du cas général : liste d'au moins trois éléments

— Dessiner le **principe de modification des liens de chaînage dans le cas général**. On fera apparaître deux variables contenant les **adresses du dernier et de l'avant-dernier** élément de la liste.

— Donner la **réalisation de l'algorithme** correspondant (**y compris le calcul des adresses du dernier et de l'avant-dernier élément de la liste**).

3. Différence symétrique de deux ensembles (représentation chaînée)



La *différence symétrique* de deux ensembles **E1** et **E2**, notée **E1 Δ E2**, est l'ensemble formé d'une part des éléments qui appartiennent à **E1** et n'appartiennent pas à **E2** et d'autre part des éléments qui appartiennent à **E2** et n'appartiennent pas à **E1**.

Dans ce qui suit, on traite des ensembles d'entiers représentés par des **séquences sans répétition**, elles-mêmes sous forme de **listes simplement chaînées**. De plus, ces listes sont munies d'un **élément fictif de tête** et d'une **adresse de queue**. Le champ successeur du dernier élément a la valeur **Nil**.

On utilise le lexique suivant :
adCel : type pointeur de Cel
Cel : type <E : entier ; Suc : adCel>
Liste : type <F : adCel ; Q : adCel>
*{si L est de type Liste : **L.F** est l'adresse de l'élément fictif de tête et **L.Q** est l'adresse de l'élément de queue. Si la liste est vide, L.Q = L.F.}*

Q3 [10 points]

(i) Illustration de la représentation des ensembles

— **Dessiner** les listes correspondant à un ensemble vide et à un ensemble de trois éléments. Faire apparaître la variable **L** de type **Liste** donnant accès à ces listes.

(ii) Construction de la différence symétrique de deux ensembles

Dans cette question, **aucun ordre n'est imposé sur les listes représentant les ensembles**.

On spécifie une action nommée **CréerDifSym**, pour créer la différence symétrique (Δ).
CréerDifSym : action (donnée L1, L2 : Liste ; résultat L3 : Liste)
{à l'état initial L1 et L2 représentent deux ensembles E1 et E2. À l'état final, L3 représente la différence symétrique E1Δ E2 ; les listes L1 et L2 sont inchangées.}

Exemple : si **E1** = **[5 ; 6 ; 7 ; 3]** et **E2** = **[4 ; 8 ; 7 ; 6 ; 2]** alors **E1ΔE2** = **[5 ; 3 ; 4 ; 8 ; 2]**

La différence symétrique (Δ) est l'union de deux différences (\) : **E1ΔE2** = **(E1\E2) ∪ (E2\E1)**.
Sur notre exemple, **E1\E2** = **[5 ; 3]** et **E2\E1** = **[4 ; 8 ; 2]**.

Pour réaliser l'action, on part du principe suivant : **construire les listes correspondant à chacune des deux différences ; puis réaliser leur union par une simple connexion** (en effet, **(E1\E2) ∩ (E2\E1) = ∅**).

On introduit une action nommée **CréerDif** pour créer uniquement la différence (\) :

CréerDif : action (donnée LA, LB : Liste ; résultat LC : Liste)

{à l'état initial LA et LB représentent deux ensembles A et B. À l'état final, LC représente la différence A\B et les listes LA et LB sont inchangées.}

— Donner une **réalisation itérative** de l'action **CréerDif**.
— Donner une **réalisation itérative** de l'action **CréerDifSym** en utilisant **CréerDif** et en appliquant le principe énoncé.
Pour gérer les cellules, on utilisera les primitives **Allouer** et **Libérer**.

(iii) Modification d'une des listes données - Cas de listes en ordre croissant

On donne deux listes **L1** et **L2 triées en ordre croissant** représentant deux ensembles **E1** et **E2**. On veut modifier ces listes de telle sorte qu'à l'état final, **L1** représente la différence symétrique **E1ΔE2** et que **L2** soit vide. L'algorithme doit procéder par **modification des liens de chaînage** (pas d'allocation de nouvelles cellules) et **les cellules non utilisées doivent être libérées**.

On spécifie à cet effet une action nommée **ModifierParDifSym** :

ModifierParDifSym : action (donnée-résultat L1, L2 : Liste)

{à l'état initial L1 et L2 représentent deux ensembles E1 et E2. À l'état final, L1 représente la différence E1ΔE2, L2 est vide et les cellules non utilisées ont été libérées.}

— **Dessiner** les états initiaux et finaux des listes **L1** et **L2**, après un appel à **ModifierParDifSym(L1, L2)**,

où **L1** est la liste chaînée triée en ordre croissant qui représente l'ensemble **E1** du **(ii)**, soit **[3 ; 5 ; 6 ; 7]**

et **L2** est la liste chaînée triée en ordre croissant qui représente l'ensemble **E2** du **(ii)**, soit **[2 ; 4 ; 6 ; 7 ; 8]**.

— Donner une **réalisation** de l'action **ModifierParDifSym** en exploitant le fait que **les séquences sont triées** et que la différence symétrique est la différence entre l'union et l'intersection des deux ensembles : **E1ΔE2 = (E1∪E2) \ (E1∩E2)**. En effet, **E1ΔE2** contient les entiers qui sont dans **E1** ou dans **E2** (union de **E1** et **E2**) mais qui ne sont pas à la fois dans **E1** et dans **E2** (intersection de **E1** et **E2**).

Pour cela, appliquer le ***principe d'interclassement*** de la manière suivante (**toute autre solution ne sera pas considérée lors de la correction**) :

Lors du parcours simultané des listes représentant **E1** et **E2**,

pour trouver les éléments qui appartiennent ou non à **E1∩E2**, on exploite le fait que les séquences sont triées

on conserve dans **L1** les éléments qui n'appartiennent pas à **E1∩E2** (i.e. soit on laisse les éléments concernés de **L1** en place soit on déplace les éléments concernés de **L2** dans **L1**)

on supprime dans **L1** et dans **L2** les éléments qui appartiennent à **E1∩E2**.

À l'issue du parcours simultané, la liste **L1** est complétée, si besoin, par ce qui reste dans la liste **L2**.

M2 CCI – Algorithmique

AL – DS 2 : des exemples de solutions **7 décembre 2023**

1. Suppression des éléments redondants

Q1

(i) 3 points

{Parcours de la liste donnée. Recherche avec sentinelle pour déterminer si l'élément courant doit être supprimé. La suppression se fait par remplacement par le dernier élément (aucun ordre imposé).}

```
SupRedT (T, L) :
   i, j : entier sur [1…n+1]                          {pour le parcours et la recherche}

   i ← 2
   tant que i ≠ L+1
         j ← 1 ; tant que Ti ≠ Tj : j← j+1            {la position i sert de sentinelle}
         si j = i alors i ← i+1
         sinon {supprimer Ti }
               Ti ← TL
               L ← L-1                    {remplacement par le dernier : aucun ordre imposé}
```

(ii) 3 points

{Parcours de la liste donnée avec maintien de l'adresse du prédécesseur de l'élément courant. Un élément est supprimé s'il est égal à son prédécesseur.}

```
SupRedL (T) :
   AC, AP : adCelE                {adresses de l'élément courant et de son prédécesseur}
   X : adCelE                                                    {pour la suppression}

   si T ≠ Nil alors
         AP ← T ; AC← AP↑.Suc
         tant que AC ≠ Nil
               si AC↑.E = AP↑.E alors          {AC↑.E est redondant ; supprimer AC}
                     X ← AC ; AP↑.Suc ← AC↑.Suc ; libérer (X)
               sinon     AP ← AC
               AC← AP↑.Suc
```

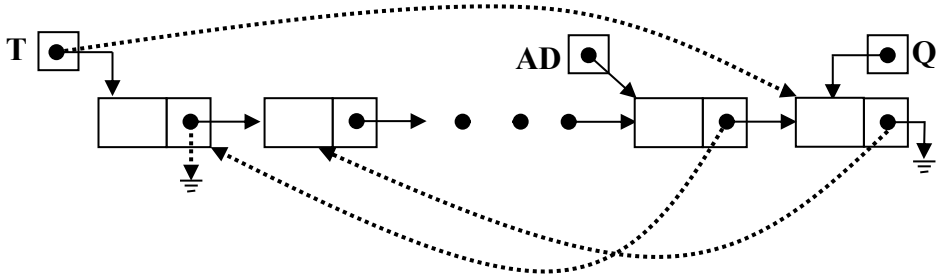
2. Échange des extrémités d'une liste

Q2

(i) Liste de deux éléments 1 point

```
Q : adCel
Q ← T↑.Suc ; Q↑.Suc ← T ; T↑.Suc ← Nil ; T ← Q
```

(ii) Liste d'au moins trois éléments 1 point pour le dessin, 2 points pour l'algo

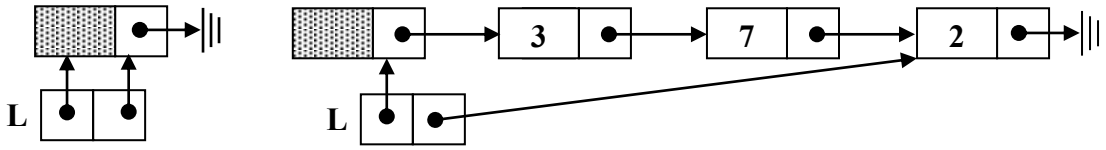


```
AD, Q : adCel
AD ← T ; Q ← AD↑.Suc
Tant que Q↑.Suc ≠ Nil
        AD ← Q ; Q ← AD↑.Suc
AD↑.Suc ← T ; Q↑.Suc ← T↑.Suc ; T↑.Suc ← Nil ; T ← Q
```

3. Différence symétrique de deux ensembles

Q3

(i) 1 point



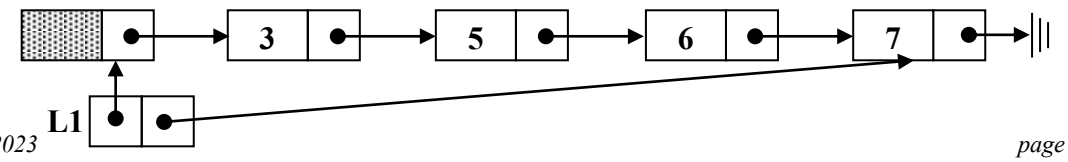
(ii) 2.5 points pour CréerDif, 1.5 points pour CréerDifSym

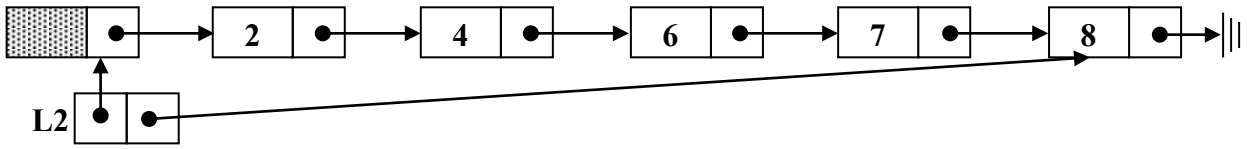
La réalisation de **CréerDif** s'appuie sur le principe suivant : pour chaque élément de **A** (*parcours*), on le place dans **C** (*ajout en queue*), s'il n'appartient pas à **B** (*recherche*).

```
CréerDif (LA, LB, LC) :
        AC1, AC2 : adCel                    {pour le parcours de LA et la recherche dans LB}
        N : adCel                                          {pour la création de cellules}

        Allouer(LC.F)
        LC.Q ← LC.F ; (LC.Q)↑.Suc ← Nil                    {création de la liste vide}
        AC1 ← (LA.F)↑.Suc
        tant que AC1 ≠ Nil
                AC2 ← (LB.F)↑.Suc
                tant que AC2 ≠ Nil et puis AC1↑.E ≠ AC2↑.E
                    AC2 ← AC2↑.Suc
                si AC2 = Nil alors                                  {ajout en queue de LC}
                        Allouer(N)
                        N↑.E ← AC1↑.E ; N↑.Suc ← Nil
                        (LC.Q)↑.Suc ← N
                        LC.Q ← N
                AC1 ← AC1↑.Suc

CréerDifSym(L1, L2, L3) :
        L4 : Liste
        CréerDif(L1, L2, L3)
        CréerDif(L2, L1, L4)
        {Connecter les deux listes puis supprimer le fictif de L4}
        (L3.Q)↑.Suc ← (L4.F)↑.Suc
        si L4.F ≠ L4.Q alors L3.Q ← L4.Q
        Libérer(L4.F)
```

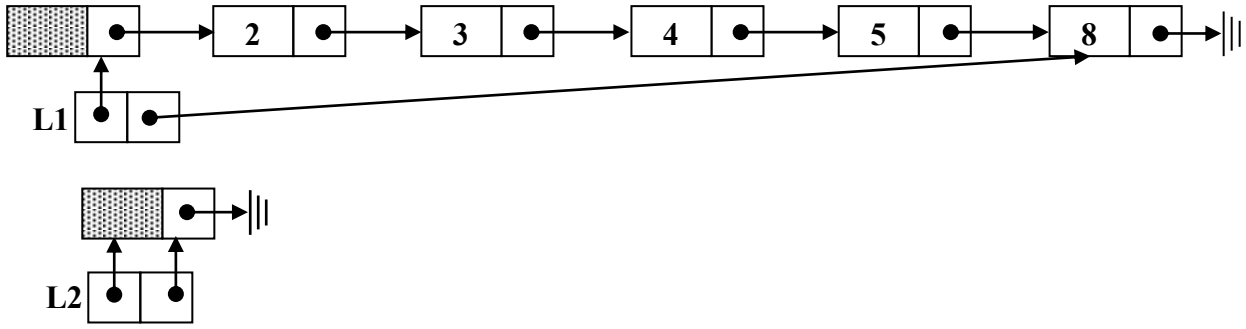
(iii) 1 point pour le dessin, 4 points pour la réalisation

Etat initial :





Etat final :




```
ModifierParDifSym(L1, L2) :                          {principe d'interclassement}
   AP1, AC1 : adCel                                  {pour le parcours de L1 ; AC1 = AP1↑.Suc}
   AP2, AC2 : adCel                                  {pour le parcours de L2 ; AC2 = AP2↑.Suc}

   AP1 ← L1.F ; AC1 ← AP1↑.Suc
   AP2 ← L2.F ; AC2 ← AP2↑.Suc
   tant que AC1≠Nil et AC2≠Nil
         selon AC1↑.E, AC2↑.E
            AC1↑.E < AC2↑.E :                        {AC1↑.E ∈ E1\E2, on avance dans L1}
                  AP1 ← AC1 ; AC1 ← AP1↑.Suc
            AC1↑.E > AC2↑.E :                        {AC2↑.E ∈ E2\E1, on le déplace dans L1}
                  {Déconnecter AC2}
                  AP2↑.Suc ← AC2↑.Suc
                  si AC2 = L2.Q alors L2.Q ← AP2     {cas du déplacement de la queue}
                  {Connecter AC2 entre AP1 et AC1}
                  AP1↑.Suc ← AC2 ; AC2↑.Suc ← AC1
                  {mise à jour de AP1 et AC2}
                  AP1 ← AC2 ; AC2 ← AP2↑.Suc
            AC1↑.E = AC2↑.E :                        {AC1↑.E ∈ E1∩E2, on supprime dans les deux listes}
                  {supprimer AC1 et AC2, cas particuliers des queues}
                  AP1↑.Suc ← AC1↑.Suc ; si AC1 = L1.Q alors L1.Q ← AP1
                  AP2↑.Suc ← AC2↑.Suc ; si AC2 = L2.Q alors L2.Q ← AP2
                  Libérer(AC1) ; Libérer(AC2)
                  {avancer dans les deux listes}
                  AC1 ← AP1↑.Suc ; AC2 ← AP2↑.Suc
   si AC2 ≠ Nil alors                                {AC1 = Nil, connecter toute la fin de L2}
      {connecter AC2 à la fin de la liste L1 et mettre à jour L1.Q}
      (L1.Q)↑.Suc ← AC2
      L1.Q ← L2.Q
      L2.Q ← L2.F
```