Correction de TD
Listes doublement chaînées

Ce document contient quelques éléments de correction pour les TD. Il n'est pas voué à être exhaustif et ne se substitue **pas** à la correction vue en TD.

1 Primitives

Anciennes (simplement chaînées) Une liste doublement chaînée est une liste simplement chaînée enrichie : on peut aller dans le sens inverse. On reprend donc les mêmes primitives que pour les simplement chaînées. Attention, on veut qu'elles permettent les mêmes opérations, mais pour des listes **doublement** chaînées ! L'implémentation ne sera donc pas la même.

```
list creerListe()
bool estVide(list L)
obj valeur(list L)
list suivant(list L)
list insererDebut(list L, obj x)
list supprimerDebut(list L)
list insererApres(list L, list curseur, obj x)
list supprimerApres(list L, list curseur)
bool estDernier(list L) #facultative
```

Notez que `estDernier` est facultative. En effet, pour une liste non vide, `estDernier`(L) = `estVide`(`suivant`(L)). On peut la prendre parmi les primitives puisqu'elle est en temps constant, `estVide` et `suivant` l'étant aussi.

Nouvelles (doublement chaînées) Il faut maintenant pouvoir utiliser les liens allant dans le sens inverse.

```
list precedent(list L)
list insererAvant(list L, list curseur, obj x) #facultative
list supprimerAvant(list L, list curseur) #facultative
bool estPremier(list L) #facultative
```

Notez que les trois dernières peuvent s'écrire à partir des autres. Cependant, vous avez vu en cours qu'il pouvait être bien plus pratique d'utiliser `insererAvant` et `supprimerAvant` plutôt que de les réecrire à partir des autres primitives.

Algorithm 1: insererAvant

Input: $L, curseur$ deux listes, x un objet

```
1 if estVide(L) ou estVide(precedent(L)) then
2 |   return insererEnTete(L, x)
3 end
4 else
5 |   return insererApres(L, precedent(curseur), x)
6 end
```

Si notre curseur est en tête de liste (voire qu'elle soit vide), on ne peut rien supprimer.

Algorithm 2: supprimerAvant

```
Input: L, curseur deux listes
if estVide(curseur) ou estVide(precedent(curseur)) then
    #ERROR
end
else
    return supprimerApres(L, precedent(curseur))
end
```

2 Affichage inversé d'une liste doublement chaînée

Pour la version itérative (Algorithme 3), l'idée de l'algorithme est de la parcourir jusqu'à la fin avec une première boucle. Ensuite, on revient en arrière en affichant les éléments un à un, ce qui permet de les avoir dans l'ordre inverse.

Algorithm 3: afficherInverseIter

```
Input: L une liste
curseur ← L
if non estVide(L) then
    while non estVide(suivant(curseur)) do
        curseur ← suivant(curseur)
    end
    while non estVide(curseur) do
        afficher(valeur(current))
        curseur ← precedent(curseur)
    end
end
```

Pour la version récursive (Algorithme 4), on reproduit le raisonnement classique pour prouver une récurrence :

Hérédité : on suppose qu'une propriété est valide pour n et on la prouve pour $n+1$. Ici, on suppose que `afficherInverseRec` affiche bien à l'envers une liste de taille n. Soit L une liste de taille $n+1$. Alors `suivant`(L) est taille n. Donc `afficherInverseRec(suivant`(L)`)` affiche bien à l'envers la liste `suivant`(L). Il reste à afficher `valeur`(L) pour afficher L entièrement. Il faut donc l'afficher après avoir affiché `suivant`(L).

Initialisation : ne pas oublier le cas de base ! Si la liste est vide, rien à afficher. Il suffit donc de rajouter un test, et de n'appliquer les étapes de l'hérédité que si la liste est non vide.

Algorithm 4: afficherInverseRec

```
Input: L une liste
if non estVide(L) then
    afficherInverseRec(suivant(L))
    afficher(valeur(L))
end
```

Exemple :

On prend la liste $L = [2, 4, 7, 6]$, et on appelle `afficherInverseRec`(L)

```
# 1er appel recursif :
afficherInverseRec([4,7,6])
afficher(2)

# 2eme appel recursif :
afficherInverseRec([7,6])
afficher(4)
afficher(2)

# 3eme appel recursif :
afficherInverseRec([6])
afficher(7)
afficher(4)
afficher(2)

# 4eme appel recursif :
afficherInverseRec([])
afficher(6)
afficher(7)
afficher(4)
afficher(2)

# 5eme appel recursif :
afficher(6)
afficher(7)
afficher(4)
afficher(2)
```

Complexités

Les deux versions sont linéaires (i.e. en $O(n)$):

- dans la version itérative, on fait deux parcours de la liste, un dans chaque sens. De plus, à chaque pas, c'est à dire pour passer d'un élément au suivant ou au précédent, on fait un nombre constant d'opérations élémentaires (primitives, tests, affichages). Il y a n étapes pour chaque parcours.
- dans la version récursive, pareil (la remontée récursive simulant le parcours inverse).

Si vous voulez donner une complexité plus précise, vous pouvez expliciter le nombre constant d'opérations élémentaires. Attention cependant à ce qu'une primitime peut en cacher deux autres, comme ce qu'on a vu avec `estDernier` au début du TD.

3 Copie de l'inverse d'une liste

Pour copier une liste à l'envers, l'algorithme est le même pour une liste simplement ou doublement chaînée (la différence est cachée dans les primitives). On se donne une liste L. On crée une liste vide *nouvelleListe* qui contiendra L inversée à la fin. Ensuite, on prend la tête de L et on la place à la tête de *nouvelleListe*. Puis on itère sur `suivant`(L). Vous pourrez trouver un exemple dans la Table 1.

Algorithm 5: Inverser

Input: L une liste

```
1 curseur ← L
2 nouvelleListe ← creerListe()
3 while non estVide(curseur) do
4 |   nouvelleListe ← insererDebut(nouvelleListe, valeur(curseur))
5 |   curseur ← suivant(curseur)
6 end
7 return nouvelleListe
```

L'algorithme 5 est linéaire puisqu'on parcourt la liste initiale en faisant à chaque étape un nombre constant d'opération.

Table 1: Exemple de copie de l'inverse de la liste simplement chaînée [a , b , c , d]

1ère étape	curseur	a → b → c → d
	nouvelle	
2ème étape	curseur	b → c → d
	nouvelle	a
3ème étape	curseur	c → d
	nouvelle	b → a
4ème étape	curseur	d
	nouvelle	c → b → a
5ème étape	curseur	
	nouvelle	d → c → b → a

4 Copie d'une liste à l'endroit

Afin de copier (à l'endroit) une liste, on va lire dans l'ordre les éléments de la liste initiale pour les placer en queue de la nouvelle liste. On propose deux méthodes :

- une version naïve. On a donné une fonction `insererEnQueue` pour les listes simplement chaînées. Cette dernière prend une liste et un objet en paramètre et renvoie la liste avec l'élément inséré en queue. Ici, on peut l'utiliser (en version doublement chaînée bien sûr). InsererEnQuue va reparcourir toute la liste à chaque appel. Sa complexité étant linéaire, la somme des appels est donc quadratique, i.e. en $O(n^2)$.

- Plutôt d'éviter de tout reparcourir à chaque fois, on peut garder un curseur sur la fin de la nouvelle liste et utiliser insererApres dessus. Le cas de base sur la liste vide requiert d'utiliser insererDebut. Cette distinction de cas correspond à celle faite dans insererEnQueue.

Remarque. *Afin de ne pas réécrire la même distinction de cas que dans insererEnQueue, on peut vouloir utiliser cette dernière, en lui donnant non pas la liste entière mais le curseur sur la fin. On aurait un appel en temps constant. D'un point de vue utilisateur, le problème est que cet appel renverrait juste la liste à partir du curseur, sans les cases avant, donc juste les deux dernières cases. Il* ***faut*** *donc lui donner en premier paramètre la liste entière. Vous avez ensuite pu remarquer qu'avec le type concret que l'on a choisi, donner le curseur à insererEnQueue ne pose pas de problème. Cela pourrait être le cas avec une autre implémentation.*

Algorithm 6: Copie avec insererEnQueue

```
   Input: L une liste
1  nList ← creerVide()
2  oCurseur ← L
3  while non estVide(oCurseur) do
4  |  nListe ← insererEnQueue(nListe, valeur(oCurseur))
5  |  oCurseur ← suivant(oCurseur)
6  end
7  return nListe
```

Algorithm 7: Copie

```
   Input: L une liste
1  nListe ← creerVide()
2  nCurseur ← nListe
3  oCurseur ← L
4  if non estVide(L) then
5  |  nListe ← insererDebut(nListe, valeur(L))
6  end
7  while non estVide(oCurseur) do
8  |  nListe ← insererApres(nListe, nCurseur, valeur(oCurseur))
9  |  nCurseur ← suivant(nCurseur)
10 |  oCurseur ← suivant(oCurseur)
11 end
12 return nListe
```

5 Echange de deux cases consécutives

On veut échanger deux éléments consécutifs d'une liste. On voudrait naturellement échanger les valeurs des deux cases correspondantes. Cependant, les primitives que l'on s'est donné ne le permettent pas : on ne peut pas modifier directement la valeur d'une case. Pour modifier une case, il est nécessaire de la supprimer et d'en insérer une nouvelle. Une solution alternative consiste donc à ajouter des primitives modifiant les valeurs dans les cases.

En paramètre, on se donne la liste entière, ainsi que les curseurs sur la première et la deuxième case à échanger. Le deuxième curseur peut-être retrouvé à partir du premier à l'aide de la primitive

`suivant`. On propose deux algorithmes, chacun centré sur un des deux curseurs.

Attention : vous supprimez une case dans les deux cas donc un des deux curseurs est invalidé ! De plus, l'autre curseur a changé de position dans la liste. Par exemple, dans l'algorithme 8, on supprime la deuxième case. Donc le deuxième curseur est invalidé et le premier se retrouve en deuxième position. C'est l'inverse pour l'algorithme 9.

Algorithm 8: Swap version 1

```
Input: L une liste, c1, c2 deux curseurs
L ← insererAvant(L, c1, valeur(c2))
L ← supprimerApres(L, c1)
return L
```

Algorithm 9: Swap version 2

```
Input: L une liste, c1, c2 deux curseurs
L ← insererApres(L, c2, valeur(c1))
L ← supprimerAvant(L, c2)
return L
```

6 Tri à bulles

Pour le tri à bulles, *curseur*1 représente le point de départ des inversions. On lance une chaîne d'inversion à partir de ce point de départ, puis on l'avance d'une case. Cette chaîne d'inversion inverse deux cases consécutives si elles sont inversées, puis avance d'une case. Pour inverser deux cases, on utilise ici la première version du `swap` ci-dessus, afin de ne pas invalider le premier curseur, qui se retrouve avancé d'une case. Il ne faut donc pas l'avancer dans ce cas !

Algorithm 10: Tri à bulles

```
Input: L une liste de réels
if estVide(L) then
    return L
end
curseur1 ← L
while non estVide(curseur1) do
    curseur2 ← curseur1
    while non estVide(suivant(curseur2)) do
        if valeur(curseur2) > valeur(suivant(curseur2)) then
            L ← swap1(L, curseur2, suivant(curseur2))
        end
        else
            curseur2 ← suivant(curseur2)
        end
    end
    curseur1 ← suivant(curseur1)
end
return L
```

7 Implémentation des primitives

Puisque les listes doublement chaînées sont des listes simplement chaînées avec un lien en plus pour retourner en arrière, on va reprendre l'implémentation des listes simplement chaînées et rajouter ces liens en sens inverse.

Pour rappel : le type *obj* est le type d'objet que contient la liste ; vous pouvez le remplacer par celui que vous voulez, *int* par exemple. Attention qu'avec le typedef, *list* est un pointeur vers une *struct list*. De plus, on se donne une fonction `error` qui gère les cas où les paramètres sont incorrects. Pour les fonctions `insererApres`, `supprimerApres`, `insererAvant` et `supprimerAvant`, on considère que le curseur donné en paramètre fait bien partie de la liste. On propose ici deux versions d'`insererAvant` et `supprimerAvant`, une à la main, l'autre utilisant `insererDebut`. Comme expliqué au début, on peut aussi proposer une troisième version en implémentant les algorithmes 1 et 2.

```
struct list{
  obj val;
  struct list * suiv;
  struct list * prev;
};

typedef struct list * list;

list creerList(){
  return NULL; //puisque la liste est vide
}

bool estVide(list L){
  return L==NULL;
}

list valeur(list L){
  if(estVide(L)) error();
  return L->val;
}

list suivant(list L){
  if(estVide(L)) error();
  return L->suiv;
}

list precedent(list L){
  if(estVide(L)) error();
  return L->prev;
}

list insererDebut(list L, obj x){
  new = list_alloc(); //malloc(sizeof(struct list));
  new->val = x;
  new->suiv = L;
```

```
  new->prev = NULL;
  if(!estVide(L)) L->prev = new;
  return new;
}

list supprimerDebut(list L){
  if(estVide(L)) error();
  newList = L->suiv;
  if(!estVide(newList)) newList->prev = NULL;
  free(L);
  return newList;
}

//version 1 (on rajoute tous les liens a la main)
list insererApres(list L, list curseur, obj x){
  if(estVide(L) || estVide(curseur)) error();
  list newList = list_alloc();
  list nextNewList = curseur->suiv;
  newList->val = x;
  newList->suiv = nextNewList;
  newList->prev = curseur;
  curseur->suiv = newList;
  if(!estVide(nextNewList)) nextNewList->prev = newList;
  return L;
}

//version 2 (on utilise insererDebut)
list insererApres(list L, list curseur, obj x){
  if(estVide(L) || estVide(curseur)) error();
  curseur->suiv = insererDebut(curseur->suiv,x);
  (curseur->suiv)->prev = curseur;
  return L;
}

list supprimerApres(list L, list curseur){
  if(estVide(L) || estVide(curseur) || estVide(curseur->suiv)) error();
  newList = curseur->suiv->suiv;
  free(curseur->suiv);
  curseur->suiv = newList;
  if(!estVide(newList)) newList->prev = curseur;
  return L;
}

//version 1 (on rajoute tous les liens a la main)
list insererAvant(list L, list curseur, obj x){
  list newElem = list_alloc();
  list prevNewElem = NULL;
  if(!estVide(curseur)) prevNewElem = curseur->prev;
  newElem->val = x;
  newElem->suiv = curseur;
```

```
  newElem->prev = prevNewElem;
  if(!estVide(curseur)) curseur->prev = newElem;
  if(!estVide(prevNewElem)) prevNewElem->suiv = newElem;
  return L;
}


//version 2 (on utilise insererDebut)
list insererAvant(list L, list curseur, obj x){
  if(estVide(L) || estVide(curseur)) error();
  list oldPrev = NULL;
  if(!estVide(curseur)) oldPrev = curseur->prev;
  list newElem = insererDebut(curseur,x);
  if(!estVide(oldPrev)){
    newElem->prev = oldPrev;
    oldPrev->suiv = newElem;
  }
  return L;
}

list supprimerAvant(list L, list curseur){
  if(estVide(L) || estVide(curseur) || estVide(curseur->prev)) error();
  newList = curseur->prev->prev;
  free(curseur->prev);
  curseur->prev = newList;
  if(!estVide(newList)) newList->suiv = curseur;
  return L;
}
```
