Correction de TD
Listes simplement 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

```
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)
```

2 Recherche dans une liste

On veut savoir si un élément x est dans une liste simplement chaînée L. On parcourt les éléments de L. Si on trouve x, on renvoie Vrai, sinon c'est que l'on atteint la fin de la liste, dans quel cas on renvoie Faux.

On propose une version itérative (Algorithme 1) qui fait avancer un curseur le long de la liste L. Ensuite, une version récursive (Algorithme 2) avance dans la liste en faisant appel à elle même sur `suivant`(L). Les deux versions sont linéaires car elles peuvent être amenées à parcourir toute la liste.

Algorithm 1: RechercheIter

```
Input: L une liste, x un objet
curseur ← L
while non estVide(curseur) do
    if valeur(curseur) = x then
        return Vrai
    end
    else
        curseur ←suivant(curseur)
    end
end
return Faux
```

Algorithm 2: RechercheRec

```
Input: L une liste, x un objet
1 if estVide(L) then
2 |  return Faux
3 else if valeur(L) = x then
4 |  return Vrai
5 else
6 |  return Recherche(suivant(L))
7 return Faux
```

3 Position d'un élément donné

On veut connaître la position d'un élément x dans la liste. On propose une version itérative, puis récursive, et enfin récursive terminale. Une position étant forcément un entier positif, on choisit de renvoyer -1 si x n'est pas dans la liste.

Algorithm 3: PositionIter

```
Input: L une liste, x un objet
1 position ← 0
2 curseur ← L
3 while non estVide(curseur) do
4 |  if valeur(curseur) = x then
5 |  |  return position
6 |  end
7 |  curseur ←suivant(curseur)
8 |  position ← position + 1
9 end
10 return −1
```

Pour la version récursive, on utilise propose deux versions. La première renvoie $-\infty$ si la liste ne contient pas x ce qui permet de compenser les $+1$ lors de la remontée des appels récursifs.

Algorithm 4: PositionRec

```
Input: L une liste, x un objet
1 if estVide(L) then
2 |  return −∞
3 else if valeur(L) = x then
4 |  return 0
5 else
6 |  return PositionRec(suivant(L)) + 1
```

Une deuxième traite le cas -1 à part lors de la remontée (et peut donc renvoyer -1 si la liste ne contient pas x).

Algorithm 5: PositionRec

```
Input: L une liste, x un objet
1  if estVide(L) then
2  |  return −1
3  else if valeur(L) = x then
4  |  return 0
5  else
6  |  posSuivant ←PositionRec(suivant(L))
7  |  if posSuivant = −1 then
8  |  |  return −1
9  |  end
10 |  else
11 |  |  return posSuivant + 1
12 |  end
```

Enfin, la version récursive terminale fait passer la position courante par paramètre, que l'on met à 0 pour le premier appel récursif dans l'algorithme 7.

Algorithm 6: PositionRecTerminale

```
Input: L une liste, x un objet, i indice courant
1  if estVide(L) then
2  |  return −1
3  else if valeur(L) = x then
4  |  return i
5  else
6  |  return PositionRecTerminale(suivant(L), x, i + 1)
```

Algorithm 7: Position

```
Input: L une liste, x un objet, i indice courant
1  return PositionRecTerminale(L, x, 0)
```

4 Insertion en queue de liste

On veut insérer un élément x à la fin d'une liste L. On doit la parcourir intégralement. Attention qu'il faut que la boucle s'arrête avant d'arriver sur la liste vide donc le test `estVide` doit être sur `suivant`(*curseur*). De plus, faites attention au cas particulier de la liste vide.

Algorithm 8: InsertionQueue

Input: L une liste, x un objet

```
1  if estVide(L) then
2  |  return insererDebut(L, x)
3  end
4  else
5  |  curseur ← L
6  |  while non estVide(suivant(curseur)) do
7  |  |  curseur ← suivant(curseur)
8  |  end
9  |  return insertionApres(L, curseur, x)
10 end
```

5 Implémentation des primitives

On implémente les primitives vues en cours. 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` et `supprimerApres`, on considère que le curseur donné en paramètre fait bien partie de la liste.

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

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

```
list supprimerDebut(list L){
  if(estVide(L)) error();
  newList = L->suiv;
  free(L);
  return newList;
}
list insererApres(list L, list curseur, obj x){
  if(estVide(L) || estVide(curseur)) error();
  curseur->suiv = insererDebut(curseur->suiv,x);
  return L;
}
list supprimerApres(list L, list curseur){
  if(estVide(L) || estVide(curseur) || estVide(curseur->suiv)) error();
  newList = L->suiv->suiv;
  free(curseur->suiv);
  curseur->suiv = newList;
  return L;
}
```