Exercice 2.3 Extraire deux listes à partir d'une liste ***

Soit une liste de nombres relatifs. Écrire un algorithme permettant de séparer cette liste en deux listes : la première ne comportant que des entiers positifs ou nuls, et la seconde ne comportant que des nombres négatifs.

Exercice 2.4 Permuter deux places d'une liste ***

Écrire un algorithme sur une liste simplement chaînée qui échange les positions des nœuds données par deux pointeurs `t` et `v`.

Exercice 2.5 Supprimer des éléments **

Soit L une liste chaînée. Écrire trois algorithmes tels que :

- dans le premier on supprime toutes les occurrences d'un élément donné `x` ;
- dans le deuxième, on ne laisse que les *k* premières occurrences de cet élément et on supprime les suivantes ;
- dans le troisième, pour chaque élément de la liste, on ne laisse que sa première occurrence.

Concevez le second algorithme en adaptant le premier, puis concevez le dernier en exploitant le second.

Exercice 2.6 Inverser une liste **

Écrire un algorithme qui inverse une liste chaînée sans recopier ses éléments. Réaliser une version itérative et une autre récursive.

Exercice 2.7 Construire une liste à partir d'une source de données *

Concevoir un premier algorithme qui construit une liste chaînée linéaire à partir d'un tableau, et un second qui duplique une liste chaînée existante.

Exercice 2.8 Refermer une liste sur elle-même *

Concevoir un algorithme qui transforme une liste simplement chaînée linéaire en liste simplement chaînée circulaire.

Exercice 2.9 Effectuer et retourner deux calculs sur une liste **

Concevoir un algorithme qui calcule et retourne respectivement le produit des éléments positifs et celui des éléments négatifs d'une liste simplement chaînée d'entiers.

Exercice 2.10 Couper une liste en deux **

Concevoir un algorithme qui sépare une liste simplement chaînée d'entiers en deux listes :

- A – à partir d'un pointeur sur le maillon qui devient tête de la seconde liste de sortie ;
- B – juste devant le premier maillon contenant une valeur donnée *x* ;
- C – à partir de la $k^{ième}$ position (c'est-à-dire *k* maillons dans la première liste de sortie et le reste dans la seconde) ;
- D – juste devant le maillon contenant la valeur minimale de la liste ;
- E – telles que les deux listes de sortie aient même longueur *n*, ou que la première soit de longueur *n+1* et la seconde de longueur *n*.

Exercice 2.11 Supprimer un sous-ensemble d'une liste *

Concevoir un algorithme qui supprime :

- A – un maillon sur deux, en commençant par la tête de liste,
- B – les maillons contenant des éléments impairs,
- C – les maillons contenant des éléments pairs,
- D – les maillons contenant des éléments supérieurs à un seuil donné,
- E – les maillons contenant des éléments inférieurs à un seuil donné,

d'une liste simplement chaînée (d'entiers).

Problème

Problème 2.1 Saisir, enregistrer puis évaluer un polynôme***

Enregistrer les coefficients et les exposants d'un polynôme dans une liste chaînée. Évaluer ce polynôme pour une valeur donnée de x.

Utiliser au moins deux modules dans ce programme :

- l'un pour construire la liste sur la base des coefficients et des exposants, qui sont saisis au clavier ;
- et le second afin d'évaluer le polynôme pour la valeur x, également saisie au clavier.

Corrigés des exercices et des problèmes

En préambule

Pour la réalisation en C de tous les algorithmes spécifiés ci-dessous, on définit la structure de liste chaînée suivante dont on précisera au cas pas cas, le type `<element>`. Les structures utilisées pour les spécifications des algorithmes portent le même nom et sont construites de la même manière.

```
typedef struct listNode
{
  <element> content;
  struct place *succ;  // listes simplement chaînées
  struct place *prev;  // listes doublement chaînées
} listNode;
typedef listNode* list;
typedef list* plist;
#define ERR -1  // code d'erreur (lisibilité des exemples)
```

Les quelques implantations C de méthodes élémentaires de manipulation de la liste chaînée conformes au contrat du type abstrait, pourront s'avérer utiles dans l'écriture de vos programmes d'implantation C de vos algorithmes, notamment en épurant votre code d'une trop grande quantité

de '->', '*' et '&' qui nuisent à leur lisibilité. Vous pourrez ainsi vous rapprocher de la vision algorithmique du « qu'est-ce que ça fait ? » plutôt que de la vision programmatique du « comment ça se fait ? ».

```
<element> content(list l) {return (l->content);}

int isempty(list l) {return (l == NULL);}

list succ(list l) {return l->succ;}

int islast(list l) {return isempty(succ(l));}
```

pour les solutions algorithmiques, les fonctions sont les suivantes :

```
FONCTION content(l : list)
DEBUT
  RETOURNER l⟶content
FIN
FONCTION isempty(l : list)
VAR vide : booléen
DEBUT
  SI(l=NULL)ALORS
    vide ⟵ vrai
  SINON
    vide ⟵ faux
  FINSI
  RETOURNER vide
FIN
FONCTION succ(l : list) : list
DEBUT
  RETOURNER l⟶succ
FIN
FONCTION islast(l : list)
DEBUT
  RETOURNER isempty(succ(l))
FIN
```

Enfin, le constructeur suivant pourra vous aider à simplifier vos écritures pour l'allocation d'un nouveau nœud de liste :

```
list newSingletonList(int content)
{
  list l = malloc(sizeof(listNode));  // allocation mémoire
  l->content = content;               // affectation du contenu
  l->succ = NULL;
  return l;
}
```

Et son équivalent en pseudo langage :

```
FONCTION newSingletonList(content : entier) : list
VAR nouveau : list
DEBUT
  RESERVER(nouveau)
  nouveau⟶content ⟵ content
  nouveau⟶succ ⟵ NULL
  RETOURNER nouveau
FIN
```

Corrigés des exercices

Exercice 2.1 Rechercher l'élément maximal d'une liste

Spécification de l'algorithme itératif

C'est une simple adaptation de l'algorithme « recherche (b) » du rappel de cours vu plus haut.
Entrée : liste de positifs (non vide)
Sortie : un positif

```
FONCTION maxOf(l: liste<positif> ≠ ∅): positif
VAR max: positif, p: place
DEBUT
  p ⟵ tête(l)
  max ⟵ contenu(p)
  TANTQUE ¬ dernier(p) FAIRE
    p ⟵ succ(p)
    SI contenu(p) > max ALORS
      max ⟵ contenu(p)
    FINSI
  FAIT
  RETOURNER max
FIN
```

Réalisation en C de l'algorithme itératif

Ici, le champ content est de type `unsigned`.

```
int maxOf(list<unsigned> l)
{
  if (l == NULL) return ERR;
  int max = l->content;    // valeur en première place
  while (l->succ != NULL)
  {
    l = l->succ;  // itération de la liste
    if (l->content > max) max = l->contenu;
  }
  return max;
}
```

Et réécrite avec les méthodes utilitaires données en introduction :

```
int maxOf(list<unsigned> l)
{
  if (isempty(l)) return ERR;
  int max = content(l);  // valeur en première place
  while (! islast(l))
  {
    l = succ(l);  // itération de la liste
    if (content(l) > max) max = content(l);
  }
  return max;
}
```

Spécification de l'algorithme récursif

Entrée : liste de positifs (non vide)
Sortie : un positif

```
FONCTION maxOf(l: liste<positif> ≠ ∅): positif
VAR max, contenu: positif
DEBUT
  RETOURNER maxOfRec(tête(l))
FIN
FONCTION maxOfRec(p: place): positif
VAR max, contenu: positif
DEBUT
  contenu ← contenu(p)
  SI dernier(p) ALORS
    RETOURNER contenu
```

```
  FINSI
  max ← maxOfRec(succ(p))
  SI contenu > max ALORS
    RETOURNER contenu
  FINSI
  RETOURNER max
FIN
```

Réalisation en C de l'algorithme récursif

```
int maxOfRec(list<unsigned> l)
{
  if (l == NULL) return ERR;
  int content = l->content;
  if (l->succ == NULL) return content;
  int max = maxOfRec(l->succ);
  if (content > max) return content;
  return max;
}
```

Et réécrite avec les méthodes utilitaires données en introduction :

```
int maxOfRec(list<unsigned> l)
{
  if (isempty(l)) return ERR;
  int content = content(l);
  if (islast(l)) return content;
  int max = maxOfRec(succ(l));
  if (content > max) return content;
  return max;
}
```

Exercice 2.2 Concaténer deux listes

Étude

La contrainte d'entiers positifs du premier exercice disparaît, mais pas le cas des listes vides.

Nous avons quatre cas à considérer :

- l_1 et l_2 vides,
- l_1 vide mais pas l_2,
- l_2 vide mais pas l_1,
- ni l_1, ni l_2 vides.

Si l_2 est vide, on laisse l_1 inchangée. Si l_1 est vide mais pas l_2, on affecte directement l_2 à l_1. Si ni l_1 ni l_2 ne sont vides, on parcourt l_1 jusqu'à sa dernière cellule et on y rattache l_2.

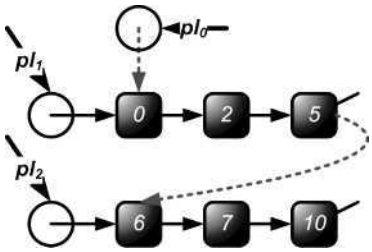

Figure 2.5 **Concaténation standard**

Enfin, si on va un peu trop vite dans l'étude, on peut oublier le cas $l_1 = l_2$ (c'est-à-dire même liste et non pas, listes dont les valeurs sont égales), cas particulier qui croise le premier et le dernier des quatre cas. Ce cas donne une bonne illustration de la différence entre copies et références.

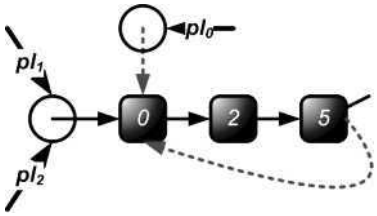

Figure 2.6 **Concaténation circulaire**

Ce dernier cas d'utilisation de la fonction de concaténation revient, si on l'autorise, à transformer la liste chaînée linéaire en liste chaînée circulaire.

Autorisons-le, en notant que nous avons alors un constructeur de liste circulaire par fermeture d'une liste chaînée linéaire (nous utiliserons cette facilité dans le problème 3.1 (problème de Joseph)).

Pour la concaténation, une approche itérative s'impose d'elle-même, un procédé récursif n'ayant pas grand intérêt pour une simple jonction.

Nous travaillons par référence, c'est-à-dire qu'il n'y a pas de copie des deux listes (on « rattache » les deux listes).

Spécification de l'algorithme

Entrées modifiées : deux listes d'entiers

```
FONCTION concat(l1, l2: liste<entier>)
VAR p1, p2 : place
DEBUT
  SI ¬ estvide(l1) ALORS
    SI estvide(l2) ALORS
      l1 ⟵ l2
    SINON
```

```
      p1 ⟵ tête(l1)
      TANTQUE ¬ dernier(p1) FAIRE
        p1 ⟵ succ(p1)
      FAIT
      succ(p1) ⟵ tête(l2)
    FINSI
  FINSI
FIN
```

Réalisation en C

```
void concat(list *pl1, list *pl2)
{
  if (*pl2 == NULL) return;  // ras, l1 soit nulle ou non
  if (*pl1 == NULL)          // affectation directe de l2 à l1
  {
    *pl1 = *pl2;
    return;
  }
  // sinon :
  list l = *pl1;
  while (l->succ != NULL) l = l->succ;  // itération l1
  l->succ = *pl2;                       // rattachement de l2
}
```

Et réécrite avec les méthodes utilitaires données en introduction :

```
void concat(list *pl1, list *pl2)
{
  if (isempty(*pl2)) return; // ras, l1 soit nulle ou non
  if (isempty(*pl1))         // affectation directe de l2 à l1
  {
    *pl1 = *pl2;
    return;
  }
  // sinon :
  list l = *pl1;
  while ((! islast(l)) l = succ(l);  // itération l1
  l->succ = *pl2;                    // rattachement de l2
}
```

Exercice 2.3 Extraire deux listes à partir d'une liste

Étude

Plusieurs approches sont possibles selon que :

- On travaille à partir de la liste de départ en la préservant (copie des éléments), ou bien en recyclant ses cellules (réutilisation des éléments).
- On réalise une seule fonction complexe qui produit simultanément les deux listes, ou bien deux fonctions simples spécialisées appelées successivement.
- On élabore une structure complexe nouvelle pour combiner les deux listes produites dans l'unique sortie de la fonction, ou bien on passe deux pointeurs respectivement sur les deux listes (pointeurs sur pointeurs) initialement non initialisés (paramètres inout).

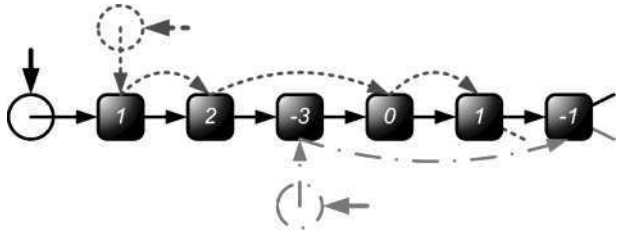

Figure 2.7 **Déconstruction d'une liste avec reconstruction sous forme de deux listes**

Nous nous proposons d'utiliser le même algorithme itératif pour réaliser l'opération, que ce soit par copie ou par référence. Cet algorithme consiste à parcourir la liste de départ et à aiguiller chaque nouvel élément rencontré vers l'une ou l'autre des deux listes selon sa valeur, en les complétant ainsi progressivement jusqu'à épuisement de la liste de départ. Il existe d'autres solutions.

Spécification de l'algorithme

Entrées modifiées :

- La liste d'entier initiale
- Les deux listes d'entiers produites

```
FONCTION separer(l0, l1, l2: liste<entier>)
VAR p0, p1, p2: place
DEBUT
  SI ¬ estvide(l0) ALORS
    p0 ← tête(l0)
    SI contenu(p0) ⩾ 0 ALORS
      p1 ← tête(l1) ← tête(l0)
    SINON
    p2 ← tête(l2) ← tête(l0)
    FINSI
    TANTQUE ¬ dernier(p0) FAIRE
```

```
      p0 ← succ(p0)
      SI contenu(p0) ⩾ 0 ALORS
        SI estvide(l1) ALORS
          p1 ← tête(l1) ← p0
        SINON
          p1 ← succ(p1) ← p0
        FINSI
      SINON
        SI estvide(l2) ALORS
          p2 ← tête(l2) ← p0
        SINON
          p2 ← succ(p2) ← p0
        FINSI
      FINSI
    FAIT
  FINSI
FIN
```

Réalisation en C

```
void separate(list *pl0, list *pl1, list *pl2)
{
  liste l0 = *pl0, l1 = NULL, l2 = NULL;
  if (l0 == NULL) return;     // si l0 vide ne rien faire
  if (l0->content >= 0) l1 = *pl1 = *pl0; // tête de l1
  else l2 = *pl2 = *pl0;                  // tête de l2
  while (l0->succ != NULL)  // itération de l0
  {
    l0 = l0->succ;          // itération de l0
    if (l0->content >= 0)   // transfert vers l1
    {
      if (l1 == NULL) *pl1 = l1 = l0;  // tête de l1
      else                 // au-delà de la tête
      {
        l1->succ = l0;     // complète l1
        l1 = l1->succ;     // itération de l1
      }
    }
    else                   // transfert vers l2
    {
      if (l2 == NULL) *pl2 = l2 = l0;  // tête de l2
      else                 // au-delà de la tête
      {
        l2->succ = l0;     // complète l2
```

```
            l2 = l2->succ;      // itération de l1
          }
        }
      }
      // important pour la clôture des listes l1 et l2 :
      l1->succ = NULL;          // finalisation de l1
      l2->succ = NULL;          // finalisation de l2
      *pl0 = NULL;              // libération de l0 (opt.)
}
```

Exercice 2.4 Permuter deux places d'une liste

Étude

Il convient de prévoir un code d'erreur pour les cas suivants :

- Liste vide mais *t* ou *v* non null.
- Liste non vide, mais *t* ou *v* pointant nul ou pointant un élément n'appartenant pas à la liste.
- La fonction retournera –1 en cas d'erreur, 0 sinon.

Pour le reste, l'idée est d'identifier les deux éléments, et d'interchanger (en utilisant une variable temporaire) les pointeurs de leurs prédécesseurs respectifs et de leurs successeurs respectifs.

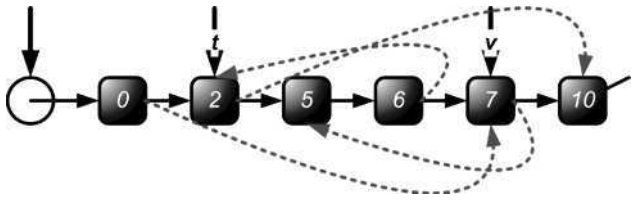

Figure 2.8 **Permutation : le cas général…**

Les cas suivants feront l'objet d'un traitement spécial :

- Les pointeurs *t* et *v* sont égaux : c'est l'opération identité, il n'y a rien à faire sinon ne rien faire.
- L'un de deux pointeurs ou les deux sont respectivement tête ou queue (début ou fin de liste).

Les deux pointeurs identifient des places contigües (cf. figure 2.9 ce cas est « dangereux » si traité comme précédemment).

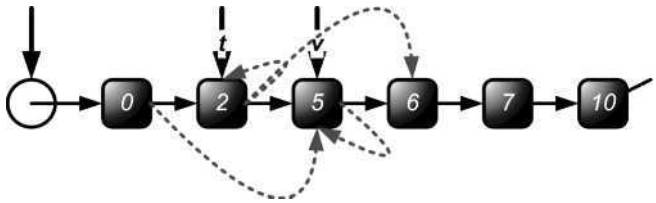

Figure 2.9 **… avec son exception dangereuse en l'absence de traitement spécifique**

Spécification de l'algorithme

```
FONCTION swap(modif pl : list, modif t, v : place) : entier
VAR resultat : entier ; term_l, term_t, term_v, l : list
VAR prec_t, succ_t, prec_v, succ_v : list
DEBUT
  SI (pl = NULL) ALORS
    SI (t = NULL) ET (v = NULL) ALORS
      RETOURNER 0
    SINON
      RETOURNER -1
    FINSI
  SINON
    SI (t = NULL) OU (v = NULL) ALORS
      RETOURNER -1
    FINSI
      term_l ⟵ dernierElt(*pl)
      term_t ⟵ dernierElt(t)
      term_v ⟵ dernierElt(v)
    SI (¬ ((term_t = term_l) ET (term_v = term_l))) ALORS
      RETOURNER -1
    FINSI
  FINSI
  SI (t = v) ALORS
    RETOURNER 0
  FINSI
  l = pl;
  trouverPrecEtSucc(l, t, &prec_t, &succ_t)
  trouverPrecEtSucc(l, v, &prec_v, &succ_v)
  SI ((prec_v = t) OU (prec_t = v)) ALORS
    SI prec_v = t ALORS
      SI prec_t = NULL ALORS
        *pl ⟵ v
      SINON
        prec_t⟶succ ⟵ v
      FINSI
      v⟶succ ⟵ t
      t⟶succ ⟵ succ_v
    SINON
      SI prec_v = NULL ALORS
        *pl ⟵ t
      SINON
        prec_v⟶succ ⟵ t
      FINSI
```

```
      t→succ ← v
      v→succ ← succ_t
    FINSI
  SINON
    SI ((prec_t = NULL) OU (prec_v = NULL) ALORS
      SI prec_t = NULL ALORS
        *pl ← v
        prec_v→succ ← t
      FINSI
      SI prec_v = NULL ALORS
        *pl ← t
        prec_t→succ ← v
      FINSI
    SINON
      prec_v ← t
      prec_t ← v
    FINSI
    t→succ ← succ_v
    v→succ ← succ_t
  FINSI
FIN
```

Réalisation en C de la fonction principale

```
// Permutation de deux éléments
int swap(list *pl, place *t, place *v)
{
  if (*pl == NULL)
  {
    if ((t == NULL) && (v == NULL)) return 0; // tout va bien !!
    else return -1;
  }
  else
  {
    if ((t == NULL) || (v == NULL)) return -1; // non consistant !!
    // Maintenant test d'appartenance
    // on utilise un truc : même terminal !
    list term_l = lastElement(*pl);
    list term_t = lastElement(t);
    list term_v = lastElement(v);
    if (!((term_t == term_l) && (term_v == term_l))) return -1;
  }
  // préconditions vérifiées : on peut commencer
  if (t == v) return 0; // identité -- tout va bien !!
```

```
// note : le cas du singleton est implicitement déjà traité :
// on travaille maintenant sur une liste l multiple
// sinon, cas général :
list l = *pl;
list prec_t, succ_t;
getPrecAndSucc(l, t, &prec_t, &succ_t);
l = *pl;
list prec_v, succ_v;
getPrecAndSucc(l, v, &prec_v, &succ_v);
// cas de contigüité ->[v]->[t]-> ou ->[t]->[v]->
if ((prec_v == t) || (prec_t == v))
{
  // cas : ->[t]->[v]-> => ->[v]->[t]->
  if (prec_v == t)
  {
    // cas .->[t] => .->[v] vs autres cas
    if (prec_t == NULL)
    {
      printf("case : .->[t]->[v]-> => .->[v]->[t]->\n");
      *pl = v;
    }
    else
    {
      printf("case : [..]->[t]->[v]-> => [..]->[v]->[t]->\n");
      prec_t->succ = v;
    }
    v->succ = t;
    t->succ = succ_v;
  }
  // cas : ->[v]->[t]-> => ->[t]->[v]->
  else
  {
    // cas .->[v] => .->[t] vs autres cas
    if (prec_v == NULL)
    {
      printf("case : .->[v]->[t]-> => .->[t]->[v]->\n");
      *pl = t;
    }
    else
    {
      printf("case : [..]->[v]->[t]-> => [..]->[t]->[v]->\n");
      prec_v->succ = t;
    }
```

```
      t->succ = v;
      v->succ = succ_t;
    }
  }
  // cas ->[v]->..->[t]-> ou ->[t]->..->[v]->
  else
  {
    // cas .->[t] => .->[v] vs autres cas
    if ((prec_t == NULL) || (prec_v == NULL))
    {
      if (prec_t == NULL)
      {
        printf("case : .->[t]->[..]->[v]-> => .->[v]->[..]->[t]->\n");
        *pl = v;
        prec_v->succ = t;
      }
      if (prec_v == NULL)
      {
        printf("case : .->[v]->[..]->[t]-> => .->[t]->[..]->[v]->\n");
        *pl = t;
        prec_t->succ = v;
      }
    }
    else
    {
      printf("case : [..]->[t/v]->[..]->[v/t]-> =>
[..]->[v/t]->[..]->[t/v]->\n");
      prec_v = t;
      prec_t = v;
    }
    // et dans tous les cas :
    t->succ = succ_v;
    v->succ = succ_t;
  }
  return 0; // that's all folks !!
}
```

Fonction déléguée de récupération du prédécesseur et du successeur d'un nœud

```
// (fonction sécurisée par l'appelant :
// paramètres supposés cohérents entre eux)
int getPrecAndSucc(list l, list t, list* prec, list* succ)
{
  if (l == t)       // cas de t en tête
```

```
  {
    *prec = NULL;
    *succ = l->succ;
    return;
  }
  else          // cas de t en milieu ou en queue
  {
    while (l->succ != NULL)
    {
      if (l->succ == t)
      {
        *prec = l;
        *succ = l->succ->succ;
        return;
      }
      else l = l->succ;
    }
  }
}
```

Exercice 2.5 Supprimer des éléments

Étude

Nous proposons de réaliser le second algorithme en premier lieu, puis de construire le premier par un appel récursif du second avec pour condition d'arrêt, un dernier appel qui laisse la liste inchangée.

Pour supprimer un élément, il s'agit de repérer l'élément, et de court-circuiter celui-ci en reliant le pointeur *succ* de son prédécesseur directement sur son successeur.

Algorithme récursif

- *Retrait de toutes les occurrences d'un élément*

Entrée : x, l'entier dont on supprime toutes les occurrences dans la liste.

Entrée modifié : la liste dont on supprime des occurrences de x.

Sortie : un code de statut d'exécution

- `SUPPRESSION (1)` : il y a eu au moins un retrait effectif.
- `IDENTITE (0)` : il n'y a eu aucun retrait car la liste ne comporte aucun x.

Cette fonction utilise un type énuméré (énumération) qui est un type défini par les valeurs que peuvent prendre les variables de ce type. Ainsi, dans la fonction suivante, les variables *statut* et *st2* ne peuvent prendre que les valeurs *SUPPRESSION* ou *IDENTITE*.

```
FONCTION supprimerOccurrences(*l: liste<entier>, x: entier)
VAR statut, st2: énumération {SUPPRESSION, IDENTITE}; e: entier
DEBUT
```

```
  statut ← IDENTITE
  SI ¬ estvide(l) ALORS
    SI premier(l) = x ALORS
      l ← fin(l)
      statut ← SUPPRESSION
      SI ¬ estvide(l) ALORS
         supprimerOccurrences(l, x)
      FINSI
    SINON
      e ← premier(l)
       l ← fin(l)
      SI ¬ estvide(l) ALORS
         st2 ← supprimerOccurrences(l, x)
         SI statut = IDENTITE ALORS
           statut ← st2
         FINSI
      FINSI
       l ← cons(e, l)
     FINSI
   FINSI
  RETOURNER statut
FIN
```

- *Retrait de toutes les occurrences d'un élément après la $k^{ième}$*

Très semblable au précédent.
Entrées : x, l'entier dont on supprime toutes les occurrences sauf les k premières dans la liste, k le nombre d'occurrences qu'on laisse sauves.
Entrées modifiées : la liste dont on supprime certaines occurrences de x.
Sortie : un code de statut d'exécution

- `SUPPRESSION (1)` : il y a eu au moins un retrait effectif.
- `IDENTITE (0)` : il n'y a eu aucun retrait car la liste ne comporte aucun x.

```
FONCTION supprOccurAprèsKième(*l: liste<entier>, x: entier, k: entier)
VAR statut, st2: énumération {SUPPRESSION, IDENTITE}; e: entier;
DEBUT
  statut ← IDENTITE
  SI ¬ estvide(l) ALORS
    SI premier(l) = x ALORS
      SI k < 1 ALORS
         l ← fin(l)
        statut ← SUPPRESSION
       SINON
        e ← premier(l)
         l ← fin(l)
```

```
          k ← k -- 1
        FINSI
        SI ¬ estvide(l) ALORS
           supprOccurAprèsKième(l, x, k)
        FINSI
        SI k > 0 ALORS
           l ← cons(e, l)
        FINSI
      SINON
        e ← premier(l)
        l ← fin(l)
        SI ¬ estvide(l) ALORS
          st2 ← supprOccurAprèsKième(l, x, k)
          SI statut = IDENTITE ALORS
            statut ← st2
          FINSI
        FINSI
        l ← cons(e, l)
      FINSI
    FINSI
  RETOURNER statut
FIN
```

Exercice 2.6 Inverser une liste

Étude

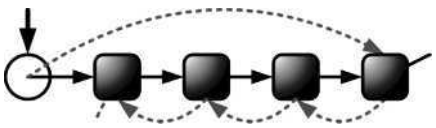

Figure 2.10 **Inversion d'une liste**

Deux approches, l'une itérative, l'autre récursive.

Algorithme itératif

```
FONCTION reverse(*pl : list)
VAR prev, curr, succ : list
DEBUT
  SI *pl=NULL ALORS   // cas de la liste vide
    RETOURNER
  FINSI
  SI (*pl) →succ = NULL ALORS // cas de la liste à un seul élément
```

```
    RETOURNER
  FINSI
  prev ⟵ *pl
  curr ⟵ prev⟶succ
  SI curr⟶succ = NULL ALORS
    curr⟶succ ⟵ prev
    prev⟶succ ⟵ NULL
    *pl ⟵ curr
    RETOURNER
  SINON
    prev⟶succ ⟵ NULL
  FINSI
  TANTQUE curr⟶succ ≠ NULL FAIRE
    succ ⟵ curr⟶succ
    curr⟶succ ⟵ prev
    prev ⟵ curr
    curr ⟵ succ
  FAIT
  curr⟶succ ⟵ prev
  *pl ⟵ curr
FIN
// Inversion d'une liste (utilisée pour l'exo 1.4 (normalisation))
void reverse(list *pl)
{
  if (*pl == NULL) return;          // liste vide
  if ((*pl)->succ == NULL) return;  // singleton
  list prev = *pl;                  // tête
  list curr = prev->succ;           // 2d élément
  if (curr->succ == NULL)           // si cas à 2 éléments
  {
    curr->succ = prev;              // pointage du 2d élt sur le 1er
    prev->succ = NULL;              // et du 1er sur NULL (queue)
    *pl = curr;                     // le 2d devient tête
    return;                         // terminé
  }
  else                              // sinon, au moins 3 elts :
  {
    prev->succ = NULL;              // pointage du 1er->NULL (queue)
  }
  list succ;
  while (curr->succ != NULL)        // Expl. sur 1ère itération
  {
    succ = curr->succ;              // sauv. du succ (le 3ème élt)
```

```
    curr->succ = prev;              // pointage du 2d elt sur le 1er
    prev = curr;                    // le nouveau prev c'est le 2d
    curr = succ;                    // le nouveau curr, c'est le 3ème
  }
  // si par exemple le 3ème n'a pas de successeur :
  // c'est la queue (pas d'autre passage dans la boucle) :
  curr->succ = prev;                // pointage du 3ème sur le 2d
  *pl = curr;                       // le 3ème (dernier) devient tête
}
```

Algorithme récursif

```
FONCTION esreverse(*pl : list) : list*
VAR head : list
DEBUT
  SI *pl=NULL ALORS   // cas de la liste vide
    RETOURNER
  FINSI
  SI (*pl) →succ = NULL ALORS // cas de la liste à un seul élément
    RETOURNER
  FINSI
  head ← *pl
  *esreverse(&(head→succ)) → succ ← head
  *pl ← head→succ
  head→succ ← NULL
  RETOURNER &head
FIN
// reverse2 en méthode récursive
list * esreverse(list* pl)
{
  if (*pl == NULL) return;              // cas liste vide
  if ((*pl)->succ == NULL) return pl;  // cas singleton
  // autres cas :
  list head = *pl;
  (*esreverse(&(head->succ)))->succ = head;
  *pl = head->succ;
  head->succ = NULL;
  return &head;
}
```