Langage C avancé : Séance 7

Types de données utilisateurs : listes, arbres

Fonctions récursives

Marc TAJCHMAN
@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Types Union

Union

Le but est d'utiliser des variables qui peuvent contenir des valeurs de type différents.
On ne peut pas définir un « type universel ».

Par contre, on peut définir un pointeur universel : void *

C propose de définir un type qui peut représenter un ensemble fini de types existants, grâce au mot-clef union:

```
union _nombre
{
  int n;
  float f;
  double d;
};
typedef union _nombre nombre;
```

Les variables de type nombre peuvent contenir une valeur entière, réelle simple précision ou réelle double précision.
Mais **un seul** de ces 3 types (et pas de chaine de caractère, pas de pointeur, …)

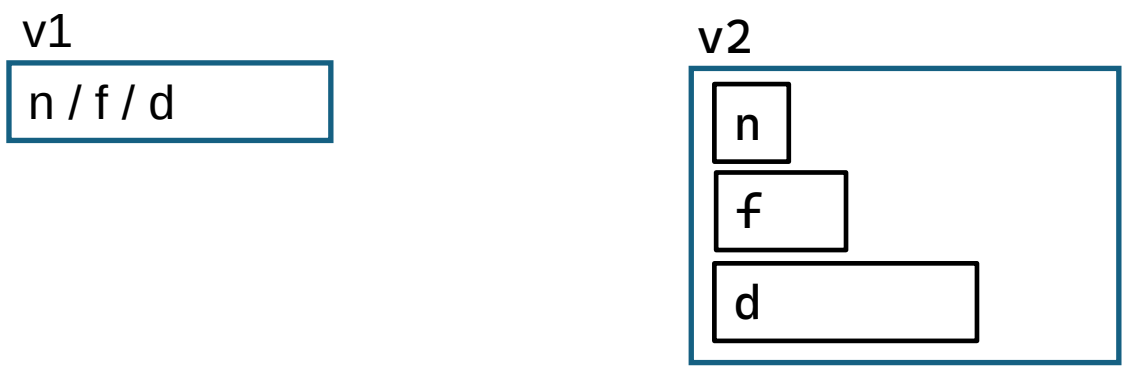
Différence entre union et struct

Les 2 types utilisateurs ont une déclaration et une syntaxe très proches :

```c
union _nbre1 {
  int n;
  float f;
  double d;
};
typedef union _nbre1 Nombre1;
Nombre1 v1;
```

```c
struct _nbre2 {
  int n;
  float f;
  double d;
};
typedef struct _nbre2 Nombre2;
Nombre2 v2;
```

Mais les variables de type struct et union sont organisées différemment en mémoire:



Différence entre union et struct (2)

Une variable de type Nombre1, défini en utilisant union:

- Occupe une zone mémoire dont la taille est celle du plus grande sous-type (ici sizeof(double));
- Contient un int ou un float ou un double.

Une variable de type Nombre2, défini en utilisant struct:

- Occupe une zone mémoire dont la taille est la somme des tailles de tous les sous-type (ici sizeof(double) + sizeof(float) + sizeof(int));
- Contient un int et un float et un double

Exemple d'utilisation

```c
union _nbre {
  int n;
  float f;
  double d;
  const char *s;
};
typedef union _nbre Nombre;
Nombre v;

v.n = 1;
v.f = 1.4; /* ecrase v.n */

printf("v.f = %g\n", v.f);
printf("v.n = %d\n", v.n);
```

```c
void affiche(Nombre x)
{
  printf("%d", x.n); // ???
  printf("%f", x.f); // ???
  printf("%g", x.d); // ???
}
```

Dans la fonction affiche, on ne sait pas ce qui est contenu dans le paramètre x.

Donc il faut passer une autre information en paramètre : le type exact contenu dans x

En pratique

En pratique, on met l'union dans un type struct avec un indicateur de type:

```c
#define Entier 1
#define Float  2
#define Double 3
#define Chaine 4
struct _nombre {
  int type;
  union _valeur {
    int n;
    float f;
    double d;
    const char *s;
  } v;
};
typedef struct _nombre nombre;
```

Exemple d'une fonction qui affiche une variable de ce type (tester en compilant union2.c) :

```c
void affiche(nombre x) {
  if (x.type == Entier)
    printf("%d", x.v.n);
  else if (x.type == Float)
    printf("%f", x.v.f);
  else if (x.type == Double)
    printf("%lf", x.v.d);
  else if (x.type == Chaine)
    printf("%s", x.v.s);
  printf("\n");
}
```

En pratique (2)

On peut (un peu) simplifier en utilisant une union anonyme dans struct:

```c
#define Entier 1
#define Float  2
#define Double 3
#define Chaine 4
struct _nombre {
  int type;
  union {
    int n;
    float f;
    double d;
    const char *s;
  };
};
typedef struct _nombre nombre;
```

Exemple d'une fonction qui affiche une variable de ce type (tester en compilant union2.c) :

```c
void affiche(nombre x) {
  if (x.type == Entier)
    printf("%d", x.n);
  else if (x.type == Float)
    printf("%f", x.f);
  else if (x.type == Double)
    printf("%lf", x.d);
  else if (x.type == Chaine)
    printf("%s", x.s);
  printf("\n");
}
```

Type vecteur (suite et fin)

Types de données définis par l'utilisateur

Les types C standards ne sont pas toujours suffisants (on a vu l'exemple des vecteurs « pointeur C » où la taille du vecteur n'accompagne pas les composantes).

On a donc défini un type utilisateur « vecteur » avec struct (qui définit l'organisation des données dans le type utilisateur) et le mot-clef typedef (qui facilite l'écriture du type utilisateur):

```
struct _vecteur {
  int n;
  double * c;
  const char *nom;
};
typedef struct _vecteur vecteur;
```

Ici les composantes sont de type réel (double) mais peuvent être de n'importe quel type standard ou utilisateur
On a ajouté un pointeur sur le nom du vecteur dans la struct (peut être utile)

Utilisation de variables de type utilisateur

```
Vecteur V;
V.Nom = "V";
V.n = 10;
V.c = (double *) malloc(V.n*sizeof(double);

for (i=0; i<V.n; i++)
  V.c[i] = 2.5*i;

free(V.c);
```

Définition d'une variable de type vecteur et réservation de la mémoire interne dynamique éventuelle

Utilisation de la variable V

Libération de la mémoire réservée à la définition de la variable V

Examiner et compiler le code dans le répertoire vecteurs/v1

La syntaxe des types utilisateurs en C n'est pas très élégante, d'autres langages de programmation (C++, python, …) permettent de définir des types utilisateurs avec une syntaxe plus « naturelle »

Fonctions utilitaires pour un type utilisateur

On définit souvent des fonctions qui:

- préparent une variable de type utilisateur (constructeur)
- nettoient la mémoire utilisée par une variable de type utilisateur (destructeur)
- font des opérations « standards » avec des variables de type utilisateur

Ces fonctions sont intéressantes parce que le code est plus facile à lire, mais aussi parce que cela diminue la possibilité d’erreurs.
Par contre, il faut les écrire soigneusement.

Exemple de fonctions utilitaires pour le type vecteur (version 1)

```
Vecteur construitVecteur(int n, const char *s)
{
  Vecteur v;
  v.nom = s;
  v.n = n;
  v.c = (double *) malloc(sizeof(double) *n);
  return v;
}
```

Fonction
« constructeur »

```
void detruitVecteur(Vecteur v)
{
  v.n = 0; v.nom = "";
  free(v.c);
  v.c = NULL;
}
```

Fonction
« destructeur »

Examiner les fichiers dans le répertoire vecteurs/v2. Le type vecteur est défini dans le fichier vecteur.h et les fonctions utilitaires dans le fichier vecteur.c

Exemple de fonctions utilitaires pour le type vecteur (version 1)

La compilation dans le répertoire vecteurs/v2 produit 2 fichiers exécutables: install/exec2a et install/exec2b

Exécuter install/exec2a et install/exec2b.

L'exécution de install/exec2b produit des erreurs.
Essayer de les expliquer.

De plus, cette version des constructeur/destructeur a aussi des désavantages.
Quels sont-ils ?

Dans le fichier main2b.c

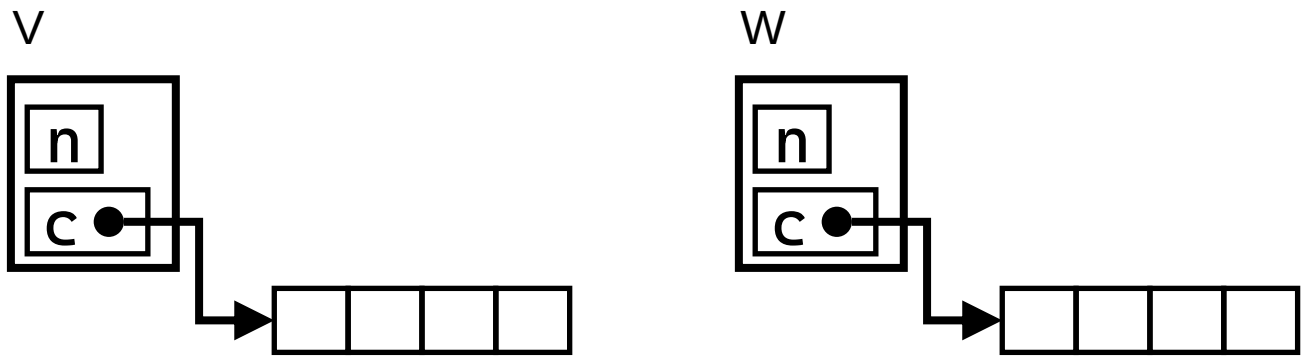
```
Vecteur V = construitVecteur(10);
Vecteur W = construitVecteur(10);

…
W = V;
W.c[5] = -56.0;

…
detruitVecteur(V);
detruitVecteur(W);
```

Erreurs dans le fichier main2b.c

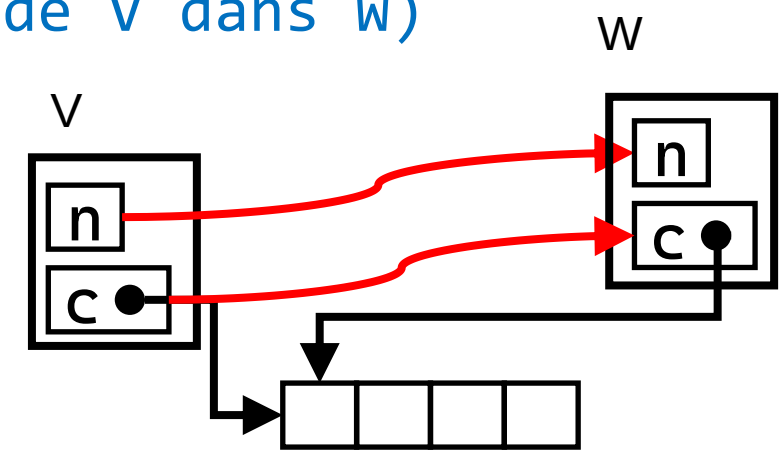
L'instruction W = V, où V et W sont des structures, copie les composantes de la structure V dans W.

Mais cette copie est une copie de surface (**shallow copy**).

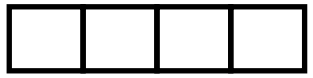
Avant W = V:



W = V (copie de V dans W)



Zone mémoire réservée pour V par le code, avec 2 pointeurs qui contiennent son adresse (V.c et W.c)



Zone mémoire réservée pour W par le code mais inutilisable (« décrochée » de W, plus aucun pointeur ne contient son adresse)

Erreurs dans le fichier main2b.c

```
W.c[2] = -56.0;
```

Modifie à la fois W.c[2] et V.c[2] puisque W.c et V.c contiennent la même adresse. D'où l'erreur dans les valeurs affichées.

```
detruitVecteur(V);
```

Rend au système la mémoire réservée pour V

```
detruitVecteur(W);
```

W contient un pointeur vers la mémoire réservée par V (à cause de W = V). Donc la mémoire réservée par V est rendue 2 x au système.

Ce qui provoque une erreur d'exécution et l'exécution du code s'arrête brutalement (dans les meilleurs des cas).

Désavantages dans l'écriture des fonctions utilitaires (1)

```
Vecteur construitVecteur(int n, const char *s)
{
  Vecteur v;
  v.n = n; v.nom = s;
  v.c = (double *) malloc(sizeof(double) *n);
  return v;
}
```

Dans cette fonction, on crée 2 variables temporaires de type Vecteur:

- la variable locale v dans la fonction construitVecteur et
- la valeur de retour de cette fonction qui est une copie de v
- ensuite cette valeur de retour est copiée dans la variable vecteur du programme principal

Cela ne pose pas de problème parce qu'on alloue qu'une seule zone mémoire qui sera libérée une seule fois (dans detruitVecteur).

Mais on crée des variables temporaires et on fait des copies inutiles

Désavantages dans l'écriture des fonctions utilitaires (2)

```
void detruitVecteur(Vecteur vl)
{
  vl.n = 0; v.nom = "";
  free(vl.c);
  vl.c = NULL;
}
```

Dans cette fonction, on crée une copie locale vl de type Vecteur:

- le paramètre local vl dans la fonction detruitVecteur
- cette variable locale rend la zone mémoire qui contient ses coefficients au système (free)
- le contenu de la variable locale vl est mis à zéro

Cela ne pose pas de problème parce que la mémoire réservée par le vecteur du programme principal, est rendue au système par la variable locale.

Mais quand on sort de la fonction, le vecteur v a un pointeur qui n'a pas changé et désigne une zone mémoire qui ne lui est pas affectée. Potentiellement, cela peut provoquer des erreurs plus tard.

Exemple de fonctions utilitaires pour le type vecteur (version 2)

```
void construitVecteur(Vecteur * v, int n)
{
  v->n = n;
  v->c = (double *) malloc(sizeof(double) *n);
}
```

Fonction « constructeur »

```
void detruitVecteur(Vecteur *v)
{
  v->n = 0;
  free(v->c);
  v->c = NULL;
}
```

Fonction « destructeur »

Dans cette version, c'est un pointeur sur vecteur qui est passé en paramètre. On a des copies de pointeurs, ce qui coute beaucoup moins de temps que des copies de vecteurs. De plus on met bien à zéro le vecteur passé en paramètre à detruitVecteur.

Exemple de fonctions utilitaires pour le type vecteur (version 2)

A la place de W = V, il faut utiliser une fonction utilitaire de copie dont le code est écrit ci-dessous

```
void copieVecteurs(Vecteur * w, const Vecteur *v)
{
  if (w->n != v->n) {
    detruitVecteur(w);
    construitVecteur(w, v->n);
  }
  memcpy(w->c, v->c, v->n * sizeof(double));
}
```

Fonction « copie »

Ce type de fonction de copie effectue une copie complète (y compris de la mémoire dynamique).

On parle aussi de « **deep copy** ».

Dans le fichier main2c.c

```
Vecteur V, W;
construitVecteur(&V, 10);
construitVecteur(&W, 10);
…
copieVecteur(&W, &V);
W->c[5] = -56.0;
…
detruitVecteur(&V);
detruitVecteur(&W);
```

Cette version ne contient pas les erreurs de la version précédente mais utilise beaucoup de pointeurs.

Evolution du temps de calcul quand la taille des structures ou du problème varie.
On parle parfois de l’évaluation de la complexité du code : notation O(…).

Notation O (1)

C'est une indication de l'ordre de grandeur du temps mis pour réaliser une opération ou plusieurs.

Exemple 1: `v.c[k] = 1.2` où v est une variable de type vecteur, k est un entier entre 0 et n (n est la taille du vecteur).

On peut décomposer cette instruction en plusieurs parties:

```
double *p = v.c;     // pointeur sur la première composante
p = p + k;           // décalage de p de k * sizeof(double) octets
*p = 1.2;            // on rentre la valeur dans la
                     // composante pointée par p
```

Le temps d'exécution de l'instruction est la somme des temps de chacune des parties (création/initialisation/destruction d'un pointeur, décalage du pointeur : 1 addition et 1 multiplication d'entier, copie d'une valeur double).

Il ne dépend pas de n, la taille du vecteur.
On dit que l'instruction est en **O(1)**.

Notation O (2)

2ème exemple: recherche d'un minimum dans un vecteur

Deux algorithmes, parmi d'autres :

```
if (v.n > 0) {
  vmin = v.c[0];
  for (i=1; i<v.n; i++)
    if (v.c[i] < vmin) vmin = v.c[i];
}
```

```
#include <float.h>

vmin = DBL_MAX;
for (i=0; i<v.n; i++)
  if (v.c[i] < vmin) vmin = v.c[i];
```

Chaque instruction, prise séparément, est en O(1), mais l'ensemble de chaque algorithme contient C_1 n + C_0 instructions simples (C_1 et C_0 sont des constantes) où n est la taille du vecteur.

Pour n grand, le nombre d'instructions simples (et donc le temps de calcul) est (à peu près) proportionnel à n.
On dit que l'algorithme est en **O(n)**.

Notation O (3)

3ème exemple: multiplication de matrices w = u * v, où u, v, w sont des matrices, u de taille (N x M), v de taille (M x P) et w de taille (N x P) (pour que a multiplication ait un sens).

Pour chaque coefficient de w: $w_{ij} = \sum_{k=0}^{k=M-1} u_{ik} v_{kj}$ pour 0≤i<N, 0≤j<P

Un algorithme, parmi d'autres, est:

```
for (i=0; i < N; i++)
  for (j=0; j < P; j++) {
    double s = 0.0;
    for (k=0; k < M; k++)
      s += u.c[i*M + k] * v.c[k*P + j];
    w.c[i*P + j] = s;
  }
```

Le nombre d'instructions simples est $C_3n^3 + C_2n^2 + C_1n + C_0$. Quand n est grand, le terme le plus important est proportionnel à n^3.

On dit que l'algorithme est en **O(N x M x P)** si les matrices sont rectangulaires ou **O(n^3)** si les matrices sont carrées de taille n x n.

Notation O (4)

En fonction de l'algorithme considéré, l'exposant k dans $O(n^k)$ peut être évalué en comptant:

- Toutes les instructions
- Les opérations arithmétiques sur des entiers et des réels
- Les opérations seulement entre des entiers
- Les opérations seulement entre des réels
- Les lectures et/ou les écritures en mémoire de travail
- Les lectures et/ou les écritures dans des fichiers
- La taille de la zone mémoire nécessaire

Chaque fois que la complexité d'un code est fournie, il faut aussi préciser la méthode de calcul.

Opérations bien adaptées et moins bien adaptées sur des vecteurs

Accès aux composantes d’un vecteur dans un ordre quelconque

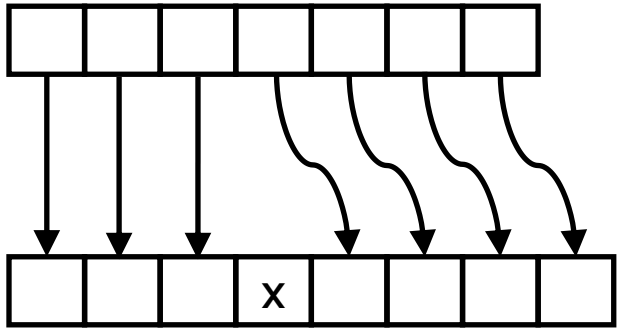
L’utilisation ou la modification d’une composante d’indice donné d’un vecteur est efficace (c’est une opération en O(1))

Faire cette opération k fois dans un ordre quelconque, a une complexité d’ordre O(k). L’opération ne demande pas de zone mémoire supplémentaire.

Un vecteur est bien adapté à ce type d’algorithme.

Insertion d'une nouvelle composante dans un vecteur

Par contre l'insertion d'une nouvelle composante est moins adaptée.
Principe (insertion de la valeur x à la 4ème position) :



Vecteur avant insertion

Vecteur après insertion

Augmenter la taille d'un vecteur n'est pas immédiat parce que le vecteur peut être coincé entre des zones mémoire déjà utilisées par d'autres variables.
Donc il faut:

- Demander au système une nouvelle zone mémoire de taille n+1
- Recopier les composantes existantes à la bonne place dans la nouvelle zone
- Mettre la valeur de la nouvelle composante dans la nouvelle zone
- Rendre au système l'ancienne zone mémoire des composantes

Insertion d'une nouvelle composante dans un vecteur

Un exemple d'algorithme pour insérer une valeur à la $k^{ème}$ position:

```
void insere(Vecteur *v, int k, double x) {
  /* v : vecteur de taille n */
  int i, n_old = v->n, n_new = n_old+1;
  double *c_old = v->c;
  v->c = NULL;
  construitVecteur(v, n_new, v->name);

  for (i=0; i<k; i++)
    v->c[i] = c_old[i];
  for(i=k; i<n_old; i++)
    v->c[i+1] = c_old[i];
  v->c[k] = x;
}
```

Construit un vecteur avec n+1 composantes mais garde une copie des composantes existantes (complexité O(1), mais O(n) en mémoire supplémentaire)

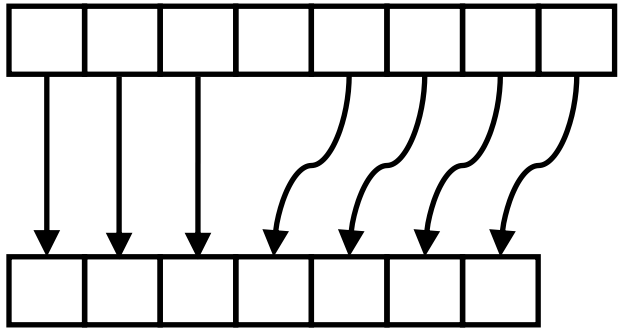
Recopie les anciennes composantes en gardant une place libre pour la valeur à insérer (complexité O(n))
Remarque: on peut remplacer les boucles par deux appels à memcpy (qui prennent moins de temps mais qui sont aussi en O(n))

Faire k insertions demande O(n+k) mémoires supplémentaires et a une complexité en O(nk).

Il manque une instruction pour que l'algorithme soit « propre », laquelle ?

Suppression d'une composante dans un vecteur

La situation plus simple pour les suppressions.
Principe (suppression de la 4ème composante) :



On n'a pas la nécessité de changer la taille du vecteur, puisque cette taille diminue (et donc pas besoin de mémoire supplémentaire).
On aura des (petites) zones mémoires inutilisées (c'est un inconvénient mineur).

La complexité en nombre d'opérations est en O(n) pour déplacer les composantes après la composante supprimée.

Suppression d’une composante dans un vecteur

Un exemple d’algorithme pour supprimer la $k^{ème}$ composante:

```
void supprime(Vecteur *v, int k)
{
  int i;
  for(i=k; i<v->n-1; i++)
    v->c[i] = v->c[i+1];
  v->n -= 1;
}
```

Déplace les composantes, après la composante à supprimer, d’une position vers la gauche (complexité O(n))

Capacité des vecteurs

Pour pouvoir faire plus simplement à la fois des insertions et suppressions dans un vecteur, on distingue les notions de taille (taille courante) et de capacité (taille maximale) dans la structure vecteur:

```
struct _vecteur
{
  const char *name;
  int n;
  int n_max;
  double *c;
};
typedef struct _vecteur Vecteur;
```

Capacité des vecteurs (2)

Avec des règles (exemple) :

- Si le vecteur est presque plein (n > 0.75 * n_max), on augmente la capacité (par ex. n_max → n_max*2)
- Si le vecteur utilise peu de composantes par rapport à la capacité (par ex., n < n_max/2) on diminue la capacité (par ex. n_max → n_max/2).

Ces règles sont un exemple, ceux qui développent les structures vecteur font en général beaucoup de tests pour trouver les règles les plus optimales pour les algorithmes utilisés (en temps calcul et en occupation mémoire).

Ne pas oublier de réallouer les composantes quand on change la capacité !!

Capacité des vecteurs (3)

Le répertoire vecteurs/capacite contient une définition de vecteur avec les indicateur de taille et de capacité.

Examiner les fichiers sources, les compiler et les exécuter.

Types liste

Définitions

Une liste permet de représenter un ensemble d'éléments, avec un ordre donné entre les éléments et un (ou des) sens de parcours entre les éléments bien défini. On appelle parfois les éléments des « nœuds ».

Pour utiliser une liste, on doit pouvoir au moins :

- Créer une liste vide
- Insérer un nouvel élément dans une liste
- Supprimer un élément existant dans une liste
- Accéder au début de la liste (si elle est non vide)
- Passer d'un élément au suivant dans la liste
- Tester si on se trouve à la fin de la liste

En fonction des besoins, il faudra peut-être pouvoir:

- Accéder au dernier élément de la liste
- Passer d'un élément au précédent dans la liste
- Tester si on se trouve en début de liste

Comparaison des listes et vecteurs

- Un vecteur a pour principale fonctionnalité l'accès facile et rapide à chacune de ses composantes, dans un ordre quelconque (avec un indice entier).

  Par contre, les opérations d'insertion, suppression, déplacement de composantes sont moins « naturelles » et peuvent avoir une complexité non négligeable.

- Une liste d'éléments est conçue pour rendre rapide les opérations d'insertion, suppression, déplacement dans la liste.

  Par contre, il y a peu de façons naturelle de se déplacer dans la liste (une ou deux en général)

Types liste chainée

Type nœud

La première étape est de définir un type pour représenter un nœud de la liste :

- il contiendra l'information utile (la valeur entière),
- il permettra d'accéder au nœud suivant pendant un parcours de la liste

```
struct _node
{
  int value;
  struct _node *next;
};
typedef struct _node Node;
```

Remarquer qu'on utilise la structure _node à un endroit où cette structure n'est pas complètement définie, cela ne pose pas de problème parce qu'on utilise seulement un pointeur sur _node (voir l'exemple lists/list1/main.c). C'est un exemple de définition récursive : une structure contient un pointeur sur une structure de même type.

Dans la structure _node, l'information utile est dans « value », « next » est un pointeur qui permet de passer au nœud suivant dans la liste.

Création/destruction de nœud

Il sera peut-être utile de définir des fonctions pour créer/détruire des nœuds:

```
Node* creeNode(int v) {
  Node* newNode = (Node*) malloc(sizeof(Node));
  newNode→value = data;
  newNode→next = NULL;
  return newNode;
}
```

```
void detruitNode(Node ** n) {
  if (*n) {
    free(*n);
    *n = NULL;
}
```

Type liste en C

On veut un type liste qui puisse représenter une collection de plusieurs nœuds (y compris 0 nœuds).

Le plus simple est de définir une liste comme un pointeur sur un nœud: `typedef Node * List;`

```
List L = NULL;

Node * n1 = (Node *) malloc(sizeof(Node));
n1->value = 1; n1->next = NULL;

L = n1;

Node * n2 = (Node *) malloc(sizeof(Node));
n2->value = 2;  n2->next = NULL;

Node * start = L;
start->next = n2;
```

Liste vide

Nouveau nœud

La liste contient un nœud

2ème nouveau nœud

Ajout du 2ème nœud à la suite du 1er dans la liste

Insertion d’un nœud dans une liste

L’insertion consiste en une manipulation de pointeurs.

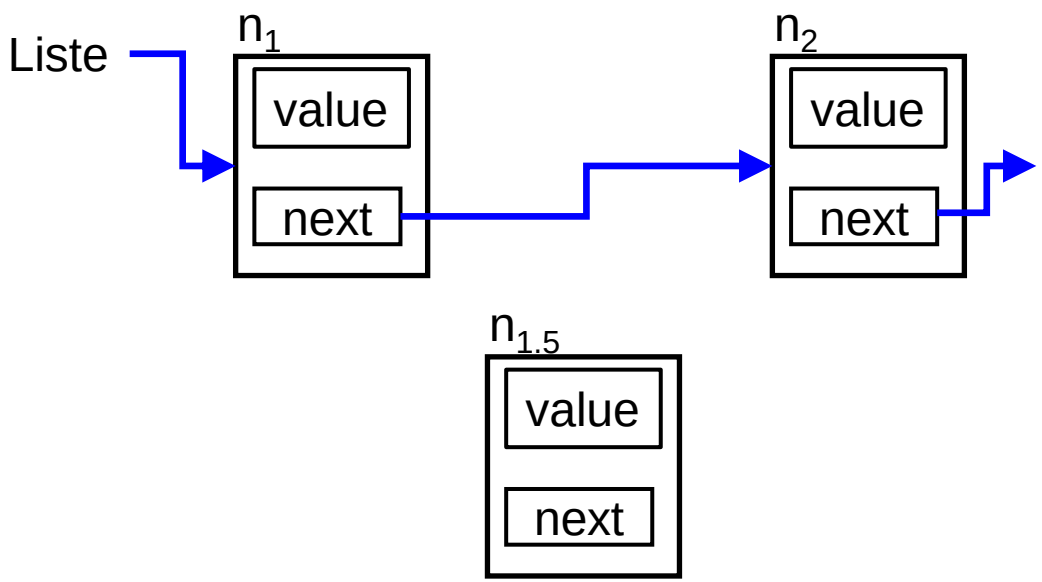
Il faut considérer tous les cas possibles:

- Insertion d’un nœud à l’intérieur d’une liste non vide
- Insertion d’un nœud en début de liste
- Insertion d’un nœud en fin de liste
- Insertion dans une liste vide

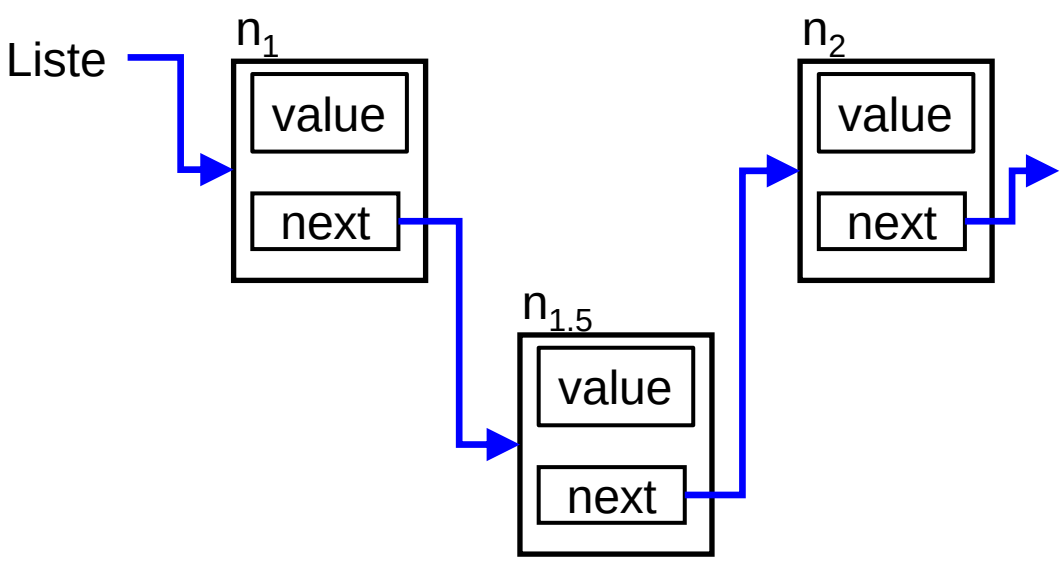
Insertion dans une liste

On considère dans un premier temps l'insertion d'un nouveau nœud $n_{1.5}$ entre 2 nœuds n_1 et n_2 qui se suivent :

Avant insertion :



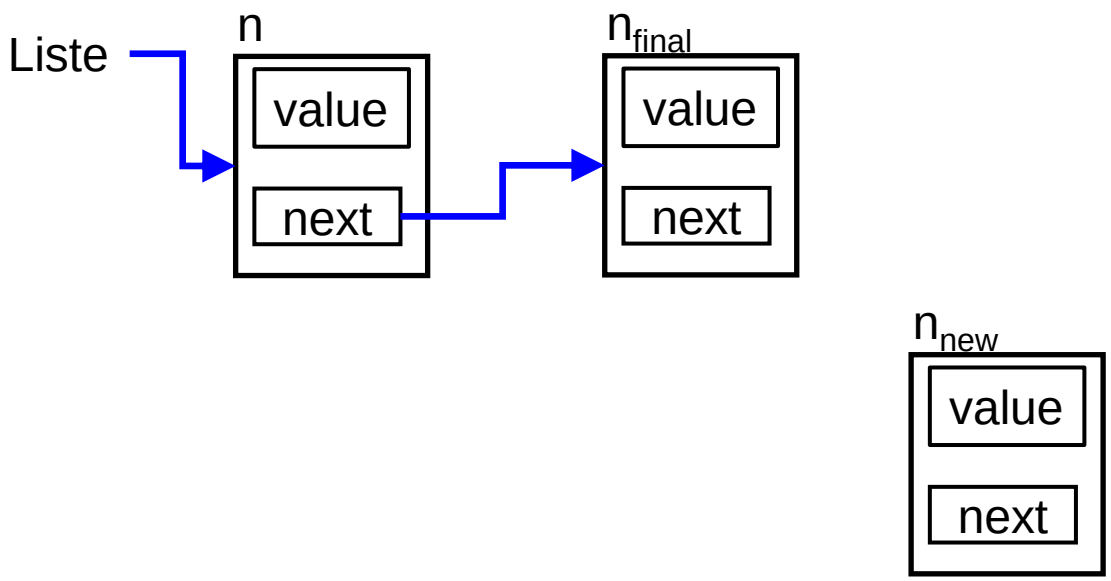
Après insertion :



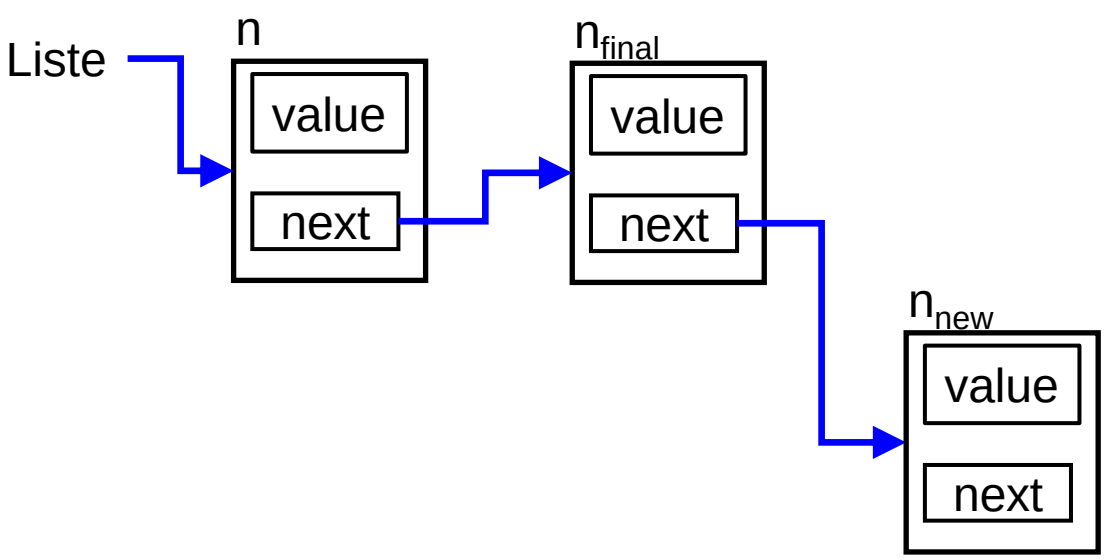
Insertion dans une liste (2)

On considère ensuite l'insertion d'un nouveau nœud n_{new} après le dernier nœud n_{final} d'une liste :

Avant insertion :



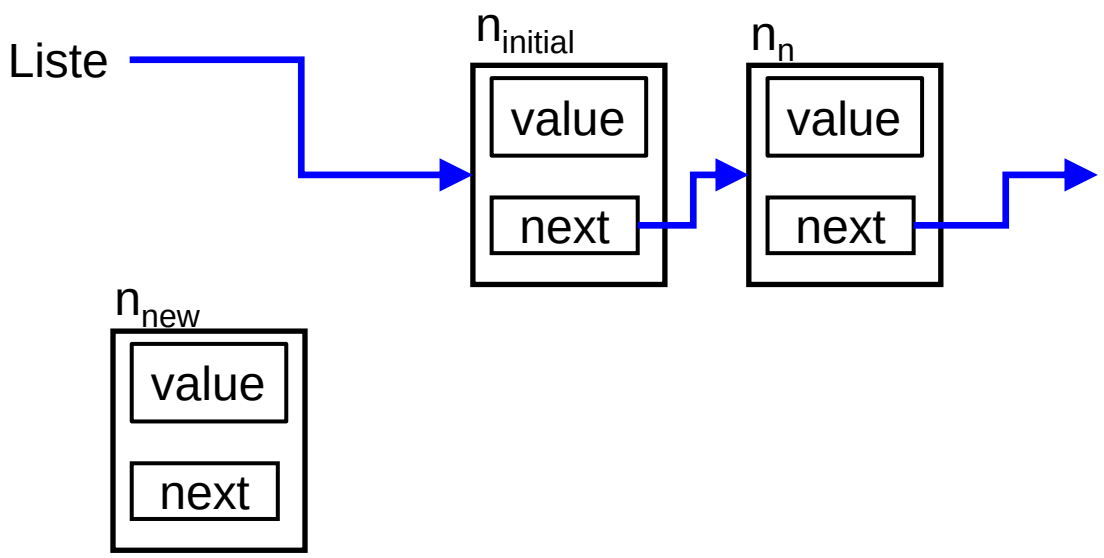
Après insertion :



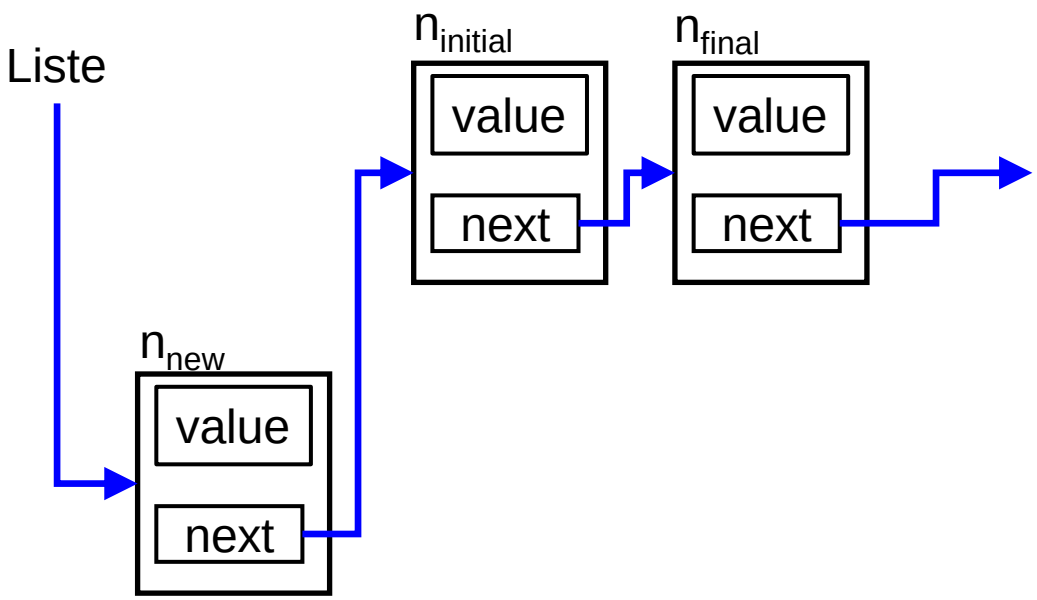
Insertion dans une liste (3)

On considère enfin l'insertion d'un nouveau nœud n_{new} au debut d'une liste non vide (premier nœud $n_{initial}$) :

Avant insertion :



Après insertion :



Insertion dans une liste (4)

Il faudra considérer le cas de l’insertion d’un nœud dans une liste vide (c’est simple mais il ne faut pas oublier de le coder).

La plupart du temps, on le fait en testant si la liste est vide au début de l’insertion.

Insertion dans une liste (5)

Le plus simple (pour l'utilisateur du type liste) est de fournir une ou plusieurs fonctions d'insertion, par exemple:

(exemple inspiré des listes standards C++)

```
void push_front(List *L, int value);
void push_back(List *L, int value);
void insert_after(Node *N, int value);
```

Ajout en début de liste

Ajout en fin de liste

Ajout à l'intérieur d'une liste (après le nœud passé en paramètre)

(voir le code dans le répertoire list/list_insertion)

Suppression d’un nœud d’une liste

Comme pour l’insertion, la suppression consiste en une manipulation de pointeurs.

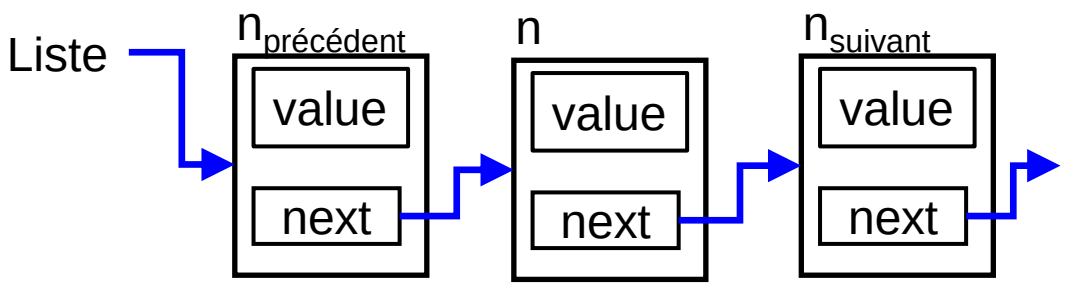
Il faut considérer tous les cas possibles:

- Suppression d’un nœud à l’intérieur d’une liste
- Suppression d’un nœud en début de liste
- Suppression d’un nœud en fin de liste
- Suppression du seul nœud d’une liste

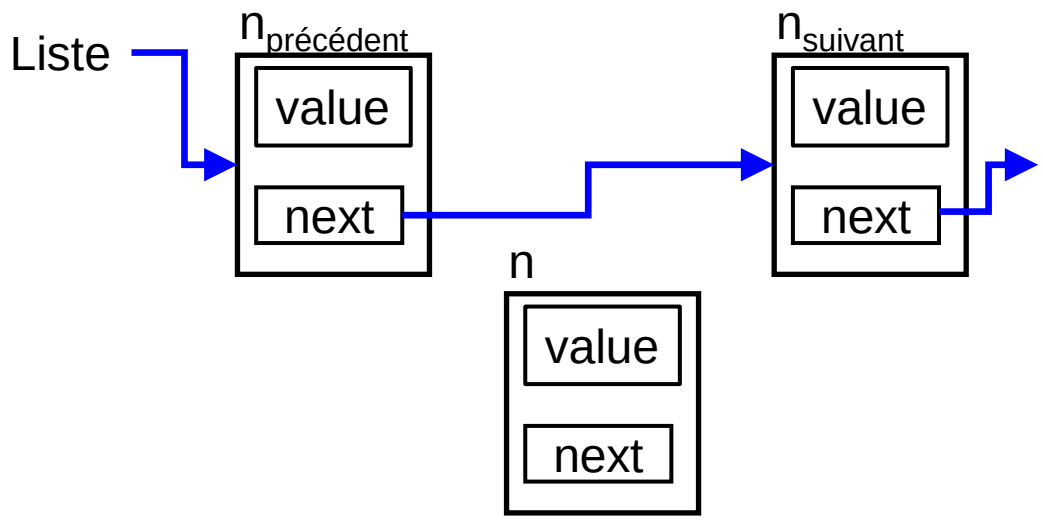
Suppression d'un nœud d'une liste (2)

On considère dans un premier temps la suppression d'un nœud n qui n'est ni le premier ni le dernier nœud d'une liste:

Avant suppression :



Après suppression :

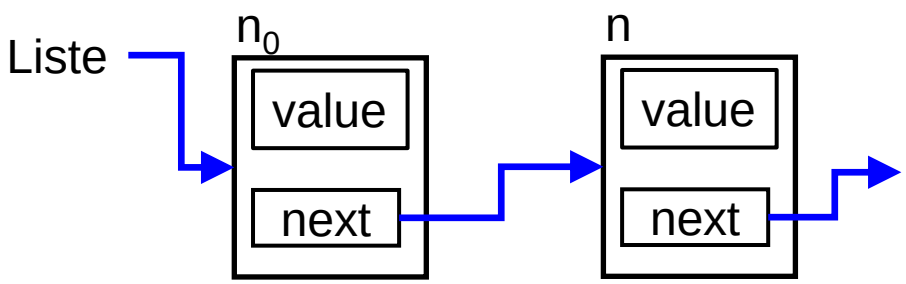


A noter qu'après avoir enlevé le nœud de la liste, il faut bien penser à mettre à zéro son pointeur next pour éviter de rentrer dans la liste par un nœud qui ne lui appartient plus.

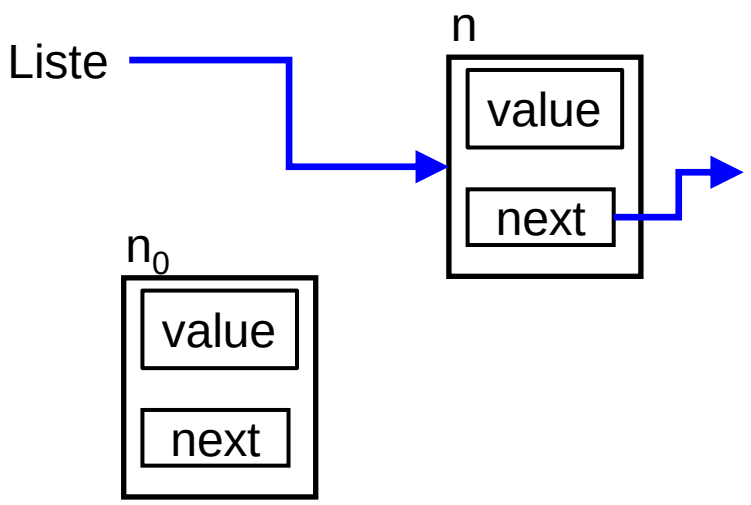
Suppression d'un nœud d'une liste (3)

On considère ensuite la suppression du premier nœud n_0 d'une liste :

Avant suppression :



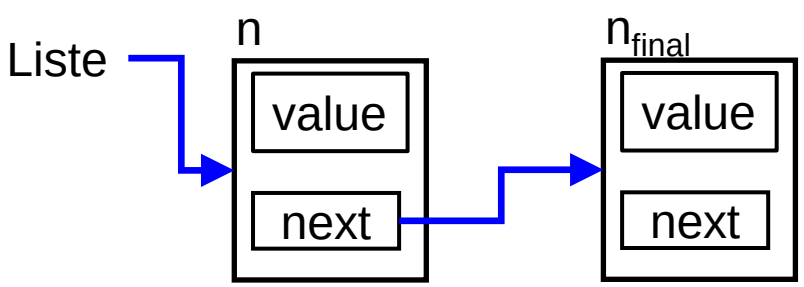
Après suppression :



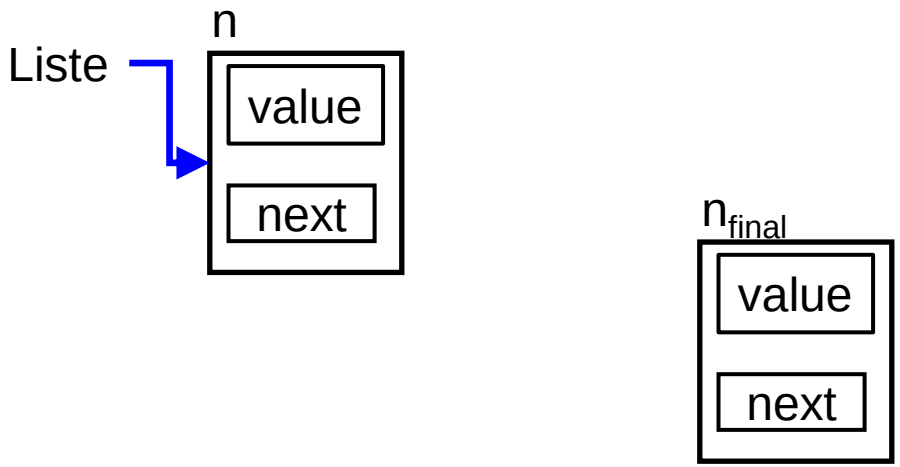
Suppression d'un nœud d'une liste (4)

On considère ensuite la suppression du dernier nœud n_{final} d'une liste :

Avant suppression :



Après suppression :



Suppression dans une liste (5)

Le plus simple (pour l'utilisateur du type liste) est de fournir une ou plusieurs fonctions de suppression, par exemple:

(exemple inspiré des listes standards C++)

```
int pop_front(List *L);

void pop_back(List *L);

void pop(List *L, k);
```

Suppression en début de liste (et retourne la valeur du premier nœud)

Suppression en fin de liste (et retourne la valeur du dernier nœud)

Suppression du $k^{ème}$ à l'intérieur d'une liste (et retourne la valeur du nœud)

(voir le code dans le répertoire list/list_suppression)

Types liste doublement chainée

Liste doublement chainée

Le type liste doublement chainée est une extension du type liste vu précédemment.
Chaque nœud possède les pointeurs pour pouvoir accéder au nœud précédent **et** au nœud suivant (s'ils existent)

```
struct _node
{
  int value;
  struct _node *next;
  struct _node *previous;
};
typedef struct _node Node;
```

Création/destruction de nœud

Il sera peut-être utile de définir des fonctions pour créer/détruire des nœuds:

```
Node* creeNode(int v) {
  Node* newNode = (Node*) malloc(sizeof(Node));
  newNode→value = data;
  newNode→next = NULL;
  newNode→previous = NULL;
  return newNode;
}
```

```
void detruitNode(Node ** n) {
  if (*n) {
    free(*n);
    *n = NULL;
}
```

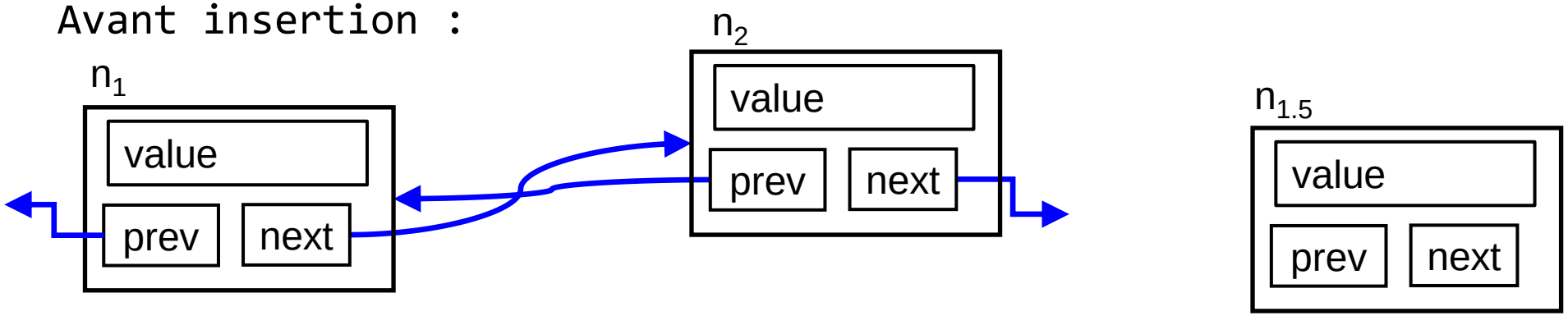
Opération d’insertion/suppression de nœuds dans une liste doublement chainée

Toutes les opérations vues pour les listes (simplement) chainées peuvent être adaptées aux listes doublement chainées

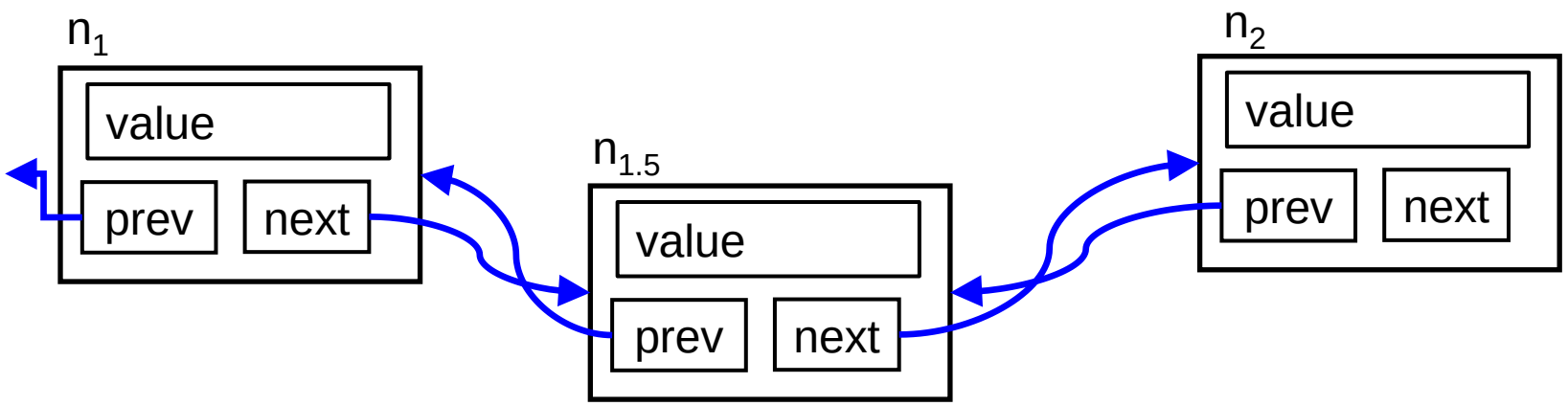
Insertion dans une liste doublement chainée

On considère l'insertion d'un nouveau nœud $n_{1.5}$ entre 2 nœuds n_1 et n_2 qui se suivent :

Avant insertion :



Après insertion :



Insertion dans une liste doublement chainée

Les fonctions réalisant l’insertion et la suppression des nœuds dans une liste doublement chainée sont laissées à titre d’exercice.