Algorithmique et structures de données

CM 2

Pointeurs, structures récursives et listes chaînées

Plan du CM 2

Plan du CM2

Types – rappels

Variables et types

Les données sont représentées par des variables

- toute variable possède un nom/identifiant qui désigne un emplacement mémoire
- elle a un type qui détermine la place (le nombre d'octets) que cette variable occupe en mémoire

Types de base et types composés

- types de base ou simples (disponibles dans le langage)
  - booléen
  - entier
  - réel
  - caractère et chaîne de caractères
  - ...
- types composés (créés par l'utilisateur)
  - structure ou enregistrement
  - tableau
  - liste chaînée
  - arbre
  - ...

Pour notre langage algorithmique, nous considérerons un typage statique

Enregistrements ou structures

Regroupement de types variés

Permet de regrouper des données avec des types différents

```
<nom attribut> : <nom type>

structure personne
   nom : chaîne de caractères
   prénom : chaîne de caractères
   age : entier
```

Attributs

- les données sont appelées champs ou attributs
- il n'y a pas d'ordre entre les attributs dans la définition de la structure

Enregistrements ou structures

Notation pointée

On accède aux attributs avec un point

```
<nom donnée>.<nom attribut>
```

Déclaration d'un attribut

Déclaration d'une variable de type personne

```
untel : personne
```

Affectation d'un attribut

Affectation des attributs de la donnée *untel*

```
untel.prénom = "Pierre" ; untel.nom = "Quiroule" ; untel.age = 20

untel.age = untel.age + 1 //on incrémente l'âge de 1
```

Accès à un attribut

Affichage des attributs d'une personne

```
affichagepersonne(P : personne)
   afficher P.nom ; afficher P.prénom ; afficher P.age
```

Plan du CM2

Les pointeurs – définition

A quoi sert un pointeur

- Un pointeur sert à pointer l'emplacement de la mémoire où se trouve un objet
- cet objet est appelé l'objet pointé
- le pointeur contient l'adresse de cet objet dans la mémoire

Constitution d'un pointeur

- Un pointeur a un type qui précise le type de l'objet pointé
- la valeur d'une variable pointeur est l'adresse de l'objet pointé

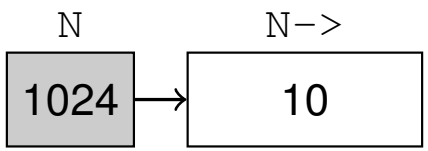
Les pointeurs – Notation de l'objet pointé

Notation de l'objet pointé

- si N est un pointeur, l'objet pointé est noté $N->$ dans notre langage algorithmique
- il est aussi souvent noté $*N$ dans les langages de programmation comme C et C++

Exemple

```
N : pointeur sur entier
```



- la valeur de N vaut 1024 qui est l'adresse de $N->$, elle permet de retrouver l'emplacement de $N->$
- la valeur de $N->$ vaut 10, elle est de type entier

Les pointeurs – allocation (1)

Allocation

L'allocation de la mémoire pour la variable pointée peut se faire explicitement avec la commande Nouveau(<Type>).

```
R : pointeur sur réel
R = Nouveau(réel) // R-> est créé et son adresse est donnée à R
R-> = 12.31
afficher R->
```

- la commande Nouveau(<Type>) entraîne une allocation de mémoire
- la taille de l'emplacement dépend du type

L'allocation est obligatoire

```
R : pointeur sur réel
R-> = 3.21
```

Dans notre langage algorithmique, la dernière instruction entraîne une erreur car l'objet $R->$ n'existe pas.

Les pointeurs – allocation (2)

Copie d'adresse

On peut également copier l'adresse d'un autre pointeur.

```
S : pointeur sur réel
S = R // le pointeur S contient la même adresse que R
afficher S->
S-> = 4.2
afficher R-> , S-> // on affiche 4.2 pour les deux pointeurs
```

- cela permet d'avoir un deuxième pointeur qui pointe sur un même objet
- il n'y a pas besoin d'effectuer une allocation de mémoire
- il faut être sûr de vouloir mettre la même valeur par la suite

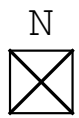
Les pointeurs – allocation (3)

L'adresse None

- None signifie que le pointeur ne pointe sur aucun objet
- cette adresse est indispensable pour définir des structures récursives

```
N : pointeur sur entier
N = None
```

On représente None avec le schéma suivant



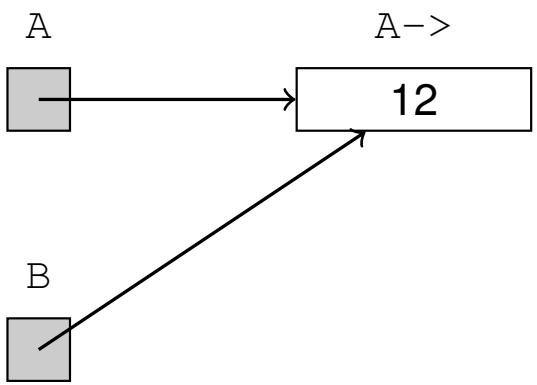
Différence entre copie d'adresse et de valeur

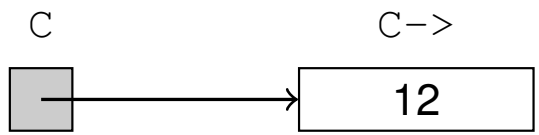
Qu'affiche-t-on à la fin ?

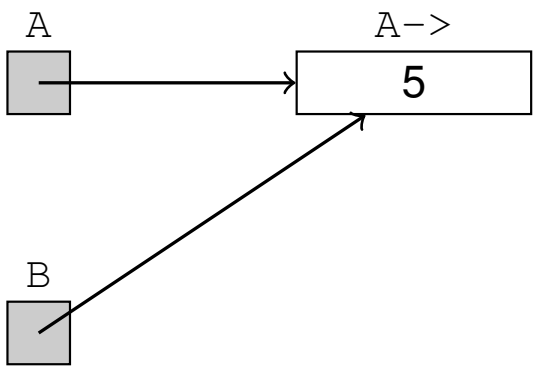
```
A, B, C : pointeur sur entier
A = Nouveau(entier) ; A-> = 12
B = A
C = Nouveau(entier) ; C-> = A->
A-> = 5
afficher A->, B->, C->
```

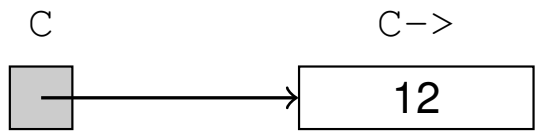
Les pointeurs – allocation (4)

- A et B partagent la même adresse
- C possède sa propre adresse









Plan du CM2

Les structures récursives (1)

Autoréférence

Une structure ou enregistrement est dite récursive si elle contient une ou plusieurs références (pointeurs) sur cette même structure.

Structure nœud

```
structure noeud
    valeur : entier // le type est fixé
    suivant : pointeur sur noeud
```

La structure noeud contient l'attribut suivant qui est défini à partir de la structure.

Les structures récursives (2)

Intérêt des structures récursives

- Cela permet de construire le nombre de nœuds que l'on souhaite
- On utilise None pour définir le dernier nœud.

Accès à la valeur

```
p : pointeur sur noeud
p = Nouveau(noeud)
p->valeur = 10
P->suivant = None
```

Simplification de la notation

On devrait écrire p->.valeur et p->.suivant au lieu de p->valeur et p->suivant

Les structures récursives (3)

Structures imbriquées

Il est possible d'imbriquer plusieurs structures entre-elles (cf composition en POO)

Définition d'une personne

```
structure personne
   nom : chaîne de caractères
   prénom : chaîne de caractères
   age : entier
```

Noeudpersonne

```
structure noeudPersonne
    valeur : personne // le type est fixé
    suivant : pointeur sur noeudPersonne
```

La structure noeudPersonne contient un attribut valeur qui est lui même une structure personne.

Les structures récursives (4)

Exemple avec une personne

```
p : pointeur sur noeudPersonne//déclaration de p
p = Nouveau(noeudPersonne)  //allocation mémoire pour p
p->valeur.nom = "Dupond"  //affectation des attributs de p->valeur
p->valeur.prenom = "Jean"
p->valeur.age = 37
unNoeud = p-> // p-> est un noeudPersonne
untel = unNoeud.valeur // untel est une personne
afficher untel.age, p->valeur.age // on affiche 37 dans les deux cas
untel.nom = "Dupont"
afficher p->valeur.nom // on affiche "Dupont"
```

Plan du CM2

Listes chaînées – définition

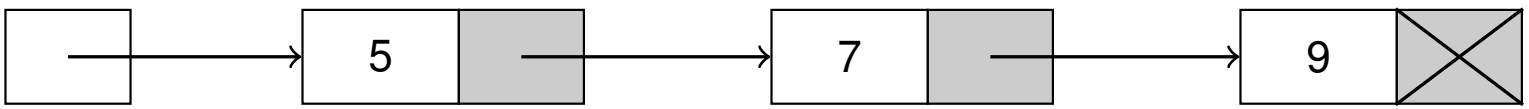
Intérêt des structures récursives

Les structures récursives ne sont intéressantes que si l'on manipule plusieurs objets.

- nombre d'objets non fixé au départ $\neq$ tableaux
- gestion dynamique : insertion et suppression

Liste chaînée

Les éléments de la liste sont chaînés en utilisant la structure de nœud.
Exemple : une liste chaînée contenant les entiers 5, 7, 9.

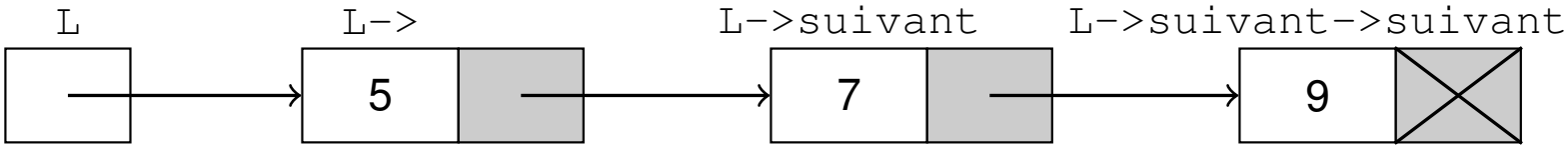


Type liste

```
type liste = pointeur sur noeud
L1, L2 : liste
```

- il s'agit d'un alias
- nous pouvons indifféremment utiliser le terme *pointeur sur nœud* ou *liste*
- nous pouvons définir autant d'alias que nous le souhaitons

Listes chaînées – construction



Construction à la main d'une liste chaînée

```
L : pointeur sur noeud
L = Nouveau(noeud) //allocation mémoire du premier noeud
L->valeur = 5 //affectation de la valeur du premier noeud
L->suivant = Nouveau(noeud) //allocation mémoire du second noeud
L->suivant->valeur = 7 //affectation de la valeur du second noeud
L->suivant->suivant = Nouveau(noeud) //allocation mémoire du troième noeud
L->suivant->suivant->valeur = 9 //affectation de la valeur du troisième noeud
L->suivant->suivant->suivant = None //le troisième noeud pointe sur None
```

- La méthode n'est pas pratique
- Nous souhaitons définir des procédures pour pouvoir construire une liste de taille quelconque

Listes chaînées – affichage

Affichage d'une liste

```
affichageListe(L : liste)
tant que L <> None faire //le dernier noeud doit pointer sur None
   afficher L->valeur//affichage du noeud courant
   L = L->suivant //on passe au noeud suivant
```

Rappel : le passage de paramètre s'effectue par valeur.

```
L : liste
L = Nouveau(noeud) ; L->valeur = 5 ; L->suivant = None
affichageListe(L)
```

Après l'appel de la procédure *affichageListe*, la liste L reste inchangée.

Rappel : Alias entre pointeur sur nœud et liste

```
p = pointeur sur noeud
p = L
affichageListe(p)
```

Listes chaînées – insertion en début de liste

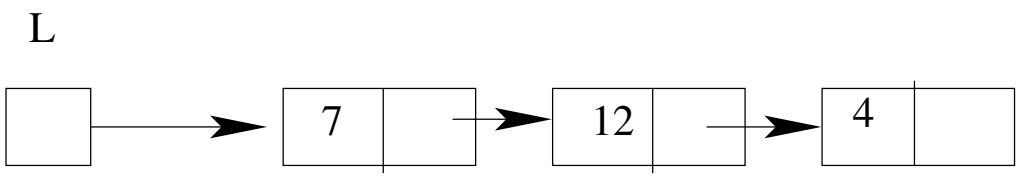
Insertion en début de liste

La variable tampon est une variable intermédiaire

```
insertionDebut(L : liste, n : entier) : liste
   tampon : pointeur sur noeud   (1)
   tampon = Nouveau(noeud)       (1)
   tampon->valeur = n            (1)
   tampon->suivant = L           (2)
   retourner tampon              (3)
```

Le coût de l'opération est le même quelle que soit la longueur de la liste.

Exemple, insertion de la valeur 8



Listes chaînées – insertion en début de liste

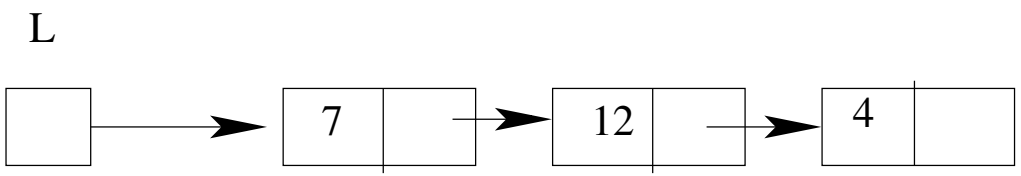
Insertion en début de liste

La variable tampon est une variable intermédiaire

```
insertionDebut(L : liste, n : entier) : liste
   tampon : pointeur sur noeud   (1)
   tampon = Nouveau(noeud)       (1)
   tampon->valeur = n            (1)
   tampon->suivant = L           (2)
   retourner tampon              (3)
```

Le coût de l'opération est le même quelle que soit la longueur de la liste.

Exemple, insertion de la valeur 8



Listes chaînées – insertion en début de liste

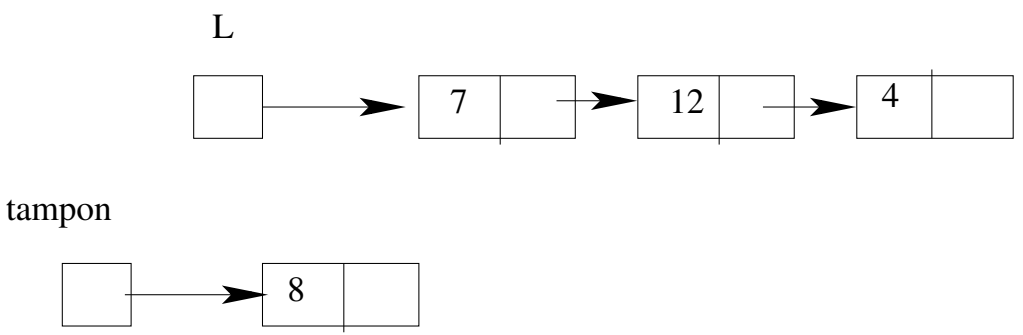
Insertion en début de liste

La variable tampon est une variable intermédiaire

```
insertionDebut(L : liste, n : entier) : liste
   tampon : pointeur sur noeud   (1)
   tampon = Nouveau(noeud)       (1)
   tampon->valeur = n            (1)
   tampon->suivant = L           (2)
   retourner tampon              (3)
```

Le coût de l'opération est le même quelle que soit la longueur de la liste.

Exemple, insertion de la valeur 8, étape (1)



Listes chaînées – insertion en début de liste

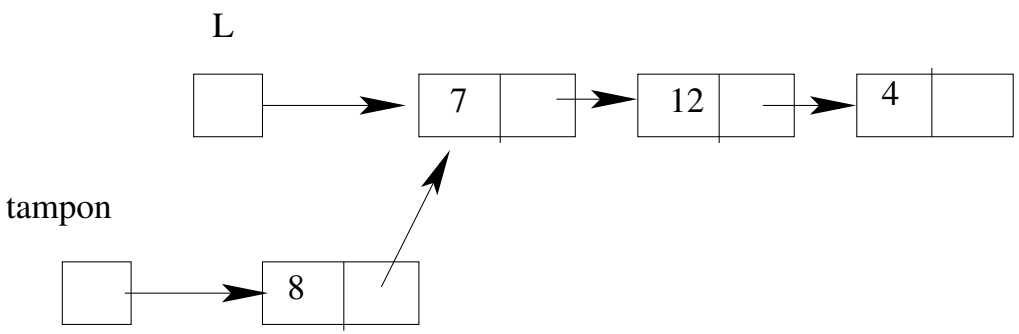
Insertion en début de liste

La variable tampon est une variable intermédiaire

```
insertionDebut(L : liste, n : entier) : liste
   tampon : pointeur sur noeud   (1)
   tampon = Nouveau(noeud)       (1)
   tampon->valeur = n            (1)
   tampon->suivant = L           (2)
   retourner tampon              (3)
```

Le coût de l'opération est le même quelle que soit la longueur de la liste.

Exemple, insertion de la valeur 8, étape (2)



Listes chaînées – insertion en début de liste

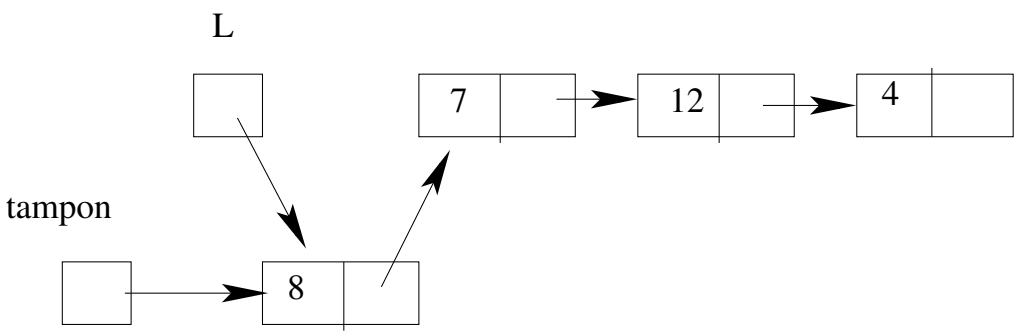
Insertion en début de liste

La variable tampon est une variable intermédiaire

```
insertionDebut(L : liste, n : entier) : liste
   tampon : pointeur sur noeud   (1)
   tampon = Nouveau(noeud)       (1)
   tampon->valeur = n            (1)
   tampon->suivant = L           (2)
   retourner tampon              (3)
```

Le coût de l'opération est le même quelle que soit la longueur de la liste.

Exemple, insertion de la valeur 8, étape (3)



Listes chaînées – insertion en début de liste

Construction de la liste contenant les valeurs 5, 7 et 9

```
maliste  : liste
maListe = insertionDebut(maListe,9)
maListe = insertionDebut(maListe,7)
maListe = insertionDebut(maListe,5)
affichageListe(maListe)
```

Remarque

L'ordre est inversé.
On insère donc d'abord les derniers nœuds.

Listes chaînées – copie d'une liste (1)

Procédure récursive

```
copieListeRec(L : liste) : liste
   si L = None alors
      retourner None
   sinon
      res : liste
      res = Nouveau(noeud)
      res->valeur = L->valeur
         res->suivant = copieListeRec(L->suivant)
         retourner res
```

L'appel récursif se fait sur le nœud suivant

Listes chaînées – copie d'une liste (2)

Procédure itérative

```
copieListeIt(L : liste) : liste
   res, tmp : liste ; res = None
   tmp = res
   si L = None alors retourner res
   tmp=Nouveau(noeud);tmp->valeur=L->valeur;tmp->suivant=None
   L = L->suivant
   tant que L <> None faire
        tmp->suivant=Nouveau(noeud);tmp->suivant->valeur=L->valeur
        tmp = tmp->suivant ; L = L->suivant
   tmp->suivant = None
   retourner res
```

Remarque

On utilise la boucle tant que L <> None faire lorsque l'on parcourt L sans modifier cette liste (copie, affichage,...).

Listes chaînées – copie d'une liste (3)

Méthode

Pour copier une liste en inversant l'ordre, on insère au début un à un les éléments.

Procédure itérative – méthode la plus simple

```
inversionListeIt(L : liste) : liste
   res = None
   tant que L <> None faire
      res = insertionDebut(res,L->valeur)
      L = L->suivant
   retourner res
```

Procédure récursive * – méthode plus difficile

*L*1 : liste de départ, *L*2 : liste en sortie.

```
inversionListeRec(L1, L2 : liste) : liste
   si L1 = None alors retourner L2
   sinon retourner inversionListeRec(L1->suivant,
                        insertionDebut(L2,L1->valeur)
```

Pour l'inversion récursive, nous sommes obligés d'avoir un second argument pour la liste de sortie.

Modification d'une liste – suppression du dernier nœud

Première tentative

```
suppressionFin(L : liste) : liste
    tmp : liste ; tmp = L
    si tmp = None retourner None
    tant que tmp <> None faire
        tmp = tmp->suivant
    tmp = None
    retourner L
```

En fait la procédure ne fait rien !
Il faut s'arrêter sur le dernier nœud.

Modification d'une liste – suppression du dernier nœud

Procédure correcte

```
suppressionFin(L : liste) : liste
   tmp : liste ; tmp = L
   si tmp = None retourner None
   tant que tmp->suivant <> None faire
      tmp = tmp->suivant
   désallouer(tmp)
   tmp = None
   retourner L
```

désallouer(tmp) permet de libérer la mémoire occupée par le nœud vers lequel pointe tmp.