Programmation en C
Esisar - CS210

Collections (rappels et compléments)

Problématique (1)

- Dans toute application informatique, cohabitent :
  - des données
  - des traitements qui s 'appliquent sur ces données.
- Selon le domaine, l 'application comprendra plus ou moins de données.
- Dans certains cas, certains domaines, les données à traiter sont en nombre limité, la taille nécessaire au stockage des informations n'est pas très importante.

Problématique (2)

- Dans d'autres domaines, la quantité d 'information est très très importante, et de taille très variable, ce qui peut conduire (avec d'autres considérations) à séparer franchement les traitements des données traitées (Cf « bases de données »)
- Sans aller jusqu 'à cette séparation, certains problèmes impliquent de stocker et gérer un « certain nombre » (dynamique!) de données présentant des propriétés communes.
- Se pose alors pour des raisons d 'efficacité la problématique de « comment » les organiser au mieux pour le problème à résoudre.

Notion de collection

- Les données (dynamiques) sont souvent regroupées et organisées selon quelques schémas classiques désignés sous le nom de « collections ».
- Parmi les « collections », on trouve entre autres (revoir cours d'algorithmique) :
  - les listes
  - les dictionnaires (« map »)
  - les « ensembles » (« set »)
  - les arbres (« tree »)

Traitements sur les collections

Les traitements classiques sur les collections sont :

- traitements faisant évoluer la collection :
  - création
  - ajout/insertion d 'un élément
  - suppression d 'un élément
  - destruction de la collection
- traitements divers sur la collection :
  - recherche d 'un élément
  - copie d 'un élément
  - tri de la collection

Collections en C

- On peut réaliser (« à la main… ») si nécessaire des collections en C sous forme de tableaux ou de listes chaînées.
- Tableaux :
  - avantages : accès direct à l'information, par indice.
  - inconvénient : taille statique, ne convient pas pour des collections d'objets de grande taille de nombre très variable.
    (Problèmes de complexité en espace).

Collections en C (2)

- liste chaînées
  - avantages : taille complètement dynamique.
  - inconvénient : accès essentiellement séquentiel, certains algorithmes utilisant de façon importante des " recherches " peuvent devenir inexploitables (Problèmes de complexité en temps).
- Bien entendu, il existe des bibliothèques (non standard) (ex : glib) implémentant les-dites collections…

Collection d 'entiers

- Dans la suite de ce poly-résumé, pour simplifier, je donnerai des exemples utilisant exclusivement des collections d 'entiers, les autres collections se déduisant facilement de ces exemples.
- Je vais envisager quelques types de collections et leurs opérations associées.
- Une distinction importante est le fait que la collection soit triée ou non.

Collection triée implémentée par un tableau

- Principe :
  - La collection est constituée d 'un tableau permettant de stocker les données, et d 'un entier indiquant le nombre de données valides.
  - Les données sont gérées dans le tableau de sorte que la collection soit toujours triée.
- Insertion/Suppression :
  - la seule stratégie possible est celle du décalage (pour suppression et insertion).

Implémentation possible en C

- Exemple de collection d'entiers

```
int tab[NBMAX];
int nbEntiers ;
```

- On peut également encapsuler plus :

```
typedef struct {
   int nbEntiers ;
   int tab[NBMAX]; } Collection ;
```

- utilisation :

```
Collection maCollec ; /* réserve l'espace
                         dans la pile !!!*/
```


Implémentation possible pour une collection plus générale (sujet du TP)

- Exemple de collection d '«Etudiant»s

```
Etudiant tab[NBMAX];
int nbEtudiants ;
```

(avec « `Etudiant` » structure-typedef contenant les infos nécessaires, cf « etudiant.h »)

- On peut également encapsuler plus :

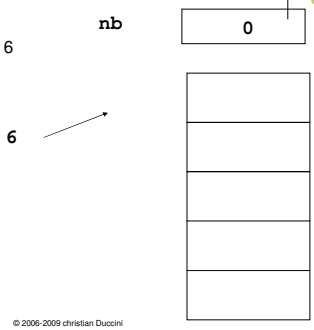
```
typedef struct {
   int nbEtudiants ;
   Etudiant tab[NBMAX]; } Collection ;
```

- utilisation :

```
Collection maCollec ; /* réserve l'espace
                         dans la pile !!!*/
```

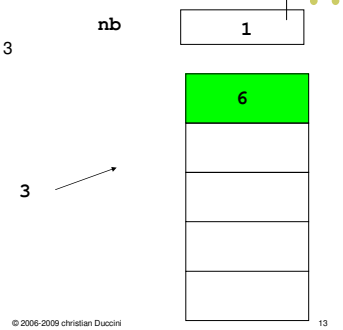

Exemple d 'insertion (collection triée!)

- Insertion d 'un élément : 6



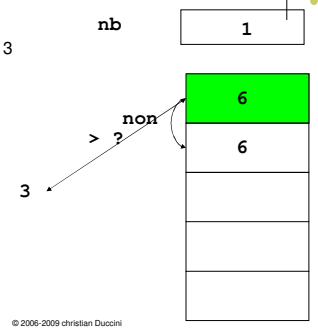
Exemple d 'insertion (collection triée!)

- Insertion d 'un élément : 3



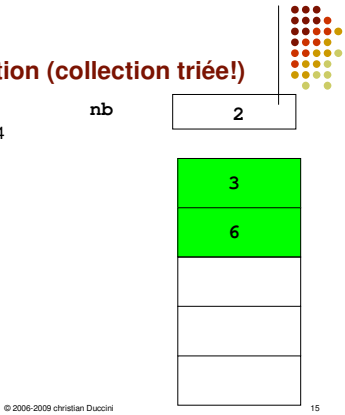
Exemple d 'insertion (collection triée!)

- Insertion d 'un élément : 3



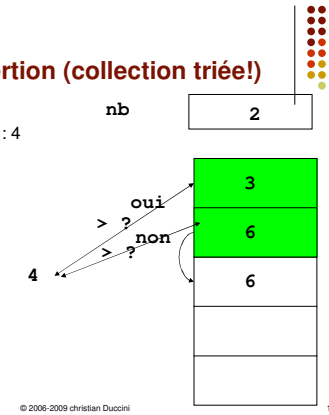
Exemple d 'insertion (collection triée!)

- Insertion d 'un élément : 4



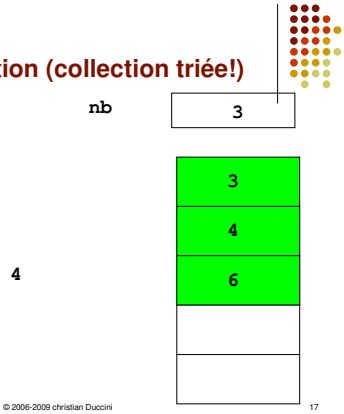
Exemple d 'insertion (collection triée!)

- Insertion d 'un élément : 4



Exemple d 'insertion (collection triée!)

- Etat final

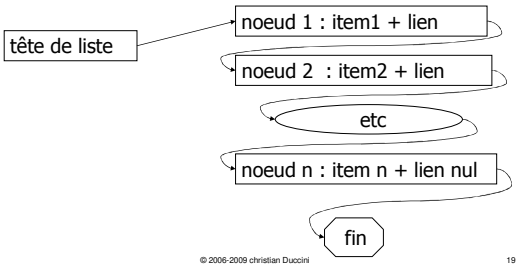


Collection implémentée dans une liste chaînée simple (non triée)

- Principe :
  - Une liste est constituée de " nœuds ", chaînés entre eux.
  - Chacun des « nœuds » contient un élément, et un lien vers le prochain « nœud ».
  - On utilise une valeur non significative du lien pour indiquer la fin de la liste.
  - L 'espace mémoire pour conserver un « nœud », est obtenu par allocation, uniquement lorsque c 'est nécessaire.
- Pour définir une liste, il suffit de donc de définir sa " tête de liste ".

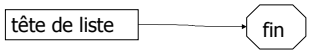
Collection implémentée dans une liste chaînée simple

- Représentation schématique (cas général) :



Collection implémentée dans une liste chaînée simple

- Représentation schématique (liste vide) :



Principe des traitements

- Mécanisme d'insertion : 3 phases
  - Allocation du bloc
  - Initialisation du bloc
  - Chaînage du bloc : en tête, en queue, en ordre.
- Mécanisme de suppression : 2 phases
  - Déchaînage du bloc
  - Libération du bloc
- Recherche : uniquement séquentielle.

Collection implémentée dans une liste chaînée simple

- Implémentation possible en 'C' :

```
typedef struct node{
          int data ;   /* ici, un entier */
          struct node * lien ;
          } Node ;
/* initialisation (collection vide!)*/
Node* head = NULL ; /* tête de liste */
/* on peut aussi conserver le nombre
  d 'éléments, mais c 'est une information
  redondante */
  int nb = 0 ;
```

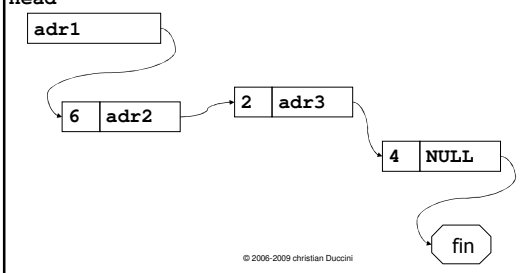

exemple de parcours dans une liste

- Ex : affichage de tous les éléments

```
Node* ptr ;/* pour parcours de liste */
ptr = head ;  /* init en tête de liste */
/* tant qu'on est pas en fin de liste */
while (ptr != NULL) {
         /* traiter l'info */
         printData( ptr→data ) ;
   /* ou ici printf ("%d\n",ptr->data) ; */
         /* se positionner sur le suivant */
         ptr = ptr→lien ;
         }
```

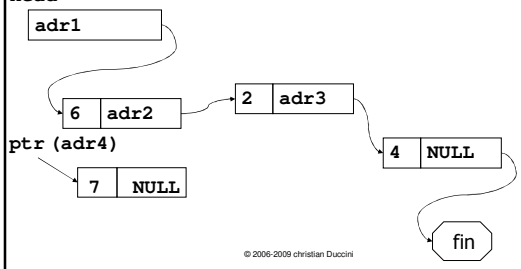

Insertion d'un nœud

- Etat initial : liste à 3 éléments (6, 2 et 4), à laquelle on veut rajouter un élément (7)



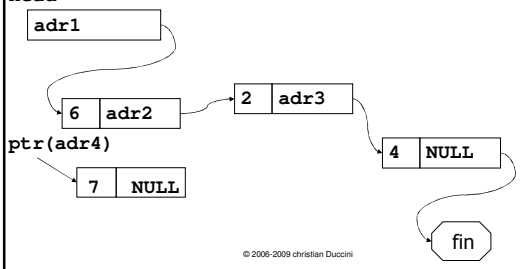
Insertion d'un nœud

- Phase 1+2 :
  allocation + initialisation :



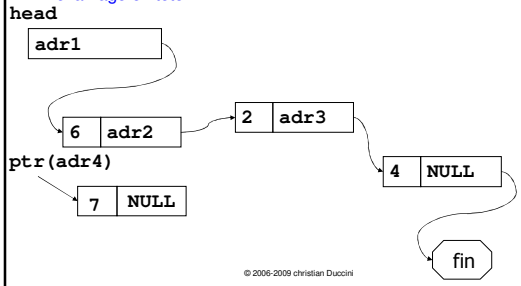
Insertion d'un nœud

- Phase 3 : cas particulier:
  chaînage en tête :



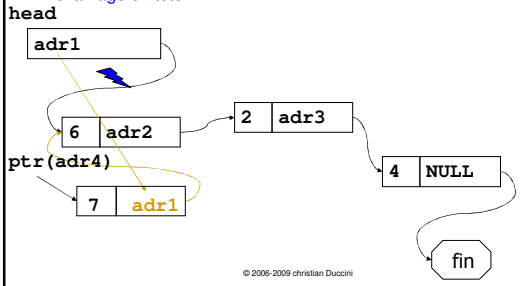
Insertion d'un nœud

- Phase 3 : cas particulier:
  chaînage en tête :



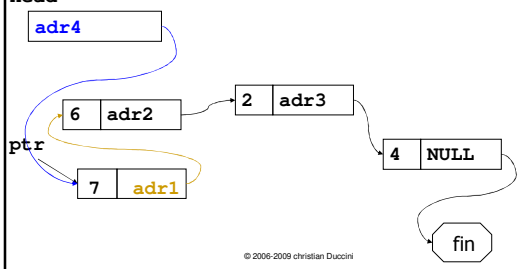
Insertion d'un nœud

- Phase 3 : cas particulier:
  chaînage en tête :



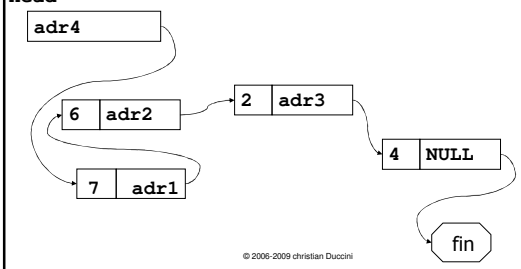
Insertion d'un nœud

- Phase 3 : cas particulier:
  chaînage en tête :



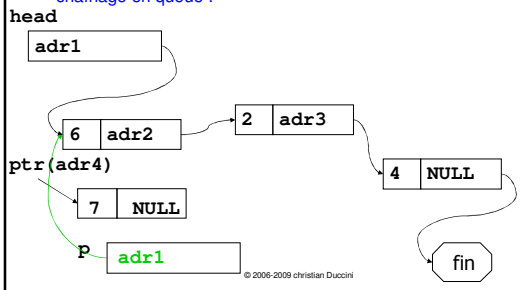
Insertion d'un nœud

- Etat final :



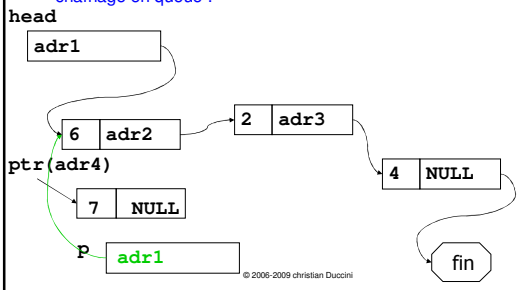
Insertion d'un nœud

- Phase 3 :
  chaînage en queue :



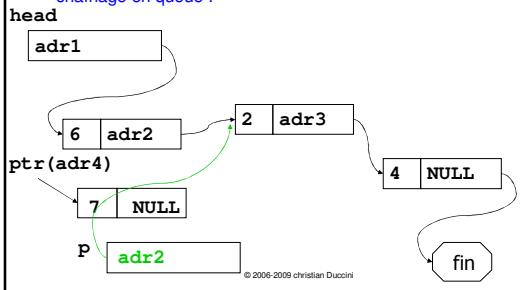
Insertion d'un nœud

- Phase 3 :
  chaînage en queue :



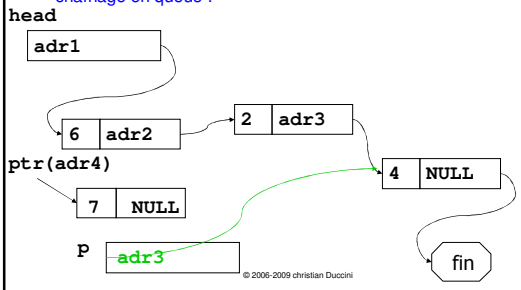
Insertion d'un nœud

- Phase 3 :
  chaînage en queue :



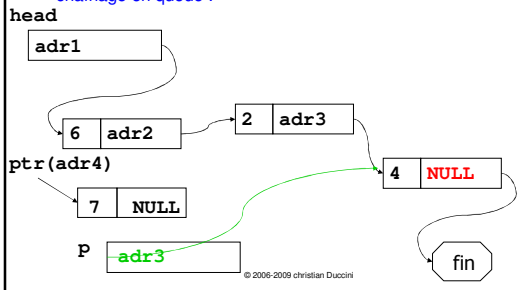
Insertion d'un nœud

- Phase 3 :
  chaînage en queue :



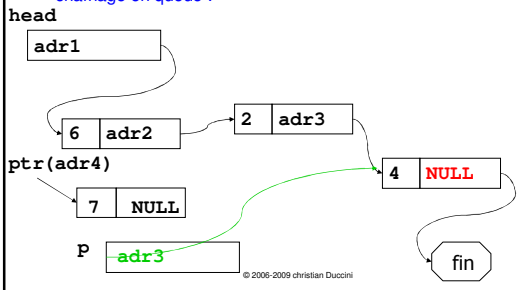
Insertion d'un nœud

- Phase 3 :
  chaînage en queue :



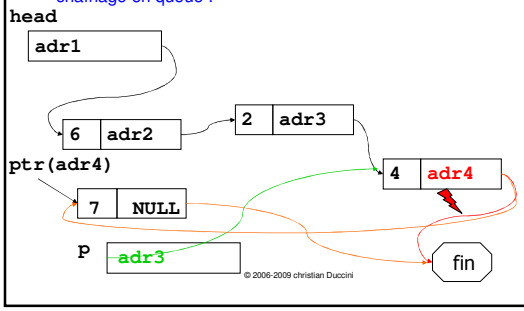
Insertion d'un nœud

- Phase 3 :
  chaînage en queue :



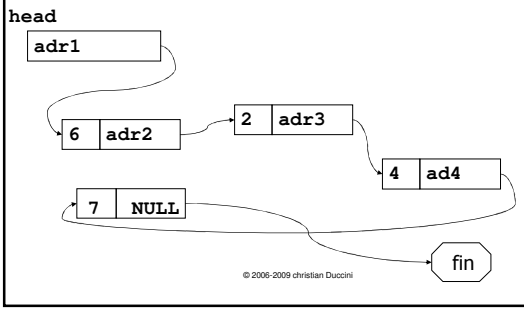
Insertion d'un nœud

- Phase 3 :
  chaînage en queue :



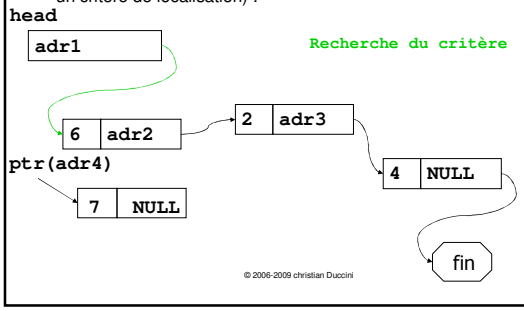
Insertion d'un nœud

- Etat final :



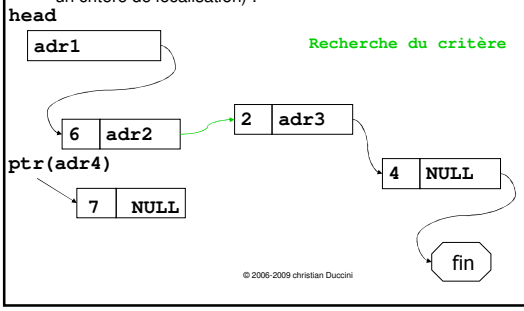
Insertion d'un nœud

- Phase 3 : chaînage intermédiaire (nécessite un critère de localisation) :



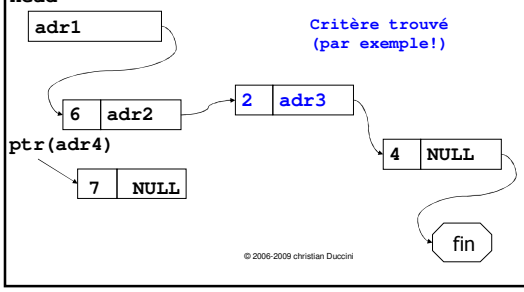
Insertion d'un nœud

- Phase 3 : chaînage intermédiaire (nécessite un critère de localisation) :



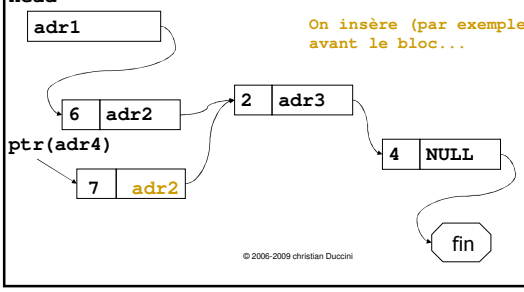
Insertion d'un nœud

- Phase 3 : chaînage intermédiaire (nécessite un critère de localisation) :



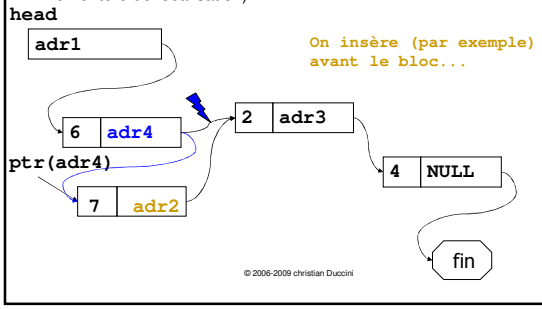
Insertion d'un nœud

- Phase 3 : chaînage intermédiaire (nécessite un critère de localisation) :



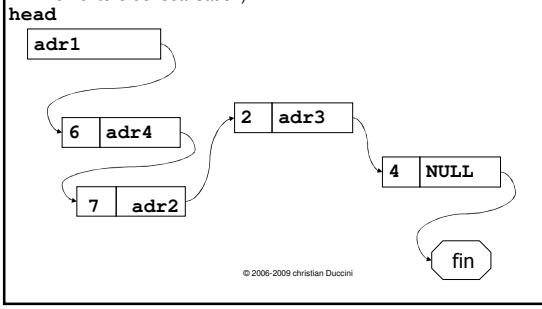
Insertion d'un nœud

- Phase 3 : chaînage intermédiaire (nécessite un critère de localisation) :



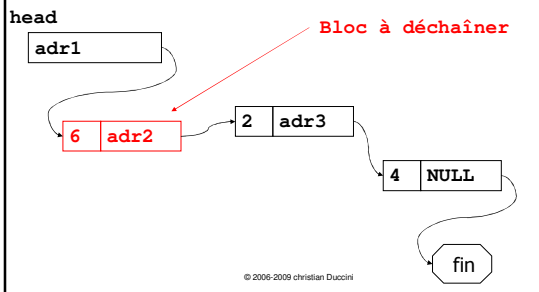
Insertion d'un nœud

- Phase 3 : chaînage intermédiaire (nécessite un critère de localisation) :



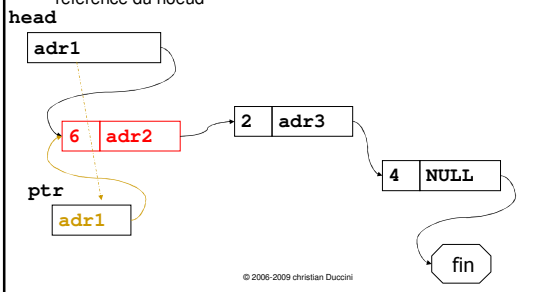
Suppression d'un nœud

- Etat initial : cas particulier: déchaînage en tête



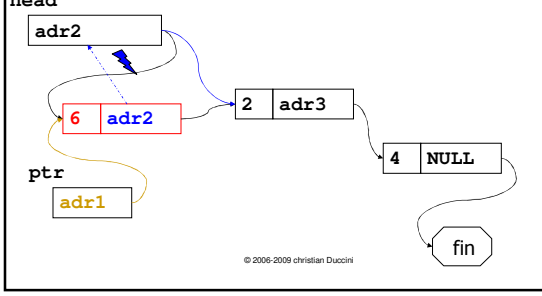
Suppression d'un nœud

- Phase 1 : déchaînage en tête, et conservation de la référence du noeud



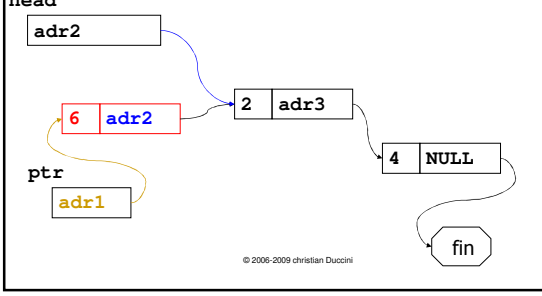
Suppression d'un nœud

- Phase 1 : déchaînage en tête, et conservation de la référence du noeud



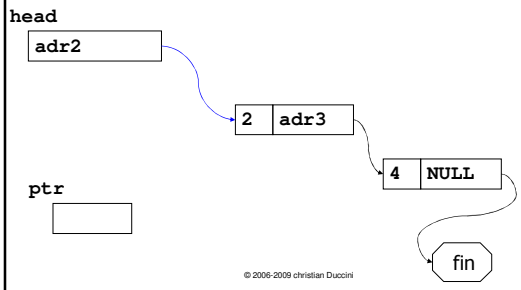
Suppression d'un nœud

- Phase 1 : déchaînage en tête, et conservation de la référence du noeud



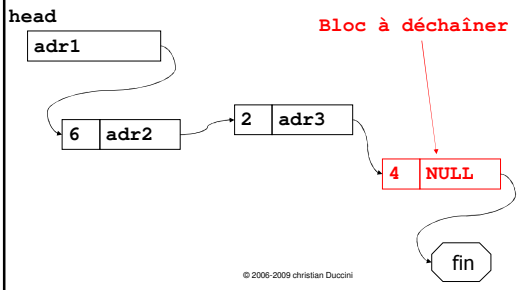
Suppression d'un nœud

- Puis libération…



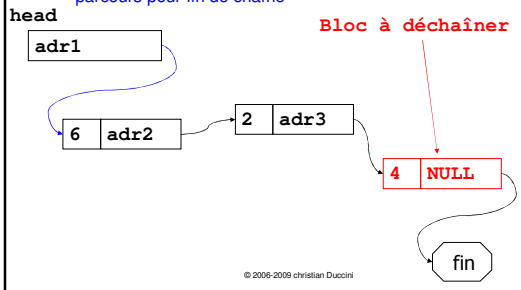
Suppression d'un nœud

- Etat initial : cas particulier : déchaînage en queue :



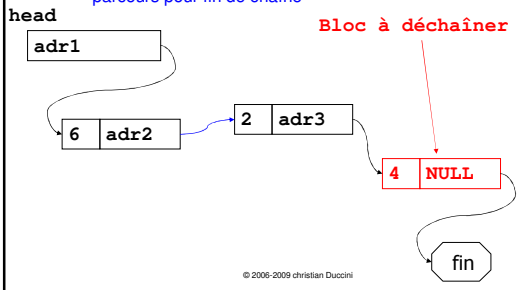
Suppression d'un nœud

- Etat initial : cas particulier : déchaînage en queue :
  parcours pour fin de chaîne



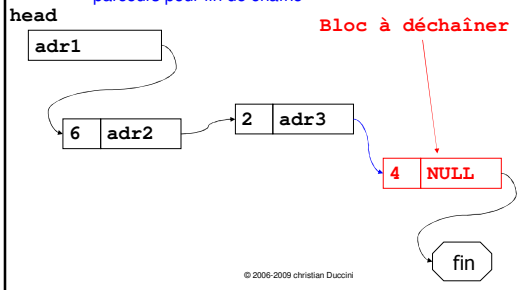
Suppression d'un nœud

- Etat initial : cas particulier : déchaînage en queue :
  parcours pour fin de chaîne



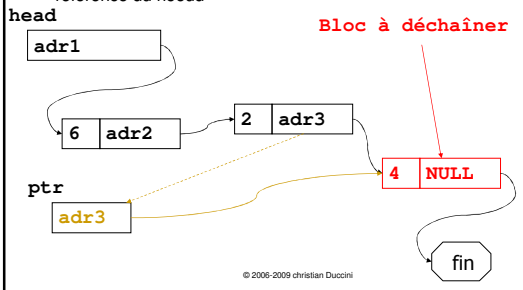
Suppression d'un nœud

- Etat initial : cas particulier : déchaînage en queue :
  parcours pour fin de chaîne



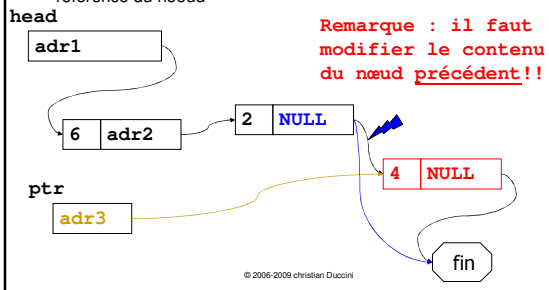
Suppression d'un nœud

- Phase 1 : déchaînage en queue, et conservation de la référence du noeud



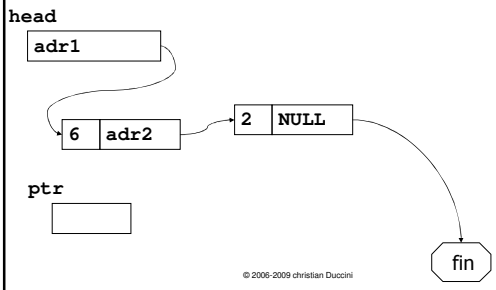
Suppression d'un nœud

- Phase 1 : déchaînage en queue, et conservation de la référence du noeud



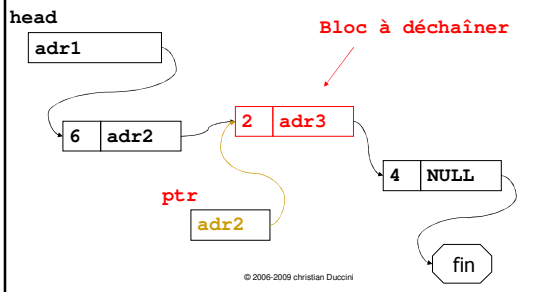
Suppression d'un nœud

- Puis libération



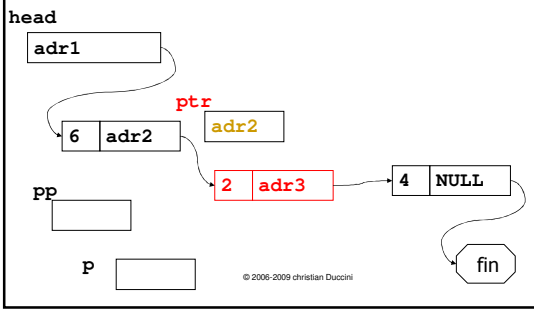
Suppression d'un nœud

- Etat initial (déchaînage intermédiaire) :



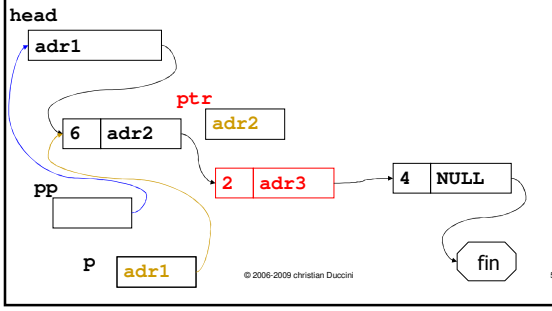
Suppression d'un nœud

- Etat initial (déchaînage intermédiaire) : parcours pour recherche



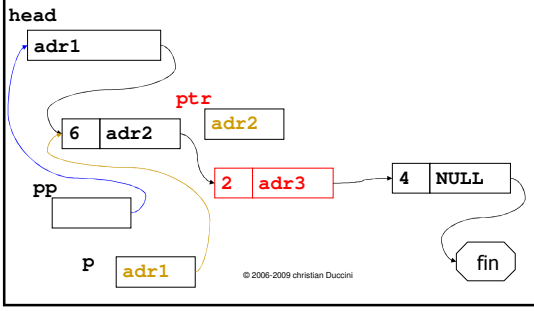
Suppression d'un nœud

- Etat initial (déchaînage intermédiaire) : parcours pour recherche



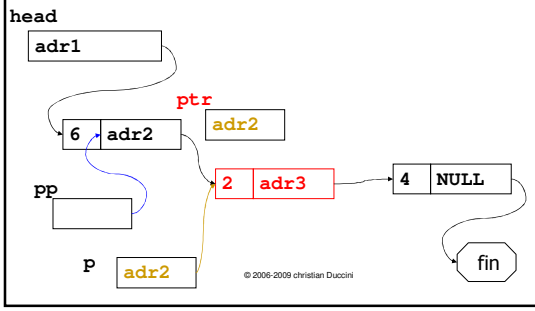
Suppression d'un nœud

- Etat initial (déchaînage intermédiaire) : parcours pour recherche



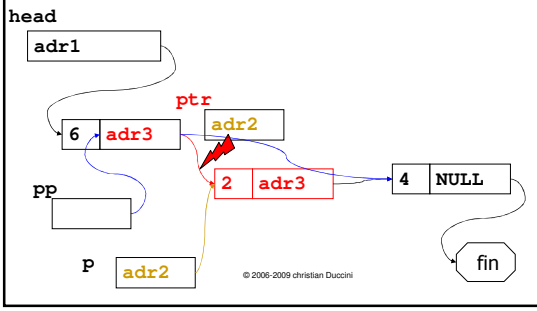
Suppression d'un nœud

- Etat initial (déchaînage intermédiaire) : parcours pour recherche



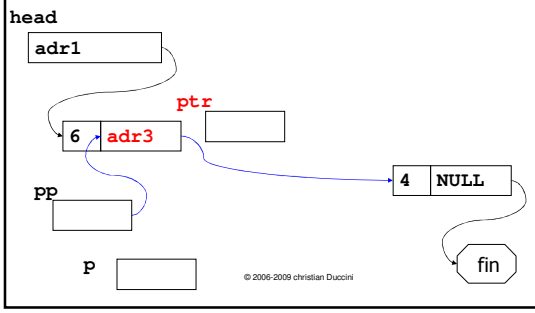
Suppression d'un nœud

- Etat initial (déchaînage intermédiaire) : parcours pour recherche



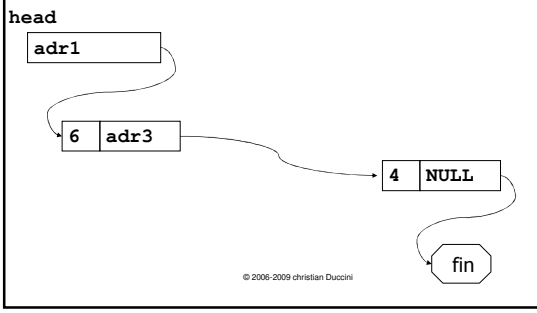
Suppression d'un nœud

- Etat initial (déchaînage intermédiaire) : parcours pour recherche



Suppression d'un nœud

- Etat final :



liste doublement chaînée (pour mémoire)

- Principe :
  - Idem liste simple, mais chaque « nœud » contient un lien avant et un lien arrière.
  - On conserve à la fois la tête de liste et la fin de liste.
- Avantage :
  - recherche ou déplacements avant-arrière facilités.

Représentation schématique

- Exemple :