1.1

Algorithmique et Structures de Données

Jean-Charles Régin

Licence Informatique 2ème année

1.2 Listes

Jean-Charles Régin

Licence Informatique 2ème année

Plan

- Présentation
- Listes doublement chaînées
- Implémentation
- Exemple d'utilisation de listes

Variable

1.4

- Une variable sert à mémoriser de l'information
- Ce qui est mis dans une variable est en fait mis dans une partie de la mémoire

Structures de données

1.5

- Permettent de gérer et d'organiser des données
- Sont définies à partir d'un ensemble d'opérations qu'elles peuvent effectuer sur les données
- Une structure de données ne regroupe pas nécessairement des objets du même type.

Besoin d'indirections

- Présentation habituelle de certains algorithmes :
  - On a un tableau d'entiers
  - On veut trier ce tableau

- Un élément du tableau est directement un type de base (un entier, un flottant, un booléen…)

- Parfois on ne voudrait pas avoir accès à la valeur en soit, mais plutôt à un objet lié à l'indice et associé à cette valeur

- On veut simplement parcourir les éléments d'un ensemble, pas uniquement les valeurs de ces éléments : on associe l'élément à une valeur

Besoin d'indirections

- Recherche dichotomique : ce qui nous intéresse
  - n'est pas la valeur,
  - n'est pas uniquement l'appartenance de la valeur
  - c'est la position de la valeur dans le tableau, donc son indice

- On pourrait travailler uniquement avec des indices et des tableaux
  - Un indice représentant un objet particulier

- Inconvénient : c'est compliqué
  - quand on veut supprimer un objet (que devient son indice ?)
  - quand on veut insérer un objet (que devient son indice ?)
  - quand on veut ajouter un objet (les tableaux doivent être agrandis)

Besoin d'indirections

- Il est plus pratique de travailler directement avec des objets et d'associé des valeurs à ces objets
- En java :
  - Class onObjet {…}
    MonObjet mobj1=new MonObjet(…);
    MonObjet mobj2=new MonObjet(…);
    définit 2 objets
  - MonObjet obj; // définit un autre objet
  - obj=mobj1;
    obj.setValue(8); // change une donnée de mobj1
  - obj=mobj2;
    obj.setValue(12); // change une donnée de mobj2
  - obj change indirectement mobj1 et mobj2, c'est une indirection
- On a donc besoin d'indirections !

Pointeur

- **Un pointeur est un type de données dont la valeur fait référence (référencie) directement (pointe vers) à une autre valeur.**

- Un pointeur référencie une valeur située quelque part d'autre en mémoire habituellement en utilisant son adresse

- Un pointeur est une variable qui contient une adresse mémoire

- Un pointeur permet de réaliser des indirections : désigner des objets, sans être ces objets

Pointeur

- Un pointeur est un type de données dont la valeur **pointe vers** une autre valeur.

- Obtenir la valeur vers laquelle un pointeur pointe est appelé **déréférencer** le pointeur.

- Un pointeur qui ne pointe vers aucune valeur aura la valeur **nil**

Listes

- Une **liste chaînée** désigne une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments.
- L'accès aux éléments d'une liste se fait de manière séquentielle
  - chaque élément permet l'accès au suivant (contrairement au cas du tableau dans lequel l'accès se fait de manière absolue, par adressage direct de chaque cellule dudit tableau).
- **Un élément contient un accès vers une donnée**

Liste

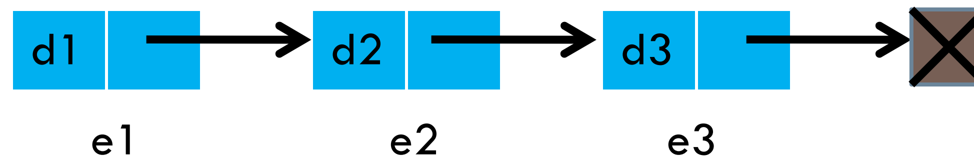
- Le principe de la liste chaînée est que chaque élément possède, en plus de la donnée, des pointeurs vers les éléments qui lui sont logiquement adjacents dans la liste.

- **premier(L)** désigne le premier élément de la liste
- **nil** désigne l'absence d'élément

- **Liste simplement chaînée :**
  - **donnée(elt)** désigne la donnée associée à l'élément elt
  - **suivant(elt)** désigne l'élément suivant elt

Liste simplement chaînée

- Représentation : ⟶ correspondant au suivant



- premier(L)=e1
- suivant(e1) = e2
- suivant(e2) = e3
- suivant(e3) = nil

Liste : opérations

- Trois opérations principales
  - Parcours de la liste
  - Ajout d'un élément
  - Suppression d'un élément

- A partir de là d'autres opérations vont être obtenues : recherche d'une donnée, remplacement, concaténation de liste, fusion de listes, etc.

Liste vs Tableau

- Le principal avantage des listes sur les tableaux
  - L'ordre des éléments de la liste peut être différent de leur ordre en mémoire.
  - Les listes chaînées vont permettre l'ajout ou la suppression d'un élément en n'importe quel endroit de la liste en temps constant.

- En revanche, certaines opérations peuvent devenir coûteuses comme la recherche d'un élément contenant une certaine donnée. Pas de recherche dichotomique dans une liste : on ne peut pas atteindre le ième élément sans parcourir !

Inventions des listes chaînées

- La représentation de listes chaînées à l'aide du diagramme avec une flèche vers le suivant a été proposé par Newell and Shaw dans l'article "Programming the Logic Theory Machine" Proc. WJCC, February 1957.

- Newell et Simon ont obtenu l'ACM Turing Award en 1975 pour avoir "made basic contributions to artificial intelligence, the psychology of human cognition, and list processing".

Lvalue et Rvalue

- Pour se simplifier la vie, on accepte de faire `suivant(elt) ← valeur`
- On remarque qu'il n'y a pas d'ambigüité.
- Cela s'appelle une Lvalue ou Left-value (on accepte de mettre à gauche de l'affectation)
- Le cas normal est la Rvalue (right-value)

Liste : parcours

- **initListe(L)**

```
premier(L) ← nil
```

- **nombreElements(L) : entier**

```
cpt ← 0;
elt ← premier(L)
tant que(elt ≠ nil){
    cpt ← cpt +1
    elt ← suivant(elt)
}
retourner cpt
```

Liste : ajout d'un élément

- On ajoute un élément elt au début de la liste.
- On suppose qu'il n'est pas déjà dans la liste (sinon que se passe t'il ?)

- Principes :
  - Le premier de la liste deviendra elt
  - Mais où est le premier ? Il devient le suivant de elt
  - Attention à l'ordre de mise à jour ! On ne doit pas perdre le premier. Donc
    - Le suivant de elt est mis à jour
    - Puis le premier de la liste

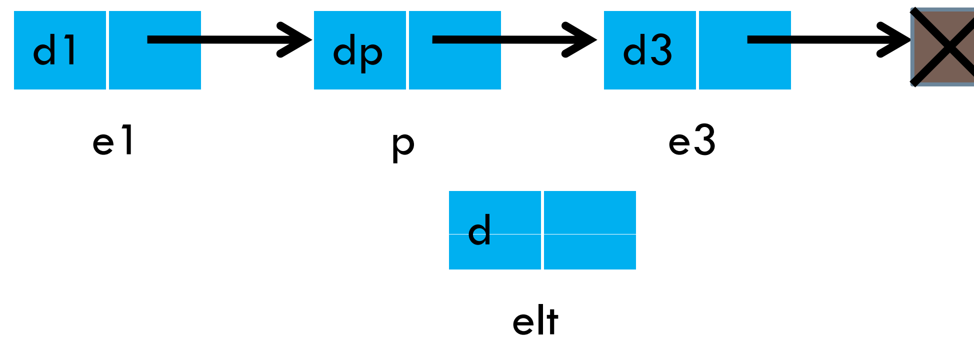
Liste : ajout d'un élément

- **ajouteAuDébut(elt,L)**

```
// elt n'est pas dans L
suivant(elt)← premier(L);
premier(L)← elt
```

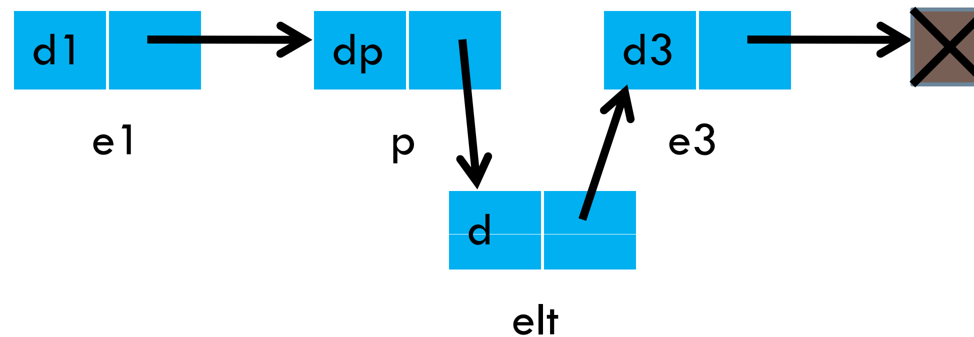
Liste : insertion d'un élément

- On insère un élément elt après un autre p.
- On suppose que elt n'est pas déjà dans la liste et que p y est (sinon que se passe-t-il ?)



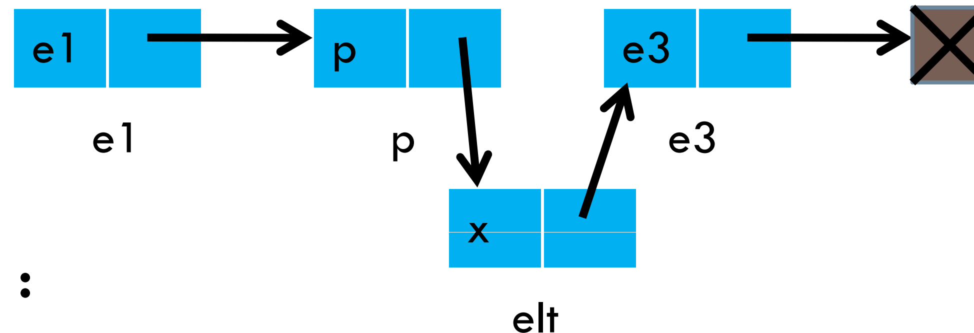
Liste : insertion d'un élément

- On insère un élément elt après un autre p.
- On suppose que elt n'est pas déjà dans la liste et que p y est (sinon que se passe t'il ?)



Liste : insertion d'un élément

- On insère un élément elt après un autre p.
- On suppose que elt n'est pas déjà dans la liste et que p y est (sinon que se passe t'il ?)



- Principes :
  - Le suivant de elt devient le suivant de p
  - Le suivant de p devient elt
  - Attention à l'ordre de mise à jour !

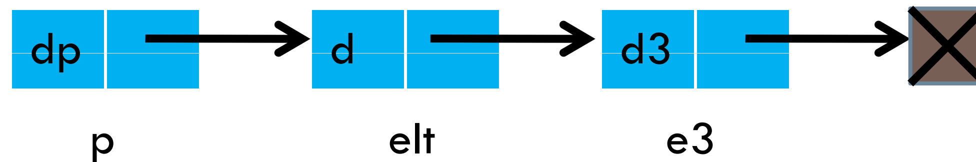
Liste : insertion d'un élément

- **insèreAprès(elt,p,L)**

```
//elt pas dans L, p est dans L
suivant(elt)← suivant(p);
suivant(p)← elt
```

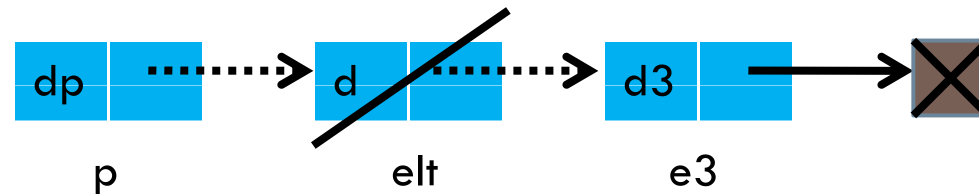
Liste : suppression d'un élément

- On supprime un élément de la liste.
  - On a besoin du précédent ! (Pourquoi ?)
  - Le premier peut changer !
  - Le suivant du précédent devient le suivant de elt



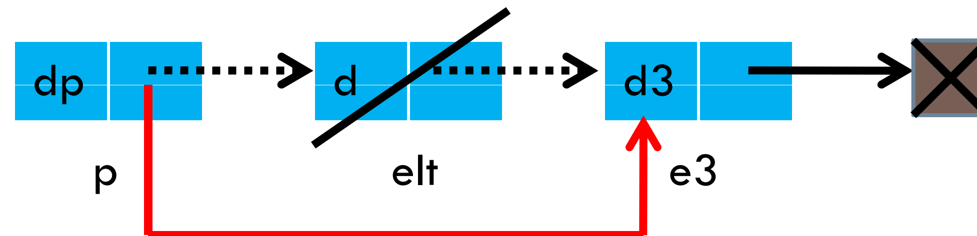
Liste : suppression d'un élément

- On supprime un élément de la liste.
  - On a besoin du précédent ! (Pourquoi ?)
  - Le premier peut changer !
  - Le suivant du précédent devient le suivant de elt



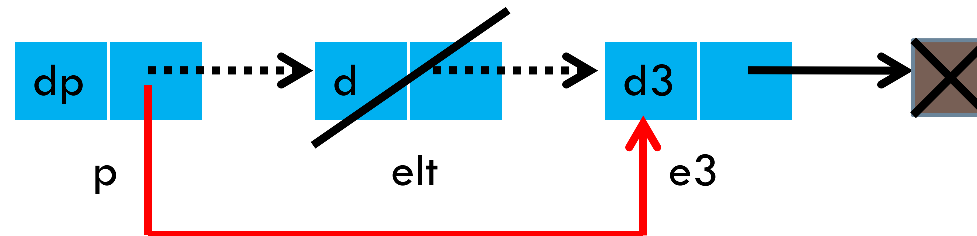
Liste : suppression d'un élément

- On supprime un élément de la liste.
  - On a besoin du précédent ! (Pourquoi ?)
  - Le premier peut changer !
  - Le suivant du précédent devient le suivant de elt



Liste : suppression d'un élément

- Gestion de tous les cas
  - elt est le premier
  - elt n'est pas le premier
  - p est bien le précédent de elt



Liste : suppression d'un élément

- **supprime(elt,p,L)**

```
//elt est dans L, p son précédent
if (premier(L) = elt){
   premier(L) ← suivant(elt);
} else {
   if (suivant(p) = elt){
       suivant(p)← suivant(elt);
   }
}
```

Plan

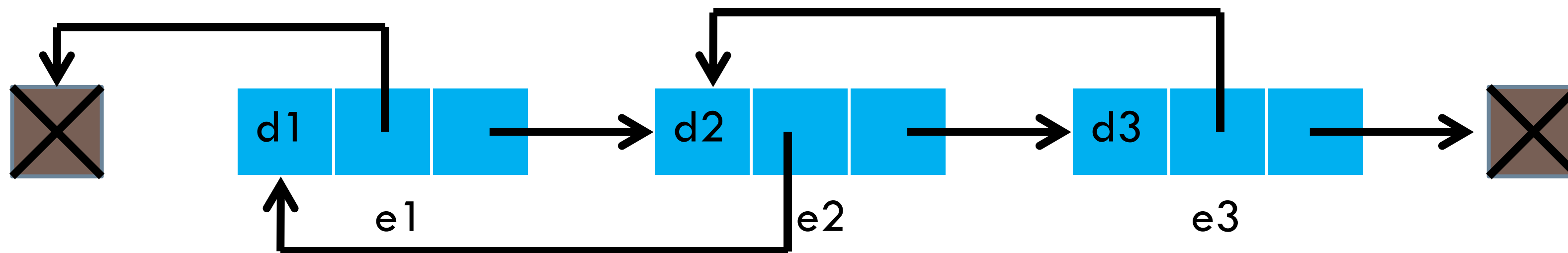
- Présentation
- **Listes doublement chaînées**
- Implémentation
- Exemple d'utilisation de listes

Liste

- **Liste simplement chaînée :**
  - **donnée(elt)** désigne la donnée associée à l'élément elt
  - **suivant(elt)** désigne l'élément suivant elt
- **Liste doublement chaînée :**
  - **donnée(elt)** désigne la donnée associée à l'élément elt
  - **suivant(elt)** désigne l'élément suivant elt
  - **précédent(elt)** désigne l'élément précédant elt

Liste doublement chaînée

- Représentation : ⟶ lien



- premier(L)=e1
- suivant(e1) = e2; précédent(e1) = nil
- suivant(e2) = e3; précédent(e2) = e1
- suivant(e3) = nil; précédent(e3) = e2

Liste : opérations

- Trois opérations principales
  - Parcours de la liste
  - Ajout d'un élément
  - Suppression d'un élément

- A partir de là d'autres opérations vont être obtenues : recherche de donnée, remplacement, concaténation de liste, fusion de listes, etc.

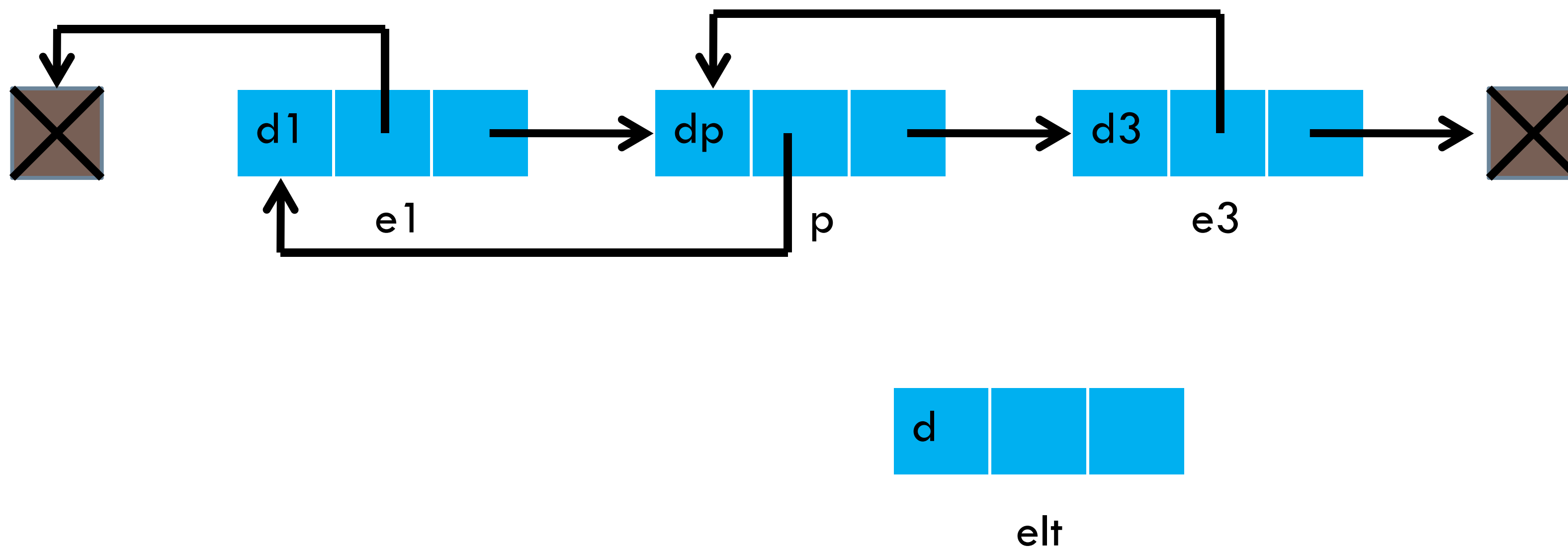
Liste : ajout d'un élément

- **ajouteAuDébut(elt,LD)**

```
//elt n'est pas dans LD
suivant(elt)← premier(LD);
précédent(premier(LD))← elt;
précédent(elt)← nil;
premier(LD)← elt
```

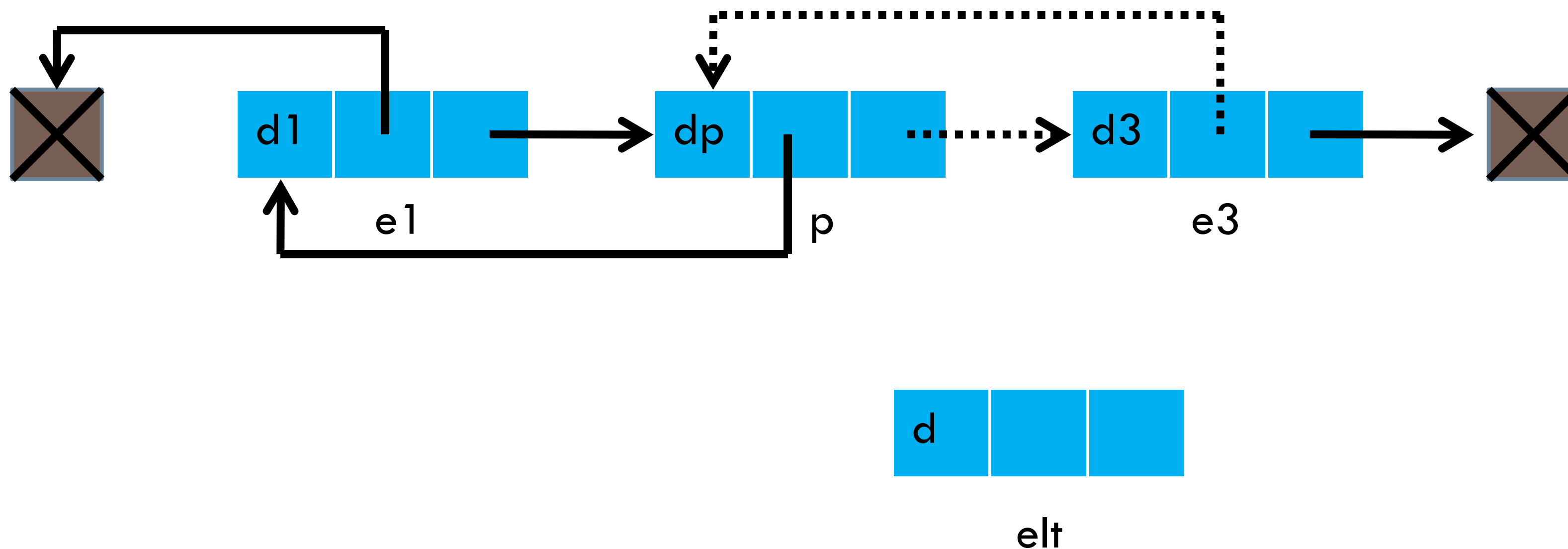
Liste : insertion d'un élément

- On insère un élément elt après un autre p.
- On suppose que elt n'est pas déjà dans la liste et que p y est (sinon que se passe t'il ?)



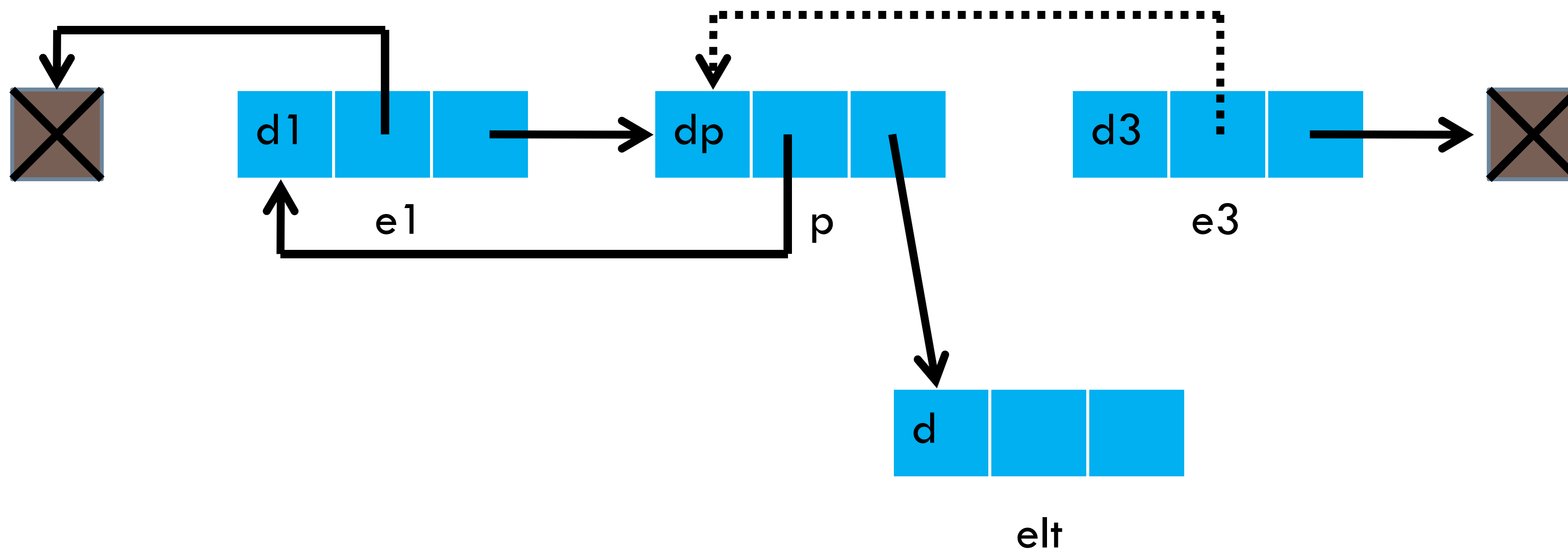
Liste : insertion d'un élément

- On insère un élément elt après un autre p.
- On suppose que elt n'est pas déjà dans la liste et que p y est (sinon que se passe-t-il ?)



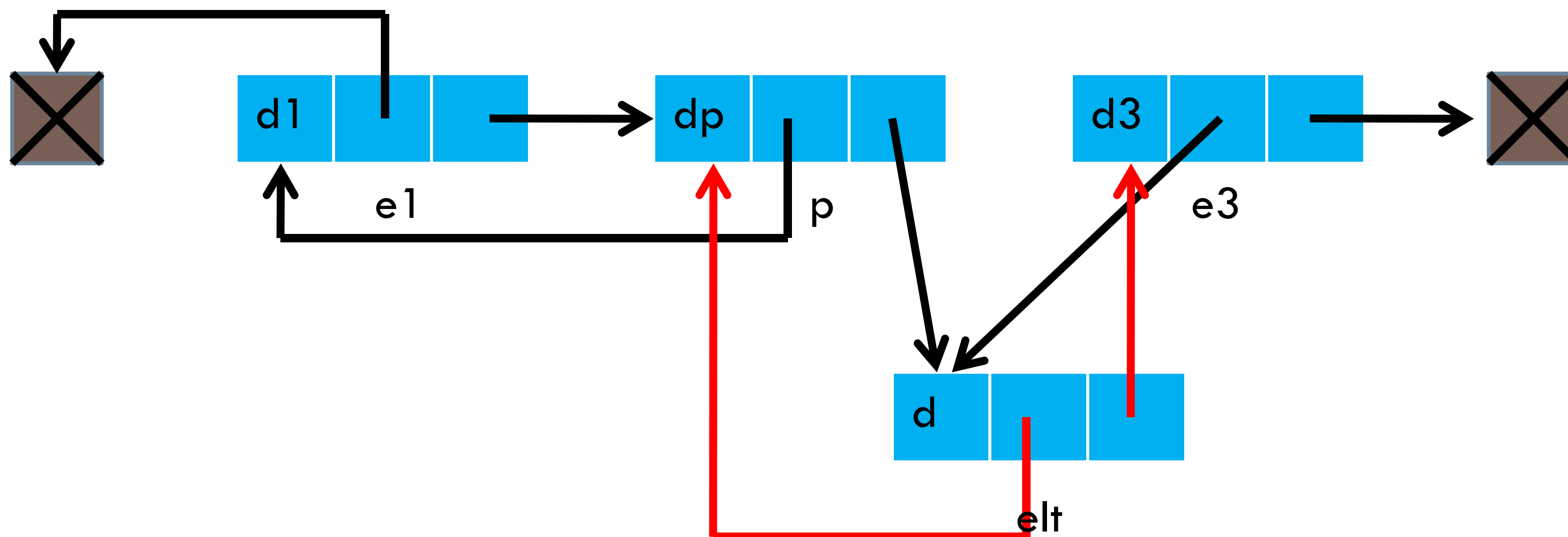
Liste : insertion d'un élément

- On insère un élément elt après un autre p.
- On suppose que elt n'est pas déjà dans la liste et que p y est (sinon que se passe-t-il ?)



Liste : insertion d'un élément

- On insère un élément elt après un autre p.
- **Sont modifiés** : suivant(p), précédent(e3) et suivant(elt) ; précédent(elt)



Liste : insertion d'un élément

- **insèreAprès(elt,p,LD)**

```
//elt pas dans LD, p dans LD
suivant(elt)← suivant(p);
précédent(elt)← p;
précédent(suivant(p))← elt;
suivant(p)← elt;
```

Liste : insertion d'un élément

- **insèreAprès(elt,p,LD)**

```
//elt pas dans LD, p dans LD
suivant(elt)← suivant(p);
précédent(elt)← p;
précédent(suivant(p))← elt;
suivant(p)← elt;
```

- **Mauvais code ! Pourquoi ?**

Liste : insertion d'un élément

- **insèreAprès(x,y,LD)**

```
//elt pas dans LD, p dans LD
suivant(elt)← suivant(p);
précédent(elt)← p;
if (suivant(p)≠ nil){
   précédent(suivant(p))← elt;
}
suivant(p)← elt;
```

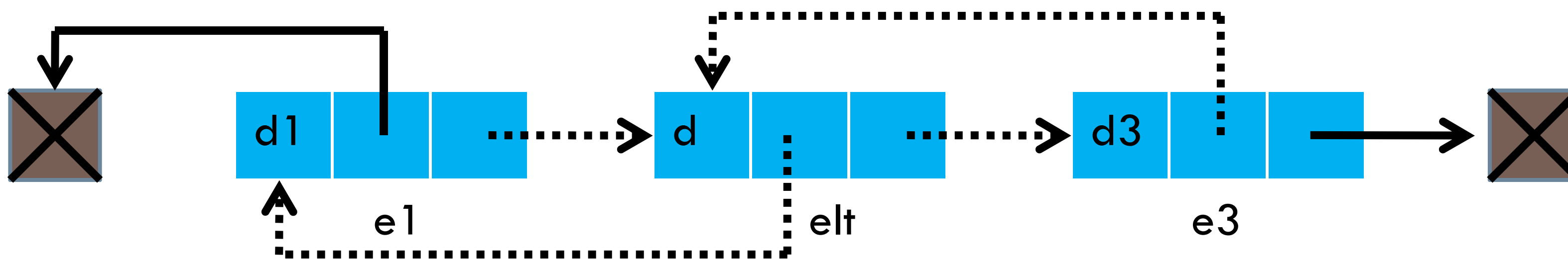
Liste : suppression d'un élément

- On supprime un élément de la liste.
  - On n'a pas besoin du précédent ! (Pourquoi ?)
  - Le premier peut changer !
  - Le suivant du précédent devient le suivant de elt
  - Le précédent du suivant devient le précédent de elt

d1
d
d3
e1
elt
e3

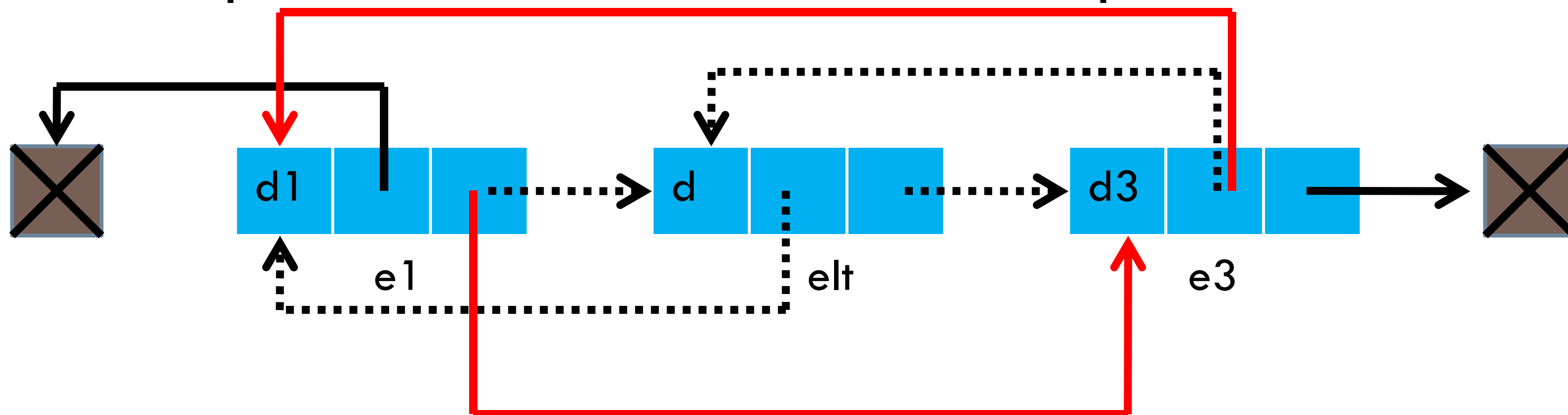
Liste : suppression d'un élément

- On supprime un élément de la liste.
  - On n'a pas besoin du précédent ! (Pourquoi ?)
  - Le premier peut changer !
  - Le suivant du précédent devient le suivant de elt
  - Le précédent du suivant devient le précédent de elt



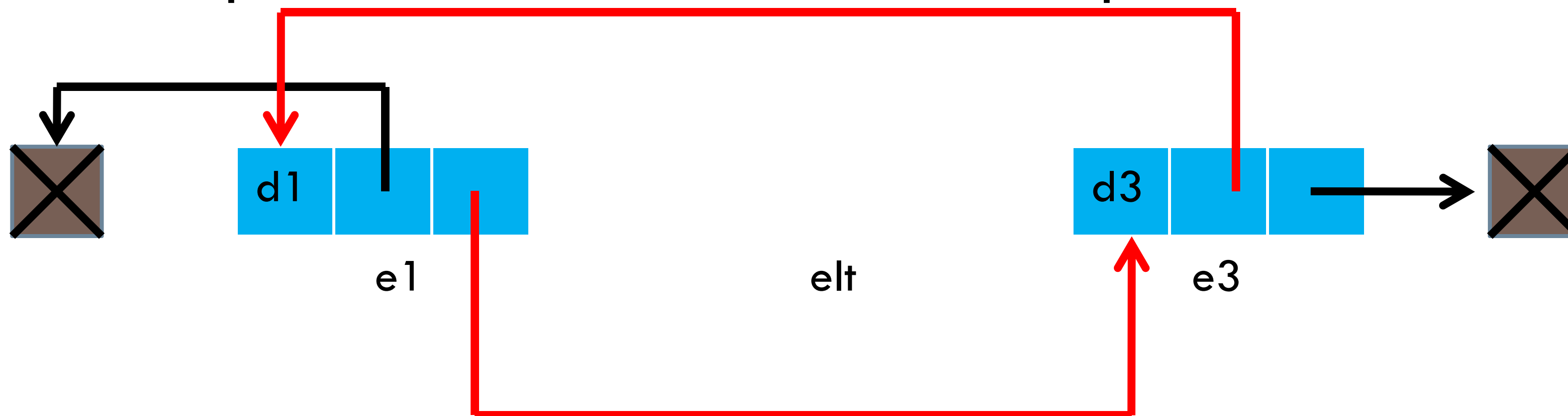
Liste : suppression d'un élément

- On supprime un élément de la liste.
  - On n'a pas besoin du précédent ! (Pourquoi ?)
  - Le premier peut changer !
  - Le suivant du précédent devient le suivant de elt
  - Le précédent du suivant devient le précédent de elt



Liste : suppression d'un élément

- On supprime un élément de la liste.
  - On n'a pas besoin du précédent ! (Pourquoi ?)
  - Le premier peut changer !
  - Le suivant du précédent devient le suivant de elt
  - Le précédent du suivant devient le précédent de elt



Liste : suppression d'un élément

- **supprime(elt,LD)**

```
// elt dans LD
suiv ← suivant(elt);
prec ← précédent(elt);
if (prec = nil){
    premier(LD) ← suiv;
} else {
    suivant(prec)← suiv;
}
if (suiv ≠ nil){
    précédent(suiv)← prec;
}
```

Plan

- Présentation
- Listes doublement chaînées
- **Implémentation**
- Exemple d'utilisation de listes

Implémentation

- Par un tableau
- A l'aide de pointeur

Tableaux simulant une liste

- Tableau S des suivants
  - S[i] indice de l'élément suivant l'élément d'indice i
- Tableau P des précédents
  - P[i] indice de l'élément précédant l'élément d'indice i
- Liste 2 – 3 – 1 – 5 – 4
- S : [5,3,1,0,4]
- P : [3,0,2,5,1]

Pointeur

- Un pointeur est un type de données dont la valeur **pointe vers** une autre valeur.
- Obtenir la valeur vers laquelle un pointeur pointe est appelé **déréférencer** le pointeur.
- Un pointeur qui ne pointe vers aucune valeur aura la valeur **nil**

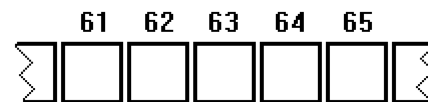
Pointeur Implémentation

- Un pointeur c'est un indice dans le grand tableau de la mémoire comme vu en TD

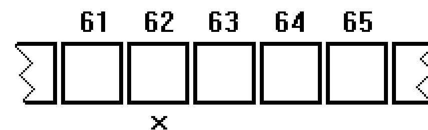
Pointeur Implémentation

- `int x; // Réserve un emplacement pour un entier en mémoire.`
- `x = 10; // Ecrit la valeur 10 dans l'emplacement réservé.`

- Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée.
- Physiquement cette valeur se situe en mémoire.



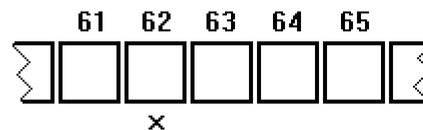
- int x;



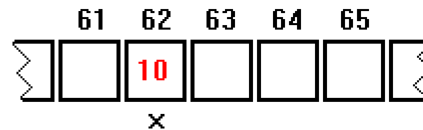
Pointeur Implémentation

- `int x; // Réserve un emplacement pour un entier en mémoire.`
- `x = 10; // Ecrit la valeur 10 dans l'emplacement réservé.`

- int x;



- x=10;

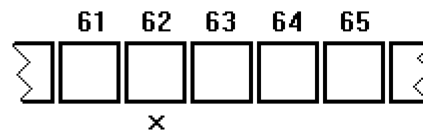


- &x : adresse de x en C : ici 62

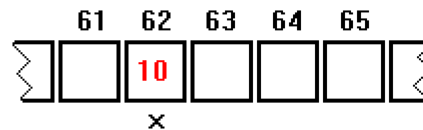
Pointeur Implémentation

- `int x; // Réserve un emplacement pour un entier en mémoire.`
- `x = 10; // Ecrit la valeur 10 dans l'emplacement réservé.`

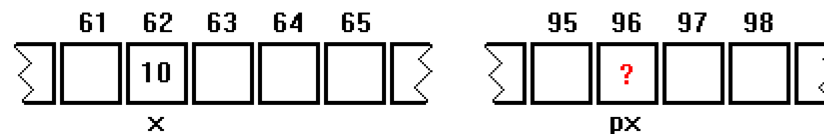
- int x;



- x=10;

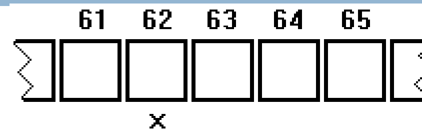


- En C : int* px; // pointeur sur un entier

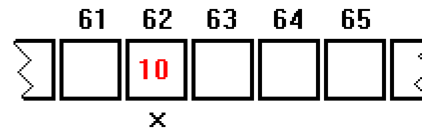


Pointeur Implémentation

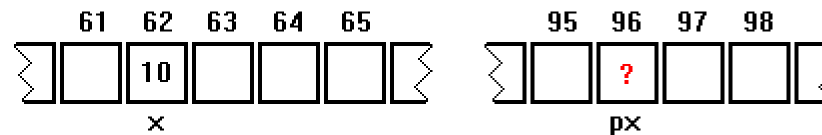
- int x;



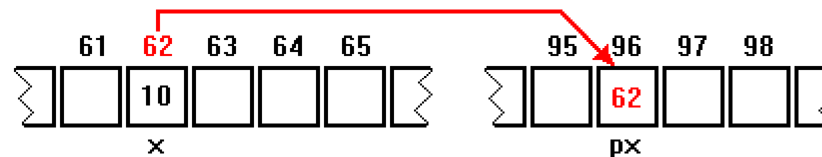
- x=10;



- En C : int* px; // pointeur sur un entier

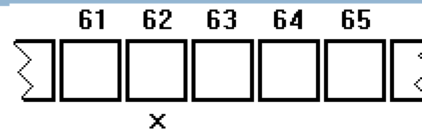


- px=&x; (adresse de x)

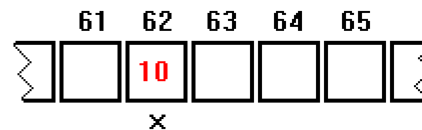


Pointeur Implémentation

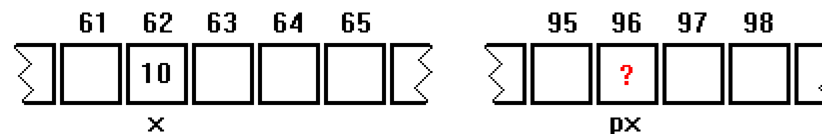
- int x;



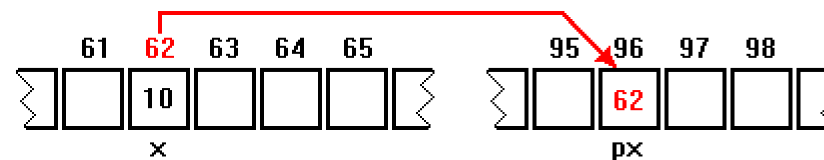
- x=10;



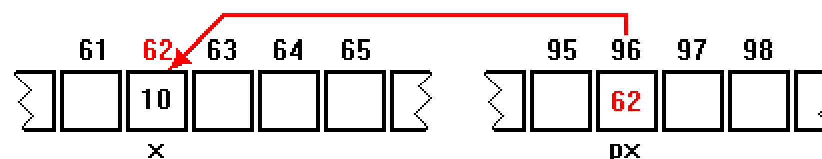
- En C : int* px; // pointeur sur un entier



- px=&x; (adresse de x)

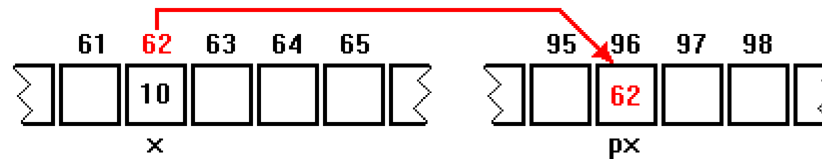


- int y=*px

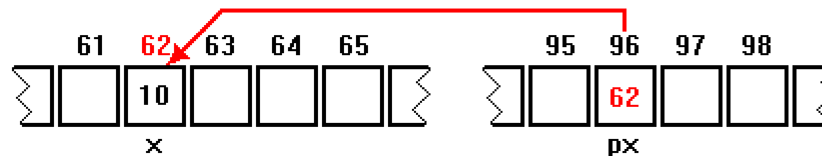


Pointeur Implémentation

- Si px contient l'adresse de x

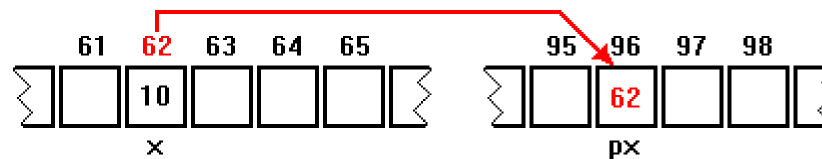


- Alors *px contient la valeur qui se trouve à l'adresse de x, donc la valeur de x

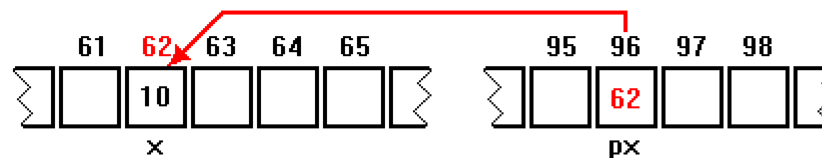


Pointeur Implémentation

- Si px contient l'adresse de x



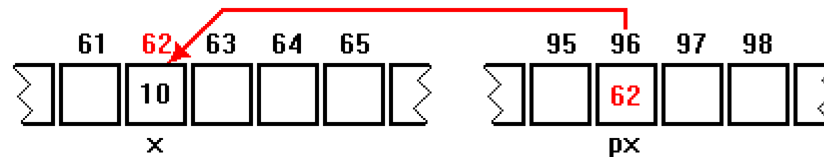
- Alors *px contient la valeur qui se trouve à l'adresse de x, donc la valeur de x



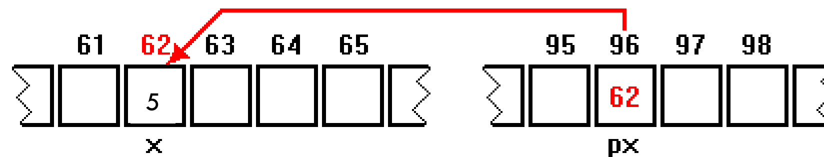
- Si je change la valeur qui se trouve à l'adresse de x, alors je change la valeur de x,

Pointeur Implémentation

- Si px contient l'adresse de x, donc *px la valeur de x



- Si je change la valeur à l'adresse de x, alors je change la valeur de x : *px=5



Pointeur Implémentation

- px=&x;
- Si je modifie *px alors je modifie x
- px=&y;
- Si je modifie *px alors je modifie y
- px=&bidule;
- Si je modifie *px alors je modifie bidule
- px désigne **l'objet pointé**
- *px modifie l'**objet pointé**

Pointeur et référence

- Une **référence** est une valeur qui permet l'accès en lecture et/ou écriture à une donnée située soit en mémoire principale soit ailleurs.

- Une référence n'est pas la donnée elle-même mais seulement une information de localisation

- Ressemble à quelque chose de connu, non ?

Pointeur et référence

- Le typage des références permet de manipuler les données référencées de manière abstraite tout en respectant leurs propres contraintes de type.
- Le type de référence le plus simple est le pointeur. Il s'agit simplement d'une adresse mémoire.
- Mais **pointeur typé = référence typé**
- Mais on peut changer le type d'un pointeur, pas d'une référence (cast/transtypage autorisé)

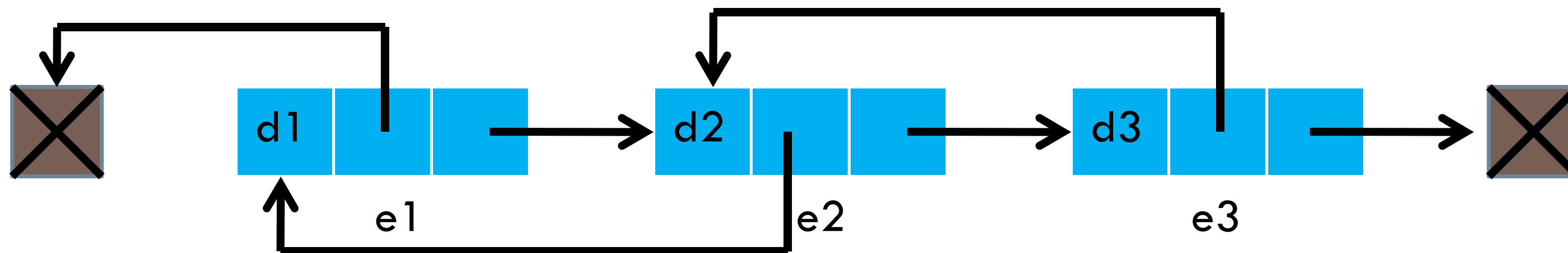
Pointeur et référence

- En Java : uniquement des références typées
  - 2 objet a et b de type MyObject
  - MyObject obj;
  - obj=a; si on modifie obj, alors on modifie a
  - obj=b; si on modifie obj, alors on modifie b
  - Fonc(obj) : obj est passé en entrée/sortie

- En C/C++ : pointeurs typés mais type changeable
  - 2 objet a et b de type MyObject
  - MyObject* obj;
  - obj=&a; si on modifie obj, alors on modifie a
  - obj=&b; si on modifie obj, alors on modifie b
  - Fonc(MyObject* obj): en entree/sortie
  - Fonc(MyObject obj) : en entree

Liste doublement chaînée

- Représentation : lien



- ListeElement
  - suivant : pointeur vers ListeElement
  - précédent : pointeur vers ListeElement
  - donnée : pointeur vers l'objet
- Liste
  - Premier : pointeur vers ListeElement

Liste doublement chaînée


```
class ListElt {
  ListElt _suiv;
  ListElt _prec;
  MaClasseDonnee _data;
  ListElt(){
      _suiv=null;
      _prec=null;
      _data=null;
  }
}
```

Liste doublement chaînée


```
Class List {
ListElt _premier;
List(){
        _premier=null;
}
void ajouterEnTete(ListElt elt){ // on ne traite pas les cas elt == null
        elt._suiv = _premier;
        if (_premier != null)
                    _premier._prec=elt;
        _premier = elt;
        elt._prec=null;
}
void insererApres(ListElt prec, ListElt elt){
        if (prec == null) ajouterEnTete(elt);
        else {
            elt._suiv=prec._suiv;
            elt._prec=prec;
            if (prec._suiv != null){prec._suiv._prec=elt;}
            prec._suiv=elt;
        }
}
void supprimer(ListElt elt){
        if (elt == _premier){_premier = elt._suiv;}
        else { // elt._prec n'est pas null
            _elt._prec._suiv=elt._suiv;
        }
        if (elt._suiv != null){elt._suiv._prec=elt._prec;}
}
```

Plan

- Présentation
- Listes doublement chaînées
- Implémentation
- **Exemple d'utilisation de listes**

Pile : implémentation par une liste

- Une liste L simplement chaînée
- Sommet(P) : renvoyer premier(L)
- Empiler(P,elt) : ajouterEnTête(L,elt)
- Dépiler(P) : supprimerPremier(L)
- estVide(P) : renvoyer estVide(L)

- Nécessite par rapport à un tableau
  - Plus de place
  - Plus d'opérations
  - Plus d'allocations

File : implémentation par une liste

- Une liste L simplement chaînée avec fin
- début(F) : renvoyer premier(L)
- Enfiler(F,elt) : ajouterEnFin(L,elt)
- Défiler(F) : supprimerPremier(L)
- estVide(F) : renvoyer estVide(L)

- Beaucoup plus simple qu'un tableau cette fois
  - Gestion mémoire plus complexe