Partie III

La programmation orientée objets

Informatique 1
Introduction à la programmation

Mission 11 : INTRODUCTION

Les listes chaînées

Kim Mens – Siegfried Nijssen – Charles Pecheur

Mission 11 : Matière à lire

Objects

- 1 - Classes and objects - Basics
- 2 - Classes and objects - Advanced
- 3 - Even more object-oriented programming
- 4 - Overloading and polymorphism
- 5 - Collections of objects
- 6 - Inheritance
- 7 - Linked lists
- Appendix - Source code of phone class
- Appendix - Worked out example: accounts
- Appendix - Source code of card game
- Appendix - Source code of linked lists

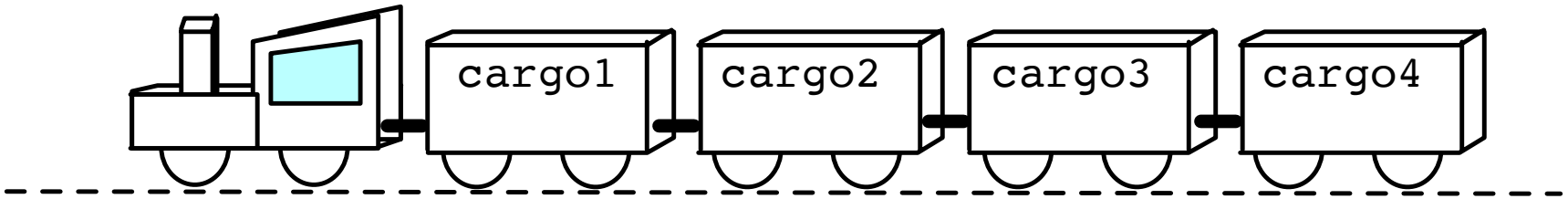
listes chaînées

principes

opérations

variantes

classes internes



Définition

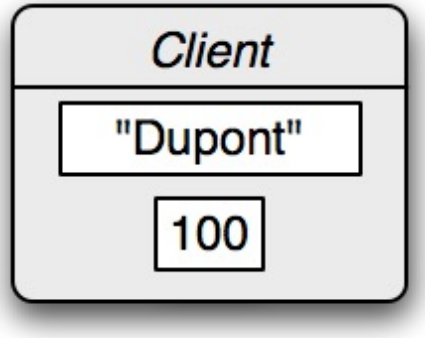
Une **structure de données** est une manière de grouper et d'organiser des données afin de faciliter certains opérations sur ces données.

> Par exemple, traverser tous les éléments d'une collection de données.

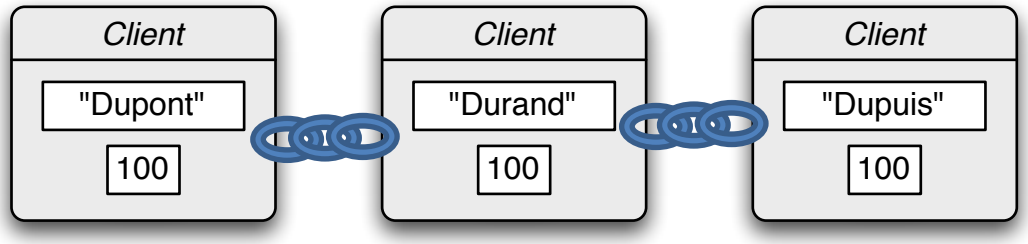
Une **liste chaînée** est une structure de données, groupant une collection d'éléments en une séquence de nœuds chainées qui peut être parcouru séquentiellement.

Réaliser une liste

```
class Client:
  def __init__(self,n,c):
    self.name = n
    self.credit = c
```

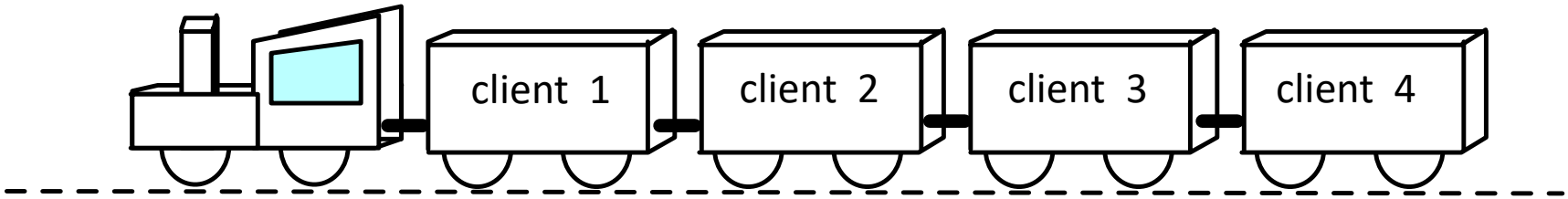


Une **liste** de **Client** ?

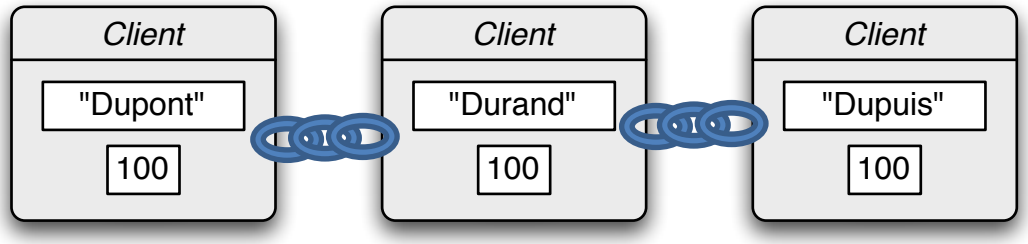


structure chaînée

Une liste chainée



Une liste chaînée peut être implémentée en chaînant un ensemble de nœuds.



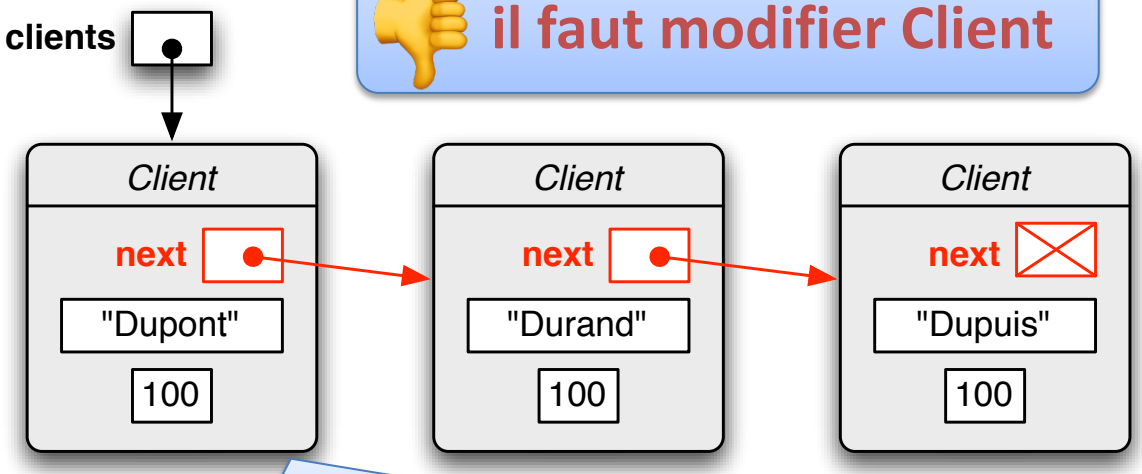
structure chaînée

Chaînage des éléments

```
class Client:
   def __init__(self,n,c,nx):
      self.name = n
      self.credit = c
      self.next = nx
```

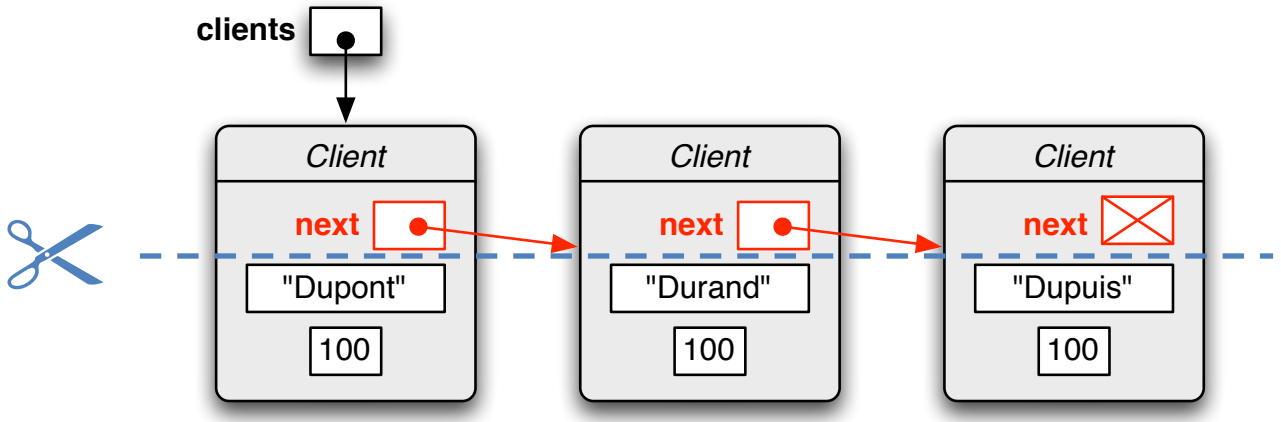
1 chaîner les éléments

il faut modifier Client



un Client dans une seule liste

Séparer la chaîne

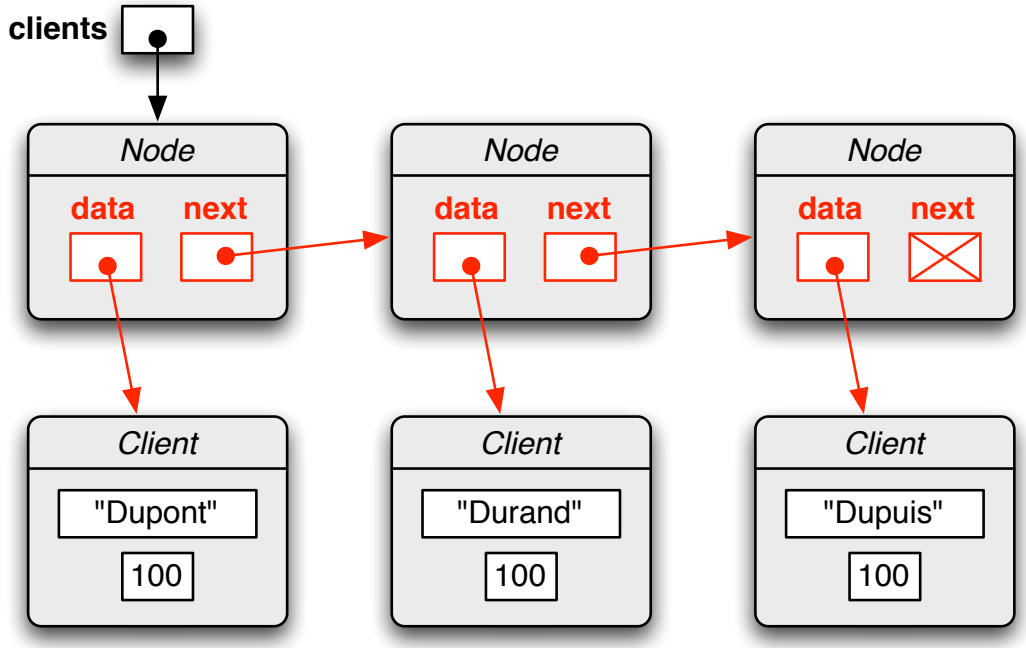


Chaîne de nœuds

Chaque nœud est un objet contenant:

- Un attribut `data` pour stocker une donnée (par ex. une instance de la classe `Client`)
- Un attribut `next` contenant une référence vers le nœud suivant

Les nœuds servent essentiellement à chaîner les données.



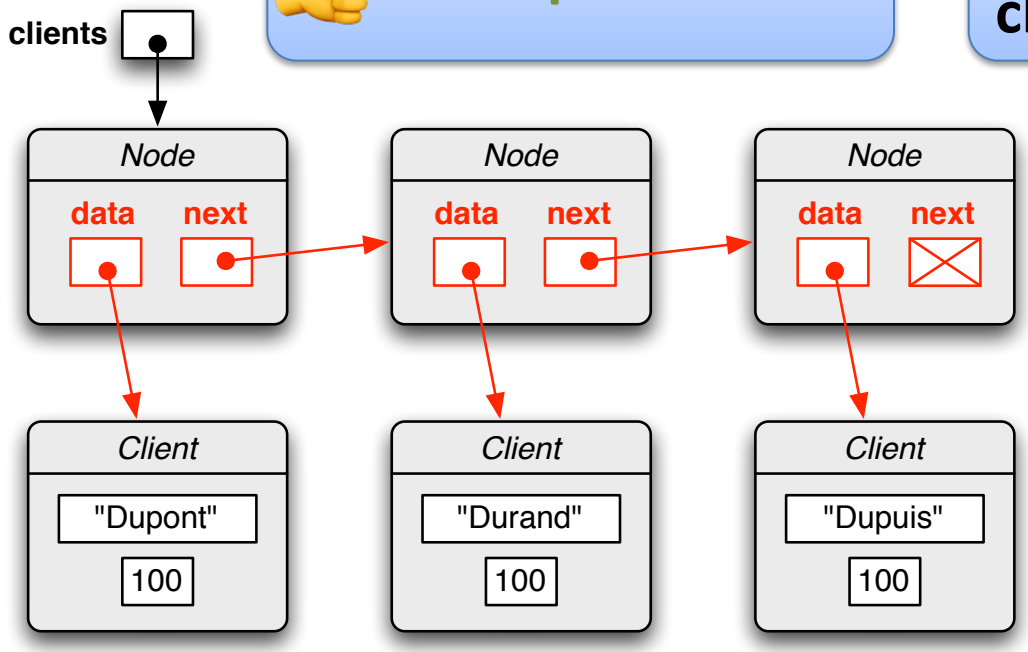
Chaîne de nœuds

```
class Client:
  def __init__(self,n,c):
    self.name = n
    self.credit = c
```

```
class Node:
  def __init__(self,d,nx):
    self.data = d
    self.next = nx
```

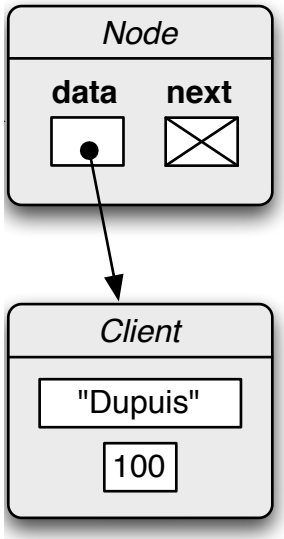
Client pas modifié

Classe séparée pour chaîner des nœuds



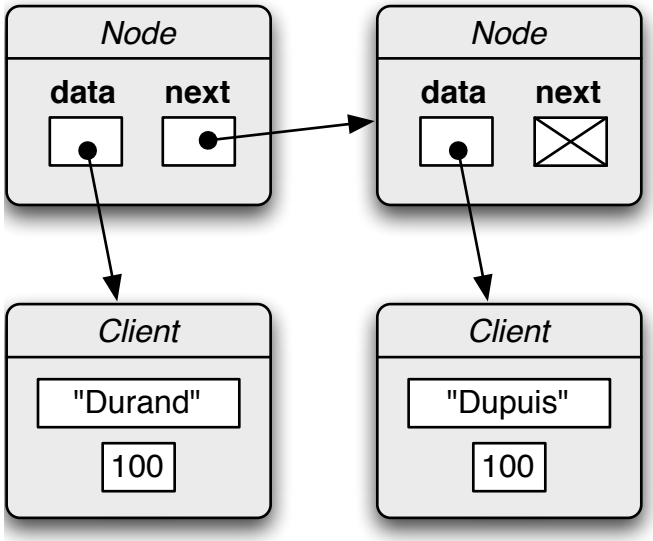
OK pour plusieurs listes

Liste simplement liée



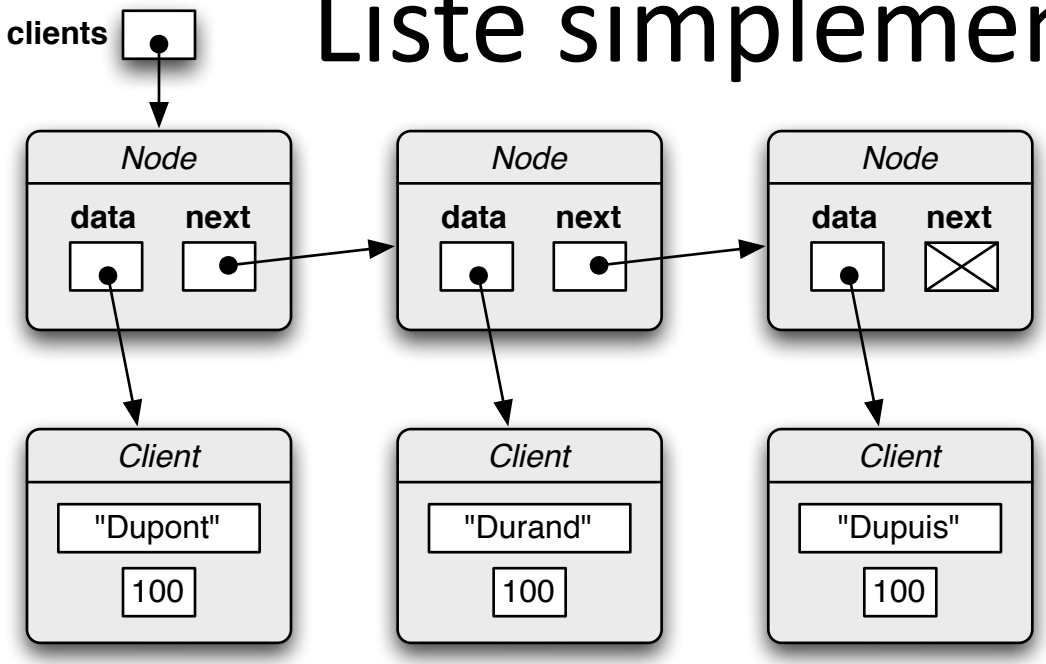

```
>>> troisieme  = Node(Client("Dupuis",100),None)
```

Liste simplement liée



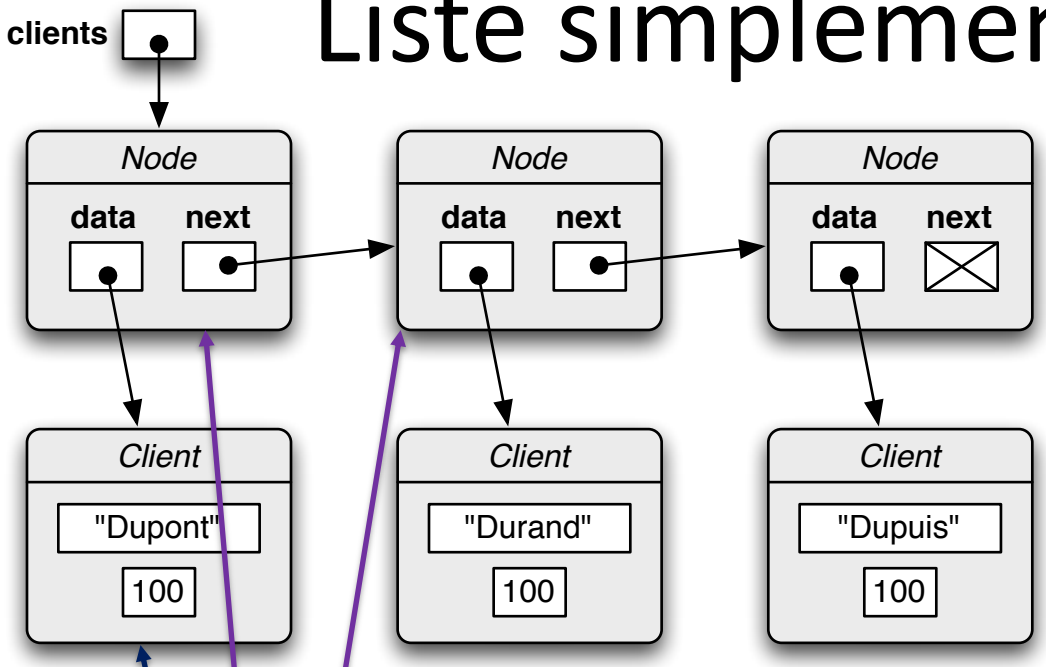

```
>>> troisieme  = Node(Client("Dupuis",100),None)
>>> deuxieme   = Node(Client("Durand",100),troisieme)
```

Liste simplement liée




```
>>> troisieme  = Node(Client("Dupuis",100),None)
>>> deuxieme   = Node(Client("Durand",100),troisieme)
>>> clients    = Node(Client("Dupont",100),deuxieme)
```

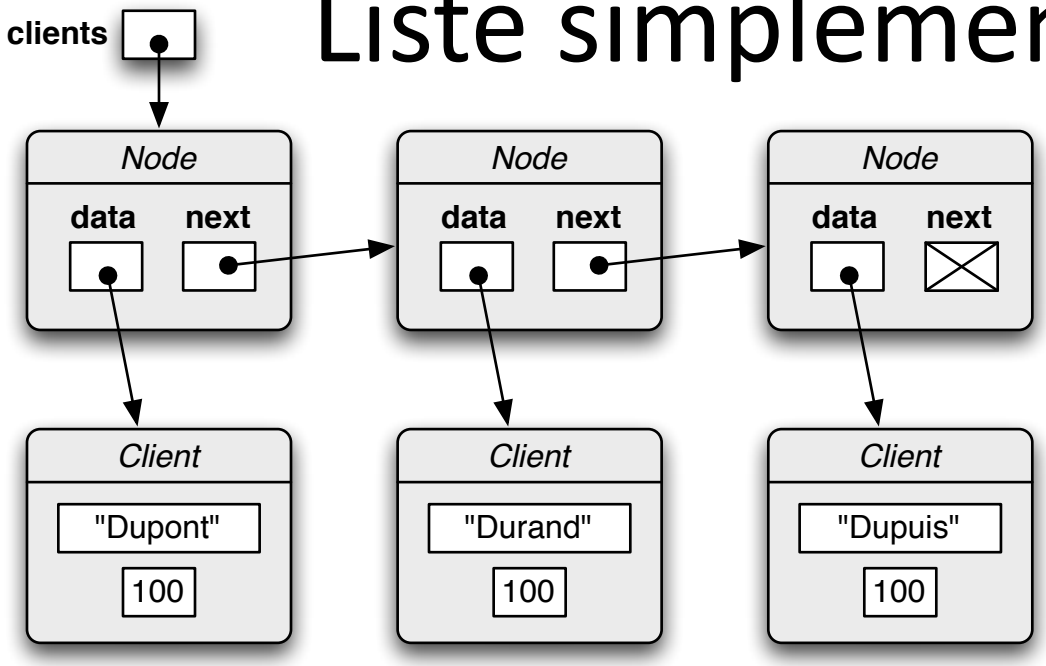
Liste simplement liée




```
>>> troisieme  = Node(Client("Dupuis",100),None)
>>> deuxieme   = Node(Client("Durand",100),troisieme)
>>> clients    = Node(Client("Dupont",100),deuxieme)
>>> print(clients)
<__main__.Node instance at 0x10210c320>
>>> print(clients.data)
<__main__.Client instance at 0x101edeab8>
>>> print(clients.next)
<__main__.Node instance at 0x10210c098>
```

Dupont : 100

Liste simplement liée




```
>>> clients = Node(Client("Dupont",100),
                   Node(Client("Durand",100),
                        Node(Client("Dupuis",100),None)))
```

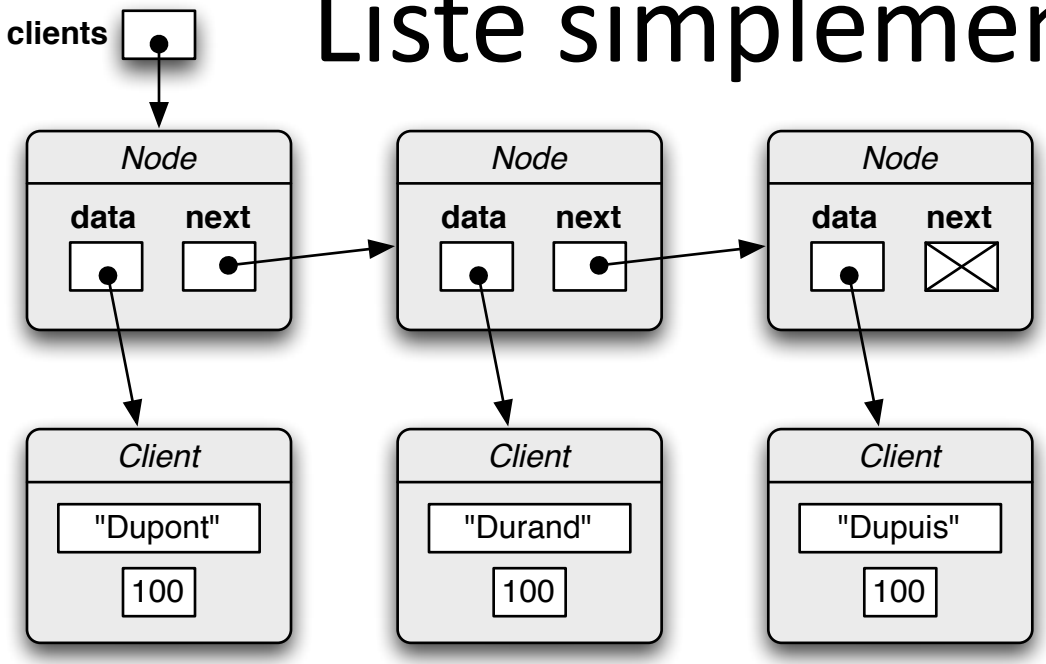
Chaîne de nœuds

```
class Client:
  def __init__(self,n,c):
    self.name = n
    self.credit = c
  def __str__(self):
    return str(self.name) + ": " + str(self.credit)
```

```
class Node:
  def __init__(self,d,nx):
    self.data = d
    self.next = nx
```

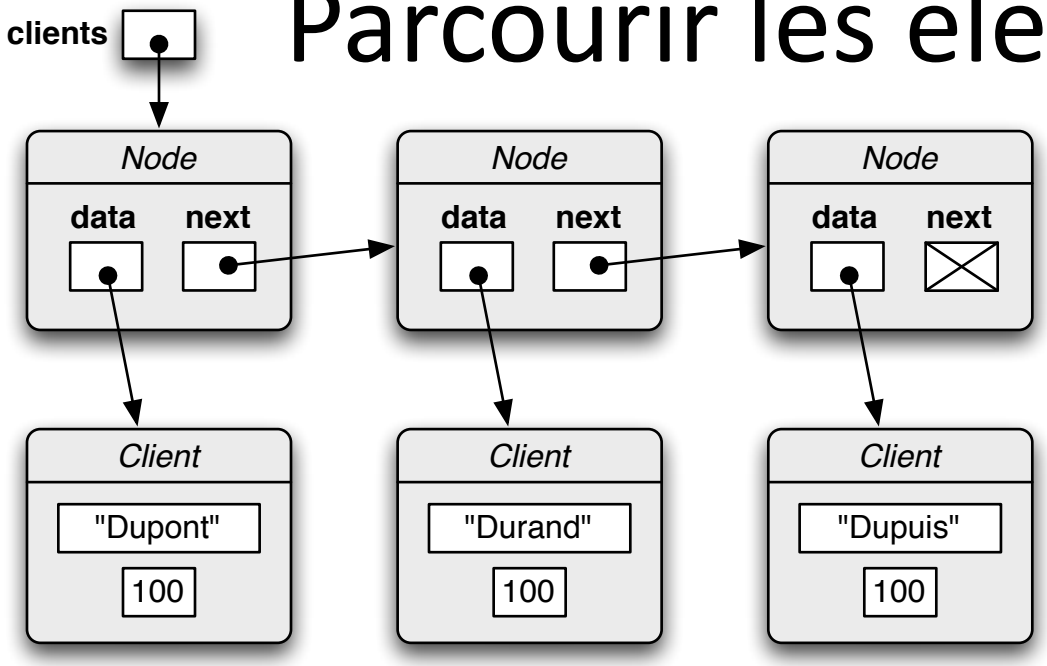
```
>>> print(Client("Dupont",100))
Dupont: 100
```

Liste simplement liée




```
>>> clients = Node(Client("Dupont",100),
                   Node(Client("Durand",100),
                        Node(Client("Dupuis",100),None)))
>>> print(clients.data)
Dupont: 100
>>> print(clients.next.data)
Durand: 100
>>> print(clients.next.next.data)
Dupuis: 100
```

Parcourir les éléments




```
# boucle while :
n = clients
while n is not None:
    print(n.data)
    n = n.next
```

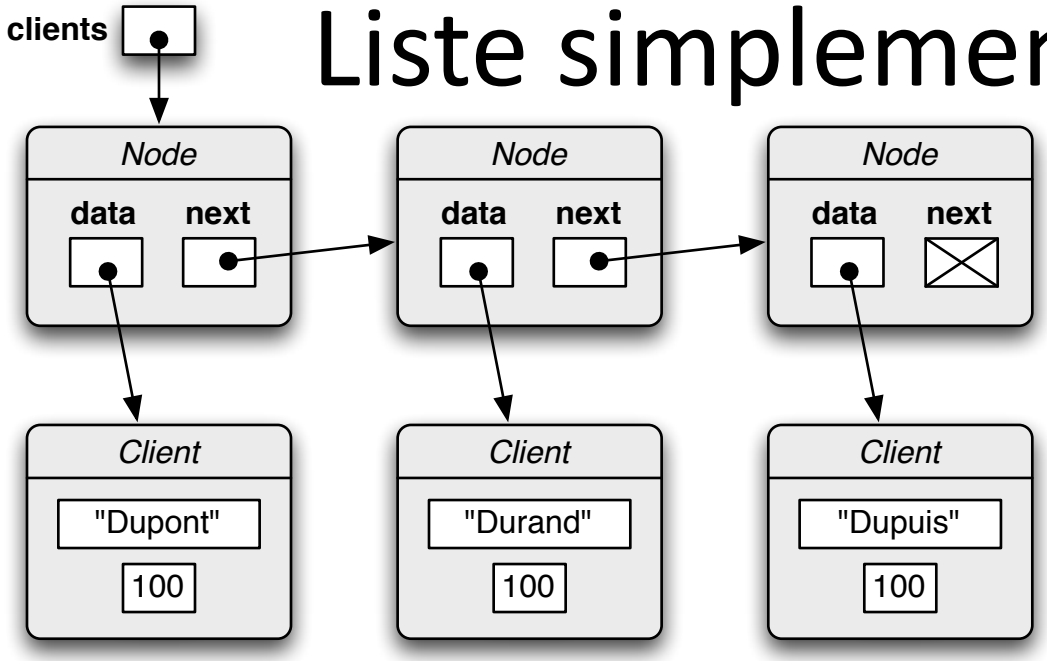
```
Dupont: 100
Durand: 100
Dupuis: 100
```

Chaîne de nœuds

```
class Client:
  def __init__(self,n,c):
    self.name = n
    self.credit = c
  def __str__(self):
    return str(self.name) + ": " + str(self.credit)
```

```
class Node:
  def __init__(self,d,nx):
    self.data = d
    self.next = nx
  def __str__(self):
    s = str(self.data)
    n = self.next
    while n is not None: # boucle while
        s += " " + str(n.data)
        n = n.next
    return s
```

Liste simplement liée

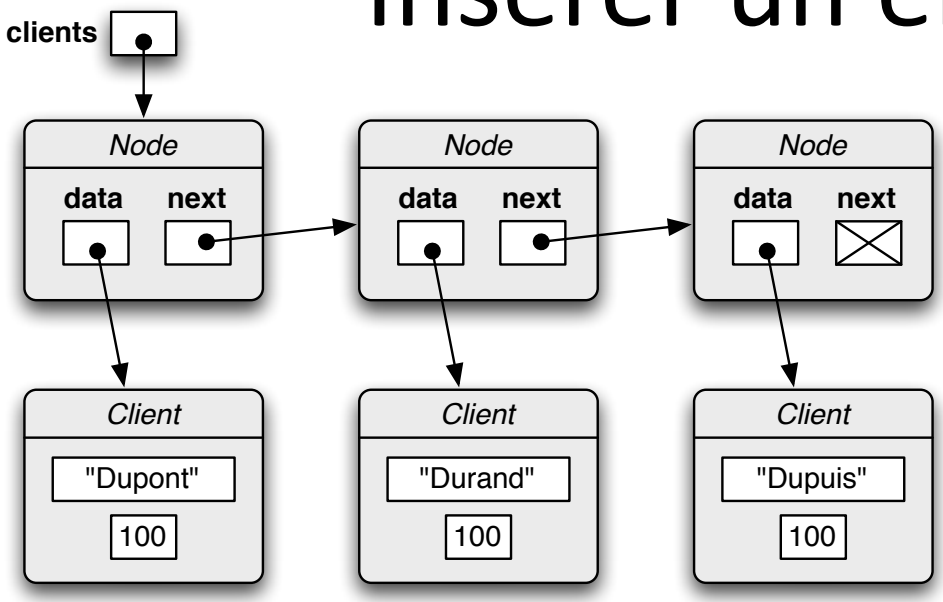


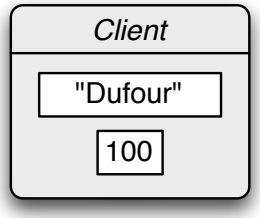

```
>>> print(clients)
Dupont: 100 Durand: 100 Dupuis: 100
```

```
class Client:
  def __init__(self,n,c):
    self.name = n
    self.credit = c
  def __str__(self):
    return str(self.name) + ": "
           + str(self.credit)
```

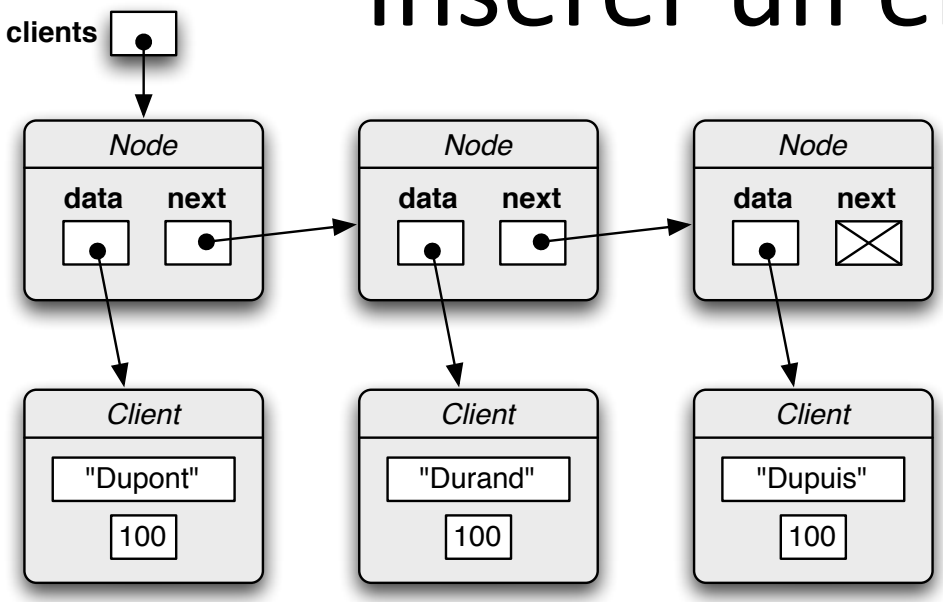
```
class Node:
  def __init__(self,d,nx):
    self.data = d
    self.next = nx
  def __str__(self):
    s = str(self.data)
    n = self.next
    while n is not None: # boucle while
        s += " " + str(n.data)
        n = n.next
    return s
```

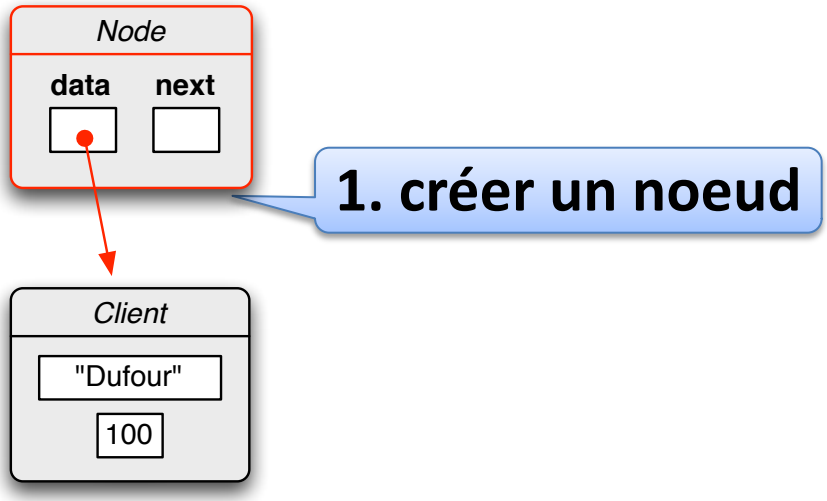
Insérer un élément



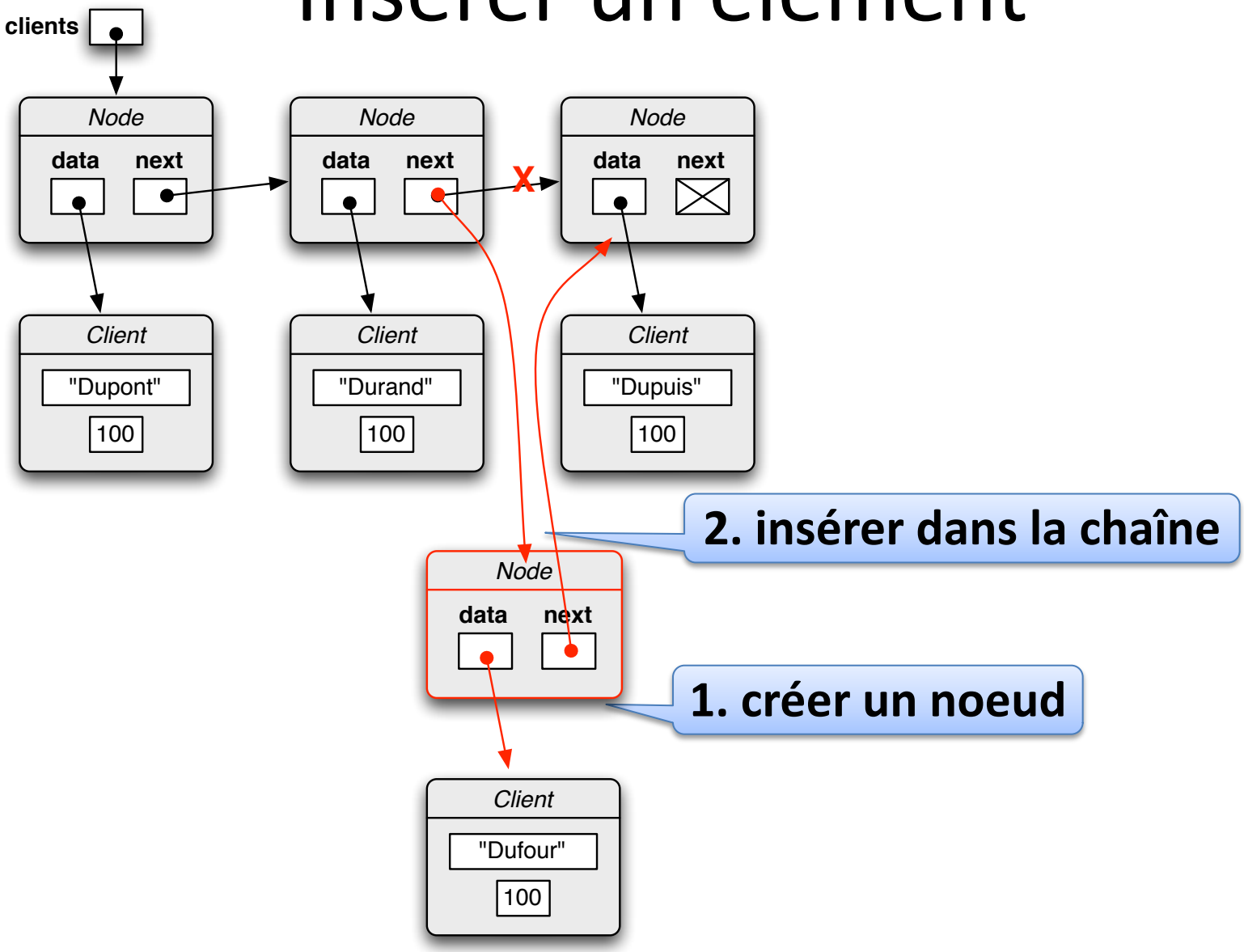


Insérer un élément

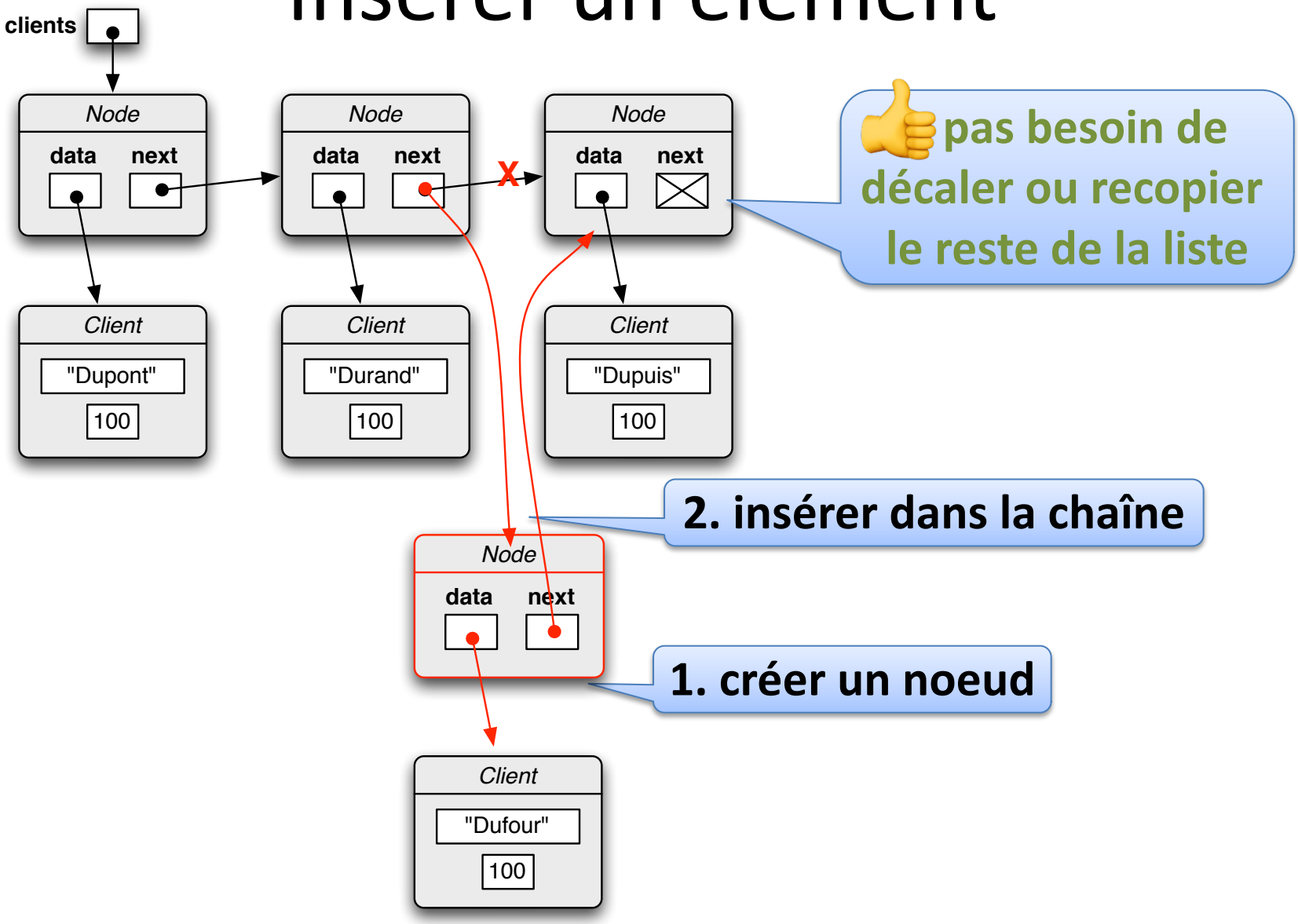




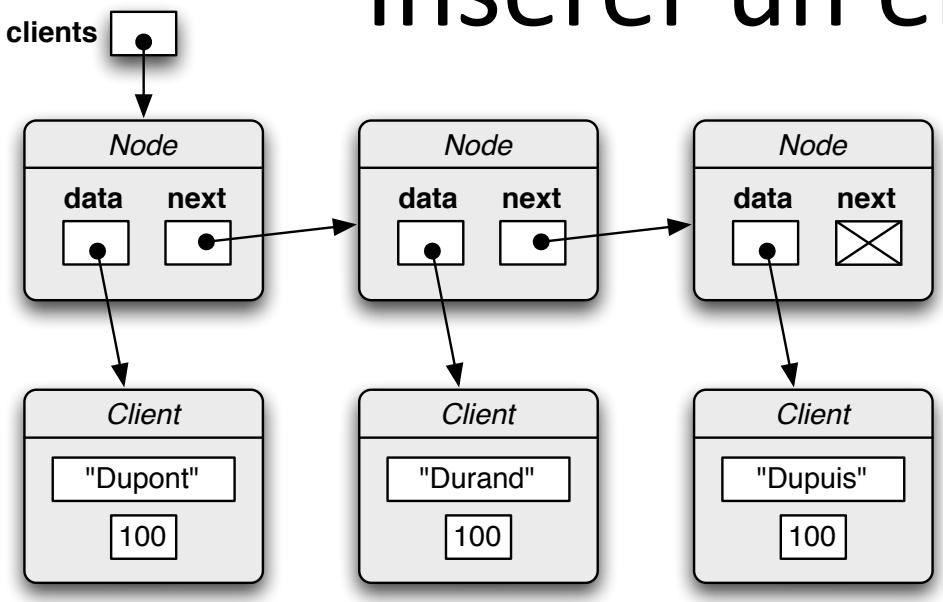
Insérer un élément

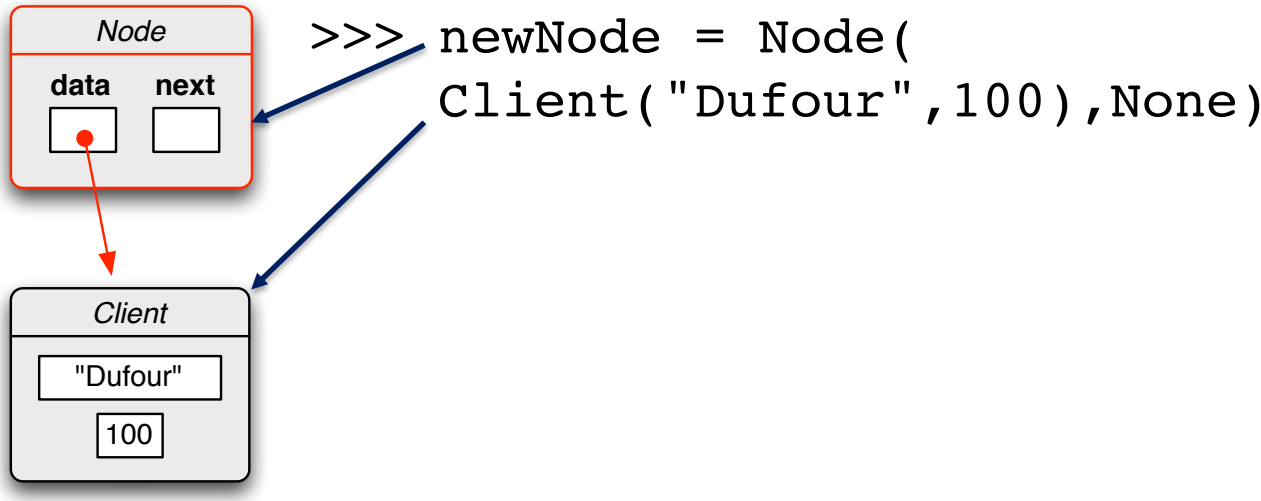


Insérer un élément

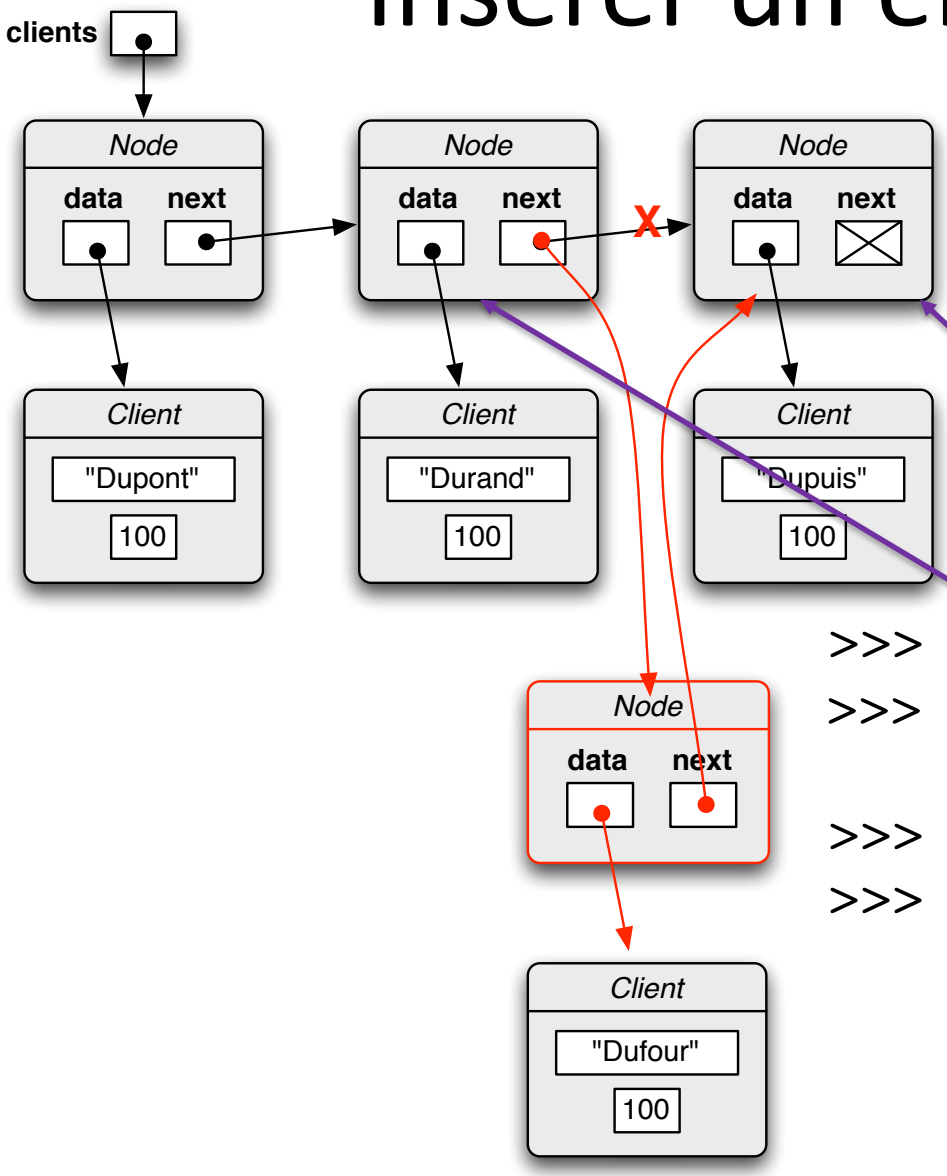


Insérer un élément



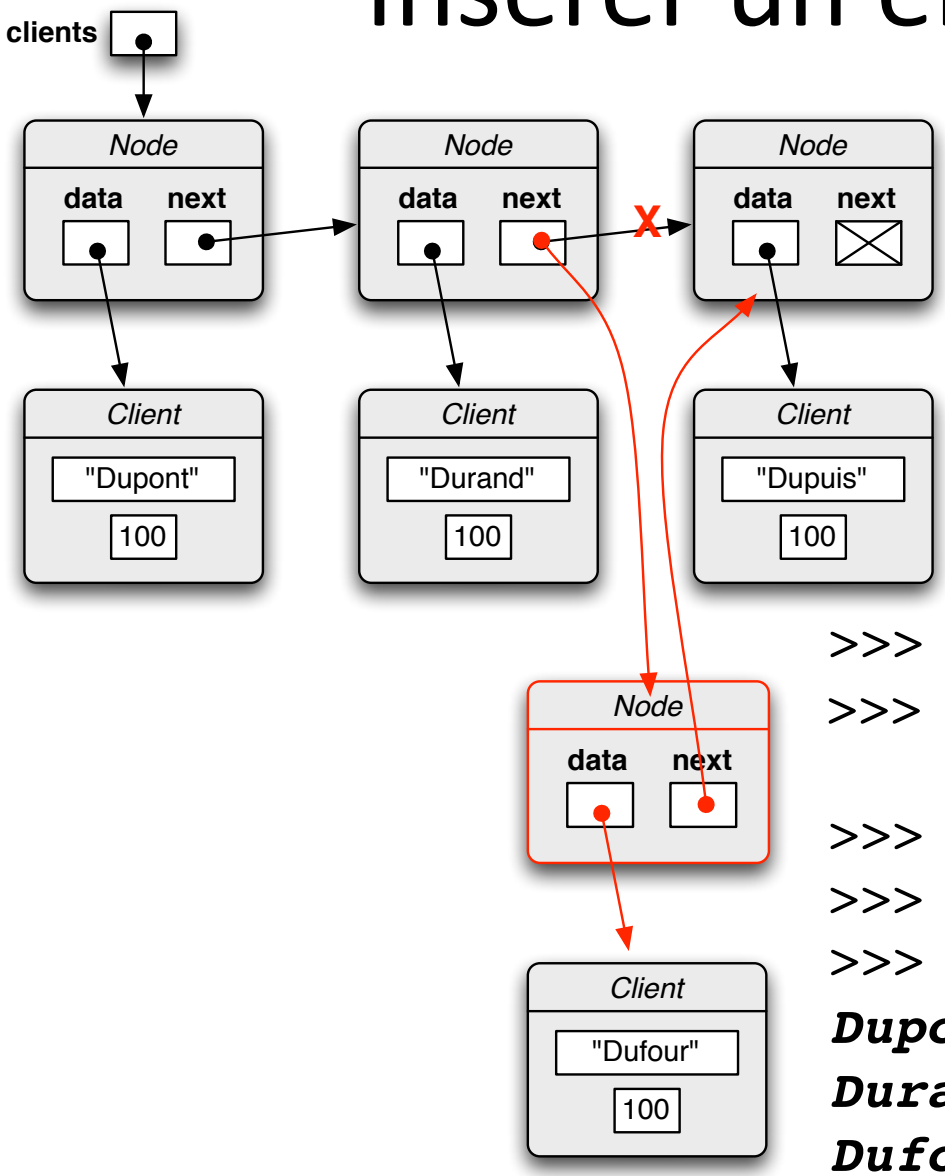


Insérer un élément



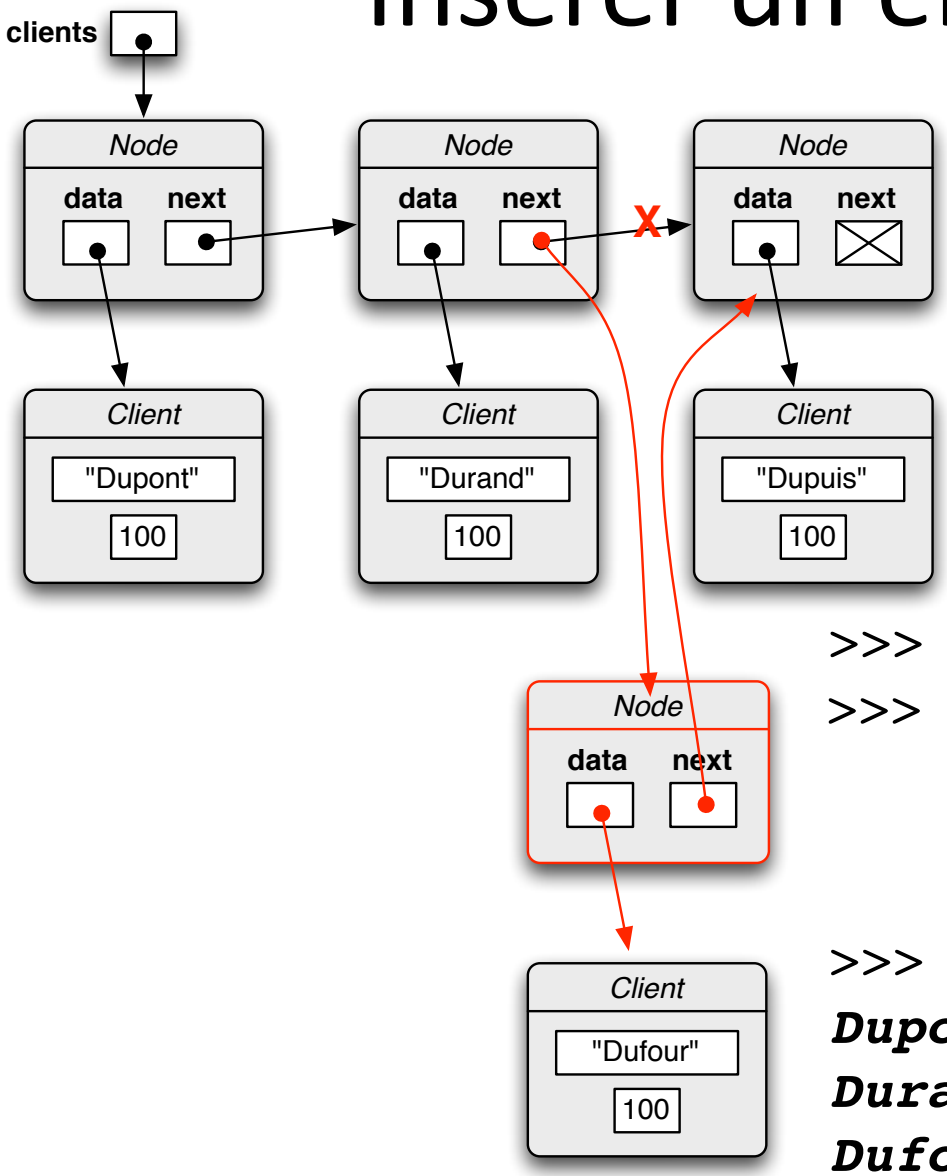

```
>>> n = clients.next
>>> newNode = Node(
    Client("Dufour",100),None)
>>> newNode.next = n.next
>>> n.next = newNode
```

Insérer un élément



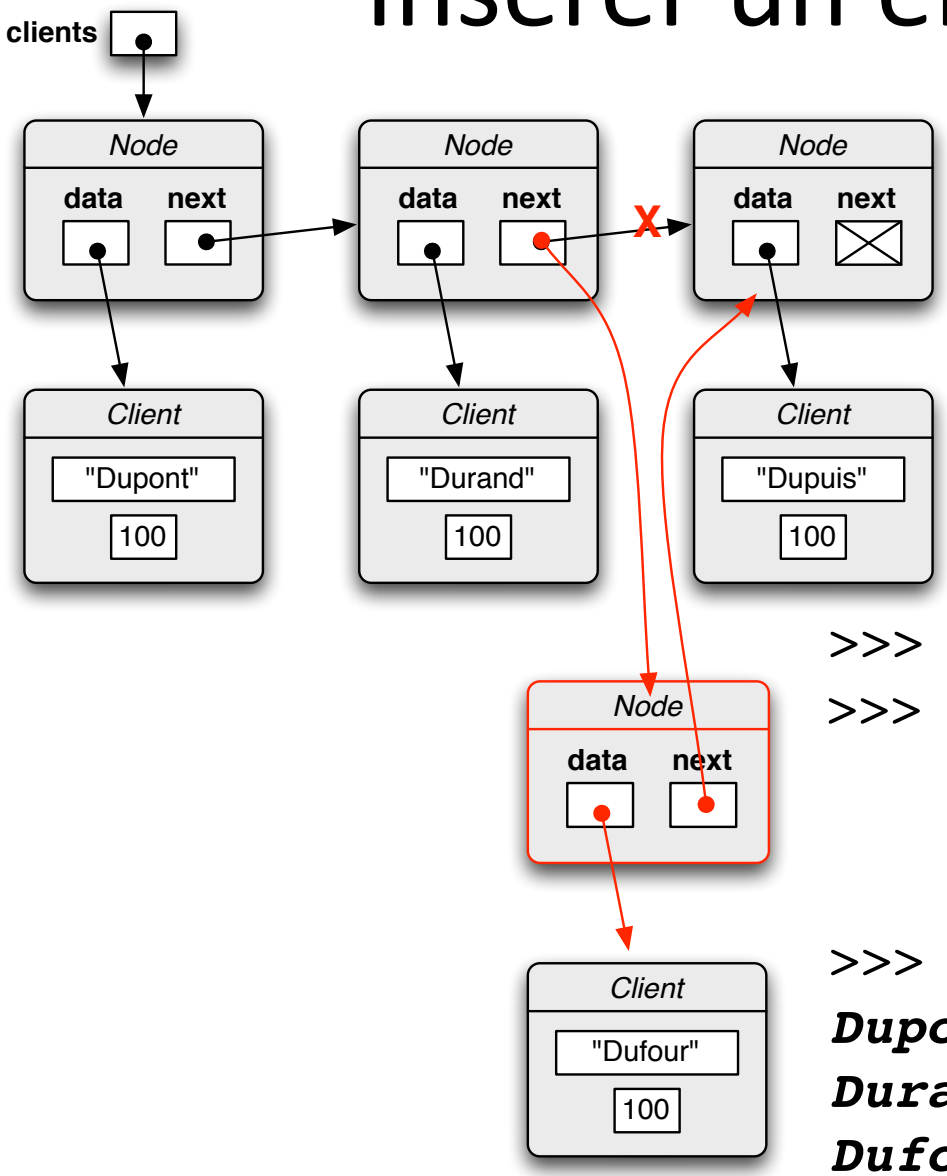

```
>>> n = clients.next
>>> newNode = Node(
    Client("Dufour",100),None)
>>> newNode.next = n.next
>>> n.next = newNode
>>> print(clients)
Dupont: 100
Durand: 100
Dufour: 100
Dupuis: 100
```

Insérer un élément




```
>>> n = clients.next
>>> n.next =
        Node(
            Client("Dufour",100),
            n.next)
>>> print(clients)
Dupont: 100
Durand: 100
Dufour: 100
Dupuis: 100
```

Insérer un élément

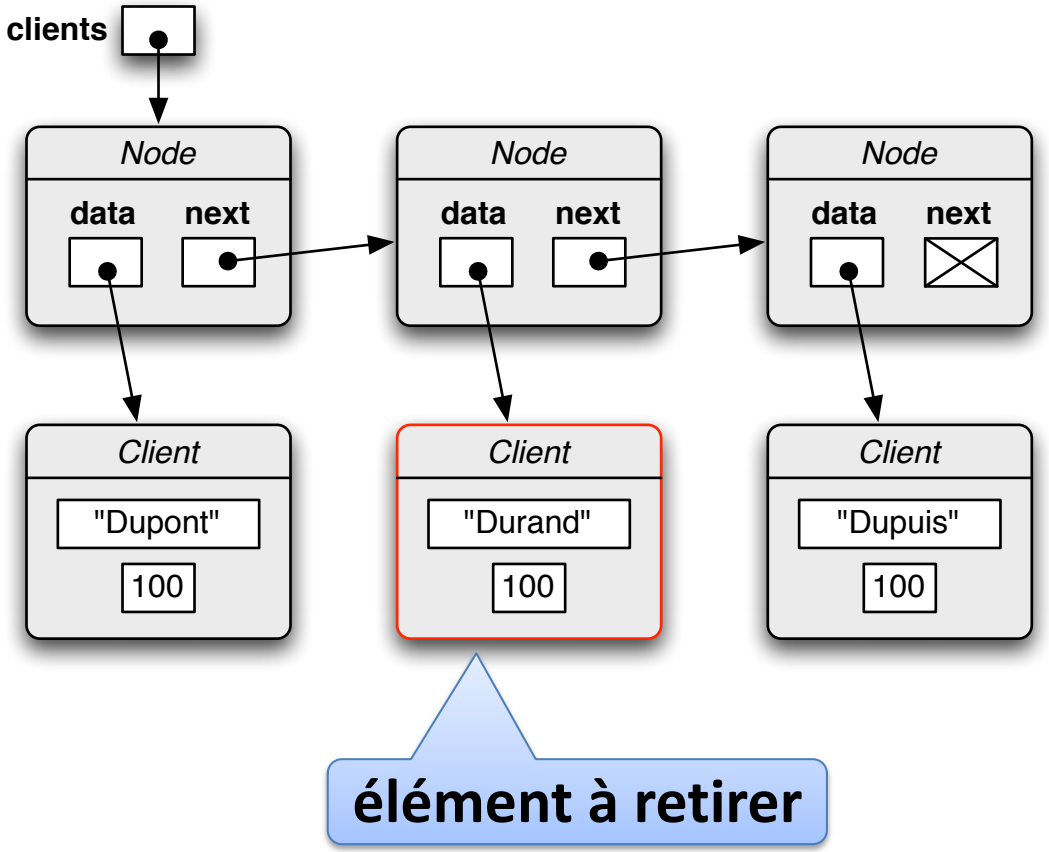


Cas particuliers !

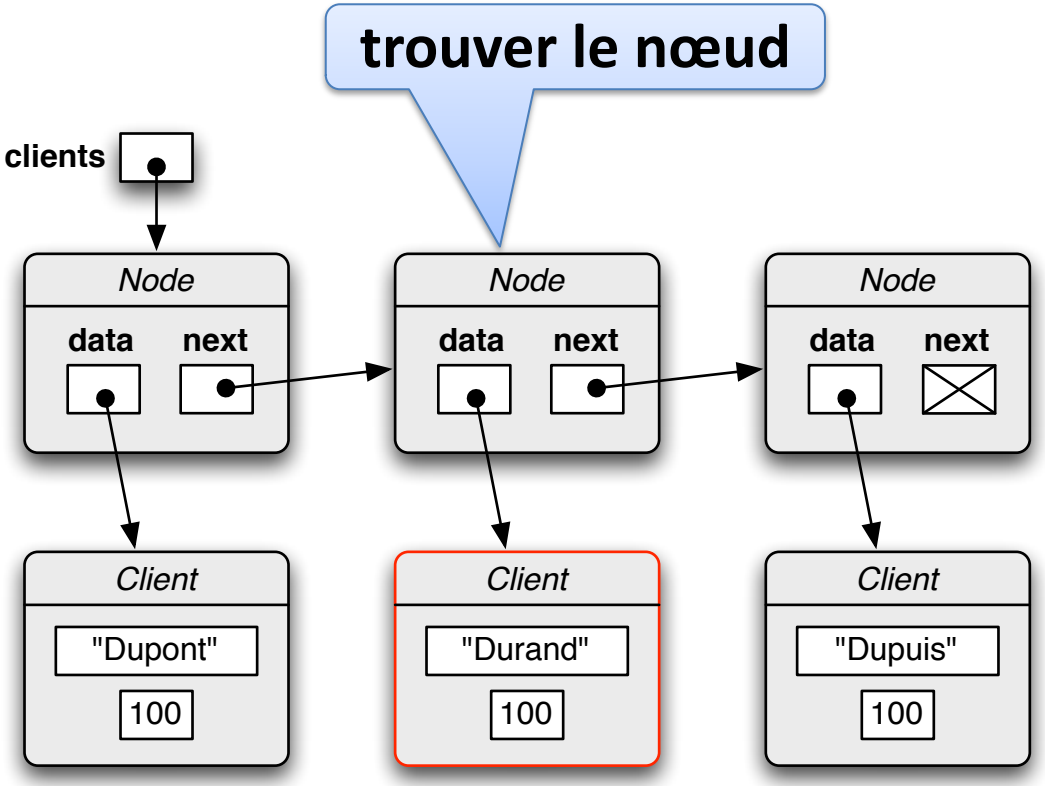
- début de liste
- fin de liste
- liste vide

```
>>> n = clients.next
>>> n.next =
        Node(
           Client("Dufour",100),
           n.next)
>>> print(clients)
Dupont: 100
Durand: 100
Dufour: 100
Dupuis: 100
```

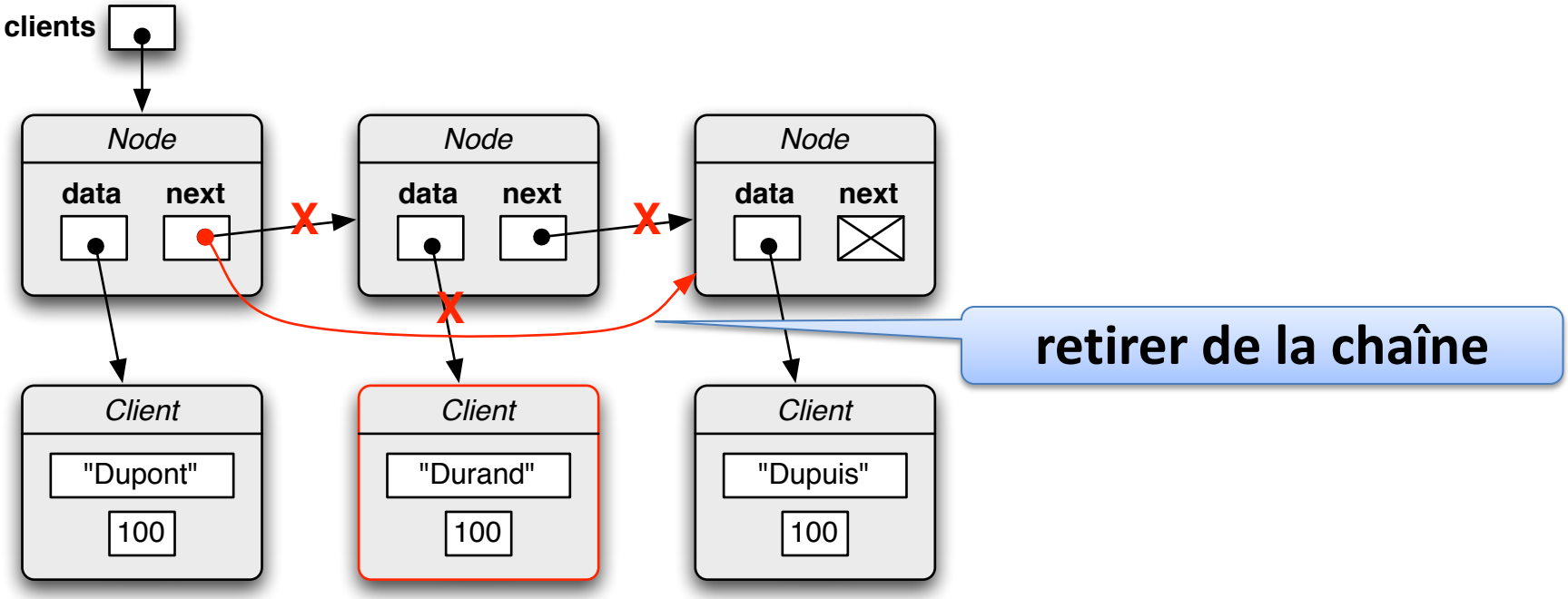
Retirer un élément



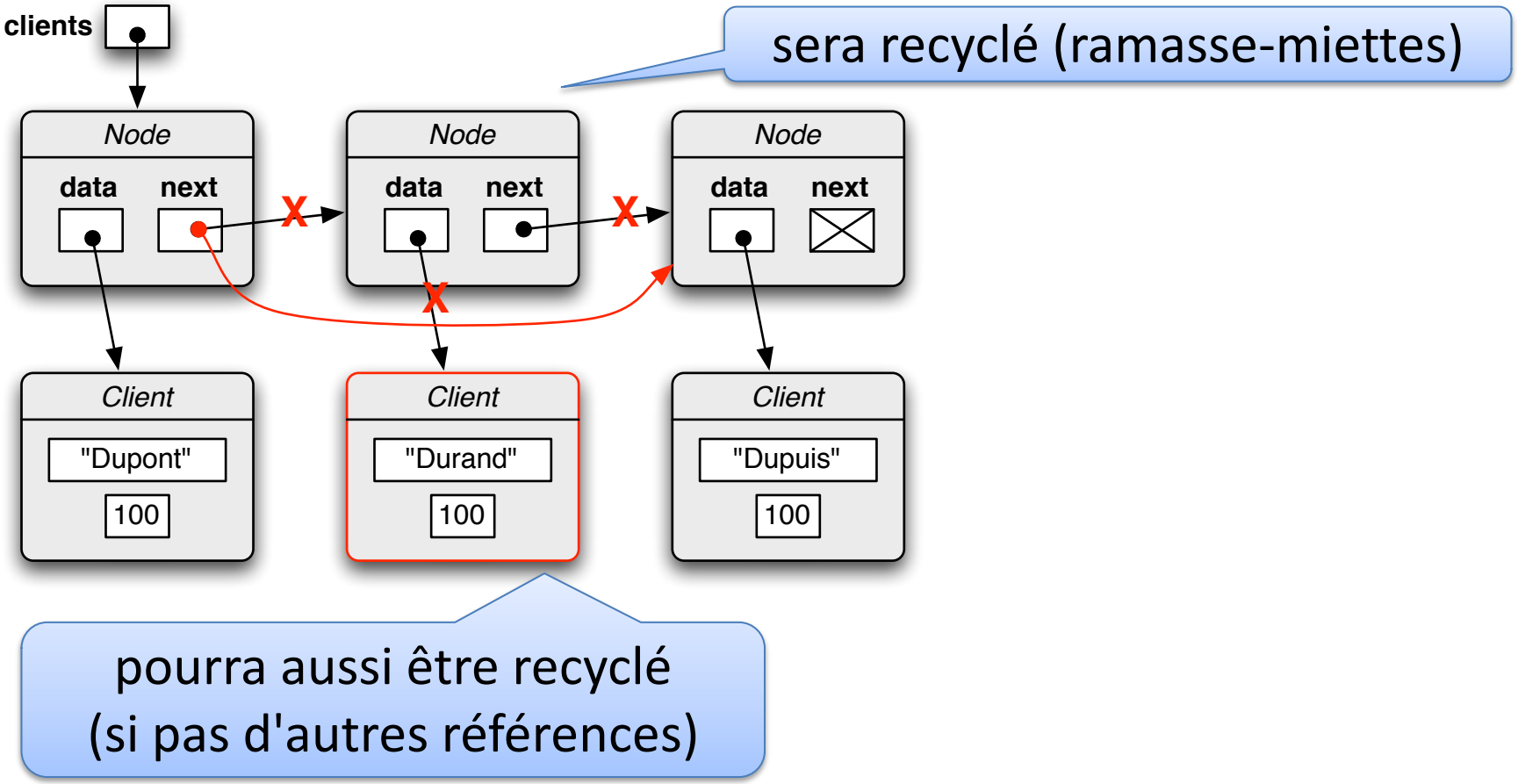
Retirer un élément



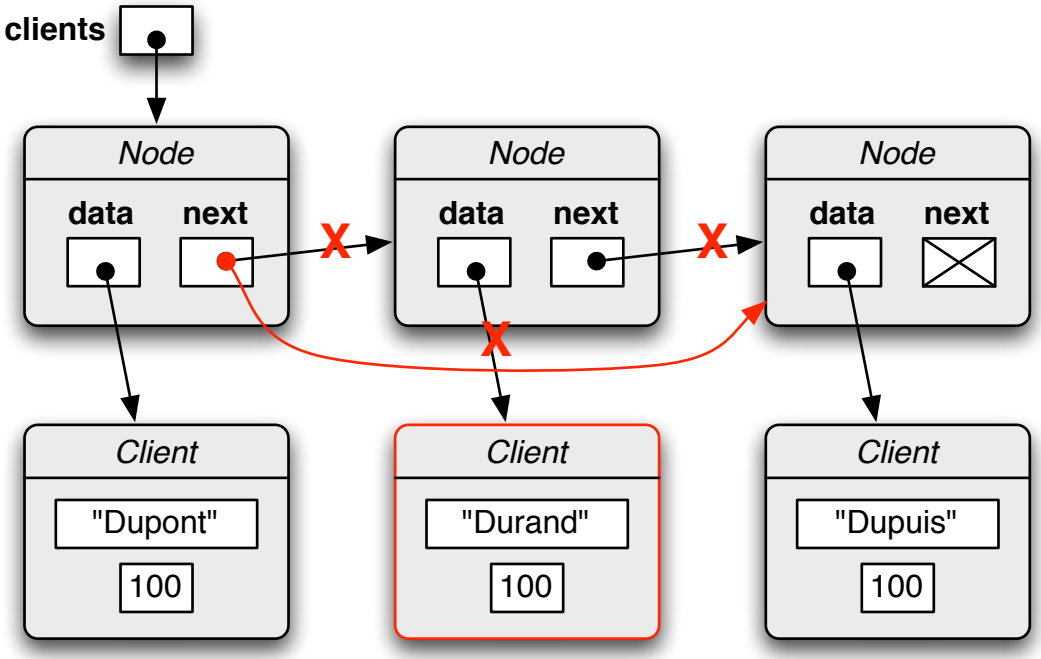
Retirer un élément



Retirer un élément

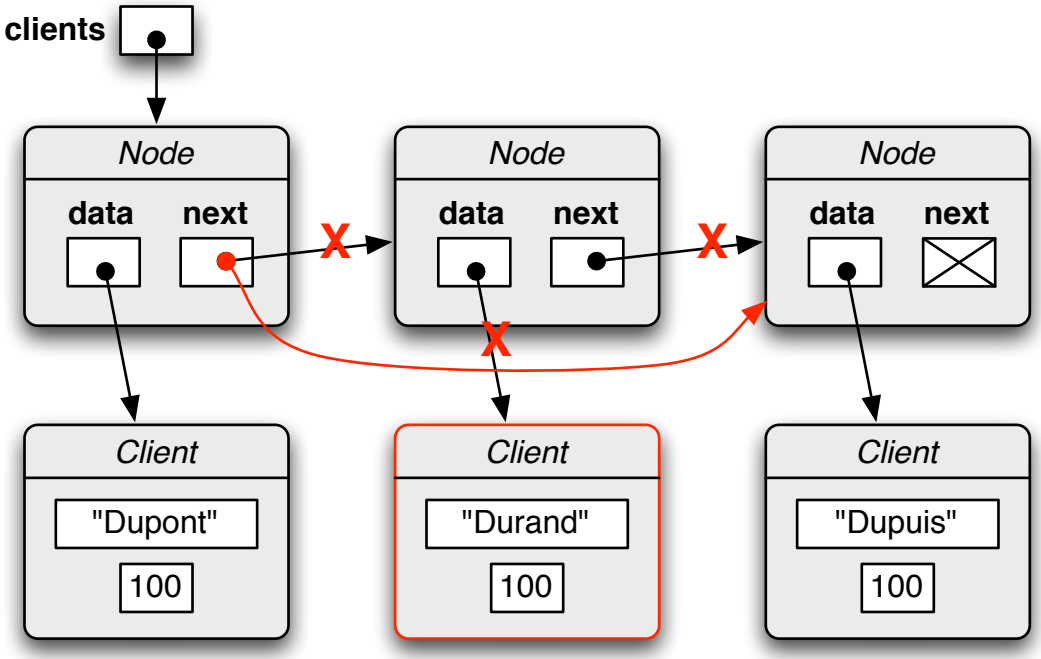


Retirer un élément




```
>>> clients.next =
    clients.next.next
>>> print(clients)
Dupont: 100 Dupuis: 100
```

Retirer un élément



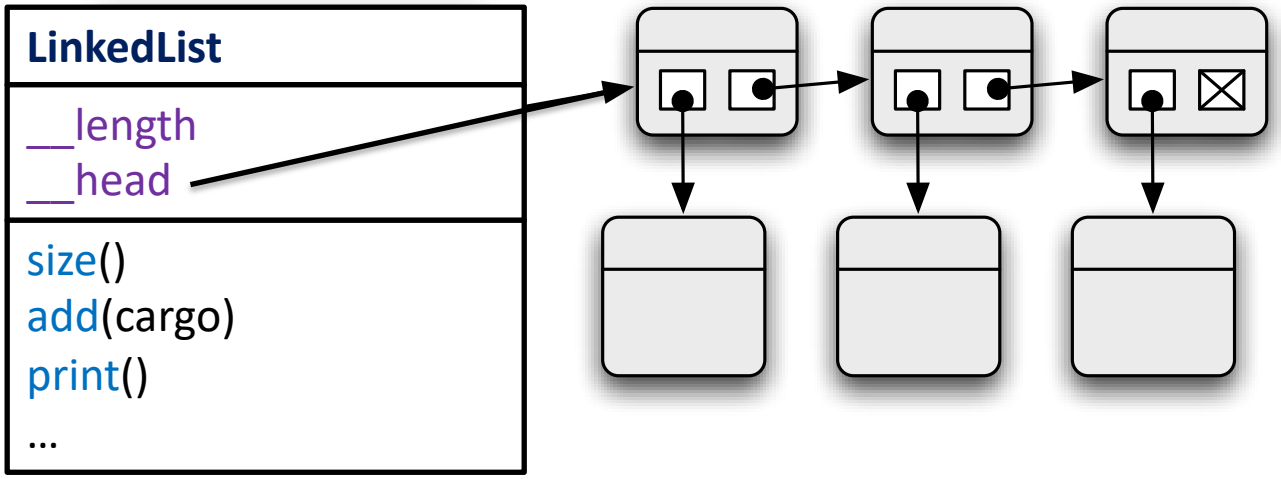
+ cas particuliers !

```
>>> clients.next =
    clients.next.next
>>> print(clients)
Dupont: 100 Dupuis: 100
```

Classe **LinkedList**

Implémentation d'une liste (simplement) chaînée comme classe Python :

- Utilise une **chaîne de nœuds** comme composant, en gardant un référence `__head` vers le premier nœud
- Ainsi qu'un attribut `__length` représentant la taille
- Gère automatiquement le **chaînage des nœuds**
- Offre des méthodes spécifiques supplémentaires :
  - ajout, impression, insertion et retrait d'éléments, …



Classe **LinkedList**

```
class LinkedList :

  def __init__(self):
    self.__length = 0
    self.__head = None

  def size(self):
    return self.__length

  def add(self, cargo):
    node = Node(cargo,self.__head)
    self.__head = node
    self.__length += 1

  …
```

```
clients = LinkedList()

clients.add(Client("Dupuis",100))

clients.add(Client("Durand",100))

clients.add(Client("Dupont ",100))
```

```
class Node:
  def __init__(self,d,nx):
    self.data = d
    self.next = nx

  …
```

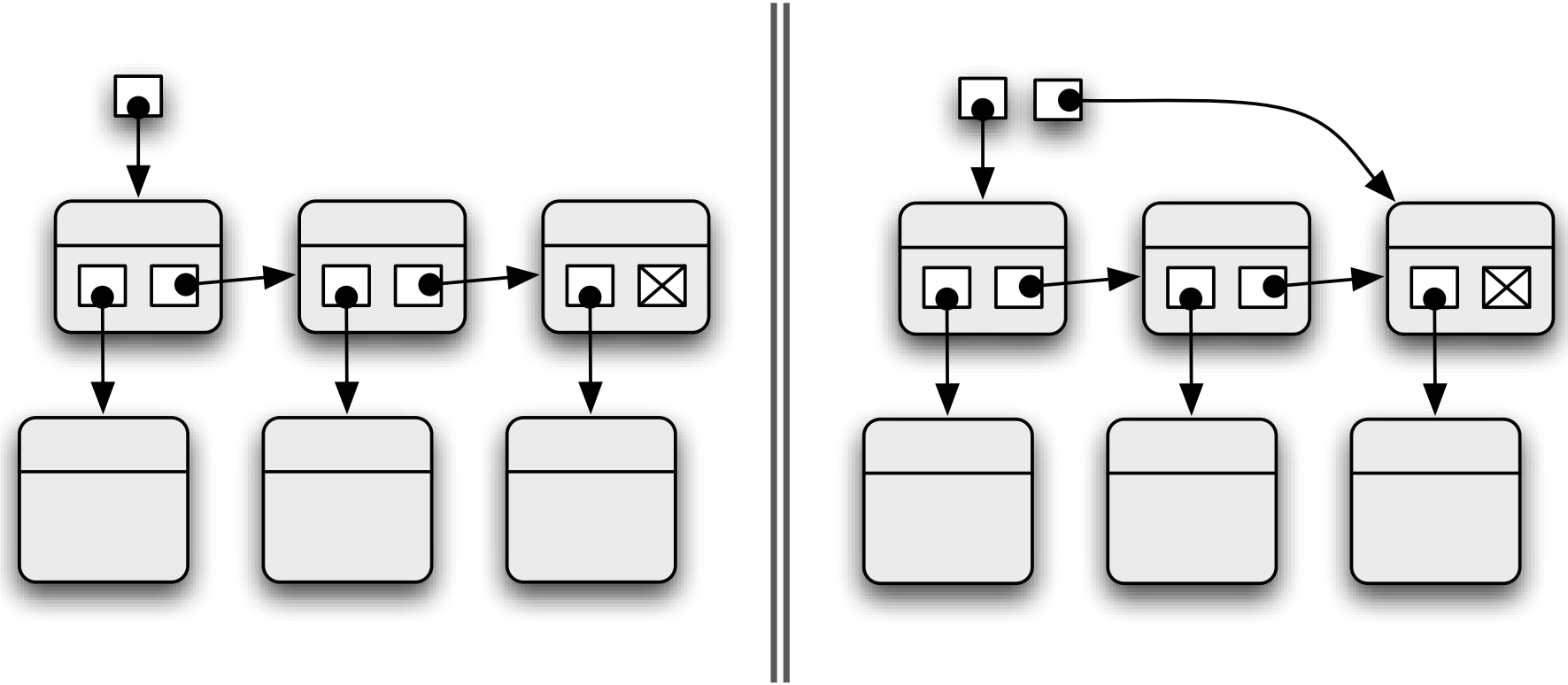
Mission 11

- Objectifs
  - Listes chaînées
- Problème
  - Gérer un classement
  - Vous fournissez une **liste ordonnée**

RANG	COUREUR	DOSSARD	ÉQUIPE	TEMPS	ÉCART
1.	pher	1	SKY PROCYCLING	**51' 33"**	
2.		91	TEAM SAXO-TINKOFF	**51' 42"**	+ 00' 09"
3.	RODRIGUEZ OLIVER Joaquin	101	KATUSHA TEAM	**51' 43"**	+ 00' 10"
4.	KREUZIGER Roman	94	TEAM SAXO-TINKOFF	**51' 56"**	+ 00' 23"
5.	VALVERDE Alejandro	121	MOVISTAR TEAM	**52' 03"**	+ 00' 30"
6.	QUINTANA ROJAS Nairo Alexander	128	MOVISTAR TEAM	**52' 44"**	+ 01' 11"
7.	al	153	OMEGA PHARMA-QUICK STEP	**53' 06"**	+ 01' 33"
8.		63	ASTANA PRO TEAM	**53' 07"**	+ 01' 34"
9.			GARMIN - SHARP	**53' 14"**	+ 01' 41"
10.				**53' 24"**	+ 01' 51"
11.	MOLLEMA Bauke	164	BELKIN PRO CYCLING	**53' 42"**	+ 02' 09"
12.	MONFORT Maxime	47	RADIOSHACK LEOPARD	**53' 50"**	+ 02' 17"
13.	ROGERS Michael	98	TEAM SAXO-TINKOFF	**53' 58"**	+ 02' 25"

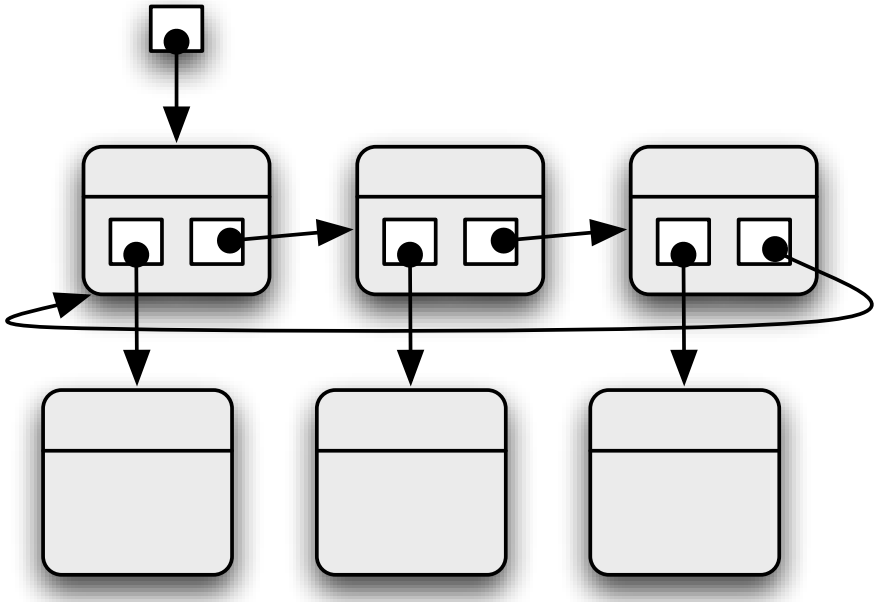
Variantes de listes chaînées

liste simplement chaînée

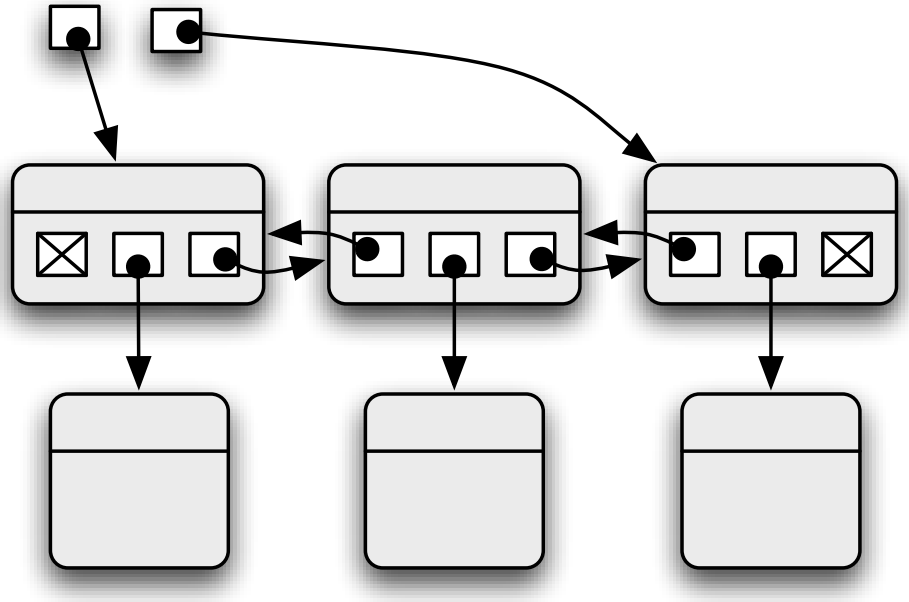


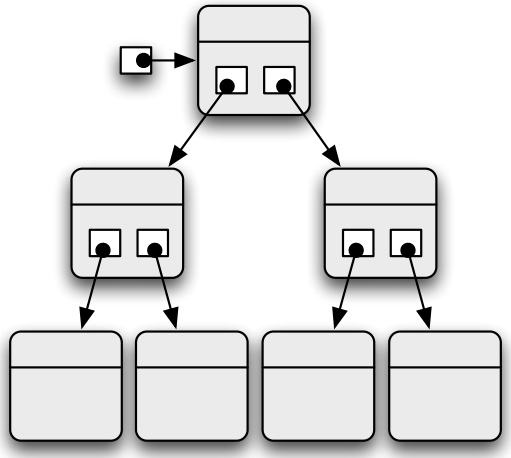
Variantes de listes chaînées

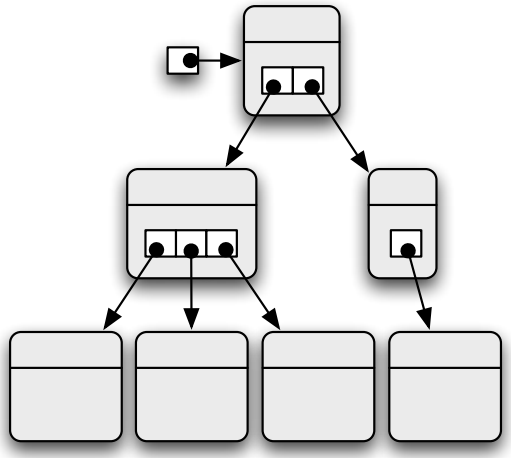
liste circulaire

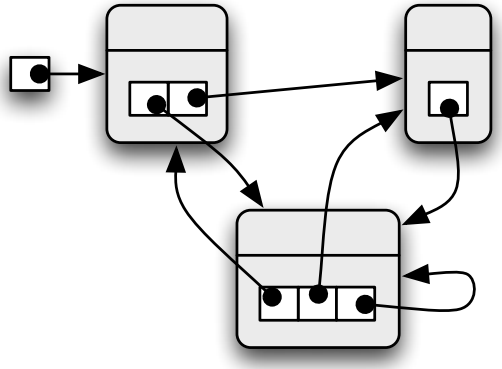


liste doublement chaînée









arbre binaire

arbre quelconque

graphe

Autres structures chaînées

Classe **LinkedList**

```
class LinkedList :

  def __init__(self):
    self.__length = 0
    self.__head = None

  def size(self):
    return self.__length

  def add(self, cargo):
    node = Node(cargo,self.__head)
    self.__head = node
    self.__length += 1

  ...
```

```
class Node:
  def __init__(self,d,nx):
    self.data = d
    self.next = nx

 ...
```

Détail d'implémentation de LinkedList?

➔ à cacher dans LinkedList ?

Classe interne **Node**

```
class LinkedList :
    class Node:
        def __init__(self,d,nx):
            self.__data = d
            self.__next = nx
        ...
    def __init__(self):
        self.__length = 0
        self.__head = None
    def size(self):
        return self.__length
    def add(self, cargo):
        node = self.Node(cargo,self.__head)
        self.__head = node
        self.__length += 1
    ...
```

Visible uniquement dans **LinkedList**

Python supporte la notion de **classes internes** :

- une classe définie à l'intérieur d'une autre classe
- permet de cacher une classe dans une autre
- utile si la classe interne est une classe auxiliaire
- et n'est pas vraiment utile en elle-même