Partie III

La programmation orientée objets



Informatique 1
Introduction à la programmation

Mission 11 : RESTRUCTURATION

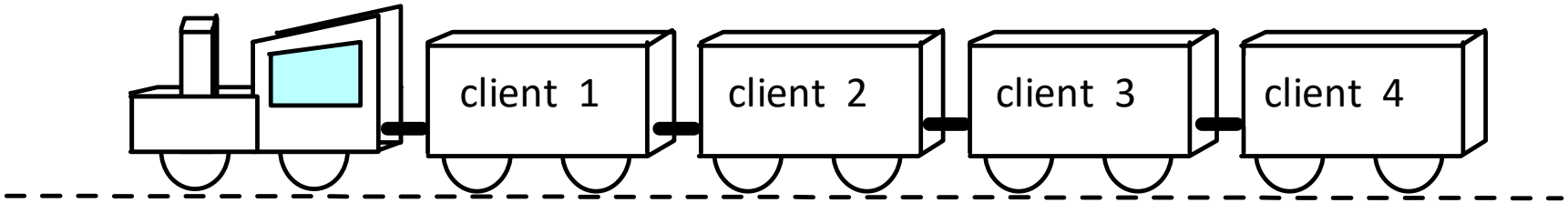
Les listes chaînées

Kim Mens – Cristel Pelsser – Charles Pecheur

Liste chaînée (rappel)



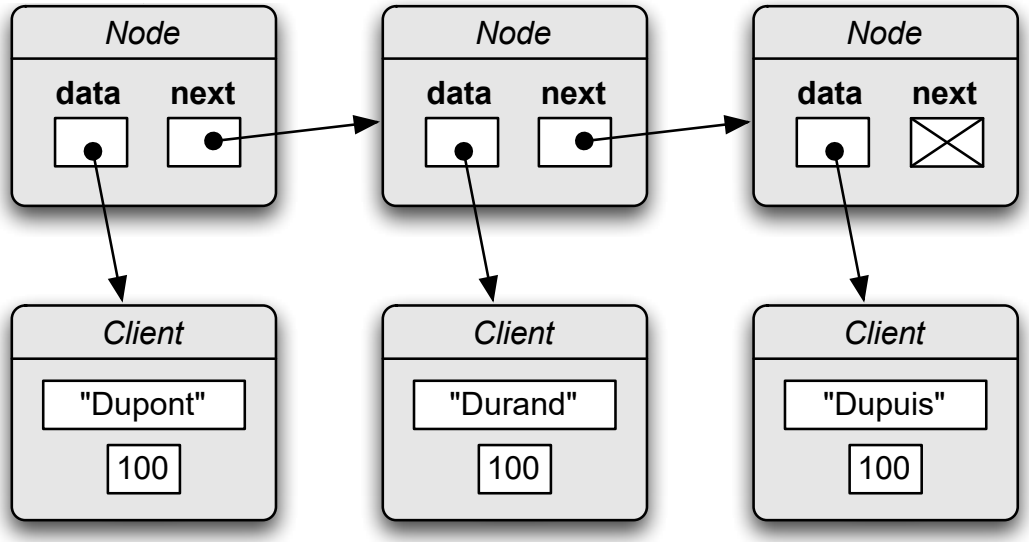
Une **liste chaînée** est une structure de données, groupant une collection d'éléments en une séquence de nœuds chainées qui peut être parcouru séquentiellement.



Liste chaînée (rappel)



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

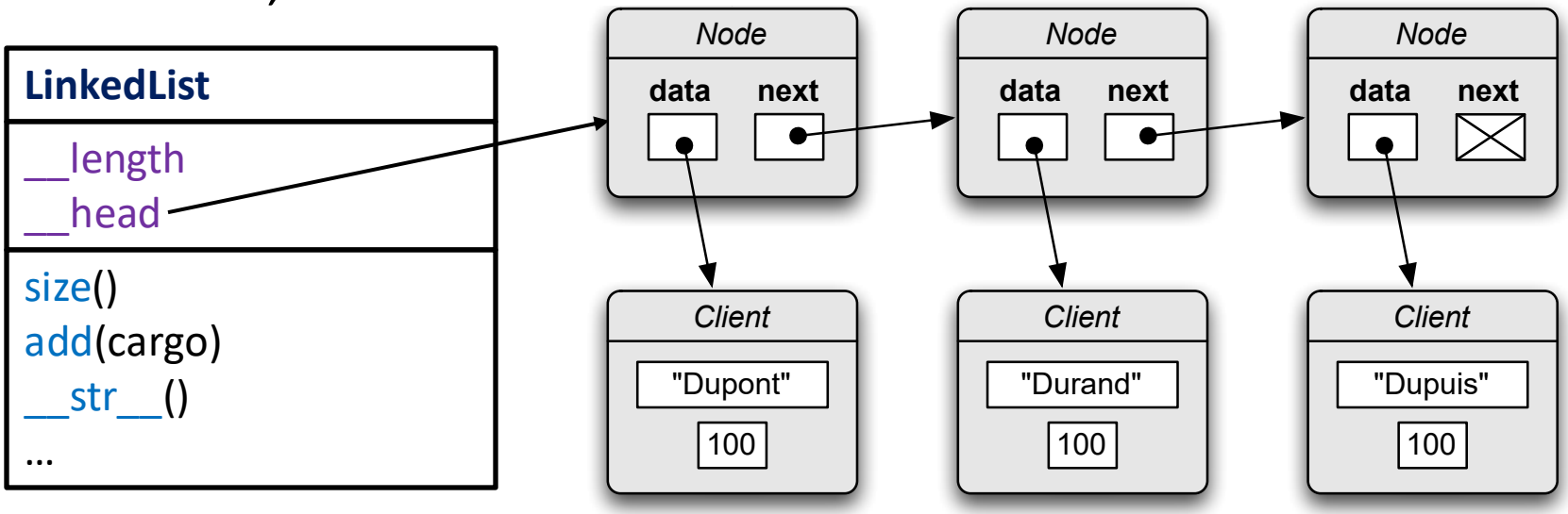


Liste chaînée (rappel)



Implémentation d'une liste chaînée comme classe Python :

- Un attribut `__head` pointant vers le premier nœud
- Ainsi qu'un attribut `__length` représentant la taille
- Et des méthodes pour : taille, ajout, impression, suppression, insertion, …



Classe **LinkedList** et sa classe **Node**

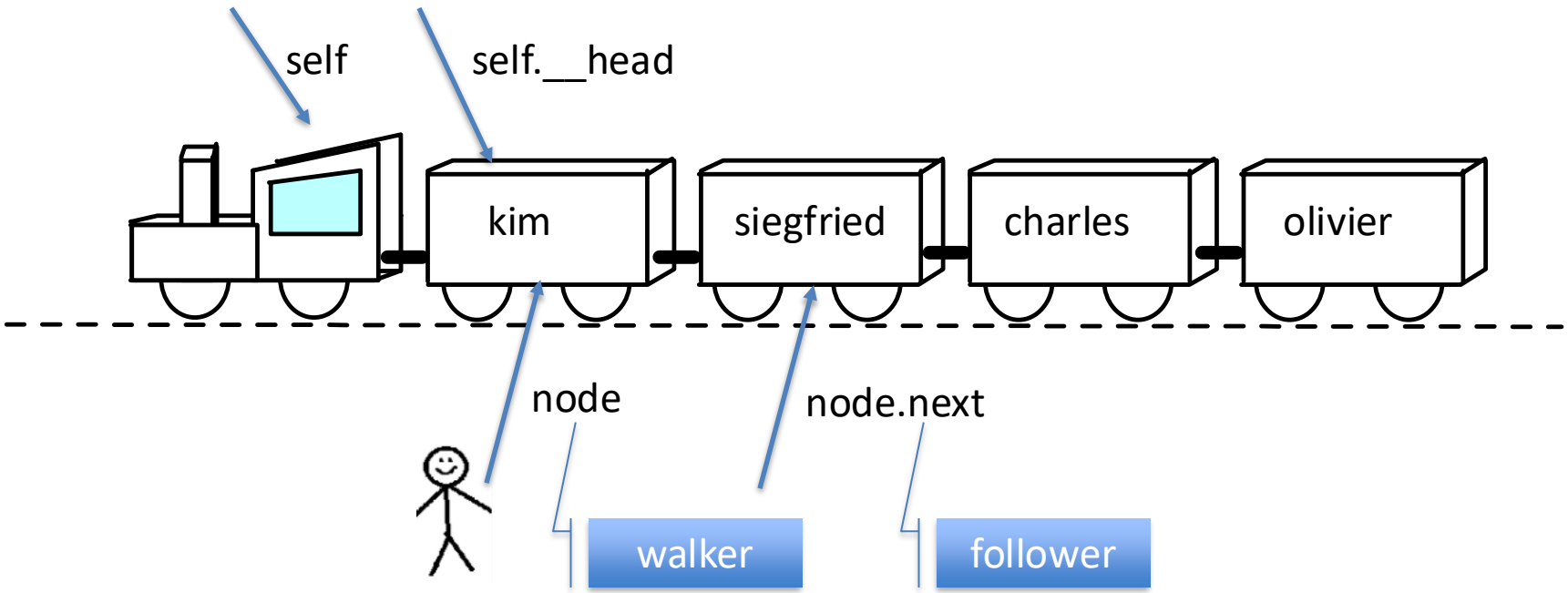
```
class Node:
    def __init__(self,d,n) :
        self.data = d
        self.next  = n
```

```
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
    def __str__(self):
        s = "[ "
        node = self.__head
        while node is not None:
            s += str(node.data) + " "
            node = node.next
        return s + "]"
```

```
l = LinkedList()
print(l)
# [ ]
print(l.size())
# 0
l = LinkedList()
l.add(3)
l.add(2)
l.add(1)
print(l)
# [ 1 2 3 ]
print(l.size())
# 3
```

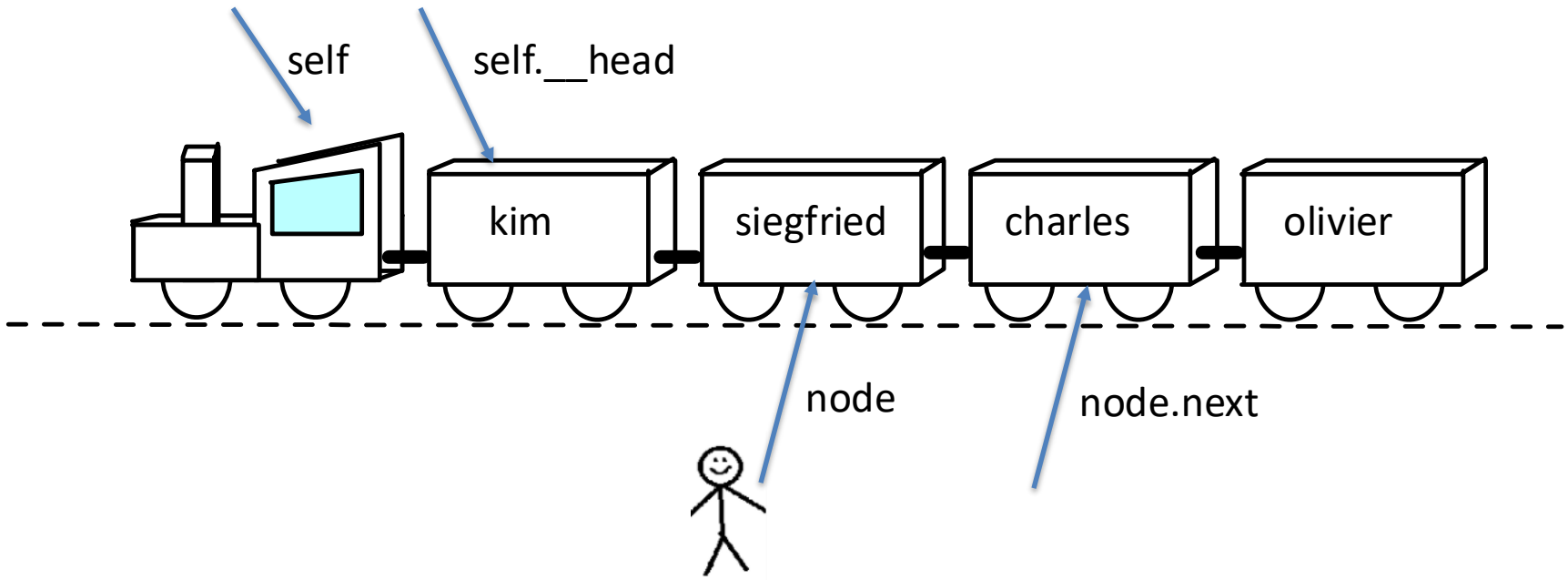
```
class LinkedList :
    ...
    def __str__(self):
        s = "[ "
        node = self.__head
        while node is not None:
            s += str(node.data) + " "
            node = node.next
        return s + "]"
```

```
l = LinkedList()
l.add('olivier')
l.add('charles')
l.add('siegfried')
l.add('kim')
print(l)
# [ kim siefgried charles olivier ]
print(l.size())
# 3
```



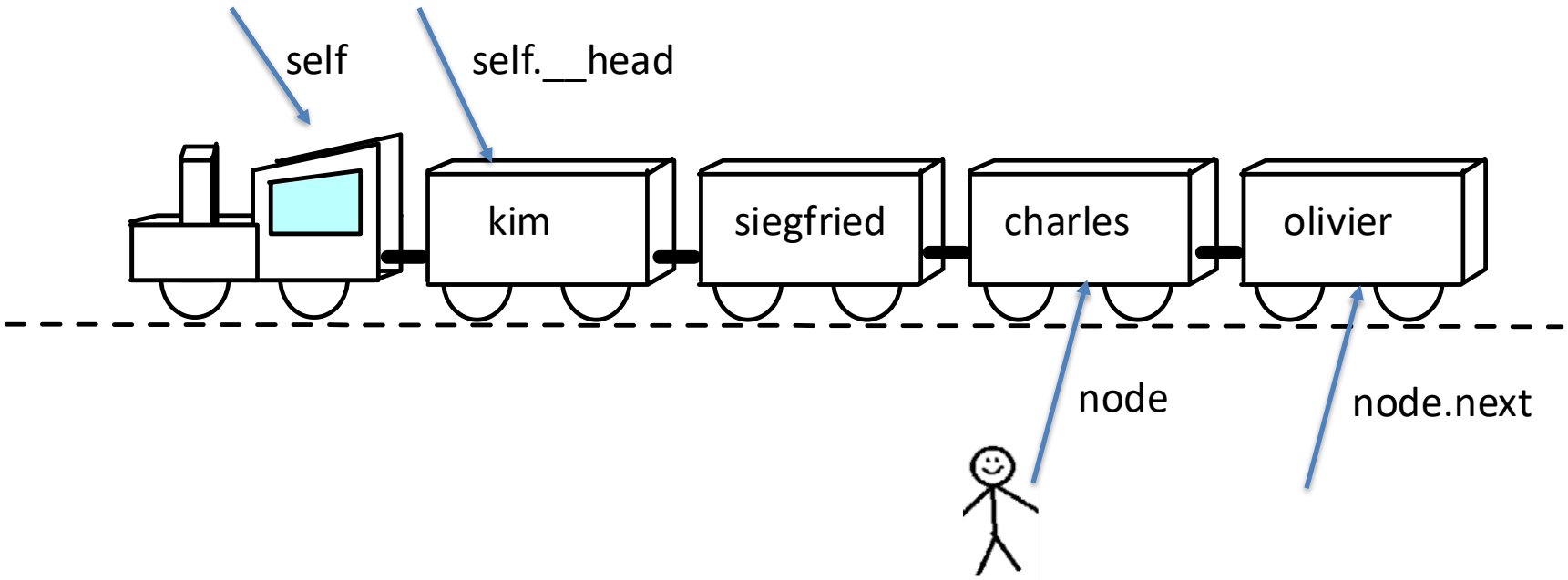
```
class LinkedList :
    ...
    def __str__(self):
        s = "[ "
        node = self.__head
        while node is not None:
            s += str(node.data) + " "
            node = node.next
        return s + "]"
```

```
l = LinkedList()
l.add('olivier')
l.add('charles')
l.add('siegfried')
l.add('kim')
print(l)
# [ kim siefgried charles olivier ]
print(l.size())
# 3
```



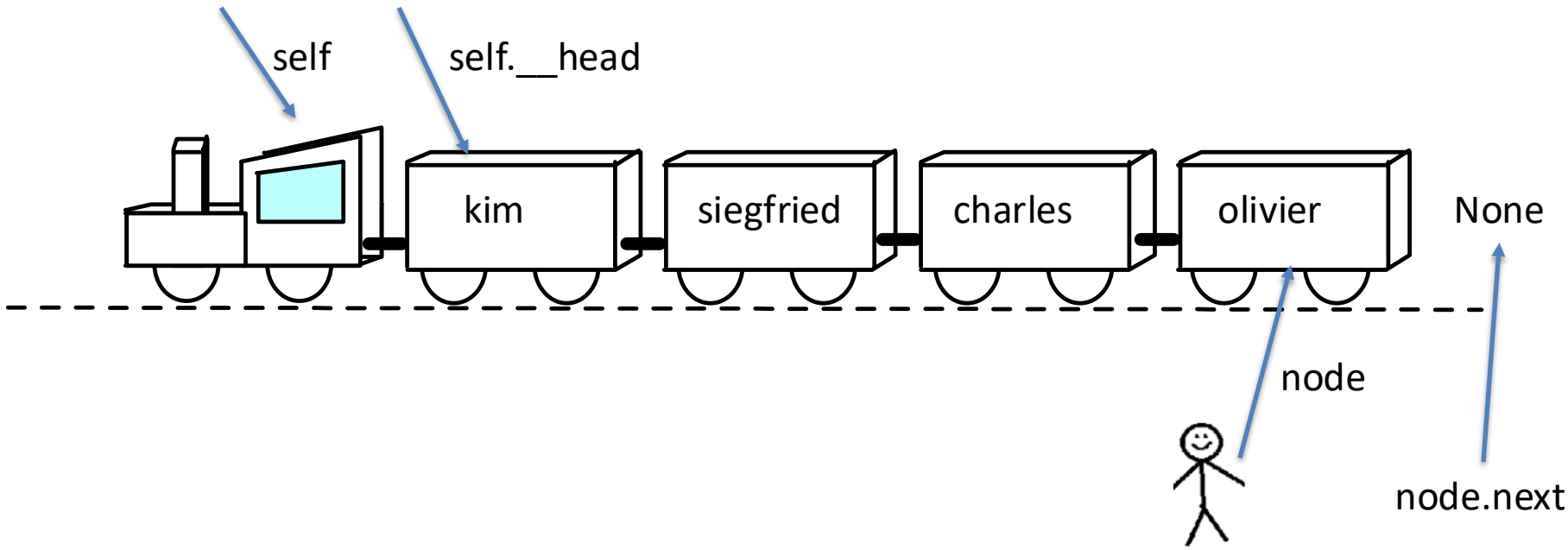
```
class LinkedList :
    ...
    def __str__(self):
        s = "[ "
        node = self.__head
        while node is not None:
            s += str(node.data) + " "
            node = node.next
        return s + "]"
```

```
l = LinkedList()
l.add('olivier')
l.add('charles')
l.add('siegfried')
l.add('kim')
print(l)
# [ kim siefgried charles olivier ]
print(l.size())
# 3
```



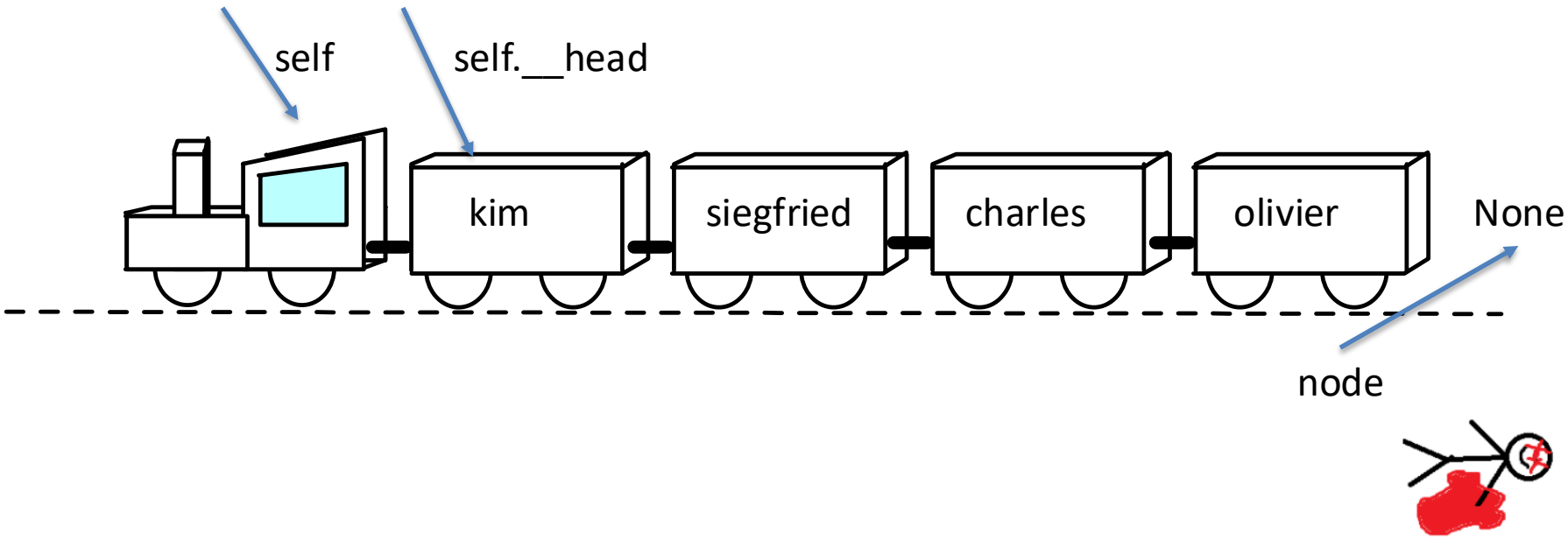
```
class LinkedList :
    ...
    def __str__(self):
        s = "[ "
        node = self.__head
        while node is not None:
            s += str(node.data) + " "
            node = node.next
        return s + "]"
```

```
l = LinkedList()
l.add('olivier')
l.add('charles')
l.add('siegfried')
l.add('kim')
print(l)
# [ kim siefgried charles olivier ]
print(l.size())
# 3
```

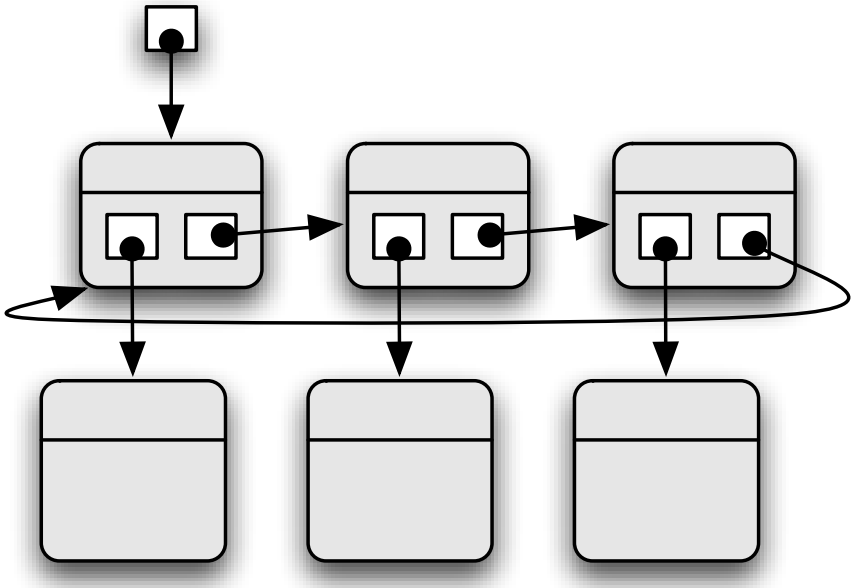


```
class LinkedList :
    ...
    def __str__(self):
        s = "[ "
        node = self.__head
        while node is not None:
            s += str(node.data) + " "
            node = node.next
        return s + "]"
```

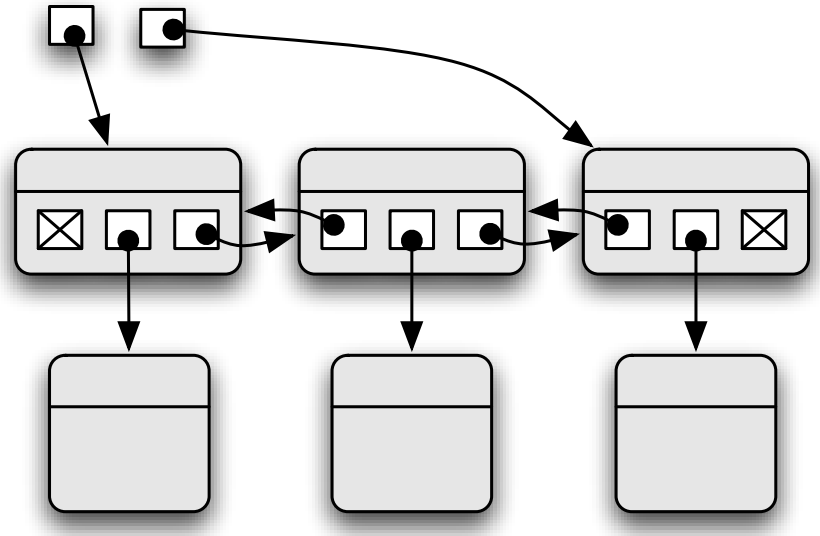
```
l = LinkedList()
l.add('olivier')
l.add('charles')
l.add('siegfried')
l.add('kim')
print(l)
# [ kim siefgried charles olivier ]
print(l.size())
# 3
```



liste circulaire

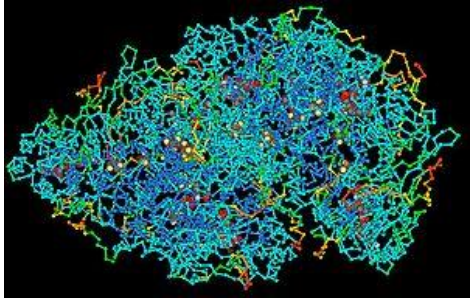


liste doublement chaînée



Variantes de listes chaînées

Problème



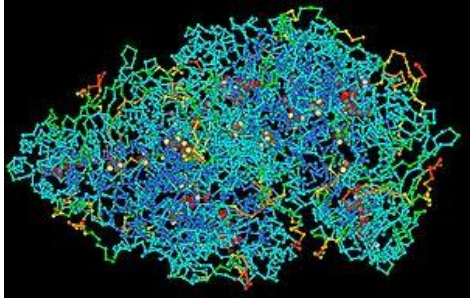
(Source: https://adventofcode.com/2018/day/5)

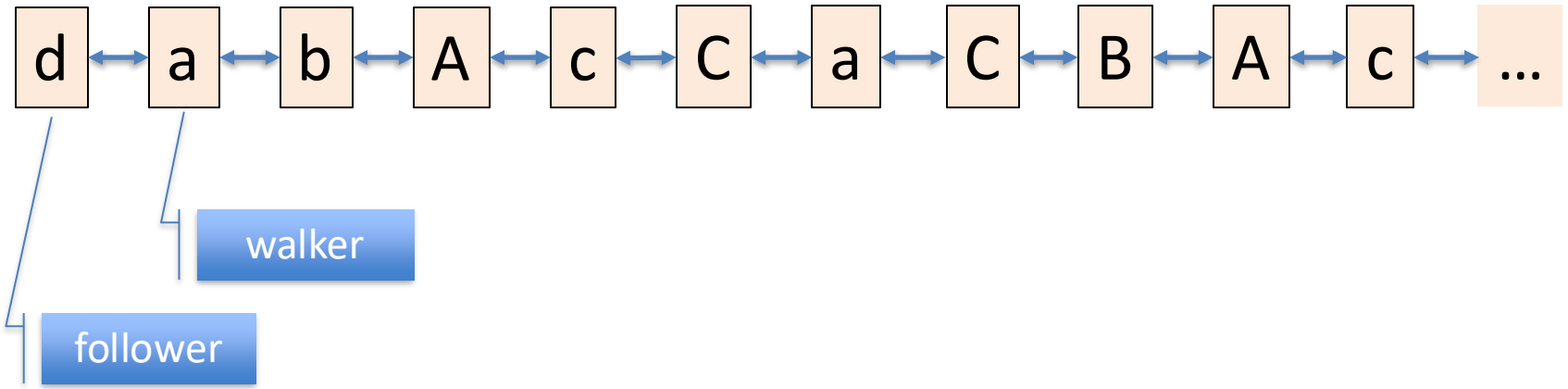
Étant donné une chaîne de caractères, supprimer tous les caractères consécutifs qui sont identiques (indépendamment de leur casse), jusqu'à ce qu'aucun caractère ne puisse être supprimé.

Par exemple, étant donné la chaîne
"dabAcCaCBAcCcaDA"

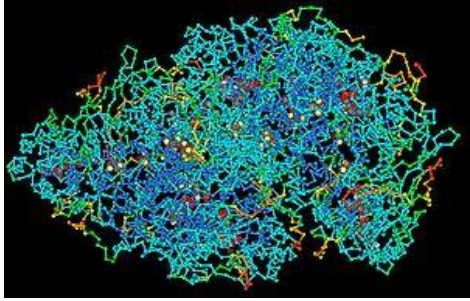
dabA**cC**aCBAcCcaDA	supprimer le premier "cC"
dab**Aa**CBAcCcaDA	ensuite supprimer "Aa"
dabCBA**cC**caDA	supprimer le deuxième "cC"
dabCBAcaDA	plus rien à supprimer

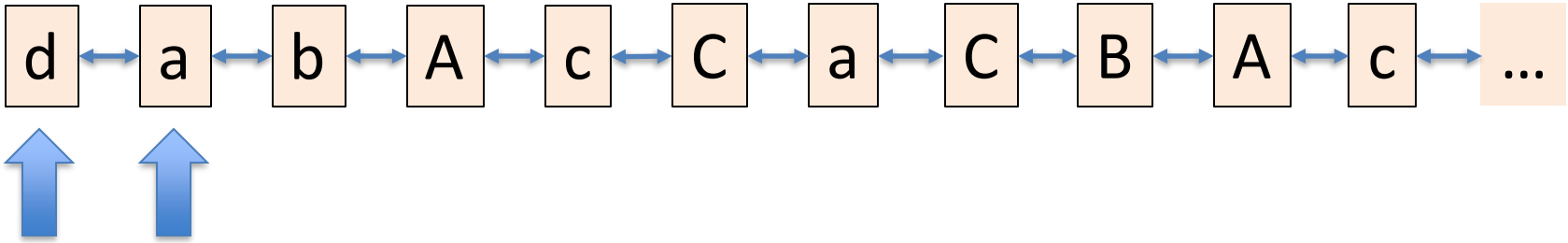
Une solution efficace



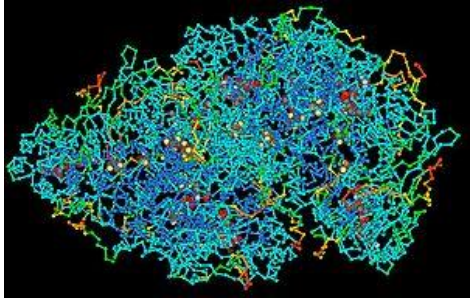


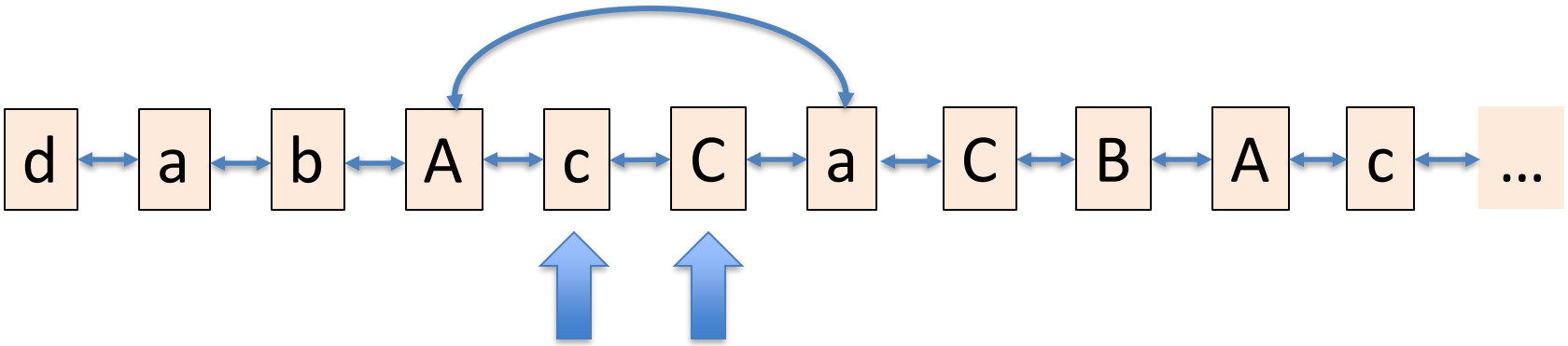
Une solution efficace



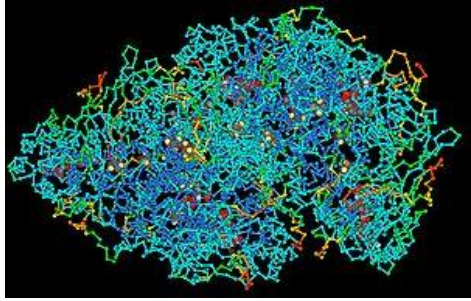


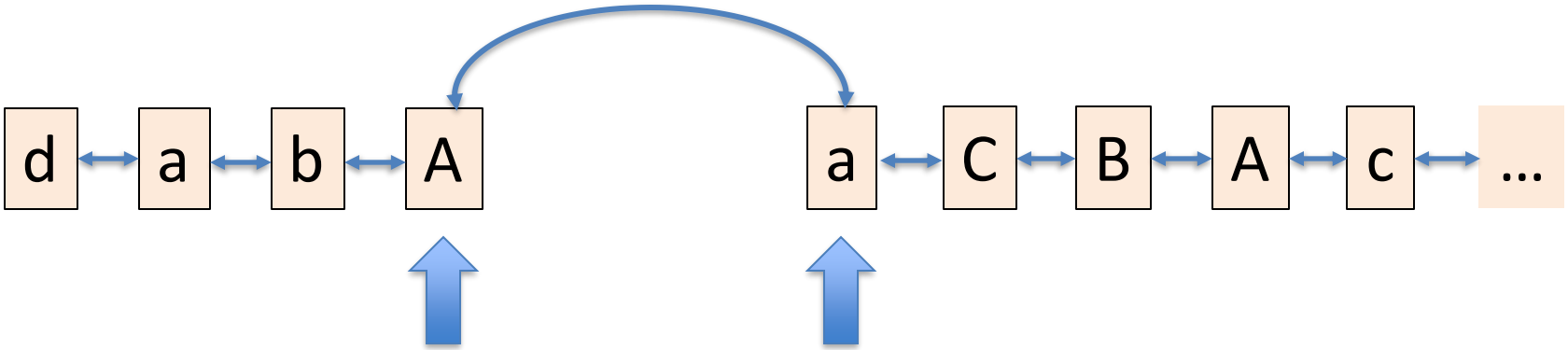
Une solution efficace



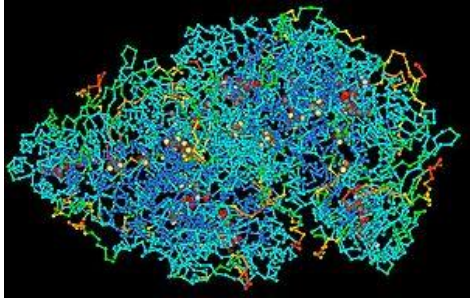


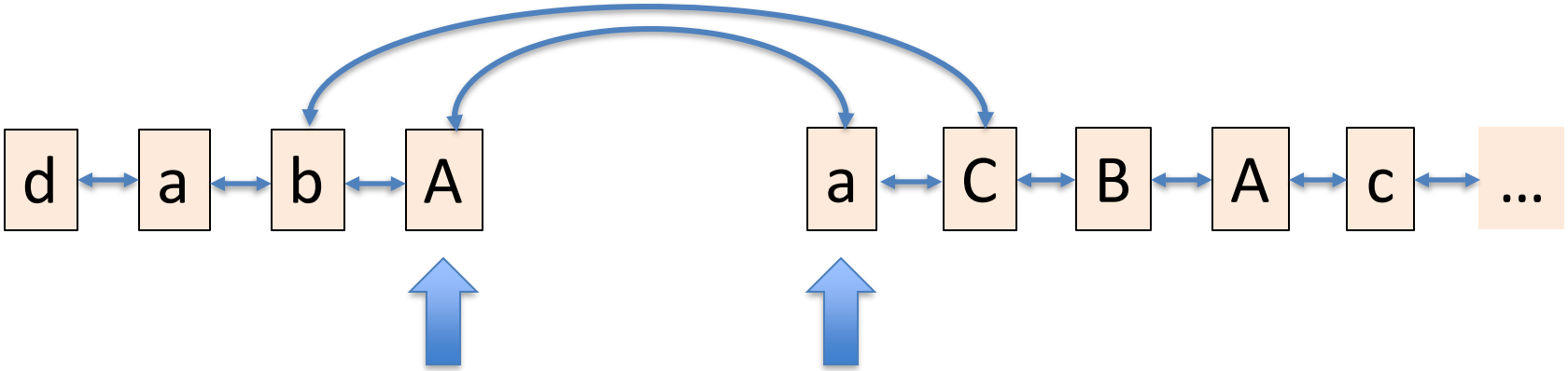
Une solution efficace



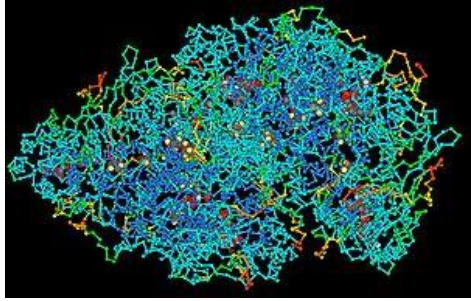


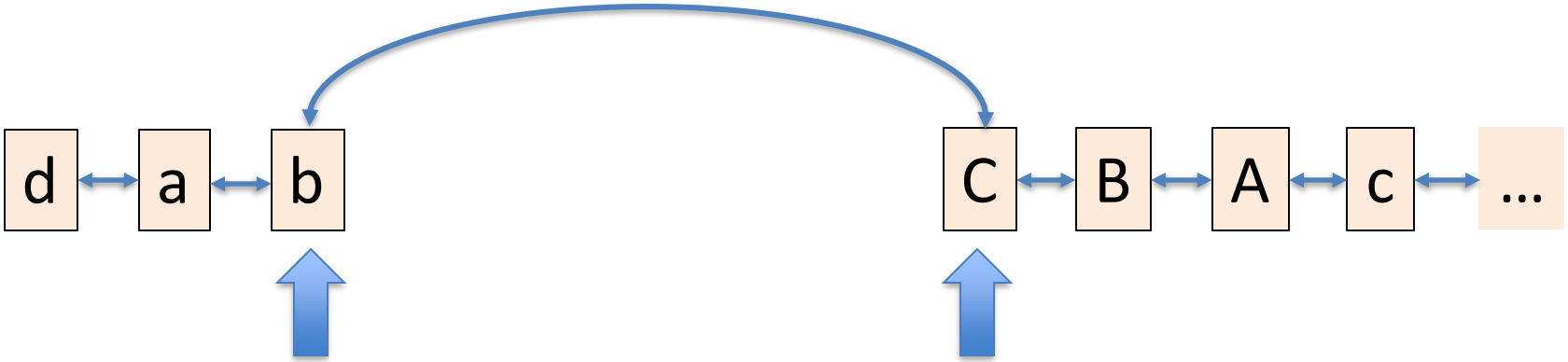
Une solution efficace



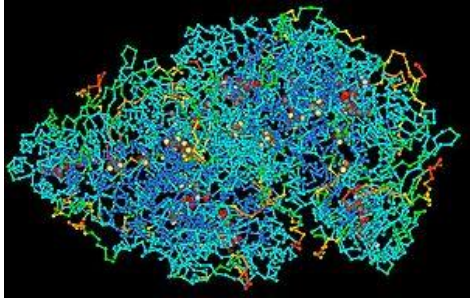


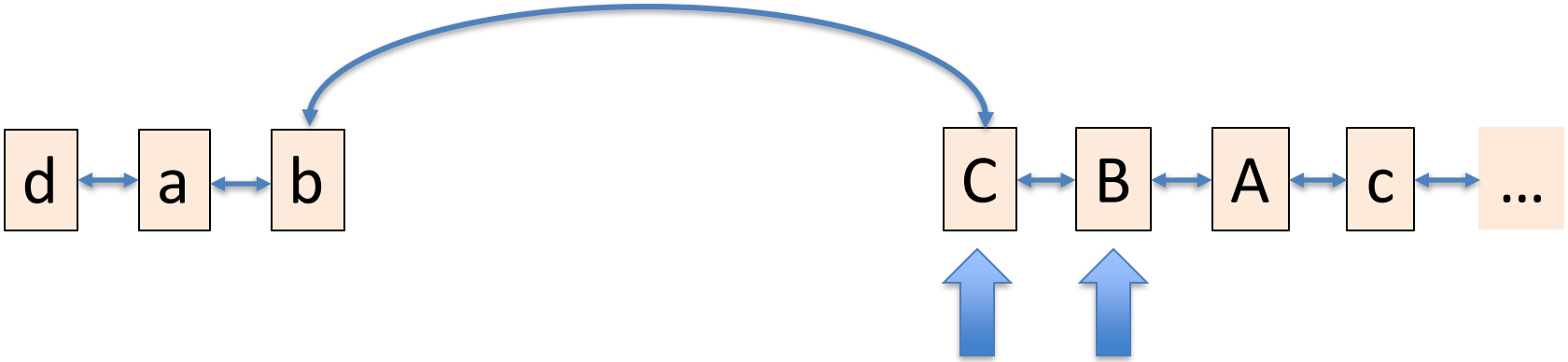
Une solution efficace





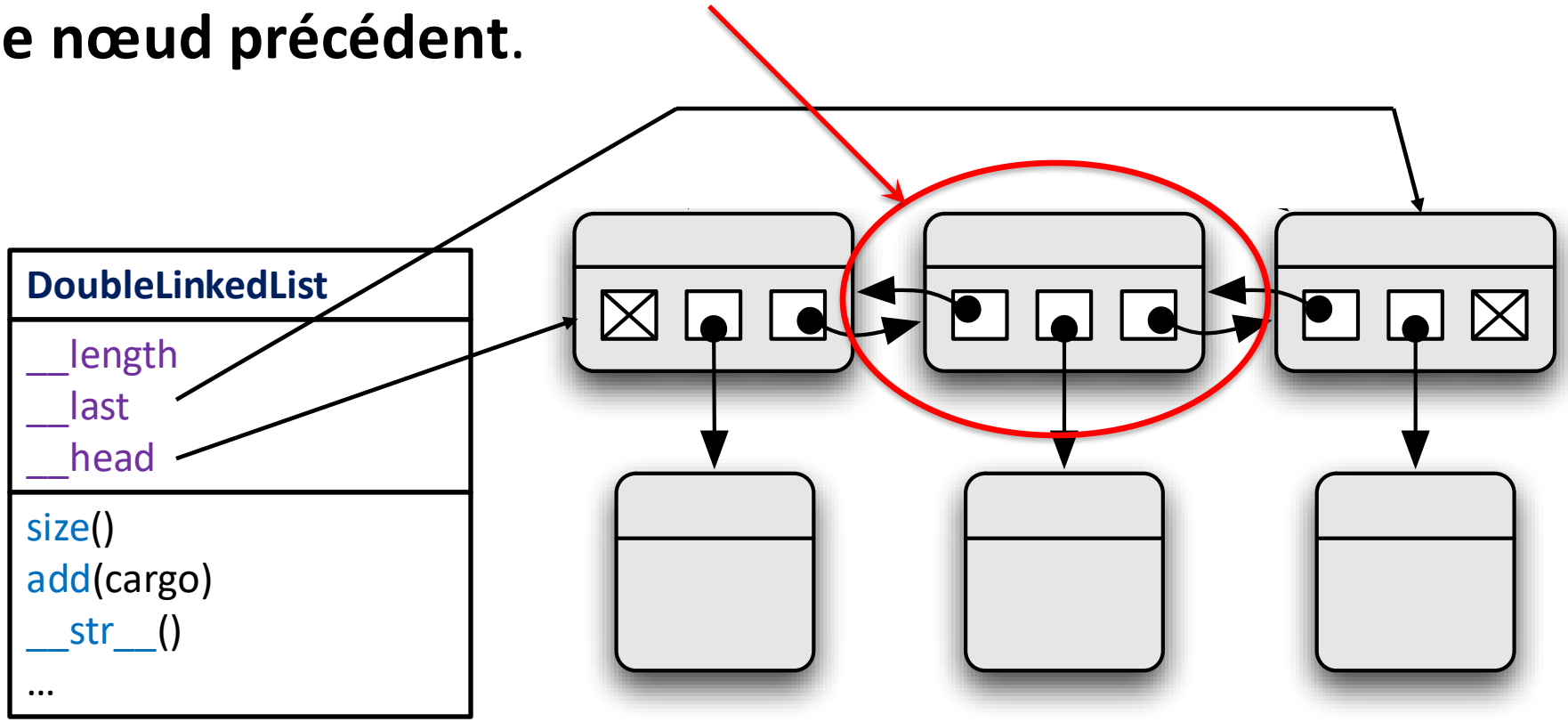
Une solution efficace





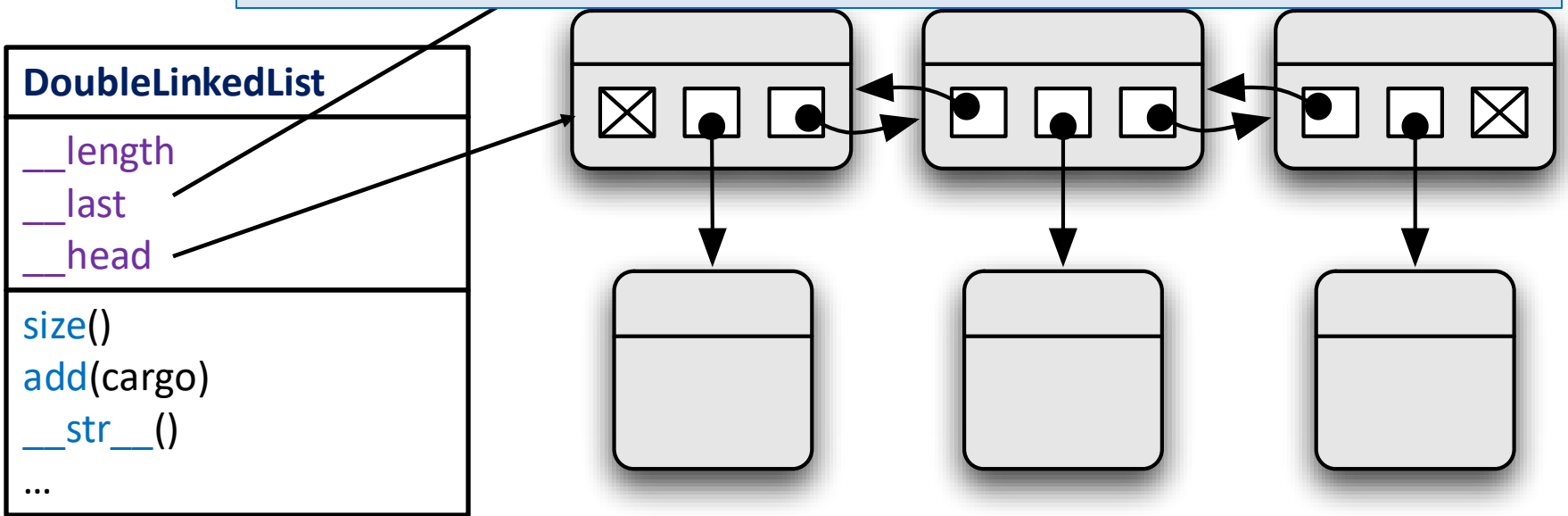
Liste doublement chaînée

Comme une liste (simplement) chaînée, mais chaque nœud garde une référence vers le nœud suivant **et vers le nœud précédent**.



Implémentation

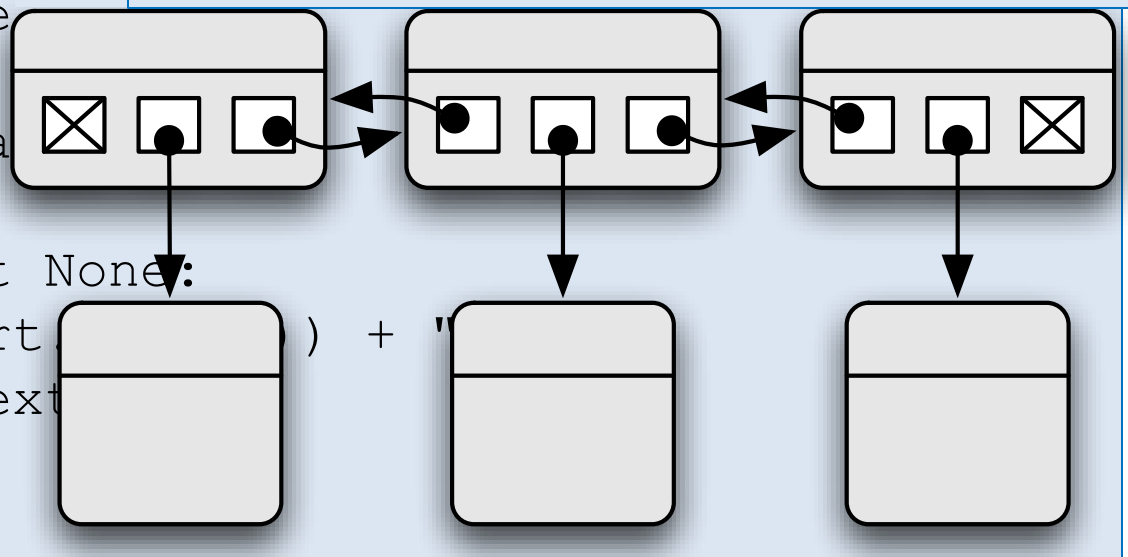
```
class DoubleNode:
  def __init__(self,cargo,nxt=None,prv=None):
    self.__cargo = cargo
    self.__next  = nxt
    self.__prev  = prv
  def __str__(self):
    return str(self.value())
  def value(self):
    return self.__cargo
```



Implémentation

```
class DoubleNode:
  def __init__(self,cargo,nxt=No
    self.__cargo = cargo
    self.__next  = nxt
    self.__prev  = prv
  def __str__(self):
    return str(self.value())
  def value(self):
    return self.__cargo
```

```
class DoubleLinkedList:
  def __init__(self):
    self.__length = 0
    self.__head = None
    self.__last = None
  def __str__(self):
    start = self.__hea
    s = ""
    while start is not None:
      s = s + str(start.          ) + "
      start = start.next
    s = s[:-1]
    return s
```



Implémentation

```
class DoubleNode:
  def __init__(self,cargo,nxt=No
    self.__cargo = cargo
    self.__next  = nxt
    self.__prev  = prv
  def __str__(self):
    return str(self.value())
  def value(self):
    return self.__cargo
```

```
class DoubleLinkedList:
  def __init__(self):
    self.__length = 0
    self.__head = None
    self.__last = None
  def __str__(self):
    start = self.__head
    s = ""
    while start is not None:
      s = s + str(start.value()) + " "
      start = start.next()
    s = s[:-1]
    return s
```

Utiliser une variable temporaire pour parcourir la liste chaînée

Implémentation

```
class DoubleLinkedList:
  def __init__(self):
    self.__length = 0
    self.__head = None
    self.__last = None
  def __str__(self):
    start = self.__head
    s = ""
    while start is not Nor
      s = s + str(start.va
      start = start.next()
    s = s[:-1]
    return s
```

```
class DoubleNode:
  def __init__(self,cargo,nxt=No
    self.__cargo = cargo
    self.__next  = nxt
    self.__prev  = prv
  def __str__(self):
    return str(self.value())
  def value(self):
    return self.__cargo

+ méthodes accesseurs
    prev()
    next()

+ méthodes mutateurs
    set_prev(node)
    set_next(node)
```

Points importants

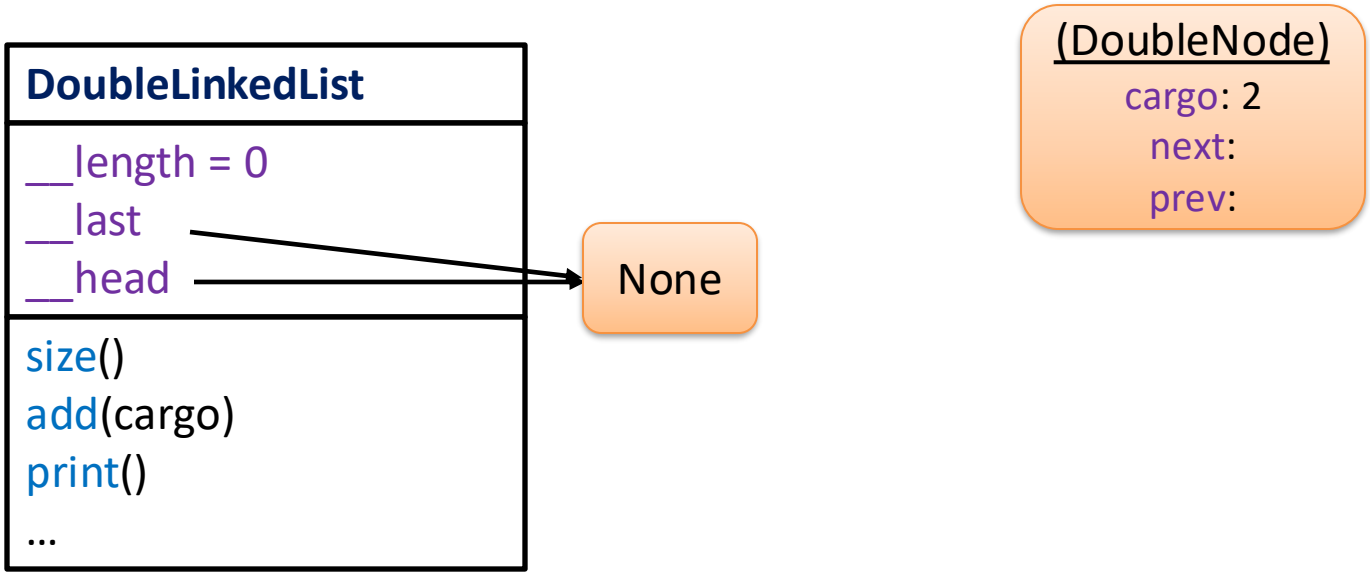
```
class DoubleLinkedList :
```

1. Lors de l'implémentation des opérations (imprimer, *ajout*, *suppression*, *insertion*, …):
   - bien tenir compte des différents cas comme *vide* / *début* / *fin*
   - ne pas oublier de mettre à jour les attributs d'instance *__head*, *__last* et *__length*
   - ne pas utiliser *__head* mais une variable temporaire pour parcourir la chaîne
   - mettre à jour les références des nœuds après l'ajout, suppression, insertion d'un nœud
2. La différence entre « is » et « == »

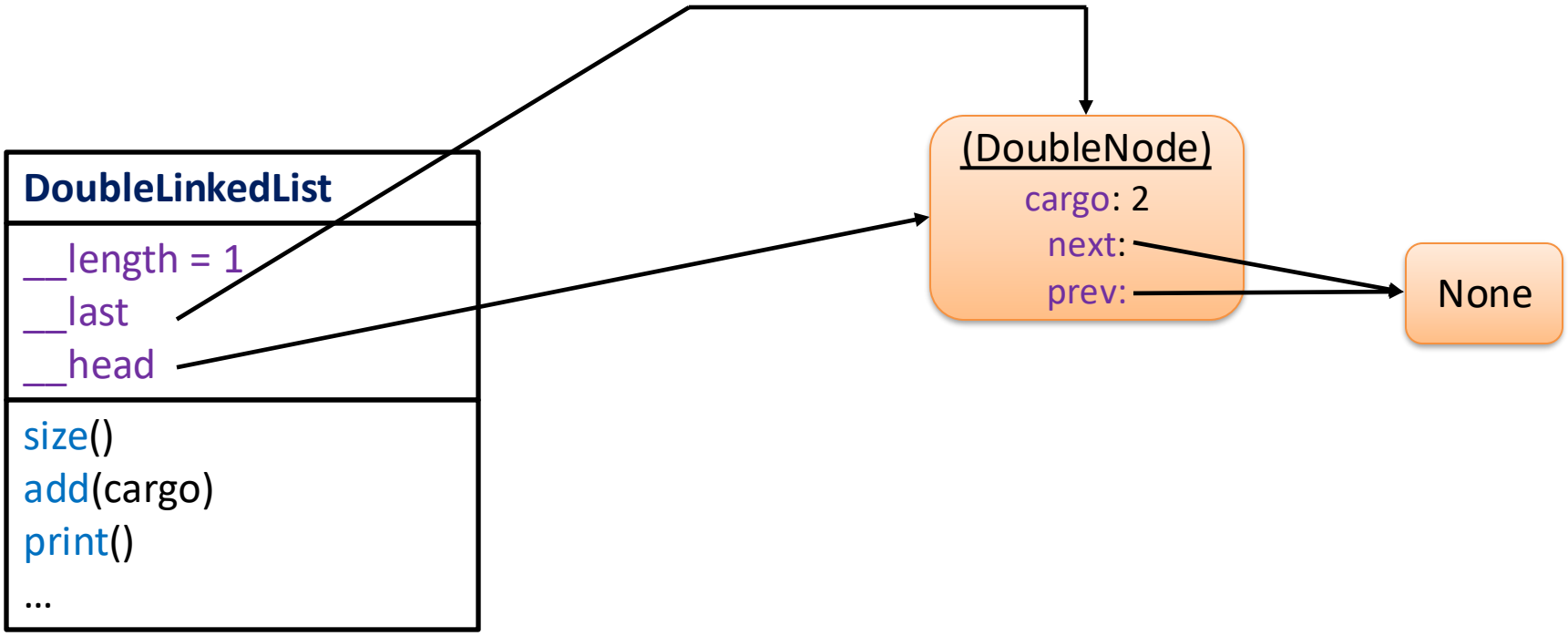


```
    def remove(self,node):
```

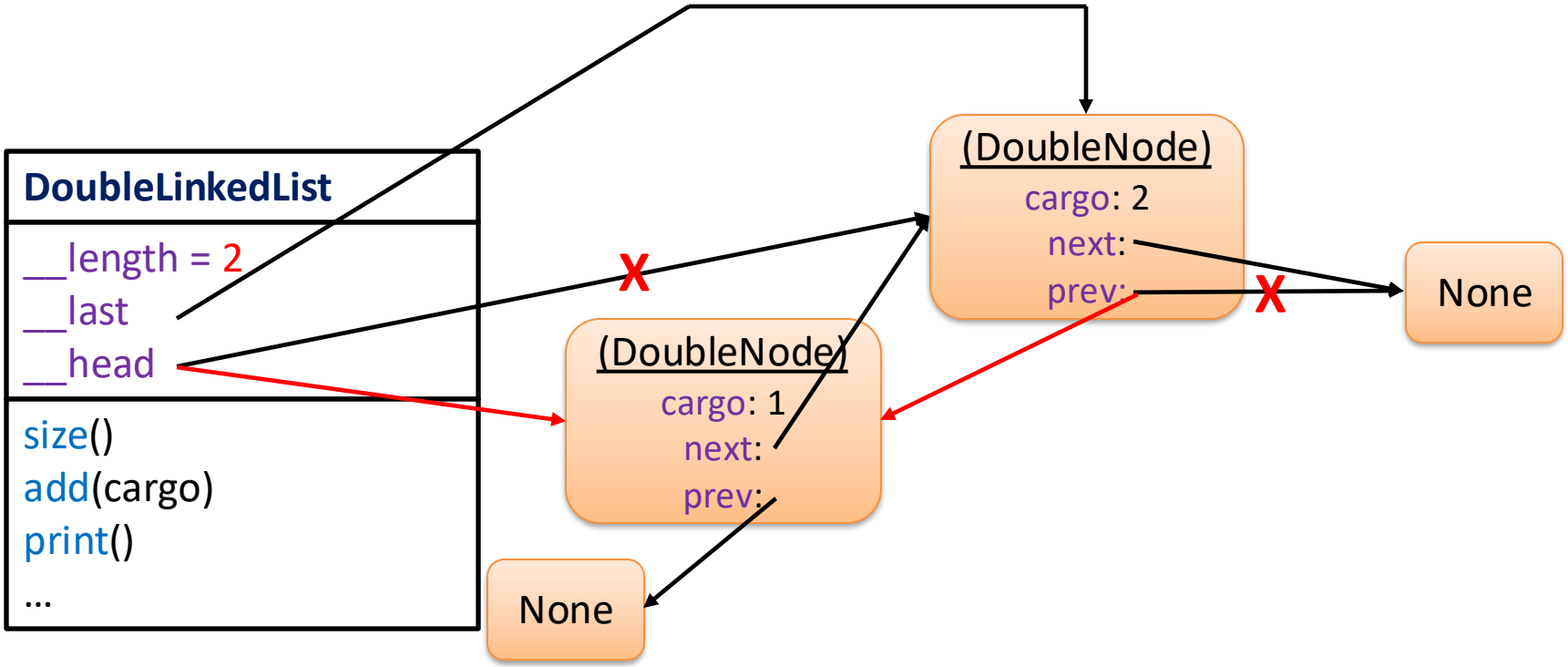
Liste doublement chaînée : ajouter (au début)



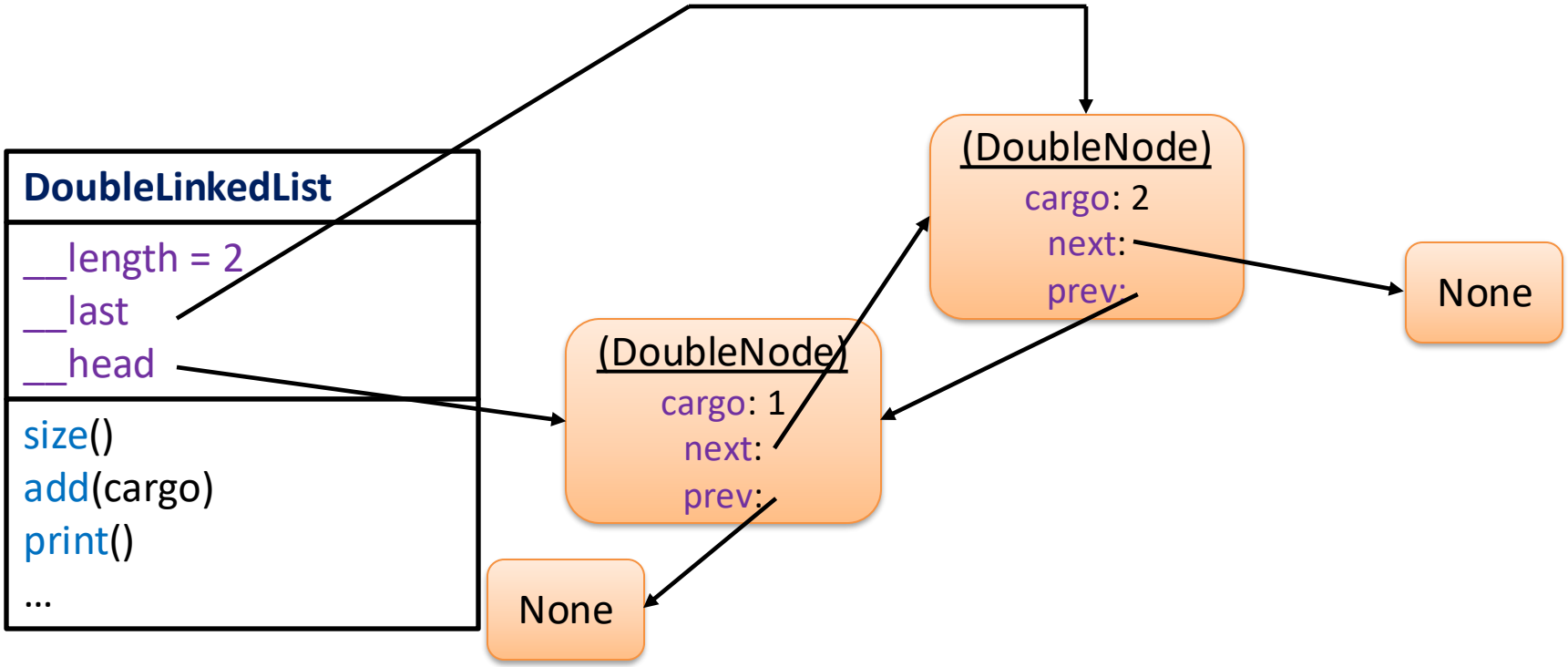
Liste doublement chaînée : ajouter (au début)



Liste doublement chaînée : ajouter (au début)



Liste doublement chaînée : ajouter (au début)



Liste doublement chaînée : ajouter (au début)

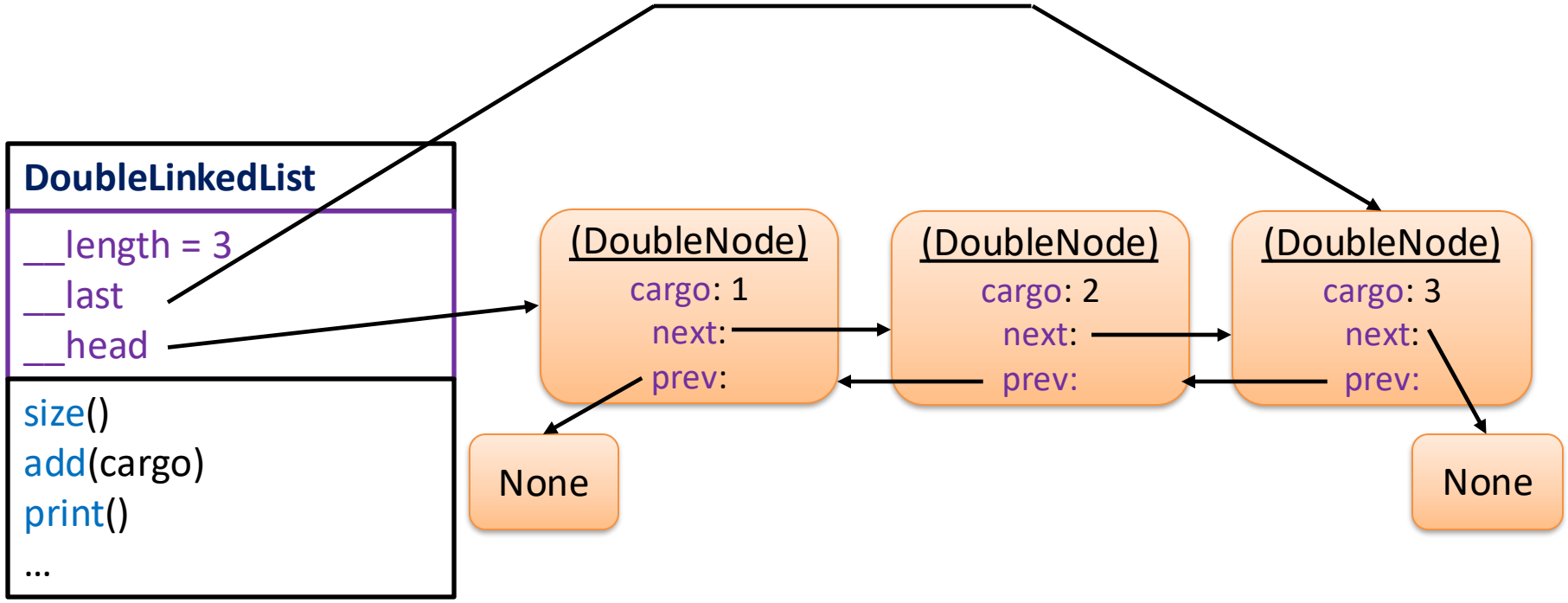
```
class DoubleLinkedList:
  …
  def add_to_start(self, cargo):
    nxt =  self.__head
    prv =  None
    node = DoubleNode(cargo,nxt,prv)
    if self.size() == 0 :
      self.__last = node
    else :
      nxt.set_prev(node)
    self.__head = node
    self.__length += 1
```

Cas spécial : ajout du 1ier nœud.

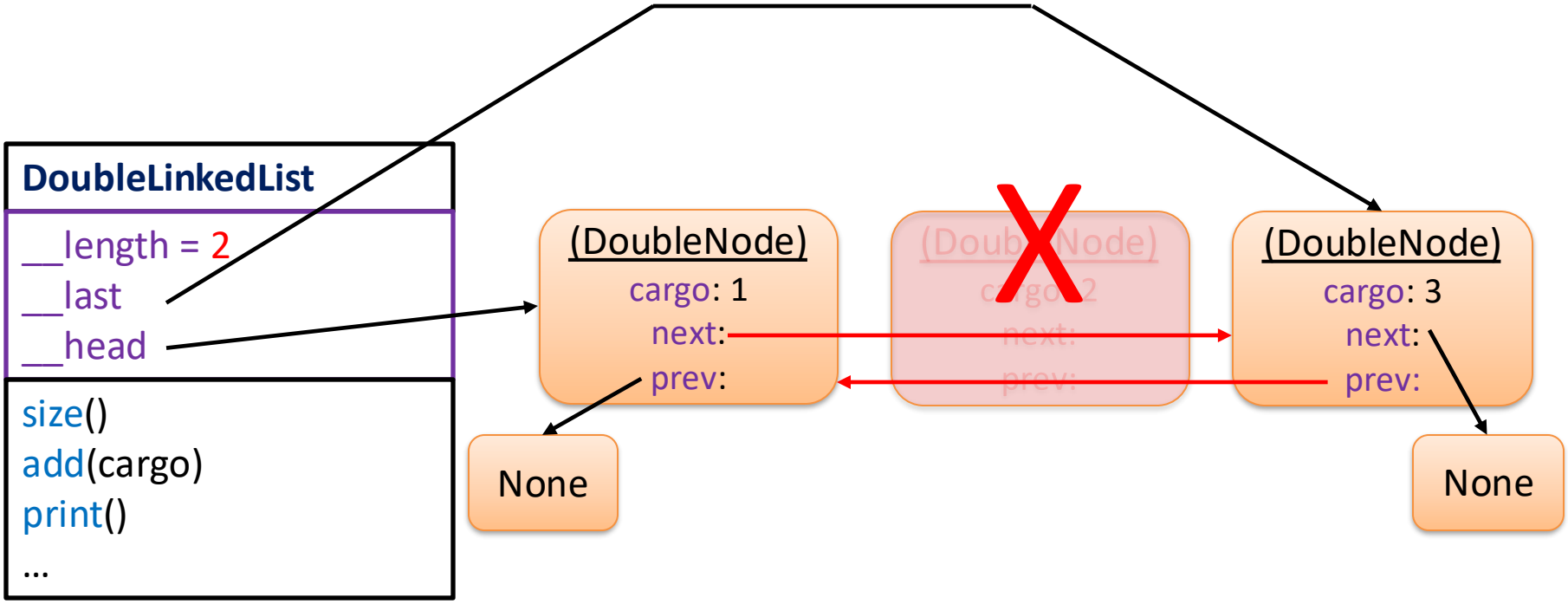
Ne pas oublier de mettre à jour __last, __head, et __length.

Aussi mettre à jour les pointeurs des nœuds existants.

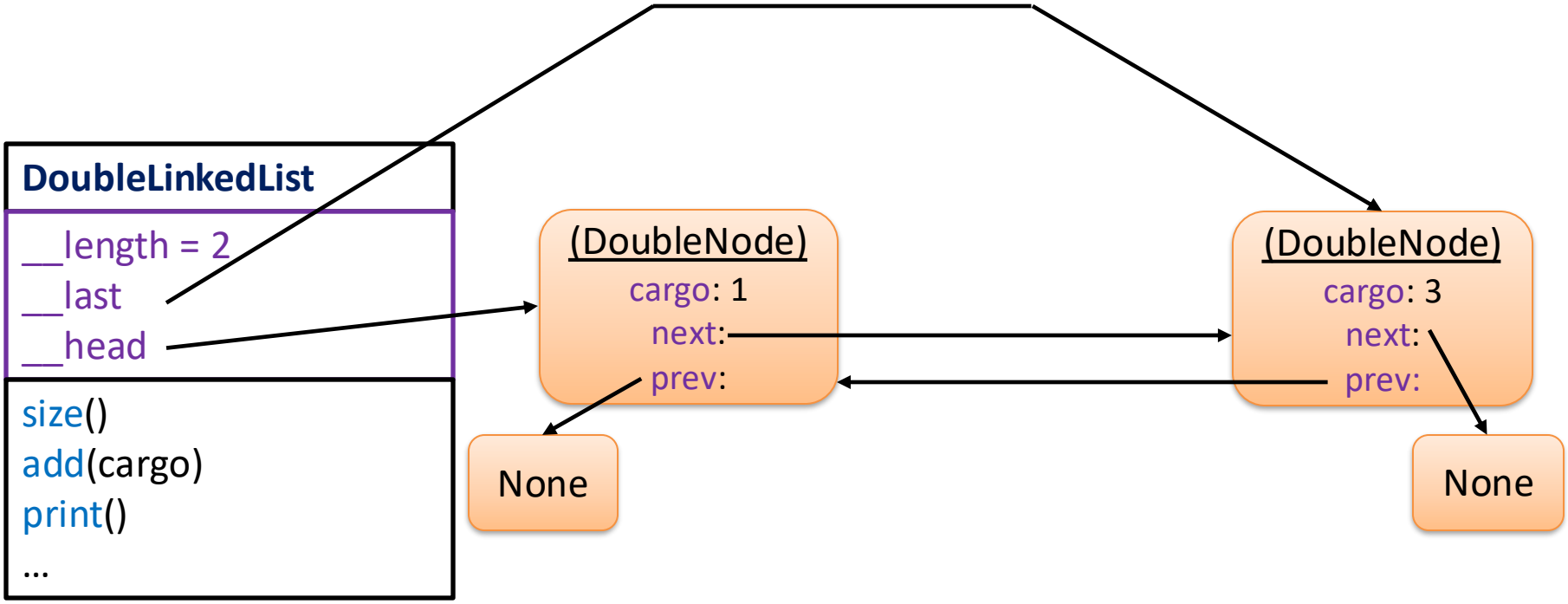
Liste doublement chaînée : suppression (d'un nœud donné)



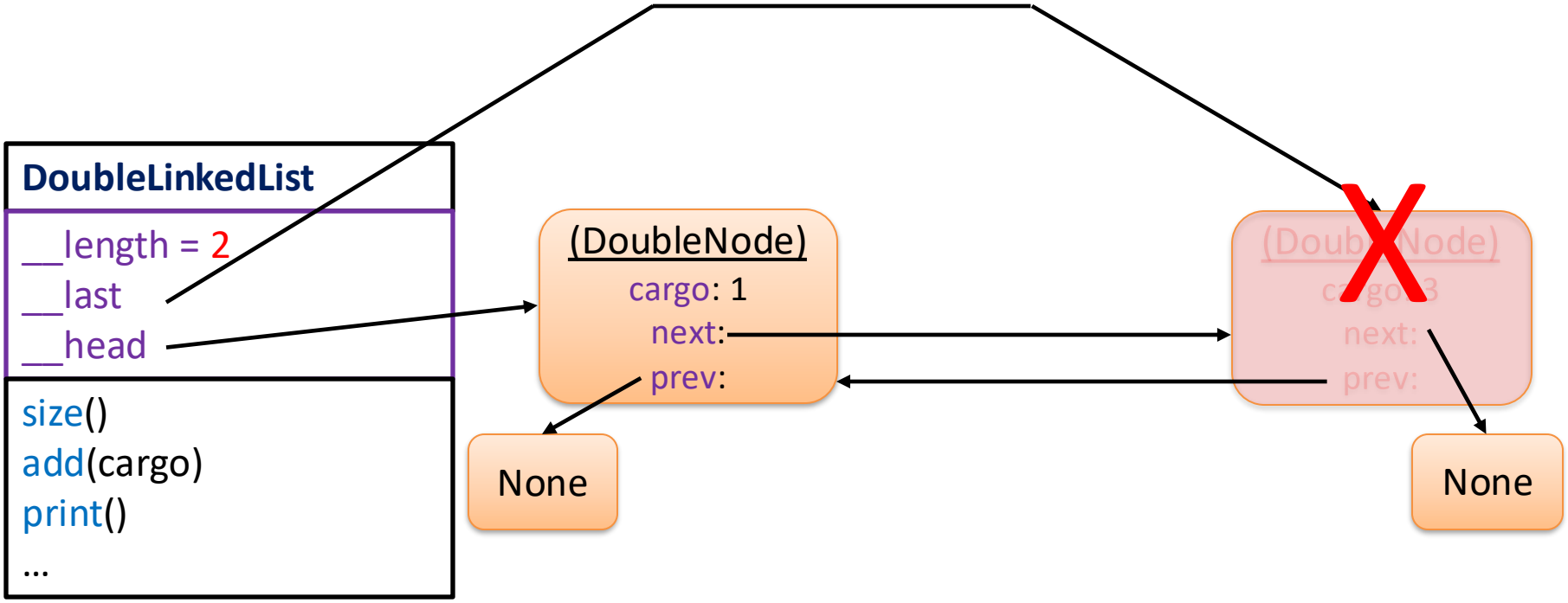
Liste doublement chaînée : suppression (d'un nœud donné)



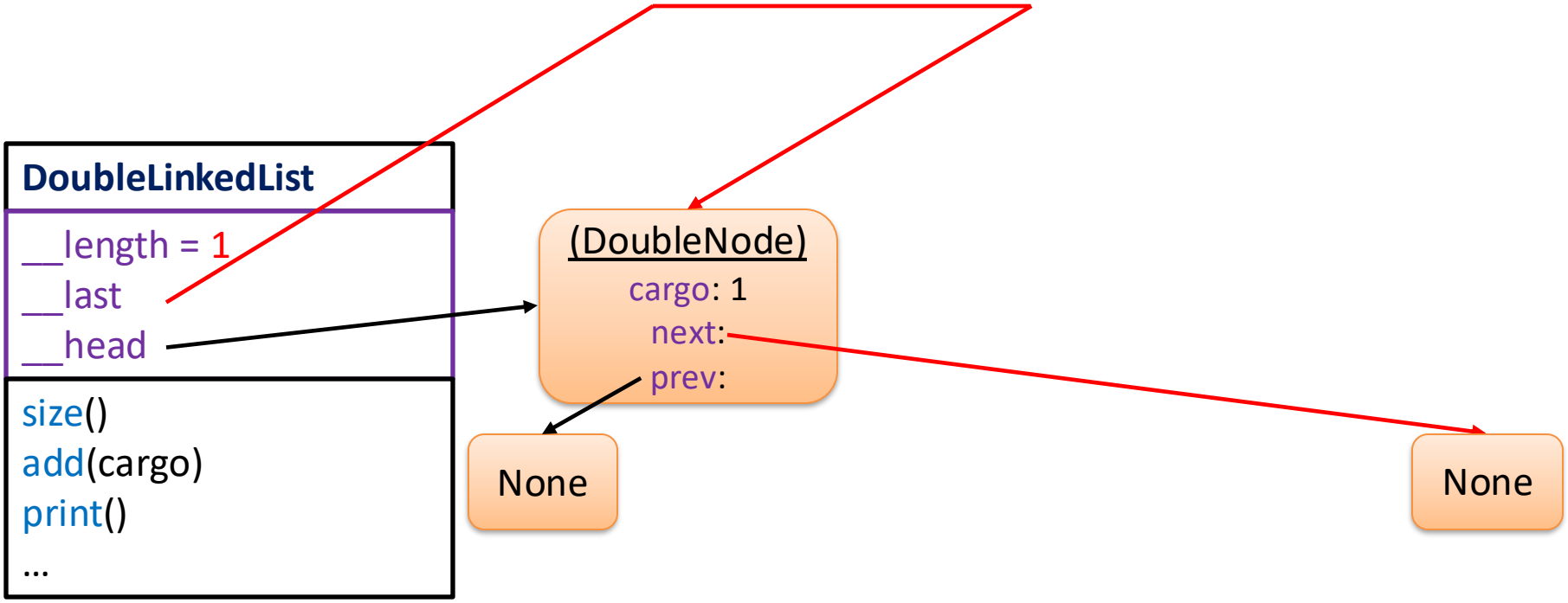
Liste doublement chaînée : suppression (d'un nœud donné)



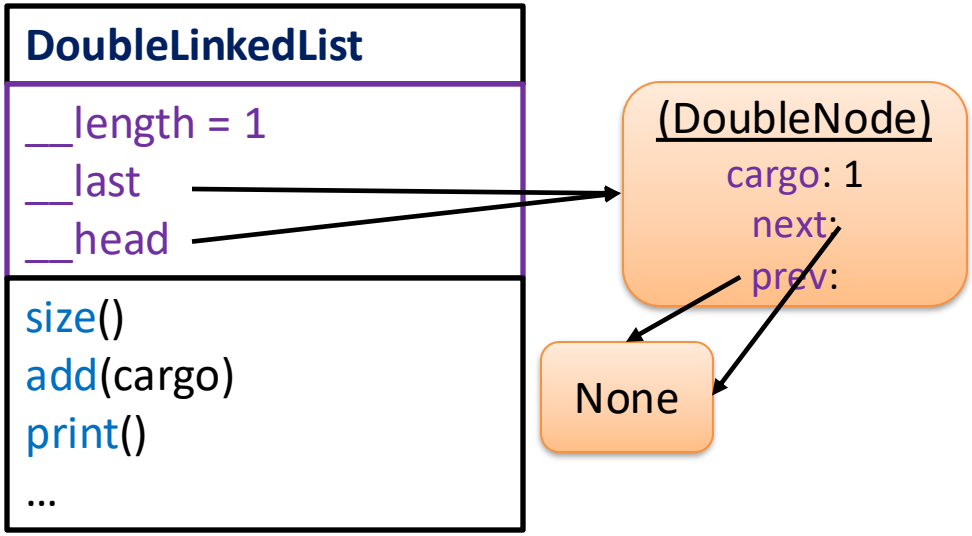
Liste doublement chaînée : suppression (d'un nœud donné)



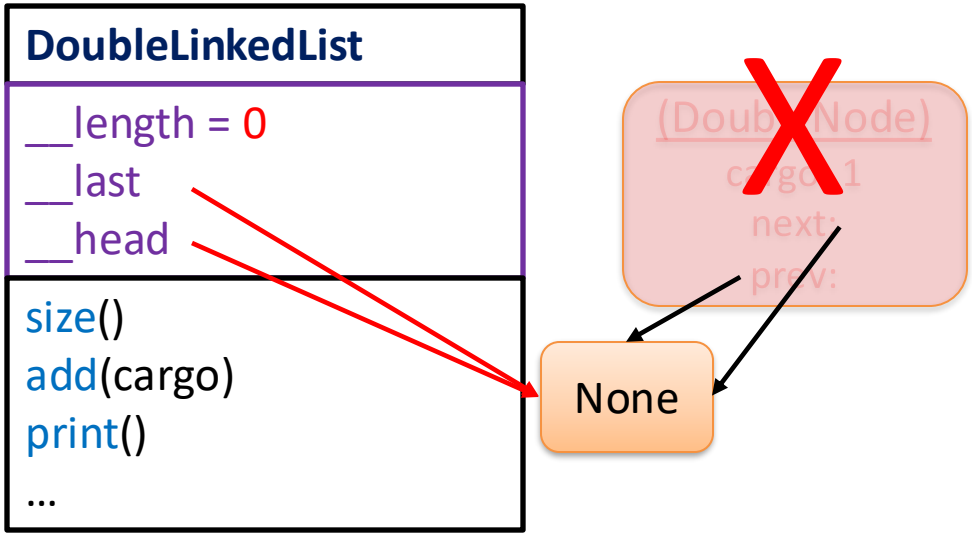
Liste doublement chaînée : suppression (d'un nœud donné)



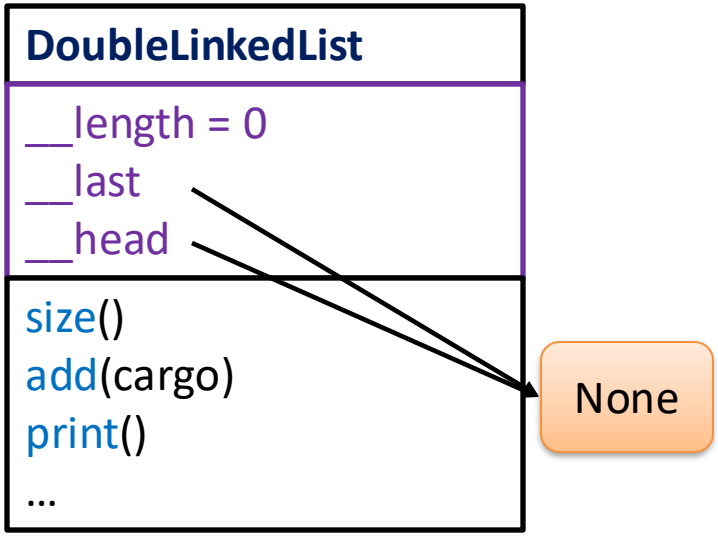
Liste doublement chaînée : suppression (d'un nœud donné)



Liste doublement chaînée : suppression (d'un nœud donné)



Liste doublement chaînée : suppression (d'un nœud donné)



Liste doublement chaînée : suppression (d'un nœud donné)

```
class DoubleLinkedList:
  …

  def remove(self,node):
    if self.size() == 1:
      self.__head = None
      self.__last = None
    elif node is self.__head:
      self.__head = node.next()
      node.next().set_prev(None)
    elif node is self.__last:
      self.__last = node.prev()
      node.prev().set_next(None)
    else :
      node.prev().set_next(node.next())
      node.next().set_prev(node.prev())
    self.__length -= 1
```

Cas 1 : suppression du dernier nœud restant

Cas 2 : suppression du nœud en tête de la liste

Cas 3 : suppression du nœud en queue de la liste

Tout autres cas

Liste doublement chaînée : suppression (d'un nœud donné)

```
class DoubleLinkedList:
  …

  def remove(self,node):
    if self.size() == 1:
      self.__head = None
      self.__last = None
    elif node is self.__head:
      self.__head = node.next()
      node.next().set_prev(None)
    elif node is self.__last:
      self.__last = node.prev()
      node.prev().set_next(None)
    else :
      node.prev().set_next(node.next())
      node.next().set_prev(node.prev())
    self.__length -= 1
```

Ne pas oublier de mettre à jour __last *et* __head

Ne pas oublier de mettre à jour __head

Ne pas oublier de mettre à jour __last

Ne pas oublier de mettre à jour __length

Liste doublement chaînée : suppression (d'un nœud donné)

```
class DoubleLinkedList:
  …

  def remove(self,node):
    if self.size() == 1:
      self.__head = None
      self.__last = None
    elif node is self.__head:
      self.__head = node.next()
      node.next().set_prev(None)
    elif node is self.__last:
      self.__last = node.prev()
      node.prev().set_next(None)
    else :
      node.prev().set_next(node.next())
      node.next().set_prev(node.prev())
    self.__length -= 1
```

Aussi mettre à jour les références des nœuds existants.

Points importants

```
class DoubleLinkedList:
```

1. Lors de l'implémentation des opérations (*ajout*, *suppression*, *insertion*, …): …
2. La différence entre « is » et « == »

   « x is y » vérifie si x et y font référence à un même objet

   « x == y » vérifie si x et y ont la même valeur (selon __eq__)

   « x is None » est préféré pour comparer avec None

Liste doublement chaînée : suppression (d'un nœud donné)

```
class DoubleLinkedList:
  …

  def remove(self,node):
    if self.size() == 1:
      self.__head = None
      self.__last = None
    elif node is self.__head:
      self.__head = node.next()
      node.next().set_prev(None)
    elif node is self.__last:
      self.__last = node.prev()
      node.prev().set_next(None)
    else :
      node.prev().set_next(node.next())
      node.next().set_prev(node.prev())
    self.__length -= 1
```

Est-ce que node et __head pointent vers le même objet?

Est-ce que node et __last pointent vers le même objet?