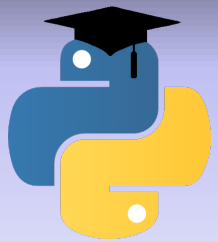
Langage et algorithmique
- Listes chaînées -

A. Malek TOUMI

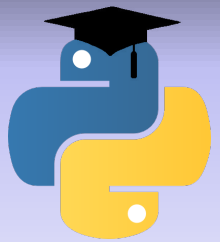
toumiab@ensta-bretagne.fr

2015/2016

ENSTA Bretagne



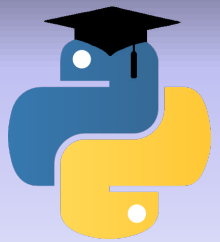
Sommaire



Principe

Définition (Liste chaînée)

Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.

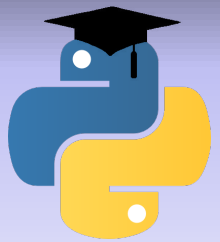


Principe

Définition (Liste chaînée)

Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.

Plusieurs types de listes :



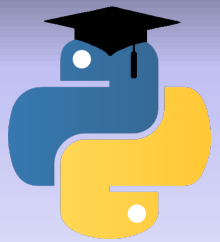
Principe

Définition (Liste chaînée)

Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.

Plusieurs types de listes :

- simplement chaînée



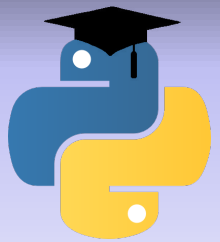
Principe

Définition (Liste chaînée)

Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.

Plusieurs types de listes :

- simplement chaînée
- doublement chaînée



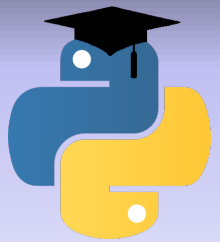
Principe

Définition (Liste chaînée)

Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.

Plusieurs types de listes :

- simplement chaînée
- doublement chaînée
- triée



Principe

Définition (Liste chaînée)

Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.

Plusieurs types de listes :

- simplement chaînée
- doublement chaînée
- triée
- circulaire

Principe

Définition (Liste chaînée)

Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.

Principe

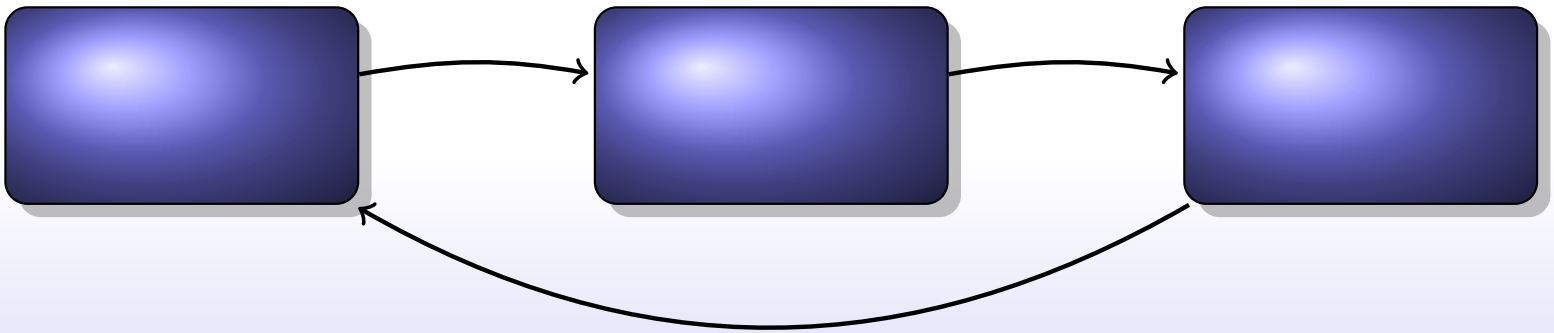
Définition (Liste chaînée)

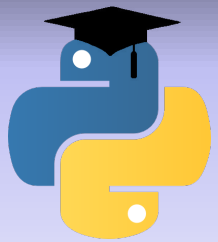
Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.

Principe

Définition (Liste chaînée)

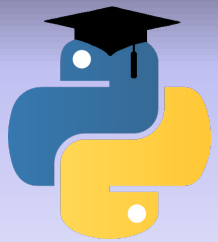
Liste chaînée : structure algorithmique **dynamique**. Accès **séquentiel** aux éléments.





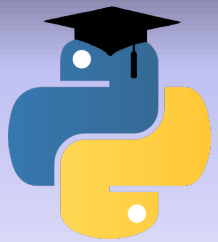
TAD liste

- Liste composée d'éléments (noeuds)



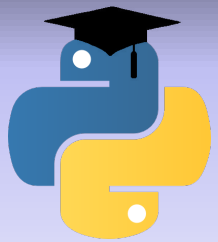
TAD liste

- Liste composée d'éléments (noeuds)
- Noeuds chaînés entre eux :



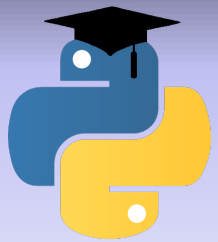
TAD liste

- Liste composée d'éléments (noeuds)
- Noeuds chaînés entre eux :
- Chaque noeud connaît son successeur



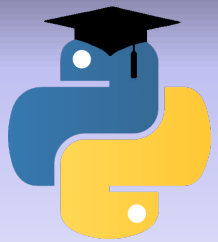
TAD liste

- Liste composée d'éléments (noeuds)
- Noeuds chaînés entre eux :
- Chaque noeud connaît son successeur
- Fin de liste : **None**



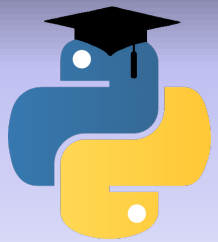
TAD liste

- Liste composée d'éléments (noeuds)
- Noeuds chaînés entre eux :
- Chaque noeud connaît son successeur
- Fin de liste : **None**
- Chaque noeud peut connaître son prédécesseur (liste doublement chaînée)



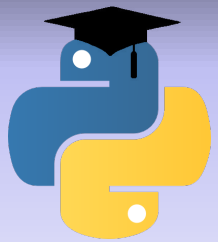
TAD liste

- Liste composée d'éléments (noeuds)
- Noeuds chaînés entre eux :
- Chaque noeud connaît son successeur
- Fin de liste : **None**
- Chaque noeud peut connaître son prédécesseur (liste doublement chaînée)
- Liste = premier élément (noeud) + opérations



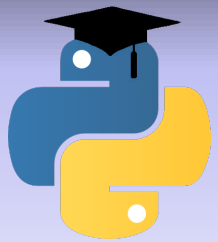
TAD liste

- Liste composée d'éléments (noeuds)
- Noeuds chaînés entre eux :
- Chaque noeud connaît son successeur
- Fin de liste : **None**
- Chaque noeud peut connaître son prédécesseur (liste doublement chaînée)
- Liste = premier élément (noeud) + opérations
  - Insertion



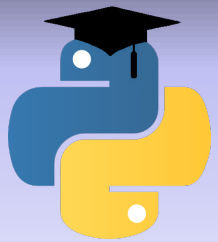
TAD liste

- Liste composée d'éléments (noeuds)
- Noeuds chaînés entre eux :
- Chaque noeud connaît son successeur
- Fin de liste : **None**
- Chaque noeud peut connaître son prédécesseur (liste doublement chaînée)
- Liste = premier élément (noeud) + opérations
  - Insertion
  - Suppression



TAD liste

- Liste composée d'éléments (noeuds)
- Noeuds chaînés entre eux :
- Chaque noeud connaît son successeur
- Fin de liste : **None**
- Chaque noeud peut connaître son prédécesseur (liste doublement chaînée)
- Liste = premier élément (noeud) + opérations
  - Insertion
  - Suppression
  - Recherche

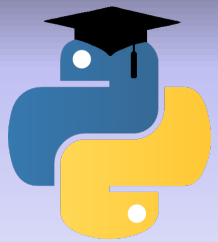


Classe noeud

- Constructeur (Noeud) :
- $\Longrightarrow$Noeud : données

Exemple

```
class Node (object):
   def __init__(self, donnee = 0):
      """ Creation d'un noeud
      le noeud suivant est fixé à None
      """
      self.val = donnee # La valeur de noeud
```

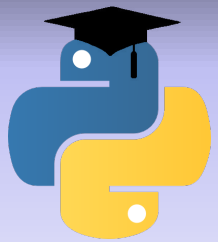


Classe noeud

- Constructeur (Noeud) :
- $\Longrightarrow$Noeud : données + variable pour le chaînage

Exemple

```
class Node (object):
   def __init__(self, donnee = 0):
      """ Creation d'un noeud
      le noeud suivant est fixé à None
      """
      self.val = donnee # La valeur de noeud
      self.next = None # Le noeud suivant
```

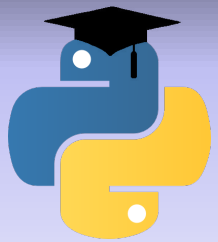


Classe noeud

- Constructeur (Noeud) :
- $\Longrightarrow$Noeud : données + variable pour le chaînage
- Les opérations sur les noeuds : affichage, comparaison (==, >, >,<=, ...)

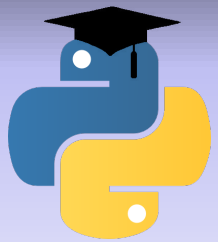
Exemple

```
class Node (object):
   def __init__(self, donnee = 0):
      """ Creation d'un noeud
      le noeud suivant est fixé à None
      """
      self.val = donnee # La valeur de noeud
      self.next = None # Le noeud suivant
    ...
```



Classe List

- Constructeur (List) :
- $\Longrightarrow$ contenu : tête de liste
- Les opérations sur les listes :affichage, Parcours, insertion, Suppression, ...

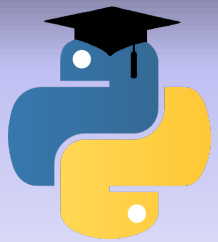


Classe List

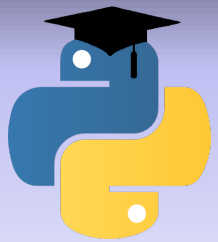
- Constructeur (List) :
- $\Longrightarrow$ contenu : tête de liste
- Les opérations sur les listes :affichage, Parcours, insertion, Suppression, ...

Exemple

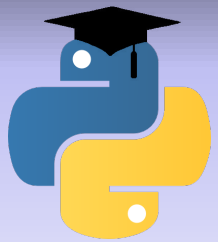
```
class List(object):
   def __init__(self):
      """ Creation d'une liste vide
      la tete de liste est fixée à None
      """
      self.__first = None

    ...
```



Sommaire



Sommaire



Parcours de liste

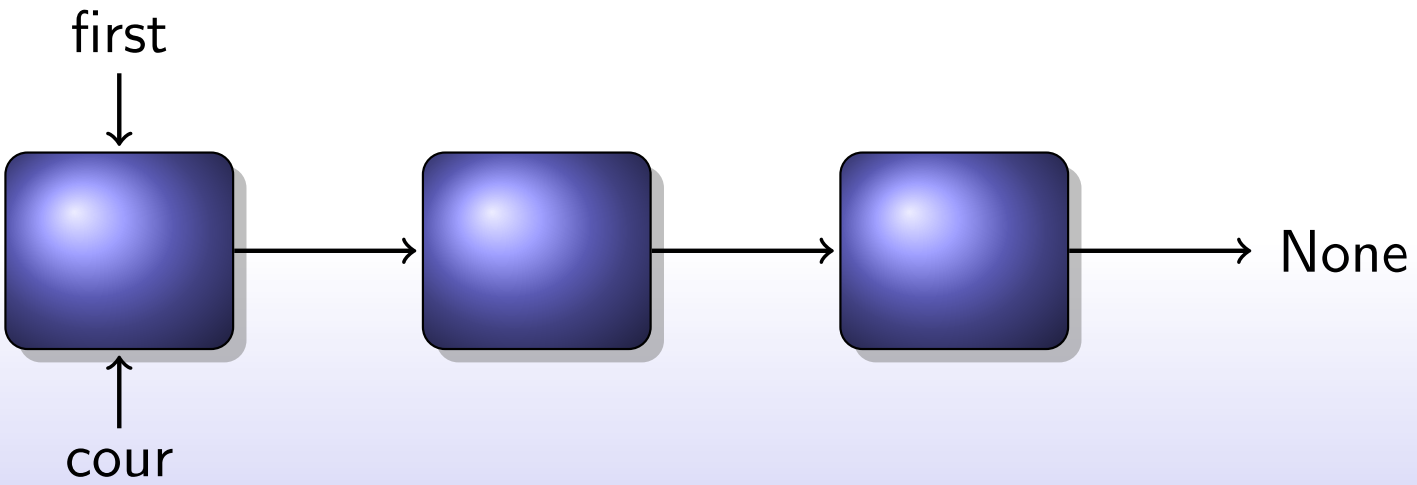
Principe :

- partir du début de la liste
- avancer tant que l'élément est différent de **None**

Parcours de liste

Principe :

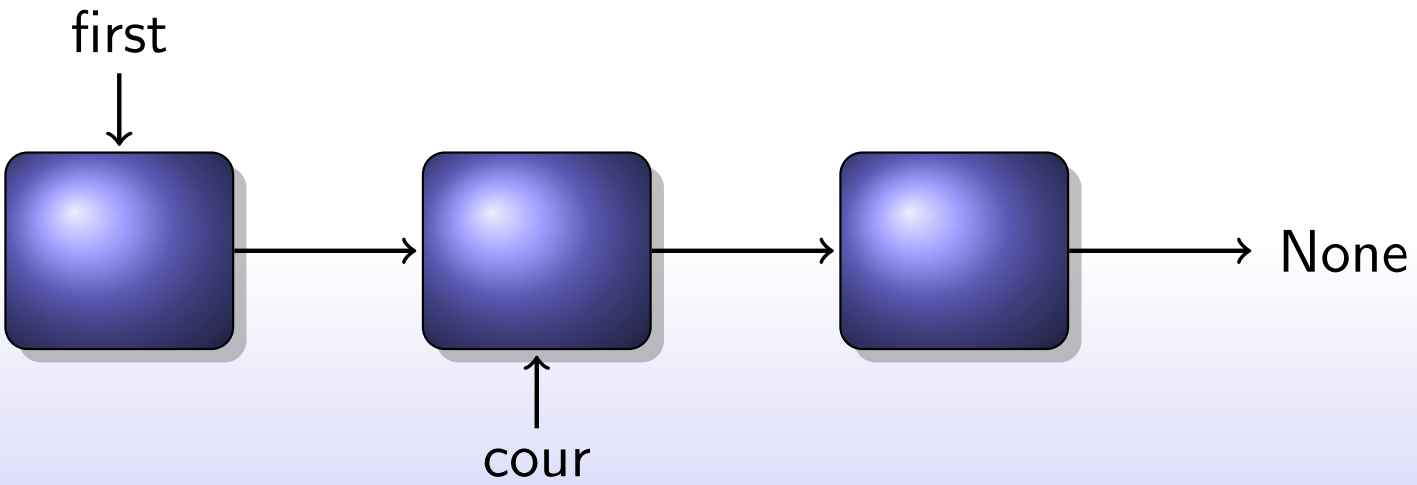
- partir du début de la liste
- avancer tant que l'élément est différent de **None**



Parcours de liste

Principe :

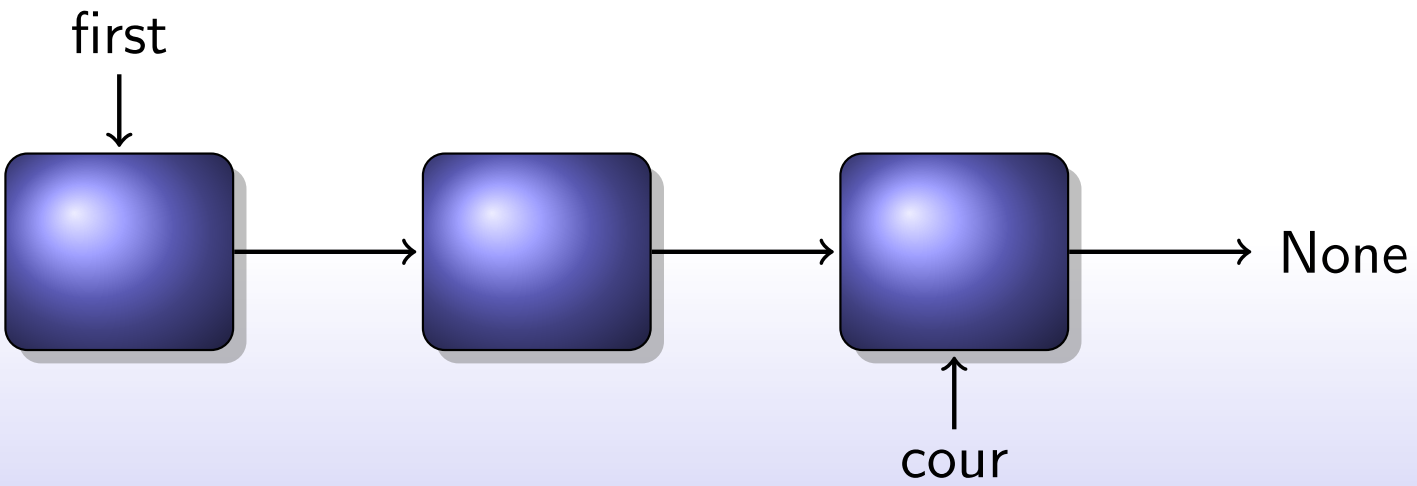
- partir du début de la liste
- avancer tant que l'élément est différent de **None**



Parcours de liste

Principe :

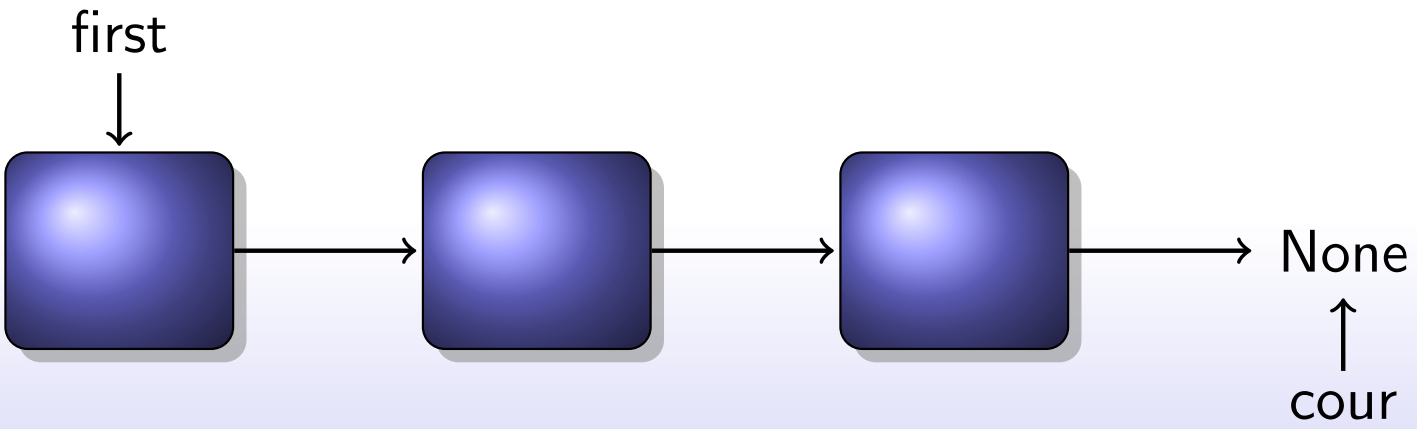
- partir du début de la liste
- avancer tant que l'élément est différent de **None**

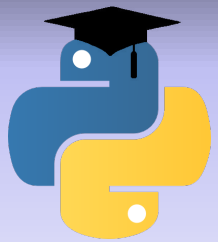


Parcours de liste

Principe :

- partir du début de la liste
- avancer tant que l'élément est différent de **None**





Algorithme de parcours

Algorithme 1: Parcours de liste

Entrées : premier : noeud `// tête de liste`

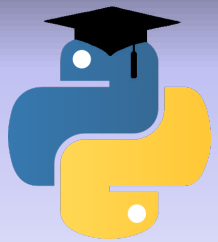
Données : n : noeud `// noeud courant`

$n \leftarrow premier$;

tant que $n \neq None$ **faire**

    $n \leftarrow n.suivant$;

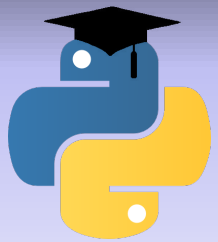
fin



Algorithme de parcours

Parcours d'une liste avec une boucle **while** :

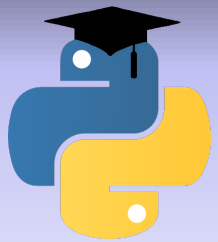
```
class List(object):
   def __init__(self):
      self.__first = None
      # ----------------------------------
   def parcours(self):
      n = self.__first
      while n is not None:
         # action à faire : exemple : print()
         print(str(n.val))
         # passer au noeud suivant
         n = n.next
```



Algorithme de parcours

Parcours **récursif** d'une liste :

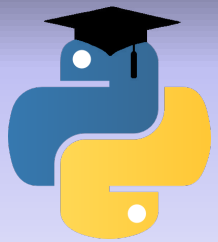
```
def parcours(self, n):
    if n is not None
        # action à faire : exemple : print()
        print(str(n.val))
        # passer au noeud suivant
        self.parcours(n.next)
```



Recherche d'un élément dans une liste

Exemple de parcours : recherche d'élément
Principe :

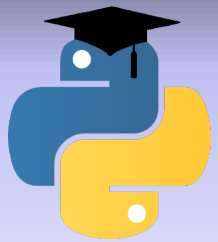
- Partir du début de la liste



Recherche d'un élément dans une liste

Exemple de parcours : recherche d'élément
Principe :

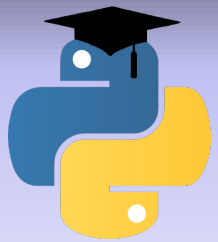
- Partir du début de la liste
- Avancer tant que l'élément n'est pas trouvé



Recherche d'un élément dans une liste

Exemple de parcours : recherche d'élément
Principe :

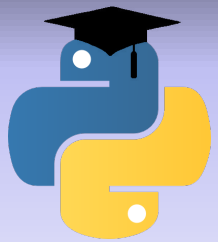
- Partir du début de la liste
- Avancer tant que l'élément n'est pas trouvé
- Et que la fin de liste n'est pas atteinte



Recherche d'un élément dans une liste

Exemple de parcours : recherche d'élément
Principe :

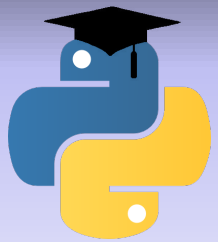
- Partir du début de la liste
- Avancer tant que l'élément n'est pas trouvé
- Et que la fin de liste n'est pas atteinte
- Si l'algorithme se termine sur la fin de liste, l'élément n'est pas présent



Algorithme

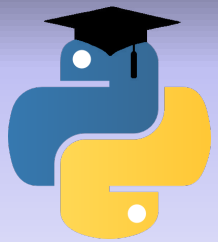
Algorithme 2: Recherche d'élément dans une liste

Entrées : int val
Données : debut : noeud `// tête de liste`
Données : n : noeud `// noeud courant`
$n \leftarrow debut$;
tant que $n \neq None$ **et** $n.valeur \neq val$ **faire**
    $n \leftarrow n.suivant$;
fin
retourner **n** ;



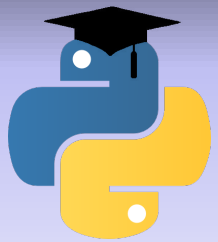
Complexité

- Nombre d'opérations nécessaire pour rechercher un élément ?



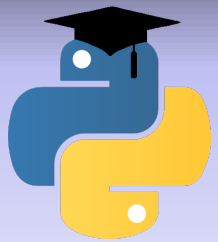
Complexité

- Nombre d'opérations nécessaire pour rechercher un élément ?
- Liste de taille n



Complexité

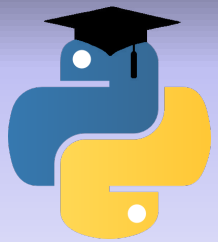
- Nombre d'opérations nécessaire pour rechercher un élément ?
- Liste de taille n
- Complexité $\Theta(n)$ (pire des cas et moyenne)



Complexité

- Nombre d'opérations nécessaire pour rechercher un élément ?
- Liste de taille n
- Complexité $\Theta(n)$ (pire des cas et moyenne)

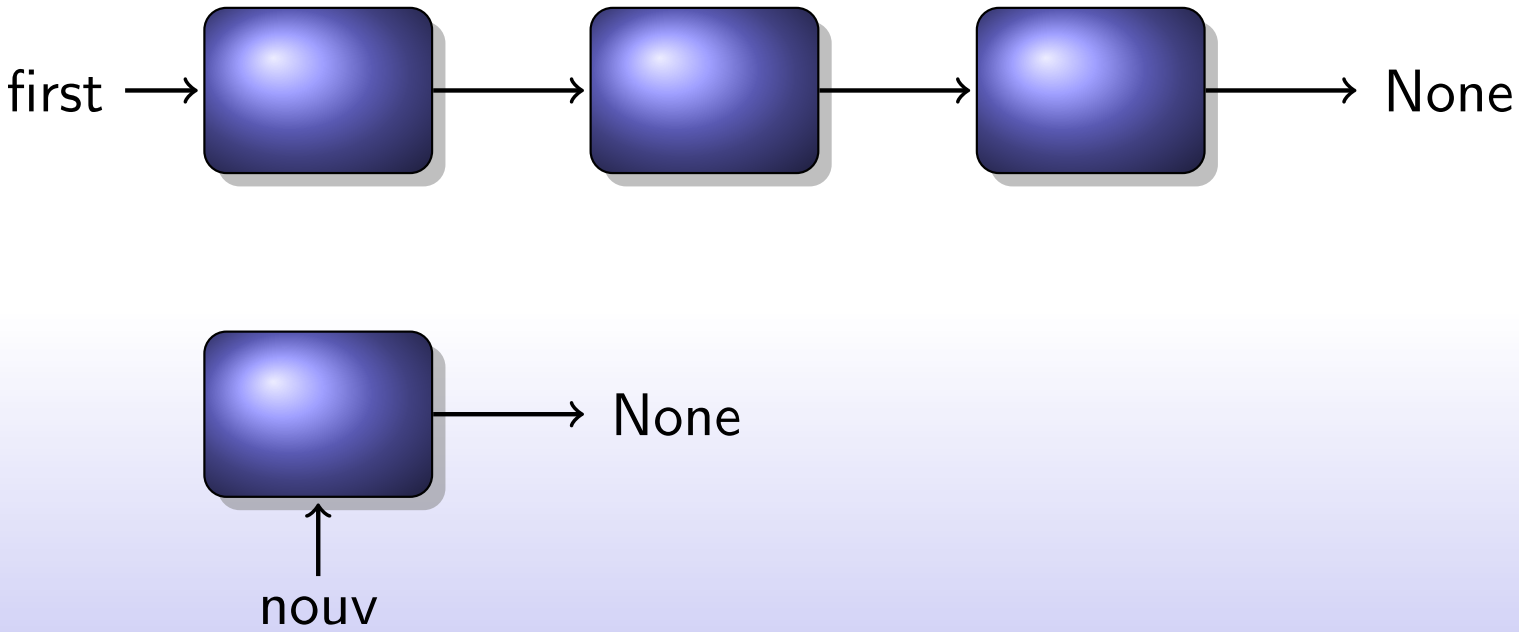
$\Rightarrow$ Méthode de représentation peu efficace



Sommaire

Insertion en début de liste

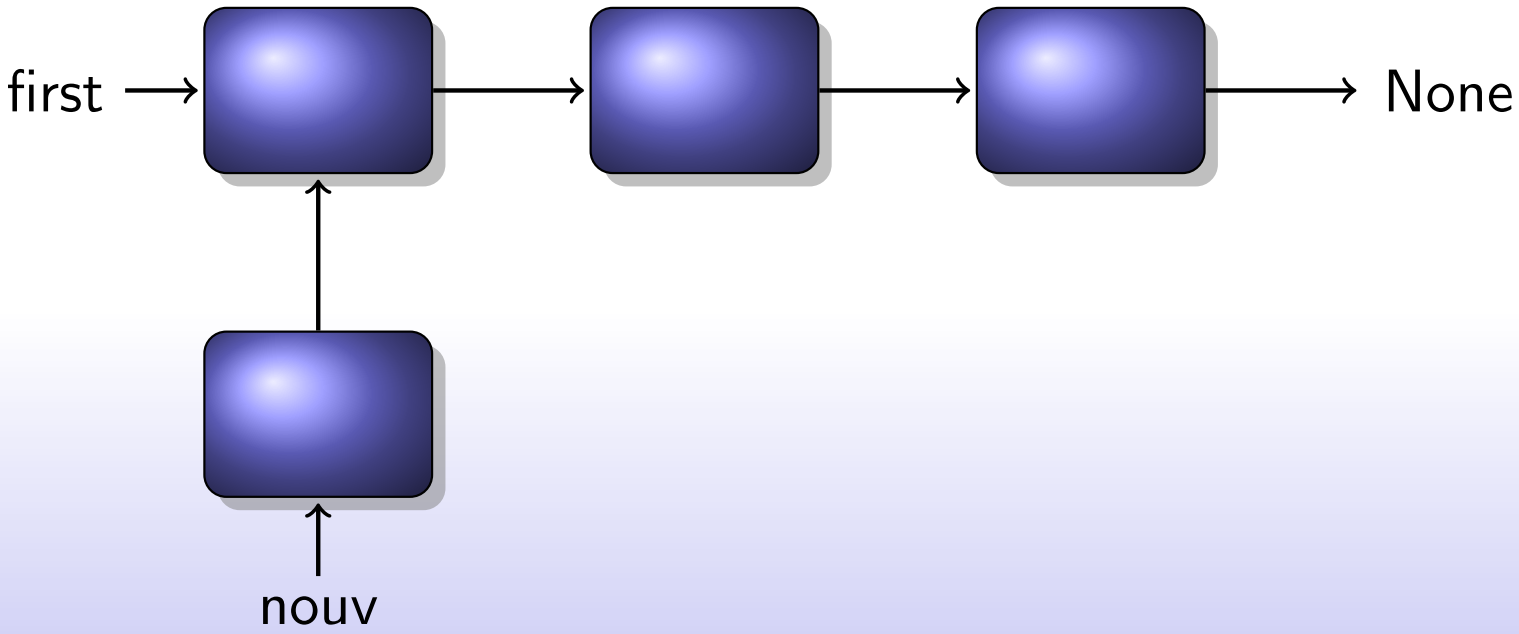
Principe :



Insertion en début de liste

Principe :

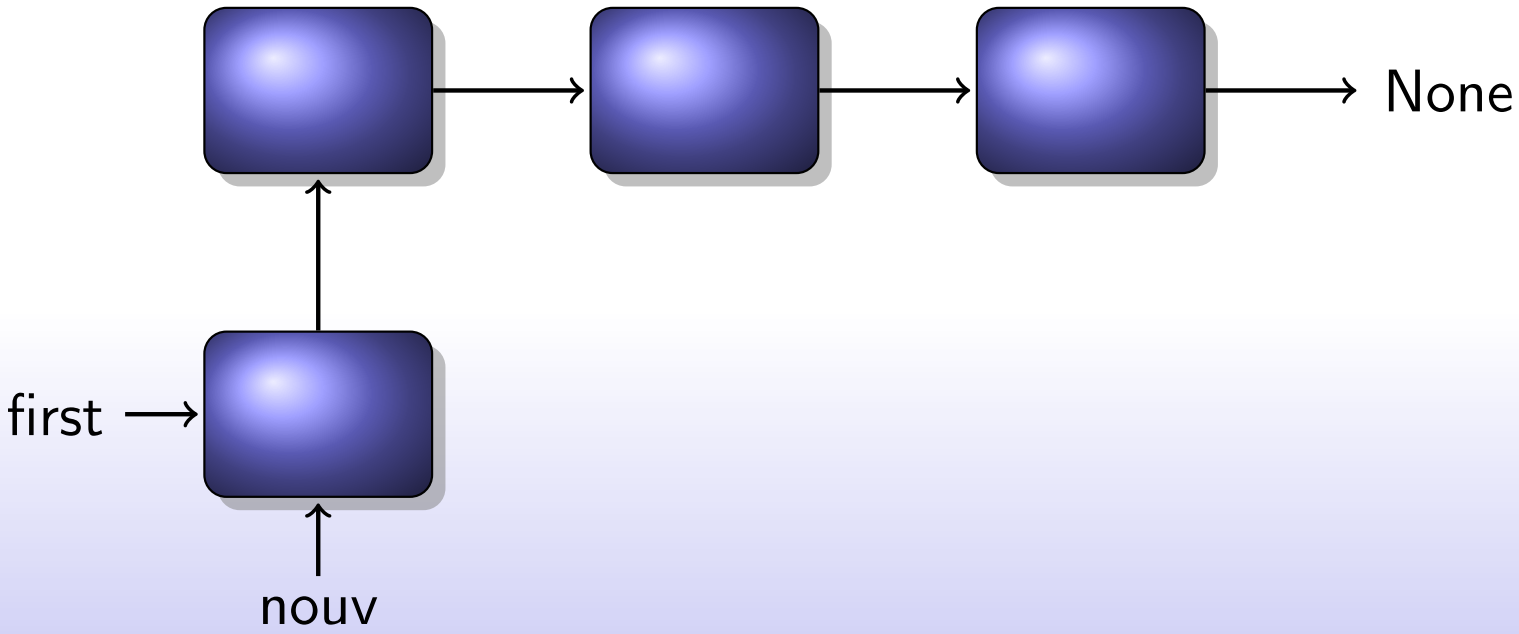
- Attacher le début de la liste à **nouv**

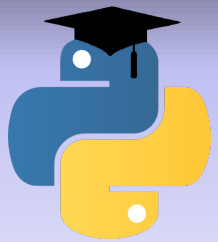


Insertion en début de liste

Principe :

- Attacher le début de la liste à **nouv**
- Le nouveau début de liste est **nouv**





Algorithme

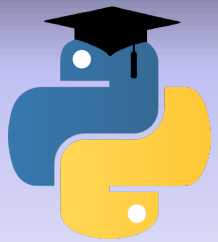
Algorithme 3: Insertion en début de liste

Entrées : nouv : noeud à ajouter

Données : debut : noeud `// tête de liste`

nouv.suivant $\leftarrow$ *debut*;

debut $\leftarrow$ *nouv*;



Algorithme

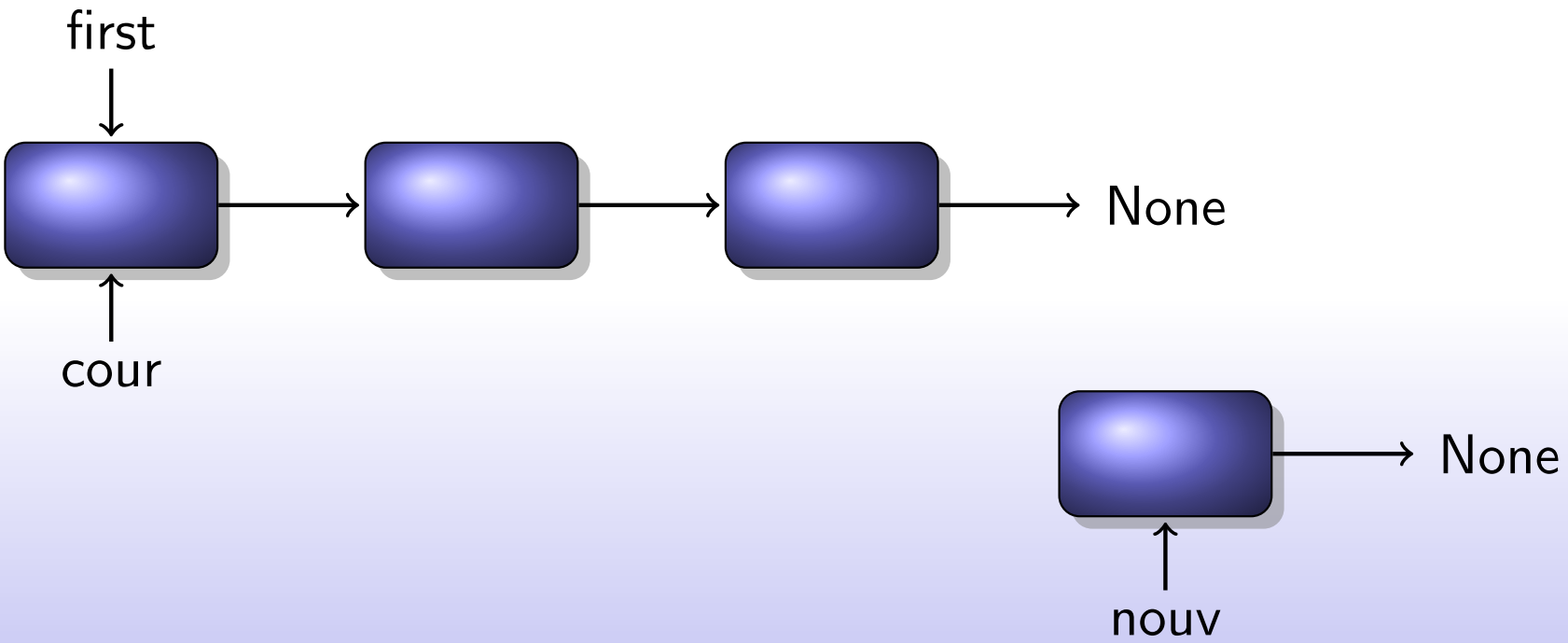
```
def insertFirst(self, val) :
    nouv = Node(val)
    nouv.next = self.__first
    self.__first = nouv;
```

Complexité : $\Theta(1)$

Insertion en fin de liste

Principe :

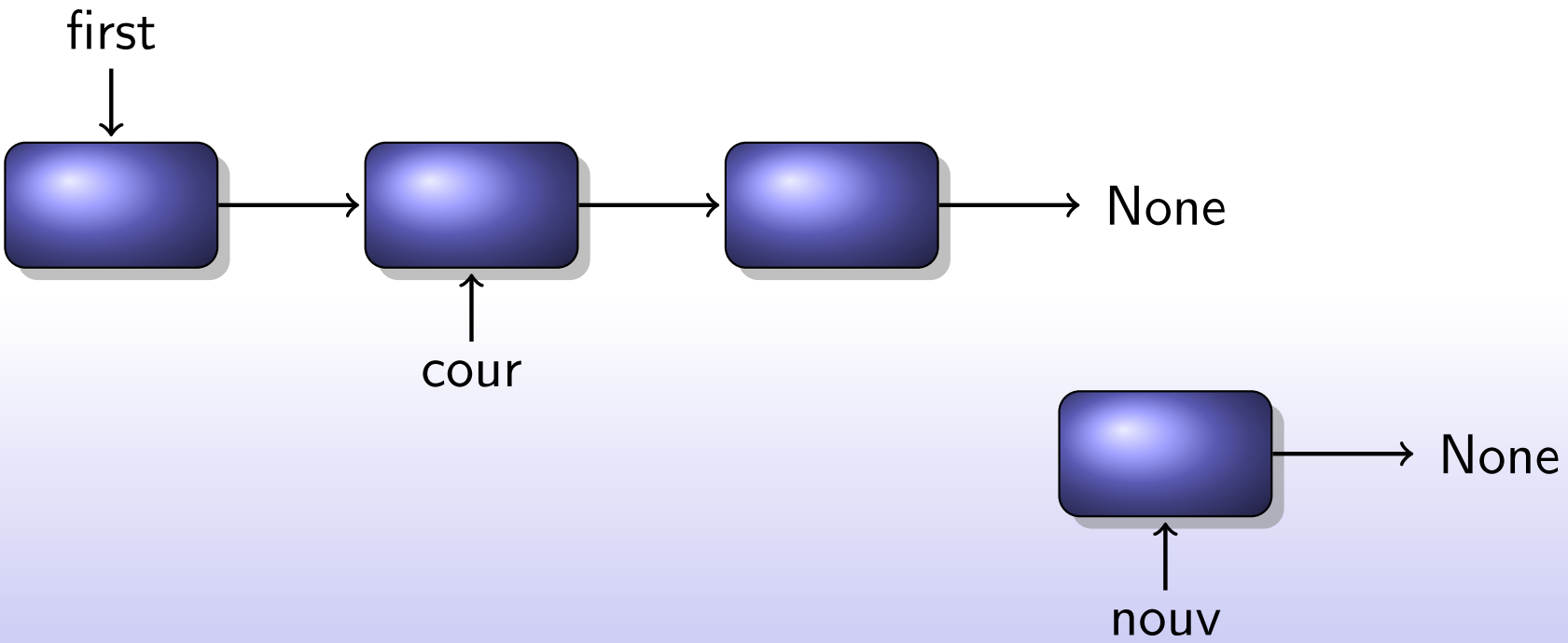
- Se placer en début de liste



Insertion en fin de liste

Principe :

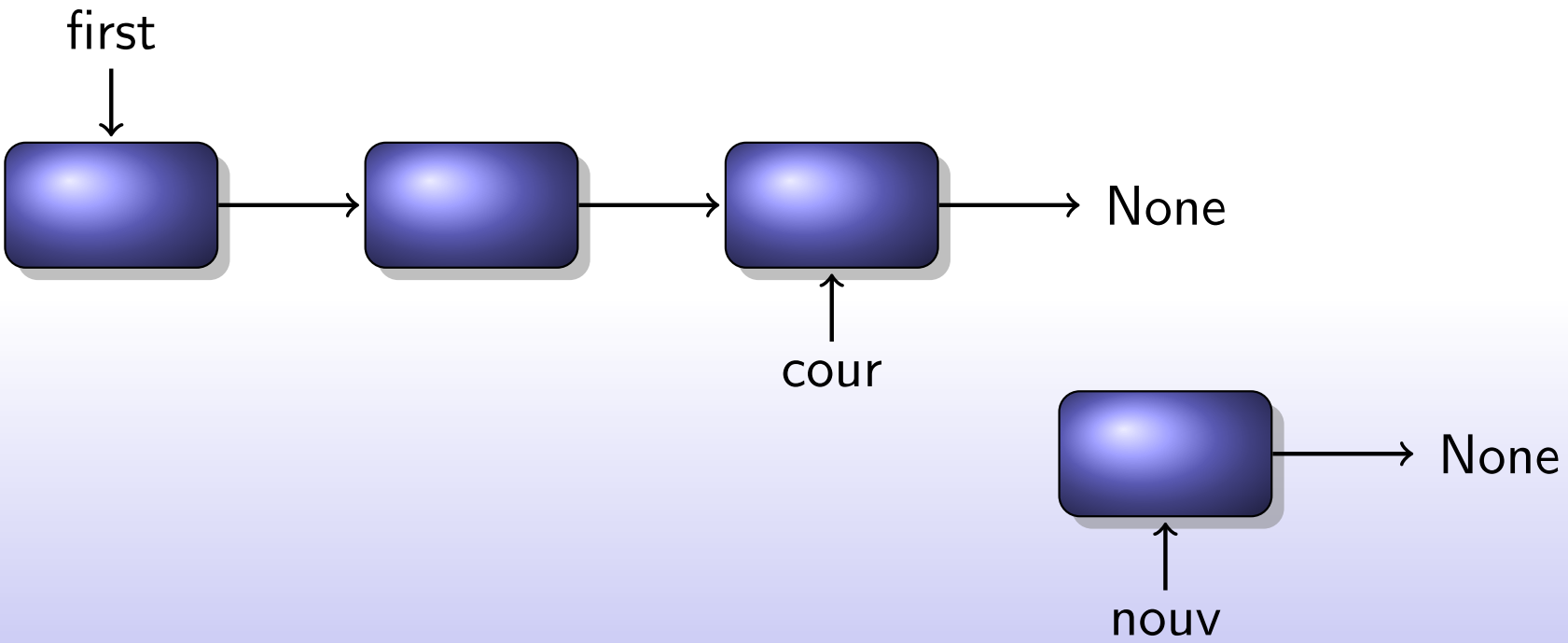
- Se placer en début de liste
- Avancer jusqu'au dernier élément



Insertion en fin de liste

Principe :

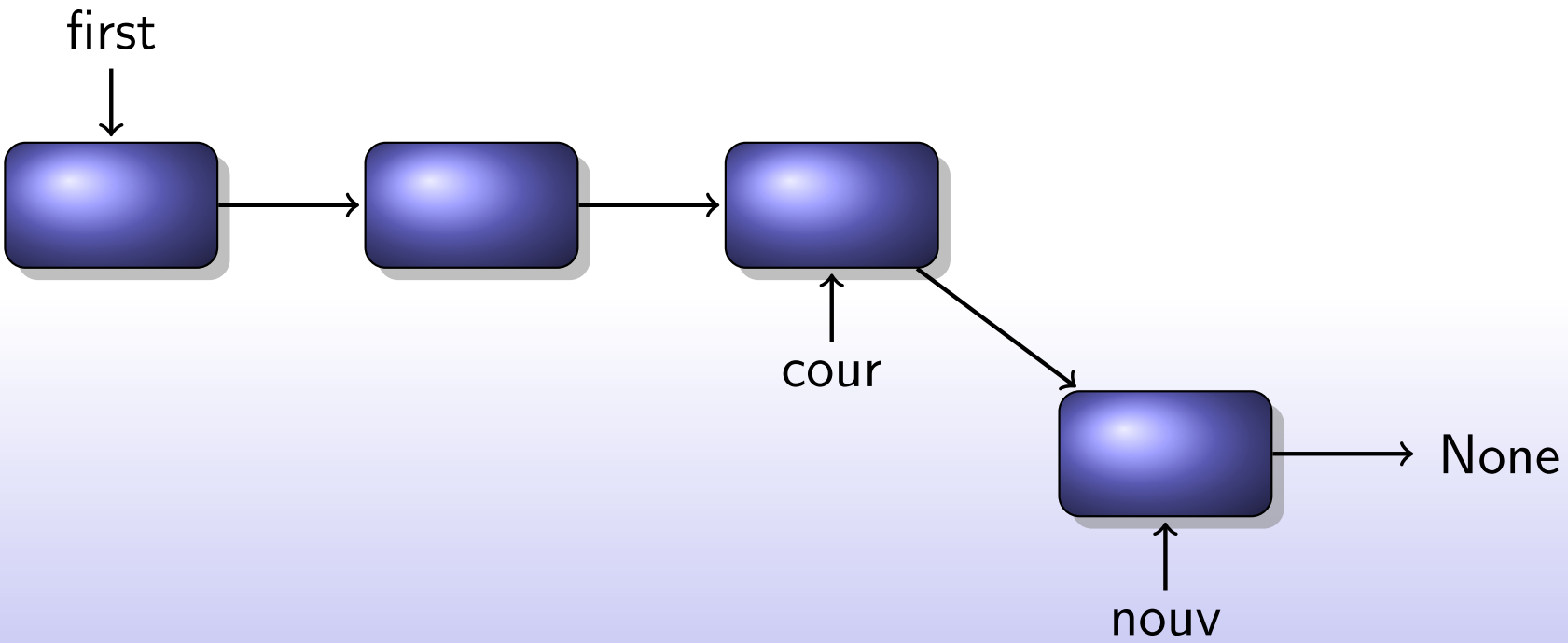
- Se placer en début de liste
- Avancer jusqu'au dernier élément

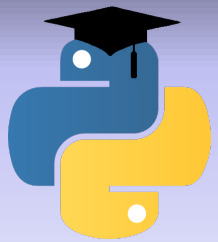


Insertion en fin de liste

Principe :

- Se placer en début de liste
- Avancer jusqu'au dernier élément
- Accrocher le nouvel élément après le dernier élément





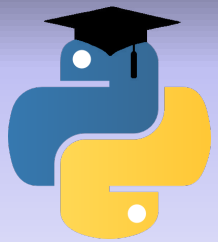
Insertion en fin de liste

Principe :

- Se placer en début de liste
- Avancer jusqu'au dernier élément
- Accrocher le nouvel élément après le dernier élément

Remarques

- Ne pas oublier le cas particulier où la liste est vide



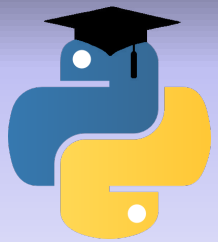
Insertion en fin de liste

Principe :

- Se placer en début de liste
- Avancer jusqu'au dernier élément
- Accrocher le nouvel élément après le dernier élément

Remarques

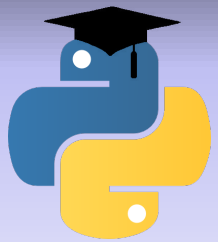
- Ne pas oublier le cas particulier où la liste est vide
- Complexité : $\Theta(n)$



Algorithme

Algorithme 4: Insertion en fin de liste

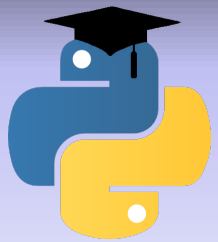
Entrées : nouv : noeud à ajouter
Données : debut : noeud `// tête de liste`
Données : n : noeud `// noeud courant`



Algorithme

Algorithme 5: Insertion en fin de liste

Entrées : nouv : noeud à ajouter
Données : debut : noeud `// tête de liste`
Données : n : noeud `// noeud courant`
si *liste* $= \emptyset$ `// cas particulier liste vide`
alors
  debut $\leftarrow$ *nouv* ;
sinon

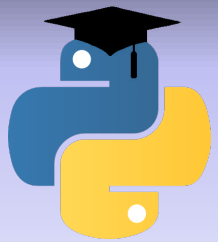


Algorithme

Algorithme 6: Insertion en fin de liste

Entrées : nouv : noeud à ajouter
Données : debut : noeud `// tête de liste`
Données : n : noeud `// noeud courant`
si *liste* $= \emptyset$ `// cas particulier liste vide`
alors
  debut $\leftarrow$ *nouv* ;
sinon

fin
$n \leftarrow debut$;
tant que $n.suivant \neq None$ **faire**
  $n \leftarrow n.suivant$;
fin



Algorithme

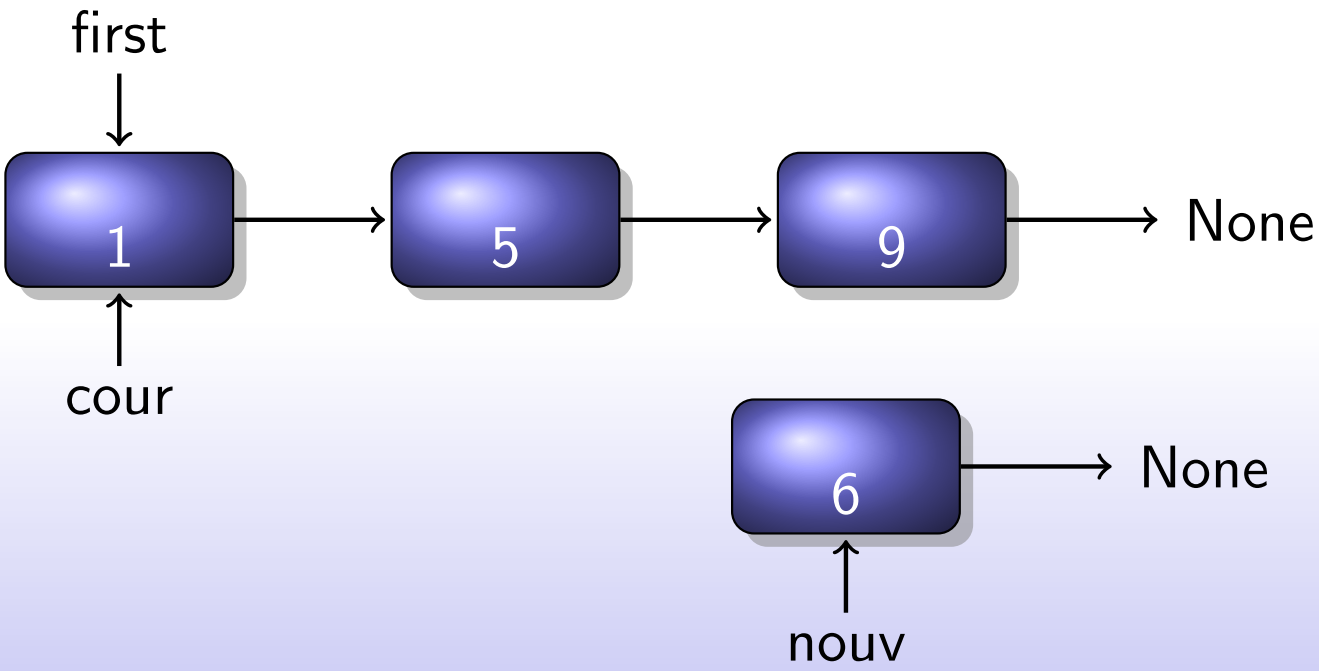
Algorithme 7: Insertion en fin de liste

Entrées : nouv : noeud à ajouter
Données : debut : noeud `// tête de liste`
Données : n : noeud `// noeud courant`
si *liste* $= \emptyset$ `// cas particulier liste vide`
alors
  debut $\leftarrow$ *nouv* ;
sinon
fin
$n \leftarrow debut$;
tant que $n.suivant \neq None$ **faire**
  $n \leftarrow n.suivant$;
fin
$n.suivant \leftarrow nouv$;

Insertion dans une liste triée

Principe $\simeq$ insertion en fin de liste :

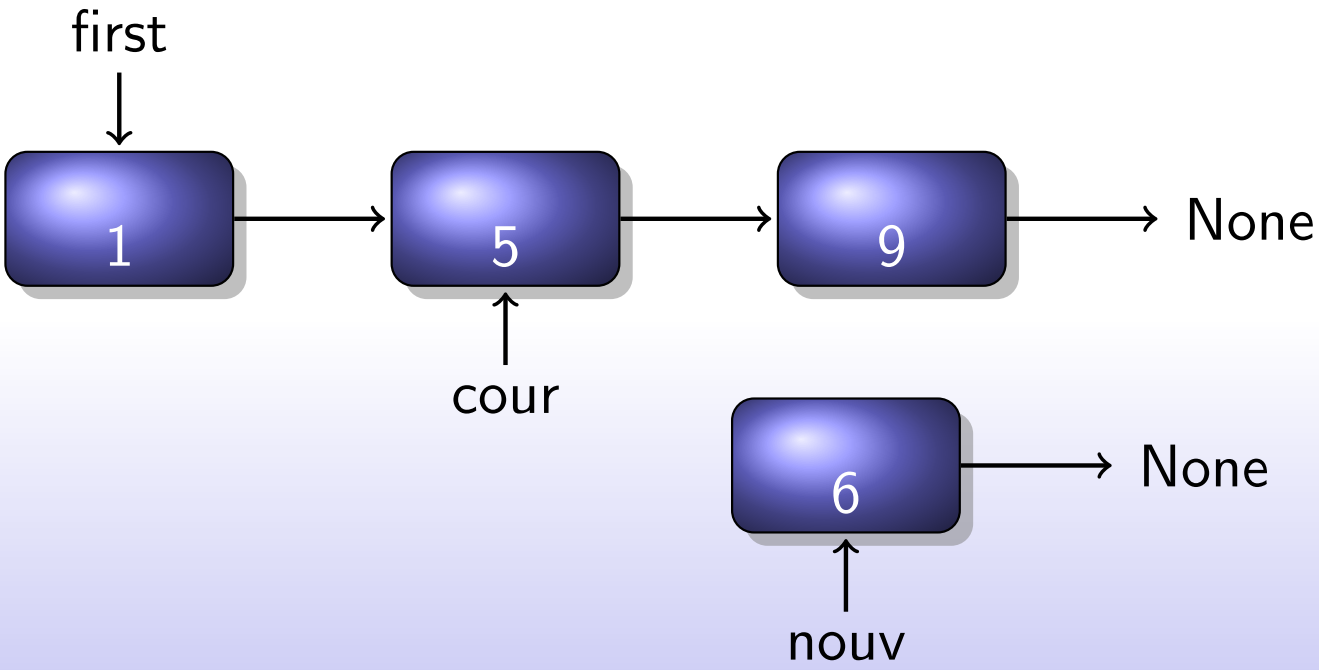
- Se placer en début de liste



Insertion dans une liste triée

Principe $\simeq$ insertion en fin de liste :

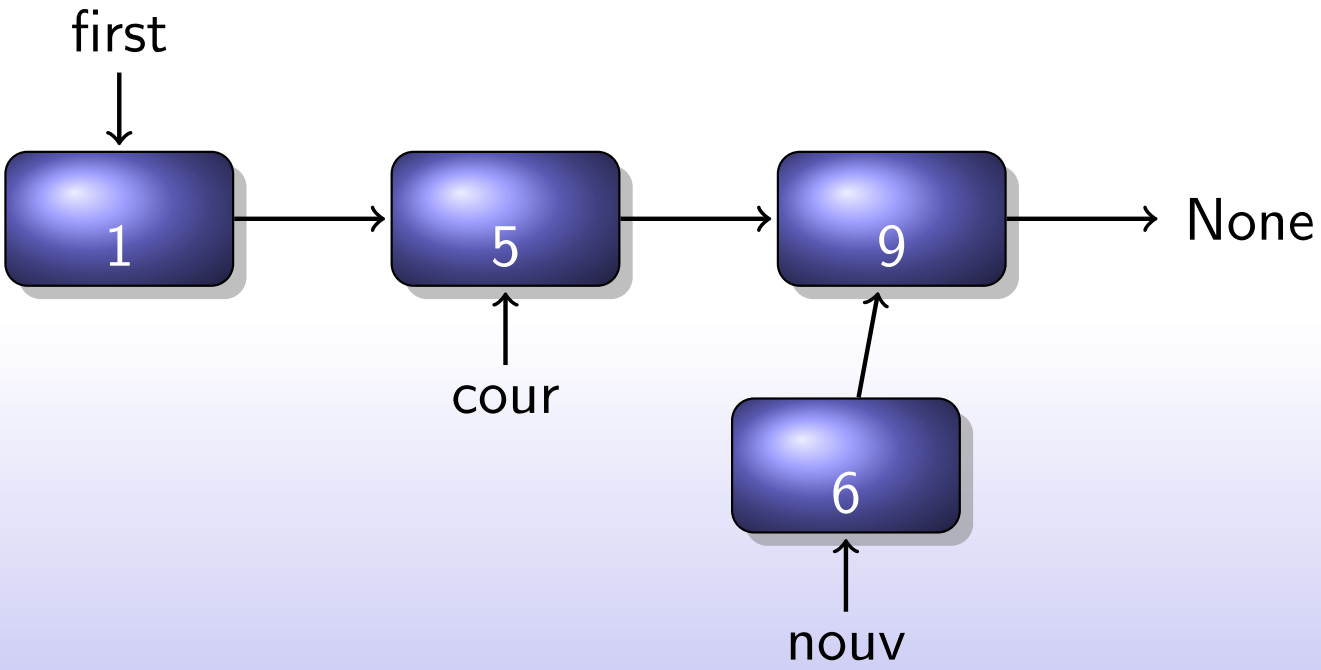
- Se placer en début de liste
- Avancer jusqu'au dernier élément inférieur à l'élément à insérer



Insertion dans une liste triée

Principe $\simeq$ insertion en fin de liste :

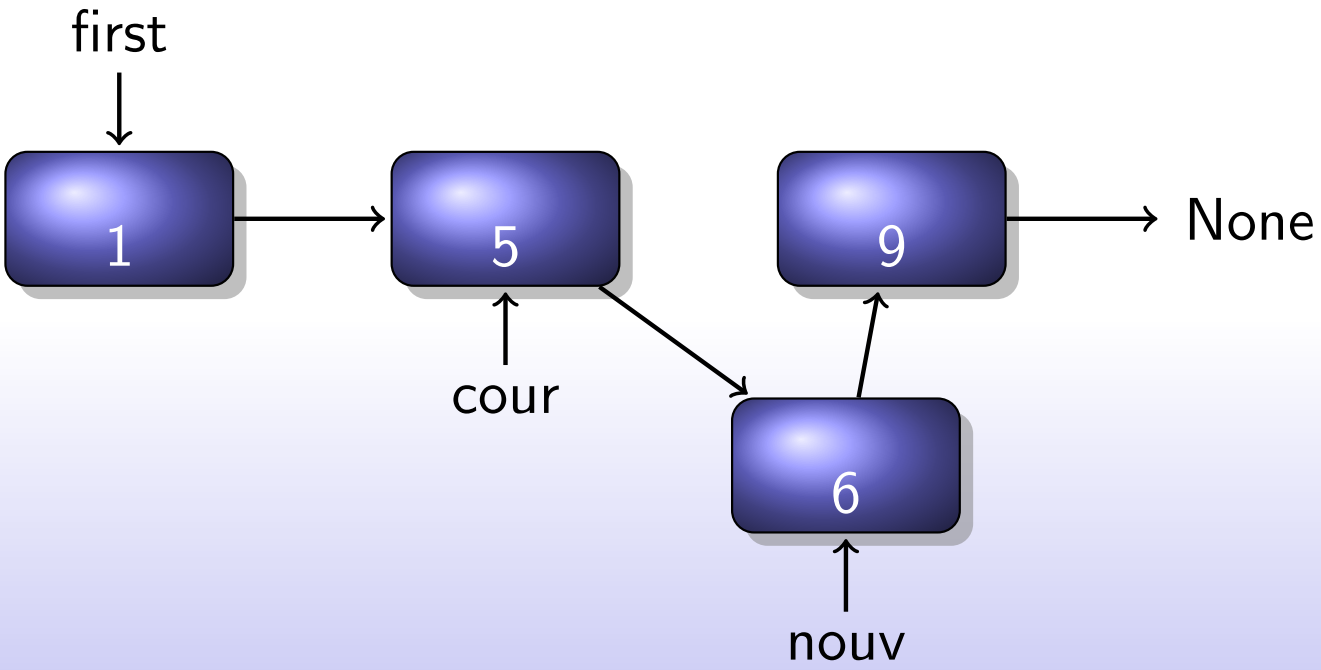
- Se placer en début de liste
- Avancer jusqu'au dernier élément inférieur à l'élément à insérer
- Accrocher le nouvel élément après l'élément courant

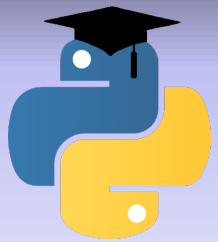


Insertion dans une liste triée

Principe $\simeq$ insertion en fin de liste :

- Se placer en début de liste
- Avancer jusqu'au dernier élément inférieur à l'élément à insérer
- Accrocher le nouvel élément après l'élément courant





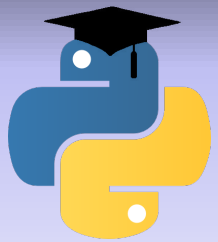
Insertion dans une liste triée

Principe $\simeq$ insertion en fin de liste :

- Se placer en début de liste
- Avancer jusqu'au dernier élément inférieur à l'élément à insérer
- Accrocher le nouvel élément après l'élément courant

Remarques

- Attention aux cas particuliers :



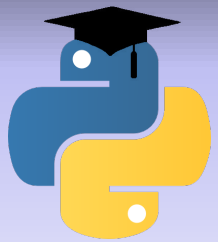
Insertion dans une liste triée

Principe $\simeq$ insertion en fin de liste :

- Se placer en début de liste
- Avancer jusqu'au dernier élément inférieur à l'élément à insérer
- Accrocher le nouvel élément après l'élément courant

Remarques

- Attention aux cas particuliers :
  - liste vide



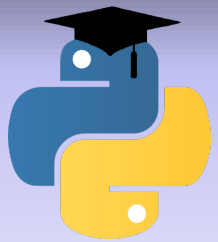
Insertion dans une liste triée

Principe $\simeq$ insertion en fin de liste :

- Se placer en début de liste
- Avancer jusqu'au dernier élément inférieur à l'élément à insérer
- Accrocher le nouvel élément après l'élément courant

Remarques

- Attention aux cas particuliers :
  - liste vide
  - insertion du plus petit élément



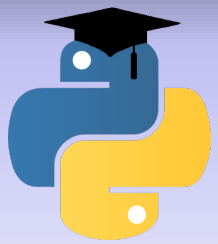
Insertion dans une liste triée

Principe $\simeq$ insertion en fin de liste :

- Se placer en début de liste
- Avancer jusqu'au dernier élément inférieur à l'élément à insérer
- Accrocher le nouvel élément après l'élément courant

Remarques

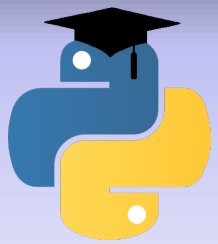
- Attention aux cas particuliers :
  - liste vide
  - insertion du plus petit élément
- Complexité : $\Theta(n)$



Remarque sur l'insertion dans une liste triée

Remarque

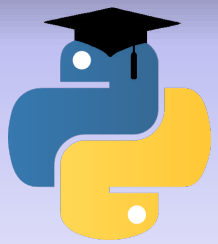
- Deux cas particuliers à tester



Remarque sur l'insertion dans une liste triée

Remarque

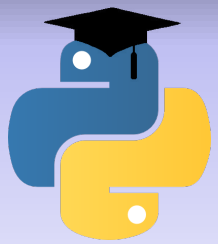
- Deux cas particuliers à tester
- Ruse :



Remarque sur l'insertion dans une liste triée

Remarque

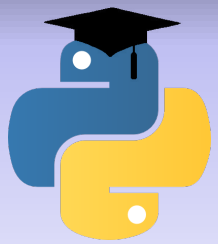
- Deux cas particuliers à tester
- Ruse :
  - créer un élément temporaire **tmp**



Remarque sur l'insertion dans une liste triée

Remarque

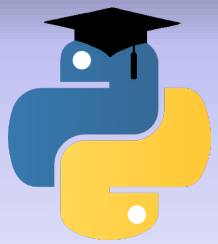
- Deux cas particuliers à tester
- Ruse :
  - créer un élément temporaire **tmp**
  - accrocher le début de la liste après **tmp**



Remarque sur l'insertion dans une liste triée

Remarque

- Deux cas particuliers à tester
- Ruse :
  - créer un élément temporaire **tmp**
  - accrocher le début de la liste après **tmp**
  - parcourir la liste à partir de **tmp**

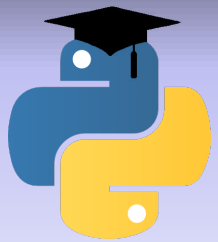


Remarque sur l'insertion dans une liste triée

Remarque

- Deux cas particuliers à tester
- Ruse :
  - créer un élément temporaire **tmp**
  - accrocher le début de la liste après **tmp**
  - parcourir la liste à partir de **tmp**

$\Rightarrow$ plus de cas particulier !



Algorithme

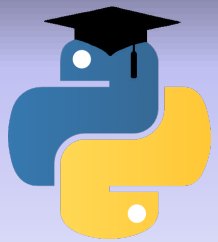
Algorithme 8: Insertion dans une liste triée

Entrées : nouv : noeud à ajouter

créer noeud **tmp** ;

$tmp.next \leftarrow debut$;

$n \leftarrow tmp$;



Algorithme

Algorithme 9: Insertion dans une liste triée

Entrées : nouv : noeud à ajouter

créer noeud **tmp** ;

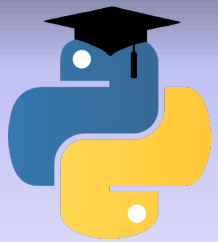
$tmp.next \leftarrow debut$;

$n \leftarrow tmp$;

tant que $n.suivant \neq None$ **et** $n.suivant < nouv$ **faire**

    $n \leftarrow n.suivant$;

fin



Algorithme

Algorithme 10: Insertion dans une liste triée

Entrées : nouv : noeud à ajouter

créer noeud **tmp** ;

$tmp.next \leftarrow debut$;

$n \leftarrow tmp$;

tant que $n.suivant \neq None$ **et** $n.suivant < nouv$ **faire**

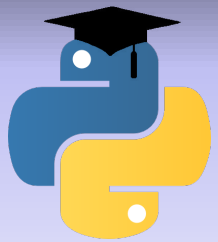
  $n \leftarrow n.suivant$;

fin

$nouv.suivant \leftarrow n.suivant$;

$n.suivant \leftarrow nouv$;

$debut \leftarrow tmp.suivant$ // supprimer l'élément temporaire

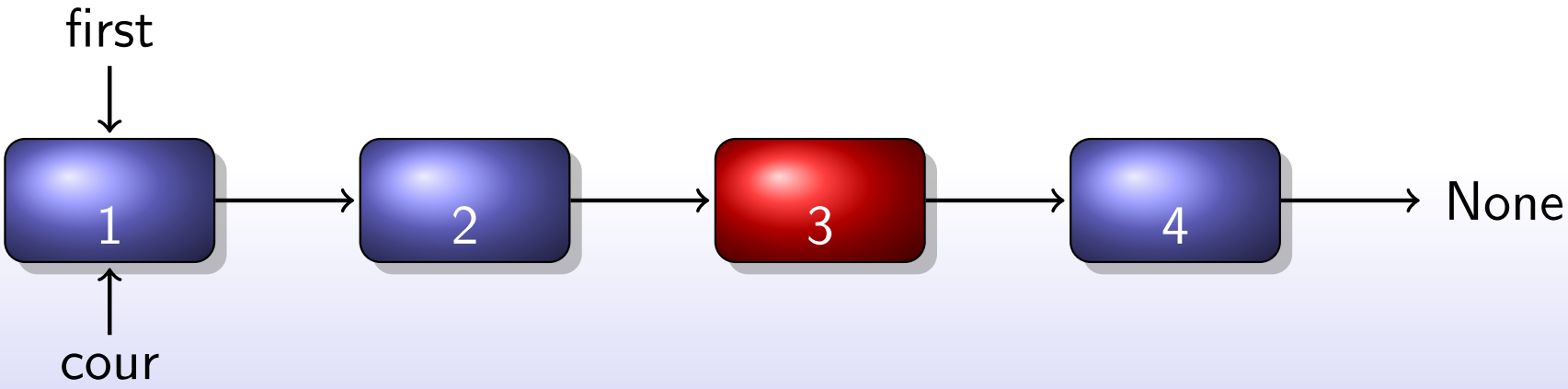


Sommaire

Suppression

Principe :

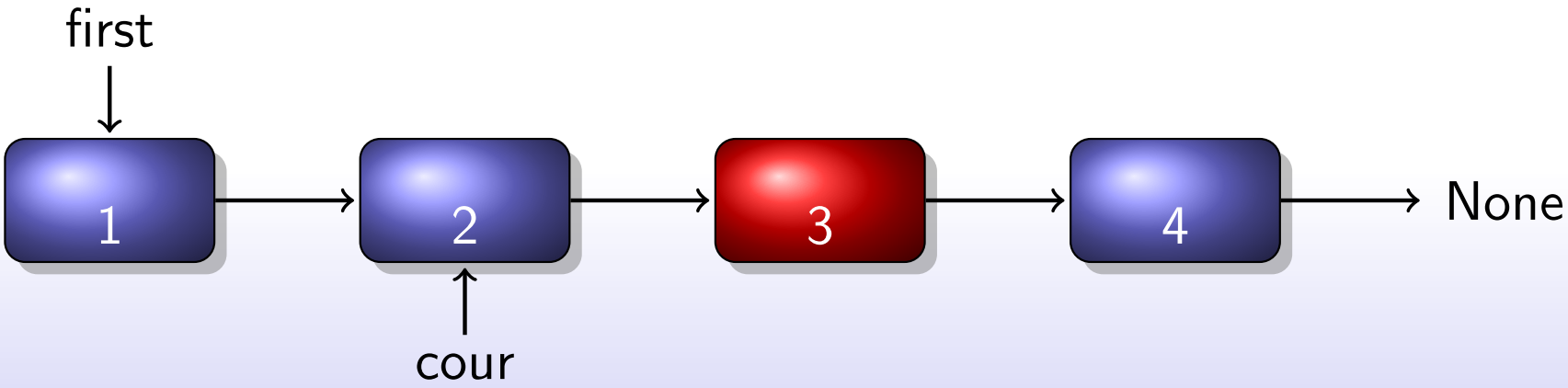
- Partir du début de la liste
- Se placer avant l'élément à supprimer
- Modifier le chaînage pour supprimer l'élément



Suppression

Principe :

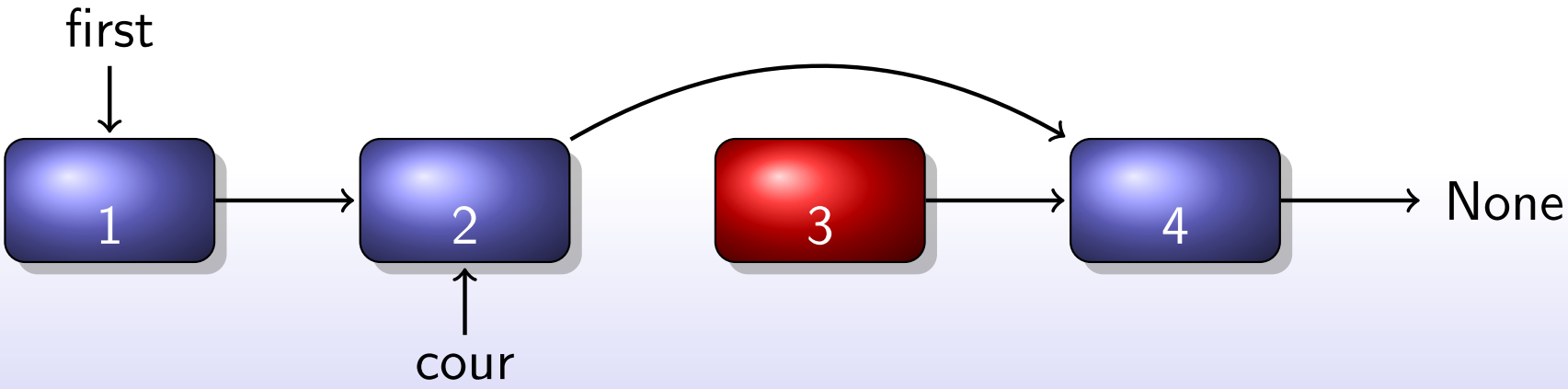
- Partir du début de la liste
- Se placer avant l'élément à supprimer
- Modifier le chaînage pour supprimer l'élément



Suppression

Principe :

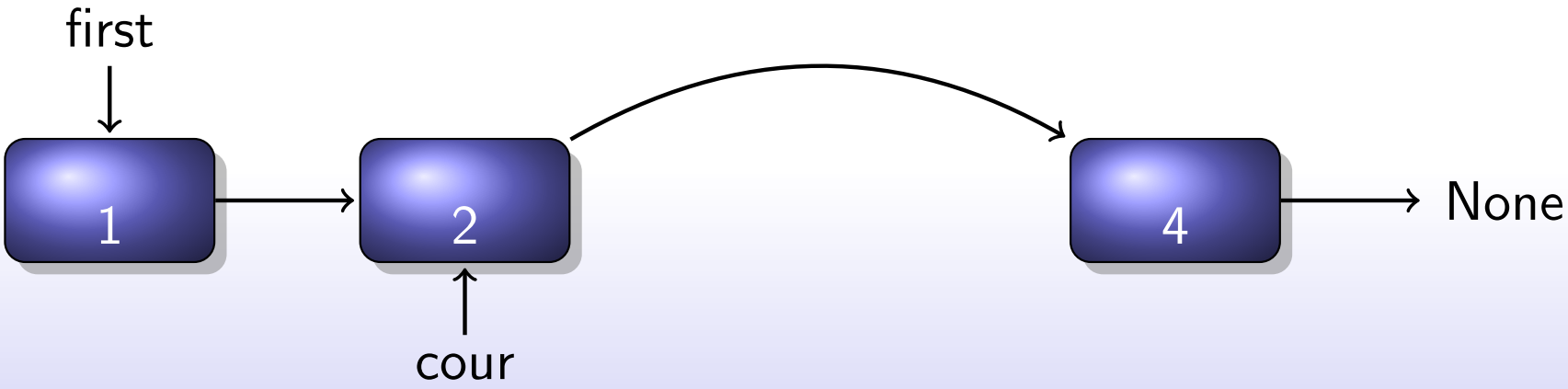
- Partir du début de la liste
- Se placer avant l'élément à supprimer
- Modifier le chaînage pour supprimer l'élément

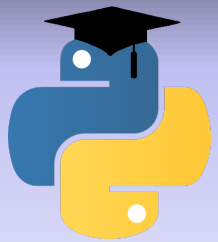


Suppression

Principe :

- Partir du début de la liste
- Se placer avant l'élément à supprimer
- Modifier le chaînage pour supprimer l'élément





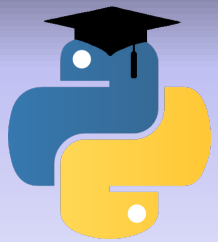
Suppression

Principe :

- Partir du début de la liste
- Se placer avant l'élément à supprimer
- Modifier le chaînage pour supprimer l'élément

Remarques

- Cas particuliers :



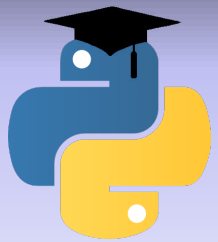
Suppression

Principe :

- Partir du début de la liste
- Se placer avant l'élément à supprimer
- Modifier le chaînage pour supprimer l'élément

Remarques

- Cas particuliers :
  - liste vide



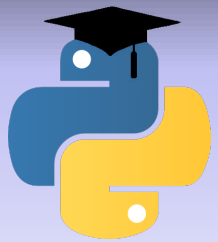
Suppression

Principe :

- Partir du début de la liste
- Se placer avant l'élément à supprimer
- Modifier le chaînage pour supprimer l'élément

Remarques

- Cas particuliers :
  - liste vide
  - suppression du premier élément



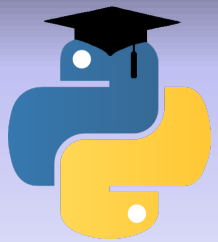
Suppression

Principe :

- Partir du début de la liste
- Se placer avant l'élément à supprimer
- Modifier le chaînage pour supprimer l'élément

Remarques

- Cas particuliers :
  - liste vide
  - suppression du premier élément
- Complexité : $\Theta(n)$



Algorithme

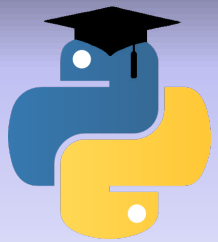
Algorithme 11: Suppression d'un élément dans une liste

Entrées : val : entier `// val : valeur du noeud à supprimer`

créer noeud **tmp** ;

$tmp.next \leftarrow debut$;

$n \leftarrow tmp$;



Algorithme

Algorithme 12: Suppression d'un élément dans une liste

Entrées : val : entier `// val : valeur du noeud à supprimer`

créer noeud **tmp** ;

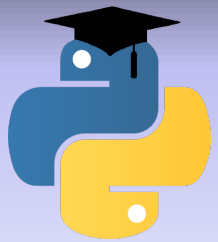
$tmp.next \leftarrow debut$;

$n \leftarrow tmp$;

tant que $n.suivant \neq None$ **et** $n.suivant.val \neq val$ **faire**

  $n \leftarrow n.suivant$;

fin



Algorithme

Algorithme 13: Suppression d'un élément dans une liste

Entrées : val : entier `// val : valeur du noeud à supprimer`

créer noeud **tmp** ;

$tmp.next \leftarrow debut$;

$n \leftarrow tmp$;

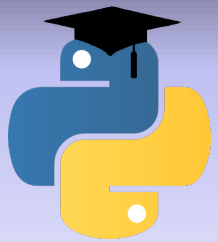
tant que $n.suivant \neq None$ **et** $n.suivant.val \neq val$ **faire**

  $n \leftarrow n.suivant$;

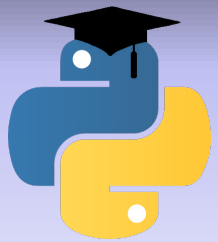
fin

$n.suivant \leftarrow n.suivant.suivant$;

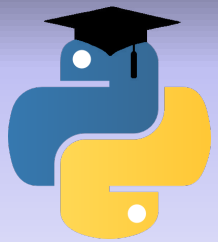
$debut \leftarrow tmp.suivant$;



Sommaire



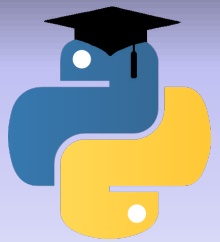
Sommaire



Principe

- Chaque noeud connaît son successeur
- et son prédécesseur

$\Rightarrow$ variable d'instance supplémentaire



Principe

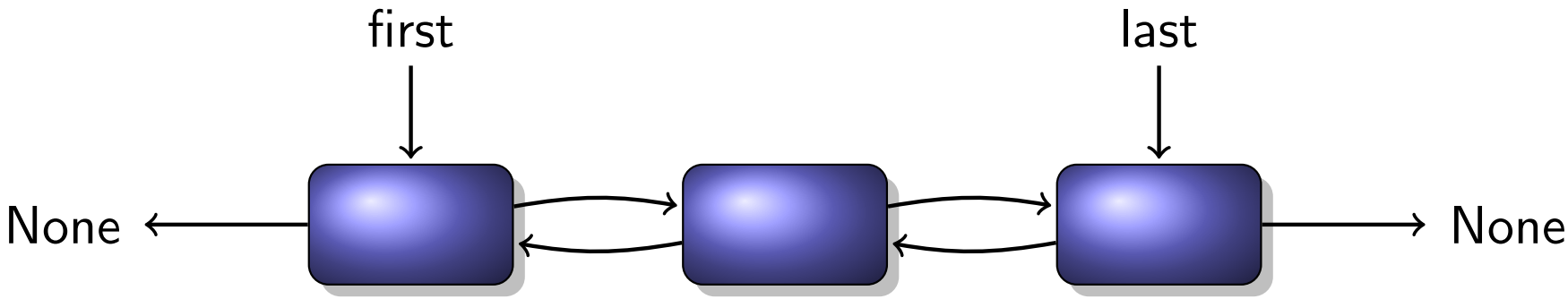
- Chaque noeud connaît son successeur
- et son prédécesseur

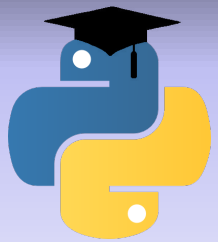
$\Rightarrow$ variable d'instance supplémentaire

Attention

Bien gérer la cohérence du double chaînage

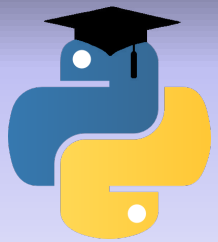
Liste doublement chaînée





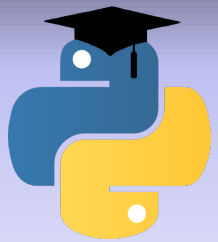
Classes Node et Liste

```
class Node (object):
   def __init__(self, donnee = 0):
      self.val = donnee # La valeur de noeud
      self.next = None # Le noeud suivant

   ...
```



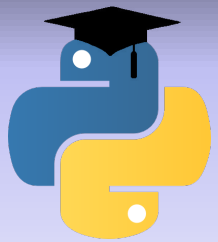
Classes Node et Liste

```
class Node (object):
   def __init__(self, donnee = 0):
      self.val = donnee # La valeur de noeud
      self.next = None # Le noeud suivant

   ...
```



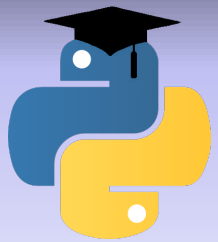
Classes Node et Liste

```
class Node (object):
   def __init__(self, donnee = 0):
      self.val = donnee # La valeur de noeud
      self.next = None # Le noeud suivant
      self.prev = None # Le noeud précédent
   ...
```



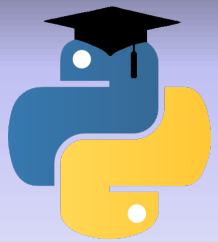
Classes Node et Liste

```
class Node (object):
   def __init__(self, donnee = 0):
      self.val = donnee # La valeur de noeud
      self.next = None # Le noeud suivant
      self.prev = None # Le noeud précédent
   ...
class List(object):
   def __init__(self):
      self.__first = None
```

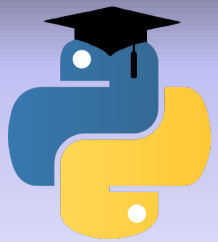


Classes Node et Liste

```
class Node (object):
   def __init__(self, donnee = 0):
      self.val = donnee # La valeur de noeud
      self.next = None # Le noeud suivant
      self.prev = None # Le noeud précédent
   ...
class List(object):
   def __init__(self):
      self.__first = None
      self.__last = None
```



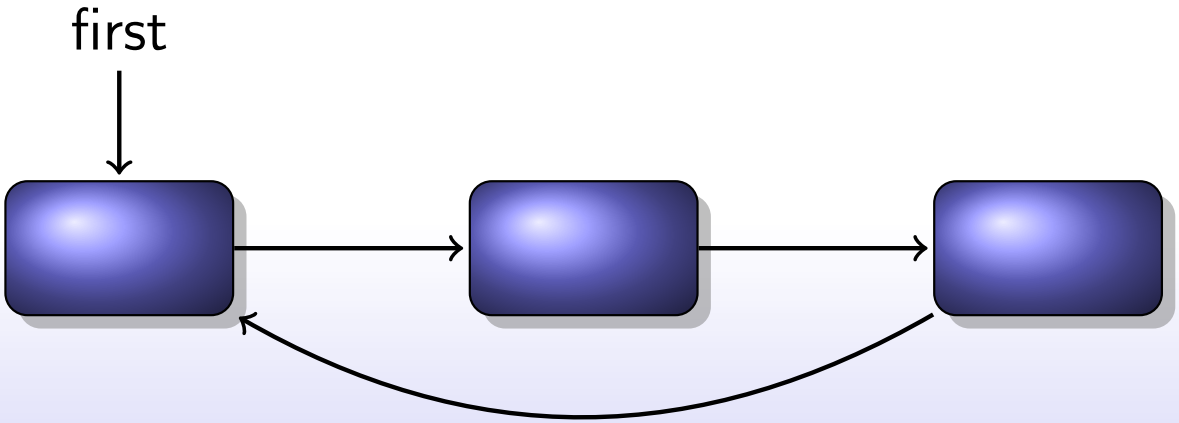
Sommaire

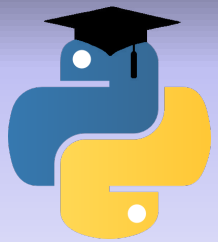


Principe

- Le successeur d'un noeud peut être un noeud précédent
- Attention lors des parcours

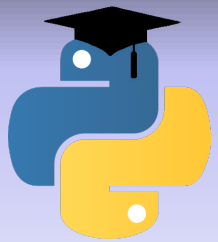
$\Rightarrow$ risque de boucle infinie





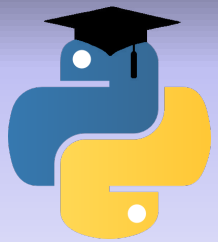
Détection de cycle

- Problème : comment détecter que la liste contient un cycle ?



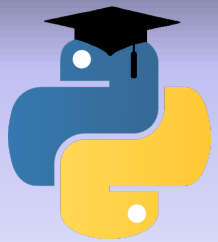
Détection de cycle

- Problème : comment détecter que la liste contient un cycle ?
- Solution :



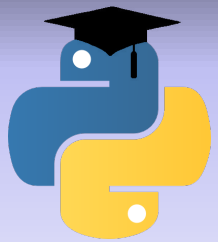
Détection de cycle

- Problème : comment détecter que la liste contient un cycle ?
- Solution :
  - utiliser deux variables :



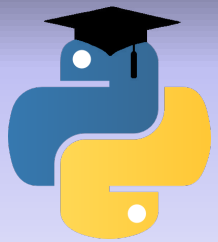
Détection de cycle

- Problème : comment détecter que la liste contient un cycle ?
- Solution :
  - utiliser deux variables :
  - une qui avance d'un noeud par tour



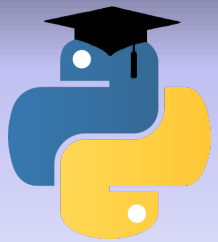
Détection de cycle

- Problème : comment détecter que la liste contient un cycle ?
- Solution :
  - utiliser deux variables :
  - une qui avance d'un noeud par tour
  - une qui avance de deux noeuds par tour



Détection de cycle

- Problème : comment détecter que la liste contient un cycle ?
- Solution :
  - utiliser deux variables :
  - une qui avance d'un noeud par tour
  - une qui avance de deux noeuds par tour
  - si un aboutit à la fin de la liste : liste non circulaire



Détection de cycle

- Problème : comment détecter que la liste contient un cycle ?
- Solution :
  - utiliser deux variables :
  - une qui avance d'un noeud par tour
  - une qui avance de deux noeuds par tour
  - si un aboutit à la fin de la liste : liste non circulaire
  - si les deux se rejoignent : liste circulaire