DIU Enseigner l'informatique au lycée

Notes de cours

Kim Nguyễn

kim.nguyen@lri.fr

Listes chaînées

Notions introduites

- Structure de liste chaînée
- Opérations élémentaires sur les listes
- Notion de partage en mémoire
- Interface objet

Rappel : la mémoire

- la mémoire d'un ordinateur est un grand tableau linéaire d'octets
- l'indice d'une case du tableau est appelé une adresse mémoire
- le processeur peut accéder à des adresses (écrire et lire les octets qui s'y trouvent)

(plein d'autres subtilités qu'on ignore ici, cf. le cours d'architecture des machines)

Comment représenter une collection d'objets ?

On veut stocker une collection d'entiers 64 bits : 42, 43, 31, 2.

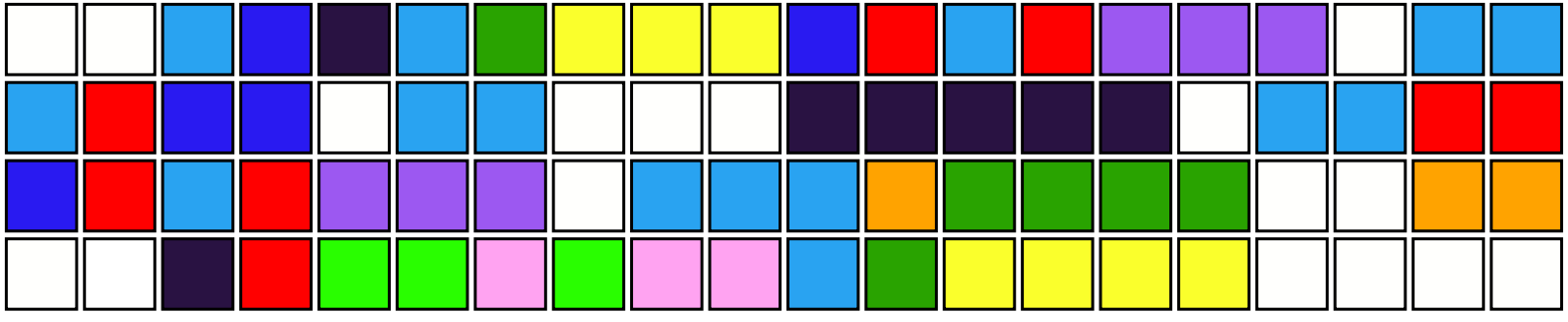
- On choisit une adresse (par ex. 100)
- On écrit les octets correspondants à partir de cette adresse

...	42	43	31	2	...
	100	108	116	124	

Oui mais . . .

La mémoire d'un ordinateur stocke **tout** ce dont l'ordinateur à besoin pour fonctionner :

- Les données en cours de manipulation (contenu de fichiers, résultats de calculs, . . .)
- Les programmes
  - Les programmes utilisateurs
  - Le système d'exploitation



Allocation mémoire

- Le système d'exploitation gère la mémoire de tout l'ordinateur
- Les programmes demandent au système une quantité de mémoire (par ex : 32 octets)
- Le système donne une adresse mémoire vers une zone de la taille demandée
- Le système peut aussi lever une erreur s'il n'y a plus de mémoire disponible
- Le programme libère la zone mémoire lorsqu'il n'en a plus besoin

Retour sur les collections d'objets

Que faire si on ne connaît pas la taille a priori ?

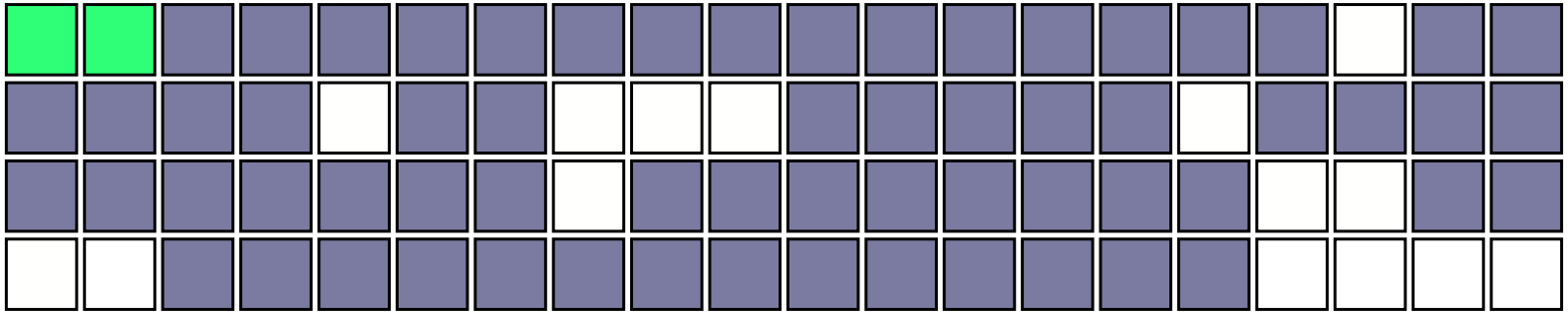
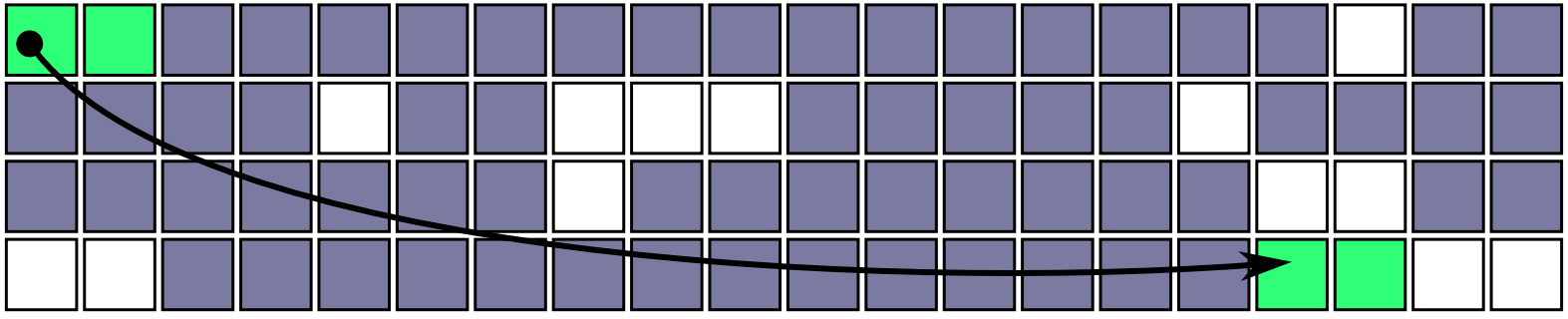


Tableau de deux cases

Retour sur les collections d'objets

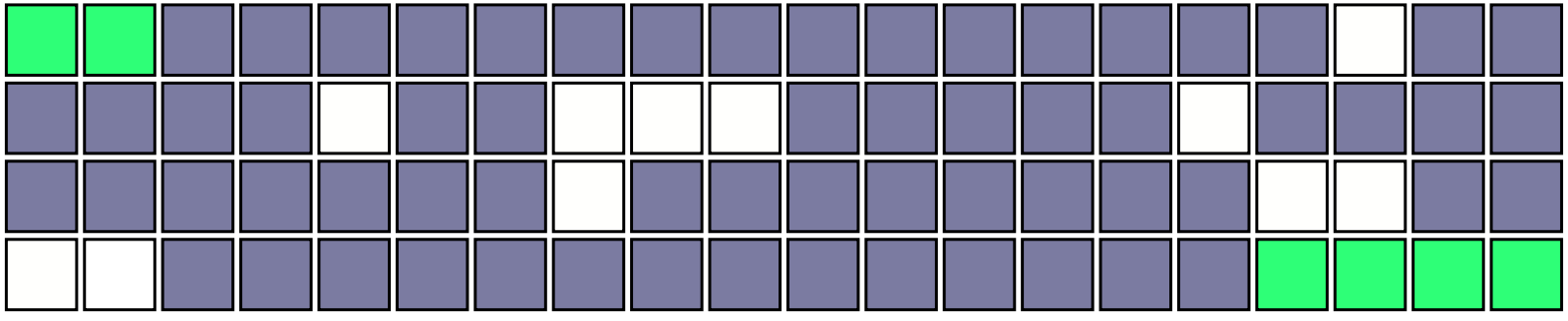
Que faire si on ne connaît pas la taille a priori ?



Allocation d'une zone de 4 et copie des deux cases existantes

Retour sur les collections d'objets

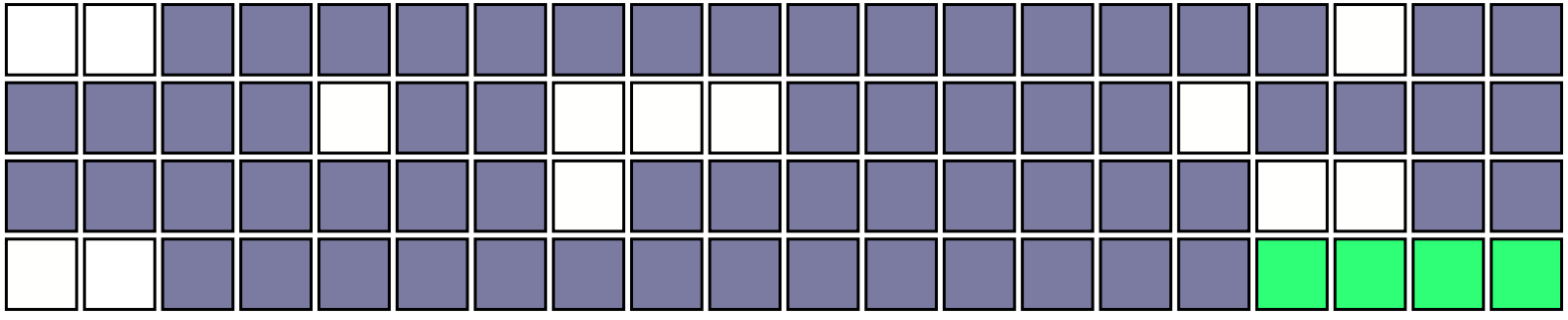
Que faire si on ne connaît pas la taille a priori ?



Ajout de nouvelles valeurs dans l'espace libre

Retour sur les collections d'objets

Que faire si on ne connaît pas la taille a priori ?



Libération de l'ancien tableau

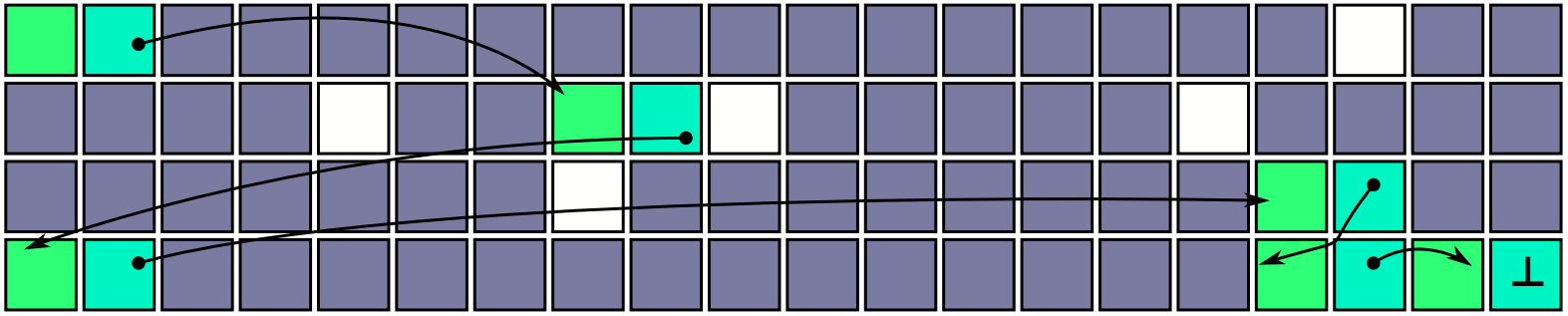
Tableaux redimensionnables

- On alloue une zone mémoire d'une certaine taille initiale
- On remplit cette zone
- Lorsqu'elle est pleine, on demande au système un zone plus grande
- On **recopie** l'ancienne zone dans la nouvelle
- On libère l'ancienne zone

Les tableaux redimensionables de python (type `list`) fonctionnent exactement comme ça.

Est-ce qu'on peut faire autrement ?

On peut « chaîner » les zones mémoires :



- Pour chaque élément de la collection on alloue un mot pour l'élément et un pointeur
- Le pointeur est l'adresse de la zone contenant l'élément suivant
- Pour le dernier élément on utilise un pointeur spécial, $\perp$ qui représente une adresse invalide

Avantages et inconvénients de l'approche

+ Pas besoin de connaître a priori le nombre d'éléments
+ Pas besoin de disposer d'une zone mémoire contiguë
- On ne peut plus retrouver un élément par décalage

Structure particulièrement adaptée lorsque la recopie est prohibitive (par exemple système de fichiers du disque dur).

Mais en pratique, en terminale ?

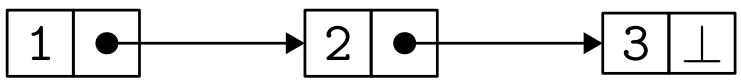
- \- Les tableaux Python font parfaitement l'affaire dans 99.999% des cas
- \+ Pédagogiquement : permet de préparer les structures arborescentes
- ! Expose encore plus les choix mutable/non-mutable que les tableaux

Listes chaînées

Une liste chaînée est une structure de données composée de *cellules* (ou maillons). Chaque cellule contient

- une donnée
- une référence vers la cellule suivante

La liste 1, 2, 3 peut être représentée graphiquement comme ceci :



Listes chaînées en Python

```
class Cellule:
  """une cellule d'une liste chaînée"""

  def __init__(self,v , s):
      self.valeur = v
      self.suivante = s

lst = Cellule(1, Cellule(2, Cellule(3, None)))
```

On représente la fin de liste, $\perp$ par la valeur Python `None`.

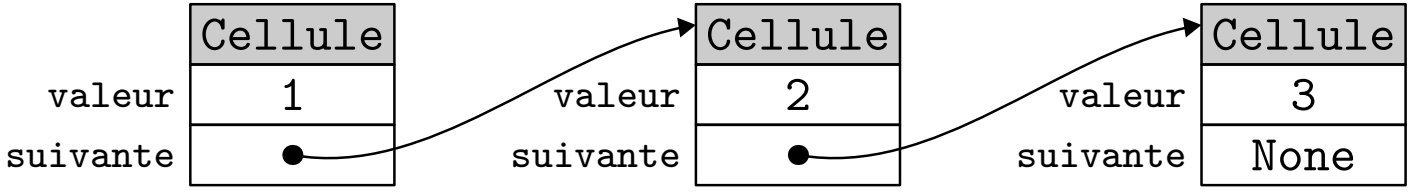
Listes chaînées en Python

```
class Cellule:
    """une cellule d'une liste chaînée"""

    def __init__(self,v , s):
        self.valeur = v
        self.suivante = s

lst = Cellule(1, Cellule(2, Cellule(3, None)))
```

On représente la fin de liste, $\perp$ par la valeur Python `None`.



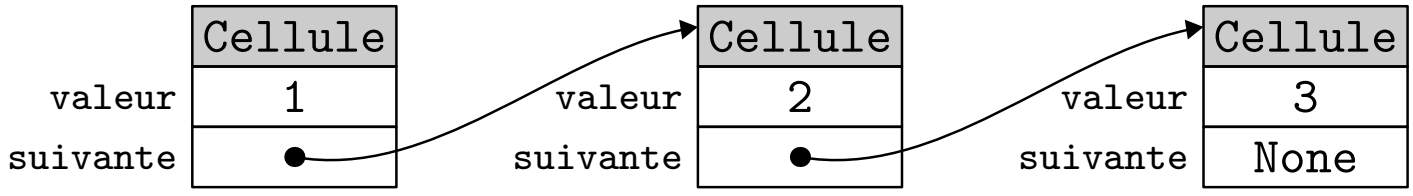
Listes chaînées en Python

```
class Cellule:
    """une cellule d'une liste chaînée"""

    def __init__(self,v , s):
        self.valeur = v
        self.suivante = s

lst = Cellule(1, Cellule(2, Cellule(3, None)))
```

On représente la fin de liste, $\perp$ par la valeur Python `None`.




```
print(lst.valeur)  #1
print(lst.suivante.valeur) #2
print(lst.suivante.suivante.valeur) #3
```

Remarques

- On aurait pu utiliser d'autres représentations :
  - Paires : `(1,(2,(3, None)))`
  - Tableaux : `[1,[2,[3, None]]]`

  L'utilisation d'objets est idiomatique en Python.
- Le type des listes chaînées est un type *concret*
- C'est un type *récursif*. Une liste chaînée est soit :
  - Une liste vide (`None`)
  - Une cellule contenant une valeur et une liste chaînée

Opération sur les listes

On présente quelques opérations élémentaires sur les listes. On propose, lorsque c'est judicieux des versions récursives et itératives des algorithmes.

Calcul de la longueur d'une liste chaînée (rec)

Calcul de la longueur d'une liste chaînée (rec)

```
def longueur (l):
    """Calcul de la longueur d'une liste"""
    if l is None:
        return 0
    else:
        return 1 + longueur(l.suivante)
```

Schéma de récursion sur une liste chaînée

Le type liste est récursif :

- Soit une liste vide (`None`)
- Soit une cellule contenant une valeur et une liste chaînée

Parcours récursif d'une liste chaînée :

```
def f (l):
  if l is None:
    # gérer le cas de base
    ...
  else:
    # utiliser l.valeur
    # effectuer un appel récursif sur l.suivante
    ...
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
  = 1 + longueur (lst.suivante)
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
  = 1 + longueur (lst.suivante)
  = 1 + 1 + longueur (lst.suivante.suivante)
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
  = 1 + longueur (lst.suivante)
  = 1 + 1 + longueur (lst.suivante.suivante)
  = 1 + 1 + 1 + longueur (lst.suivante.suivante.suivante
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
  = 1 + longueur (lst.suivante)
  = 1 + 1 + longueur (lst.suivante.suivante)
  = 1 + 1 + 1 + longueur (lst.suivante.suivante.suivante
  = 1 + 1 + 1 + longueur (None)
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
  = 1 + longueur (lst.suivante)
  = 1 + 1 + longueur (lst.suivante.suivante)
  = 1 + 1 + 1 + longueur (lst.suivante.suivante.suivante
  = 1 + 1 + 1 + longueur (None)
  = 1 + 1 + 1 + 0
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
  = 1 + longueur (lst.suivante)
  = 1 + 1 + longueur (lst.suivante.suivante)
  = 1 + 1 + 1 + longueur (lst.suivante.suivante.suivante
  = 1 + 1 + 1 + longueur (None)
  = 1 + 1 + 1 + 0
  = 1 + 1 + 1
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
  = 1 + longueur (lst.suivante)
  = 1 + 1 + longueur (lst.suivante.suivante)
  = 1 + 1 + 1 + longueur (lst.suivante.suivante.suivante
  = 1 + 1 + 1 + longueur (None)
  = 1 + 1 + 1 + 0
  = 1 + 1 + 1
  = 1 + 2
```

Illustration du calcul récursif de la longueur

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
t = longueur (lst)
  = 1 + longueur (lst.suivante)
  = 1 + 1 + longueur (lst.suivante.suivante)
  = 1 + 1 + 1 + longueur (lst.suivante.suivante.suivante
  = 1 + 1 + 1 + longueur (None)
  = 1 + 1 + 1 + 0
  = 1 + 1 + 1
  = 1 + 2
  = 3
```

Calcul de la longueur d'une liste chaînée (it)

Calcul de la longueur d'une liste chaînée (it)

```
def longueur (l):
    """Calcul de la longueur d'une liste"""
    res = 0
    c = l
    while c is not None:
        res = res + 1
        c = c.suivante
    return res
```

Schéma d'itération sur une liste chaînée

```
def f (l):
  c = l   # référence vers la cellule courante
  while c is not None:
    # faire quelque chose avec c.valeur
    ...
    c = c.suivante
  #Fin de boucle
  ...
```

Autres opérations simples

(à faire en exercice)

- Ajouter/Supprimer un élément en début de liste
- Récupérer le $n^{ème}$ élément d'une liste

Concaténation de listes

Opération qui consiste à mettre bout à bout deux listes (opération + des tableaux en Python). Code récursif :

Concaténation de listes

Opération qui consiste à mettre bout à bout deux listes (opération + des tableaux en Python). Code récursif :

```
def concatener (l1, l2):
    """Concatène les listes l1 et l2 et renvoie
       une nouvelle liste"""
    if l1 is None:
        return l2
    else:
        return Cellule(l1.valeur, concatener(l1.suivant, l2))
```

Remarque : aucune des deux listes n'est modifiée

Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

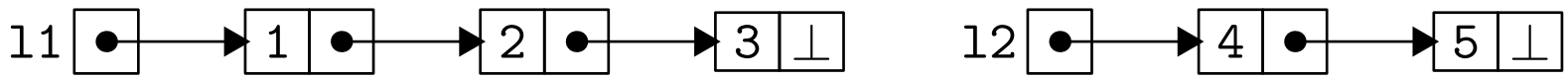


Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

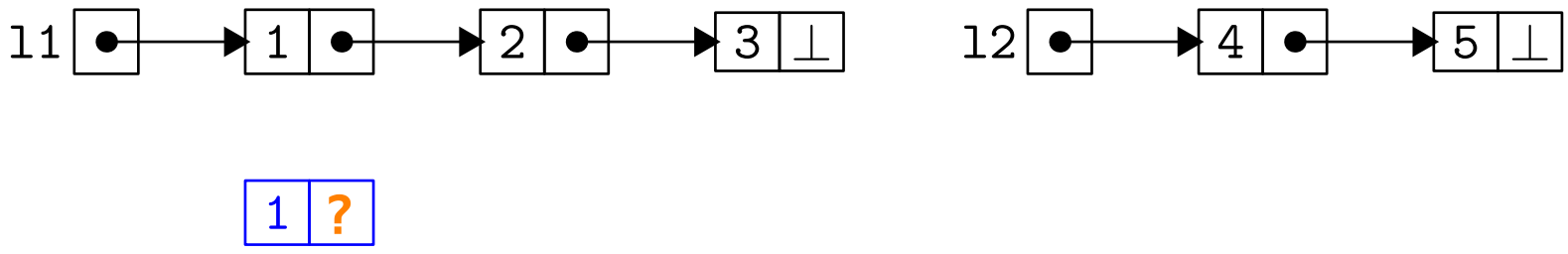


Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

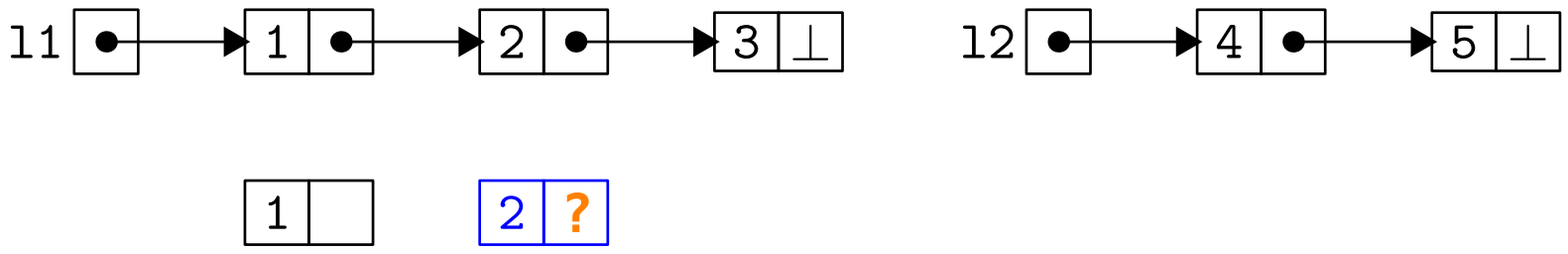


Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

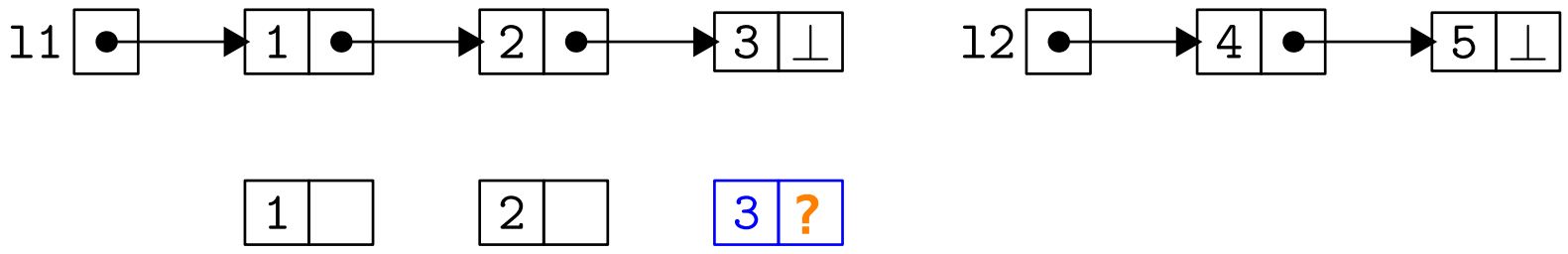


Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

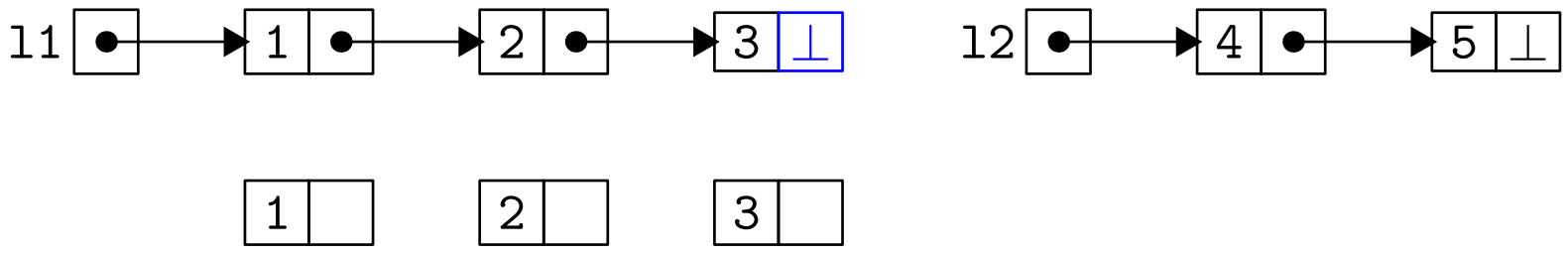


Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

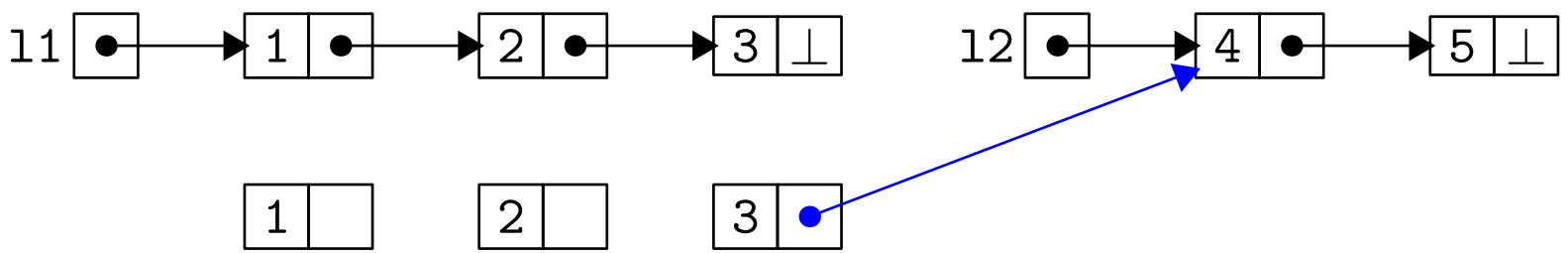


Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

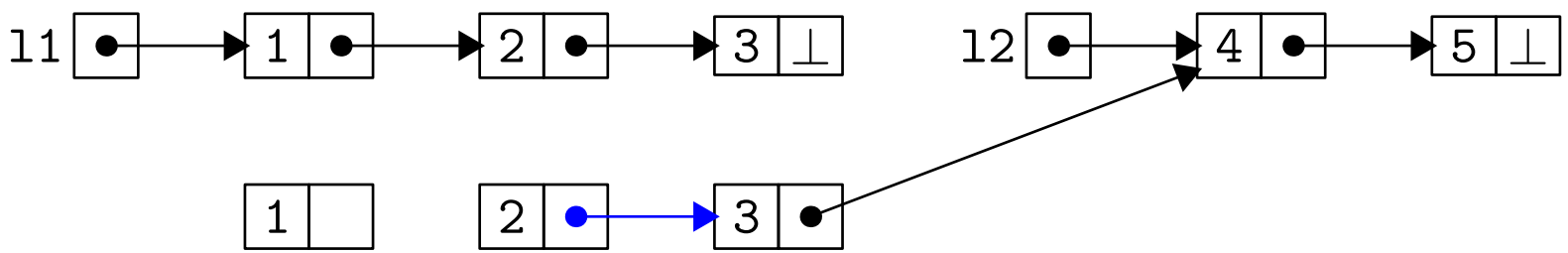


Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

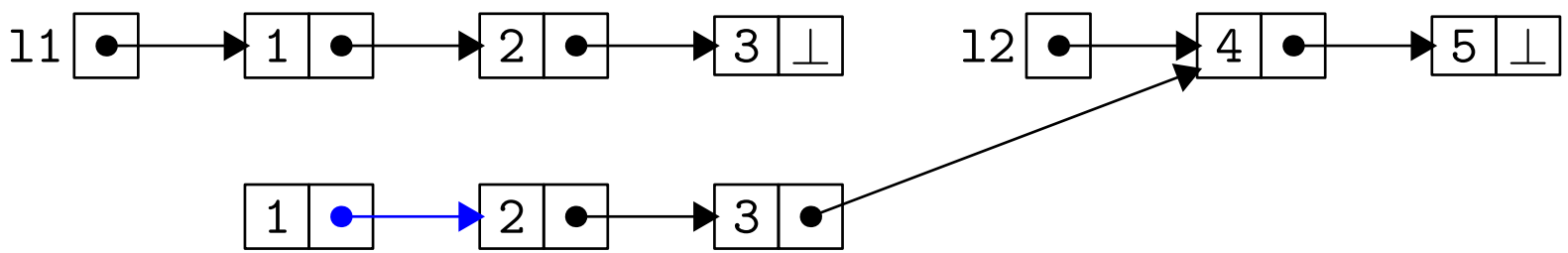
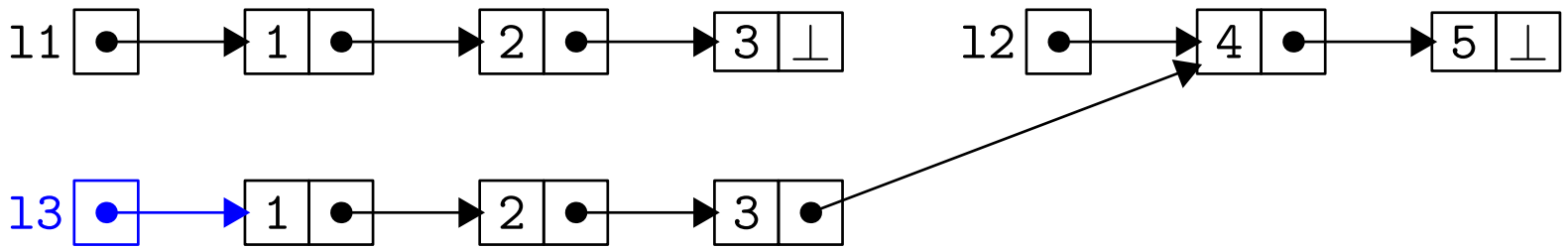


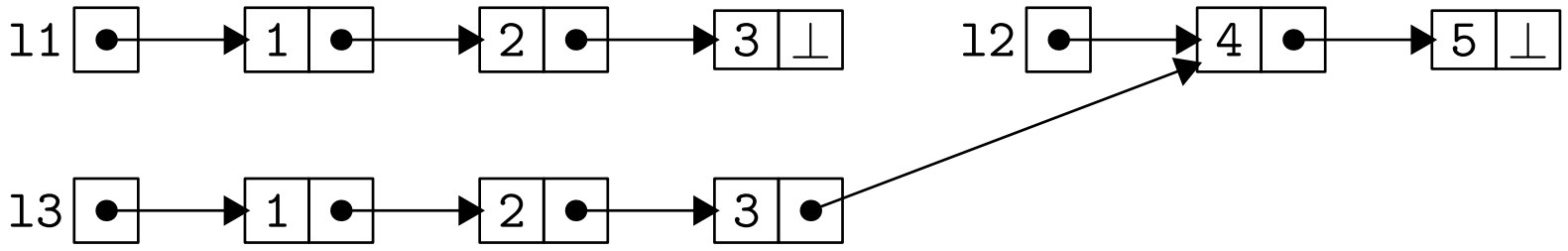
Illustration de la concatenation

```
def concatener (l1, l2):
  if l1 is None:
    return l2
  else:
    return Cellule(l1.valeur, concatener(l1.suivante, l2))

l1 = Cellule(1,Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```



Partage en mémoire



- On a parcouru `l1`
- On crée au fur et à mesure **une copie** de `l1`
- On chaîne la fin de la copie à `l2`

C'est la bonne façon^{TM} (la plus sûre) de faire la concaténation !
Exercice : donner une version itérative de cet algorithme.

Modifications en place, danger

Rien n'interdit en Python de modifier une liste en place

```
lst = Cellule(1,Cellule(2, Cellule(3, None)))
lst.suivante.valeur = 4
```

Ce phénomène est le même que la modification d'une case d'un tableau Python :

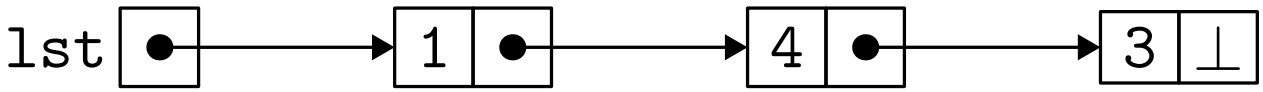
```
tab = [1, 2, 3]
tab[1] = 4
```

Comme pour les tableaux, ces modifications peuvent être dangereuses, par exemple si elles sont faites dans le corps d'une fonction à l'insu de l'appelant.

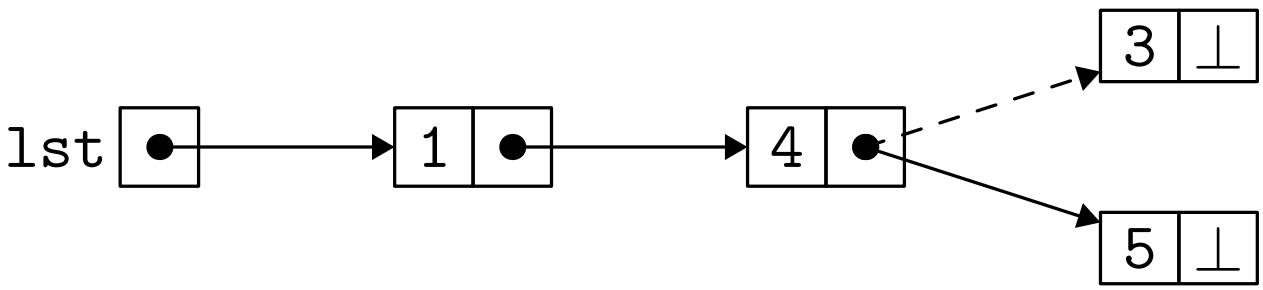
Modifications en place, danger (suite)

On peut non seulement modifier le contenu de la liste, mais aussi sa structure :

```
lst = Cellule(1,Cellule(2, Cellule(3, None)))
lst.suivante.valeur = 4
```




```
lst.suivante.suivante = Cellule(5, None)
```



(cf Exercice sur la concaténation en place)

Modifications en place, danger (fin)

- La modification de données potentiellement partagée est toujours dangereuse
- On conseille de privilégier un style de programmation où on ne modifie pas en place la structure des données

Encapsulation dans un objet

On souhaite donner une interface objet au type de liste chaînées.
En effet, la classe `Cellule` ne remplit pas ce rôle car la liste vide, représentée par `None`, n'est pas du type `Cellule`.

Encapsulation dans un objet (constructeur)

On crée une classe `Liste` contenant la liste chaînée telle que définie précédemment.

```
class Liste:
"""une liste chaînée"""

  def __init__(self):
    self.tete = None
```

La classe possède un seul attribut, `tete` qui contient la liste chaînée (i.e. soit une cellule, soit `None`). Le constructeur crée une liste initialement vide.

Encapsulation dans un objet (méthodes)

On applique maintenant les principes d'encapsulation et de programmation orientée objet :

- On cache à l'utilisateur les choix d'implémentation
- On propose un ensemble de méthodes regroupant les opérations de base

Encapsulation dans un objet (méthodes)

On crée une classe `Liste` contenant la liste chaînée telle que définie précédemment.

```
...
  def est_vide(self):
  """Teste si la liste est vide"""
    return self.tete is None

  def ajoute(self, x):
  """Ajoute un élément en tête de liste"""
    self.tete = Cellule(x, self.tete)

lst = Liste()
lst.ajoute(1)
lst.ajoute(2)
lst.ajoute(3) # on a construit la liste 3,2,1
```

Encapsulation dans un objet (méthodes)

```
...
  def longueur(self):
  """Renvoie la longueur de la liste"""
    return longueur(self.tete)
```

On utilise ici la **fonction** `longueur` définie plus tôt.

Intégration avec Python

Le langage Python permet de traiter un type particulier de collection (comme les listes chaînées) comme les autres collections prédéfinies (`str`, `list`, `bytes`, ...).
Il suffit d'utiliser des noms particuliers pour certaines méthodes :

- `o[i]` $\rightsquigarrow$ `o.__getitem__(i)` (`neme_element`)
- `len(o)` $\rightsquigarrow$ `o.__len__()` (`longueur`)
- `o1 + o2` $\rightsquigarrow$ `o1.__add__(o2)` (`concatener`)

Il est aussi possible de supporter la notation :

```
for v in l:
    ...
```

avec `l` un objet de type `Liste` (cf. exercice avancé)