Structures de données linéaires

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Complexité d'un algorithme

Analyser un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Complexité d'un algorithme

Analyser un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Il est nécessaire de préciser les instructions élémentaires disponibles :

- opérations arithmétiques ;
- comparaisons de données ;
- transferts de données ;
- instructions de contrôle.

mais même ainsi, nous ne sommes pas assurés qu'elles se réalisent en coût constant.

Complexité d'un algorithme

Analyser un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Il est nécessaire de préciser les instructions élémentaires disponibles :

- opérations arithmétiques ;
- comparaisons de données ;
- transferts de données ;
- instructions de contrôle.

mais même ainsi, nous ne sommes pas assurés qu'elles se réalisent en coût constant.

> *Les entiers* PYTHON *sont de type long : les opérations arithmétiques ne sont pas de coût constant.*

→ Sauf mention contraire on les supposera de taille bornée.

Complexité d'un algorithme

Analyser un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Il est nécessaire de préciser les instructions élémentaires disponibles :

- opérations arithmétiques ;
- comparaisons de données ;
- transferts de données ;
- instructions de contrôle.

mais même ainsi, nous ne sommes pas assurés qu'elles se réalisent en coût constant.

> *La comparaison entre chaînes de caractères n'est pas de coût constant.*

→ Seule la comparaison entre deux caractères sera supposée de coût constant.

Complexité d'un algorithme

Analyser un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Il est nécessaire de préciser les instructions élémentaires disponibles :

- opérations arithmétiques ;
- comparaisons de données ;
- transferts de données ;
- instructions de contrôle.

mais même ainsi, nous ne sommes pas assurés qu'elles se réalisent en coût constant.

La recopie d'un tableau entier n'est pas de coût constant.

→ Seule la copie d'une case d'un tableau sera supposée de coût constant.

Complexité d'un algorithme

Analyser un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Quelques questions restent en suspens :

- coût de l'ajout d'un élément dans un tableau (méthode append) ?
- coût de la suppression d'un élément dans un tableau (méthode pop) ?

Pour y répondre, il faut étudier sur l'implémentation des tableaux en PYTHON.

Notations mathématiques

La taille de l'entrée est un (ou plusieurs) entiers dont dépendent les paramètres du problème :

- nombre d'éléments d'un tableau ;
- nombre de bits nécessaire à la représentation des données ;
- nombre de sommets et d'arêtes d'un graphe ;
- etc.

Notations mathématiques

La taille de l'entrée est un (ou plusieurs) entiers dont dépendent les paramètres du problème :

- nombre d'éléments d'un tableau ;
- nombre de bits nécessaire à la représentation des données ;
- nombre de sommets et d'arêtes d'un graphe ;
- etc.

Pour exprimer l'*ordre de grandeur* du nombre d'opérations élémentaires requis par l'algorithme, on utilise les notations de Landau :

- $f(n) = \mathrm{O}(\alpha_n) \iff \exists B > 0 \,\middle|\, f(n) \leqslant B\alpha_n$
- $f(n) = \Omega(\alpha_n) \iff \exists B > 0 \,\middle|\, f(n) \geqslant B\alpha_n$
- $f(n) = \Theta(\alpha_n) \iff f(n) = \mathrm{O}(\alpha_n)$ et $f(n) = \Omega(\alpha_n)$

Notations mathématiques

La taille de l'entrée est un (ou plusieurs) entiers dont dépendent les paramètres du problème :

- nombre d'éléments d'un tableau ;
- nombre de bits nécessaire à la représentation des données ;
- nombre de sommets et d'arêtes d'un graphe ;
- etc.

Pour exprimer l'*ordre de grandeur* du nombre d'opérations élémentaires requis par l'algorithme, on utilise les notations de Landau :

- $f(n) = \mathrm{O}(\alpha_n) \iff \exists B > 0 \,\middle|\, f(n) \leqslant B\alpha_n$
- $f(n) = \Omega(\alpha_n) \iff \exists B > 0 \,\middle|\, f(n) \geqslant B\alpha_n$
- $f(n) = \Theta(\alpha_n) \iff f(n) = \mathrm{O}(\alpha_n)$ et $f(n) = \Omega(\alpha_n)$

La notation la plus fréquente est le O, en sous-entendant qu'il existe des configurations de l'entrée pour lesquelles $f(n)$ est effectivement proportionnel à α_n.

Ordre de grandeur et temps d'exécution

En s'appuyant sur une base de 10^9 opérations par seconde on obtient :

	$\log n$	n	$n \log n$	n^2	n^3	2^n
10^2	7 ns	100 ns	0,7 μs	10 μs	1 ms	$4 \cdot 10^{13}$ a
10^3	10 ns	1 μs	10 μs	1 ms	1 s	10^{292} a
10^4	13 ns	10 μs	133 μs	100 ms	17 s	
10^5	17 ns	100 μs	2 ms	10 s	11,6 j	
10^6	20 ns	1 ms	20 ms	17 mn	32 a	

Ordre de grandeur et temps d'exécution

En s'appuyant sur une base de 10^9 opérations par seconde on obtient :

	$\log n$	n	$n \log n$	n^2	n^3	2^n
10^2	7 ns	100 ns	0,7 µs	10 µs	1 ms	$4 \cdot 10^{13}$ a
10^3	10 ns	1 µs	10 µs	1 ms	1 s	10^{292} a
10^4	13 ns	10 µs	133 µs	100 ms	17 s	
10^5	17 ns	100 µs	2 ms	10 s	11,6 j	
10^6	20 ns	1 ms	20 ms	17 mn	32 a	

Les coûts *raisonnables* sont les coûts :

- logarithmique en $\mathrm{O}(\log n)$;
- linéaire en $\mathrm{O}(n)$;
- semi-linéaire en $\mathrm{O}(n \log n)$;
- quadratique en $\mathrm{O}(n^2)$.

Au delà, les coûts sont prohibitifs.

Structures de données linéaires

Une structure de données spécifie la façon de représenter des données en mémoire en décrivant :

- la manière d'attribuer de la mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Structures de données linéaires

Une structure de données spécifie la façon de représenter des données en mémoire en décrivant :

- la manière d'attribuer de la mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Lorsque la quantité de mémoire allouée est fixée au moment de la création la structure de données est statique.
Lorsque l'attribution de la mémoire peut varier au cours du temps, la structure de données est dynamique.
Lorsque le contenu d'une structure de donnée est modifiable, la structure de donnée est mutable.

Structures de données linéaires

Une structure de données spécifie la façon de représenter des données en mémoire en décrivant :

- la manière d'attribuer de la mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Lorsque la quantité de mémoire allouée est fixée au moment de la création la structure de données est statique.
Lorsque l'attribution de la mémoire peut varier au cours du temps, la structure de données est dynamique.
Lorsque le contenu d'une structure de donnée est modifiable, la structure de donnée est mutable.
En PYTHON la classe *list* est dynamique et mutable :

```
>>> l = [1, 2, 3]
>>> l.append(4)
>>> l[0] = 5
>>> l
[5, 2, 3, 4]
```

Structures de données linéaires

Une structure de données spécifie la façon de représenter des données en mémoire en décrivant :

- la manière d'attribuer de la mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Lorsque la quantité de mémoire allouée est fixée au moment de la création la structure de données est statique.
Lorsque l'attribution de la mémoire peut varier au cours du temps, la structure de données est dynamique.
Lorsque le contenu d'une structure de donnée est modifiable, la structure de donnée est mutable.
En PYTHON les classes *tuple* et *str* sont statiques et non mutables :

```
>>> t = (1, 2, 3)
>>> t.append(4)
AttributeError: 'tuple' object has no attribute 'append'
>>> t[0] = 5
TypeError: 'tuple' object does not support item assignment
```

Structures de données linéaires

Une structure de données spécifie la façon de représenter des données en mémoire en décrivant :

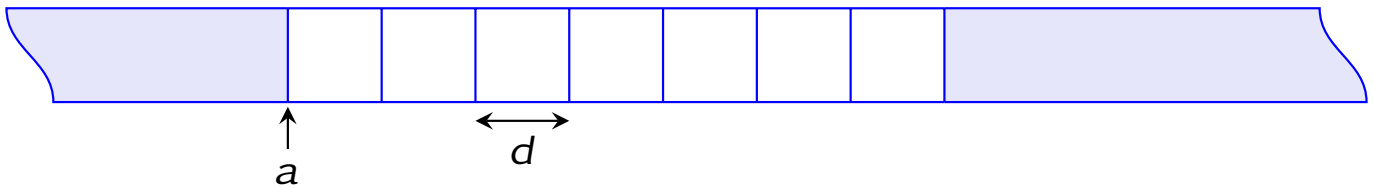
- la manière d'attribuer de la mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Les structures de données classiques appartiennent le plus souvent aux familles suivantes :

- les structures linéaires (listes, tableaux, piles, files) ;
- les *matrices* ou *tableaux multidimensionnels* ;
- les *structures arborescentes* (arbres binaires) ;
- les *structures relationnelles* (bases de données ou graphes).

Tableaux et listes

Les tableaux forment une suite de variables de même type associées à des emplacements consécutifs de la mémoire :

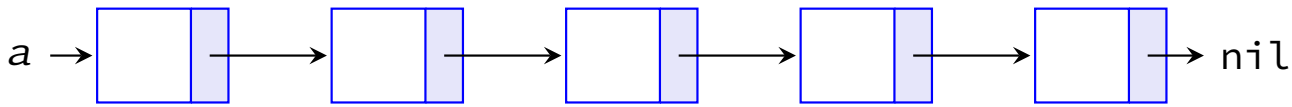


- un tableau est une structure de donnée statique mutable ;
- les éléments du tableau sont accessibles en lecture et en écriture en temps constant $O(1)$.

Les tableaux existent en PYTHON : c'est la classe *array* fournie par la bibliothèque numpy.

Tableaux et listes

Les listes associent à chaque donnée un pointeur indiquant la localisation dans la mémoire de la donnée suivante :

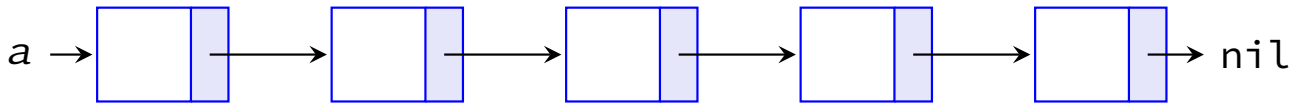


- une liste est une structure de donnée dynamique (souvent non mutable) ;
- le n^e élément d'une liste est accessible en temps $O(n)$.

Les listes n'existent pas en PYTHON (la classe *list* n'est pas une liste chaînée).

Tableaux et listes

Les listes associent à chaque donnée un pointeur indiquant la localisation dans la mémoire de la donnée suivante :



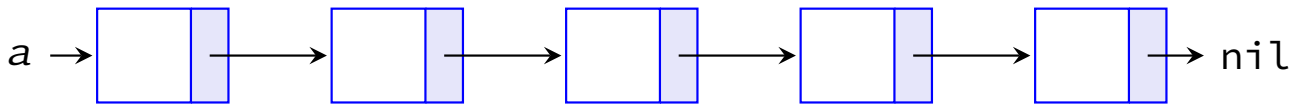
- une liste est une structure de donnée dynamique (souvent non mutable) ;
- le n^e élément d'une liste est accessible en temps $O(n)$.

Les listes n'existent pas en PYTHON (la classe *list* n'est pas une liste chaînée).

Il existe d'autres types de listes : listes doublement chaînées permettant l'accès à la donnée précédente, listes circulaires dans lesquelles la dernière case pointe vers la première, etc.

Tableaux et listes

Les listes associent à chaque donnée un pointeur indiquant la localisation dans la mémoire de la donnée suivante :



- une liste est une structure de donnée dynamique (souvent non mutable) ;
- le n^e élément d'une liste est accessible en temps $O(n)$.

Les listes n'existent pas en PYTHON (la classe *list* n'est pas une liste chaînée).

Il existe d'autres types de listes : listes doublement chaînées permettant l'accès à la donnée précédente, listes circulaires dans lesquelles la dernière case pointe vers la première, etc.

Il existe enfin des structures de données qui tentent de concilier les avantages des tableaux (accès des éléments en coût constant) et des listes (structure dynamique) : la classe *list* en est un exemple.

La classe *list* de PYTHON

Les principales opérations et méthodes qui mutent une liste sont :

`l[i] = x`	remplace `l[i]` par `x`
`del l[i]`	supprime l'élément `l[i]`
`l.append(x)`	ajoute un élément `x` en queue de liste
`l.remove(x)`	supprime la première occurrence de `x`
`l.insert(i, x)`	insère `x` en position `i`
`l.pop(i)`	supprime et renvoie le i^e élément de `l`

Chacune de ses opérations a un coût, qui nous est pour l'instant inconnu.

La classe *list* de PYTHON

Les principales opérations et méthodes qui mutent une liste sont :

`l[i] = x`	remplace `l[i]` par `x`
`del l[i]`	supprime l'élément `l[i]`
`l.append(x)`	ajoute un élément `x` en queue de liste
`l.remove(x)`	supprime la première occurrence de `x`
`l.insert(i, x)`	insère `x` en position `i`
`l.pop(i)`	supprime et renvoie le i^e élément de `l`

Chacune de ses opérations a un coût, qui nous est pour l'instant inconnu.

On peut procéder par expérimentation en mesurant le temps mis :

- pour insérer n fois en queue de liste par la méthode append ;
- pour insérer n fois en tête de liste par la méthode `insert(0, _)` ;

et en faisant varier n.

La classe *list* de PYTHON

Les principales opérations et méthodes qui mutent une liste sont :

`l[i] = x`	remplace `l[i]` par `x`
`del l[i]`	supprime l'élément `l[i]`
`l.append(x)`	ajoute un élément `x` en queue de liste
`l.remove(x)`	supprime la première occurrence de `x`
`l.insert(i, x)`	insère `x` en position `i`
`l.pop(i)`	supprime et renvoie le i^e élément de `l`

Chacune de ses opérations a un coût, qui nous est pour l'instant inconnu.

Scripts à tester pour différentes valeurs de n :

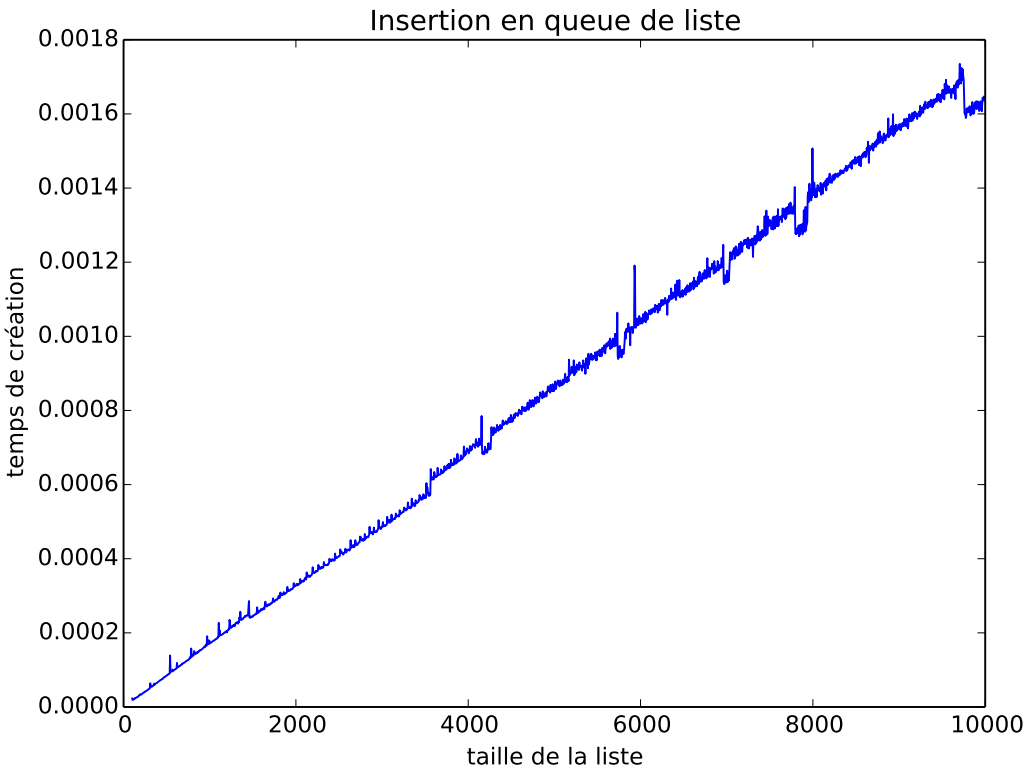
```
l = []
for i in range(n):
    l.append(0)
```

```
l = []
for i in range(n):
    l.insert(0, 0)
```

La classe *list* de PYTHON

Comparaison des méthodes append et `insert`

Temps d'exécution pour insérer n fois en queue de liste par la méthode append :

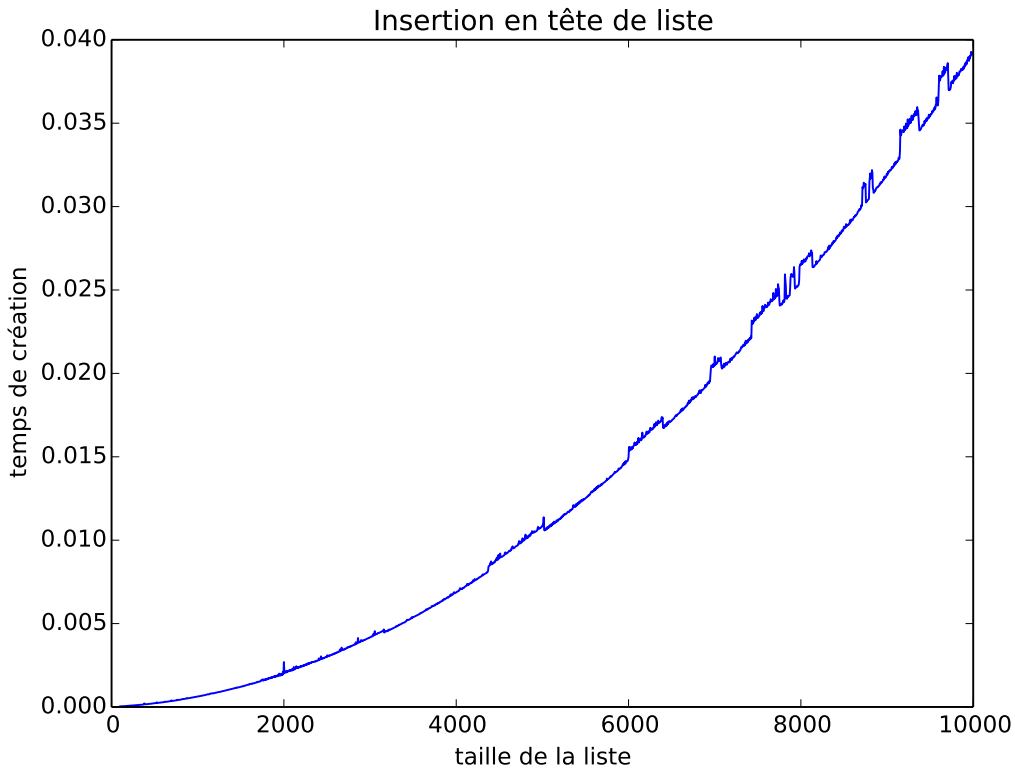


Le coût temporel du script est proportionnel à n ;
$\rightarrow$ la méthode append semble de coût constant.

La classe *list* de PYTHON

Comparaison des méthodes append et `insert`

Temps d'exécution pour insérer n fois en tête de liste par la méthode `insert` :



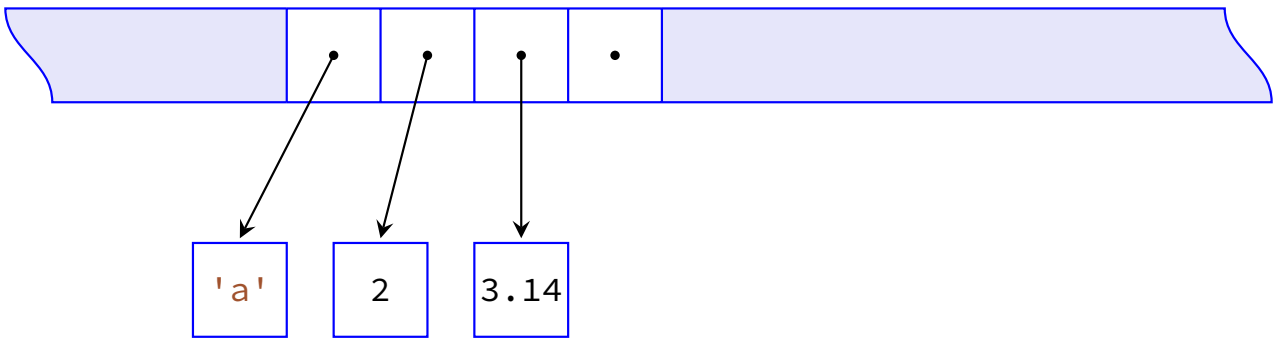
Le coût temporel ne croît pas linéairement avec n ;
→ le coût de la méthode `insert` semble dépendre de la taille de la liste.

La classe *list* de PYTHON

Création et mutation d'une liste

Lorsqu'on crée une liste de taille ℓ, un espace mémoire légèrement plus grand est alloué.

```
l = ['a', 2, 3.14]
```

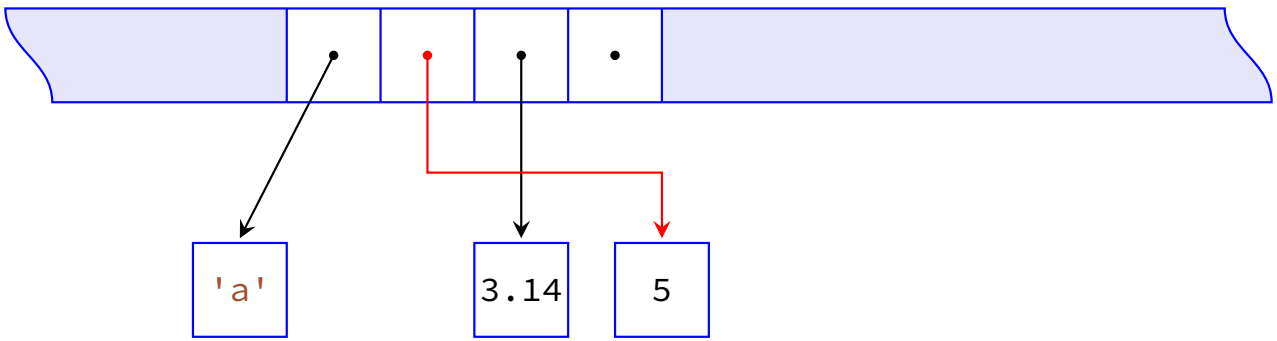


La classe *list* de PYTHON

Création et mutation d'une liste

Puisqu'il s'agit d'un tableau, chaque modification de valeur s'exécute en $O(1)$.

```
l[1] = 5
```

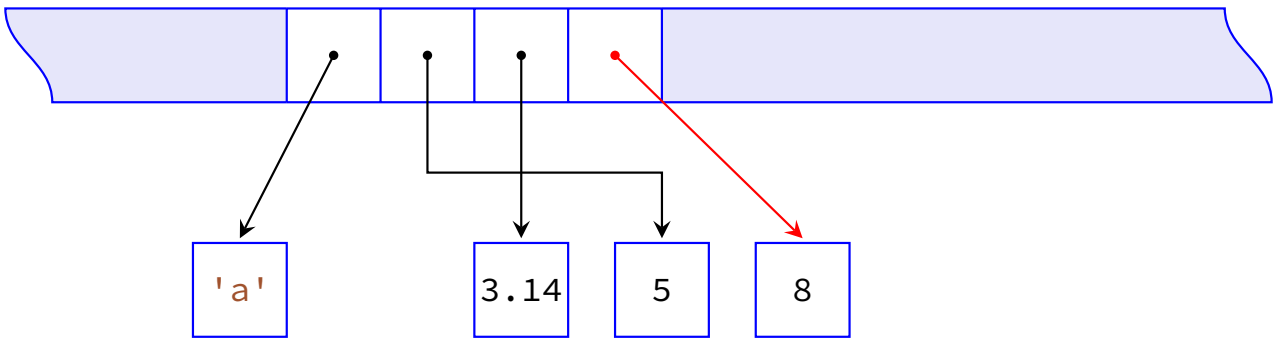


La classe *list* de PYTHON

Création et mutation d'une liste

Les espaces libres sont alloués au fur et à mesure que la liste s'agrandit, en $O(1)$ tant qu'il reste de l'espace.

```
l.append(8)
```

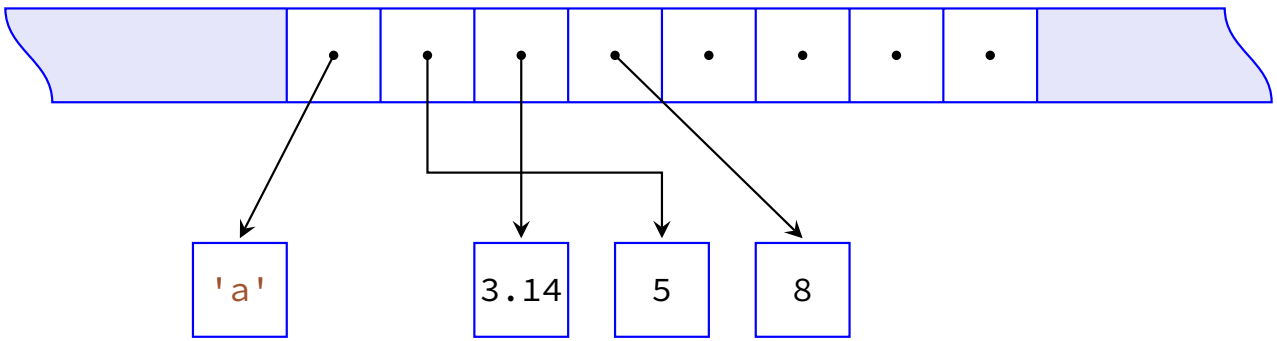


La classe *list* de PYTHON

Création et mutation d'une liste

S'il n'y a plus d'espace libre, la liste est redimensionnée en lui allouant un espace plus grand [...]

```
l.insert(1, 'b')
```

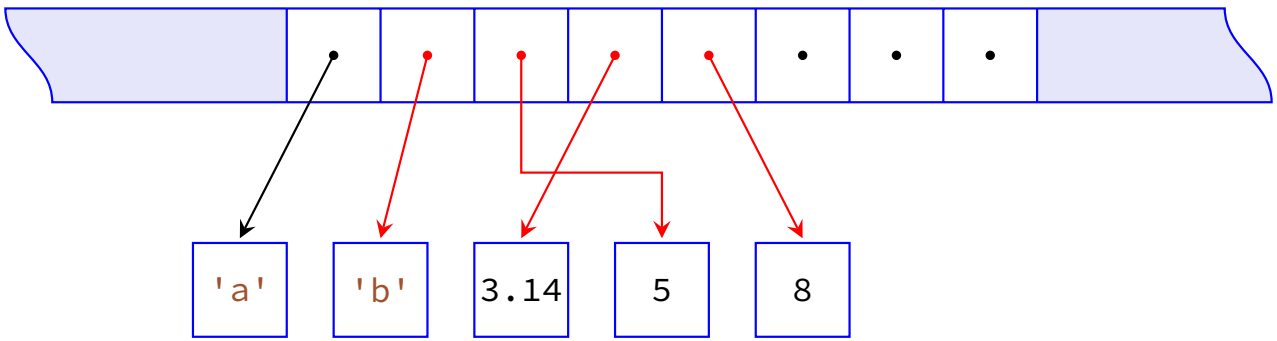


La classe *list* de PYTHON

Création et mutation d'une liste

[...] un nouveau pointeur est ensuite créé et les pointeurs existants modifiés, ce qui se réalise en $O(n)$.

```
l.insert(1, 'b')
```

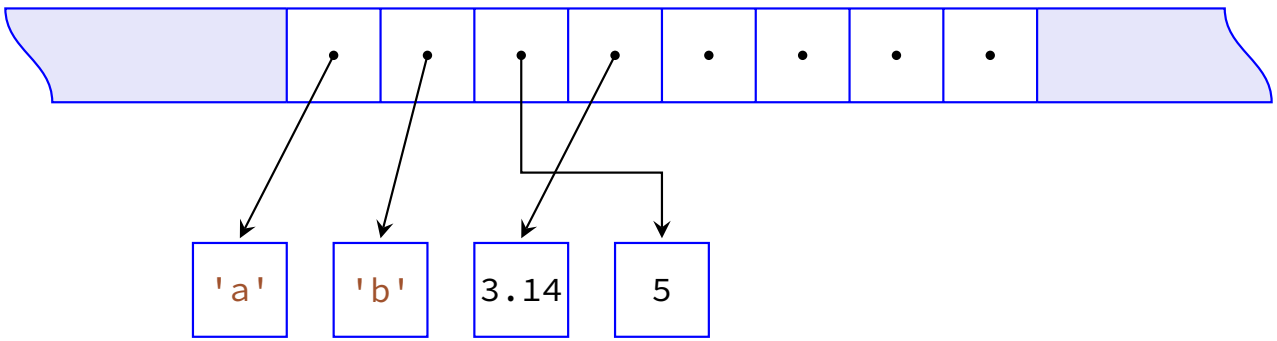


La classe *list* de PYTHON

Création et mutation d'une liste

La suppression en queue de liste par la méthode `pop()` se réalise en $O(1)$.

```
x = l.pop()
```

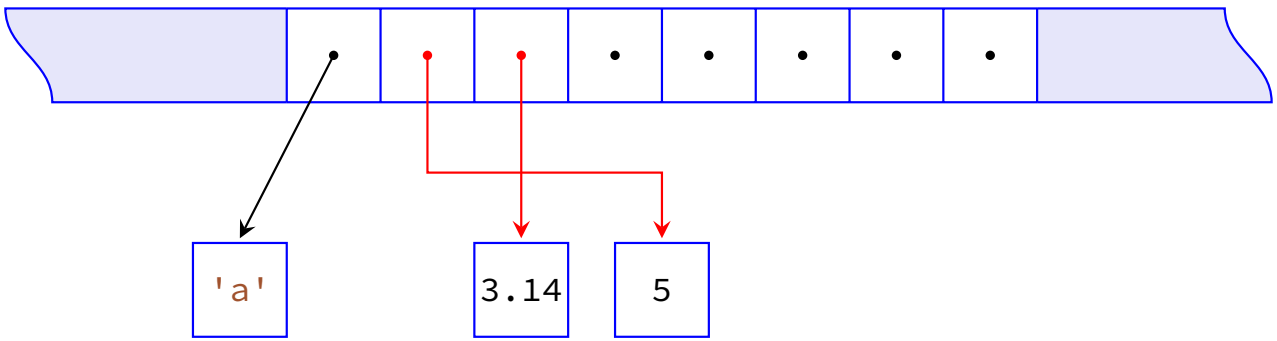


La classe *list* de PYTHON

Création et mutation d'une liste

La suppression d'un élément quelconque de la liste se réalise en $O(n)$ (il faut modifier les pointeurs) [...]

```
l.remove('b')
```

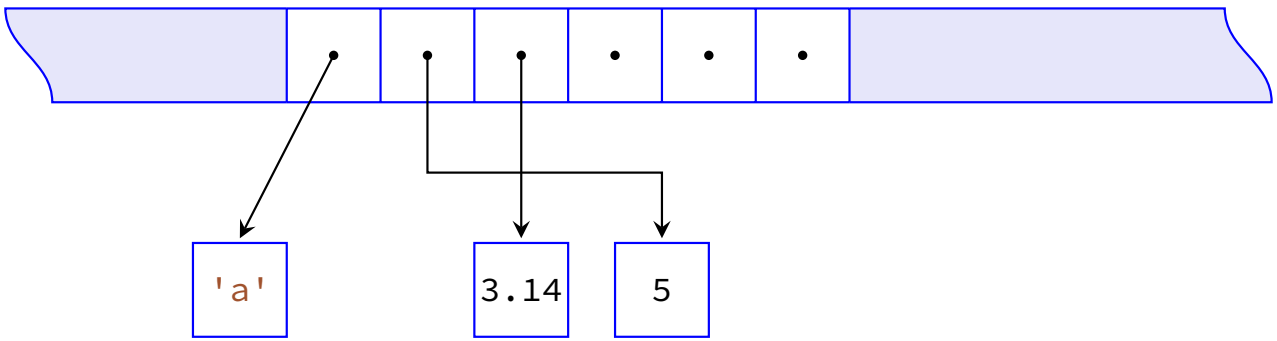


La classe *list* de PYTHON

Création et mutation d'une liste

[...] en outre, lorsque la liste devient trop petite on libère une partie de l'espace alloué.

```
l.remove('b')
```

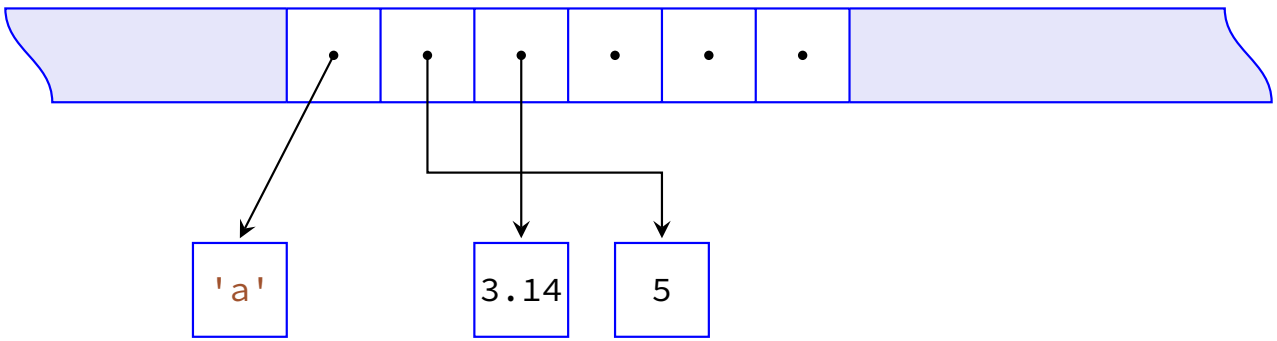


La classe *list* de PYTHON

Création et mutation d'une liste

[...] en outre, lorsque la liste devient trop petite on libère une partie de l'espace alloué.

```
l.remove('b')
```



Compte tenu de cette description, on peut établir les coûts suivants :

`l[i] = x`	$O(1)$
`l.append(x)`	$O(1)$
`l.pop()`	$O(1)$
`l.insert(i, x)`	$O(n)$
`del l[i]` / `l.pop(i)`	$O(n)$
`l.remove(x)`	$O(n)$

dilatation	$O(n)$
contraction	$O(1)$

La classe *list* de PYTHON

Complexité amortie

Peut-on parler de coût constant pour append alors que ponctuellement va se produire un redimensionnement de coût linéaire ?

La classe *list* de PYTHON

Complexité amortie

Peut-on parler de coût constant pour append alors que ponctuellement va se produire un redimensionnement de coût linéaire ?
Considérons de nouveau le script :

```
l = []
for i in range(n):
    l.append(0)
```

Pour évaluer son coût, on dénombre les n ajouts en queue de liste et les redimensionnements à chaque fois que la taille de la liste a atteint 1, 2, 4, 8, 16, ..., 2^{p-1} avec $2^{p-1} < n \leqslant 2^p$.

La classe *list* de PYTHON

Complexité amortie

Peut-on parler de coût constant pour append alors que ponctuellement va se produire un redimensionnement de coût linéaire ?
Considérons de nouveau le script :

```
l = []
for i in range(n):
    l.append(0)
```

Pour évaluer son coût, on dénombre les n ajouts en queue de liste et les redimensionnements à chaque fois que la taille de la liste a atteint 1, 2, 4, 8, 16, …, 2^{p-1} avec $2^{p-1} < n \leqslant 2^p$.
Coût des insertions : $\mathrm{O}(1+1+\cdots+1) = \mathrm{O}(n)$;
Coût des redimensionnements : $\mathrm{O}(1+2+4+\cdots+2^{p-1}) = \mathrm{O}(2^p-1) = \mathrm{O}(n)$.
Le coût du script reste linéaire.

On dira que la complexité amortie de la méthode append est un $\mathrm{O}(1)$.

La classe *list* de PYTHON

Complexité amortie

Extrait du code source CPYTHON :

```
list_resize:
    new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6)
    new_allocated += newsize
```

Si n désigne la taille de la liste, le nombre d'emplacements mémoire alloués sera donc égal à :

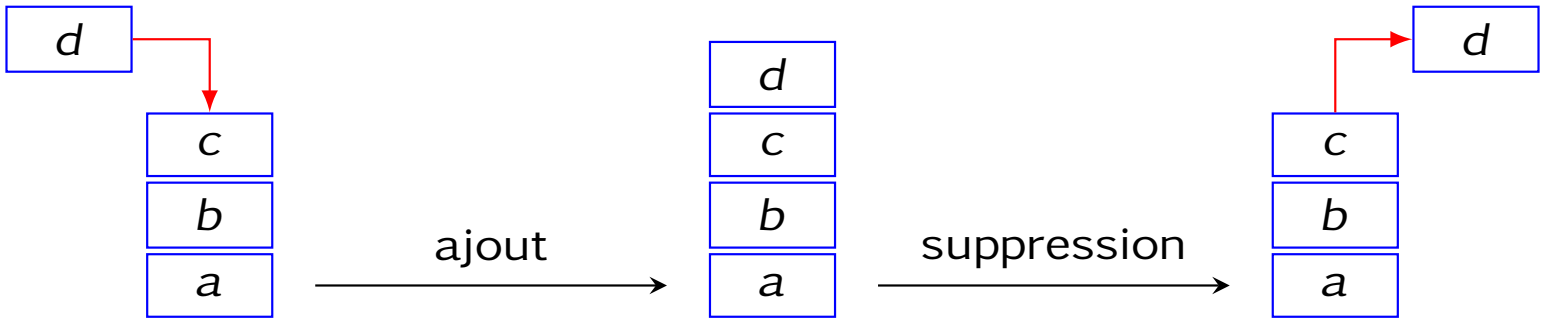
$$n + \left\lfloor \frac{n}{8} \right\rfloor + \begin{cases} 3 & \text{si } n < 9 \\ 6 & \text{sinon} \end{cases}$$

Les redimensionnements du script ont lieu lorsque la taille de la liste atteint :

$$0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, 269, 309, \ldots$$

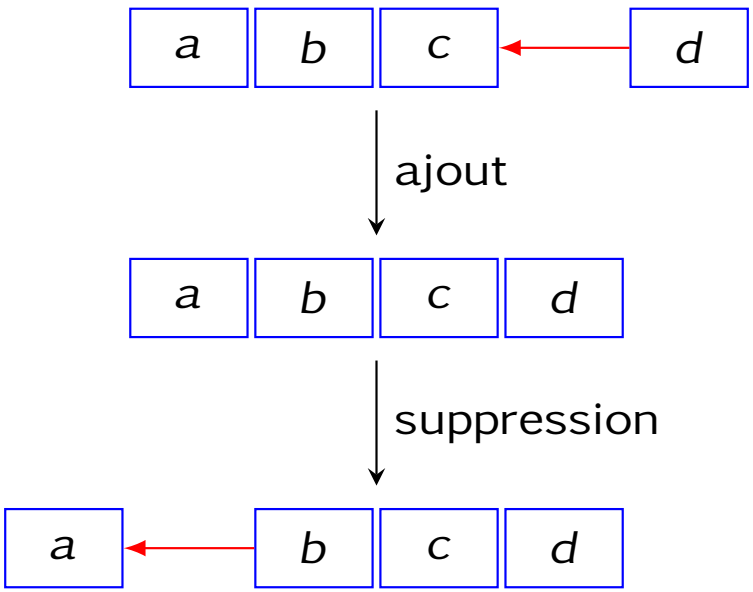
Piles et files

- Les piles sont fondées sur le principe LIFO (*Last In, First Out*).



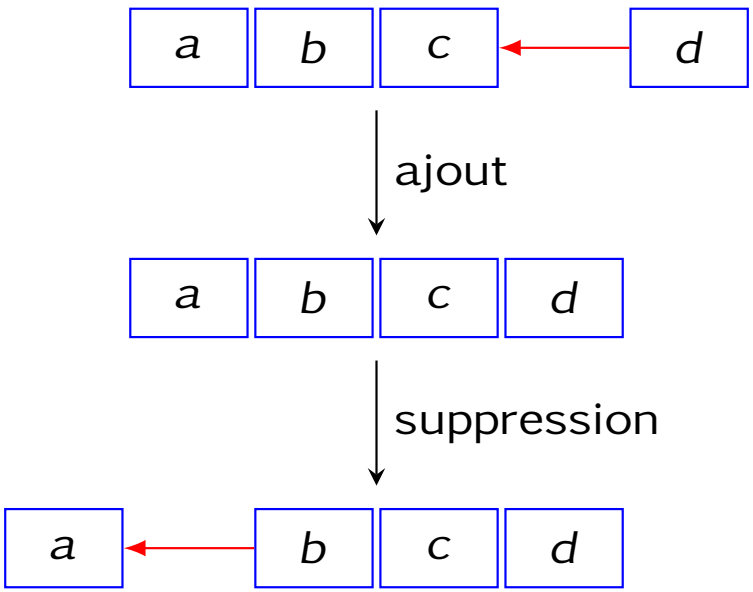
Piles et files

- Les piles sont fondées sur le principe LIFO (*Last In, First Out*).
- Les files sont fondées sur le principe FIFO (*First In, First Out*).



Piles et files

- Les piles sont fondées sur le principe LIFO (*Last In, First Out*).
- Les files sont fondées sur le principe FIFO (*First In, First Out*).



Une réalisation concrète de ces structures doit fournir dans l'idéal :

- une fonction de création d'une pile/file vide ;
- deux fonctions d'ajout et de suppression à coût constant ;
- une fonction vérifiant si une pile/file est vide.

Implémentation pratique d'une pile

à partir d'une liste PYTHON

On utilise une liste pour représenter une pile, en utilisant append pour empiler et pop() pour dépiler, ce qui garantit un coût amorti constant.

```
class Pile:
    def __init__(self):
        self.lst = []

    def empty(self):
        return self.lst == []

    def push(self, x):
        self.lst.append(x)

    def pop(self):
        if self.empty():
            raise ValueError("pile vide")
        return self.lst.pop()
```

Implémentation pratique d'une pile

à partir d'une liste PYTHON

On utilise une liste pour représenter une pile, en utilisant append pour empiler et pop() pour dépiler, ce qui garantit un coût amorti constant.

Exemple d'utilisation :

```
>>> p = Pile()

>>> for i in range(1, 11):
...     p.push(i)

>>> while not p.empty():
...     print(p.pop(), end=' - ')
10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 -
```

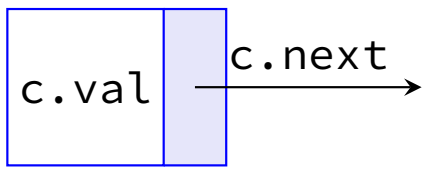
- Lors de la création d'une pile, c'est la méthode `__init__` qui est exécutée.
- L'instance de classe doit apparaître en premier argument des méthodes mais n'apparait dans les paramètres de ces dernières lorsqu'on les utilise.

Implémentation pratique d'une pile

à partir d'une liste chaînée

On commence par définir la classe *Cell*.

```
class Cell:
    def __init__(self, x):
        self.val = x
        self.next = None
```

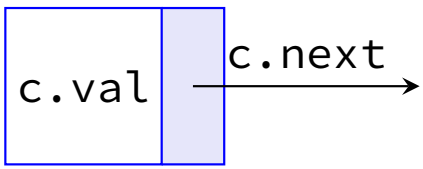


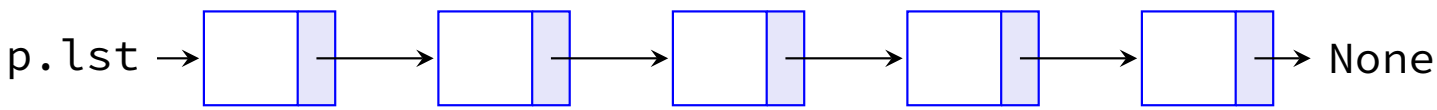
Implémentation pratique d'une pile

à partir d'une liste chaînée

On commence par définir la classe *Cell*. Puis on définit la classe *Pile*.

```
class Cell:
    def __init__(self, x):
        self.val = x
        self.next = None
```





Implémentation pratique d'une pile

à partir d'une liste chaînée

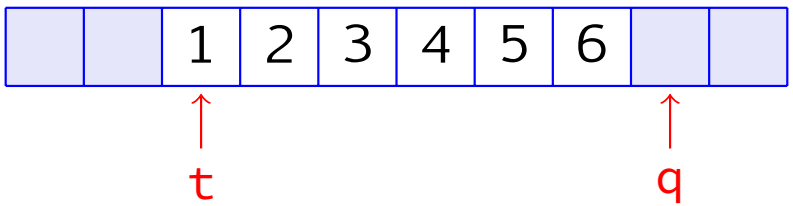
On commence par définir la classe *Cell*. Puis on définit la classe *Pile*.

```
class Pile:
    def __init__(self):
        self.lst = None

    def empty(self):
        return self.lst is None

    def push(self, x):
        c = Cell(x)
        c.next = self.lst
        self.lst = c

    def pop(self):
        if self.empty():
            raise ValueError("pile vide")
        c = self.lst
        self.lst = c.next
        return c.val
```

Implémentation pratique d'une file

à l'aide d'un tableau de taille fixe

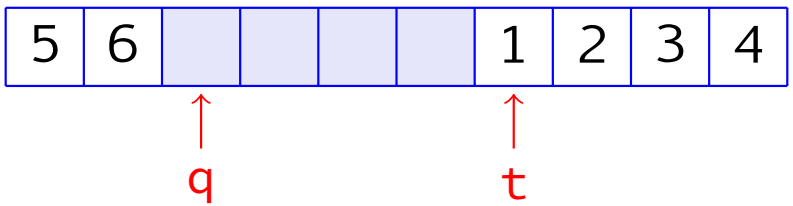
On maintient deux curseurs indiquant la tête et la queue de la file dans le tableau. Les éléments seront retirés à la tête et ajoutés à la queue.



Implémentation pratique d'une file

à l'aide d'un tableau de taille fixe

On maintient deux curseurs indiquant la tête et la queue de la file dans le tableau. Les éléments seront retirés à la tête et ajoutés à la queue.

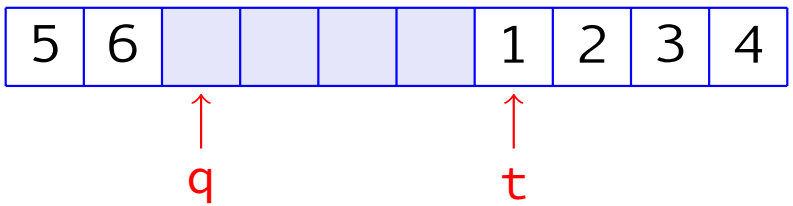


Le tableau est rempli de façon circulaire : on peut avoir `q` < `t`.

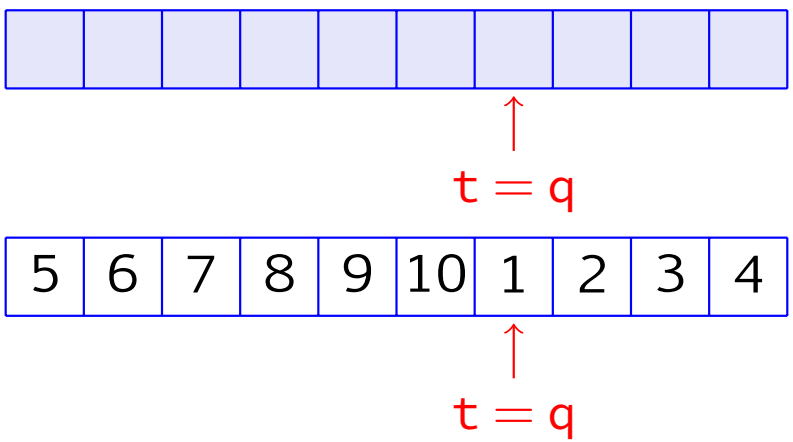
Implémentation pratique d'une file

à l'aide d'un tableau de taille fixe

On maintient deux curseurs indiquant la tête et la queue de la file dans le tableau. Les éléments seront retirés à la tête et ajoutés à la queue.



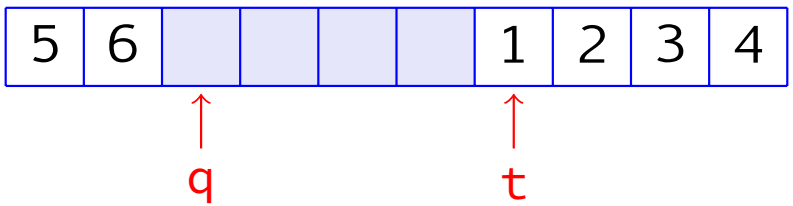
Le tableau est rempli de façon circulaire : on peut avoir $q < t$.
Problème : comment distinguer une file vide d'une file pleine ?



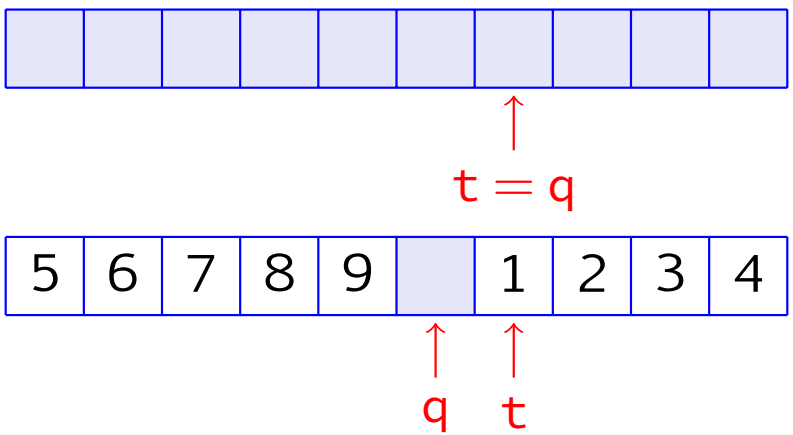
Implémentation pratique d'une file

à l'aide d'un tableau de taille fixe

On maintient deux curseurs indiquant la tête et la queue de la file dans le tableau. Les éléments seront retirés à la tête et ajoutés à la queue.



Le tableau est rempli de façon circulaire : on peut avoir $q < t$.
Problème : comment distinguer une file vide d'une file pleine ?



Solution : on ne remplit pas la dernière case du tableau ; $q = t$ caractérise la file vide et $q = t - 1 \mod n$ la file pleine.

Implémentation pratique d'une file

à l'aide d'un tableau de taille fixe

```
class File:
    """ définition d'une file à l'aide d'un tableau """
    def __init__(self, n):
        self.lst = [None] * n
        self.size = n
        self.t = 0
        self.q = 0

    def empty(self):
        return self.t == self.q

    def full(self):
        return (self.q + 1) % self.size == self.t

[...]
```

Implémentation pratique d'une file

à l'aide d'un tableau de taille fixe

```
class File:
    """ définition d'une file à l'aide d'un tableau """
    def __init__(self, n):
        self.lst = [None] * n
        self.size = n
        self.t = 0
        self.q = 0
[...]

def add(self, x):
        if self.full():
            raise ValueError("file pleine")
        self.lst[self.q] = x
        self.q = (self.q + 1) % self.size

    def take(self):
        if self.empty():
            raise ValueError("file vide")
        x = self.lst[self.t]
        self.t = (self.t + 1) % self.size
        return x
```

Implémentation pratique d'une file

à l'aide d'un tableau de taille fixe

Exemple d'utilisation :

```
>>> f = File(20)       # la file peut contenir au maximum 19 éléments

>>> for i in range(1, 11):
...     f.add(i)

>>> while not f.empty():
...     print(f.take(), end=' - ')
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
```

Implémentation pratique d'une file

à l'aide d'un tableau de taille fixe

Exemple d'utilisation :

```
>>> f = File(20)       # la file peut contenir au maximum 19 éléments

>>> for i in range(1, 11):
...     f.add(i)

>>> while not f.empty():
...     print(f.take(), end=' - ')
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
```

Cette solution présente une contrainte : passer en paramètre la taille du tableau à utiliser lors de la création de la file. D'autres solutions existent, par exemple en utilisant une liste doublement chaînée ou une liste circulaire, ne présentant pas cette contrainte.