Chapitre 1
Structures de données linéaires

1. Complexité d'un algorithme

Analyser un algorithme revient le plus souvent à évaluer les ressources nécessaires à son exécution (la quantité de mémoire requise) et le temps de calcul à prévoir. Bien évidemment, ces deux notions dépendent de nombreux paramètres matériels qui sortent du domaine de l'algorithmique : nous ne pouvons attribuer une valeur absolue ni à la quantité de mémoire requise ni au temps d'exécution d'un algorithme donné. En revanche, il est souvent possible d'évaluer l'*ordre de grandeur* de ces deux quantités de manière à identifier l'algorithme le plus efficace au sein d'un ensemble d'algorithmes résolvant le même problème.

Pour réaliser cette évaluation, il est nécessaire de préciser un modèle de la technologie employée ; en ce qui nous concerne, il s'agira d'une machine à processeur unique pour laquelle les instructions seront exécutées l'une après l'autre, sans opération simultanées. Il faudra aussi préciser les instructions élémentaires disponibles ainsi que leurs coûts. Ceci est particulièrement important lorsqu'on utilise un langage de programmation tel que PYTHON pour illustrer ce cours car ce langage possède de nombreuses instructions de haut niveau qu'il serait irréaliste de considérer comme ayant un coût constant : par exemple, la fonction `sort` permet effectivement de trier un tableau en une instruction, mais il serait illusoire de croire que son temps d'exécution est indépendant de la taille du tableau. En outre, pour évaluer cette dépendance il n'y a guère d'autre solution que de se plonger dans le code source de PYTHON ou sa documentation ; lorsque nous étudierons les algorithmes de tri il sera plus sage de ne pas tenir compte de l'existence de cette instruction.

1.1 Instructions élémentaires

Les instructions élémentaires (et qui seront considérées comme ayant un coût constant) sont présentes dans la plupart des langages de programmation :

- opérations arithmétiques (addition, soustraction, multiplication, division, modulo, partie entière, ...)
- comparaisons de données (relation d'égalité, d'infériorité, ...)
- transferts de données (lecture et écriture dans un emplacement mémoire)
- instructions de contrôle (branchement conditionnel et inconditionnel, appel à une fonction auxiliaire, ...)

mais là encore il est parfois nécessaire de préciser la portée de certaines de ces instructions. En arithmétique par exemple, il est impératif que les données représentant les nombres soient codées sur un nombre *fixe* de bits. C'est le cas en général des nombres flottants (la classe *float*) et des entiers relatifs (la classe *int*) représentés usuellement sur 64 bits [1], mais dans certains langages existe aussi un type *entier long* dans lequel les entiers ne sont pas limités en taille. C'est le cas en PYTHON, où coexistaient jusqu'à la version 3.0 du langage une classe *int* et une classe *long*. Ces deux classes ont depuis fusionné, le passage du type *int* au type *long* étant désormais transparent pour l'utilisateur.
Dans le cas des nombres entiers, l'exponentiation peut aussi être source de discussion : s'agit-t'il d'une opération de coût constant ? En général on répond à cette question par la négative : le calcul de n^k nécessite un nombre d'opérations élémentaires (essentiellement des multiplications) qui dépend de k. Cependant, certains processeurs possèdent une instruction permettant de décaler de k bits vers la gauche la représentation binaire d'un entier, autrement dit de calculer 2^k en coût constant.

Les comparaisons entre nombres (du moment que ceux-ci sont codés sur un nombre fixe de bits) seront aussi considérées comme des opérations à coût constant, de même que la comparaison entre deux caractères. En revanche, la comparaison entre deux chaînes de caractères ne pourra être considérée comme une opération élémentaire, même s'il est possible de la réaliser en une seule instruction PYTHON. Il en sera de même des opérations d'affectation : lire ou modifier le contenu d'un case d'un tableau est une opération élémentaire, mais ce n'est plus le cas s'il s'agit de recopier tout ou partie d'un tableau dans un autre, même si la technique du *slicing* en PYTHON permet de réaliser très simplement ce type d'opération.

1. Voir cours de première année.

1.2 Notations mathématiques

Une fois précisé la notion d'opération élémentaire, il convient de définir ce qu'on appelle la *taille* de l'entrée. Cette notion dépend du problème étudié : pour de nombreux problèmes, il peut s'agir du nombre d'éléments constituant les paramètres de l'algorithme (par exemple le nombre d'éléments du tableau dans le cas d'un algorithme de tri) ; dans le cas d'algorithmes de nature arithmétique (le calcul de n^k par exemple) il peut s'agir du nombre de bits nécessaire à la représentation des données. Enfin, il peut être approprié de décrire la taille de l'entrée à l'aide de deux entiers (le nombre de sommets et le nombre d'arêtes dans le cas d'un algorithme portant sur les graphes).
Une fois la taille n de l'entrée définie, il reste à évaluer en fonction de celle-ci le nombre $f(n)$ d'opérations élémentaires requises par l'algorithme. Mais même s'il est parfois possible d'en déterminer le nombre exact, on se contentera le plus souvent d'en donner l'ordre de grandeur à l'aide des notations de Landau.
La notation la plus fréquemment utilisée est le « grand O » :

$$f(n) = \mathrm{O}(\alpha_n) \iff \exists \mathrm{B} > 0 \;\Big|\; f(n) \leqslant \mathrm{B}\alpha_n.$$

Cette notation indique que dans le pire des cas, la croissance de $f(n)$ ne dépassera pas celle de la suite (α_n). L'usage de cette notation exprime l'objectif qu'on se donne le plus souvent : déterminer le temps d'exécution dans le cas le plus défavorable. On notera qu'un usage abusif est souvent fait de cette notation, en sous-entendant qu'il existe des configurations de l'entrée pour lesquelles $f(n)$ est effectivement proportionnel à α_n.
D'un usage beaucoup moins fréquent, la notation Ω exprime une minoration du meilleur des cas :

$$f(n) = \Omega(\alpha_n) \iff \exists \mathrm{B} > 0 \;\Big|\; f(n) \geqslant \mathrm{B}\alpha_n.$$

L'expérience montre cependant que pour de nombreux algorithmes le cas « moyen » est beaucoup plus souvent proche du cas le plus défavorable que du cas le plus favorable. En outre, on souhaite en général avoir la certitude de voir s'exécuter un algorithme en un temps raisonnable, ce que ne peut exprimer cette notation.
Enfin, lorsque le pire et le meilleur des cas ont même ordre de grandeur, on utilise la notation Θ :

$$f(n) = \Theta(\alpha_n) \iff f(n) = \mathrm{O}(\alpha_n) \text{ et } f(n) = \Omega(\alpha_n).$$

Cette notation exprime le fait que quelle que soit le configuration de l'entrée, le temps d'exécution de l'algorithme sera *grosso-modo* proportionnel à α_n.

Ordre de grandeur et temps d'exécution
Nous l'avons dit, la détermination de la complexité algorithmique ne permet pas d'en déduire le temps d'exécution mais seulement de comparer entre eux deux algorithmes résolvant le même problème. Cependant, il importe de prendre conscience des différences d'échelle considérables qui existent entre les ordres de grandeurs usuels que l'on rencontre. En s'appuyant sur une base de 10^9 opérations par seconde, le tableau de la figure 1 est à cet égard significatif.

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

Figure 1 – Temps nécessaire à l'exécution d'un algorithme en fonction de son coût.

La lecture de ce tableau est édifiante : il faut autant que faire se peut éviter toute complexité temporelle supérieure à un coût quadratique.

2. Structures de données linéaires

Dans son acceptation la plus générale, une *structure de données* spécifie la façon de représenter en mémoire machine les données d'un problème à résoudre en décrivant :

$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	semi-linéaire
$O(n^2)$	quadratique
$O(n^k) \quad (k \geqslant 2)$	polynomiale
$O(k^n) \quad (k > 1)$	exponentielle

FIGURE 2 – Qualifications usuelles des complexités.

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

Dans certains cas, la quantité de mémoire allouée à la structure de donnée est fixée au moment de la création de celle-ci et ne peut plus être modifiée ensuite ; on parle alors de structure de données *statique*. Dans d'autres cas l'attribution de la mémoire nécessaire est effectuée pendant le déroulement de l'algorithme et peut donc varier au cours de celui-ci ; il s'agit alors de structure de données *dynamique*. Enfin, lorsque le contenu d'une structure de donnée est modifiable, on parle de structure de donnée *mutable*.
Par exemple, en PYTHON la classe *tuple* et la classe *str* sont des structures de données statiques et non mutables, contrairement à la classe *list* qui est une structure de donnée dynamique et mutable.

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

```
>>> 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
```

FIGURE 3 – la classe *list* est dynamique et mutable, pas la classe *tuple*.

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

- les *structures linéaires* : il s'agit essentiellement des structures représentables par des suites finies ordonnées ; on y trouve les listes, les tableaux, les piles, les files ;
- les *matrices* ou *tableaux multidimensionnels* ;
- les *structures arborescentes* (en particulier les arbres binaires) ;
- les *structures relationnelles* (bases de données ou graphes pour les relations binaires).

Nous nous intéresserons avant tout aux deux premières.

2.1 Tableaux et listes

Tableaux et listes constituent les principales structures de données linéaires.

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

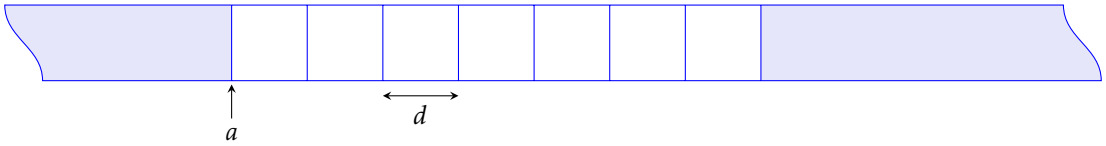


FIGURE 4 – Une représentation d'un tableau en mémoire.

Puisque tous les emplacements sont de même type, ils occupent tous le même nombre d de cases mémoire ; connaissant l'adresse a de la première case du tableau, on accède en coût constant à l'adresse de la case d'indice k en calculant $a + kd$. En revanche, ce type de structure est statique : une fois un tableau créé, la taille de ce dernier ne peut plus être modifiée faute de pouvoir garantir qu'il y a encore un espace mémoire disponible au delà de la dernière case. En résumé :

– un tableau est une structure de donnée statique ;
– 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`.

Listes chaînées
Les *listes* associent à chaque donnée (de même type) un pointeur indiquant la localisation dans la mémoire de la donnée suivante (à l'exception de la dernière, qui pointe vers une valeur particulière indiquant la fin de la liste).

FIGURE 5 – Une représentation d'une liste en mémoire.

Dans une liste, il est impossible de connaître à l'avance l'adresse d'une case en particulier, à l'exception de la première. Pour accéder à la n^e case il faut donc parcourir les $n-1$ précédentes : le coût de l'accès à une case est linéaire. En contrepartie, ce type de structure est dynamique : une fois la liste crée, il est toujours possible de modifier un pointeur pour insérer une case supplémentaire. En résumé :

– une liste est une structure de donnée dynamique ;
– le n^e élément d'une liste est accessible en temps O(n).

On notera que le type de liste que l'on vient de présenter est le plus courant (il s'agit de listes *chaînées*) mais il en existe d'autres : listes *doublement chaînées* permettant l'accès non seulement à la donnée suivante mais aussi à la donnée précédente, *listes circulaires* dans lesquelles la dernière case pointe vers la première, etc.

Contrairement à ce que pourrait laisser croire son nom, la classe *list* en PYTHON n'est pas une liste au sens qu'on vient de lui donner, mais une structure de donnée plus complexe qui cherche à concilier les avantages des tableaux et des listes, à savoir être une structure de donnée dynamique dans laquelle les éléments sont accessibles à coût constant. Bien que la description de ce type de structure sorte du cadre strict du programme, il peut être intéressant d'en donner un aperçu pour en dévoiler l'ingéniosité ; c'est ce que nous allons faire dans la section suivante.

2.2 La classe *list* de PYTHON

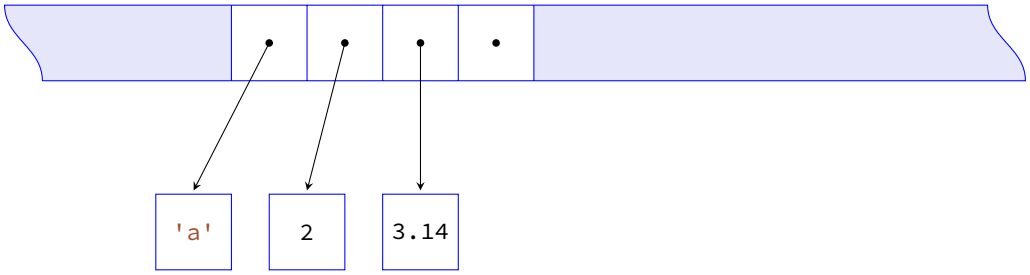
Avant de décrire la façon dont sont représentés les objets de la classe *list* en PYTHON, passons en revue les principales opérations et méthodes qui mutent une liste. Dans le tableau suivant, `l` est une instance de la classe *list*, `i` un entier et `x` un objet qu'on écrit, ajoute, supprime ou recherche dans `l`.

`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`

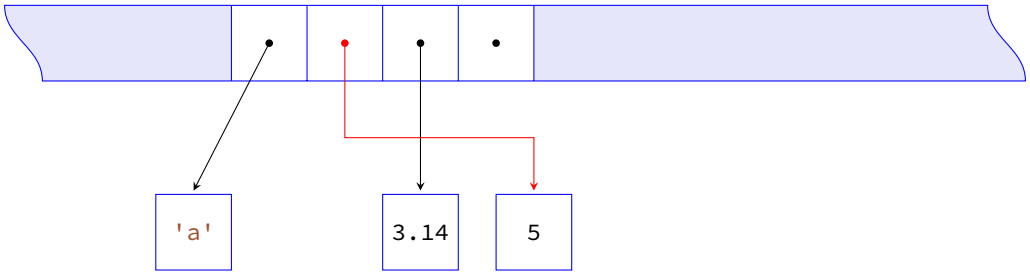
Chacune de ses opérations à un coût, mais il nous est pour l'instant impossible de dire lequel faute de connaître la représentation en mémoire d'une liste.

Commençons par décrire la méthode de création [2]. Lorsqu'on crée une liste de taille ℓ, un espace mémoire légèrement plus grand est alloué (on verra plus loin dans quelle proportion). Par exemple, lors de la création de la liste à trois éléments `l = ['a', 2, 3.14]`, un espace mémoire à 4 emplacements est créé, les trois premiers contenant des pointeurs en direction des valeurs de la liste. C'est pour cette raison qu'une liste peut accueillir des valeurs de types différents : une instance de la classe *list* n'est en réalité qu'un tableau de pointeurs. Le dernier espace pour l'instant ne contient rien.

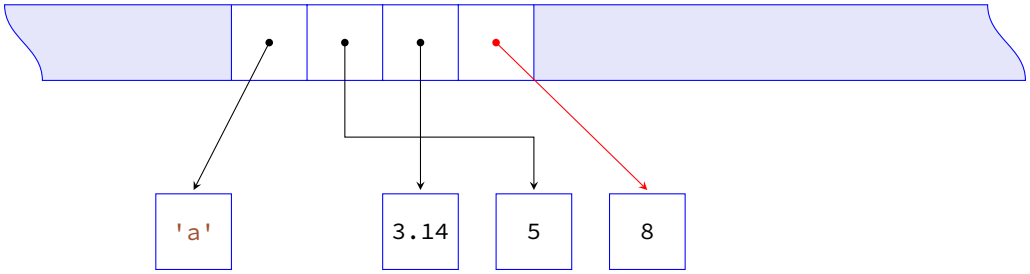
2. Ce qui va suivre n'est valable que pour CPYTHON, l'implémentation écrite en C du langage PYTHON. Il s'agit de la principale des implémentations du langage, celle sur laquelle est basée PYZO, par exemple.



Puisqu'il s'agit d'un tableau, chaque pointeur est accessible à coût constant et l'opération `l[1] = 5` s'exécute en O(1) puisqu'il suffit de modifier un pointeur :

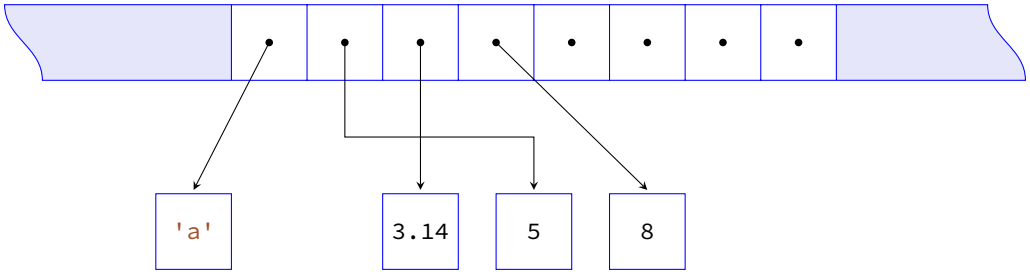


Les espaces libres sont alloués au fur et à mesure que la liste s'agrandit ; dans le cas de notre exemple, après l'instruction `l.append(8)` la situation en mémoire devient :

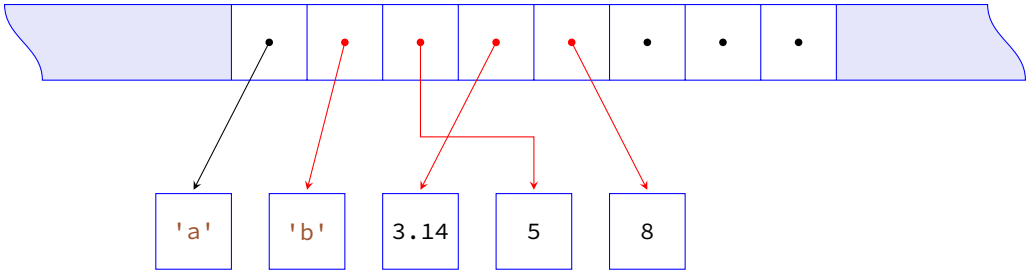


Bien entendu, tant qu'il reste des emplacements disponibles ces ajouts en fin de liste se font à coût constant.

Supposons maintenant que l'on souhaite insérer une valeur supplémentaire avec `l.insert(1, 'b')`. Puisqu'il n'y a plus d'emplacements libres, la liste doit d'abord être redimensionnée : il faut lui allouer un espace plus grand, qui pour notre exemple sera de 8 emplacements.

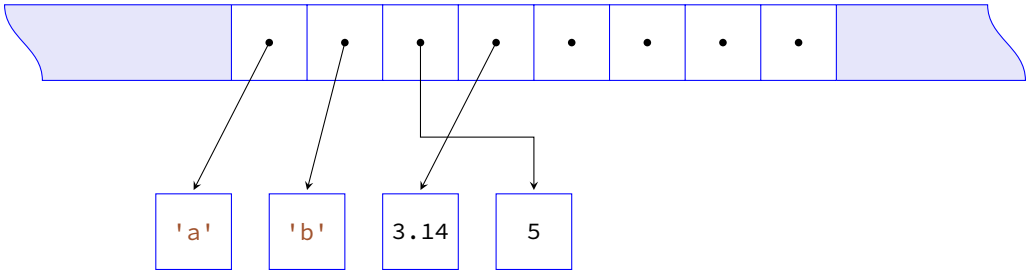


Un nouveau pointeur est ensuite créé, et les pointeurs existants sont modifiés pour refléter le nouvel ordre des éléments de la liste :



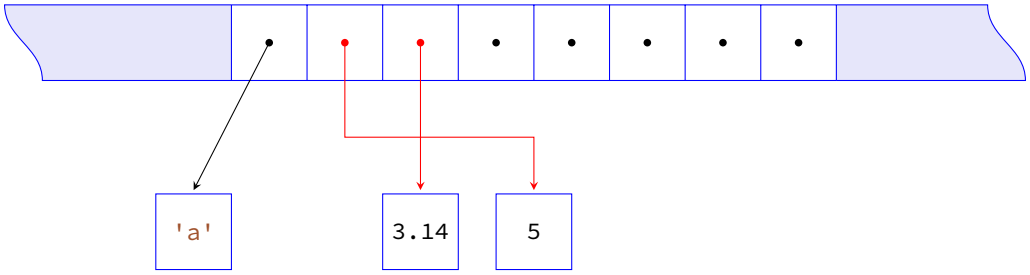
Si on ne tient pas compte du redimensionnement (dont le coût sera évoqué plus loin), le coût de la méthode `insert` est linéaire.

Supprimons maintenant un élément, par exemple avec `l.pop()`. En l'absence de paramètre, le dernier élément de la liste est supprimé, ce qui revient à supprimer le dernier pointeur :

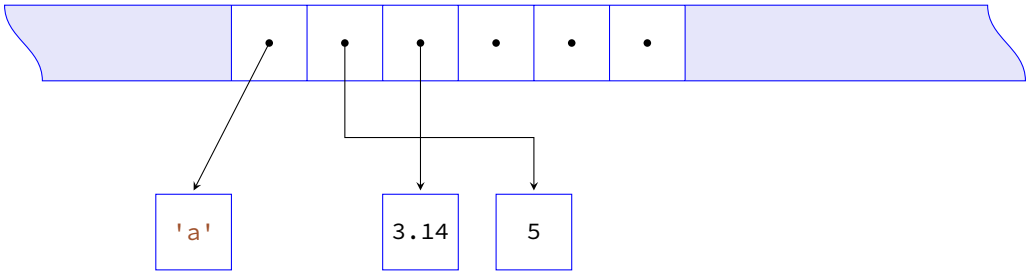


On notera que cette opération, lorsqu'elle s'applique au dernier élément de la liste, est manifestement de coût constant.

Supprimons-en maintenant un second, avec `l.remove('b')`. Il est cette fois nécessaire de modifier certains pointeurs, donc le coût de cette opération est linéaire :



En outre, nous n'avons plus maintenant qu'une liste à 3 éléments alors que 8 emplacements ont été dégagés. Puisque la taille de la liste est strictement inférieure à la moitié de l'emplacement prévu, ce dernier est redimensionné en libérant de l'espace pour ne plus garder que le double de la taille de la liste, à savoir 6 emplacements :



Que retenir de cette description ? Avant tout que la classe *list* correspond à des *tableaux dynamiques*. En l'absence de redimensionnement, les opérations qui modifient la taille d'une liste présentent les coûts suivants, fonction de la taille n de la liste :

`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)`	$O(n)$

À ces coûts peuvent s'ajouter ponctuellement les coûts de dilatation ou de contraction de la taille de l'espace mémoire alloué à la liste :

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

La dilatation a un coût plus important car il n'est pas toujours possible d'allouer un espace plus grand à la liste sans la déplacer dans son entier. Dans ce cas, tous les pointeurs doivent être recréés, ce qui explique le coût linéaire. En revanche, la contraction est de coût constant car il suffit de libérer de l'espace sans modifier l'emplacement de la liste et donc les pointeurs qu'elle contient.

Complexité amortie
Peut-on réellement parler de coût constant pour la méthode append alors que ponctuellement va se produire un redimensionnement de coût linéaire ? Nous allons voir que d'une certaine façon on peut répondre par l'affirmative, en nous livrant à un petit calcul. Nous allons considérer le script suivant :

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

Pour évaluer le coût de ce script, il faut compter, outre les n ajouts en queue de liste, les redimensionnements. Pour simplifier le calcul, nous allons pour l'instant supposer que ces redimensionnements ont eu lieu à chaque fois que la taille de la liste a atteint $1,\ 2,\ 4,\ 8,\ 16,\ \ldots,\ 2^{p-1}$ avec $2^{p-1} < n \leqslant 2^p$.
Chaque redimensionnement pouvant conduire à une recopie de la liste dans son entier, le coût de ceux-ci est un $O(1+2+4+8+\cdots+2^{p-1}) = O(2^p-1) = O(n)$. Autrement dit, le coût du script reste linéaire, ce qui nous permet d'affirmer qu'*en moyenne* les coût de chacune de ces insertions en fin de liste est de coût constant. On dira que la complexité *amortie* [3] de la méthode append est bien un $O(1)$.
Le calcul que nous venons de faire pourrait être répété avec les mêmes conclusions pour toute croissance géométrique : pour que le coût amorti soit constant, il suffit que chaque nouvelle allocation de mémoire soit égale à la taille de la précédente multipliée par un certain facteur $k > 1$. Il reste à se poser la question de savoir si un facteur est préférable à un autre, mais là on rentre dans des considérations très techniques liées à la notion d'allocation de mémoire que nous n'aborderons pas.
Une étude du code source Cpython nous apprend que le facteur de croissance choisi est approximativement $k = 1,125$. Plus précisément, voici la portion du code source correspondant à la ré-allocation de mémoire ; il s'agit d'un code écrit en C, mais il reste facile à comprendre :

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

En C l'opération `>> 3` correspond à la suppression des trois derniers bits d'un entier écrit en base 2, autrement dit au calcul de $\lfloor n/8 \rfloor$. Si n désigne la taille de la liste, le nombre d'emplacements mémoire qui lui sera alloué sera donc égal à :

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

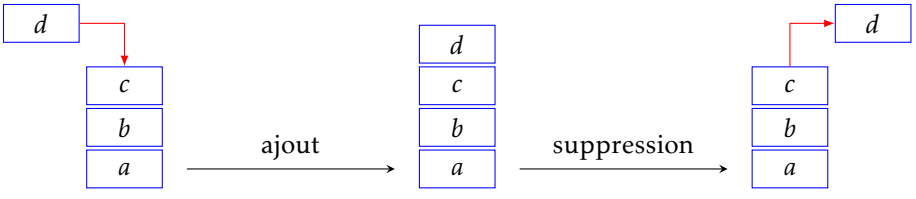
Lors du script Python écrit plus haut, les paliers au delà desquels la liste est redimensionnée sont donc :

$$0, 4, 8, 16, 25, 35, 46, 58, 72, 88, 106, 126, 148, 173, 201, 233, 269, 309, 354, 405, 462, 526, 598, 679, 771, 874, \ldots$$

2.3 Piles et files

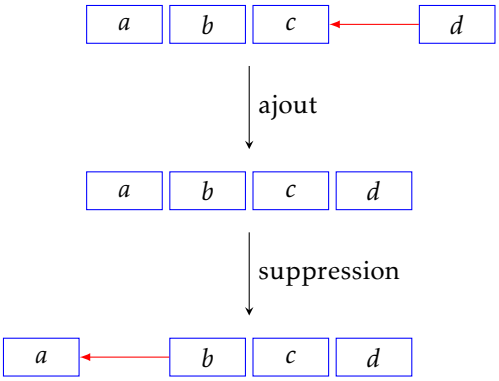
Nous allons terminer ce chapitre en nous intéressant à deux structures de données simples : les piles et les files. Il s'agit de structures linéaires dynamiques qui se distinguent par les conditions d'ajout et d'accès aux éléments :

– Les piles sont fondées sur le principe du « dernier arrivé, premier sorti » ; on les dit de type LIFO (*Last In, First Out*). C'est le principe même de la pile d'assiette : c'est la dernière assiette posée sur la pile d'assiettes sales qui sera la première lavée.



3. La complexité amortie d'un algorithme est son temps d'exécution moyen lorsqu'il est répété plusieurs fois de suite.

– Les files sont fondées sur le principe du « premier arrivé, premier sorti » ; on les dit de type FIFO (*First In, First Out*). C'est le principe de la file d'attente devant un guichet.



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.

Nous allons maintenant discuter pour chacune de ces deux structures la manière de les implémenter en PYTHON.

2.4 Implémentation pratique d'une pile

En PYTHON, définir une structure de donnée se fait par l'intermédiaire de la création d'une *classe*. Créer une classe, c'est définir un nouveau type d'objet avec ses constructeurs et ses méthodes propres. Dans le cas de la classe *Pile*, nous aurons un constructeur pour créer une pile vide, deux méthodes pour empiler et dépiler une pile, et une méthode pour déterminer si la pile est vide.

Compte tenu de ce qui a été dit au sujet de la classe *list*, le choix d'un objet de ce type pour représenter une pile est possible, à condition d'utiliser la méthode append pour empiler et `pop()` pour dépiler, ce qui garantit un coût amorti constant. Ceci conduit à la définition de la classe :

```
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()
```

On crée une nouvelle pile en exécutant la commande `p = Pile()` (on dit qu'on crée une nouvelle *instance* de la classe *Pile*). Lors de cet appel, c'est toujours la méthode `__init__` qui est exécutée ; dans le cas présent, cette méthode attribue à l'objet créé une liste vide nommée `lst`. On notera que traditionnellement l'objet créé (l'instance de classe) est nommé `self` dans les définitions des différentes méthodes. Il doit apparaître en premier argument de celles-ci mais n'apparaitra pas lors de l'utilisation de ces dernières. Par exemple, pour empiler un objet `x` dans la pile `p` on écrira simplement `p.push(x)` et pour dépiler on écrira `p.pop()`.

La figure 6 illustre la création d'une pile, l'empilement des entiers de 1 à 10, puis le dépilement de ces mêmes valeurs (on verifiera au passage de la règle LIFO est bien respectée).

```
p = Pile()
for i in range(1, 11):
    p.push(i)
while not p.empty():
    print(p.pop(), end=' - ')
print()
```

```
10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 -
```

Figure 6 – Un exemple d'utilisation de la classe *Pile*.

Une définition alternative

Il est aussi possible de créer une classe *Pile* sans s'appuyer sur une classe existante (ici la classe *list*), mais la démarche est plus délicate. Nous allons utiliser la notion de liste chaînée décrite à la section 2.1 : il faudra tout d'abord définir une première classe créant un objet (une cellule) constitué de deux attributs : une valeur et un pointeur (vers une autre cellule), ce qui nous permettra ensuite de définir une pile comme une liste chaînée de cellules. La pile vide se distinguera par son attribut égal à la constante `None`.

La nouvelle définition de la classe *Pile* est la suivante :

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

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
```

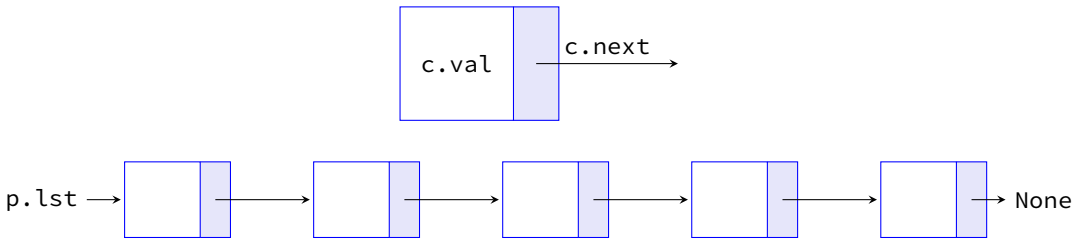


Figure 7 – Une instance c de la classe *Cell* et une instance p de la classe *Pile*.

On vérifiera que le script de la figure 6 s'exécute exactement de la même façon avec cette nouvelle définition de la classe *Pile* (si ce n'est peut-être en terme de performance). C'est une des grandes forces de la programmation orientée objet : à partir du moment où sont précisées la liste des attributs et des méthodes associées à l'objet, ceux-ci peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il y ait risque d'interférence. ce résultat est obtenu grâce au concept d'*encapsulation* : le fonctionnement interne de l'objet est en quelque sorte enfermé dans l'objet lui-même.

2.5 Implémentation pratique d'une file

Utiliser un objet de la classe *list* pose ici un problème : si l'ajout d'un élément dans la file peut se faire à coût constant à l'aide de la méthode append, le retrait doit se faire à l'aide de la méthode `pop(0)` et induit un coût linéaire. Nous allons donc rejeter cette solution pour en proposer une autre, certe limitée car nous allons devoir donner une limite supérieure à la taille de la liste, mais qui en contrepartie proposera des méthode d'ajout et de suppression de coût constant.

Utilisation d'un tableau de taille fixe
L'idée principale est de maintenir 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. Enfin, on notera que le tableau sera rempli de façon circulaire : l'indice de la tête `t` ne sera pas nécessairement inférieur à celui de la queue `q`.

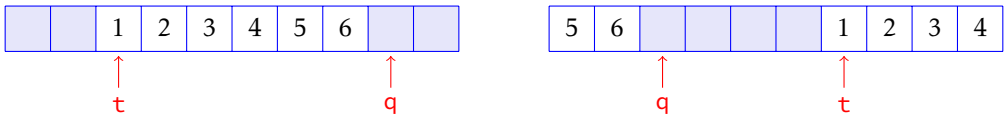


Figure 8 – Deux positions possible de la même file dans le tableau.

L'inconvénient de ce mode de représentation est qu'il est *a priori* impossible de distinguer une file vide d'une file pleine : dans les deux cas on aura `q = t`. Une solution (ce n'est pas la seule) consiste à ne jamais remplir la dernière case du tableau ; de la sorte l'égalité `q = t` caractérise la file vide, et $q = t - 1 \bmod n$ la file pleine (n désignant la taille du tableau).

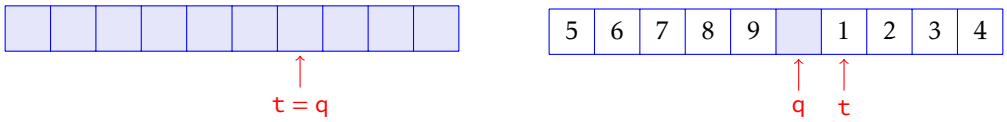


Figure 9 – Une file vide et une file pleine

La réalisation concrète de cette classe prend la forme suivante :

```
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

    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
```

Le script de la figure 10 illustre la création d'une file ainsi que l'ajout et le retrait suivant la règle FIFO.

```
f = File(20)
for i in range(1, 11):
    f.add(i)
while not f.empty():
    print(f.take(), end=' - ')
print()
```

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
```

FIGURE 10 – Un exemple d'utilisation de la classe *File*.

Bien entendu d'autres solutions existent, par exemple en utilisant une liste doublement chaînée, ou encore une liste circulaire (voir l'exercice 4 à ce sujet). Si on veille à respecter les spécifications du type, ce script fonctionnera à l'identique, indépendamment de la mise en œuvre choisie.

3. Exercices

Exercice 1 Implémenter la classe *Pile* à l'aide d'un tableau de taille fixe.

Exercice 2 Implémenter la classe *File* à l'aide de deux instances de la classe *Pile*.
Analyser le temps d'exécution des opérations `add` et `take`.

Exercice 3 Implémenter la classe *Pile* à l'aide de deux instances de la classe *File*.
Analyser le temps d'exécution des opérations `push` et `pop`.

Exercice 4 Dans une liste doublement chaînée, chaque cellule est un objet constitué de trois attributs : une valeur et deux pointeurs vers la cellule qui le précède et qui lui succède (illustration figure 11). Proposer une implémentation de la classe *File* à l'aide d'une liste doublement chaînée.

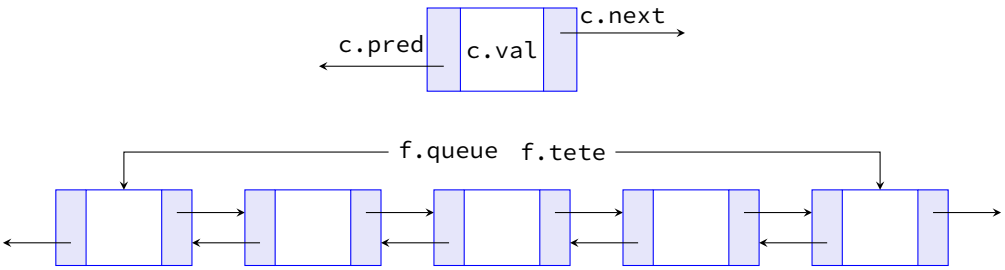
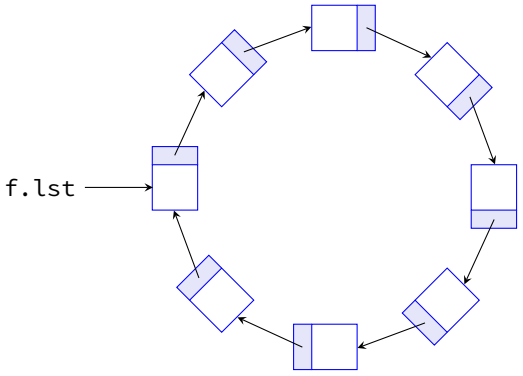


FIGURE 11 – Une file représentée par une liste doublement chaînée.

Exercice 5 Une liste circulaire est une liste chaînée dont le dernier élément pointe vers le premier :



Utiliser une liste circulaire pour implémenter la classe *File*.

Exercice 6 **Évaluation d'une expression postfixe.**
La notation *postfixe* d'une expression algébrique consiste à placer les opérateurs après son ou ses opérandes. Par exemple, l'addition de a et de b sera écrite "a b +" en notation postfixe, la racine carrée de a sera écrite "a $\sqrt{}$". L'intérêt majeur de cette notation est qu'une expression postfixe n'est jamais ambiguë : alors que expression infixe "1 + 2 × 3" peut avoir deux significations : "(1 + 2) × 3" ou "1 + (2 × 3)", ce n'est jamais le cas d'une expression postfixe, ce qui rend l'usage des parenthèses superflu : "1 2 + 3 ×" ne peut être compris que de cette façon : "(1 2 +) 3 ×" et "1 2 3 × +" de cette façon : "1 (2 3 ×) +".

Nous allons montrer comment, à l'aide d'une pile, on peut évaluer une expression algébrique postfixe.

Dans cet exercice, les expressions algébriques seront représentée par les listes qui pourront contenir des nombres (de type *int* ou *float*) ou des chaînes de caractères représentant des opérateurs unaires ou binaires (comme par exemple `'sqrt'` ou `'+'`).

Par exemple, l'expression $\frac{1+2\sqrt{3}}{4}$ sera représentée par la liste `[1, 2, 3, 'sqrt', '*', '+', 4, '/']`.

On suppose donné deux dictionnaires répertoriant pour l'un les opérateurs unaires, pour l'autre les opérateurs binaires, et qui associent à chaque chaîne de caractère la fonction correspondante. On peut par exemple définir ces deux dictionnaires à l'aide du script suivant, et les compléter en suivant le même modèle :

```
from numpy import sqrt, exp, log

op_uni = {'sqrt': sqrt, 'exp': exp, 'ln':log}

def add(x, y):
    return x + y

def sous(x, y):
    return x - y

def mult(x, y):
    return x * y

def div(x, y):
    return x / y

op_bin = {'+': add, '-': sous, '*': mult, '/': div}
```

L'évaluation d'une expression postfixe consiste à utiliser une pile initialement vide et à parcourir les éléments de la liste représentant l'expression à évaluer en appliquant les règles suivantes :

- si l'élément est un nombre, il est empilé ;
- si l'élément est un opérateur unaire, le sommet de la pile est dépilé, l'opérateur lui est appliqué et le résultat ré-empilé ;
- si l'élément est un opérateur binaire, deux éléments de la pile sont dépilés, l'opérateur leur est appliqué et le résultat ré-empilé.

Si l'expression postfixe est correcte sur le plan syntaxique (et mathématique), à la fin du traitement de la liste la pile ne contient plus qu'un seul élément égal au résultat de l'évaluation de l'expression.

On suppose donnée une implémentation de la classe *Pile* ainsi que les deux dictionnaires `op_uni` et `op_bin`. Rédiger une fonction qui évalue une expression postfixe donnée sous forme de liste. Dans un premier temps, on pourra supposer que l'expression est syntaxiquement correcte, puis rédiger une seconde fonction d'évaluation qui détecte les erreurs de syntaxe.