Tableaux extensibles pour implémenter `List`
Insertions et suppressions en $O(n)$
Conséquence sur le temps d'insertion de deux stratégies d'extension : additive et multiplicative
Temps amorti d'une opération
La stratégie de `ArrayList` en Java

# Voir le code...

**`List.java`**
**`ArrayList.java`**

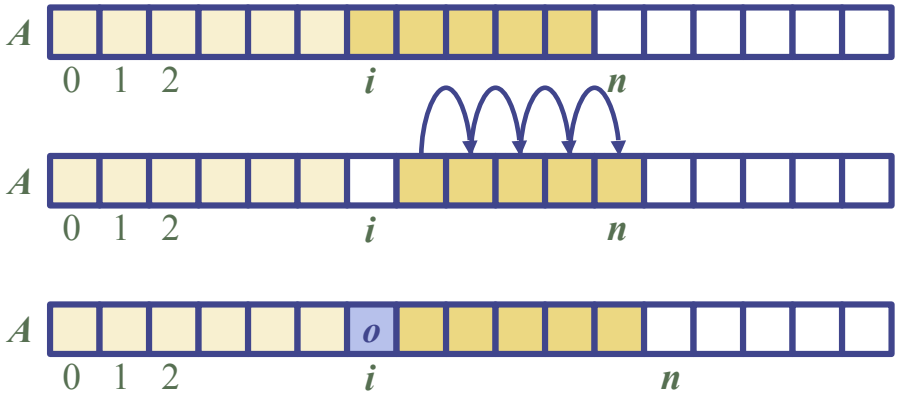
# Insertion

Dans une opération **add**( i, o ), nous devons faire de la place pour le nouvel élément en décalant vers l'avant les n - i éléments `A[i], ..., A[n-1]`

Dans le meilleur cas, i = n; 0 décalage, $O(1)$

Dans le pire des cas, $i = 0$ ; n décalages, $O(n)$

En moyenne, $\frac{\sum_{i=1}^{n} i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$, $O(n)$



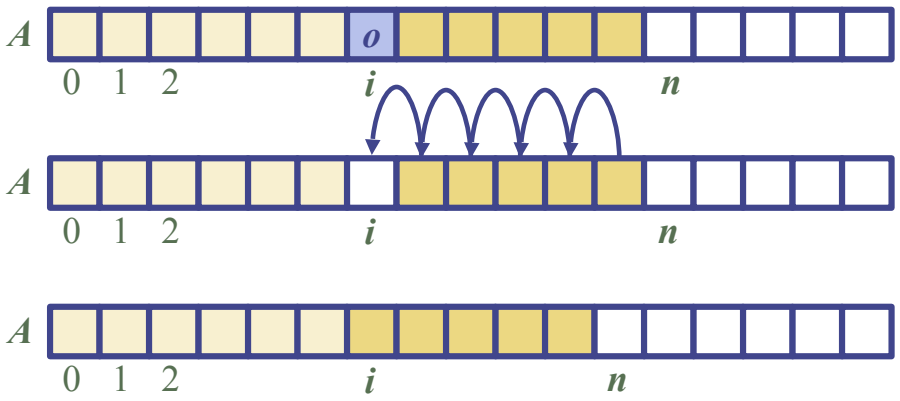
# Suppression

Dans une opération **remove( i )**, nous devons remplir le trou laissé par l'élément retiré en déplaçant vers l'arrière les $n - i - 1$ éléments `A[i + 1], ..., A[n - 1]`

Dans le meilleur cas, $i = n\text{-}1$; $0$ décalage, $O(1)$

Dans le pire des cas, $i = 0$ ; $n\text{-}1$ décalages, $O(n)$

En moyenne, $\frac{\sum_{i=1}^{n-1} i}{n} = \frac{n(n-1)}{2n} = \frac{n-1}{2}$, $O(n)$



# Performances

Dans une implémentation de l'interface `List` basée sur un tableau dynamique :

- L'espace utilisé par la structure de données est dans $O(n)$
- L'indexation d'un élément prend un temps dans $O(1)$
- **add( i, e )**, **addFirst()**, **removeFirst()**, **remove( i )** et **remove( e )** sont dans $O(n)$ en pire cas

- Dans une opération **add** lorsque le tableau est plein, au lieu de lancer une exception, on remplace le tableau actuel par un autre plus grand ...

# `List` implémentée avec un tableau extensible

Dans une opération `addLast( o )`, par exemple (sans index), lorsque le tableau est plein nous le remplaçons par un autre plus grand.

Quelle devrait être la taille du nouveau tableau ?

- Stratégie additive : augmenter la taille par une constante $c$
- Stratégie géométrique : multiplier la taille par une constante $c$

Que devient le coût pour une insertion à la fin d'un tableau plein ?

# Temps amorti dans un ADT

Le **temps amorti** fait référence au temps moyen nécessaire pour effectuer une opération dans une séquence d'opérations sur une structure de données. Contrairement à la complexité en pire des cas, qui se concentre sur l'opération la plus coûteuse, l'analyse amortie fournit une vue plus réaliste des performances en calculant la moyenne du temps sur une série d'opérations.

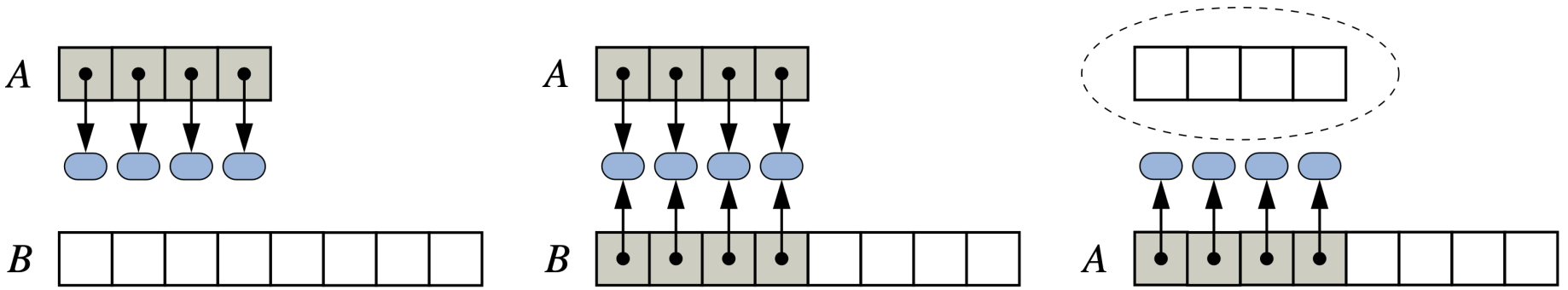
L'analyse amortie est particulièrement utile lorsque, dans une structure de données, une opération coûteuse est occasionnellement nécessaire (comme le redimensionnement du tableau ici), mais cette opération coûteuse ne se produit pas assez souvent pour avoir un impact significatif sur le temps moyen de chaque opération.

# Redimensionnement d'un tableau

Supposons que nous commençons avec un tableau dynamique vide et que nous effectuons une séquence de $n$ insertions :

- **Opérations d'insertion** : La plupart du temps, insérer un élément (à la fin du tableau) prend $O(1)$.
- **Opérations de redimensionnement** : Occasionnellement, lorsque le tableau est plein, nous devons le redimensionner, ce qui prend $O(n)$.

L'idée de l'analyse amortie est que le coût du redimensionnement peut être réparti sur de nombreuses insertions. Le redimensionnement est coûteux mais il ne se produit pas fréquemment pour dominer le coût total.



# Comparaison des 2 stratégies

Nous comparons la stratégie additive et la stratégie géométrique en analysant le coût total $T(n)$ nécessaire pour effectuer une série de $n$ opérations **`add( o )`**

Au départ, nous supposons une liste vide représentée par un tableau de taille 1

Le temps amorti d'une opération **`add`** est le temps moyen par ajout sur une série de n opérations, c'est-à-dire $T(n) / n$

# Analyse de la stratégie **additive** (on ajoute c espaces)

Pour $n$ **add** et des extensions de $c$ espaces lorsque nécessaire, nous avons :

- **n** insertions.
- ***k*** extensions de $c$ espaces, où $k = n / c$, nécessitant de recopier

  1 élément à la 1ère extension

  c+1 éléments à la 2ème extension

  2c+1 éléments à la 3ème extension

  ...

  (k-1)c+1 éléments à la $k$ème extension.

- Si on enlève les +1 (il y en a **k** au total), il reste :

  $c + 2c + \ldots + (k-1)c = c(1 + 2 + \ldots + k-1) =$ ***ck(k-1)/2***.

Le temps total $T(n)$ pour effectuer une série de $n$ **add** est donc proportionnel à :

***n + k + ck(k-1)/2***

$T(n)$ est dans $O(k^2)$, c'est-à-dire $O(n^2)$, puisque $c$ est une constante

Le temps amorti d'une opération **add** est donc dans $O(n)$

# Analyse de la stratégie **géométrique** (on double l'espace)

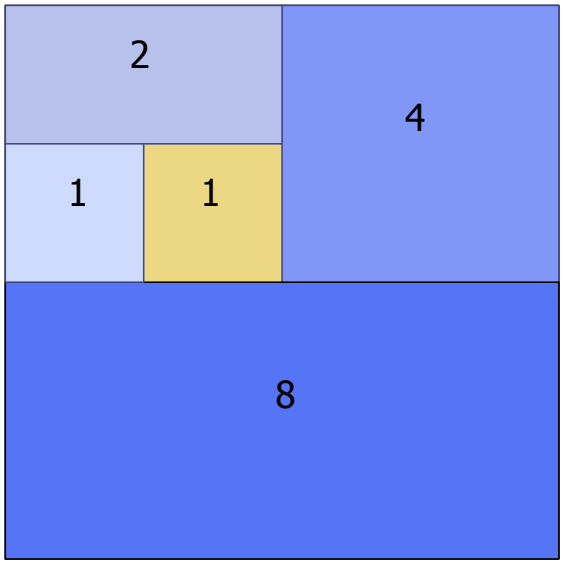
Pour n **add** en doublant l'espace lorsque le tableau est plein, nous avons :

***n*** insertions

**k** = $log_2 n$ extensions nécessitant de recopier

- 1 élément à la 1ère extension
- 2 éléments à la 2ème extension
- 4 éléments à la 3ème extension
- ...
- $2^{k-1}$ éléments à la kème extension

Série géométrique




Le temps total $T(n)$ pour effectuer une série de $n$ **add** est donc proportionnel à :

$$1 + 2 + 4 + \ldots + 2^{k-1}$$

$$\sum_{i=0}^{k-1} 2^i$$

$2^k - 1$, comme $k = log_2 n = n - 1$

$T(n)$ est donc dans $O(n)$

Le temps amorti d'une opération **add** est donc dans **$O(1)$**

# `ArrayList` en Java

En Java, la classe `ArrayList` utilise une politique d'extension multiplicative de 50 % chaque fois qu'elle manque d'espace.

**Capacité Initiale** : Lorsque vous créez une ArrayList, vous pouvez spécifier une capacité initiale. Si vous ne spécifiez pas de capacité, la capacité initiale par défaut est de 10.

Chaque fois que vous ajoutez un élément à l'ArrayList, elle vérifie s'il y a suffisamment d'espace dans le tableau actuel pour accueillir le nouvel élément.

Si le tableau est plein, l'ArrayList augmente sa capacitéde de 1.5 fois l'ancienne capacité (soit une augmentation de 50 %).

1. Une fois la nouvelle capacité déterminée, un nouveau tableau de cette taille est alloué, et les éléments de l'ancien tableau sont copiés dans le nouveau.
2. Le nouvel élément est ajouté au tableau nouvellement redimensionné.

Prenons un exemple où la capacité initiale est de 10 :

- Capacité : 10
- Ajout de 10 éléments : aucun redimensionnement nécessaire.
- Ajout du 11e élément :
- Nouvelle Capacité = 10 + (10/2) = 10 + 5 = 15.
- Ajout du 16e élément :
- Nouvelle Capacité = 15 + (15/2) = 15 + 7 = 22.
- Ajout du 23e élément :
- Nouvelle Capacité = 22 + (22/2) = 22 + 11 = 33.

**Pourquoi une Augmentation de 50 % ?**

Le choix d'une augmentation de 50 % équilibre l'efficacité de la mémoire et les performances. Une augmentation plus grande (par exemple, doubler la taille) gaspillerait plus de mémoire si la liste croît lentement ou si de nombreux éléments sont supprimés. Une augmentation plus petite (par exemple, ajouter un nombre fixe d'emplacements) entraînerait des redimensionnements plus fréquents, ce qui pourrait dégrader les performances.

# En Python ?

**Allocation initiale** :

- Python alloue un bloc de mémoire pour un petit nombre d'éléments, même si la liste est vide.

**Stratégie d'augmentation** :

- À mesure d'insertions, Python ajoute une quantité d'espace sans attendre d'en avoir besoin !

```
  0 élément   56 octets (initiale)
  1 élément   88 octets (1.57x)
  5 éléments 120 octets (1.36x)
  9 éléments 184 octets (1.53x)
 17 éléments 248 octets (1.35x)
 25 éléments 312 octets (1.26x)
 33 éléments 376 octets (1.21x)
 41 éléments 472 octets (1.55x)
```

- L'algorithme est complexe et peut varier en fonction de l'implémentation spécifique de l'interpréteur (CPython, PyPy, etc). Généralement, il augmente la capacité de manière à minimiser à la fois le gaspillage de mémoire et les opérations coûteuses de redimensionnement.
- Python utilise un **facteur d'augmentation qui varie en fonction de la taille** de la liste.
  - La croissance peut être plus agressive pour de petites listes (par exemple, augmentation de 50%).
  - La croissance devient plus modérée pour de grandes listes, pour éviter de gaspiller de mémoire.
- L'objectif est d'avoir un compromis optimal entre la mémoire utilisée et le nombre d'opérations de redimensionnement.

**Temps amorti** :

- Malgré cette stratégie sophistiquée de redimensionnement, Python offre un **temps amorti dans O(1)** pour les opérations d'ajout, similaire à Java, mais avec une gestion de la mémoire souvent plus efficace dans les scénarios où les listes subissent de nombreuses insertions et suppressions.

**Nombre d'éléments dans une liste versus sa capacité** :

- Allez voir le code de `ExtensionList.py` dans `/src/main/python` pour voir comment avoir la capacité et voir la progression de la liste en fonction du nombre d'éléments.

# Remarques conclusives

- Pour un tableau extensible (Dynamic Array), on a regardé deux politiques d'extension et montré que l'ajout d'un espace constant mène à un coût amorti dans $O(n)$, alors que l'ajout d'un espace par un multiple mène à un coût amorti dans $O(1)$.
- L'implémentation de ArrayList utilise la méthode `resize` pour changer la capacité du tableau de 2x.
- Avant de remplacer le tableau, il est possible de voir s'il est possible d'étendre l'espace alloué pour couvrir l'augmentation demandée. C'est plus rapide et enlève le travail de recopiage => $O(1)$. Cependant, le pire cas (où on doit allouer un nouveau tableau) reste en $O(n)$.
- En Java, l'implémentation de `ArrayList` utilise une extension de 50% (1.5x).
- En Python, c'est plus complexe !