Type Abstrait de Données (ADT)
Séquences basées sur un tableau
Tableaux extensibles
Implémentations

Type Abstrait de Données (ADT)

(Abstract Data Type)

Un type de données abstrait (ADT) est une abstraction d'une structure de données

Un ADT spécifie :

- Données
- Opérations sur les données
- Conditions d'erreur associées aux opérations

(*Exemple*)

Les **entiers** sont un ADT, défini pour prendre les valeurs : …, –2, –1, 0, 1, 2, …,
et effectuer les opérations d'addition, de soustraction, de multiplication, de division, de comparaison etc.

Condition d'erreur, par exemple division par 0.

Ils se comporte selon des mathématiques familières, ***indépendamment de la façon dont ils sont représentés*** dans un ordinateur.

ADT

Les ADT simplifient et encouragent la décomposition et la modularité

Les ADT permettent de découpler l'implémentation et l'utilisation

- Par exemple, une fois l'interface de l'ADT connu, plusieurs implémentations différentes sont possibles
- Changer l'implémentation de l'ADT n'affecte pas les programmes d'applications qui l'utilisent

ADT *Liste*

Une Liste spécifie :

- Une collection ordonnée d'éléments
- Des opérations pour :
  - créer une Liste
  - obtenir le nombre d'éléments
  - convertir en chaîne de caractères
  - ajouter
  - retirer
  - retrouver des éléments
  - etc.
- La détection des erreurs provoquées lorsqu'on tente de retirer un élément introuvable (e.g. qui n'est pas dans la liste ou dont l'indice n'existe pas)

```
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "23 mars 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT Liste
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

#ADT List (interface)
class List:

    # constructeur
    def __init__( self ):
        pass

    # retourne le nombre d'éléments
    def __len__( self ):
        pass

    # produit une chaîne de caractères:
    # les éléments entre crochets
    # séparés par des virgules
    # taille et capacité de la structure de données
    # indiquées lorsque pertinent
    def __str__( self ):
        pass

    # ajouter un élément à la fin de la liste
    def append( self, element ):
        pass

    # retirer le kème élément
    def remove( self, k ):
        pass

    # trouver et retourner l'index
    # de l'élément passé en argument
    # s'il est dans la liste, None sinon
    def find( self, element ):
        pass
```

Classes Python pour des séquences

Python a des types intégrés pour des séquences : ***list***, ***tuple*** et ***str***.

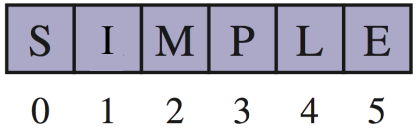
Chacun de ces types de séquences prend en charge l'indexation pour accéder à un élément individuel, en utilisant la syntaxe *A[i]*, où *A* est la variable qui réfère à la séquence et *i* l'index de l'élément.

Chacun de ces types utilise un tableau comme structure de données pour représenter la séquence.

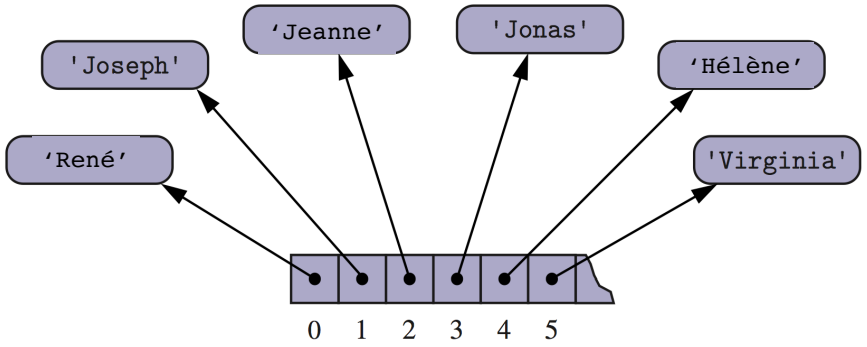
Un tableau est un ensemble d'emplacements en mémoire qui peuvent être adressés en utilisant des index consécutifs qui commencent à 0.

Tableaux de caractères ou de références à des objets

Un tableau peut stocker des éléments primitifs, tels que des caractères, ce qui donne un tableau compact :



Un tableau peut également stocker des références à des objets :



Tableaux compacts

La prise en charge principale des tableaux compacts se trouve dans un module nommé ***array***.

Ce module définit une classe, également nommée ***array***, fournissant un stockage compact pour les tableaux de types de données primitifs.

Le constructeur de la classe ***array*** requiert un code de type en tant que premier paramètre, qui est un caractère désignant le type de données qui sera stocké dans le tableau.

```
from array import array
premiers = array( 'i', [2,3,5,7,11,13,17,19] )
```

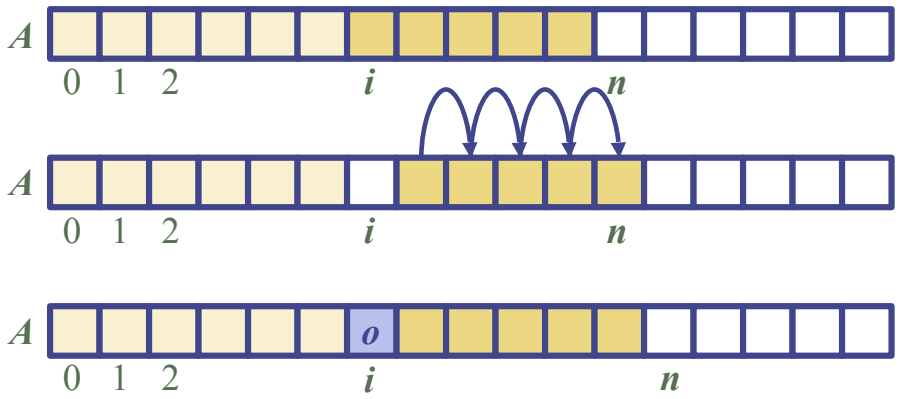
Codes pour les types dans la classe ***array***

Code	C Data Type	Typical Number of Bytes
`'b'`	signed char	1
`'B'`	unsigned char	1
`'u'`	Unicode char	2 or 4
`'h'`	signed short int	2
`'H'`	unsigned short int	2
`'i'`	signed int	2 or 4
`'I'`	unsigned int	2 or 4
`'l'`	signed long int	4
`'L'`	unsigned long int	4
`'f'`	float	4
`'d'`	float	8

Insertion

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

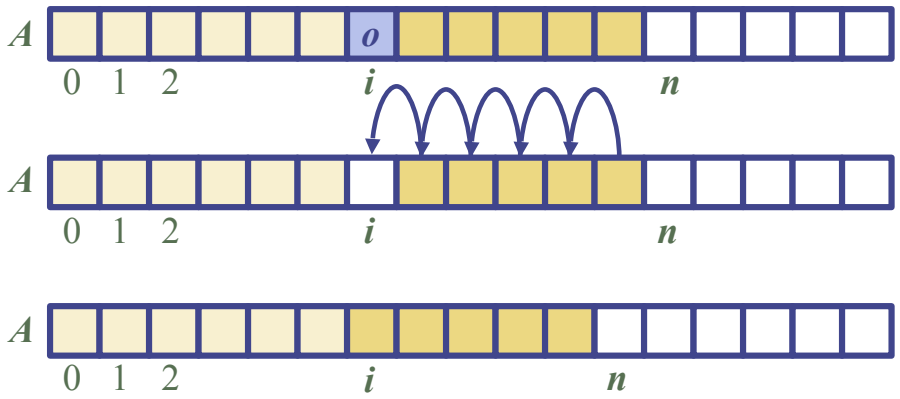
Dans le meilleur cas, i = n-1; 0 décalage, O(1)
Dans le pire des cas, *i = 0 ;* n décalages, O(n)
En moyenne, $\frac{\sum_{i=1}^{n} i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$, O(n)



Deletion

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-1; 0 décalage, O(1)
Dans le pire des cas, *i = 0 ;* n-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 du type *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*** et ***remove*** 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 par un plus grand ...

Liste implémenté avec un tableau extensible

Dans une opération ***add***(*o*) (sans index, ***append***), nous ajoutons l'élément à la fin

Lorsque le tableau est plein, nous remplaçons le tableau par un 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*

```
Algorithm add( o )
  if n = S.length then
    A = new array of
        size …
    for i = 0 to n–1 do
      A[i] = S[i]
    S = A
  S[n] = o
  n = n + 1
```

Comparaison des 2 stratégies

Nous comparons la stratégie additive et la stratégie multiplicative en analysant le temps 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

Nous appelons le temps amorti d'une opération ***add*** le temps moyen pris par un ajout sur cette série d'opérations, c'est-à-dire *T(n) / n*

Analyse de la stratégie additive

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 la 1ère fois
  - c+1 éléments la 2ème fois
  - 2c+1 éléments la 3ème fois
  - …
  - (k-1)c+1 éléments la kème fois.
- Si on enlève les **k** fois +1, il reste :
  - c + 2c + … + (k-1)c = c(1 + 2 + … + 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 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 additive (c = 2)

Pour, *n = 16 et c = 2 ; k = 8*

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

16 + 8 + 2 x 8 x 7 / 2 = ***80***

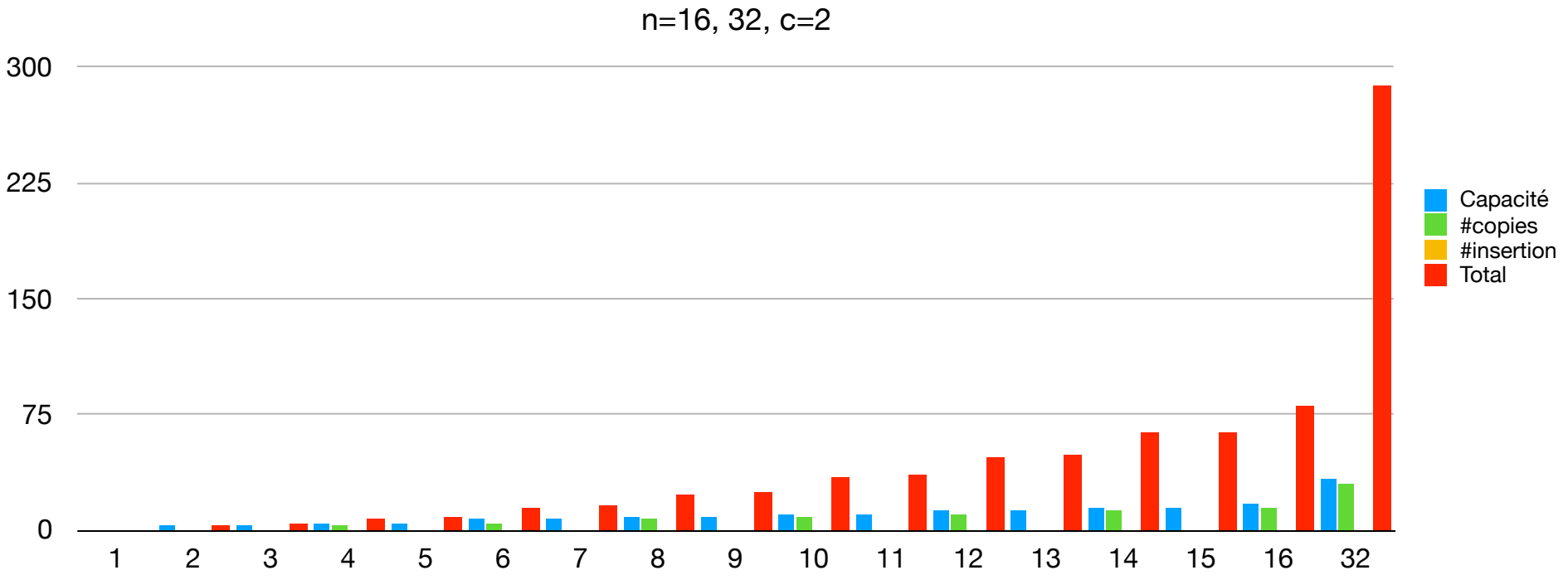
donc on aura un temps amorti proportionnel à *80/16 =* ***5*** opérations par ***add***

Pour, n = 32 et c = 2 ; k = 16

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

32 + 16 + 2 x 16 x 15 / 2 = ***288***

donc on aura un temps amorti proportionnel à *288/32 =* ***9*** opérations par ***add***



Analyse de la stratégie additive (c = 4)

Pour, *n = 16 et c = 4 ; k = 4*

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

16 + 4 + 4 x 4 x 3 / 2 = ***44***

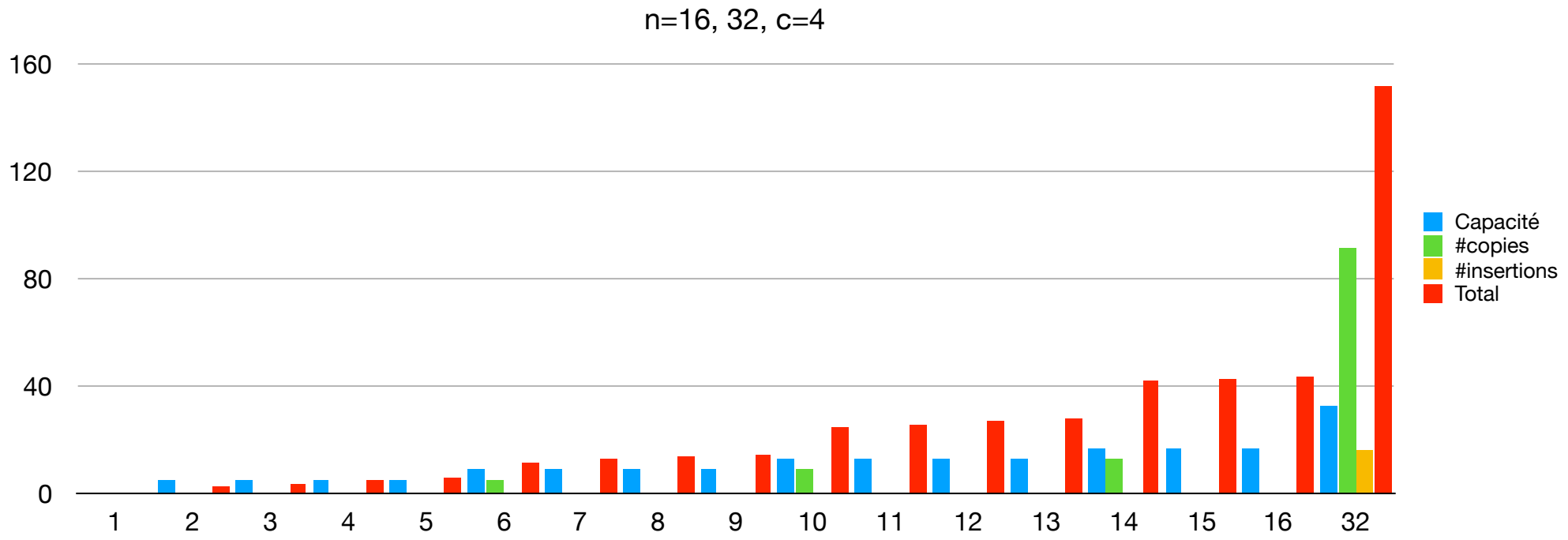
donc un temps amorti proportionnel à ***2.75*** opérations par ***add***

Pour, n = 32 et c = 4 ; k = 8

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

32 + 8 + 4 x 8 x 7 / 2= ***152***

donc un temps amorti proportionnel à 152/*32* = ***4.75*** opérations par ***add***

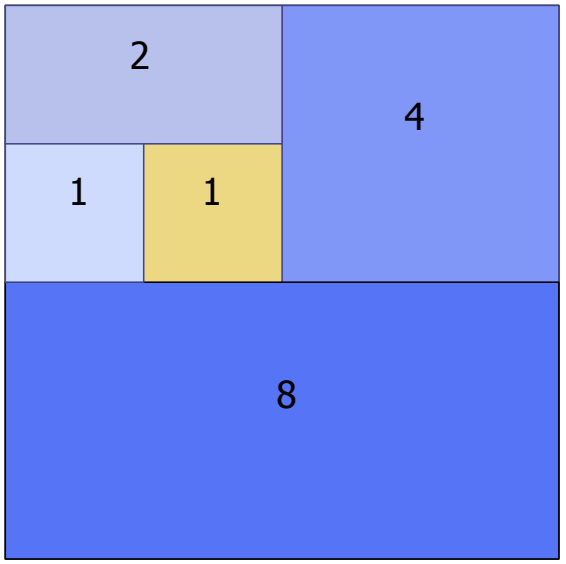


Analyse de la stratégie géométrique (*c* = 2)

Pour *n* ***add*** et des doublages lorsque nécessaire, nous avons :

n insertions
k = $log_2 n$ extensions nécessitant de recopier
- 1 élément la 1ère fois
- 2 éléments la 2ème fois
- 4 éléments la 3ème fois
- ...
- 2^{k-1} éléments la kème fois

Série géométrique



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

$$\boldsymbol{n} + \underline{1 + 2 + 4 + \ldots + 2^{k-1}} =$$
$$n + 2^k - 1 =$$
$$\boldsymbol{2n - 1}$$

T(n) est donc dans O(*n*)
Le temps amorti d'une opération ***add*** est donc dans O(*1*)

Analyse de la stratégie géométrique

Pour, *n = 16 ; k = 4*

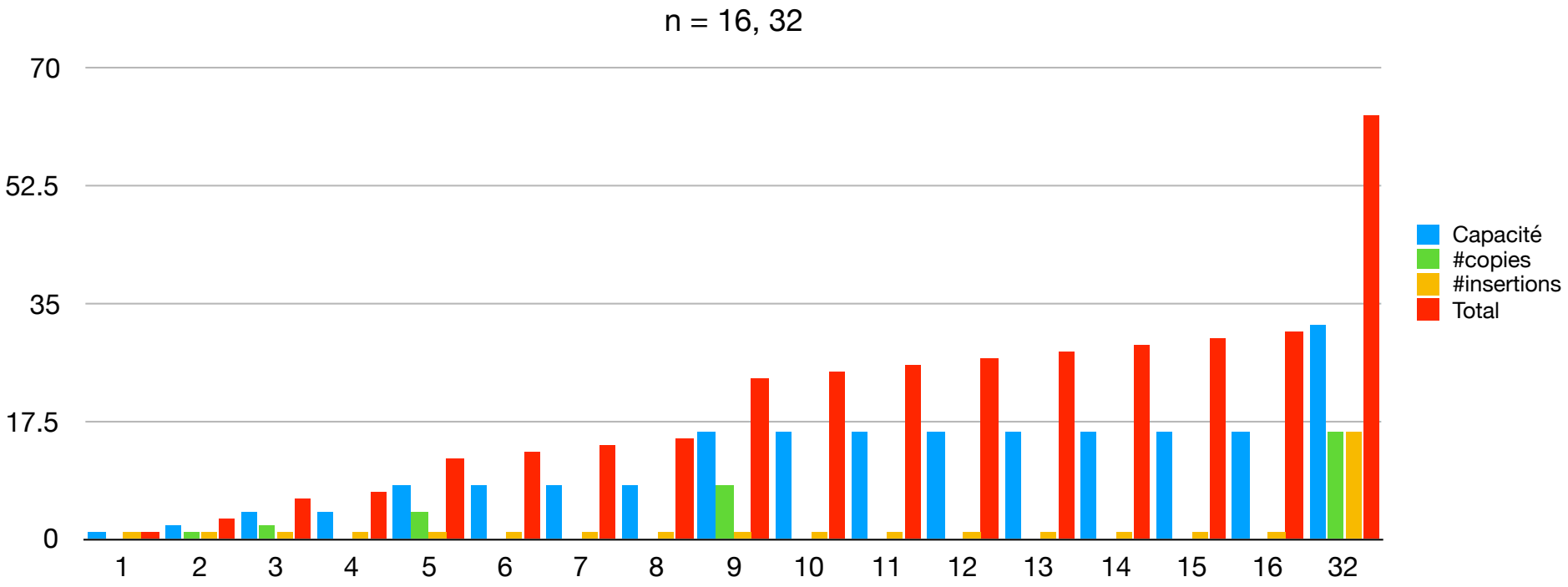
2n - 1 = ***31***

donc on aura un temps amorti proportionnel à 31/*16* = ***1.94*** opérations par ***add***

Pour, n = 32 ; k = 5

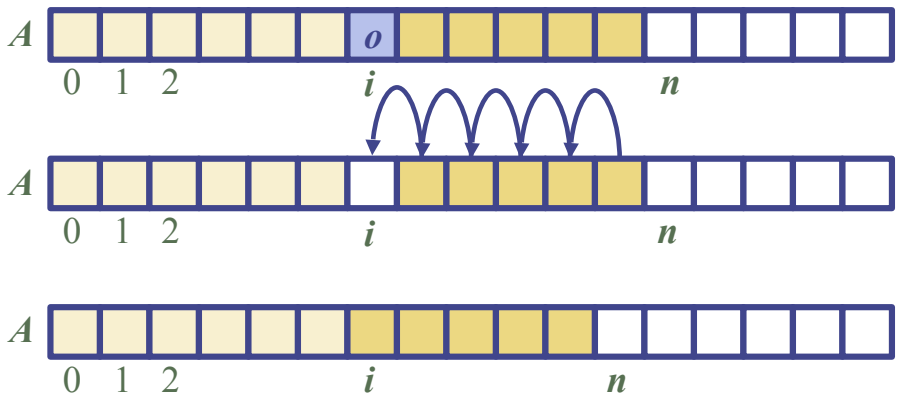
2n - 1 = ***63***

donc on aura un temps amorti proportionnel à *63/32* = ***1.97*** opérations par ***add***

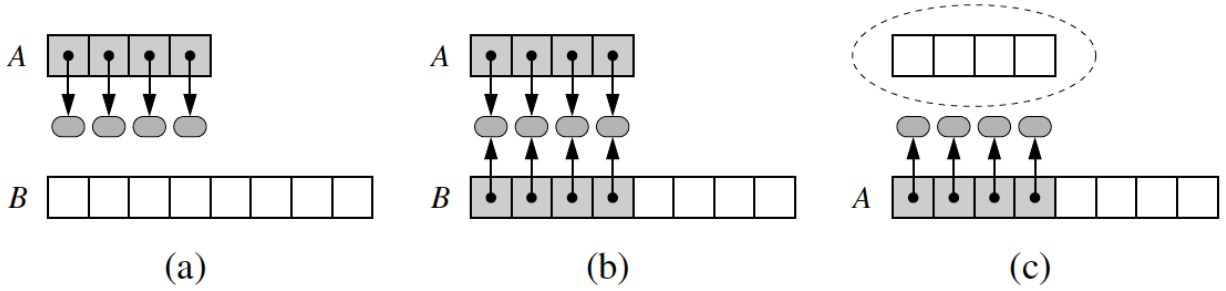


```
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "9 février 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT List
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

# Module pour créer des tableaux dynamiques
# en utilisant une stratégie géométrique
# en temps amorti dans O(1)
import ctypes

class DynamicArray:

    # retourne un pointeur sur une zone mémoire
    # pouvant stocker c objets Python contigus
    def _makeArray( self, c ):
        return( c * ctypes.py_object )()

    # constructeur
    # on commence avec 1 élément
    def __init__( self ):
        # nombre d'éléments dans le tableau
        self._n = 0
        # capacité : nombre d'éléments maximum possible
        self._capacity = 1
        # référence au tableau
        self._A = self._makeArray( self._capacity )

    # convertir un tableau en chaînes de caractères
    # utilisant les crochets pour délimiter le tableau
    # des virgules pour séparer les éléments
    # indiquant la capacité courante du tableau
    def __str__( self ):
        if self._n == 0:
            return "[](size = 0; capacity = " + str( self._capacity ) + ")"
        pp = "[" + str( self._A[0] )
        for k in range( 1, self._n ):
            pp += ", " + str( self._A[k] )
        pp += "](size = " + str( self._n )
        pp += "; capacity = " + str( self._capacity ) + ")"
        return pp

    # retourne le nombre d'éléments dans le tableau
    def __len__( self ):
        return self._n

    # retourne la capacité courante
    def capacity( self ):
        return self._capacity

    # retourne l'élément à l'index k
    # notation avec crochets
    def __getitem__( self, k ):
        if not 0 <= k < self._n:
            raise IndexError( 'index out of bounds' )
        return self._A[k]
```

```
# retourne l'élément à l'index k
def get( self, k ):
    if not 0 <= k < self._n:
        raise IndexError( 'index out of bounds' )
    return self._A[k]

# ajoute à la fin du tableau
def append( self, obj ):
    # si le tableau est plein
    if self._n == self._capacity:
        # on double sa capacité
        self._resize( 2 * self._capacity )
    # on ajoute le nouvel élément à la fin du tableau
    self._A[self._n] = obj
    # on incrémente le nombre d'éléments
    self._n += 1

#remove and return the ith element of the list
def remove( self, i ):
    # on vérifie si i est un index valide
    if not 0 <= i < self._n:
        raise IndexError( 'index out of bounds' )
    obj = self._A[i]
    # on décale les éléments i+1 à n-1
    for k in range( i+1, self._n ):
        self._A[k-1] = self._A[k]
    # on décrémente le nombre d'éléments
    self._n -= 1
    self._A[self._n] = None #avoid loitering
    return obj

#retourne et retire le dernier élément du tableau
def pop( self ):
    # si le tableau est vide, on ne peut pas retirer d'élément
    if self._n == 0:
        raise IndexError( 'index out of bounds' )
    else:
        # on garde une référence à l'objet à retourner
        obj = self._A[self._n - 1]
        # on met la valeur du dernier élément à None
        # pour libérer la mémoire (garbage collection)
        self._A[self._n - 1] = None # avoid loitering
    # on décrémente le nombre d'éléments
    self._n -= 1
    # si l'occupation descend au quart ou moins
    if self._n <= self._capacity / 4:
        # on réduit la capacité de 2 fois
        self._resize( self._capacity // 2 )
    return obj

# trouve et retourne l'index d'un élément si dans la liste
# None sinon
def find( self, obj ):
    # on parcourt les éléments de la liste
    for k in range( self._n ):
        # si l'élément est trouvé
        if self._A[k] == obj:
            # on retourne son index
            return k
    # ici l'élément n'a pas été trouvé, on retourne None
    return None

# redimensionne le tableau à capacité c
def _resize( self, c ):
    # on crée un nouveau tableau de capacité c
    B = self._makeArray( c )
    # on copie les éléments de l'ancien tableau dans le nouveau
    for k in range( self._n ):
        B[k] = self._A[k]
    # on garde la référence sur le nouveau tableau
    self._A = B
    # on met sa capacité à c
    self._capacity = c
```



Suppression d'un élément à l'index *i* d'un tableau.



Trois étapes pour "agrandir" un tableau dynamiquement

(a) créer un nouveau tableau B ;
(b) copier les éléments de A dans B ;
(c) réaffecter la référence A au nouveau tableau B. La collecte des ordures va ramasser l'espace alloué du tableau A (représenté par l'ovale en pointillé). L'insertion du nouvel élément n'est pas indiqué.

```
"""unit testing
"""
if __name__ == '__main__':

    data = DynamicArray()
    print( data )

    data.append( 'titi' )
    print( "append( 'titi' )" )
    print( data )
    data.append( 'toto' )
    print( "append( 'toto' )" )
    print( data )
    data.append( 'tata' )
    print( "append( 'tata' )" )
    print( data )
    data.append( 'lastit' )
    print( data )

    idx = data.find( 'titi' )
    if idx is not None:
        print( "found titi ranked", idx )
    else:
        print( "titi not found" )
    idx = data.find( 'cece' )
    if idx is not None:
        print( "found cece ranked", idx )
    else:
        print( "cece not found" )

    print( "remove( 0 ) = ", data.remove( 0 ) )
    print( data )
    print( "remove( 1 ) = ", data.remove( 1 ) )
    print( data )
    print( "remove( 0 ) = ", data.remove( 0 ) )
    print( data )
    print( "pop() = ", data.pop() )
```

```
[](size = 0; capacity = 1)
append( 'titi' )
[titi](size = 1; capacity = 1)
append( 'toto' )
[titi, toto](size = 2; capacity = 2)
append( 'tata' )
[titi, toto, tata](size = 3; capacity = 4)
[titi, toto, tata, lastit](size = 4; capacity = 4)
found titi ranked 0
cece not found
remove( 0 ) =  titi
[toto, tata, lastit](size = 3; capacity = 4)
remove( 1 ) =  tata
[toto, lastit](size = 2; capacity = 4)
remove( 0 ) =  toto
[lastit](size = 1; capacity = 4)
pop() =  lastit
```

```python
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "23 mars 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT List
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

# List implémentée avec un tableau dynamique
# On a besoin de DynamicArray
from DynamicArray import DynamicArray
# et de l'interface de l'ADT List
from List import List

# la classe ArrayList hérite de l'interface List
class ArrayList( List ):

    # implémente l'ADT List (List.py)
    # utilise la classe DynamicArray (DynamicArray.py)
    def __init__( self ):
        self._A = DynamicArray()

    # retourne le nombre d'éléments
    def __len__( self ):
        return len( self._A )

    # retourne une chaîne de caractères représentant la liste
    def __str__( self ):
        return str( self._A )

    # accès avec notation des crochets
    def __getitem__( self, k ):
        return self._A[k]

    # ajoute l'élément obj à la fin de la liste
    def append( self, obj ):
        self._A.append( obj )

    # retire le ième élément de la liste
    def remove( self, i ):
        return self._A.remove( i )

    # retourne l'index de l'élément obj s'il existe
    def find( self, obj ):
        return self._A.find( obj )
```

```
"""unit testing
"""
if __name__ == '__main__':

    data = ArrayList()
    print( data )

    data.append( 'titi' )
    data.append( 'toto' )
    data.append( 'tata' )
    print( data )

    idx = data.find( 'titi' )
    if idx is not None:
        print( "found titi ranked", idx )
    else:
        print( "titi not found" )
    idx = data.find( 'cece' )
    if idx is not None:
        print( "found cece ranked", idx )
    else:
        print( "cece not found" )

    print( "remove 0 =", data.remove( 0 ) )
    print( "new size = ", str( len( data ) ) )
    print( data )
    print( "remove 1 = ", data.remove( 1 ) )
    print( data )
    print( "remove 0 = ", data.remove( 0 ) )
    print( data )
```

```
[](size = 0; capacity = 1)
[titi, toto, tata](size = 3; capacity = 4)
found titi ranked 0
cece not found
remove 0 = titi
new size =  2
[toto, tata](size = 2; capacity = 4)
remove 1 =  tata
[toto](size = 1; capacity = 4)
remove 0 =  toto
[](size = 0; capacity = 4)
```

Stratégie en Python

```
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "28 janvier 2018"
#
# Programme Python pour IFT2015/Types abstraits
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

# utilise la fonction getsize de sys
import sys

data = [ ]
for k in range( 32 ):
    a = len(data) # nombre d'éléments
    b = sys.getsizeof(data) # taille en bytes
    print( 'Taille: {0:3d}; Capacité en bytes: {1:4d}'.format(a, b))
    data.append( None ) # ajout de 1 élément
```

```
Taille:   0; Capacité en bytes:   64
Taille:   1; Capacité en bytes:   96
Taille:   2; Capacité en bytes:   96
Taille:   3; Capacité en bytes:   96
Taille:   4; Capacité en bytes:   96
Taille:   5; Capacité en bytes:  128
Taille:   6; Capacité en bytes:  128
Taille:   7; Capacité en bytes:  128
Taille:   8; Capacité en bytes:  128
Taille:   9; Capacité en bytes:  192
Taille:  10; Capacité en bytes:  192
Taille:  11; Capacité en bytes:  192
Taille:  12; Capacité en bytes:  192
Taille:  13; Capacité en bytes:  192
Taille:  14; Capacité en bytes:  192
Taille:  15; Capacité en bytes:  192
Taille:  16; Capacité en bytes:  192
Taille:  17; Capacité en bytes:  264
Taille:  18; Capacité en bytes:  264
Taille:  19; Capacité en bytes:  264
Taille:  20; Capacité en bytes:  264
Taille:  21; Capacité en bytes:  264
Taille:  22; Capacité en bytes:  264
Taille:  23; Capacité en bytes:  264
Taille:  24; Capacité en bytes:  264
Taille:  25; Capacité en bytes:  264
Taille:  26; Capacité en bytes:  344
Taille:  27; Capacité en bytes:  344
Taille:  28; Capacité en bytes:  344
Taille:  29; Capacité en bytes:  344
Taille:  30; Capacité en bytes:  344
Taille:  31; Capacité en bytes:  344
```

```c
/* Ensure ob_item has room for at least newsize elements, and set
 * ob_size to newsize.  If newsize > ob_size on entry, the content
 * of the new slots at exit is undefined heap trash; it's the caller's
 * responsibility to overwrite them with sane values.
 * The number of allocated elements may grow, shrink, or stay the same.
 * Failure is impossible if newsize <= self.allocated on entry, although
 * that partly relies on an assumption that the system realloc() never
 * fails when passed a number of bytes <= the number of bytes last
 * allocated (the C standard doesn't guarantee this, but it's hard to
 * imagine a realloc implementation where it wouldn't be true).
 * Note that self->ob_item may change, and even if newsize is less
 * than ob_size on entry.
 */
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
    PyObject **items;
    size_t new_allocated, num_allocated_bytes;
    Py_ssize_t allocated = self->allocated;

    /* Bypass realloc() when a previous overallocation is large enough
       to accommodate the newsize.  If the newsize falls lower than half
       the allocated size, then proceed with the realloc() to shrink the list.
    */
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SIZE(self) = newsize;
        return 0;
    }
```

```
    /* This over-allocates proportional to the list size, making room
     * for additional growth.  The over-allocation is mild, but is
     * enough to give linear-time amortized behavior over a long
     * sequence of appends() in the presence of a poorly-performing
     * system realloc().
     * The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
     * Note: new_allocated won't overflow because the largest possible value
     *       is PY_SSIZE_T_MAX * (9 / 8) + 6 which always fits in a size_t.
     */
    new_allocated = (size_t)newsize + (newsize >> 3) + (newsize < 9 ? 3 : 6);
    if (new_allocated > (size_t)PY_SSIZE_T_MAX / sizeof(PyObject *)) {
        PyErr_NoMemory();
        return -1;
    }

    if (newsize == 0)
        new_allocated = 0;
    num_allocated_bytes = new_allocated * sizeof(PyObject *);
    items = (PyObject **)PyMem_Realloc(self->ob_item, num_allocated_bytes);
    if (items == NULL) {
        PyErr_NoMemory();
        return -1;
    }
    self->ob_item = items;
    Py_SIZE(self) = newsize;
    self->allocated = new_allocated;
    return 0;
}
```

Le "pattern" de croissance correspond à : +4, +4, +8, +9, +10, +11, +12, +14, +16, ...

On peut montrer qu'elle donne un temps amorti pour l'opération "add" en temps dans O(1).

Le copiage des données n'est pas systématique à chaque élongation. Si l'espace est suffisent, le nouvel élément est ajouté. Sinon, une fonction en C, `PyMem_Realloc`, vérifie d'abord si le bloc de mémoire alloué à la *list* peut être étendu pour accommoder la nouvelle taille de la *list* telle que décrite par le "pattern" de croissance. Si oui, le bloc est allongé et aucun élément ne bouge sauf celui qui est ajouté.

L'élongation d'un bloc mémoire ne peut être garantie par le système. Dans le cas où la demande d'élongation n'est pas possible (le pire cas), un nouveau bloc de la taille demandée est alloué et les données y sont copiées.

realloc

Defined in header <stdlib.h>

```
void *realloc( void *ptr, size_t new_size );
```

Reallocates the given area of memory. It must be previously allocated by `malloc()`, `calloc()` or `realloc()` and not yet freed with a call to `free` or `realloc`. Otherwise, the results are undefined.

The reallocation is done by either:

a) expanding or contracting the existing area pointed to by `ptr`, if possible. The contents of the area remain unchanged up to the lesser of the new and old sizes. If the area is expanded, the contents of the new part of the array are undefined.

b) allocating a new memory block of size `new_size` bytes, copying memory area with size equal the lesser of the new and the old sizes, and freeing the old block.

Le pire cas est décrit par la situation (b), lorsqu'un nouveau bloc doit être alloué et la zone mémoire copiée.