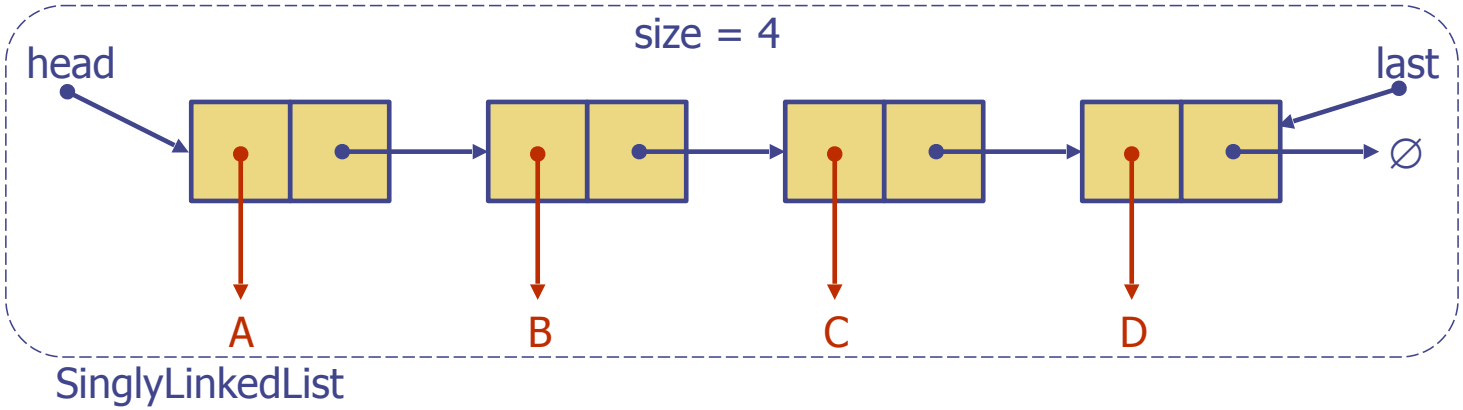
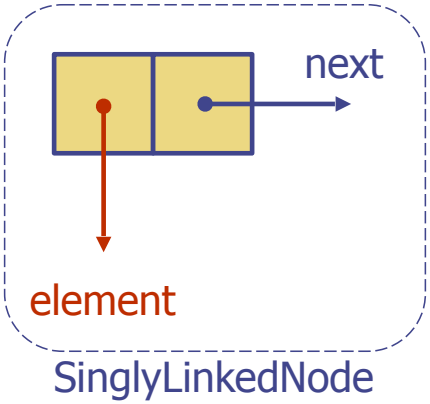
Listes simplement chaînées
Noeud simplement chaîné
Opérations sur des listes simplement chaînées
SinglyLinkedList
Listes doublement chaînées
Noeud doublement chaîné
Opérations sur les listes doublement chaînées
DoublyLinkedList
Deque
Listes positionnelles
Avantages des listes basées sur un tableau
Avantages des listes basées sur un chaînage

Listes simplement chaînées

Une liste simplement chaînée est une structure de données concrète constituée d'une séquence de nœuds à partir d'une référence vers le noeud de tête, où chaque nœud contient deux références : vers un élément et vers le nœud suivant. On gardera aussi une référence sur le dernier nœud et un entier pour le nombre d'éléments dans la liste.

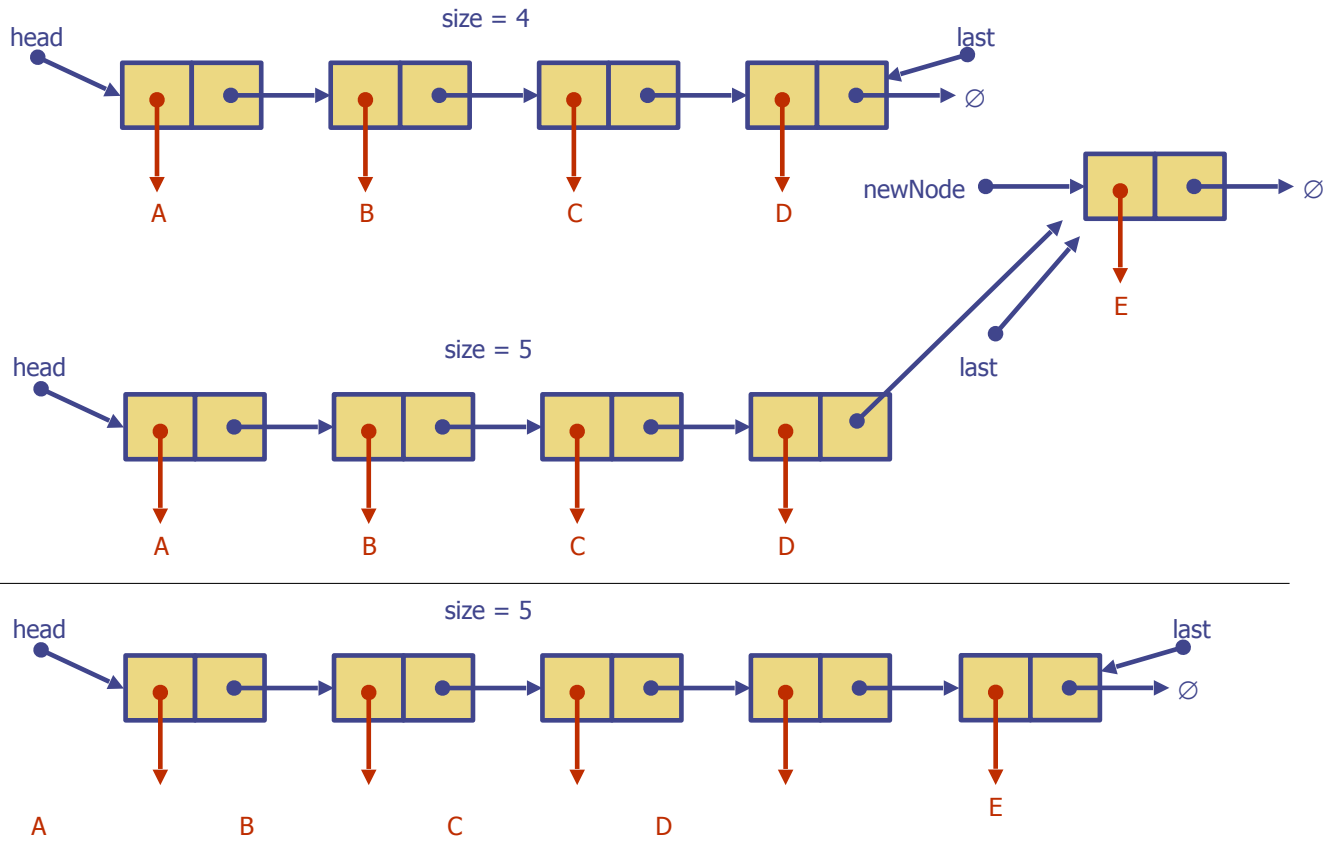




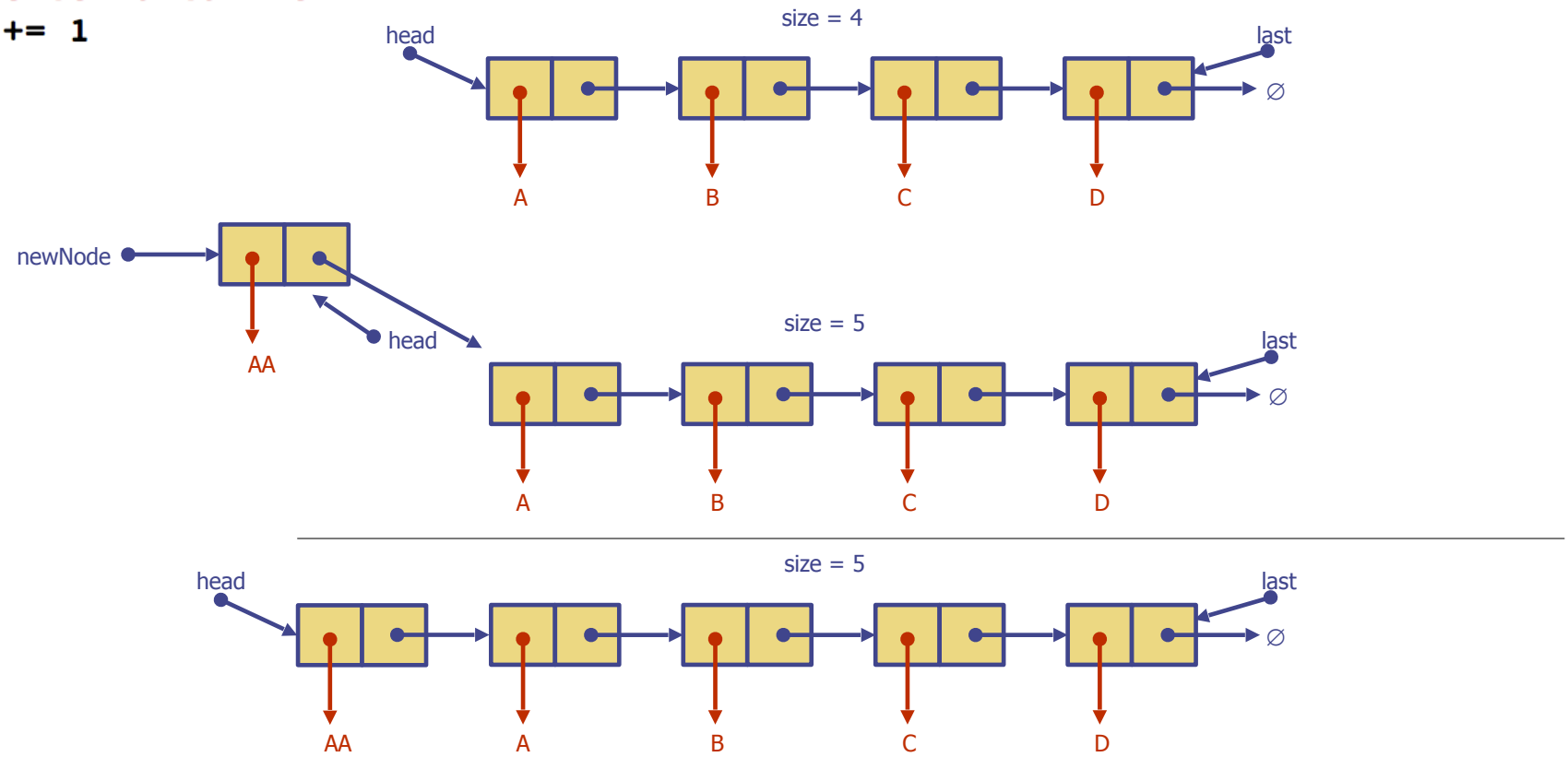
```
class SinglyLinkedNode:

    def __init__( self, element, next ):
        self.element = element # stockage d'un élément
        self.next = next       # référence au noeud suivant
```

```
from SinglyLinkedNode import SinglyLinkedNode
from List import List

class SinglyLinkedList( List ):

    def __init__( self ):
        self._head = None
        self._last = None
        self._size = 0

    def __len__( self ):
        return self._size

    def __str__( self ):
        if self.is_empty():
            return "[](size = 0)"
        else:
            pp = "["
            curr = self._head
            while curr != self._last:
                pp += str( curr.element ) + ", "
                curr = curr.next
            pp += str( curr.element ) + "]"
            pp += "(size = " + str( self._size ) + ")"
        return pp

    def is_empty( self ):
        return self._size == 0
```

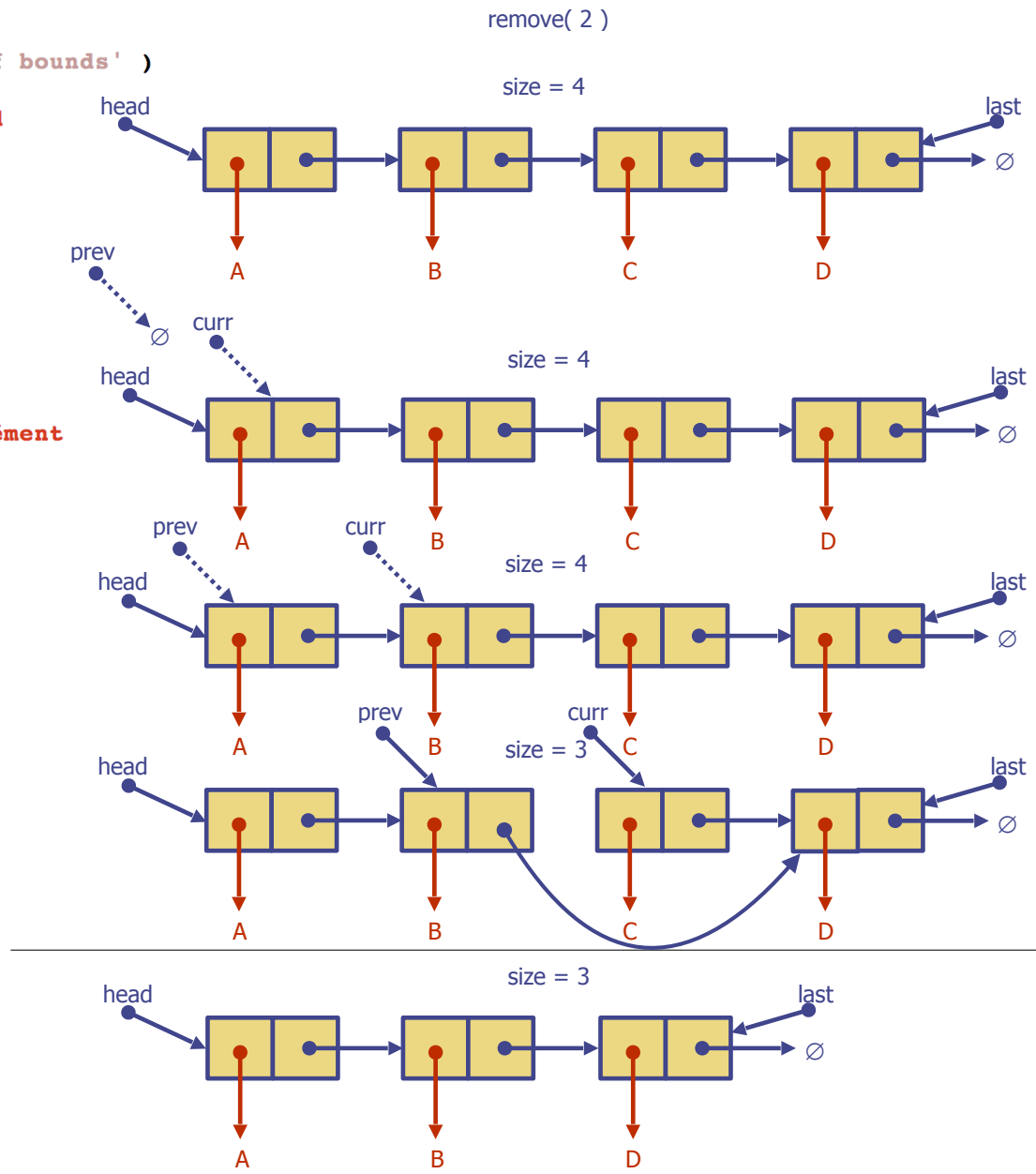
```
# ajoute un élément à la fin de la liste en O(1)
def append( self, element ):
    # on crée un nouveau noeud pour l'élément
    newNode = SinglyLinkedNode( element, None )
    # si la liste est vide
    # noeud unique, donc premier et dernier
    if self.is_empty():
        self._head = self._last = newNode
    # sinon le noeud devient le suivant du last
    # et le last de la liste
    else:
        self._last.next = newNode
        self._last = newNode
    # on incrémente la taille
    self._size += 1
```



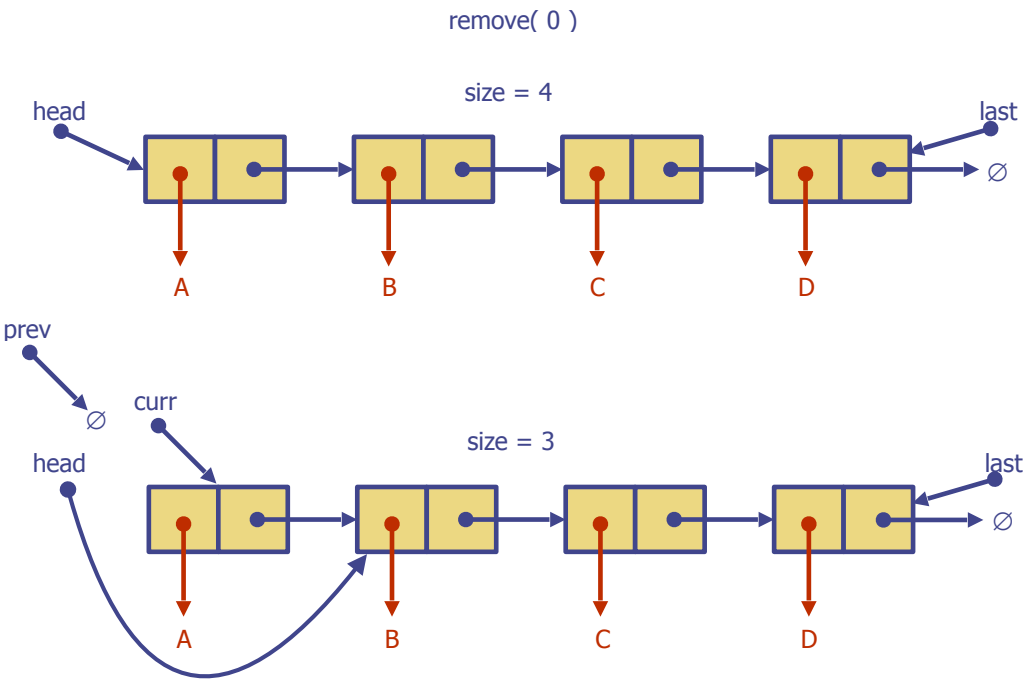
```
# ajoute un élément au début de la liste en O(1)
def insert( self, element ):
    # on crée un nouveau noeud pour l'élément
    newNode = SinglyLinkedNode( element, self._head )
    # si la liste est vide
    # noeud unique, donc dernier
    if self.is_empty():
        self._last = newNode
    # noeud devient le premier de la liste
    self._head = newNode
    # on incrémente la taille
    self._size += 1
```

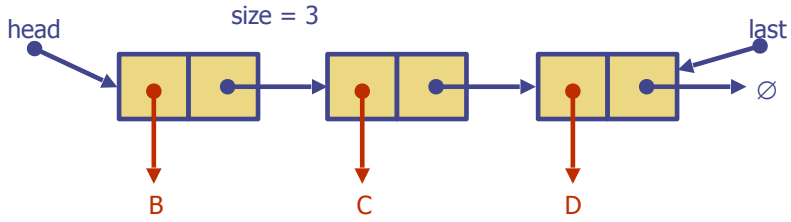


```
# retire l'élément à l'index k en O(n)
def remove( self, k ):
    # les indices de listes commencent à 0
    # on vérifie si le kème élément existe
    if not 0 <= k < self._size:
        raise IndexError( 'SinglyLinkedList: index out of bounds' )
    else:
        # on avance un pointeur, curr, vers le kème noeud
        # on commence au premier noeud
        # on garde le noeud précédent, prev,
        # qui sera utile pour compléter l'opération
        curr = self._head
        prev = None
        # on prend le noeud suivant k-1 fois
        # et fait suivre curr
        for i in range( k ):
            prev = curr
            curr = curr.next
        # si prev est None c'est qu'on est sur le 1er élément
        if prev == None:
            # on retire le premier élément
            # simplement en mettant head sur son suivant
            self._head = curr.next
        # sinon on ajuste le suivant du noeud précédent
        # sur le suivant du noeud courant
        else:
            prev.next = curr.next
        # on décrémente la taille
        self._size -= 1
        # si on a vidé la liste
        # last devient None
        if self._size == 0:
            self._last = None
        # si on a retiré le dernier élément
        # mais que la liste n'est pas vide
        # on met last sur le noeud prev
        if curr.next == None:
            self._last = prev
        # on retourne l'élément retiré
        return curr.element
```



```
# retire l'élément à l'index k en O(n)
def remove( self, k ):
    # les indices de listes commencent à 0
    # on vérifie si le kème élément existe
    if not 0 <= k < self._size:
        raise IndexError( 'SinglyLinkedList: index out of bounds' )
    else:
        # on avance un pointeur, curr, vers le kème noeud
        # on commence au premier noeud
        # on garde le noeud précédent, prev,
        # qui sera utile pour compléter l'opération
        curr = self._head
        prev = None
        # on prend le noeud suivant k-1 fois
        # et fait suivre curr
        for i in range( k ):
            prev = curr
            curr = curr.next
        # si prev est None c'est qu'on est sur le 1er élément
        if prev == None:
            # on retire le premier élément
            # simplement en mettant head sur son suivant
            self._head = curr.next
        # sinon on ajuste le suivant du noeud précédent
        # sur le suivant du noeud courant
        else:
            prev.next = curr.next
        # on décrémente la taille
        self._size -= 1
        # si on a vidé la liste
        # last devient None
        if self._size == 0:
            self._last = None
        # si on a retiré le dernier élément
        # mais que la liste n'est pas vide
        # on met last sur le noeud prev
        if curr.next == None:
            self._last = prev
        # on retourne l'élément retiré
        return curr.element
```





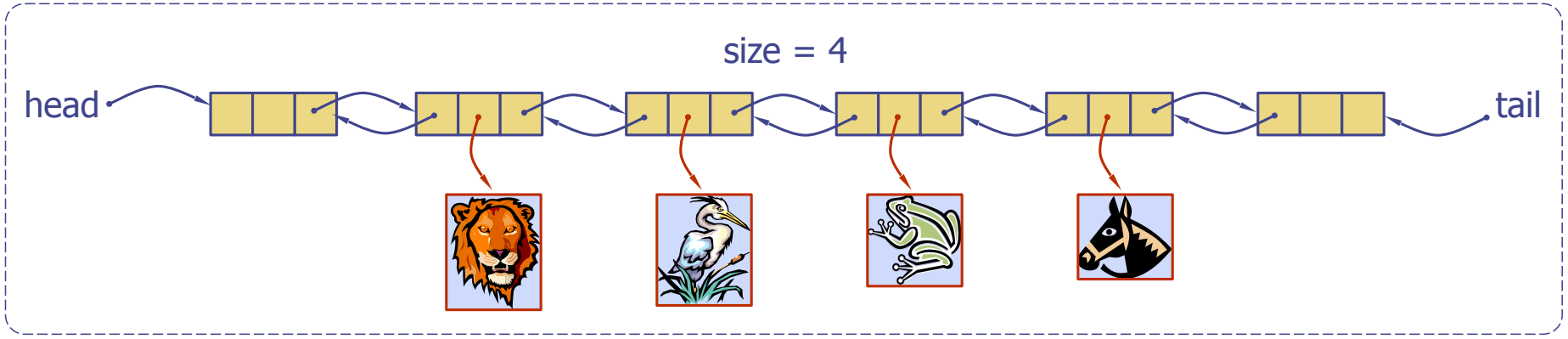
```
# retourne l'index d'un élément en O(n)
# ou None s'il n'est pas trouvé
def find( self, element ):
    # si la liste est vide l'élément n'y est pas.
    if self.is_empty():
        return None
    # sinon, on commence par la tête de liste
    # et on parcourt les éléments un à un
    # en utilisant les références au noeuds suivants
    else:
        curr = self._head
        for i in range( self._size ):
            if curr.element == element:
                return i
            else:
                curr = curr.next
    # si aucun index n'a été retourné
    # c'est que l'élément n'a pas été trouvé
    return None

# retourne le dernier élément si la liste n'est pas vide
def last( self ):
    if self.is_empty():
        return None
    else:
        return self._last.element

# retourne le premier élément si la liste n'est pas vide
def first( self ):
    if self.is_empty():
        return None
    else:
        return self._head.element
```

Performances des opérations d'une liste simplement chaînée

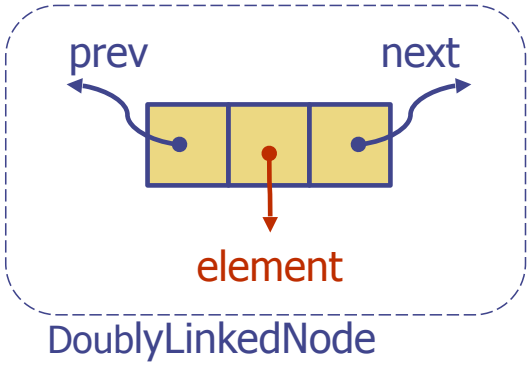
- L'espace utilisé pour *n* éléments est dans O(*n*)
- L'espace utilisé par chaque élément de la liste est dans O(1)
- Insérer un élément au début et à la fin se fait en temps dans O(1)
- Accéder au dernier et premier éléments se fait en temps dans O(1)
- Chercher, insérer et éliminer un élément quelconque se fait en temps dans O(n)

Listes doublement chaînées

Une liste doublement chaînée est une structure de données concrète constituée d'une séquence de nœuds à partir de références vers les nœuds de tête et de queue, où chaque nœud contient trois références : vers un élément, vers le nœud suivant et vers le nœud précédent. On gardera aussi un entier pour le nombre d'éléments dans la liste.



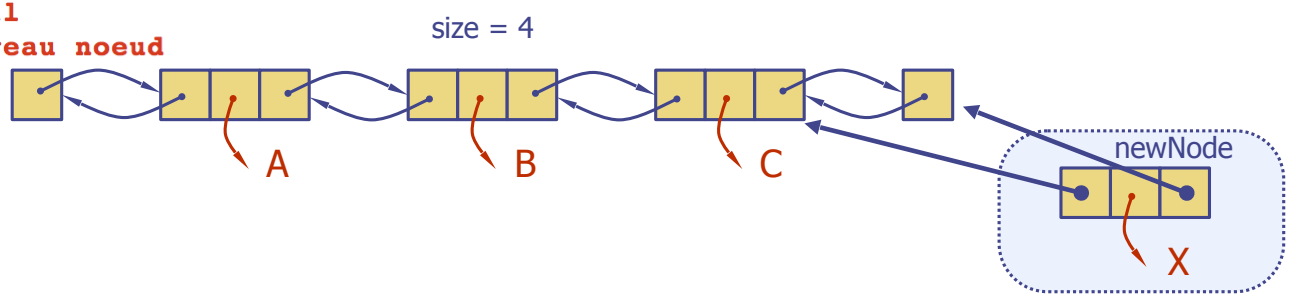
- Afin d'éviter certains cas particuliers en opérant sur les frontières d'une liste doublement chaînée, il est utile d'ajouter des nœuds spéciaux aux deux extrémités: un nœud de tête au début de la liste et un nœud de queue à la fin de la liste.
- Ces nœuds "factices" sont connus sous le nom de sentinelles (ou gardes), et ils ne stockent pas d'élément.

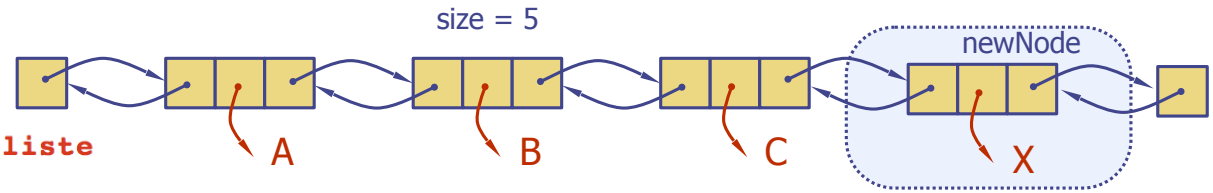


```
class DoublyLinkedNode:

    def __init__( self, element, prev, next ):
        self.element = element # stockage d'un élément
        self.prev = prev       # référence au noeud précédent
        self.next = next       # référence au noeud suivant
```

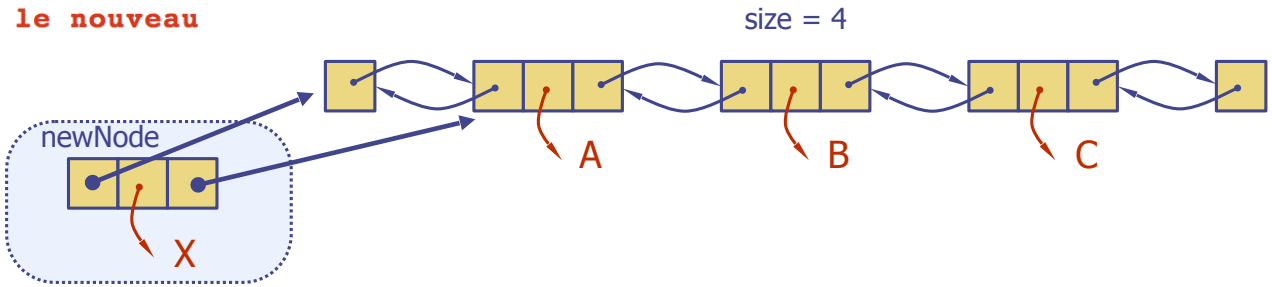
```
from DoublyLinkedNode import DoublyLinkedNode
from List import List

class DoublyLinkedList( List ):

    def __init__( self ):
        self._head = DoublyLinkedNode( None, None, None )
        self._tail = DoublyLinkedNode( None, None, None )
        self._head.next = self._tail
        self._tail.prev = self._head
        self._size = 0

    def __len__( self ):
        return self._size

    def __str__( self ):
        if self.is_empty():
            return "[](size = 0)"
        else:
            pp = "["
            curr = self._head.next
            while curr.next != self._tail:
                pp += str( curr.element ) + ", "
                curr = curr.next
            pp += str( curr.element ) + "]"
            pp += "(size = " + str( self._size ) + ")"
        return pp

    def is_empty( self ):
        return self._size == 0
```

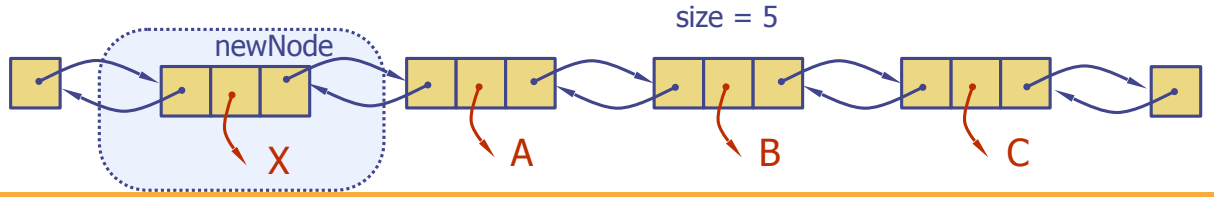
```
# ajoute un élément à la fin de la liste
def append( self, element ):
    # on crée un nouveau noeud pour l'élément à ajouter
    newNode = DoublyLinkedNode( element, self._tail.prev, self._tail )
    # le nouveau noeud sera le dernier
    # on met le suivant de l'ancien dernier sur le nouveau
    #   il se trouve avec le précédent de tail
    # on met le précédent de tail sur le nouveau noeud
    self._tail.prev.next = newNode
    self._tail.prev = newNode
    # on incrémente la taille
    self._size += 1
```






```
# insère un nouvel élément au début de la liste
def insert( self, element ):
    # on crée le nouveau noeud pour l'élément à ajouter
    newNode = DoublyLinkedNode( element, self._head, self._head.next )
    # le nouveau noeud devient le premier
    # on met le précédent de l'ancien premier sur le nouveau
    # on met le suivant de head sur le nouveau
    self._head.next.prev = newNode
    self._head.next = newNode
    # on incrémente la taille
    self._size += 1
```

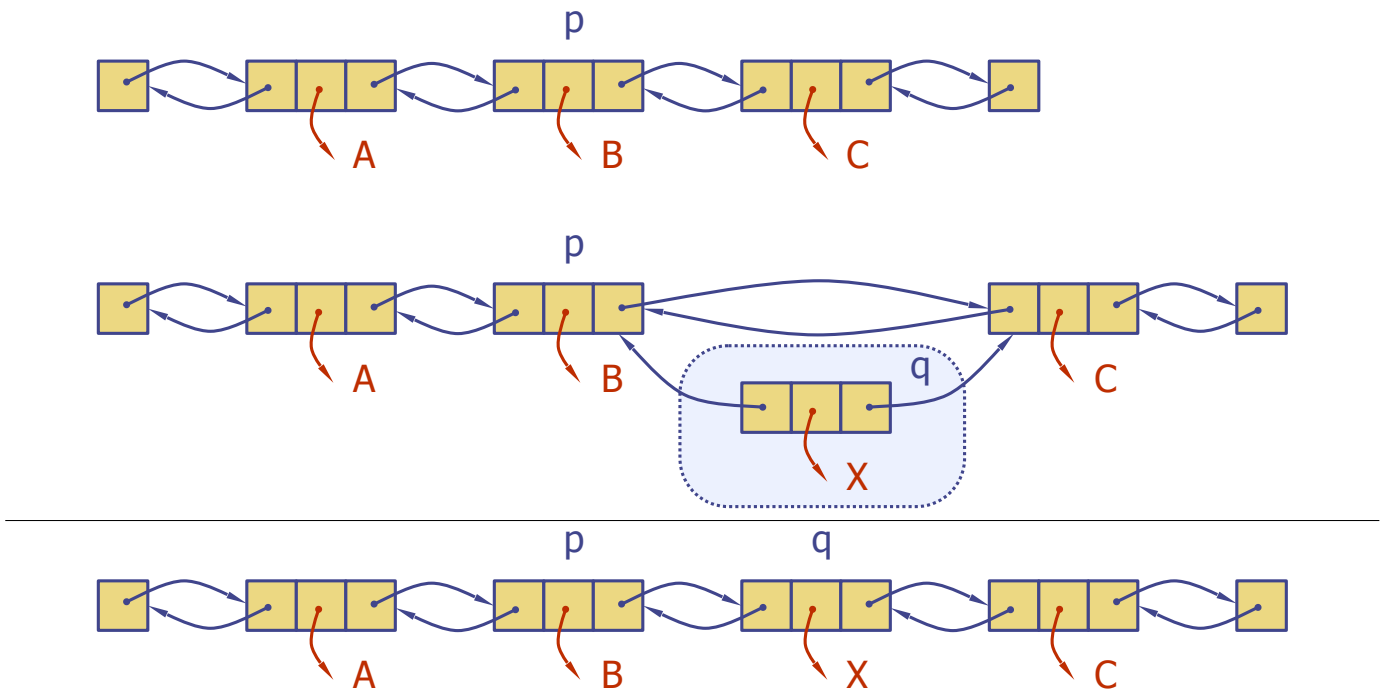




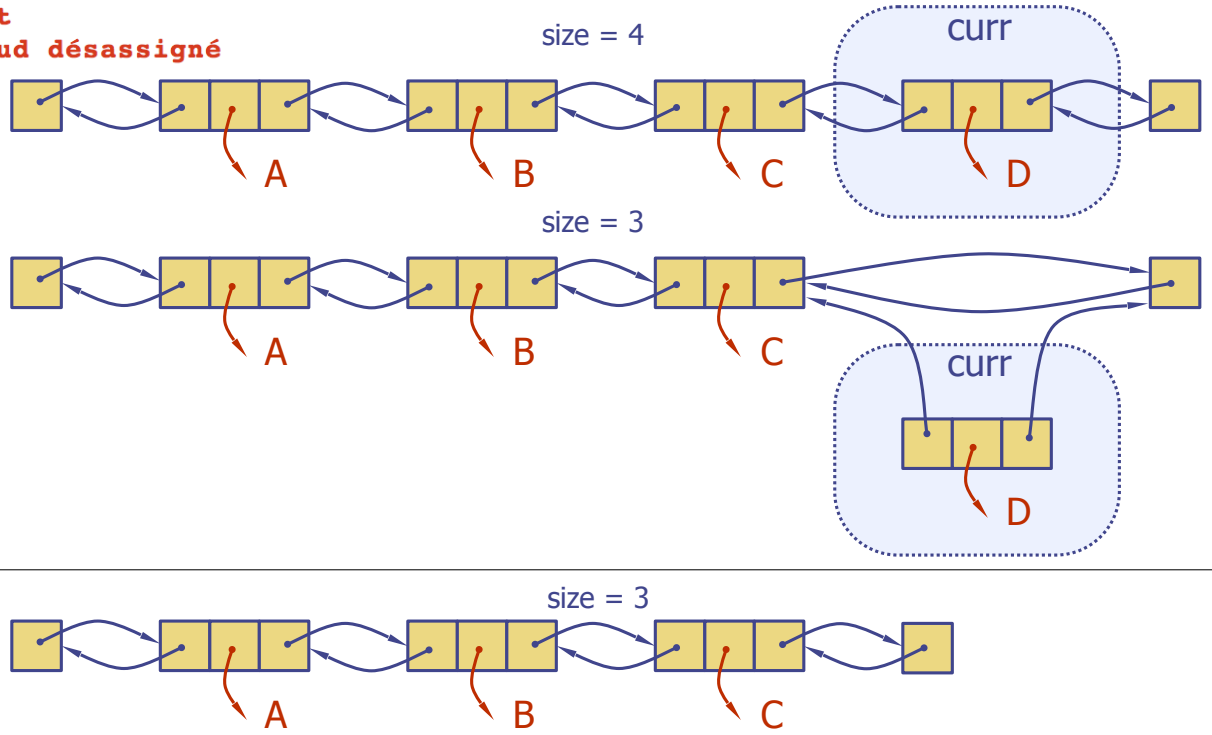
Insertion d'un noeud entre une position *p* et ses successeurs

(pas requise par l'interface *List.py*)

Se fait en O(n), car on doit trouver l'élément en position p



```
# retire l'élément à l'index k
def remove( self, k ):
    # index de liste commence à 0
    # on valide k
    if not 0 <= k < self._size:
        raise IndexError( 'DoublyLinkedList: index out of bounds' )
    else:
        # on se positionne sur le premier élément
        curr = self._head.next
        # on avance de k-1 noeuds
        for i in range( k ):
            curr = curr.next
        # curr pointe le noeud à retirer
        # on fait sauter les pointeurs :
        #   le suivant du précédent sur le suivant de curr
        #   le précédent du suivant de curr sur son précédent
        curr.prev.next = curr.next
        curr.next.prev = curr.prev
        # on libère l'espace occupé par l'élément
        curr.next = None #convention pour un noeud désassigné
        # on décrémente la taille
        self._size -= 1
        # on retourne l'élément
        return curr.element
```



```
# retourne l'index d'un élément en O(n)
# ou None s'il n'est pas trouvé
def find( self, element ):
    # si la liste est vide l'élément n'y est pas.
    if self.is_empty():
        return None
    # sinon, on commence par la tête de liste
    # et on parcourt les éléments un à un
    # en utilisant les références au noeuds suivants
    else:
        curr = self._head.next
        for i in range( self._size ):
            if curr.element == element:
                return i
            else:
                curr = curr.next
    # si aucun index n'a été retourné
    # c'est que l'élément n'a pas été trouvé
    return None

# retourne le dernier élément si la liste n'est pas vide
def last( self ):
    if self.is_empty():
        return None
    else:
        return self._tail.prev.element

# retourne le premier élément si la liste n'est pas vide
def first( self ):
    if self.is_empty():
        return None
    else:
        return self._head.next.element
```

Performances des opérations d'une liste doublement chaînée

- L'espace utilisé par une liste de n éléments est dans O(n)
- L'espace utilisé par chaque position de la liste est dans O(1)
- Insérer au début et à la fin se fait en temps dans O(1)
- Accéder au premier et dernier éléments se fait en temps dans O(1)
- Chercher et supprimer un noeud quelconque se fait en temps dans O(n)

File à deux bouts dans Python
(Deque)

Un *deque* est une structure de données semblable à une file d'attente qui prend en charge l'insertion et la suppression à la fois à l'avant et à l'arrière de la file d'attente.

L'ADT *Deque* est donc plus général que ceux de la pile et de la file d'attente

La généralisation peut être utile dans certaines applications. Par exemple, un restaurant utilisant une file d'attente. Occasionnellement, la première personne dans la file pourrait être retirée, puis on pourrait constater qu'aucune table n'est disponible. Dans ce cas, généralement, on réinsère la personne au premier rang dans la file d'attente.

Il peut également arriver qu'un client à la fin de la file d'attente peut s'impatienter et partir.

(Nous aurons besoin d'une structure de données encore plus générale si nous voulons que des clients à d'autres positions quittent la file d'attente.)

Série d'opérations sur une *Deque* initialement vide

Operation	Return Value	Deque
D.add_last(5)	–	[5]
D.add_first(3)	–	[3, 5]
D.add_first(7)	–	[7, 3, 5]
D.first( )	7	[7, 3, 5]
D.delete_last( )	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last( )	3	[7]
D.delete_last( )	7	[ ]
D.add_first(6)	–	[6]
D.last( )	6	[6]
D.add_first(8)	–	[8, 6]
D.is_empty( )	False	[8, 6]
D.last( )	6	[8, 6]

```
class Deque:

    def __init__( self ):
        pass

    def __len__( self ):
        pass

    def __str__( self ):
        pass

    def is_empty( self ):
        pass

    # ajoute un élément au début du Deque
    def add_first( self, element ):
        pass

    # ajoute un élément à la fin du Deque
    def add_last( self, element ):
        pass

    # retire le premier élément du Deque
    def delete_first( self ):
        pass

    # retire le dernier élément du Deque
    def delete_last( self ):
        pass

    # retourne le premier élément du Deque
    # sans le retirer
    def first( self ):
        pass

    # retourne le dernier élément du Deque
    # sans le retirer
    def last( self ):
        pass
```

Deque dans les collections Python

ADT Deque	collections.deque	Description
len(D)	len(D)	number of elements
D.add_first( )	D.appendleft( )	add to beginning
D.add_last( )	D.append( )	add to end
D.delete_first( )	D.popleft( )	remove from beginning
D.delete_last( )	D.pop( )	remove from end
D.first( )	D[0]	access first element
D.last( )	D[−1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear( )	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

Comparaison de notre ADT Deque et de la classe collections.deque

collections.deque

L'interface *collections.deque* a été choisie pour être cohérente avec les conventions de nommage de la classe *list* de Python, pour laquelle *append* et *pop* sont présumés agir à la fin de la liste. Par conséquent, *appendleft* et *popleft* désignent des opérations au début de la liste.

La bibliothèque *deque* imite également une liste en ce qu'elle est une séquence indexée, permettant un accès ou une modification arbitraire en utilisant la syntaxe D[j].

Le constructeur *deque* de la bibliothèque prend également en charge un paramètre optionnel *maxlen* pour forcer une *deque* de longueur fixe. Cependant, si un ajout est fait à chaque extrémité quand la *deque* est pleine, elle ne signale pas d'erreur. Au lieu de cela, l'ajout provoque l'abandon d'un élément du côté opposé. Autrement dit, un appel à *appendleft* lorsque le *deque* est pleine provoque un *pop* implicite du côté droit pour faire de la place pour le nouvel élément.

La distribution Python actuelle implémente *collections.deque* avec une approche hybride qui utilise des aspects de tableaux circulaires, mais organisés en blocs qui sont eux-mêmes organisés en une liste doublement chaînée (une structure de données que nous introduirons dans un prochain module).

La classe *deque* est formellement documentée pour garantir des opérations en temps dans O(1) à l'une ou l'autre extrémité, mais dans O(n) dans le pire cas lors de l'utilisation d'un index près du milieu de la *deque*.

Liste positionnelle

- Pour fournir une abstraction générale d'une séquence d'éléments avec la possibilité d'identifier l'emplacement d'un élément donné, nous définissons un ADT pour une liste positionnelle.
- Une position agit comme un marqueur ou un jeton dans la liste positionnelle.
- Une position *p* n'est pas affectée par des changements ailleurs dans la liste ; une position devient invalide si elle est supprimée.
- Une instance de position est un objet simple, ne supportant que la méthode suivante:

  p.element( ) : qui retourne l'élément stocké à la position *p*.

```
from DoublyLinkedList import DoublyLinkedList

class PositionalList( DoublyLinkedList ):

    class Position:
        # Une abstaction de la position d'un élément

        def __init__( self, container, node ):
            # constructeur
            # on garde une référence sur l'instance du container
            # et sur le noeud de la position
            self._container = container
            self._node = node

        def element( self ):
            # retourne l'élément stocké à cette position
            return self._node.element

        def __eq__( self, other ):
            # retourne True si other est du même type et réfère au même noeud
            return type( other ) is type( self ) and other._node is self._node

        def __ne__( self, other ):
            # retourne l'inverse de __eq__
            return not( self == other )
```

```
def _validate( self, p ):
    # retourne le noeud de la position, ou lance une exception si invalide
    # càd n'est pas une instance de Position, n'est pas une Position
    #     dans ce container, ou a été désassigné
    if not isinstance( p, self.Position ):
        raise TypeError( "p must be proper Position type" )
    if p._container is not self:
        raise ValueError( "p does not belong to this container" )
    if p._node.next is None: #convention pour noeud désassigné
        raise ValueError( "p is no longer valid" )
    return p._node

#Utilitaires
def _make_position( self, node ):
    # retourne une instance de Position pour un noeud donné (ou None si sentinelle)
    if node is self._head or node is self._tail:
        return None
    else:
        return self.Position( self, node )
```

```
#Méthodes d'accès
def first( self ):
    return self._make_position( self._head.next )

def last( self ):
    return self._make_position( self._tail.prev )

def before( self, p ):
    node = self._validate( p )
    return self._make_position( node.prev )

def after( self, p ):
    node = self._validate( p )
    return self._make_position( node.next )

def __iter__( self ):
    #itérateur des éléments de la liste
    cursor = self.first()
    while cursor is not None:
        yield cursor.element()
        cursor = self.after( cursor )
```

```
#Méthodes de mutations
#override les méthodes héritées pour retourner des Position plutôt que des noeuds.
def insert( self, e ):
    node = super().insert( e )
    return self._make_position( node )

def append( self, e ):
    node = super().append( e )
    return self._make_position( node )

def replace( self, p, e ):
    #remplace l'élément p par e
    #retourne l'élément qui était à la position p
    original = self._validate( p )
    old_value = original.element
    original.element = e
    return old_value
```

Avantages des séquences basées sur des tableaux

- Les tableaux fournissent un accès aux éléments en temps dans O(1), basé sur un index d'entier.
- La possibilité d'accéder au kième élément pour tout k en temps dans O(1) est une caractéristique à l'avantage des tableaux. En revanche, localiser le kième élément dans une liste chaînée nécessite un temps dans O(k) pour parcourir la liste depuis le début, ou éventuellement dans O(n-k) en traversant vers l'arrière à partir de la fin d'une liste doublement chaînée.
- Les opérations avec des limites asymptotiques équivalentes sont généralement, à une constante près, plus efficaces avec un tableau. Considérez l'opération de mise en file d'attente typique pour une file. Ignorez le problème du redimensionnement du tableau, cette opération pour la classe *ListQueue* implique de calculer le nouvel indice, incrémenter un entier et stocker la référence du nouvel élément dans le tableau. En revanche, le processus pour *SinglyLinkedQueue* nécessite d'instancier un nœud, ajuster les liaisons des nœuds, et incrémenter un entier. Alors que cette opération en temps est dans O(1) dans les 2 modèles, le nombre réel d'opérations est plus grande dans la version chaînée, en particulier compte tenu de l'instanciation d'un nouveau noeud.
- Les représentations basées sur des tableaux utilisent généralement moins de mémoire que les structures chaînées. Cet avantage peut sembler contre-intuitif, surtout vu que la longueur d'un tableau dynamique peut être plus longue que le nombre d'éléments qu'il contient. Les listes basées sur des tableaux et les listes chaînées sont des structures référentielles, donc la mémoire primaire pour stocker leurs éléments est le même dans chaque modèle. Ce qui diffère, ce sont les quantités auxiliaires de mémoire utilisées. Pour un conteneur basé sur un tableau de n éléments, le pire cas est 2n, celui d'un tableau récemment redimensionné. Avec les listes chaînées, la mémoire doit être consacrée non seulement à stocker une référence à chacun de ses objets, mais aussi les références explicites qui relient les nœuds. Donc, une liste simplement chaînée nécessite un minimum de 2n références, un de plus pour une référence au nœud suivant pour chaque nœud. Avec une liste doublement chaînée, on a 3n références.

Avantages des séquences basées sur des chaînages

- Les structures à base de chaînes réalisent des temps d'opérations bornés en pire cas. Ceci se distingue d'avec les temps d'opérations amortis associées à l'extension ou à la contraction d'un tableau dynamique. Sur de nombreuses opérations individuelles d'un long calcul, les 2 reviennent au même. Cependant, si les opérations sont utilisées dans un système en temps réel conçu pour fournir des réponses immédiates (e.g. un système d'exploitation, un serveur Web, un système de contrôle de circulation aérienne), un long délai causé par une seule opération (en temps amorti) peut avoir un effet négatif.
- Les structures chaînées prennent en charge les insertions et les suppressions en temps dans O(1) peu importe la position. Pensez à cette possibilité d'effectuer ces opérations avec la classe PositionalList. Ceci est probablement l'avantage le plus important d'une liste chaînée. Avec un tableau et en ignorant la question de le redimensionner, insérer ou supprimer un élément quelconque se fait en temps dans O(n) en moyenne en raison de la boucle de déplacement de tous les éléments suivants. Considérez l'exemple d'un éditeur de texte qui gère un document comme une séquence de caractères. Bien que les utilisateurs ajoutent souvent des caractères à la fin du document, iels peuvent utiliser le curseur pour insérer ou supprimer un ou plusieurs caractères à des positions arbitraires. Imaginez les conséquences d'implémenter un tel document texte avec une séquence basée sur un tableau comme la *list* Python ! Avec une liste chaînée, on a qu'à se définir une position pour l'emplacement du curseur et les opérations peuvent se faire en temps dans O(1).

Conclusions du module

- Nous nous sommes familiariser avec les Types Abstraits de Données (ADT) et nous avons vu le concept en implémentant les types de données de séquences les plus fréquemment utilisés tels que la liste, la pile, la file et le deque.
- Nous avons mis en pratique le concept d'ADT en regardant d'abord l'implantation du type list de Python basé sur le tableau. Nous avons étudié la question d'allonger et de racourcir le tableau en fonction des besoins et réalisé qu'il était possible d'implémenter cette propriété en temps amorti dans O(1).
- Nous avons introduit les types de données pile, file et deque, et nous les avons implémentés.
- Nous avons introduit les listes simplement et doublement chaînées et la possibilité d'inserer au début de la liste en temps d'exécution dans O(1).
- Finalement, nous avons introduit la liste positionnelle pour manipuler de manière abstraite la position d'un élément et permettre d'implémenter les opérations delete, remplace, ajoute_avant, et ajoute_après un élément en position arbitraire en temps d'exécution dans O(1).