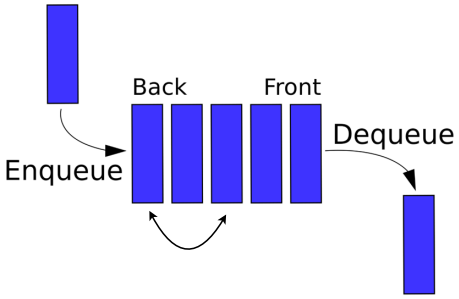
File d'attente prioritaire
Ordre total
ADT *PriorityQueue*
Implémentations avec *list* triée et non triée
Tri avec une file d'attente prioritaire
Tri par sélection
Tri par insertion
Tri "en-place"

File d'attente prioritaire



Il y des applications dans lesquelles une structure de file d'attente est utilisée mais dont la gestion des objets à traiter ne fonctionne pas avec la stratégie du premier entré premier sorti (soit la File conventionnelle vue précédemment).

Considérez un centre de contrôle du trafic aérien qui doit choisir quels vols doivent atterrir et dans quel ordre parmi tous ceux qui approchent l'aéroport. Ce choix peut être influencé par des facteurs comme leur distance de la piste, le temps passé dans un circuit d'attente, la quantité restante de carburant, etc. On voit ici qu'il est peu probable que les décisions d'atterrissage soient basées uniquement sur une politique FIFO !

Il existe d'autres situations dans lesquelles une politique de FIFO pourrait être raisonnable, mais pour laquelle d'autres priorités peuvent entrer en jeu. Imaginez la gestion de patients dans une salle d'urgence d'un hôpital. Au fur et à mesure que la salle se remplie, l'hôpital maintient une file d'attente des patients qui espèrent voir un médecin. Bien que la priorité d'un patient en attente soit influencée par son temps d'arrivée, d'autres considérations peuvent influencer l'ordre des patients, comme par exemple la gravité des symptômes, l'arrivée par ambulance, etc. On pourrait donc donner accès à un médecin à un patient qui est arrivé plus tard qu'un autre !

File d'attente prioritaire

Dans ce chapitre, nous introduisons un nouveau type de données connu sous le nom de **file d'attente prioritaire**.

Il s'agit d'une collection d'éléments prioritaires qui permet l'insertion arbitraire d'éléments et permet la suppression de l'élément qui a la plus grande priorité. Quand un élément est ajouté à une file d'attente prioritaire, l'utilisateur désigne sa priorité en fournissant une clé. L'élément avec la clé minimale sera le prochain à être retiré de la file. Un élément avec la clé x aura priorité sur un élément avec la clé y, si $x < y$.

Bien qu'il soit courant que les priorités soient exprimées numériquement, tout objet Python peut être utilisé comme clé, à condition que le type d'objet puisse être comparé, a < b, pour toutes ses instances a et b et de manière à définir un **ordre total**.

Avec une telle généralité, les applications peuvent développer leur propre notion de priorité pour chaque élément.

Pour tout x, y *et* z, on a :

Antisymétrie :

$$x \leq y \text{ et } y \leq x \Rightarrow x = y$$

Transitivité :

$$x \leq y \text{ et } y \leq z \Rightarrow x \leq z$$

Totalité :

$$x \leq y \text{ ou } y \leq x$$

ADT *PriorityQueue*

- Une file d'attente prioritaire stocke une collection d'éléments
- Chaque élément est une paire (clé, valeur)
- Les principales méthodes de la file d'attente prioritaire sont :
  - *ajouter(k, x)*, qui insère un élément avec la clé *k* et la valeur *x*
  - retirer_min( ), qui supprime et retourne l'élément avec la plus petite clé
- Quelques méthodes supplémentaires :
  - min( ), qui retourne, mais ne supprime pas, l'élément avec la plus petite clé
  - len( ), est_vide( )
- Applications:
  - Voyageurs en attente d'obtenir une place
  - Patients de la salle d'attente d'un hôpital
  - Ventes aux enchères
  - Marché boursier
  - etc.

```
#ADT PriorityQueue (Classe de base)
class PriorityQueue:

    # classe imbriquée privée pour les items
    class _Item:
        # enregistrement efficace pour les items
        # définis par une clé et une valeur
        __slots__ = '_key', '_value'

        def __init__( self, k, v ):
            self._key = k
            self._value = v

        def __lt__( self, other ):
            return self._key < other._key

        def __gt__( self, other ):
            return self._key > other._key

        def __str__( self ):
            return "(" + str( self._key ) + "," + str( self._value ) + ")"
```

```
# constructeur
def __init__( self ):
    pass

# taille de la file
def __len__( self ):
    pass

# pretty print
def __str__( self ):
    if self.is_empty():
        return "[]"
    pp = "["
    for item in self:
        pp += str( item )
    pp += "]"
    return pp

# test si vide
def is_empty( self ):
    return len( self ) == 0

# élément minimum (de plus grande priorité)
def min( self ):
    pass

# adjouter un élément x avec clé k
def add( self, k, x ):
    pass

# retirer l'élément de plus grande priorité
def remove_min( self ):
    pass
```

Exemple

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min( )	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min( )	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min( )	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min( )	(7,D)	{(9,C)}
P.remove_min( )	(9,C)	{ }
P.is_empty( )	True	{ }
P.remove_min( )	"error"	{ }

File d'attente prioritaire avec une séquence

Implémentation avec une **liste non triée**

Performance :

ajouter prend un temps dans O(1) puisque nous pouvons insérer un élément au début ou à la fin de la séquence en O(1) (sauf pour une insertion au début pour une liste Python)

retirer_min et min prennent un temps dans O(n) puisque nous devons parcourir la séquence pour trouver la plus petite clé

Implémentation avec une **liste triée**

Performance :

ajouter prend un temps dans O(n) puisque nous devons trouver où insérer l'élément en fonction de sa clé (chercher se fait en O(log n) avec la recherche dichotomique mais l'insertion est dans O(n))

retirer_min et min prennent un temps dans O(1) puisque la plus petite clé est au début de la séquence (si on utilise une liste Python, on doit inverser l'ordre pour permettre une suppression à la fin en O(1))

```python
# utilise la classe de base PriorityQueue
from PriorityQueue import PriorityQueue

# ADT UnsortedListPriorityQueue
class UnsortedListPriorityQueue( PriorityQueue ):

    # méthodes de base

    # constructeur, on utilise une liste Python
    def __init__( self ):
        self._Q = []

    # taille de la file avec len de la liste Python
    def __len__( self ):
        return len( self._Q )

    # accès direct à un élément de la liste
    def __getitem__( self, i ):
        return self._Q[i]

    # vide si len == 0
    def is_empty( self ):
        return len( self ) == 0

    # cherche et retourne l'élément de plus grande priorité
    def min( self ):
        if self.is_empty():
            return None

        # cherche le min en O(n)
        the_min = self._Q[0]
        for item in self:
            if item < the_min:
                the_min = item

        #retourne le min
        return the_min
```

```
# ajoute un élément x de clé k à la fin de la liste, O(1)
def add( self, k, x ):
    #in O(1)
    self._Q.append( self._Item( k, x ) )

# recherche, supprime et retourne l'élément de plus grande priorité
def remove_min( self ):
    if self.is_empty():
        return None

    # cherche l'index du min en O(n)
    index_min = 0
    for i in range( 1, len( self ) ):
        if self._Q[i] < self._Q[index_min]:
            index_min = i

    the_min = self._Q[index_min]
    # supprime le min en O(n)
    del self._Q[index_min]
    # retourne l'élément supprimé
    return the_min
```

```python
# utilise la classe de base PriorityQueue
from PriorityQueue import PriorityQueue

# ADT SortedListPriorityQueue
class SortedListPriorityQueue( PriorityQueue ):

    # méthodes de base

    # constructeur, on utilise une liste Python
    def __init__( self ):
        self._Q = []

    # taille de la file avec len de la liste Python
    def __len__( self ):
        return len( self._Q )

    # accès direct à un élément de la liste
    def __getitem__( self, i ):
        return self._Q[i]

    # vide si len == 0
    def is_empty( self ):
        return len( self ) == 0

    # retourne l'élément de plus grande priorité en O(1)
    def min( self ):
        if self.is_empty():
            return None

        # le min est à Q[0]
        return self._Q[0]
```

```
# ajoute l'élément x de clé k
def add( self, k, x ):
    item = self._Item( k, x )
    if self.is_empty():
        self._Q.append( item )
    else:
        # créer l'espace supplémentaire dans Q, O(1)
        self._Q.append( item )

        # cherche l'index d'insertion en O(n)
        i = 0
        while item > self._Q[i]:
            i += 1

        # décale les éléments en O(n)
        for j in range( len( self ) - 1, i, -1 ):
            self._Q[j] = self._Q[j-1]

        # insère le nouvel élément à l'index i
        self._Q[i] = item
    return item

# retourne et supprime l'élément de plus grande priorité, O(n)
def remove_min( self ):
    if self.is_empty():
        return None

    # accède min en O(1); le min est à Q[0]
    the_min = self._Q[0]

    # del doit décaler les éléments, O(n)
    del self._Q[0]
    return the_min
```

Trier avec une file d'attente prioritaire

Nous pouvons utiliser une file d'attente prioritaire pour **trier** un ensemble d'éléments comparables

On a qu'à insérer les éléments un par un avec une série d'opérations ajouter

Ensuite, on supprime les éléments dans l'ordre trié avec une série d'opérations retirer_min

Le temps d'exécution de cette méthode de tri dépend de l'implémentation de la file d'attente prioritaire

```
Algorithme Trier-FileDAttentePrioritaire( S, C )
    Input séquence S, comparateur C
    Output séquence S triée en ordre croissante selon C
    P = File d'attente prioritaire avec comparateur C
    Tantque non S.est_vide()
        e = S.retirer_suivant()
        P.ajouter( e, e )
    Tantque non P.est_vide()
        e = P.retirer_min().key()
        S.append( e )
```

Tri par sélection

Le **tri par sélection** est une variation du tri par file d'attente prioritaire, où la file d'attente prioritaire est implémentée avec une **séquence non triée**

Le temps de fonctionnement du tri par sélection :

1. L'insertion des éléments dans la file d'attente prioritaire avec *n* opérations insérer prend un temps dans O(*n*)
2. La suppression des éléments dans l'ordre trié de la file d'attente prioritaire se fait avec *n* opérations retire_min prend un temps

proportionnel à $1 + 2 + \ldots + n = \sum_{i=1}^{n} i = n(n+1)/2$

Donc, le tri par sélection s'exécute en temps dans O(n^2)

Exemple de tri par sélection

		Séquence S	File d'attente prioritaire P (non triée)
	Entrée:	(7,4,8,2,5,3,9)	()
	Phase 1		
O(n)	(a)	(4,8,2,5,3,9)	(7)
	(b)	(8,2,5,3,9)	(7,4)
	..	.. ..	
	(g)	()	(7,4,8,2,5,3,9)
	Phase 2		
$O(n^2)$	(a)	(2)	(7,4,8,5,3,9)
	(b)	(2,3)	(7,4,8,5,9)
	(c)	(2,3,4)	(7,8,5,9)
	(d)	(2,3,4,5)	(7,8,9)
	(e)	(2,3,4,5,7)	(8,9)
	(f)	(2,3,4,5,7,8)	(9)
	(g)	(2,3,4,5,7,8,9)	()

Tri par insertion

Le **tri par insertion** est une variation du tri par file d'attente prioritaire, où la file d'attente prioritaire est implémentée avec une **séquence triée**

Le temps de fonctionnement du tri par insertion :

1. L'insertion des éléments dans la file d'attente prioritaire avec n opérations insérer prend un temps proportionnel à $1 + 2 + ... + n$, donc $O(n^2)$
2. La suppression des éléments dans l'ordre trié de la file d'attente prioritaire se fait avec n opérations retire_min prend un temps dans $O(n)$

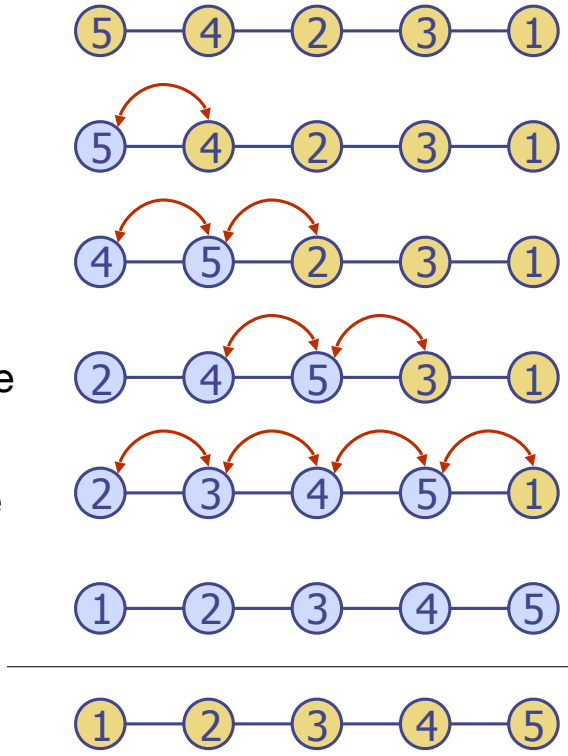
Donc, le tri par insertion s'exécute en temps dans $O(n^2)$

Exemple de tri par insertion

		Séquence S	File d'attente prioritaire P (triée)
	Input:	(7,4,8,2,5,3,9)	()
	Phase 1		
$O(n^2)$	(a)	(4,8,2,5,3,9)	(7)
	(b)	(8,2,5,3,9)	(4,7)
	(c)	(2,5,3,9)	(4,7,8)
	(d)	(5,3,9)	(2,4,7,8)
	(e)	(3,9)	(2,4,5,7,8)
	(f)	(9)	(2,3,4,5,7,8)
	(g)	()	(2,3,4,5,7,8,9)
	Phase 2		
	(a)	(2)	(3,4,5,7,8,9)
O(n)	(b)	(2,3)	(4,5,7,8,9)
	..	..	..
	(g)	(2,3,4,5,7,8,9)	()

Tri par sélection et par insertion "en-place"

- Au lieu d'utiliser une structure de données externe, nous pouvons implémenter le tri par sélection et le tri par insertion **en-place**
- On se sert d'une partie de la séquence d'entrée comme file d'attente prioritaire
- Pour le tri par insertion en-place
  - Nous gardons triés la partie gauche de la séquence
  - Nous pouvons faire des échanges au lieu de modifier la séquence



Séquence S	P
(5,4,2,3,1)	()
(4,2,3,1)	(5)
(2,3,1)	(4,5)
(3,1)	(2,4,5)
(1)	(2,3,4,5)
()	(1,2,3,4,5)
(1)	(2,3,4,5)
(1,2)	(3,4,5)
..	..
(1,2,3,4,5)	()