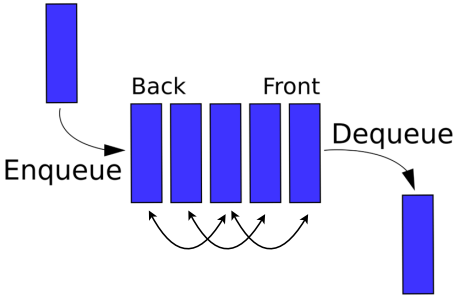
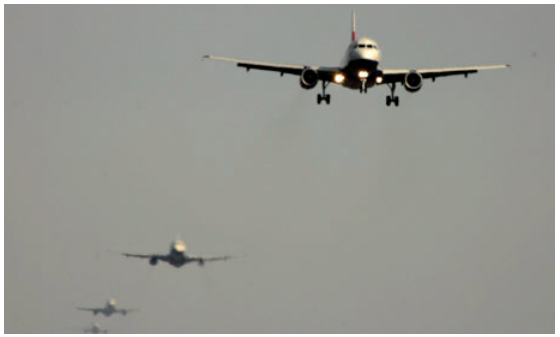
1. File d'attente avec priorités
2. Ordre naturel et entrées (clé, valeur)
3. ADT **`PriorityQueue`**
4. Ordre total et **`interface Comparable`**

File d'attente avec priorités



Pour plusieurs applications, une structure de file d'attente ("First In First Out") ne fonctionne pas.

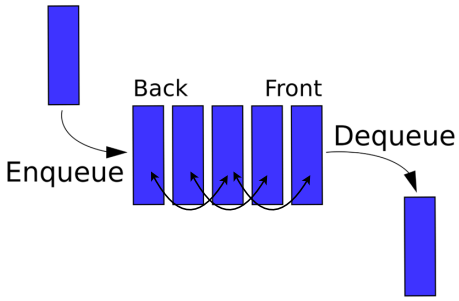
Par exemple, un centre de contrôle du trafic aérien qui doit choisir quels vols doivent atterrir et dans quel ordre. Les décisions doivent être prises en fonction de la distance de la piste, temps passé dans un circuit d'attente, quantité de carburant, etc.



On voit ici qu'il est peu probable que les décisions d'atterrissage soient basées uniquement sur une politique FIFO !

Un autre exemple est la gestion de patients dans une salle d'urgence. Au fur et à mesure que la salle se remplie, le service d'urgence maintient une file d'attente des patients qui espèrent voir un médecin. Bien que la priorité d'un patient en attente est influencée par son temps d'arrivée, d'autres considérations peuvent influencer l'ordre de traitement, comme la gravité des symptômes, la disponibilité d'un spécialiste du cas, l'arrivée par ambulance, etc. Il arrive de donner un accès plus rapide à un patient qui est arrivé plus tard qu'un autre !

File d'attente avec priorités



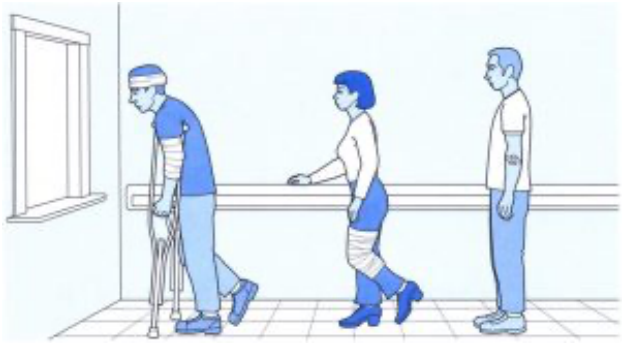
Dans ce section, nous introduisons l'ADT d'une **file d'attente avec priorités** (**`PriorityQueue`**)

Il s'agit d'une collection d'éléments qui permet l'**insertion** avec des priorités et l'**accès** au prochain élément "le plus prioritaire".

Lorsqu'un élément est ajouté, sa priorité est ajoutée, créant des insertions de paires `(priorité, élément)`.

L'élément avec la priorité minimale sera le prochain à être accédé. Ainsi, un élément avec la priorité, disons 1, sera prioritaire sur un élément de priorité 2. N'importe quel objet Java peut être utilisé comme priorité, tant qu'il existe un moyen de le comparer avec un autre et de manière à définir un ordre naturel des priorités. Avec cette généralité, les applications peuvent développer leur propre notion de priorité pour chaque élément.

Plus formellement, un élément avec la priorité x sera traité avant un élément avec la priorité y, si $x < y$.



ADT `PriorityQueue`

- Nous modélisons un item à gérer avec un tuple ou une paire (**priorité, élément**), qu'on appelle entrée (entry en anglais). Nous définirons bientôt une interface pour **`Entry`**.
- L'ADT **PriorityQueue** prend en charge les méthodes suivantes :
  - `insert`(k, v), qui crée et insère dans la file avec priorité un item avec la priorité k et l'élément v; et retourne l'entrée créée. Pour simplifer, on parle d'une entrée (**Entry**) composée d'une clé k et d'une valeur v.
  - `min`( ), qui retourne, mais qui ne supprime pas, l'entrée (k, v) qui possède la plus petite clé; retourne `null` si la file est vide.
  - `removeMin`( ), qui supprime et retourne l'entrée (k, v) qui possède la plus petite clé; retourne null si la file est vide.
  - `size`( ), qui retourne le nombre d'entrées dans la file.
  - `isEmpty`( ), qui retourne un booléen indiquant si la file est vide (`true`) ou non (`false`).

- Applications:
  - Voyageurs en attente d'obtenir une place
  - Patients d'une salle d'attente
  - File d'atterrissage
  - Ventes aux enchères
  - Marché boursier

- Une file d'attente avec priorités peut avoir plusieurs entrées avec des clés équivalentes. Dans ce cas, les méthodes `min` et `removeMin` peuvent s'appliquer à une entrée arbitraire parmi elles.
- Les valeurs peuvent être de n'importe quel type d'objet (on l'a déjà dit!)
- Dans ce modèle de base de file d'attente avec priorités, nous supposons que la clé d'un élément reste la même après qu'elle a été ajoutée dans une file. Dans le dernier module de cette section, nous verrons une extension qui permet à un utilisateur de mettre à jour la clé d'une entrée.

Exemple pour tester le fonctionnement d'une file avec priorités

Méthode	Valeur retournée	Contenu de la file prioritaire
insert(5,A)	(5,A)	{ (5,A) }
insert(9,C)	(9,C)	{ (5,A), (9,C) }
insert(3,B)	(3,B)	{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)	(7,D)	{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

- La mise en œuvre d'une file d'attente avec priorités doit garder les entrées, clé-valeur, même lorsqu'elles sont déplacées dans la structure de données.
- Cela s'apparente à comment maintenir une séquence d'éléments avec leurs fréquences d'accès. On introduit une composition ; rappelez-vous la classe `Item` (dans `FavoritesList`) qui associait à chaque élément sa fréquence d'accès (`count`).
- Pour les files d'attente avec priorités, on utilise la composition pour associer une clé k et sa valeur v dans un seul objet : **`Entry`**.

Allons voir le code de
`Entry.java`
`PriorityQueue.java`

Comparaison des clés

- En définissant l'ADT, nous autorisons n'importe quel type d'objet à servir de clé ; nous devons comparer les clés entre elles de manière **cohérente**, c'est-à-dire à ce que les résultats des comparaisons ne soient contradictoires.
- Pour qu'une règle de comparaison soit cohérente, elle doit définir une relation d'ordre total, c'est-à-dire qu'elle doit satisfaire les propriétés d'**antisymétrie**, de **transitivié** et de **totalité**.

- Pour toutes les clés x, y et z :

  Antisymétrie :

  $x \leq y$ et $y \leq x \Rightarrow x = y$

  Transitivité :

  $x \leq y$ et $y \leq z \Rightarrow x \leq z$

  Totalité :

  $x \leq y$ ou $y \leq x$

- La propriété de **totalité** indique que la règle de comparaison est définie pour chaque paire de clés. Cette propriété implique la **réflexivité** : $k \leq k$
- Une règle de comparaison qui définit une relation d'ordre total ne conduira jamais à une contradiction. Une telle règle définit un ordre linéaire parmi un ensemble de clés ; par conséquent, si un ensemble (fini) d'entrées possède un ordre total défini pour lui-même, alors la notion de clé minimale, $k_{\min}$, est bien définie :

  $k_{\min} \leq k$ pour toute autre clé k de cet ensemble.

Interface **`Comparable`**

- Java fournit deux moyens de définir des comparaisons entre les types d'objets. Le premier d'entre eux consiste à définir l'ordre naturel de ses instances en implémentant formellement l'interface **`java.lang.Comparable`**, qui inclut une seule méthode, **`compareTo`**. La syntaxe **`a.compareTo( b )`** doit retourner un entier i avec la signification suivante :

  $i < 0$ désigne que $a < b$

  $i = 0$ désigne que $a = b$

  $i > 0$ désigne que $a > b$

- Par exemple, la méthode `compareTo` de la classe `String` définit l'ordre des chaînes de caractères pour être lexicographique, soit une extension de l'ordre alphabétique de l'Unicode.

Interface **`Comparator`**

- Dans certaines applications, nous voudrions comparer des objets selon une notion autre que leur ordre naturel. Par exemple, nous pourrions vouloir choisir la plus courte de deux chaînes, ou définir nos propres règles complexes pour juger laquelle des deux actions est la plus prometteuse.
- Pour prendre en charge cette généralité, Java définit l'interface **`java.util.Comparator`**. Un comparateur est un objet "**extérieur**" à la classe des éléments qu'il compare (comparativement à **`Comparable`** que rend des éléments comparables à l'intérieur de la classe). Il fournit une méthode avec la signature **`compare( a, b )`** qui renvoie un entier avec une signification pareil à la méthode `compareTo`.

Allons voir le code de
`StringLengthComparator.java`

`Comparator` et file avec priorités

- Pour offrir une forme générale et réutilisable de file d'attente avec priorités, nous permettons à un utilisateur de choisir le type des clés et de spécifier l'instance de comparateur désirée. Nous offrirons une version du constructeur permettant de fournir ce "comparateur". Une instance de file avec priorités utilisera le comparateur passé en argument au constructeur, ou un comparateur par défaut (pour ne pas **embêter** l'utilisateur non plus), chaque fois qu'elle aura besoin de comparer deux clés.
- Le comparateur par défaut utilisera l'ordre naturel des clés données (en supposant qu'elles sont **`comparable`**).

Allons voir le code de
`DefaultComparator.java`

Allons voir le code de

`AbstractPriorityQueue.java`