1. ADT file avec priorités adaptable
2. Entrées sensibles à l'emplacement
3. Analyse des temps d'exécution
4. Implémentation
5. Conclusions du module

# File avec priorités adaptable

Les méthodes de l'ADT file d'attente avec priorités sont suffisantes pour plusieurs applications. Cependant, il existe des situations dans lesquelles des méthodes supplémentaires seraient utiles, comme le montrent les scénarios suivants, impliquant des passagers aériens en attente :

- Un passager en attente avec une attitude pessimiste peut se lasser et décider de partir avant l'heure d'embarquement, en demandant à être retiré de la file d'attente. Ainsi, nous souhaitons retirer de la file d'attente avec priorités l'entrée associée à ce passager. L'opération **`removeMin`** ne suffit pas car le passager qui part n'a pas forcément la priorité. Nous voulons une nouvelle opération, **`remove`**, qui supprime une entrée arbitraire
- Un autre passager en attente trouve sa carte de voyageur fréquent (“Gold”) et la montre à l'agent. Ainsi, sa priorité est modifiée en conséquence. Pour réaliser ce changement de priorité, nous souhaiterions disposer d'une nouvelle opération **`replaceKey`** nous permettant de remplacer la clé d'une entrée existante par une nouvelle clé
- Enfin, une troisième passagère en attente constate que son nom est mal orthographié sur le billet et demande qu'il soit corrigé. Pour effectuer ce changement, nous devons mettre à jour le dossier du passager. Par conséquent, nous aimerions avoir une nouvelle opération **`replaceValue`**, nous permettant de remplacer la valeur d'une entrée existante par une nouvelle valeur.

# ADT file avec priorités adaptable

Afin d'implémenter efficacement les méthodes **`remove`**, **`replaceKey`** et **`replaceValue`**, nous avons besoin d'un mécanisme pour trouver l'entrée d'un utilisateur dans une file d'attente, idéalement d'une manière à éviter une recherche linéaire dans toute la collection.

Dans la définition originale de l'ADT file d'attente avec priorités, un appel à **`insert`**( $k$, $v$ ) renvoie formellement une instance de type **`Entry`** à l'utilisateur. Afin de pouvoir mettre à jour ou supprimer une entrée dans notre nouvelle file d'attente avec priorités adaptable, l'utilisateur doit conserver cet objet **`Entry`** en tant que **jeton** qui peut être renvoyé comme argument pour identifier l'entrée pertinente.

- **`remove`**( $e$ ), supprime l'entrée $e$ de la file
- **`replaceKey`**( $e$, $k$ ), remplace la clé de l'entrée $e$ par $k$
- **`replaceValue`**( $e$, $v$ ), remplace la valeur de l'entrée $e$ par $v$

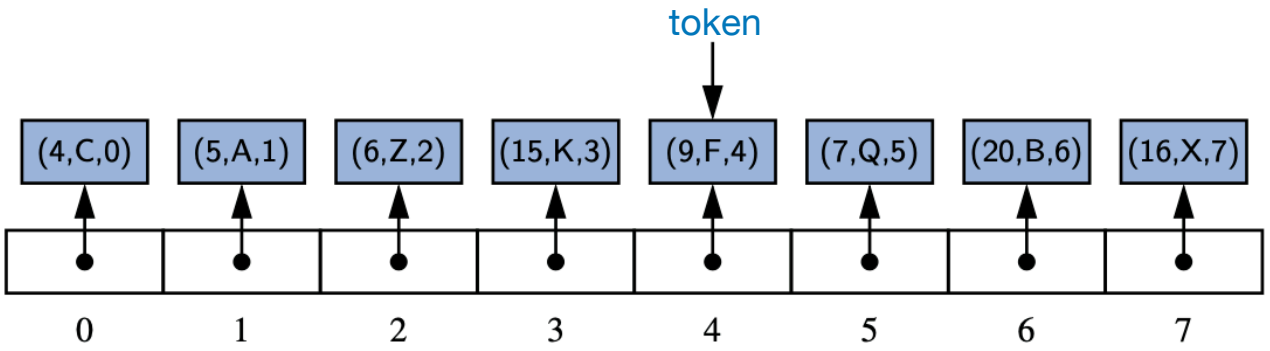
👉 Une erreur se produit avec chacune de ces méthodes si le paramètre $e$ est invalide (par exemple, parce qu'il a précédemment été supprimé de la file)

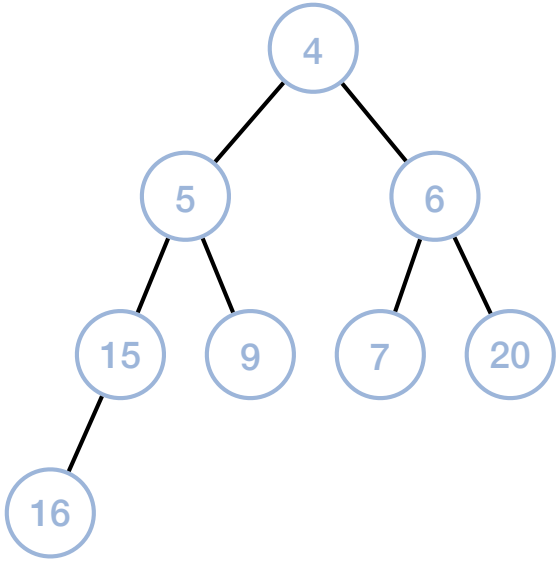
Allons voir le code de

**`AdaptablePriorityQueue.java`**

# Entrées sensibles à l'emplacement

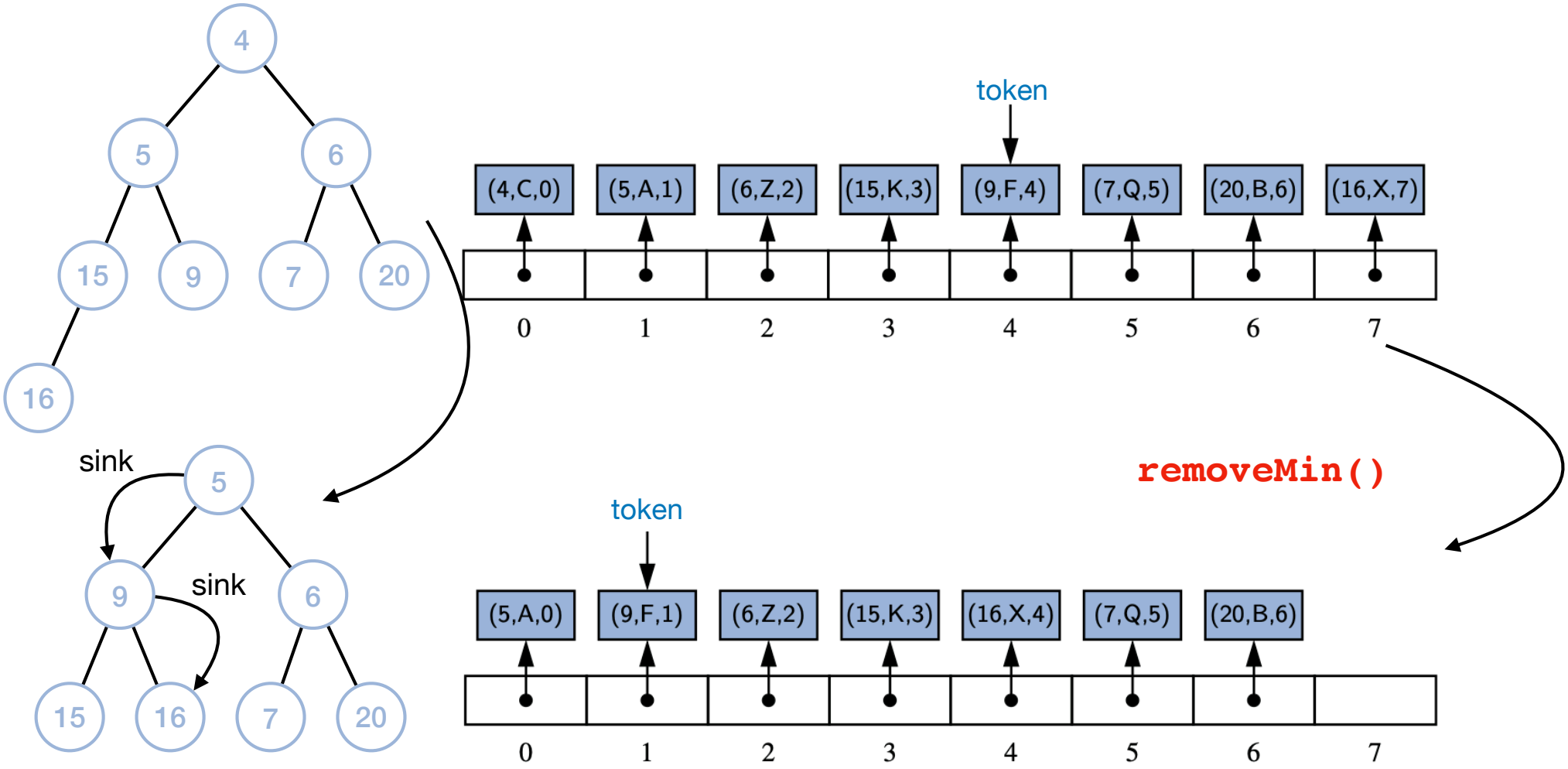
Pour permettre à une instance d'entrée de coder un emplacement dans une file d'attente avec priorités, nous étendons la classe `PQEntry` (définie à l'origine avec la classe de base `AbstractPriorityQueue`), en ajoutant un troisième champ qui désigne l'index actuel d'une entrée dans la représentation basée sur un tableau.



# Opérations sensibles à l'emplacement

Lorsque nous effectuons des opérations provoquant le déplacement des entrées, nous devons mettre à jour le troisième champ de chaque entrée affectée pour refléter son nouvel index dans le tableau.



# Implémentation de la file avec priorités adaptable

On implémente une file d'attente avec priorités adaptable en tant que sous-classe de la classe `HeapPriorityQueue`

1. Nous commençons par définir une classe **`AdaptablePQEntry`** imbriquée qui étend la classe `PQEntry` héritée, en l'augmentant d'un champ supplémentaire: **index**
2. La méthode **`insert`** héritée est surchargée, de sorte que nous créons et initialisons une instance de la classe `AdaptablePQEntry` (et non pas de la classe `PQEntry` d'origine)

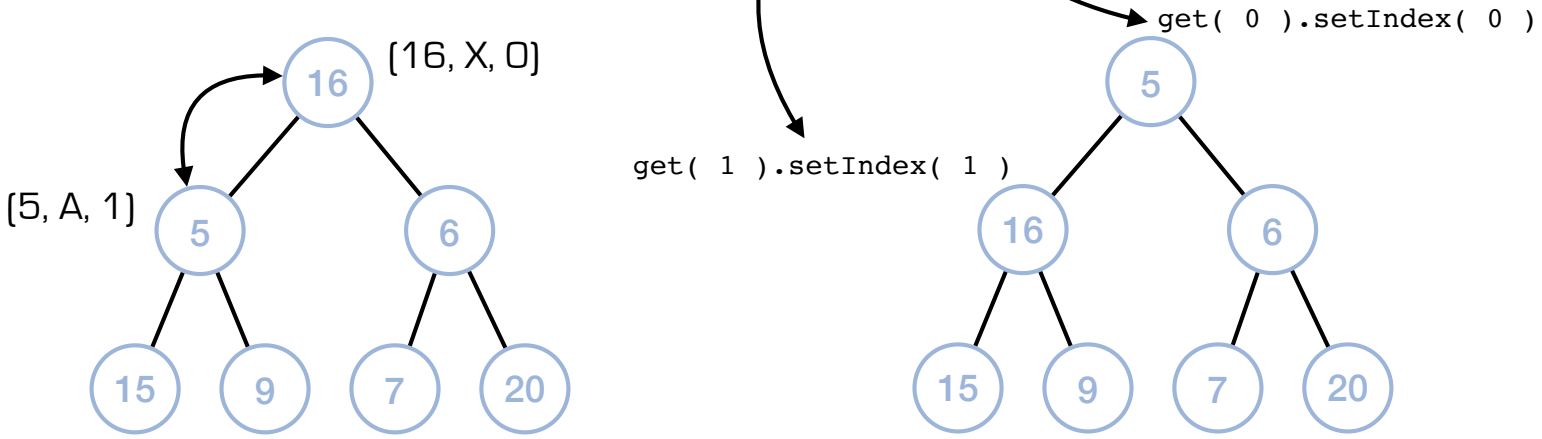
   Un aspect important de cette conception est que la classe `HeapPriorityQueue` d'origine repose exclusivement sur une méthode d'échange, **`swap`**, pour tous les mouvements de données pendant les opérations **`swim`** et **`sink`**. La classe `AdaptablePriorityQueue` remplace cet utilitaire afin de mettre à jour les index stockés de nos entrées sensibles à l'emplacement lorsqu'elles sont déplacées
3. Lorsqu'une entrée est envoyée en paramètre à **`remove`**, **`replaceKey`** ou **`replaceValue`**, nous nous appuyons sur le nouveau champ `index` de cette entrée pour désigner l'emplacement de l'élément dans le tas
4. Lorsqu'une clé existante est remplacée, cette nouvelle clé peut violer la propriété d'ordre en étant soit trop grande, soit trop petite. Nous proposons un nouvel utilitaire, **`bubble`**, qui détermine si on doit la déplacer vers le haut ou vers le bas
5. Lors de la **suppression** d'une entrée arbitraire, nous la remplaçons par la dernière entrée du tas (pour conserver la propriété de l'arbre binaire complet) et exécutons **`bubble`** car l'élément déplacé peut avoir une clé trop grande ou trop petite pour son nouvel emplacement

Allons voir le code de

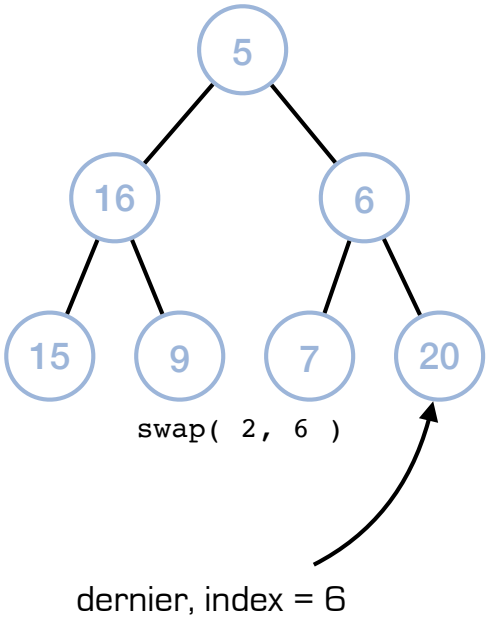
**`HeapAdaptablePriorityQueue.java`**

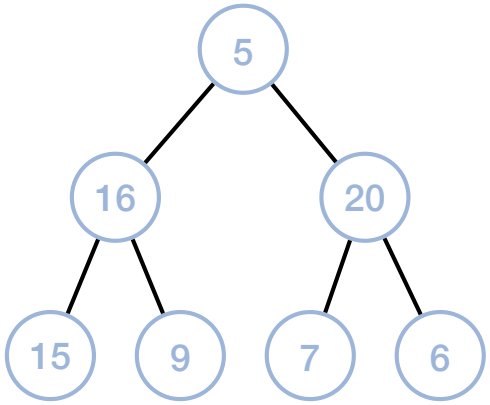
# `swap( 0, 1 )`

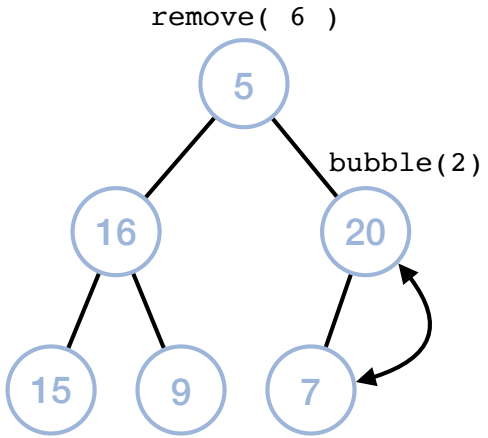
ajuste les indices des entrées après `super.swap( i, j )`
L'index de l'entrée déplacée à i est assignée i
L'index de l'entrée dép;acée à j est assignée j



# remove( (6, Z, 2) )


5
16
6
15
9
7
20
swap( 2, 6 )
dernier, index = 6


5
16
20
15
9
7
6


remove( 6 )
5
bubble(2)
16
20
15
9
7

# Analyse de complexité de la file avec priorités adaptable

La nouvelle classe fournit la même efficacité asymptotique et la même utilisation de l'espace mémoire que la version non adaptative, et fournit des performances logarithmiques pour les nouvelles méthodes **`remove`** et **`replaceKey`** basées sur un localisateur, et des performances en temps constant pour la nouvelle méthode **`replaceValue`**. Demandez-vous pourquoi replaceValue est dans $O(1)$ ?

| Méthode | Temps d'exécution |
|---|---|
| size, isEmpty, min | $O(1)$ |
| insert | $O(\log n)$ |
| remove | $O(\log n)$ |
| removeMin | $O(\log n)$ |
| replaceKey | $O(\log n)$ |
| replaceValue | $O(1)$ |

# Conclusions du module

- On a vu l'ADT pour des **files d'attente avec priorités**
- On a vu les implémentations qui utilisent des listes positionnelles (non triée et triée) et que leurs complexités associées en dépendent. Dans le cas de la **liste non triée**, l'insertion est efficace mais **`min`** ne l'est pas, alors qu'avec la **liste triée** c'est l'inverse.
- On a vu qu'on peut utiliser une file d'attente avec priorités pour trier un ensemble d'éléments. Avec une liste non triée, on a un **tri par sélection** et avec une liste triée on a un **tri par insertion**.
- On a vu qu'on pouvait utiliser un seul tableau pour **trier en-place**.
- On a vu que pour obtenir un temps raisonnable (dans $O(\log n)$) pour les opérations **`insert`** et **`removeMin`**, le **tas** est une solution.
- On a vu que les opérations déterminantes de mise à jour d'un tas, **`swim`** et **`sink`**, prennent des temps dans $O(\log n)$, correspondant à la hauteur maximale d'un tas contenant $n$ clés.
- On a vu que trier avec une file d'attente avec priorités implémentée avec un tas prend un temps dans $O(n \log n)$, un gain considérable sur les tris par sélection et insertion, et ce **simplement en changeant la structure de données**.
- On a vu comment implémenter la structure en tas dans un tableau.
- On a vu qu'on pouvait construire un tas en temps dans $O(n)$.
- On a vue une extension de l'ADT pour des files d'attente avec priorités adpatable, où on ajoute les opérations **`remove`**, **`replaceKey`** et **`replaceValue`**
- On a vu que pour que ces opérations soient efficaces, on doit rajouter aux entrées un index pour les rendre sensibles à leur emplacement.
- L'implémentation du nouvel ADT est réalisée en : 1) créant la classe `AdaptablePQEntry`, une sous-classe de `PQEntry`, dans laquelle on ajoute l'**index** de l'entrée; 2) surchargeant la méthode **`insert`** pour utiliser `AdatpablePQEntry`; 3) surchargeant la méthode **`swap`** pour mettre à jour les index des entrées déplacées; 4) s'assurant de rétablir la propriété du tas après avoir changé la clé d'une entrée ou après la suppression d'une entrée avec une nouvelle méthode **`bubble`** qui voit si l'entrée doit être déplacée vers le haut ou vers le bas.
- La version adaptable possède la même efficacité asymptotique et la même utilisation de l'espace mémoire que la version non adaptative et fournit des performances logarithmiques pour les nouvelles méthodes **`remove`** et **`replaceKey`**.