Files de priorité (Priority Queue)

TAD Files de priorité

Implémentation avec des séquences

Exemples d'application: Tris

Files de priorité

- Une collection (ensemble) d'éléments dotés d'une priorité selon des critères prédéfinis en avance
- On peut insérer les éléments dans n'importe quel ordre
- L'extraction des éléments se fait suivant l'ordre de priorité (l'élément ayant la priorité la plus grande est extrait en premier)
- Les éléments sont stockés suivant leur priorité et non plus suivant leur position (comme dans le cas de piles, files, séquences etc.)

Files de priorité

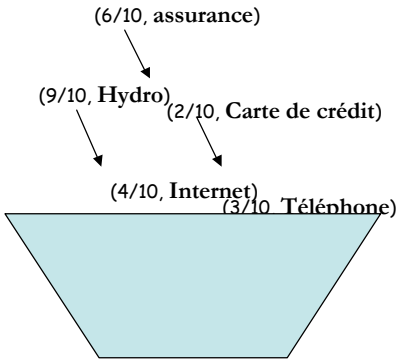
Exemple de priorité :

La date limite pour payer une facture

La date limite pour transmettre vos devoirs

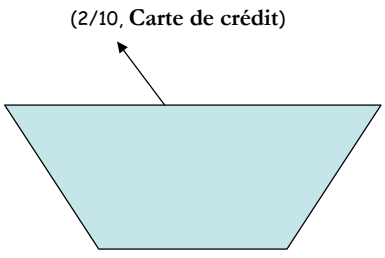
Les passagers en attente (le statut de voyageur fréquent,

le tarif à payer et le temps l'enregistrement, etc)

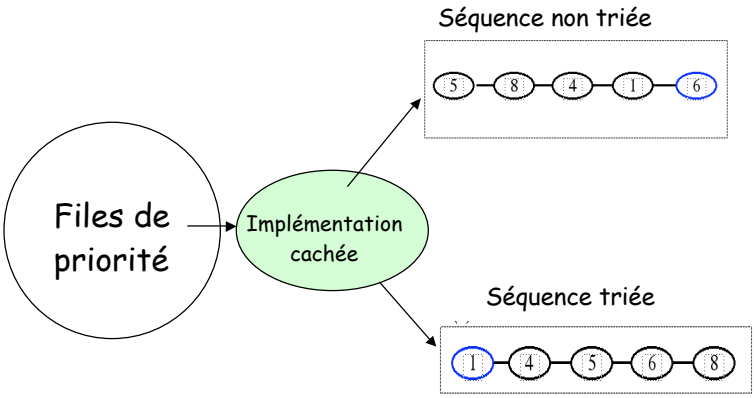


Factures avec dates limites:
Insérer(clé,élément)

remove



Files de priorité – Implémentation



Le TAD File de priorité

- Une file de priorité (Priority Queue) est une collection d'entrées (entries)
- Chaque entrée est une paire:(clé, valeur) ou (clé, élément)
- Chaque élément a sa propre clé
- La clé sert à identifier l'élément ou à décrire sa priorité
- Des entrées diverses peuvent avoir la même clé
- Il y a une relation d'ordre total sur les clés

Clés et relations d'ordre total

- Dans une file de priorité les éléments sont classés suivant leur clé soumis à une relation d'ordre total
- Une relation binaire ♣ est dite d'ordre total sur un ensemble donné E ssi elle est:

Réflexive: k ♣ k
Antisymétrique: si k1 ♣ k2 et k2 ♣ k1, alors k1 = k2
Transitive: si k1 ♣ k2 et k2 ♣ k3, alors k1 ♣ k3

Exemple d'ordre total

≤ est une relation d'ordre total sur l'ensemble des entiers

Réflexive: k ≤ k
Antisymétrique: si k1 ≤ k2 et k2 ≤ k1, alors k1 = k2
Transitive: si k1 ≤ k2 et k2 ≤ k3, alors k1 ≤ k3

Exemples d'ordre total

- $\geq$ est une relation d'ordre total sur l'ensemble des entiers

L'ordre alphabétique et l'ordre alphabétique inverse sont d'ordre total

Mais…

- $<$, $>$ sont pas d'ordres total puisque ils ne sont pas réflexive

Réflexive: $k < k$

Antisymétrique: si $k1 < k2$ et $k2 < k1$, alors $k1 = k2$

Transitive: si $k1 < k2$ et $k2 < k3$, alors $k1 < k3$

TAD Entry

- Entry (entrée) est une paire (clé, valeur)
- Une file de priorité stocke les entrées pour simplifier les opérations de suppression/insertion basées sur les clés
- Méthodes:
  - ✓Key(): Retourne la clé associée à l'entrée courante
  - ✓Value: Retourne la valeur associée à l'entrée courante

Comparateurs

- La forme la plus générale et la plus réutilisable de file de priorité utilise des objets appelés **comparateurs**
- Les comparateurs sont externes aux clés à comparer et permettent de comparer deux objets
- Une file de priorité **P** a un comparateur quand **P** est construite
- Quand **P** a besoin de comparer deux clés, elle utilise le comparateur qui lui est associé
- Ainsi, une file à priorité peut être suffisamment générique pour contenir n'importe quel type objet

Comparateurs

- Méthodes:
  - -isLessThan(a, b)
  - -isLessThanOrEqualTo(a,b)
  - -isEqualTo(a, b)
  - -isGreaterThan(a,b)
  - -isGreaterThanOrEqualTo(a,b)
  - -isComparable(a)

Le TAD File de priorité

- Une file de priorité *P* supporte les méthodes suivantes:

— size(): Retourne le nombre d'éléments dans P

— isEmpty(): Vérifie si P est vide

— insert(k,e): Insère un nouvel élément e avec sa clé k dans P

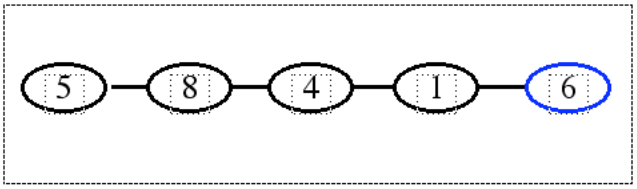
— min(): Retourne (mais ne retire pas) un élément de P à la plus petite clé; une erreur survient si P est vide

— minKey(): Retourne la plus petite clé de P; une erreur survient si P est vide

— removeMin(): Retire et retourne un élément de P à la plus petite clé; une erreur survient si P est vide

Réalisation avec séquence non-triée

- Les éléments de S sont composés des paires (clé, valeur) (k,e)
- Nous pouvons réaliser insert() en utilisant la méthode insertLast() sur les séquences. Le temps d'exécution sera alors **O(1)**
- Cependant, comme nous insérons toujours à la fin, sans tenir compte de la valeur de la clé, notre séquence n'est pas ordonnée

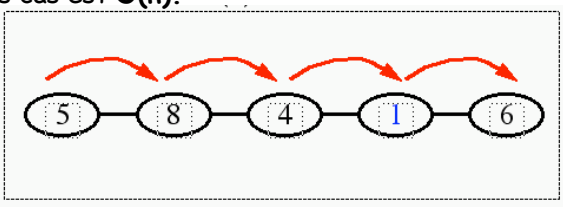


7

Réalisation avec séquence non-triée

- La séquence n'est pas ordonnée..

⇨ Pour min(), minKey(), et removeMin(), nous devons regarder tous les éléments de S. La complexité du pire des cas est **O(n)**.



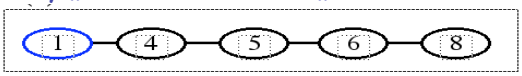
- Sommaire des performances

insertItem	$O(1)$
minKey, minElement	$O(n)$
removeMin	$O(n)$

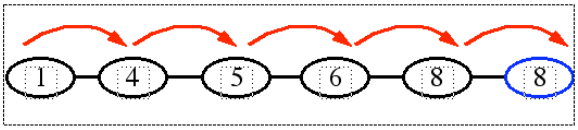
18

Réalisation avec séquence triée

- Une autre réalisation possible utilise une séquence S, triée par ordre croissant de clés.
- min(), minKey(), et removeMin() deviennent alors **O(1)**



- Cependant, pour réaliser insert(), nous devons maintenant parcourir la séquence entière dans le pire des cas. Ainsi, insert() s'exécute en temps **O(n)**



insertItem	*O(n)*
minKey, minElement	*O(1)*
removeMin	*O(1)*

Une observation…

Avec séquence non-triée

removeMin() prends toujours O(n), même dans le meilleur cas.

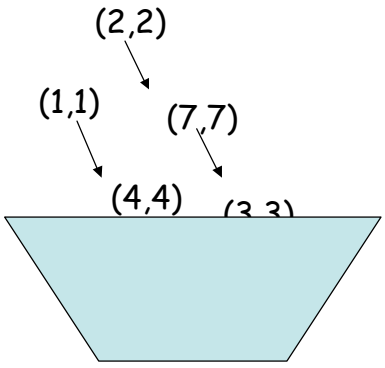
Mais avec séquence triée:

insertItem() prends au plus O(n)

Une Application: Tri

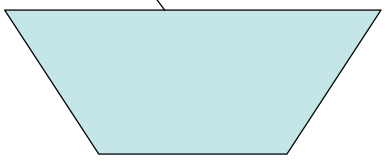
- Une file de priorité *P* peut être utilisée pour trier une séquence *S* :
  - ***en insérant*** les éléments de S dans P avec une suite d'opérations insert(e, e) — **La clé est l'élément lui même**
  - ***en retirant*** les éléments de P en ordre croissant et en les remettant dans S avec une suite d'opérations removeMin()

insert .....



remove

(1,1)



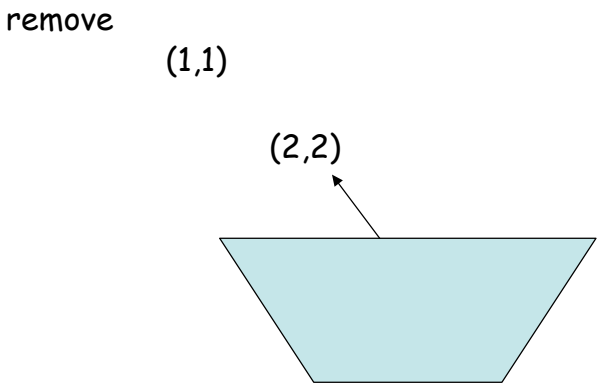


Schéma général d'un tri avec une file de priorité

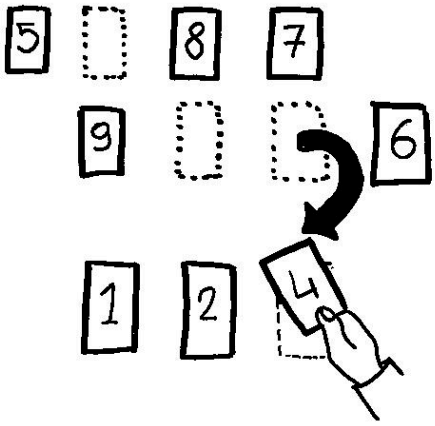
```
Algorithm PriorityQueueSort(S, P):
Entrées: Une séquence S contenant n éléments, avec une relation
d'ordre total, et une file de priorité P qui compare les clés avec
cette relation
Sortie: Une séquence S triée à l'aide de la relation d'ordre total

        while !S.isEmpty() do
                e ← S.removeFirst()
                P.insert(e, e)
        while P is not empty do
                e ← P.removeMin()
                S.insertLast(e)
```

Selection Sort
(tri par sélection)

- Le tri par sélection est basé sur le schéma du tri par file de priorité (PriorityQueueSort); Il utilise une **séquence non-triée** pour implémenter la file de priorité P :
  - ✓ Phase 1: l'insertion d'un élément de S dans P (insert == insertLast) est exécutée en un temps O(1)
  - ✓ Phase 2: le retrait d'un élément de P (removeMin()) prend un temps proportionnel au nombre d'éléments présents dans P

Selection Sort

Insertion dans une ordre non spécifié



Sélection en ordre

Selection Sort Exemple

		Séquence S		File de priorité P
Entrée:		(7,4,8,2,5,3,9)		()
Phase 1				
	(a)	(4,8,2,5,3,9)	insert() == insertLast()	(7)
	(b)	(8,2,5,3,9)		(7,4)
	..	.. ..		
	.	. .		
	(g)	()		(7,4,8,2,5,3,9)
Phase 2				
	(a)	(2)		(7,4,8,5,3,9)
	(b)	(2,3)		(7,4,8,5,9)
	(c)	(2,3,4)	removeMin()	(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)		()

Sequence non triee

Selection Sort (cont.)

- Le temps d'exécution de tri par sélection:

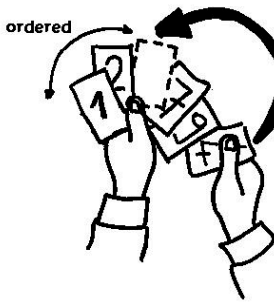
  Insérer les n éléments dans la file de priorité avec n opérations de insert() est $O(n)$

  Enlever les éléments dans l'ordre trié de la file de priorité avec n opérations de removeMin() est proportionnel à

  $$n + (n - 1) + \ldots + 2 + 1$$

- Le tri par sélection est exécuté en temps $O(n^2)$

Selection Sort In Place

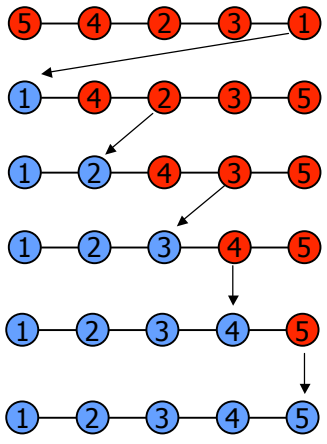


- Au lieu d'utiliser de l'espace additionnel, on peut exécuter Select Sort "en place" en divisant S en deux parties complémentaires: une triée et une non triée
- Pour select-sort en place
  - A chaque étape on cherche le plus petit élément de la partie non triée de S, on le retire de cette partie et on l'ajoute à la fin de la partie triée (vide à l'étape 0) et ainsi de suite jusqu'à trier toute la séquence S

1

Selection Sort In Place

Partie triée
Partie non triée



CSI2510 32

Insert-Sort
(Tri par insertion)

Le tri par insertion est basé sur le schéma du tri par file de priorité (PriorityQueueSort); Il utilise une **séquence triée** pour implémenter la file de priorité P :

- ✓ Phase 1: l'insertion d'un élément de S dans P (insert) est exécutée en un temps O(n) au pire des cas
- ✓ Phase 2: le retrait d'un élément de P (removeMin()) est exécutée en un temps O(1)

Insertion-Sort Exemple

	Séquence S	File de priorité P
Entrée:	(7,4,8,2,5,3,9)	()
Phase 1		
(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)
(b)	(2,3)	(4,5,7,8,9)
(c)	(2,3,4)	(5,7,8,9)
(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)	(	

Tri par insertion (suite)

Séquence triée

- Le temps d'exécution d'un tri par sélection:
  - ✓ L'insertion des n éléments dans la file de priorité est faite avec n opérations insert() en un temps total qui est proportionnel au pire des cas à:

    $n + (n - 1) + \ldots + 2 + 1$
  - ✓ Le retrait des n éléments, dans l'ordre trié, de la file de priorité est fait avec n opérations removeMin() en un temps total $O(1)$

Le tri par sélection est exécuté en un temps $O(n^2)$

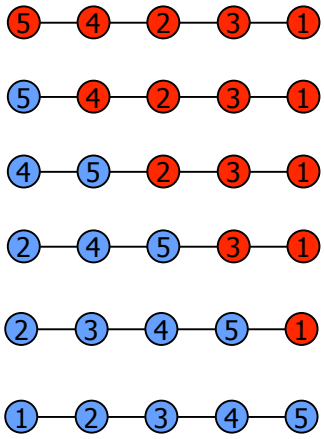
In-place Insertion-sort

- Au lieu d'utiliser une deuxième structure de données P (espace occupé en plus) pour trier une séquence S, on peut exécuter le tri par insertion « sur place » (In-place Insertion sort) en divisant S en deux parties complémentaires: une triée et une non triée

- A chaque étape on retire le premier élément de la partie non triée de S et on l'insère dans la partie triée (vide à l'étape 0) de façon à garder cette partie triée et ainsi de suite jusqu'à insérer toute la séquence S

In-place Insertion-sort

Partie triée
Partie non triée



Comparaison

- Le tri par sélection va toujours exécuter un nombre d'opérations $O(n^2)$ peu importe la séquence d'entrée
  - removeMin() est toujours exécuté en un temps $O(n)$
- Le temps d'exécution du tri par insertion varie selon la séquence d'entrée
  - insertItem() est exécuté au pire des cas en un temps $O(n)$