

# Files avec priorités (§9)

## Exemples:

- Files = futurs passagers d'un avion
  - **Priorités:**
    - 1) Classe affaires
    - 2) Personnes voyageant avec de jeunes enfants et/ou personnes à mobilité réduite
    - 3) Les autres passagers
- Files = avions près d'un aéroport voulant atterrir
  - **Priorités:**
    - 1) Urgences
    - 2) Niveau d'essence
    - 3) Distance de l'aéroport

- Les files avec priorités nous permettent de donner priorité à certains éléments de la file, selon certains critères prédéfinis au départ
- Les files avec priorités gardent en mémoire les éléments selon leur priorité et non leur position

## TAD: Files avec priorité

- Une file avec priorités garde en mémoire une collection d'entrées (items)
- Chaque **item** est une paire (clé, élément)
- Opérations principales:
  - **insérer(k,x)**: insère une entrée ayant la clé k et l'élément x
  - **enleverMin()**: enlève et retourne une entrée de clé minimale
- Opérations additionnels:
  - **minElement()**: retourne, sans enlever, un élément de clé minimale
  - **minCle()**: retourne la clé minimale
  - **taille()**, **estVide()**

## Relations d'ordre total

- Les clés d'une file avec priorités peuvent être des objets arbitraires sur lesquels on définit une relation d'ordre
- Deux entrées distinctes d'une file avec priorités peuvent avoir la même clé
- On compare les clés selon un ordre total, i.e une relation  $\leq$  qui satisfait les propriétés suivantes, pour  $x, y$  et  $z$  des clés:
  - réflexive:  $x \leq x$
  - antisymétrique:  $x \leq y$  et  $y \leq x \implies x = y$
  - transitive:  $x \leq y$  et  $y \leq z \implies x \leq z$

## TAD: Item

- Une entrée d'une file avec priorités est tout simplement une paire d'objets (clé, élément). On crée un TAD Item correspondant à cette paire d'objets et permettant un accès facile à la clé ou la valeur
- Une file avec priorités doit permettre l'insertion et la suppression facile d'item selon leur clé
- Opérations:
  - **clé()**: Retourne la valeur de la clé d'une entrée
  - **élément()**: Retourne l'élément de l'entrée
  - **SETclé(k)**: La clé de l'entrée devient k
  - **SETvaleur(e)**: L'élément de l'entrée devient e

## TAD: Comparateur

- Un comparateur est un objet qui compare deux clés, étant donné une relation d'ordre total
- Quand une file avec priorités à besoin de comparer deux clés, elle utilise un comparateur
- Opérations:
  - **compare(a,b)**: Retourne un entier  $i$  tel que:

$i < 0$  si  $a < b$   
 $i = 0$  si  $a = b$   
 $i > 0$  si  $a > b$

Retourne une erreur si  $a$  et  $b$  ne peuvent être comparés

## Exemple de comparateur

```
Lexicographic comparison of 2-D
points:

/** Comparator for 2D points under the
    standard lexicographic order. */
public class Lexicographic implements
    Comparator {
    int xa, ya, xb, yb;
    public int compare(Object a, Object b)
        throws ClassCastException {
        xa = ((Point2D) a).getX();
        ya = ((Point2D) a).getY();
        xb = ((Point2D) b).getX();
        yb = ((Point2D) b).getY();
        if (xa != xb)
            return xa - xb ;
        else
            return ya - yb ;
    }
}
```

© 2004, Goodrich, Tamassia

Point objects:

```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D {
    protected int xc, yc; // coordinates
    public Point2D(int x, int y) {
        xc = x;
        yc = y;
    }
    public int getX() {
        return xc;
    }
    public int getY() {
        return yc;
    }
}
```

## Trier en utilisant une file avec priorités

- On peut utiliser une file avec priorités pour trier un ensemble d'éléments comparables
  - Insérer les éléments un par un dans la file avec une série d'opérations **insérer(e,e)**
  - Enlever les éléments dans l'ordre croissant avec une série d'opérations **enleverMin()**
- Le temps d'exécution dépend de l'implémentation de la file avec priorités

**Algorithme TriFP(S, C)**  
**Entrée** séquence  $S$ , comparateur  $C$  pour les éléments de  $S$   
**Sortie** séquence  $S$  triée en ordre croissant selon  $C$   
 $P \leftarrow$  file avec priorité utilisant le comparateur  $C$   
**while**  $\neg S.estVide()$   
      $e \leftarrow S.enlevePremier()$   
      $P.insérer(e, e)$   
**while**  $\neg P.estVide()$   
      $e \leftarrow P.enleverMin()$   
      $S.insérerFin(e)$

© adapté de 2004, Goodrich, Tamassia

## Implémentation utilisant une séquence

- séquence non ordonnée



- séquence ordonnée



- Performances:

- insérer(k,x)**: prend un temps  $O(1)$  étant donné qu'on peut insérer le nouvel élément au début ou à la fin de la séquence
- enleverMin()**: prend un temps  $O(n)$  étant donné qu'on doit traverser la séquence entière pour trouver la clé minimale

- Performances:

- insérer(k,x)**: prend un temps  $O(n)$  étant donné qu'on doit trouver la bon endroit où insérer le nouvel élément
- enleverMin()**: prend un temps  $O(1)$  étant donné que la clé minimale est au début de la séquence

## Tri-Sélection

- Le Tri-Sélection est une variation de TriFP, où la file avec priorités est implémentée avec une séquence non ordonnée

- Temps d'exécution de Tri-Sélection:

- Insérer les éléments dans la file avec priorités avec  $n$  opérations **insérer(k,e)** prend un temps  $O(n)$
- Enlever les éléments de la file avec priorités, dans ordre croissant, avec  $n$  opérations **enleverMin()** prend un temps proportionnel à

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

- La complexité en temps de Tri-Sélection est en  $O(n^2)$

## Tri-Insertion

- Le Tri-Insertion est une variation de TriFP, où la file avec priorités est implémentée avec une séquence ordonnée

- Temps d'exécution de Tri-Insertion:

- Insérer les éléments dans la file avec priorités avec  $n$  opérations **insérer(k,e)** prend un temps proportionnel à

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

- Enlever les éléments de la file avec priorités, dans ordre croissant, avec  $n$  opérations **enleverMin()** prend un temps  $O(n)$

- La complexité en temps de Tri-Insertion est en  $O(n^2)$