

UE ALGO5 — TD2 — Séance 4 : Files et files à priorité

Objectifs

À la fin de cette séance, vous devriez être capable de :

- manipuler et concevoir des files ou des FAP comme des structures abstraites ;
- réfléchir aux propriétés de files ou FAP ;
- proposer des implémentations de files et de FAP cohérentes avec les spécifications choisies.

Exercice 1.

On souhaite programmer une structure de données destinée à gérer des requêtes d'impression envoyées à une imprimante. Pour des raisons matérielles, on impose que cette structure de données soit implémentée sans faire appel à un mécanisme d'allocation dynamique de la mémoire, c'est-à-dire uniquement à l'aide d'un (ou plusieurs) tableau(x) déclaré(s) statiquement au début du programme.

Les requêtes d'impression sont *déposées* par les utilisateurs dans une structure de donnée de type *File*, dans laquelle l'imprimante viendra *extraire* une tâche à effectuer. On souhaite dans un premier temps que les requêtes soient traitées selon une politique « premier arrivé, premier servi ».

Les primitives nécessaires à la manipulation de valeurs du type *File* sont les suivantes :

<p>1 Requête : un type File : type { ... à compléter ... }</p> <p>3 Initialiser(F) 5 { Donnée-résultat : une File F Post-condition : F est initialisé avec un contenu vide 7 Effet de bord : F est modifiée }</p> <p>9 EstVide(F) 11 { Donnée : une File F Résultat : un booléen vrai ssi F est vide }</p> <p>13 Déposer(F,x) 15 { Donnée-résultat : une File F Donnée : une Requête x 17 Post-condition : x est ajoutée dans F Effets de bord : F est modifiée }</p>	<p>19 Extraire(F,x) 21 { Donnée-résultat : une File F Résultat : une Requête x 23 Pré-condition : F non vide Post-condition : supprime la requête la plus ancienne 25 de F et la mémorise dans x Effets de bord : F est modifiée }</p> <p>27 Consulter(F) 29 { Donnée-résultat : une File F Résultat : la Requête x la plus ancienne de F 31 Pré-condition : F non vide Effets de bord : aucun }</p> <p>33</p> <p>35</p>
--	---

Q1. Discutez des différentes implémentations possibles du type *File*. Dans chaque cas, donnez une estimation du coût en temps d'exécution pour chacune des fonctions.

Q2. Choisissez une des solutions possibles et implémentez chacune des fonctions.

Corrigé —

Le premier problème est de choisir une structure de données pour le type *File*...

L'idée la plus naturelle est de mémoriser toutes les requêtes dans un grand tableau T (indexé de 0 à

IND_MAX_FILE).

Remarque : le fait que le tableau soit borné va poser le pb du débordement, on pourra ajouter un code d'erreur pour la fonction *Déposer* lorsque le tableau est plein ...

Une fois ce choix fixé il faut décider comment on implémente Déposer et Extraire. Deux solutions *a priori* :

Solution 1 :

- quand on dépose, on ajoute en queue de tableau
 - le contenu courant de la file est alors mémorisé entre $T[0]$ et $T[ind_queue]$, les éléments entre $T[ind_queue+1]$ et $T[IND_MAX_FILE]$ sont «libres».
 - lorsque l'on extrait un élément (en tête), il faut «retasser» tous les éléments de la file vers la gauche (décalage).
- ⇒ la politique FIFO est respectée.

Dans ce cas, la valeur de `ind_queue` doit être mémorisée, le type `FileRequête` est un couple

(tableau `T`, entier `ind_queue`).

→ Représentation de la file vide ?

- soit on utilise un booléen (et le type `File` devient un triplet (tableau, entier, booléen))
- soit, plus cohérent, on utilise une valeur particulière de `ind_queue`, non comprise entre 0 et `IND_MAX_FILE` (par exemple -1).

On obtient donc :

File : un couple <tableau `T` sur $0..IND_MAX_FILE$ de Requête, entier `ind_queue`>

```
2
Initialiser ()
4     F.ind_queue ← -1    /* file vide */

6 EstVide : un booléen
    renvoyer (F.ind_queue = -1)

8
Déposer (x : donnée Élément)
10    F.ind_queue = F.ind_queue + 1
    F.T[F.ind_queue] = x

12
Extraire (x : résultat Élément)
14    x = F.T[0]
    pour i allant de 0 a F.ind_queue - 1
16        F.T[i] = F.T[i+1]
    F.ind_queue = F.ind_queue - 1
18    { prendra la valeur -1 si on extrait le dernier élément }

20 Consulter () : une requête
    renvoyer (F.T[0])
```

Problème de cette solution : l'insertion est en $O(1)$, l'extraction en $O(n)$ où n est la taille de la file.

Remarque : on a choisi implicitement d'avoir toujours 0 comme indice de tête.

Solution 2 : on peut faire mieux si on mémorise les indices de tête et de queue.

- Le contenu courant de la file est mémorisé entre $T[ind_tete]$ et $T[ind_queue]$, les éléments entre $T[ind_queue+1]$ et $T[IND_MAX_FILE]$ et $T[0]$ et $T[ind_tete-1]$ sont «libres».
- Lorsque l'on insère un élément, on incrémente `ind_queue` ;
- lorsque l'on extrait un élément, on incrémente `ind_tete`.

Pour utiliser toute la place disponible il faut gérer le tableau de manière circulaire, en incrémentant modulo IND_MAX_FILE+1 .

Problème : détecter quand la file est pleine, quand elle est vide, et distinguer les 2 cas ?

La file est pleine (ou vide !) quand les deux indices se sont «rattrapés» : $ind_queue+1 = ind_tete$.

On peut distinguer les deux cas en utilisant un booléen.

On obtient alors :

File : le **type** <tableau `T` sur $0..IND_MAX_FILE$ de Requête,

```

2         ind_tete : entier, ind_queue : entier, vide : booléen>

4 succ (i : donnée entier) : un entier
    renvoyer ((i+1) modulo ( IND_MAX_FILE +1))

6 Déposer (F,x)
8     F.ind_queue := succ(F.ind_queue)
    F.T[F.ind_queue] := x
10    F.vide := faux

12 Extraire (F,x)
    x := F.T[F.ind_tete]
14    F.ind_tete := succ(F.ind_tete)
    si ind_tete = ind_queue alors F.vide := vrai

16 Initialiser(F)
18    F.ind_queue := 0
    F.ind_tete := 0
20    F.vide := vrai

22 EstVide(F) : un booléen
    renvoyer F.vide

24 Consulter(F) : une requête
26    renvoyer(F.T[F.ind_tete])

```

On a bien ici toutes les opérations en $O(1)$.

Exercice 2.

On souhaite maintenant associer une *priorité* à chaque requête : lors de l'exécution de la primitive `Extraire` c'est une requête (quelconque) de priorité maximale (parmi les requêtes présentes) qui doit être extraite. Le nombre de priorité sera supposé fini.

Q 1. Modifiez la spécification des primitives, et reprendre les questions de l'exercice 1 dans ce nouveau contexte.

Corrigé —

Rappel de la définition :

—on dispose d'un ensemble ordonné *Priorité* ($<$ est une relation d'ordre total sur cet ensemble) ;

—on dispose d'un ensemble *Élément* (ici le type *Requêtes*).

Une file d'attente avec priorité (fap) est une file dont les primitives `Insérer` et `Extraire` deviennent :

```

Insérer (F : donnée—résultat File, x : donnée Élément, p : donnée Priorité)
2  {ajoute à la file F l'élément x de priorité p}

4 Extraire (F: donnée—résultat File, x: résultat Élément, p: résultat Priorité)
    {extrait de la file F un élément de priorité maximale (n'importe lequel !)}

```

On va s'intéresser à différentes implémentations de ce type abstrait.

Solutions contiguës

Différentes solutions sont envisageables, par exemple :

—on mémorise dans le tableau des couples $\langle \text{élément}, \text{priorité} \rangle$

—insertion : en queue, comme dans une file d'attente

- extraction : recherche d'un élément de priorité maximale (parcours séquentiel), et suppression (en remplaçant l'élément supprimé par le dernier)
- on mémorise dans le tableau des couples <élément,priorité>, ordonnés par priorité croissante, ainsi que l'indice du dernier élément (ind_queue)
- insertion : recherche de la position d'insertion, puis décalage
- extraction : on extrait l'élément de queue (nécessairement de priorité maximale), comme dans une file d'attente.

Cas particulier : nombre de priorités de taille fixe

L'inconvénient de ces deux solutions est que l'une des 2 opérations au moins nécessite un parcours complet de la fap. Le coût (en nombre de comparaisons) et donc dans le cas le plus défavorable en $O(n)$ où n est le nombre d'éléments dans la file à priorités.

On peut espérer améliorer les choses si l'ensemble de Priorité est connu a priori, de taille fixe. On peut en effet dans ce cas utiliser pour cela un tableau de file d'attente, ou, plus facile à gérer, un tableau de pile TabPile indicé par priorité (bien remarquer qu'aucun ordre n'est imposé pour l'extraction des éléments de même priorité... on a donc le choix de la structure).

La fap est alors un tableau, indicé par les priorités, et indiquant pour chaque priorité p l'indice ind_sommet associée à TabPile[p].

TabPile : le **type** tableau sur 1..MAX_PRIO de tableau sur 0..IND_MAX_PILE d'éléments

```

2
Fap : le type
4   <T : TabPile, Sommet : tableau sur 1..MAX_PRIO d'entiers>
   { F.Sommet[i] représente l'indice de sommet de pile de la pile F.TabPile[i] }
6
8   Insertion (F, x, p) :
   F.Sommet[p] := F.Sommet[p] + 1
10  F.T[F.Sommet[p]] := x
12  Extraction (F, x, p) :
   { on recherche (séquentiellement) le premier indice p tq Fap[p] <> -1 }
14  i := MAX_PRIO
   tant que i > 0 et puis F.Sommet[i] = -1
16  i := i - 1
   { on dépile dans x un élément de TabPile[p] }
18  si i = 0 alors Erreur(file à priorité vide)
   sinon x := F.T[F.Sommet[i]]
20  F.Sommet[i] := F.Sommet[i] - 1

```

Avec cette nouvelle structure le coût de l'insertion est constant, et le coût de l'extraction est en $O(|\text{Priorité}|)$. Elle est donc «rentable» (du point de vue du coût en temps!) si $|\text{Priorité}|$ est inférieur à la taille de la file à priorité dans le cas le plus défavorable.

Exemple concret : file d'attente avec qqs niveaux de priorité (ex : processus Unix, 20 niveaux de priorité).
