TD1.6 – Simulation mutuelle : file, pile, FAP

Objectifs

À la fin de cette séance, vous devriez être capable de :
— faire la différence entre une file et une FAP ;
— utiliser une structure de données pour en réaliser une autre.

Exercice 1 : Échauffement

Supposons que l'on dispose d'une pile, munie des opérations habituelles. Et supposons que pour la réalisation d'un TP, on ait besoin d'utiliser une file.

On se demande s'il serait possible d'utiliser la pile dont on dispose pour programmer la file dont on a besoin : c'est-à-dire de programmer les opérations *Enfiler* et *Défiler* de la file à l'aide des opérations de la pile. On n'attache pas d'importance dans un premier temps au coût de ces opérations (en temps d'exécution), mais on ne veut pas recopier ou dupliquer tous les éléments (sauf un seul d'entre eux à la fois pour pouvoir effectuer les transferts).

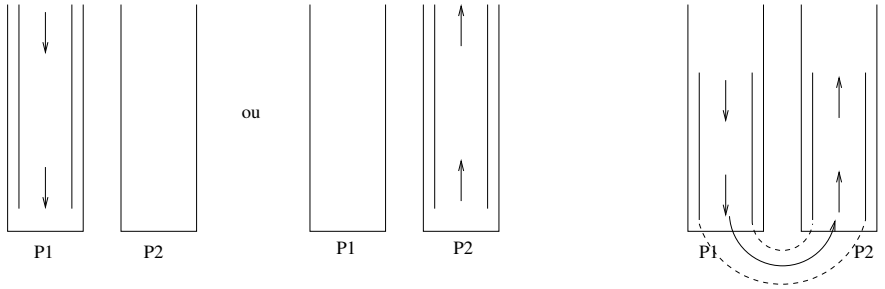
Il est facile de se convaincre que c'est impossible : pour mémoriser les éléments qui entrent dans la file, on ne peut que les empiler. Et lors d'une opération *Défiler*, on ne pourra pas accéder à l'élément entré en premier, qui sera au « fond » de la pile : il faudrait dépiler, et donc perdre les autres éléments.

Supposons maintenant que nous utilisions **deux** piles. Alors il devient possible de les utiliser pour programmer une file. En effet, pour accéder au "fond" d'une pile, il suffit de la transférer élément par élément dans l'autre pile, ce qui a pour effet d'échanger l'ordre des éléments.

1. On appelle $P1$ et $P2$ les deux piles utilisées pour programmer la file. Exprimer en fonction des opérations de $P1$ et $P2$ toutes les opérations de la file.
2. Étudiez le coût des opérations de la file, **en fonction du coût des opérations de la pile** (on suppose que les deux piles $P1$ et $P2$ sont programmées de manière identique).

Correction de l'exercice 1

1. Il y a (au moins) 2 solutions :
   — solution 1 : la file est
      — soit entièrement dans $P1$ et le sommet de $P1$ est la queue de la file,
      — soit entièrement dans $P2$ et le sommet de $P2$ est la tête de la file.
      On peut passer de l'une de ces configurations à l'autre en *transférant* tous les éléments d'une pile dans l'autre, c'est-à-dire en dépilant d'une des pile les éléments un à un et en les empilant dans l'autre. On utilise $P1$ pour *Enfiler* et de $P2$ pour *Défiler*. Si la file n'est pas dans la bonne pile, on la *transfère* préalablement.
   — solution 2 (amélioration) : la file est répartie entre les 2 piles. $P1$ contient la "fin" de la file, et $P2$ le "début" de la file. Lorsqu'aucune des piles n'est vide, la tête de la file est le somment de $P2$ et la queue de la file est le somment de $P1$. le *transfert* n'est nécessaire que lorsqu'on veut *Défiler* et que $P2$ est vide (la tête de la file ne s'y trouve donc pas).



Les opérations de la file peuvent s'écrire ainsi :
Vider_File() :
  P1.Vider_Pile()
  P2.Vider_Pile()

File_Vide() :
  P1.Pile_Vide() And P2.Pile_Vide()

Opérations de transfert nécessaire :
Transferer_P1_P2() :
while *Not(P1.Pile_Vide()* **do**
  P1.Depiler(e)
  P2.Empiler(e)
(Idem pour Transferer_P2_P1)

Défiler(e) :
if *P2.Pile_Vide()* **then**
  Transferer_P1_P2()
P2.Dépiler(e)

Solution 1 :
Enfiler(e) :
if *P1.Pile_Vide()* **then**
  Transferer_P2_P1()
P1.Empiler(e) ;

Solution 2 :
Enfiler(e) :
P1.Empiler(e) ;

2. Soit n le nombre d'éléments présents dans la file.
   Notons $c_E(m)$ et $c_D(m)$ le coût des opérations Empiler et Dépiler pour une pile de taille m. Le coût de l'opération *Transférer* est

$$c_t(n) = \sum_{i=0}^{i=n-1} c_E(i) + c_D(n-i)$$

   Si l'on suppose que c_E et c_D sont constants (indépendants de la taille de la pile), ce qui est classique, on a $c_t(n) = n(c_E + c_D)$.
   On en déduit que le coût de l'opération *Défiler* est égal à c_D dans les cas favorables, et à $n(c_E + c_D) + c_D$ dans les cas défavorables.
   Dans la solution 1, le coût de l'opération *Enfiler* est égal à c_E dans les cas favorables, et à $n(c_E + c_D) + c_E$ dans les cas défavorables.
   Dans la solution 2, le coût de l'opération *Enfiler* est toujours égal à c_E. Le coût de *Vider_File* est égal à la somme des coûts de l'opération Vider_Pile pour une pile de n éléments et une pile vide. Raisonnablement, on peut négliger ce second coût.
   Le coût de *File_Vide* est égal à la somme des coûts de l'opération Pile_Vide pour une pile de n éléments et une pile vide. Raisonnablement, il s'agit d'un coût constant.

Exercice 2 :

Appelons **simulation** ce qui a été réalisé à la question 1. On a simulé une file à l'aide de deux piles ; cette simulation s'effectue sans recopie des éléments.
Est-il possible de

1. simuler une pile à l'aide d'une file ?

2. simuler une pile à l'aide d'une file à priorités ?

3. simuler une file à l'aide d'une file à priorités ?

4. simuler une file à priorités à l'aide d'une file ?

5. simuler une file à priorités à l'aide d'une pile ?

Justifiez vos réponses : dans le cas d'une réponse positive en donnant le principe des algorithmes permettant d'effectuer la simulation (sans écrire de "programme"), dans le cas d'une réponse négative en donnant un argument qui prouve l'impossibilité.

Correction de l'exercice 2

1. C'est possible, en mémorisant le nombre d'éléments présents dans la pile (une variable de type entier, nb, suffit). Pour dépiler, on accède au dernier élément de la file par un "parcours", en remettant dans la file les éléments sortis, ce qui en conserve l'ordre.
   Empiler(e) :
   $nb \leftarrow nb + 1$
   Entrer(e)

   Dépiler(e) :
   for $i \leftarrow 1$ **to** $nb - 1$ **do**
   Défiler(x)
   Enfiler(x)
   Défiler(e)
   $nb \leftarrow nb - 1$
2. C'est possible, en mémorisant le nombre d'éléments présents dans la pile (une variable de type entier, nb, suffit). Chaque élément a pour priorité son rang dans la pile, le dernier entré a la priorité maximale.
   Empiler(e) :
   $nb \leftarrow nb + 1$
   Entrer(e, nb)

   Dépiler(e) :
   Sortir(e, p)
   $nb \leftarrow nb - 1$
3. Il suffit de mémoriser le nombre d'éléments qui sont entrés dans la file (une variable de type entier, nb, suffit). Chaque élément a pour priorité l'opposé de sa "date d'entréé' dans la file, le dernier entré a la priorité minimale.
   ATTENTION. On ne peut pas utiliser une même priorité pour tous les éléments : à priorité égale, dans une file à priorités, les éléments sortent dans un ordre quelconque.

   Enfiler(e) :
   $nb \leftarrow nb + 1$
   Entrer(e, -nb)

   Défiler(e) :
   Sortir(e, p)
4. C'est possible, avec une file dont les éléments sont des couples (valeur, priorité), et en utilisant deux variables pr et pr_max de type priorité, deux variables val et val_m de type valeur, et un entier nb (nombre d'éléments présents dans la file). Pour sortir un élément de priorité maximale, on effectue un parcours de la file (voir (a) ci-dessus). L'ordre des éléments dans la file n'intervient pas, et peut être modifié.

```
Entrer(e, p) :
  nb ← nb + 1
  Enfiler((e, p))

Sortir(e, p) :
  Défiler((val_max, pr_max))
  for i ← 1 to nb − 1 do
  |  Défiler((val, pr))
  |  if pr > pr_max then
  |  |  Enfiler((val_max, pr_max))
  |  |  pr_max ← pr
  |  |  val_max ← val
  |  else
  |  |  Enfiler((val, pr))
  e ← val_max
  p ← pr_max
  nb ← nb − 1
```

5. Si c'était possible, on pourrait par transitivité simuler une file avec une pile. Cela contredirait ce qui a été montré au début de l'énoncé. Donc, c'est impossible.