Files - Diviser pour régner

1 Files d'attente.

1.1 Généralités

Dans une file de caisse d'un supermarché, la première personne qui est rentrée dans la file et la première qui doit en sortir. En informatique, nous avons aussi besoin de gérer des files. Par exemple, quand vous tapez au clavier, les caractères correspondant aux touches frappées sont stockés dans une file d'attente puis traités dans leur ordre d'arrivée par le programme ou le système d'exploitation pour être affichés à l'écran par exemple.

Pour les listes, il existe deux opérations de base : la fonction de mise en file d'attente et la sortie de la file d'attente.

- Dans la fonction d'ajout (*add* en anglais) d'un élément en fin d'une liste d'attente.
- Dans la fonction de prise (*take* en anglais) de l'élément qui est en tête de la file d'attente et que l'on le supprime de la file.

À ces deux opérateurs, il faut ajouter une fonction de création de file. Dans la littérature anglo-saxonne, les files se nomment *queue*. Dans une file, le premier élément rentré est le premier qui sort ce qui donne en anglais : «Firt In, First Out», on parle de façon abrégée de pile **FIFO**.

Nous allons donc simuler les files de deux façons, l'une utilisant des tableaux et l'autre des listes.

1.2 Mise en œuvre informatique.

1.2.1 utilisation de tableaux

Dans ce cas, on représente une liste à l'aide d'un enregistrement contenant :

- un tableau dans lequel sont stockés les éléments de la file,
- un entier «mutable» indiquant le premier élément de la file,
- un entier «mutable» indiquant la première case vide de la file,
- un entier donnant la longueur du tableau.

Lorsque les pointeurs sur la position de fin et de début de file sont égaux, c'est que la file est vide. Au fur et à mesure de l'ajout d'éléments dans la file, le pointeur de fin de liste est incrémenté modulo la longueur du tableau. De même, au fur et à mesure du retrait d'éléments de la liste, le pointeur est incrémenté modulo la longueur du tableau. On a ainsi une occupation «tournante» du tableau, le premier élément de la file n'étant pas à une place déterminé dans le tableau. On peut symboliquement représenter notre file sous la forme suivante :

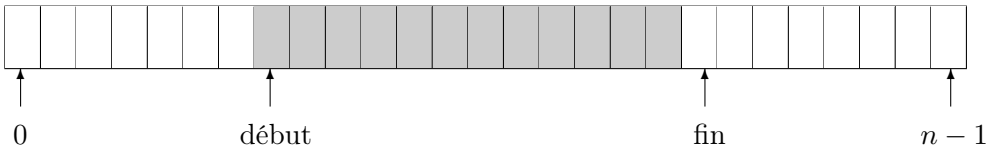

FIG. 1 – *représentation symbolique d'une file*

Pour différencier une file vide d'une file pleine, on dira que la file est pleine si la case du premier élément de la file est juste derrière la première case vide. En fait, si notre tableau est de longueur n, on n'utilisera au plus $n-1$ cases, la case précédant la case du premier élément de la file devant toujours rester vide .

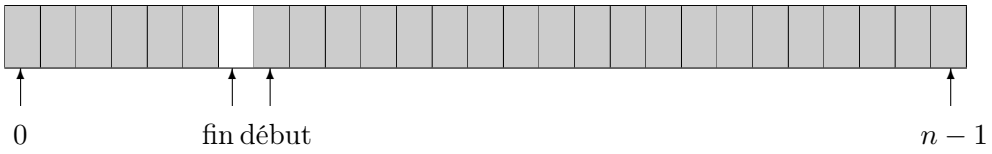

FIG. 2 – *représentation symbolique d'une file pleine*

```
#type 'a file =
{tab : 'a vect; mutable debut : int; mutable fin : int; long : int};;
Le type file est défini.
#let cree_file n a =
{tab = make_vect n a; debut=0; fin=0; long=n};;
cree_file : int -> 'a -> 'a file = <fun>
#let ajoute f a =
let pos = (f.fin+1) mod f.long in
  if pos = f.debut
      then failwith "file pleine"
      else (f.tab.(f.fin) <- a; f.fin <- pos);;
ajoute : 'a file -> 'a -> unit = <fun>
#let retire f =
if f.debut = f.fin
   then failwith "file vide"
   else (let a = f.tab.(f.debut) in f.debut <- (f.debut+1) mod
f.long; a);;
retire : 'a file -> 'a = <fun>
```

1.2.2 utilisation de listes

Dans le cas de l'utilisation des listes, c'est plus simple. La file est représentée par une liste «mutable». L'élément en tête de liste représente le début de la file et l'élément en queue de liste représente la fin de la file. Donc logiquement, pour ajouter un élément à une file ainsi représentée, on l'ajoute en queue de liste et pour retirer un élément, on prend celui qui est en tête de liste.

```
#type 'a file = {mutable file : 'a list};;
Le type file est défini.
#let créefile () = {file = []};;
créefile : unit -> 'a file = <fun>
```

```
#let ajoute f x =
    f.file <- f.file@[x];;
ajoute : 'a file -> 'a -> unit = <fun>
#let enleve f =
match f.file with
   [] -> failwith "enleve : file vide"
| x::xs -> f.file <- xs; x;;
enleve : 'a file -> 'a = <fun>
```

1.2.3 En utilisant les bibliothèques Caml.

Les files étant régulièrement utilisées en informatique, on dispose d'une bibliothèque de fonctions permettant de créer et de gérer des files (*queue* en anglais) avec Caml.

```
##open "queue";;
#add;;
- : 'a -> 'a t -> unit = <fun>
#take;;
- : 'a t -> 'a = <fun>
#new;;
- : unit -> 'a t = <fun>
```

2 Le principe de «diviser pour régner» en informatique

2.1 Généralité

Nous avons déjà rencontré sans en avoir parlé le principe de «diviser pour régner» en programmation. Quel est ce principe? Pour un gros volume de données traitées, il s'agit dans un premier temps de partitionner ce volume de données en p sous parties à peu près égales puis d'appliquer l'algorithme sur chacune de ces p parties et enfin de fusionner les résultats. Dans les cas les plus couramment rencontrés, on a $p = 2$.

Ce principe fait penser bien sûr au tri par fusion. En effet, dans le cas du tri par fusion, on coupe une liste en deux parties à peu près égales, on applique le principe du tri par fusion sur chacune des listes puis on fusionne les deux sous listes ainsi triées pour obtenir la liste de départ triée.

2.2 tri rapide

Voici un exemple du principe de «diviser pour régner» pour trier des listes différent de celui de tri fusion. Dans le tri fusion, lorsque l'on sépare la liste en deux listes, on le fait de manière *aveugle* en prenant un élément sur deux dans chacune des listes. Dans le cas du `tri rapide`, on effectue une opération de tri au moment de la séparation de la liste en deux suivant le principe suivant. Si la liste n'est pas vide, on considère le premier élément x de la liste, puis on fabrique deux sous listes, la première étant constituée des éléments plus petits que x et la deuxième par ceux qui sont plus grands que x. On applique le *tri rapide* à chacune des sous listes plus pour obtenir la liste de départ triée, il suffit de recoller la première sous liste triée puis x puis la deuxième sous liste triée. Contrairement au cas du *tri fusion*, nous ne sommes pas sûr d'avoir deux sous listes de longueurs à peu près égales (et ce ne sera jamais le cas si on travaille sur un

tableau déjà trié au départ). Cet inconvénient est compensé par le fait que le travail de fusion des deux sous listes est très simple.

```
let rec tri_rapide l =
let rec separe x = function
  [] -> [],[]
| [y] -> if y<x then [y],[] else [],[y]
| y::ys -> let l1,l2 = separe x ys in
           (match y<x with
              true -> (y::l1),l2
            | false -> l1,(y::l2))
in
match l with
 [] -> []
| x::xs -> let l1,l2 = separe x xs in
            (tri_rapide l1)@(x::tri_rapide l2);;
```

Version tableau de la fonction *tri rapide*.

```
let ttri_rapide t =
let echange t i j =
  let c = t.(i) in
  t.(i) <- t.(j); t.(j) <- c in
let rec tri t i j n =
  match j-i > 0 with
     false -> ()
   | true -> let m = ref i in
             for k = i+1 to j do
                if t.(k) < t.(i) then (m :=  !m + 1; echange t !m k)
             done;
             if !m < n
                then echange t i !m
                else echange t i n;
             tri t i (!m-1) n; tri t (!m+1) j n
in
let n = vect_length t-1 in
tri t 0 n n;;
```

2.3 calcul de puissance rapide

Le principe «diviser pour régner» ne s'applique pas seulement à des ensembles que l'on peut partager en p sous ensembles mais peut très bien s'appliquer à des problèmes mettant en jeu des entiers. Quand on veut calculer x^N avec N entier, on peut remarquer simplement que $x^0 = 1$ et $x^{N+1} = x * x^N$ ce qui conduit à un algorithme linéaire en $O(N)$. La programmation en est simple mais on atteint rapidement les limites de cette solution si on l'utilise avec des grandes valeurs de N.

On peut aussi remarquer que $x^N = (x^2)^p$ si $N = 2 * p$ et $x^N = x \times (x^2)^p$ si $N = 2p + 1$. Essayons d'évaluer le temps de calcul d'un tel procédé.

Cet algorithme appliqué aux entiers Caml donne :

```
#let rec puis x = function
 0 -> 1
| n when n mod 2 = 0 -> let x' = puis x (n/2) in x'*x'
| n -> let x' = puis x (n/2) in x*x'*x';;
puis : int -> int -> int = <fun>
#puis 2 10;;
- : int = 1024
```

En raison du typage des fonctions en Caml nous sommes obligés de réécrire la fonction pour l'appliquer aux réels :

```
#let rec puis x = function
 0 -> 1.
| n when n mod 2 = 0 -> let x' = puis x (n/2) in x'*.x'
| n -> let x' = puis x (n/2) in x*.x'*.x';;
puis : float -> int -> float = <fun>
#puis 2. 12;;
- : float = 4096.0
```

Plutôt que de réécrire 50 fois la même fonction pour des types différents, on peut créer une fonction puissance générique qui s'applique à tous les types de données à condition de préciser quel est l'opérateur utilisé : multiplication entre entiers, multiplication entre matrices,.... À partir de cette fonction générique, on peut créer toutes les fonctions puissances désirées. Illustration :

```
#let rec puiss operateur x = function
 1 -> x
| 2 -> operateur x x
| n when n mod 2 = 0 -> let x' = puiss operateur x (n/2) in
                         operateur x' x'
| n -> let x' = puiss operateur x (n/2) in
       operateur x (operateur x' x');;
puiss : ('a -> 'a -> 'a) -> 'a -> int -> 'a = <fun>
#let puiss_int = puiss mult_int;;
puiss_int : int -> int -> int = <fun>
#puiss_int 2 10;;
- : int = 1024
#let puiss_float = puiss mult_float;;
puiss_float : float -> int -> float = <fun>
#puiss_float 2.5 10;;
- : float = 9536.74316406
```

remarque : les fonctions ainsi définies sont un peu différentes des cas précédents car elles ne traitent pas le cas d'un élément élevé à la puissance 0. Ce cas ne peut pas être traité directement dans la fonction générique sans typer la fonction puisqu'il faut retourner dans ce cas l'élément neutre de l'opérateur. On peut trouver une solution à ce problème en rajoutant un argument à la fonction `puiss` qui sera l'élément neutre de l'opérateur ce qui nous permet de définir le cas $n = 0$. De même, il faut traiter le cas des puissances négatives si on ne veut pas voir le programme boucler indéfiniment et finir par un `Exception non rattrapée: Out_of_memory`.