Auteur externe



Zeste de savoir

# Tri par insertion : le retour (OCaml)

12 août 2019

# Table des matières





Ceci est une deuxième version du tutoriel déjà existant qui présente le tri par insertion.

Le principe de l'algorithme sera repris, cette fois dans un langage fonctionnel (ici, OCaml ). Ce tuto se concentrera donc principalement, non pas sur le principe de l'algorithme, qui est supposé connu, mais sur les améliorations non négligeables qu'apporte le style de programmation fonctionnel.

Le langage choisi est OCaml, mais si vous vous intéressez à la programmation fonctionnelle, il n'est pas nécessaire de le connaître pour lire ce code (même si cela vous paraîtra sûrement un peu étrange la première fois).

Au menu : utilisation des listes, récursivité , filtrages de motifs, tri générique et polymorphisme, et petites particularités de style.

## 1. Les listes

Une liste est une structure de données fondamentale en programmation : elle permet de représenter une suite d'éléments, et de les manipuler très naturellement.
Cette structure est très adaptée à certains algorithmes de tri, entre autres le tri par insertion. Cependant, elle n'est pas immédiate d'accès en C (il faut implémenter une liste chaînée soi-même : si cela vous intéresse vous pouvez regarder ce tuto ) ; c'est pourquoi je n'en ai pas parlé dans mon précédent tuto. Maintenant qu'on a un langage fonctionnel dans lequel ce type de données est accessible directement, on peut en profiter.

La liste est un type de données très intéressant, car il est récursif par nature. En effet, on définit formellement une liste de la manière suivante : une liste est :

— soit une liste vide (notée []) ;

— soit un élément suivi d'une liste : on note a : :b, a étant l'élément et b la liste ; dans certains langages, on note plutôt (cons a b) ; dans le dernier cas, on appellera l'élément la "tête", et la liste qui le suit la "queue" de la liste.

Cette définition, très élégante (on n'a pas parlé de pointeurs), appelle directement à utiliser la liste avec des algorithmes récursifs.

## 2. Découpage de l'algorithme

L'algorithme du tri par insertion est conservé : on découpe toujours le code en deux fonctions, la fonction "insérer" qui insère un élément à la bonne position dans la liste, et la fonction "tri_insertion", qui réutilise la fonction insérer pour trier le tableau, en insérant chaque élément successivement à la bonne place.

### 2.0.1. Insertion

Voici le code d'insertion. Il insère un élément dans une liste triée en ordre croissant, de manière à renvoyer une liste toujours triée.

```
let rec insere elem liste = match liste with
| [] -> elem::[]
| tete::queue ->
    if elem < tete then elem :: liste
    else tete :: insere elem queue
```

Le code se lit de lui-même : les deux arguments de la fonction insere sont elem et liste. On "regarde" la liste (la structure match .. with .. est un filtrage de motif) :

— si c'est la liste vide (premier |, premier cas), on renvoie l'élément suivi de la liste vide ;
— si la liste est une tête suivie d'une queue, on compare l'élément et la tête :
  — si l'élément est plus petit, on a trouvé la bonne place pour le mettre : on renvoie donc l'élément, suivi de la liste,
  — sinon (si l'élément est plus grand), on renvoie la tête, suivie de la queue dans laquelle on a inséré elem.

### 2.0.2. Tri

Maintenant qu'on a fait le gros du boulot, la fonction de tri vient toute seule :

```
let rec tri_insertion = function
| [] -> []
| tete::queue -> insere tete (tri_insertion queue)
```

"function" est un mot-clé qui prend un argument, et lui applique un filtrage de motif. "let rec tri = function" est équivalent à "let rec tri liste = match liste with", en plus concis. Là encore, l'algorithme est clair :

— si la liste est vide, on la renvoie ;
— si la liste est une tête suivie d'une queue, on trie la queue, et on y insère la tête.

## 3. Une amélioration : le tri générique

Notre fonction trie une liste en ordre croissant. Très bien. Mais si on voulait trier en ordre décroissant, il faudrait la recoder ?

Une solution serait en effet de recoder la fonction insertion, en remplaçant le test "if elem < tete" par "if elem > tete". Mais heureusement, les langages fonctionnels permettent, comme leur nom l'indique, de manipuler très simplement et très efficacement les fonctions. Il suffit donc de donner un argument supplémentaire à la fonction, qui soit une fonction de comparaison indiquant de quelle manière on doit trier les éléments :

```
let rec insere comparaison elem liste = match liste with
|  [] -> elem::[]
|  tete::queue ->
   if comparaison elem tete then elem :: liste
   else tete :: insere comparaison elem queue

let rec tri_insertion comp = function
|  [] -> []
|  tete::queue -> insere comp tete (tri_insertion comp queue)
```

J'ai rajouté un argument "comparaison" à la fonction insérer, et le test est maintenant "if comparaison elem tete". Il faut aussi rajouter l'argument à la fonction tri_insertion, puisqu'elle utilise insere.

On peut maintenant faire un tri décroissant, ou un tri croissant. Si vous avez installé ocaml, vous pouvez tester. Mettez le code dans un fichier "tri.ml", et dans le répertoire dans lequel vous avez mis votre fichier, lancez en ligne de commande "ocaml". Cela lancera l'interpréteur interactif ocaml. Ensuite, entrez "#use "tri.ml" ; ;", cela chargera le fichier de code, et vous affichera les fonctions déclarées.
Vous pouvez maintenant essayer les deux lignes suivantes (le code à entrer commence après le # et va jusqu'au ; ;) :

> ## 4. tri_insertion ( < ) [4; 2; 5; 1];;
>
> — : int list = [1 ; 2 ; 4 ; 5]

> **5. tri_insertion ( > ) [4; 2; 5; 1];;**
>
> — : int list = [5 ; 4 ; 2 ; 1]

Le résultat est concluant : la première fois, il a trié en ordre croissant, la deuxième fois (avec la fonction "strictement supérieur à") en ordre décroissant.

## 6. Profitons-en : le polymorphisme

Si vous avez essayé le code ocaml, vous avez sûrement vu deux lignes étranges au moment de l'inclusion du fichier source :
`val insere : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun> val tri_insertion : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun></fun></fun>`

Ce code indique que vous avez déclaré deux fonctions, et, c'est le plus important, donne leur type. OCaml étant un langage fonctionnel typé, le type est très important, et c'est un outil très très puissant pour programmer. Il permet de vérifier que les fonctions sont utilisées sûrement, et donne un gros indice sur leur signification (d'habitude, le nom d'une fonction et son type suffisent pour comprendre son fonctionnement ; pas besoin de regarder le code ou de lire la doc).

Le type de la fonction insertion (insere étant une fonction auxiliaire, on la laissera de côté) est `('a -> 'a -> bool) -> 'a list -> 'a list.`
Il faut savoir que le type d'une fonction qui prend un argument de type "int" et renvoie un argument de type "char" (par exemple) est "`int -> char`". Quand il y a deux arguments (par exemple deux entiers), le type est "`int -> int -> char`". Ici, il y a deux arguments, le premier étant de type `('a -> 'a -> bool)`, le deuxième de type `'a list`. Le premier est une fonction (c'est `comp`) qui prend deux arguments de type `'a` et renvoie un booléen (vrai ou faux, utilisé dans le `if`), et le deuxième est une liste d'éléments de type `'a`.

Que signifie le `'a` ? Pourquoi pas un type normal comme `int`, `char` ou `bool` ? 'a signifie en fait "n'importe quel type". Quand un argument est de type `'a`, il peut être de n'importe quel type, du moment que tous les `'a` d'un type désignent le même type. Par exemple, `'a -> 'a` est une fonction qui prend un argument d'un type donné et renvoie un argument du même type. Si le type de retour était n'importe quel type, pas forcément le même, on noterait `'a -> 'b`.

Ainsi, OCaml nous dit que l'on peut utiliser notre tri avec n'importe quel type d'objet. C'est assez magique, quand on pense que le tri précédent (en C) ne permettait de trier que des entiers (et à la rigueur des flottants, ou des char) !
Par exemple, on peut faire un tri sur des couples d'entiers (on prendra le classique tri lexicographique) :

```
let compare_couples (a, b) (c, d) =
  if a - c < 0 then true
  else if a = c then b < d
  else false

let tri_couples = tri_insertion compare_couples
```

La première fonction est une fonction de comparaison de couples assez simple : si la différence des deux premiers éléments est négative, on sait que le premier couple est plus petit que le second. Sinon, si les deux premiers éléments sont égaux, on compare les seconds éléments de chaque couple, et sinon, on sait que le deuxième est plus grand.

La fonction tri_couples qui est déclarée permet de trier, en réutilisant tri_insertion, des listes de couples :

```
# tri_couples [(3, 4); (1, 2); (1, 5)];;
- : (int * int) list = [(1, 2); (1, 5); (3, 4)]
```

Le polymorphisme nous permet donc, en 10 lignes de code, de faire une fonction permettant de trier n'importe quel type d'objet de manière sûre (le compilateur vérifie que les objets de la liste sont bien du bon type).

## 8. Coquetteries

Le code est déjà bien sympathique, mais il est possible de faire de petites améliorations.

La première, c'est l'utilisation d'une fonction pour cacher la fonction auxiliaire insere : étant donné qu'on veut juste un tri, on n'a pas besoin que la fonction insere soit disponible au reste du programme. Il est même possible qu'une fonction s'appelant insere existe déjà, et on n'a pas envie d'être obligé de la renommer.

En OCaml (comme dans tous les langages fonctionnels), on peut déclarer des fonctions à l'intérieur des déclarations (de fonctions ou de variables, c'est la même chose). Leur portée est alors limitée (comme en C où les variables "vont jusqu'à la fin du bloc") :

```
let tri_insertion comparaison liste =
    let rec insere elem liste = match liste with
    | [] -> elem::[]
    | tete::queue ->
        if comparaison elem tete then elem :: liste
        else tete :: insere elem queue
    in

    let rec tri = function
    | [] -> []
    | tete::queue -> insere tete (tri queue)
    in
    tri liste
```

Les deux fonctions insere et tri sont maintenant des fonctions locales (d'où le "in") à l'intérieur d'une fonction tri_insertion globale. Un autre avantage de cet englobement, c'est que la fonction comp, qui est un paramètre de la fonction globale, est accessible directement aux fonctions insere et tri, sans qu'elles n'aient besoin de le passer en argument. On a donc allégé l'écriture de ces fonctions.

En fait, on peut même faire mieux, d'une part en supprimant l'argument "liste" qui est redondant ("let fonction argument = autre_fonction argument", c'est pareil que "let fonction = autre_fonction"), et d'autre part en changeant le nom de "comparaison" en ( < ) : ainsi, on pourra utiliser la fonction de comparaison comme l'opérateur <, à l'intérieur de la fonction tri_insertion (les parenthèses qui englobent < servent à indiquer que c'est un opérateur) :

```
let tri_insertion (<) =
    let rec insere elem liste = match liste with
    | [] -> elem::[]
    | tete::queue ->
        if elem < tete then elem :: liste
        else tete :: insere elem queue
    in

    let rec tri = function
    | [] -> []
    | tete::queue -> insere tete (tri queue)
    in
    tri
```

---

J'espère que ce tutoriel vous a permis de constater une partie des avantages concrets que peuvent apporter les langages fonctionnels, dans la conception et formulation des algorithmes. Les propriétés fortes qu'apportent le style de programmation fonctionnelle permettent aussi de démontrer extrêmement facilement la correction de l'algorithme : on peut prouver en quelques lignes que l'algorithme renverra toujours le bon résultat, autrement dit qu'il ne contient aucun bug. La preuve est beaucoup plus difficile pour une version impérative de l'algorithme (par exemple, ma version C précédente).

Have fun !