Université Paris Diderot - L2 informatique Année 2014–2015

TD d'Éléments d'Algorithmique n° 4
(Correction)

I) Arbres binaires

Exercice 1. *Taille et hauteur d'un arbre.*

- Appelons *taille* d'un arbre son nombre total de nœuds. Écrire un algorithme `taille` qui explore un arbre binaire et renvoie sa taille.
- La *hauteur* d'un arbre est le nombre maximum de noeuds rencontrés en partant de la racine et en descendant vers les feuilles. Ecrire un algorithme `hauteur` explorant un arbre binaire et renvoyant sa hauteur.

Exercice 2. *Parcours.*

Écrire trois algorithmes affichant le contenu d'un arbre binaire selon les ordres préfixe, infixe et suffixe.

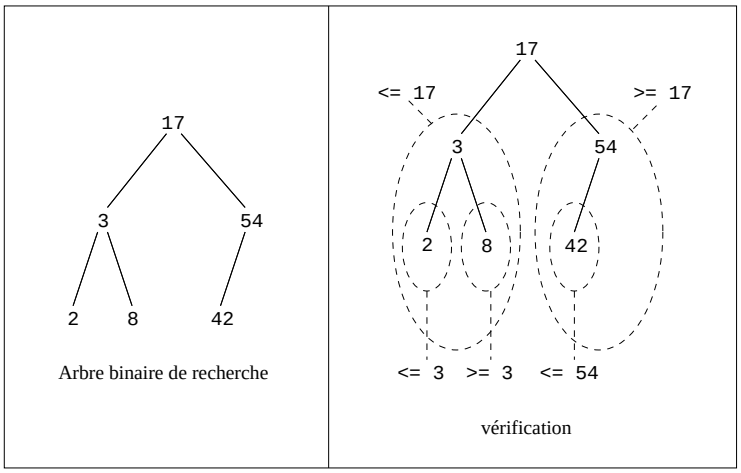
Exercice 3. *Tri à l'aide d'un ABR.*

On appelle *arbre binaire de recherche*, ou *ABR*, un arbre dont *chaque* noeud x vérifie la propriété suivante : si `n` est la clef étiquetant le nœud x, alors,

- toutes les clefs apparaissant dans le sous-arbre gauche du nœud x sont plus petites que `n`,
- et toutes les clefs apparaissant dans le sous-arbre droit du nœud x sont plus grandes que `n`.

Par convention, l'abre vide est considéré comme un ABR.

L'arbre suivant est un exemple d'ABR, comme on peut le vérifier en examinant chaque nœud :



- Vérifier sur un exemple que l'affichage d'un ABR dans l'ordre infixe affichera la suite des clefs de cet arbre dans l'ordre croissant.

Les ABR peuvent servir à trier une suite d'entiers différents. Il suffit d'ajouter successivement ces entiers à un arbre initialement vide, puis de le parcourir en ordre infixe.

- Écrire un algorithme `ajouter_abr` qui rajoute à un ABR, éventuellement vide, un nouveau noeud contenant la clef `n` de manière à ce que l'arbre résultant soit encore un ABR.

II) Backtrack

Exercice 4. *Les perles de Dijkstra.*

On souhaite construire des séquences composées de trois symboles, par exemple les lettres `a`, `b` et `c`, respectant une seule contrainte : il ne doit pas y avoir deux sous-séquences adjacentes identiques.

Par exemple `abacba` est correcte, tandis que `abaaba` ne l'est pas. En fait, `abaaba` possède deux paires de sous-séquences adjacentes identiques : `a`,`a` et `aba`,`aba`. tandis que `abacba` n'en possède aucune.

Le but de cet exercice est de programmer un algorithme qui génère les séquences correctes, jusqu'à une longueur donnée, en utilisant le backtrack.

Si on implémente les séquences par des listes, et on ajoute les symboles en tête en commençant par la liste vide, il est clair que les séquences obtenues seront correctes jusqu'à ce que, éventuellement, deux sous-séquences adjacentes identiques apparaissent en tête de la liste.

1. Écrire une fonction `test` qui prend comme paramètre une liste `l`, et qui teste si deux sous-listes adjacentes identiques apparaissent en tête de `l`.
2. Écrire une fonction `backtrack` qui prend comme paramètres une liste `l` et un entier `n` et qui affiche toute les séquences correctes de longueur au plus `n` ayant comme suffixe la séquence représentée par `l`.
3. Écrire une fonction `main`, avec un paramètre entier `n`, qui appelle `backtrack` á partir de la liste vide et de `n`. Exécuter l'appel de cette fonction sur l'entier 3.

Voici, pour quelques valeurs de n, le nombre totales de séquences de longueur au plus n, et le nombre des séquences correctes de longueur au plus n.

	$\sum_{i=0}^{n} 3^i$	$C(n)$
$n = 3$	40	22
$n = 5$	364	70
$n = 8$	9841	250
$n = 12$	797161	970
$n = 17$	193710244	4228

4. En supposant que $C(n) \in \Theta(n^3)$, calculer la complexité de l'algorithme du point 3.
5. Calculer la complexité de l'algorithme qui énumère toutes les séquences, et filtre celle qui sont correctes (il faudra en particulier redéfinir `test`. Les tableaux sont plus appropriés que les listes pour représenter les séquences, dans ce cas).

Correction :

```
type perle = A | B | C;;

(* coupe prend un collier (liste de perles) l et un entier n <= de sa longueur
et renvoie deux listes de long. n et length(l)-n respectivement *)

let  rec coupe l n = if n=0 then [],l else match l with
a::l1 -> let (l2,l3)= coupe l1 (n-1) in a::l2,l3
  | _ -> failwith ("impossible");;

(* test prend deux listes, dont la premiere n'est pas plus longue que la
```

```
deuxième, et renvoie true si la premiere est un prefixe de la 2eme,
false sinon *)

let rec test (l1,l2)= match l1,l2 with
[],_-> true
  | a::l3,b::l4 when a=b -> test (l3,l4)
  | _-> false;;

(* test2 prend une liste et teste si un carre commence au début
 de cette liste. Pour ce faire, on considère
 des prefixes de plus en plus longues,
jusqu'à la moitie de la longueur de la liste, et on utilise test*)

let test2 l =
  let rec aux l n = if n> List.length(l)/2 then false
    else let (l1,l2)=coupe l n in test (l1,l2)||(aux l (n+1))
  in aux l 1;;

(* affichage *)
let affiche_perle p= match p with
 A->print_char 'A'
  |B->print_char 'B'
  |C->print_char 'C';;

let affiche_collier l=List.iter affiche_perle l; print_newline();;

(* la fonction qui engendre les colliers de D., jusqu'à une  longueur donnée *)

let rec backtrack l n=
if List.length l <= n && test2 l then begin
affiche_collier l;
backtrack (A::l) n;
backtrack (B::l) n;
backtrack (C::l) n;
end ;;


let main n =
print_string "Voici les colliers des Dijkstra jusqu'à la longueur ";
print_int(n);
print_string ":\n";
backtrack [] n;;
```