

Projet de logique & structures informatiques
L2 informatique
2020-2021

Projet **Arbre 2-3-4**

DOURLEN Maxime
Groupe TP3

Sommaire :

Partie 1 : Notice d'utilisation

Partie 2 : Choix de programmation

Partie 3 : Liste des problèmes rencontrés

Partie 4 : Preuves des algorithmes utilisés

```
ab234_vers_abic : ab234 -> abic
abic_vers_ab234 : abic -> ab234
```

Pour insérer une valeur dans un arbre 2-3-4 :

`insérer : element -> ab234 -> ab234`

Pour créer des arbres 2-3-4 aléatoires :

`random_ab234 : int(bound) -> int(nombre d'ajout) -> ab234`

Pour supprimer une valeur dans un arbre 2-3-4 :

`supp_ab234 : element -> ab234 -> ab234`

Pour faire l'union de deux arbres 2-3-4 :

`union : ab234 -> ab234 -> ab234`

Pour faire l'intersection de deux arbres 2-3-4 :

`intersection : ab234 -> ab234 -> ab234`

Pour faire la différence de deux arbres 2-3-4 :

`dif : ab234 -> ab234 -> ab234`

Pour faire la différence symétrique de deux arbres 2-3-4 :

`delta : ab234 -> ab234 -> ab234`

Pour faire un test d'égalité entre deux arbres 2-3-4 :

`egalite : ab234 -> ab234 -> bool`

Pour faire un test d'inclusion entre deux arbres 2-3-4 :

`inclusion : ab234 -> ab234 -> bool`

Pour l'équivalent des fonctions `List.fold_left` et `List.fold_right` sur des arbres 2-3-4 :

`left_fold : ('a -> element -> 'a) -> 'a -> ab234 -> 'a`

`right_fold : (element -> 'a -> 'a) -> 'a -> ab234 -> 'a`

Pour calculer le cardinal d'un ensemble :

`cardinal : ab234 -> int`

Pour utiliser la fonction de séparation :

`separation : element -> ab234 -> ab234 * ab234`

Pour utiliser la fonction de filtrage :

`filter : (element -> bool) -> ab234 -> ab234`

Partie 2 : Choix de programmation

Pour le type de données ab234, j'ai choisi celui qui me paraissait le plus simple et évident par rapport à ce que nous avons vu en cours. Ce type permet d'être manipulé facile et polymorphe en modifiant le type de element.

```
1(* definition de arbre234 *)
2type element = int;;
3type ab234 = Vide
4         | Noeud1 of ( element * ab234 * ab234 )
5         | Noeud2 of ( element * element * ab234 * ab234 * ab234 )
6         | Noeud3 of ( element * element * element * ab234 * ab234 * ab234 * ab234 * ab234 );;
```

Les fonctions est_ab234 et est_dans234 sont des fonctions inspirées de ce que nous avons fait lors des TP.

```
79(* verifie si l'arbre fourni en parametres est un arbre 234 *)
80let est_234 ab =
81  (hauteur ab && est_recherche234 ab);;
82
83(* Q1.3 *)
84(* recherche dans un arbre234 *)
85let rec est_dans x = function
86  | Vide -> false
87  | Noeud1(r, _, _) when r == x -> true
88  | Noeud2(r, t, _, _, _) when r == x || t == x -> true
89  | Noeud3(r, t, y, _, _, _, _) when r == x || t == x || y == x -> true
90  | Noeud1(r, ag, _) when x < r -> est_dans x ag
91  | Noeud1(r, _, ad) -> est_dans x ad
92  | Noeud2(r, _, ag, _, _) when x < r -> est_dans x ag
93  | Noeud2(_ , r, _, _, ad) when x > r -> est_dans x ad
94  | Noeud2(_ , _, _, am, _) -> est_dans x am
95  | Noeud3(r, _, _, ag, _, _, _) when x < r -> est_dans x ag
96  | Noeud3(_ , r, _, _, amg, _, _) when x < r -> est_dans x amg
97  | Noeud3(_ , _, r, _, _, amd, _) when x < r -> est_dans x amd
98  | Noeud3(_ , _, _, _, _, _, ad) -> est_dans x ad;;
```

Pour la fonction d'insertion, j'ai choisi de mettre les cas d'insertions immédiates pour limiter le temps d'exécution dans les cas où la cible est trouvée rapidement.

```

127(* fonction permettant l'insertion d'une valeur dans ab234 *)
128let insérer v a =
129  let rec insertion_aux = function
130    (* cas direct *)
131    | Vide -> Noeud1(v, Vide, Vide)
132    | Noeud1(r, _, _) as a when r == v -> a
133    | Noeud2(r, t, _, _) as a when r == v || t == v -> a
134    | Noeud3(r, t, y, _, _) as a when r == v || t == v || y == v -> a
135    (* cas où il faut insérer *)
136    (* normalement ici tout les sous arbres sont Vide *)
137  | Noeud1(r, Vide, ad) when v < r -> Noeud2(v, r, Vide, Vide, ad)
138  | Noeud1(r, ag, Vide) -> Noeud2(r, v, ag, Vide, Vide)
139  | Noeud2(r, y, Vide, am, ad) when v < r -> Noeud3(v, r, y, Vide, Vide, am, ad)
140  | Noeud2(y, r, ag, Vide, ad) when v > r -> Noeud3(y, r, v, ag, Vide, Vide, ad)
141  | Noeud2(y, r, ag, am, Vide) -> Noeud3(y, v, r, ag, am, Vide, Vide)
142  (* cas où il faut avancer tant que ce n'est pas un Noeud externe *)
143  (* fonction équilibrer à chaque noeud passé *)
144  | Noeud1(r, ag, ad) when v < r -> équilibrer (Noeud1(r, (insertion_aux ag), ad))
145  | Noeud1(r, ag, ad) -> équilibrer (Noeud1(r, ag, (insertion_aux ad)))
146  | Noeud2(r, y, ag, am, ad) when v < r -> équilibrer (Noeud2(r, y, insertion_aux ag, am, ad))
147  | Noeud2(y, r, ag, am, ad) when v > r -> équilibrer (Noeud2(y, r, ag, am, insertion_aux ad))
148  | Noeud2(y, r, ag, am, ad) -> équilibrer (Noeud2(y, r, ag, insertion_aux am, ad))
149  | Noeud3(r, x, y, ag, amg, amd, ad) when v < r -> équilibrer (Noeud3(r, x, y, insertion_aux ag, amg, amd, ad))
150  | Noeud3(x, r, y, ag, amg, amd, ad) when v < r -> équilibrer (Noeud3(x, r, y, ag, insertion_aux amg, amd, ad))
151  | Noeud3(x, y, r, ag, amg, amd, ad) when v < r -> équilibrer (Noeud3(x, y, r, ag, amg, insertion_aux amd, ad))
152  | Noeud3(x, y, r, ag, amg, amd, ad) -> équilibrer (Noeud3(x, y, r, ag, amg, amd, insertion_aux ad))
153  in insertion_aux (équilibrer a);;

```

Pour la fonction de suppression j'ai choisi de faire deux fonctions d'équilibrage, une fonction d'équilibrage de la racine (équilibrer_supp_rac) cette fonction regroupe les cas d'équilibrage où la racine est Noeud1 comme le montre les figures 8 et 9. Une fonction d'équilibrage globale (équilibrer_supp) qui a pour but de transformer tous les noeud2 et noeud 3 en noeud équilibré

pour le sous arbre ciblé, comme le montre la figure 10.

Pour la fonction de suppression en elle-même (supp_ab234), j'ai choisi de mettre les cas où on supprime un Noeud complet en dernier pour éviter au maximum de créer un arbre déséquilibré. J'ai également choisi de faire des tours où cette fonction ne fait qu'équilibrer un Noeud parce que sinon la modification de ce noeud n'était pas prise en compte et le sous arbre dans lequel la fonction allait n'était pas équilibré, ce qui a pour effet d'augmenter la complexité de la fonction.

En ce qui concerne les fonctions de la question 5, j'ai choisi de passer par des listes, ce qui simplifie les fonctions, pour cela il m'a fallu une fonction qui ajoute toutes les valeurs d'un arbre 234 dans une liste(ab234_to_list), et une qui ajoute toutes les valeurs d'une liste dans un arbre(add_list), mais également sont opposés qui supprime toutes les valeurs d'un liste dans un arbre(supp_list).

```

295 (* transforme un arbre 234 en liste *)
296 let rec ab234_to_list = function
297 | Vide -> []
298 | Noeud1(e1, a1, a2) -> e1::(ab234_to_list a1) @ (ab234_to_list a2)
299 | Noeud2(e1, e2, a1, a2, a3) -> e1::e2::(ab234_to_list a1) @ (ab234_to_list a2) @ (ab234_to_list a3)
300 | Noeud3(e1, e2, e3, a1, a2, a3, a4) -> e1::e2::e3::(ab234_to_list a1) @ (ab234_to_list a2) @ (ab234_to_list a3) @ (ab234_to_list a4);;
301
302 (* Ajout de tout elements d'une list dans un arbre 234 *)
303 let add_list xs ab =
304 |> List.fold_left (fun acc x -> inserer x acc) ab xs;;
305 (* supprime tout elements d'une list dans un arbre 234 *)
306 let supp_list xs ab =
307 |> List.fold_left (fun acc x -> supp_ab234 x acc) ab xs;;
308
309 (* Cree un arbre 234 a partir d'une list *)
310 let from_list xs = add_list xs Vide;;

```

Pour les fonctions de la question 6, respectivement nommées left_fold et right_fold, j'ai choisi de prendre une liste pour avoir toutes les branches me restant à parcourir dans une liste comme nous avons pu le voir en TP. Ainsi on est sûr d'avoir passé toutes les branches.

```

352 (* fonction equivalente a List.fold_(left | right) *)
353 let left_fold f init ab =
354 |> let rec aux acc = function
355 | [] -> acc
356 | a::rest -> match a with
357 |> Vide -> aux acc rest
358 |> Noeud1(e1, a1, a2) -> aux (f acc e1) (a1::a2::rest)
359 |> Noeud2(e1, e2, a1, a2, a3) -> aux (f (f acc e1) e2) (a1::a2::a3::rest)
360 |> Noeud3(e1, e2, e3, a1, a2, a3, a4) -> aux (f (f (f acc e1) e2) e3) (a1::a2::a3::a4::rest)
361 |> in aux init [ab];
362
363 let right_fold f init ab =
364 |> let rec aux acc = function
365 | [] -> acc
366 | a::rest -> match a with
367 |> Vide -> aux acc rest
368 |> Noeud1(e1, a1, a2) -> aux (f e1 acc) (a1::a2::rest)
369 |> Noeud2(e1, e2, a1, a2, a3) -> aux (f e1 (f e2 acc)) (a1::a2::a3::rest)
370 |> Noeud3(e1, e2, e3, a1, a2, a3, a4) -> aux (f e1 (f e2 (f e3 acc))) (a1::a2::a3::a4::rest)
371 |> in aux init [ab];

```

Partie 3 : Liste des problèmes rencontrés

Les problèmes que j'ai rencontrés :

-Fonction `equilibrer_supp` : J'ai eu du mal à lister tous les cas sans faire des cas inutiles, j'ai également eu des erreurs de frappe qui ont été compliquées à repérer.

-Fonction `supp_ab234` : j'ai eu un problème qui venait d'une erreur de signe « > » qui aurait dû être « < ». J'ai également eu un problème sur mon équilibrage qui ne se faisait pas et donc ma fonction faisait une boucle infini, j'ai réglé ce problème en faisant un tour de `supp_aux` juste pour équilibrer.

-Fonction `egalite` : j'ai eu un problème qui m'est apparu lors de mes tests sur cette fonction. Les ensembles que j'avais choisis étaient inclus le premier dans le deuxième et donc la fonction était celle d'inclusion et non celle d'égalité, j'ai dû rajouter une condition pour qu'elle soit correcte.

Partie 4 : Preuves des algorithmes utilisés

Question 2 :

Les transformations de la figure 2 convertissent un arbre 234 en arbre bicolore, parce que cet arbre est toujours un arbre de recherche mais il est également impossible d'avoir deux rouges de suite car tous les cas de transformation commencent par un nœud noir. De plus comme les arbres 234 ont obligatoirement toutes les feuilles sur un même niveau cela veut forcément dire que la hauteur noire des arbres bicolores est respectée.

les cas de transformation de la figure 3 et 4 convertissent bien un arbre bicolore en un arbre 234, parce que cet arbre est un arbre de recherche. De plus comme les arbres bicolores ont une même hauteur noire sur tous les sous arbres, ce qui implique que l'arbre 234 obtenu possède la même hauteur sur toutes ses feuilles.

Question 3 :

L'algorithme d'insertion dans un arbre 234, renvoie un arbre qui respecte les propriétés des arbres 234. L'arbre renvoyé a toujours la même hauteur sur toutes ces feuilles parce que on ne crée jamais de nouvelles feuilles, on casse les branches croisées sur le chemin de la descente. Ce qui fait que l'arbre 234 grandit par la racine et donc garde forcément toutes les propriétés des arbres 234. Cet algorithme a une complexité logarithmique, $O(\log_2(n))$

Question 4 :

Algorithme de suppression de valeur dans un arbre
234 :

```
supp_ab234 v, a:
    a = equilibrer_sup_rac(v, a)
    SI a = Vide ALORS
        Vide
    SINON
        SI le sous arbre de a visé n'est pas équilibré
        ALORS
            supp_ab234(v, equilibrer_supp (v, a))
        SINON
            SI v n'est pas dans la racine de a et que le
            sous arbre visé est équilibré ALORS
                supp_ab234(v, sous arbre visé de (a))
            SINON
                SI a est une feuille et que v est dans a ALORS
                    suppression de v dans A
                SINON
                    SI v est dans la racine de a ET le sous arbre
                    est équilibré ALORS
                        descente de v dans le sous arbre a ça gauche
                        immédiate et remonté de la plus grande valeur du sous
                        arbre de a à la place de v. Puis supp_ab234(sous arbre
                        visé de a)
                    FIN SI
```

Cet algorithme, équilibre tous les sous arbres sur le chemin de la cible, puis supprime un élément de l'arbre uniquement lorsque l'élément est dans une feuille qui est obligatoirement un Noeud2 ou Noeud3 donc ne supprime jamais de feuille ce qui permet de maintenir une hauteur égale sur toutes les feuilles. La complexité de cet algorithme est $O(\log_2(n)*2)$

Question 5 :

Pour l'union : tous les éléments du premier arbre sont ajoutés dans le deuxième pour obtenir l'union des deux.

```
309 let union a1 a2 =  
310   add_list (ab234_to_list a1) a2;;
```

Pour l'intersection : seuls les éléments présents dans les deux arbres sont ajoutés dans un troisième arbre pour obtenir l'intersection des deux.

```
312 (* fonction qui fait l'intersection de deux arbres *)  
313 let intersection a1 a2 =  
314   let rec aux acc a =  
315     match a with  
316     | [] -> acc  
317     | v::a when (est_dans v a2) = true -> aux (inserer v acc) a  
318     | v::a -> aux acc a  
319   in aux Vide (ab234_to_list a1);;
```

Pour la différence : les éléments du deuxième arbre sont supprimés du premier pour obtenir le premier arbre sans les éléments du deuxième arbre.

```
321 (* fonction qui fait la difference de deux arbres a1\ a2 *)  
322 let dif a1 a2 =  
323   supp_list (ab234_to_list a2) a1;;  
324
```

Pour la différence symétrique : on utilise la fonction Union et intersection pour en faire la différence.

```
325 (* fonction qui fait la difference symetrique de deux arbr  
326 let delta a1 a2 =  
327   dif (union a1 a2) (intersection a1 a2);;  
328
```

Pour le test d'égalité : tous les éléments du premier arbre sont comparés au deuxième et si l'un des éléments n'est pas dans l'autre alors on renvoie faux sinon suppression de cet élément, si le deuxième arbre est vide alors vrai.

```
329 (* fonction qui teste l'egalité de deux arbres *)
330 let egalite a1 a2 =
331   let rec aux a ab =
332     match a with
333     | [] when (ab = Vide) -> true
334     | v::a when (est_dans v a2) = true -> aux a (supp_ab234 v ab)
335     | _ -> false
336   in aux (ab234_to_list a1) a2;;
337
```

Pour le test d'inclusion : tous les éléments du deuxième arbre sont comparés au premier et si l'un des éléments n'est pas dans l'autre alors faux sinon si tous les éléments sont dans le premier arbre vrai.

```
338 (* fonction qui teste l'inclusion d'un arbres dans un autre*)
339 let inclusion a1 a2 =
340   let rec aux a =
341     match a with
342     | [] -> true
343     | v::a when (est_dans v a2) = true -> aux a
344     | _ -> false
345   in aux (ab234_to_list a1);;
346
```