Rapport de projet

Compression d'arbres binaires

ROZOVYK Artemiy,
BARDOUX Claire,
HERNOUF Mohamed

Programmation fonctionnelle en Ocaml.
UE d'Ouverture.

Master 1 Sciences et Technologies du Logiciel

27 Novembre 2019

Chapitre 1

Présentation

1.1 Introduction

Dans ce rapport, on va décrire la démarche suivie en vue de réaliser le devoir de programmation sur la compression d'arbres binaires. Dans un premier temps, il s'agit de concevoir des algorithmes permettant la synthèse des arbres binaires de recherche *(des ABR)*, qui sont ensuite compressés suivant une démarche particulière. Dans un deuxième temps on effectue un ensemble de tester sur la complexité en temps d'exécution ainsi qu'en espace prit par ces structures. Finalement on analyse les résultats obtenus. Le projet est réalisé suivant le paradigme fonctionnel en language OCaml.

1.2 Arbres binaires de recherche

1.2.1 Syntèse de données

Tout d'abord, afin de pouvoir générer des arbres ayant une structure homogène, on a mis en place un générateur de listes, qui produit des suites d'entiers ordonnés aléatoirement étant donnée la taille de la liste souhaitée.

1.2.2 Constrtuction de l'ABR

On définit l'ABR de manière suivante :

```
type 'a binary_tree =
    | Empty
    | Node of 'a * 'a binary_tree * 'a binary_tree;;
```

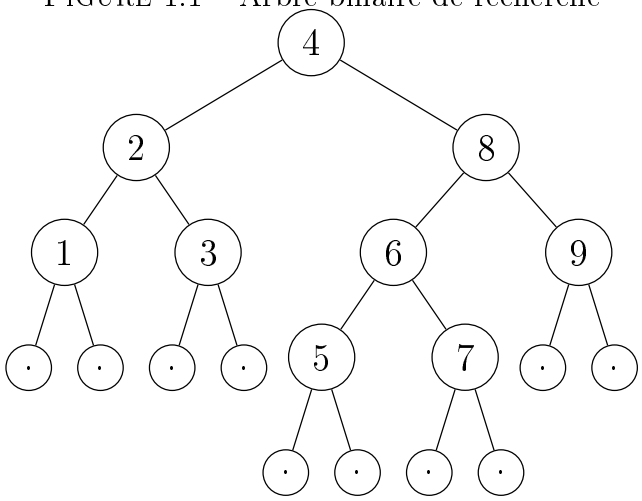
Ainsi l'insertion se fait via un parcour de la structure *binary_ tree*, guidé par la valeur inserée :

```
    match tree with
      |Empty -> Node (a,Empty,Empty)
      |Node(r,left,right) ->
          if (a<r) then Node (r,insert_a left, right)
          else Node (r,left,insert_a right);;
```

La recherche, l'insertion et la suppression d'un élément dans un tel arbre se fait en $\mathcal{O}(\log n)$ en moyenne.
L'exemple de l'énoncé est un arbre qui correspond à la liste [1] :

```
    [4; 2; 8; 1; 3; 6; 9; 5; 7]
```

FIGURE 1.1 – Arbre binaire de recherche



1. Cette liste n'est pas unique.

Chapitre 2

L'arbre compressé

Dans cette partie on va décrire l'algorithme de compression d'ABR suivant le procédé décrit dans l'énoncé. Tout d'abord, on définit une nouvelle structure d'arbres compressés :

```
type valeurABRC_listes = (int* int list) list;;
```

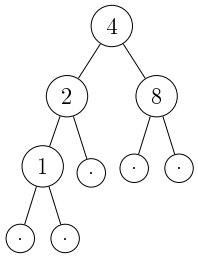
```
type abrc_listes =
  | EmptyABRC
  | NodeABRC of valeurABRC_listes *
    (abrc_listes ref * int) * (abrc_listes ref * int);;
```

2.1 Carcasse

Une fois l'ARB obtenu, on cherche à obtenir une première structure servant de base pour les modifications qui vont suivre. Cela se fait en faisant un parcours suffixe et en sauvegardant les sous-arbres ayant la structure dont on rencontre pour la première fois. Autrement dit, on construit une association avec la structure parenthésée comme la clé[1] et l'entier se trouvant dans le noeud comme la valeur :

1. Ne pas confondre avec la clé dans un noeud. Ici, il s'agit d'une clé dans une map

FIGURE 2.1 – Arbre "squelette"




```
[("((())())((())())()", 4);
("((()))()", 2);
("()", 1);
("(((())())())()", 8) ]
```

Ensuite on construit la structure initiale de manière similaire que dans un ABR normal [2].
Cela nous donne la structure suivante : voir la figure 2.1

2.2 La compression

Dans un deuxième temps on cherche à trouver des valeurs qui ont la même structure que ceux qui sont présents dans notre arbre-carcasse, en se souvenant du noeud qui est notre "jumeau". Ainsi, on obtient la liste de `(clé,valeur)` suivante :

```
[(3, 1); (6, 2); (5, 1); (7, 1); (9, 1)]
```

Ensuite on cherche encore une fois à insérer les clés dans l'arbre-carcasse en descendant suivant leur valeur*(on commence par le couple (3,1) en regardant 3)*, mais cette fois-ci on insère dès qu'on rencontre un fils `EmptyABRC`. Dans le cas de (3,1) on va faire descendre 2 jusqu'à rencontrer le fils droit de (2) qui est vide. À ce moment-là, on met à jour la référence `abrc_listes ref` droite de (2) qui pointera désormais vers le noeud identifié par la valeur droite*(snd)* du couple `(3,1)`. On insère la valeur 3 dans le noeud (1)

2. *c.f.* 1.2.2

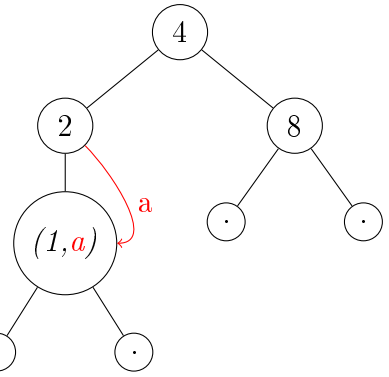
Finalement, on doit étiqueter l'arête ajoutée grâce à un compteur global :

```
let etiq = ref 0;;
let gen_etiq () = etiq:=!etiq+1; !etiq;;
let relancer_gen() = etiq:=0;;
```

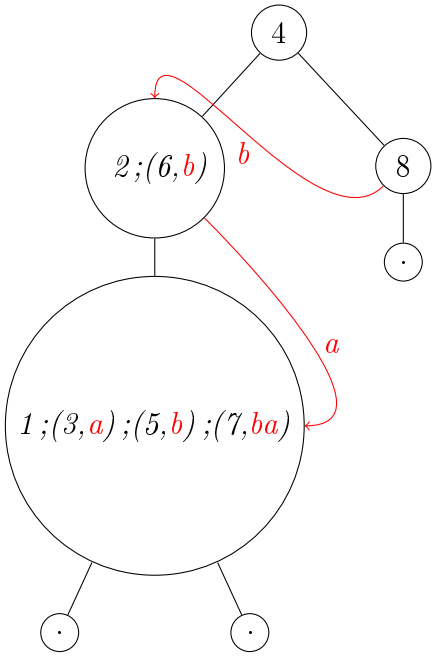
On étiquète également la valeur inserée suivant

1. L'étiquette qu'on vient d'ajouter à une arête *(Ex.1)*
2. Ainsi que l'ensemble d'étiquettes rencontrées sur les arêtes rouges lors de la descente dans l'arbre. *(Ex.2)*

Ex.1)



Ex.2)



2.2.1 Temps de recherche

La recherche d'une valeur dans notre arbre compressé est au pire en $\mathcal{O}(n)$ car même si on descend dans une seule direction (ligne droite) ce qui se fait en $\mathcal{O}(n)$, la somme de noeuds parcourus et les clés de la liste interne du noeud ne peut pas dépasser n. Le temps de recherche en moyenne est dégradé par rapport aux ABR *(qui, en pratique, ont un temps de recherche en* $\mathcal{O}\log(n)$ à cause des parcours fréquents de la liste interne aux noeuds.

2.2.2 Amelioration via map

Afin de réduire le temps de recherche en pratique, on remplace la structure interne des noeuds par une Map, qui est implémentée avec des arbres rouge-noir i. e. ont un temps de recherche en $\mathcal{O}\log(n)$:

```
module ListMap = Map.Make(OrderedList);;
type valeurABRC_maps = int ListMap.t;;
type abrc_maps =
  | EmptyABRC
  | NodeABRC of valeurABRC_maps *
        (abrc_maps ref * int) * (abrc_maps ref * int);;
```

Ainsi la complexité au pire cas est toujours en $\mathcal{O}(n)$ e.g. avec un arbre construit à partir de la liste `[5;4;3;2;1]` on va parcourir n valeurs. Mais en pratique le temps de recherche est en $\mathcal{O}\log(n)$ car on parcourt $\log(n)$ noeuds en moyenne et la recherche à l'intérieur du noeud est également en $\log(n)$. Autrement dit, on cherche une valeur parmi n en parcourant un certain nombre*(faible)* de fois des structures ayant un temps de recherche logarithmique.

Chapitre 3

Experimentations

3.1 Etude de la complexite

Afin de mesurer le temps pris par l'exécution d'une fonction *f*, on a mesuré le temps du système *t1* juste avant l'exécution de *f*, ainsi que temps *t2* au moment juste après. Le résultat est la différence *t2* - *t1* :

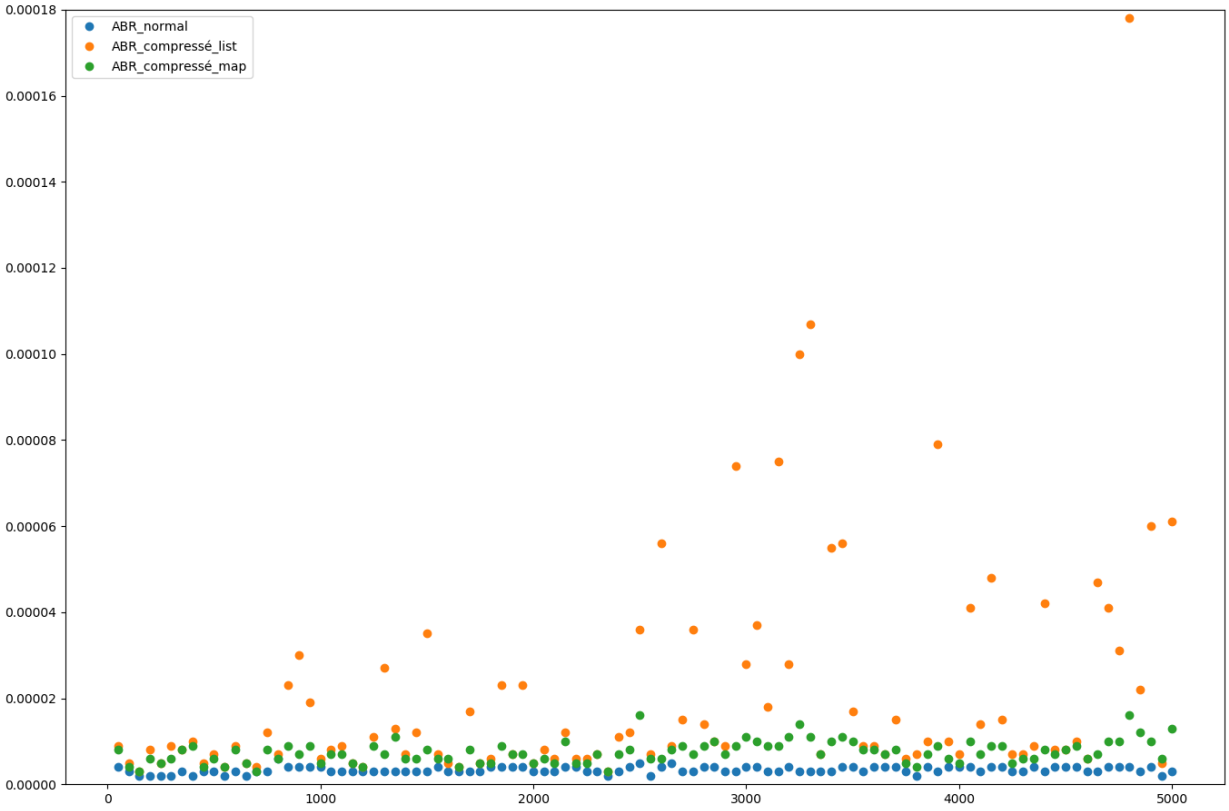
```
let time f =
  let t = Sys.time() in
  let res = f () in
    (Sys.time() -. t, res) ;;
```

Le principe de mesure d'espace pris par une fonction est similaire : on mesure le nombre d'octets vivants alloués par le programme *(on netoie d'abord en faisant un parcours complet de garbage collector)* à un moment donné, et on retourne la différence d'espace alloué juste après l'exécution de f.

```
let space f =
  let t = Gc.compact(); Gc.allocated_bytes() in
  let res = f () in
  let z =  Gc.compact(); Gc.allocated_bytes() -. t in
    (z,res);;
```

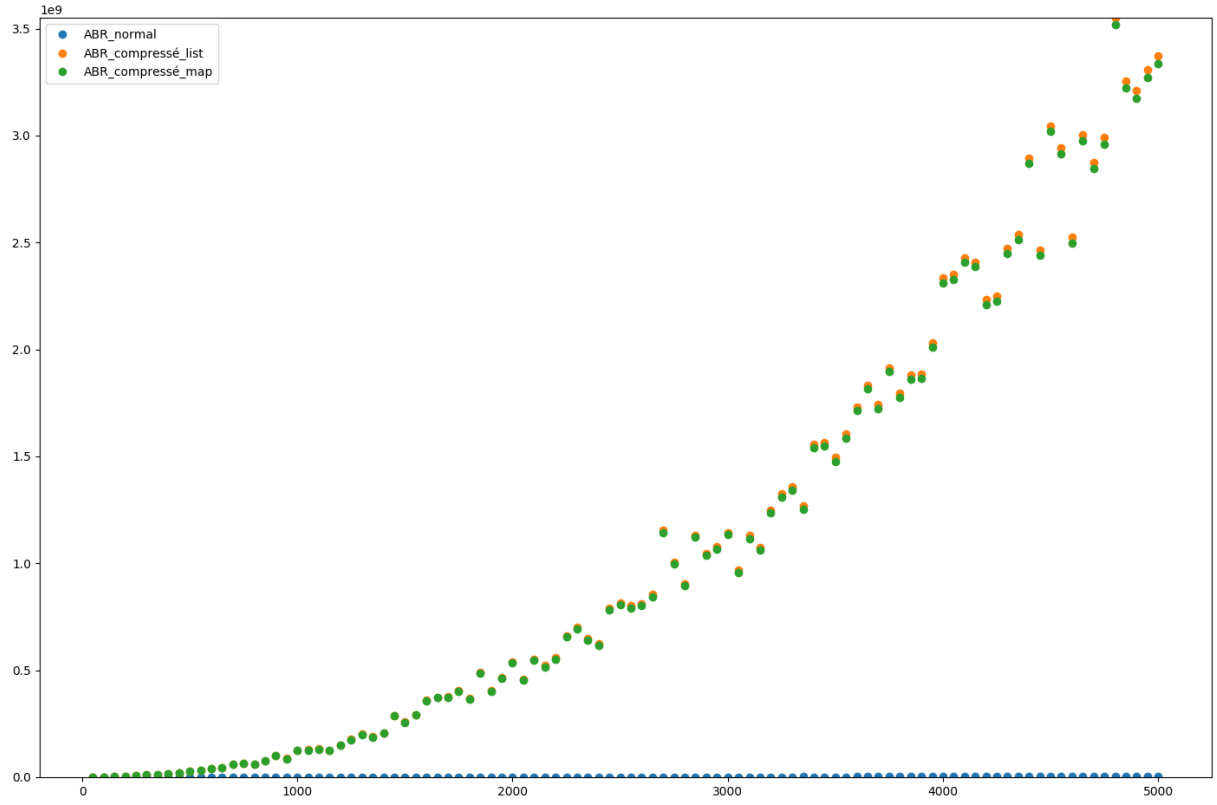
3.2 Résultats

3.2.1 Etude de temps de recherche



On voit une nette amélioration en temps de recherche entre ABR implémentés avec des listes et ABR utilisant une Map.

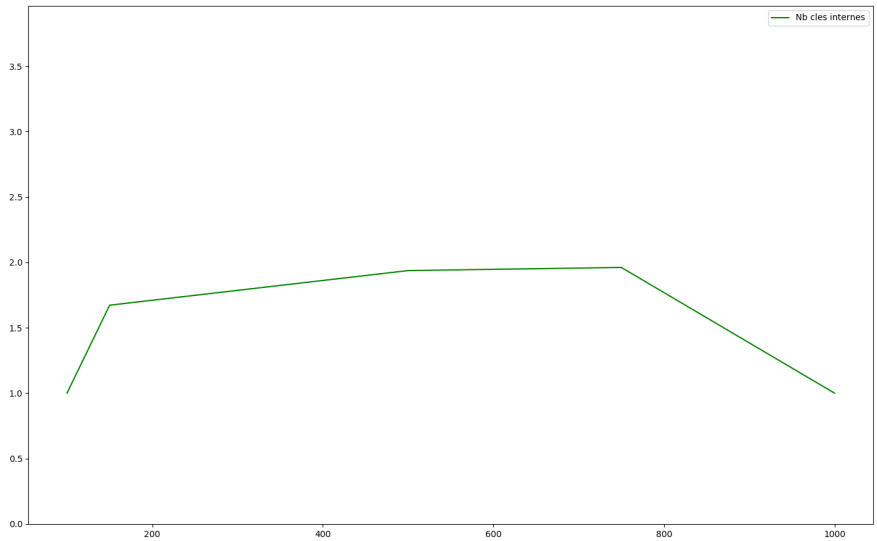
3.2.2 Etude en espace occupé en mémoire

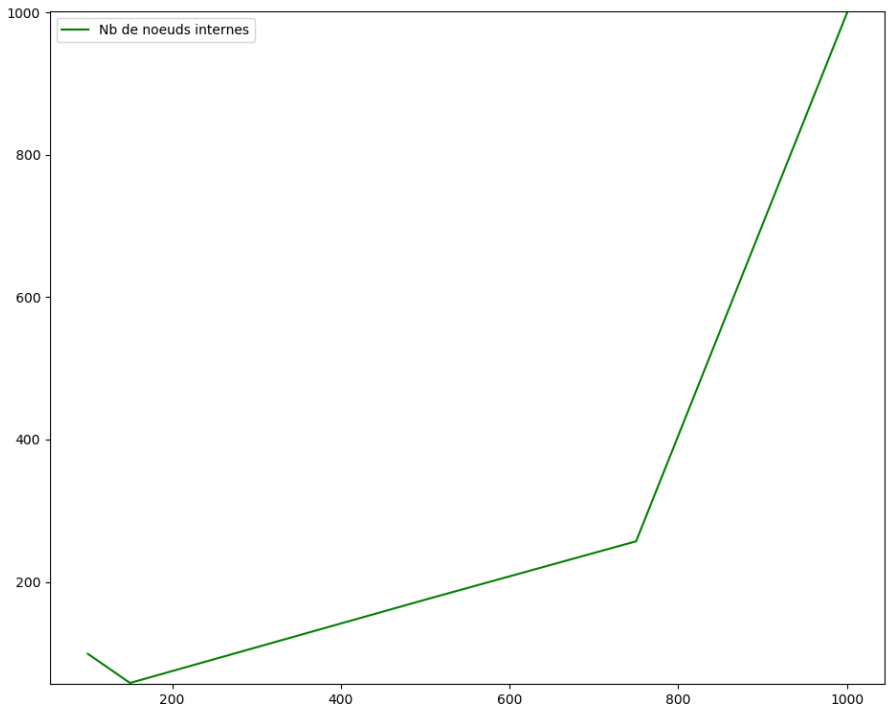


Comme on peut observer sur le graphe, on n'a pas obtenu la compression par rapport à l'ABR normal. Ceci est dû au nombre important d'étiquettes [1] qu'on stocke dans nos arêtes/noeuds ainsi qu'à la manière dont elles sont gérées en mémoire en Ocaml.

1. Même les étiquettes vides sont stockées

3.2.3 Question 3.1





Chapitre 4

Conclusion

Nous avons réussi d'implementer la structure demandée sans pour autant atteindre la compression souhaitée. Néanmoins, on a pu observer et comparer la complexité en temps de recherche de ces 3 structures ce qui nous a permis de nous familiariser encore plus avec les structures arborescentes, voir l'importance de choix d'implementation d'une structure particulière, ainsi que de nous améliorer dans la programmation fonctionnelle en OCaml.