UE Ouverture : Devoir de Programmation

Edwin Ansari
Maria Sidko

1 Introduction

Les arbres binaires de recherche sont la structure de données parmi les plus anciennes, dont le concept remonte à bien avant la naissance de l'informatique, et restant tout de même fréquemment utilisée de nos jours. Avec la nécessite toujours croissante de stocker de plus en plus de donnés, il est intéressant de se pencher sur une manière de diminuer l'espace de stockage utilisé par ces arbres, tout en gardant leur efficacité de recherche. Au cours de ce projet, nous nous penchons sur une manière, parmi d'autres, de compresser ceux-ci. Afin d'implémenter nos recherches, nous utilisons le langage `OCaml`, qui, bien que ayant un nom rappelant un mammifère camélidé d'origine asiatique, fut crée notamment par des français, et s'adapte parfaitement à notre exercice, de par sa rapidité et ses caractéristiques fonctionnelles, permettant néanmoins des éléments issus de l'impératif.

2 Présentation

Synthèse de données Les fonctions de génération de nombre ou permutation aléatoire sont implémentées dans le fichier `RandomGenerator.ml` avec une interface `RandomGenerator.mli` qui nous donne acces aux trois fonctions `range`, `gen_permutation` et `gen_permutation2`

Question 1.1 `extraction_alea l p`
Génère un nombre aléatoire r entre 0 et $taille(l)$ puis créé une nouvelle liste l' en utilisant la fonction `remove_nth` sans l'élément r-ième de l et une liste p' qui a comme head l'élément r-ieme de l et comme queue p et à la fin renvoie (l', p').

Question 1.2 `gen_permutation n`
Est une fonction récursive terminale qui utilise une fonction aux qui appelle `extraction_alea` $n = taille(l)$ fois sur l. On appelle cette fonction avec comme paramètre `range 1 n`, qui est une fonction qui génère les nombre entre 1 et n.

Question 1.3 La fonction `aux` est appelée n fois et à chaque foisellel appelle 1 fois la fonction `extraction_alea` qui appelle elle même 1 fois la fonction `Random.int` et la fonction `remove_nth`, or la fonction précédente fait au plus n pattern matching en fonction de la taille de la liste. on constate donc une complexité en $O(n)$ en nombre d'appels à `Random.int` et une complexité en $O(n^2)$ en nombre de match.

Question 1.4 `Intercale l1 l2`
Est aussi une fonction récursive terminale qui utilise une fonction `aux` avec accumulateur qui à chaque fois génère un nombre aléatoire r entre 0 et $n1 + n2$ si $n1 = taille(l1)$ et $n2 = taille(l2)$ et prend la tête de la liste $l1$ si $n1 < r$, et sinon la tête de la liste $l2$, donc on retrouve les probabilités annoncées pour la tête de la nouvelle liste.

Question 1.5 `gen_permutation2 p q`
Est un algorithme diviser pour régner qui appelle récursivement `intercale` sur les deux moitiés $(p - milieu)$ et $(milieu - q)$ avec $millieu = (p + q)/2$.

Question 1.6 Soit T(n) le temps d'exécution de l'algorithme avec $n = p + q$ on a la relation suivante : $T(n) = T(n/2) + T(n/2) + O(n)$ donc $T(n) = 2T(n/2) + O(n)$ en appliquant le "Master Theorem". On constate une complexité en $O(n \log n)$ en nombre d'appels à `Random.int` et une complexité en $O(n \log n)$ en nombre de match.

3 Construction de l'ABR

Les fonctions de cette partie sont dans le fichier `Constructor.ml`. On commence par définir le type algébrique d'un arbre binaire comme suite :

```
type binary_tree =
    | Empty
    | Node of binary_tree * int * binary_tree;;
```

Question 1.7 La fonction `insert t x` insère x dans l'arbre binaire t récursivement : si l'arbre est vide on crée un nouveau noeud avec fils gauche et fils droit type `Empty` et étiquetté x, sinon, si x est plus petit que l'étiquette de t on insère x dans le sous-arbre gauche, sinon dans le sous-arbre droit.
la fonction `search t x` recherche l'élément x dans l'arbre binaire t récursivement en guidant la recherche par l'étiquette comme vu précédemment.
la fonction `construct l` prend en entrée une liste d'entiers et appelle la fonction `insert` sur tout les éléments de la liste en appliquant d'abord a l'arbre vide.

4 Compression des ABR

Les fonctions de cette partie sont dans le fichier `Compressor.ml` avec une interface adapte `Compressor.mli`. On commence par définir le type algébrique d'un arbre binaire compressé comme suite :

```
type c_tree =
    | Empty
    | C_Node of c_tree * int array
    | Node of c_tree * int * c_tree * int ;;
```

`Empty` quand l'arbre est vide.
`Node` avec sous-arbre gauche , étiquette , sous-arbre droit , $taille(gauche)$ pour un noeud qui n'ai pas compressé et qui nous sert comme structure isomorphe à d'autres arbres.
`C_Node` avec une référence vers un autre arbre (toujours un `Node`) et un tableau qui est le parcours préfixe de l'arbre.

Question 2.8 La fonction `phi` est une fonction invective qui nous servira de retrouver la forme d'un arbre dans une liste `patterns` dans la fonction `Compress`.

Question 2.9 La fonction `prefixe (t : binary_tree) : int list`
Fait le parcours préfixe récursivement en ajoutant d'abord l'étiquette ensuite le préfixe du sous-arbre gauche et ensuite le préfixe du sous-arbre droite dans la liste.

Question 2.10 `Compress (t:binary_tree) : c_tree`
Est la fonction principale qui effectue la compression. La fonction commence par initialiser une liste de pattern, puis appelle la fonction `aux` sur `t` récursivement afin de créer à la fin un arbre compressé de type `c_tree`
si l'arbre est vide on renvoie vide.
sinon on regarde si un noeud de cette forme la existe déjà dans les patterns, et si c'est le cas on crée un `C_Node` avec comme référence l'arbre trouvé et avec comme tableau le parcours préfixe de l'arbre.
sinon si la structure n'est pas présente dans les patterns on essaie de retrouver la structure de son sous-arbre gauche/droite dans les patterns et procéder comme vu précédemment, enfin on ajoute le nouveaux noeud crée de cette manière dans notre liste de pattern en calculant le ϕ.

Question 2.11 `search (t:binary_tree) (x:int) : bool`
Appelle une fonction `aux` qui prend comme paramètres l'arbre `t` et l'élément `x` à rechercher, mais aussi l'indice `i` du tableau préfixe d'un premier `C_Node` rencontré et son tableau `arr` correspondant ensuite si on rencontre un noeud de type `Node` on regarde si il faut vraiment regarder les étiquettes de ce noeud quand `arr` est vide, sinon on guide notre recherche avec les éléments dans `arr` de la manière suivante : on avance à chaque fois de 1 case dans le tableau si l'étiquette est plus petite et de $(taille(gauche) + 1)$ case si l'étiquette et plus grande. sinon si on rencontre un `C_Node` pour une première fois à partir de maintenant notre paramètre `arr` va être toujours égale au tableau préfixe de ce dernier et ensuite en appelle récursivement la fonction `aux` avec la référence vers un `Node` et le tableau `arr` en question.
La complexité reste à peu près identique à la recherche dans un arbre binaire normal car à chaque fois si on rencontre un `Node` on parcour soit le sous-arbre droite soit le sous-arbre gauche et si on rencontre un `C_Node` on fait la même chose sur l'arbre pointé.
Dans le pire des cas pour un arbre linéaire (non équilibré) on constate une complexité en $O(n)$.

Question 2.12 On constate une complexité moyenne en $O(\log n)$.

5 Expérimentations : gains ou perte d'efficacité des ABR compressés

Question 3.13 Pour calculer le temps nous avons utilisé la différence entre deux appels à la fonction `Unix.gettimeofday()`, que nous appelons avant et après l'exécution de la recherche.

Question 3.14 Nous avons fait une somme des temps mesurés sur 1000 recherches différentes pour chaque graphe de taille n sur des arbres générés aléatoirement et avec un entier à rechercher généré aléatoirement, et nous obtenons le graphique de la Figure1. On constate des temps très petits dans les deux cas, et à priori la compression ne ralentit pas beaucoup la recherche dans l'ABR. L'observation d'une réelle différence nécessiterait une expérimentation sur des arbres beaucoup plus grands ce que la puissance de la machine utilisée ne permet pas.

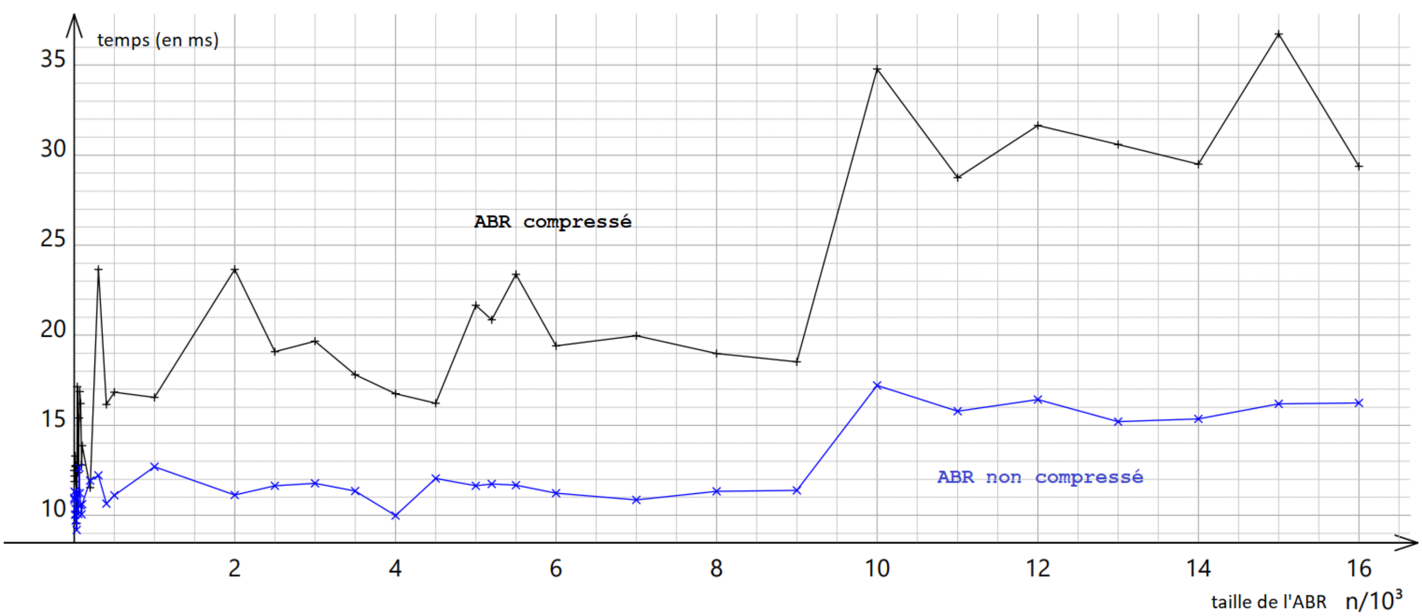

Figure 1 – Graphique des temps d'exécution de la fonction de recherche en fonction de la taille n de l'ABR

Question 3.15 En effectuant des appels successifs à la fonction `sizeof` proposée pour des ABR compressés et non compressés à taille croissante générés aléatoirement nous obtenons les espaces mémoire utilisés par les arbres. On constate que sur des valeurs de n petites les arbres non compressés prennent en moyenne plus d'espace que ceux non compressés (Figure 2). Cependant, pour des valeurs plus conséquentes (Figure 3) on constate une amélioration sur la taille en mémoire du graphe.

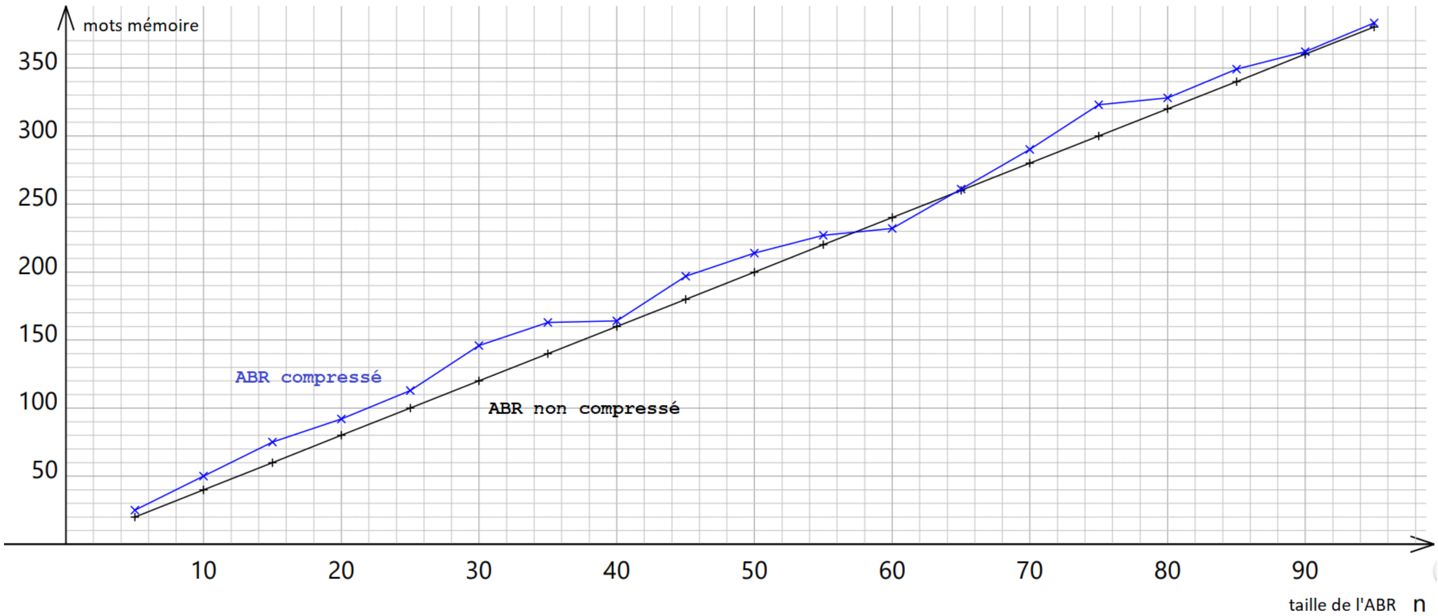

FIGURE 2 – Graphique du nombre de mots mémoire stockés sur le tas en fonction de la taille n de l'ABR

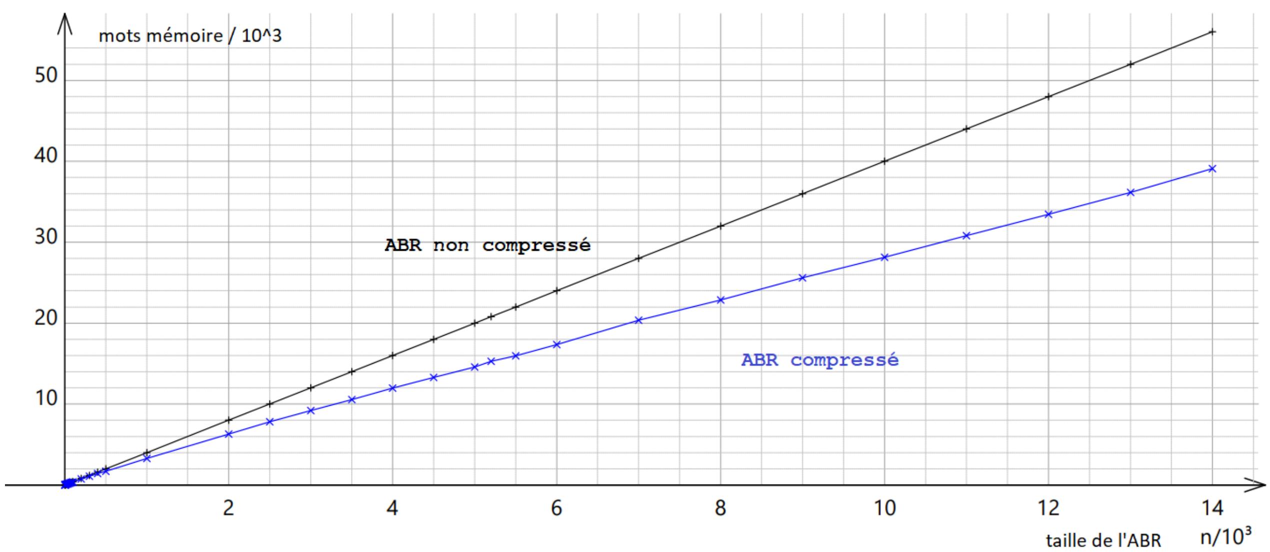

FIGURE 3 – Graphique du nombre de mots mémoire stockés sur le tas en fonction de la taille n de l'ABR

6 Conclusion

En somme, on constate une nette amélioration lors de l'implémentation de la compression, qui impacte peu le temps de la recherche. Cependant, on est forcés de constater que l'action de compression peut impacter la complexité temporelle lors de la création de l'arbre.