UE Ouverture **Devoir de Programmation**

---

*Devoir à faire en **binôme**. Les soutenance auront lieu les 8 et 15 décembre, l'ordre de passage et les détails organisationnels vous seront donnés le 1er décembre en fonction des nouvelles consignes sanitaires. Un court rapport et le code source sont à rendre par mail à vos enseignants avec pour objet `"[OUV] noms et prénoms"` au plus tard le 7 décembre.*
Langage de programmation imposé : OCaml.

1 Présentation

Le but du problème consiste à manipuler un modèle de structure de données arborescente, les Arbres de Binaires de Recherche (ABR) puis d'en construire une structure compressée suivant une procédure bien particulière. L'expérimentation consiste à calculer les taux de compression obtenus, tout en vérifiant que l'efficacité de recherche dans la structure compressée n'est pas trop dégradée.
Il est attendu un soin particulier concernant la programmation, dans le paradigme fonctionnel, et par rapport à la réflexion et la mise en place des expérimentations.

1.1 Synthèse de données

La première partie du devoir consiste à l'écriture d'algorithmes permettant la synthèse d'ABR, arbres que nous utiliserons par la suite pour expérimenter notre algorithmique de compression. Par souci de simplification, les données que nous insérons dans l'ABR sont les entiers de 1 à n.

Question 1.1 Implémenter une fonction `extraction_alea` prenant en entrée deux listes d'entiers, notée L et P. Appelons ℓ le nombre de valeurs que contient L. La fonction choisit aléatoirement un entier r entre 1 et ℓ, puis retourne un couple de listes dont la première est la liste L dans laquelle on a retiré le r-ième élément et la deuxième est la liste P dans laquelle on a ajouté en tête le r-ième élément extrait de L, (celui qui vient d'être retiré de L).
On pourra utiliser la fonction `Random.int: int -> int` de la bibliothèque standard qui prend un entier strictement positif n en argument et renvoie un entier tiré uniformément au hasard dans l'intervalle $[\![0; n-1]\!]$.

Question 1.2 Implémenter une fonction `gen_permutation` prenant en entrée une valeur entière n. Cette fonction génère la liste L des entiers de 1 à n (L est triée) et une liste vide P puis vide entièrement la liste L et remplit la liste P en appelant `extraction_alea`.

L'algorithme que nous venons d'implémenter est *l'algorithme de shuffle de Fisher-Yates.*

Question 1.3 Quelle est la complexité de cet algorithme en nombre d'appels au générateur de nombres aléatoires (en fonction de n) ? Et en nombre de filtrages de motif (match) ?

1.2 Synthèse de données améliorée

(Il n'est pas nécessaire de faire cette partie pour passer à la suite, vous pouvez passer directement à la partie 1.3 *et revenir sur cette partie à la fin du projet si vous le souhaitez.)*

Dans cette partie on implémente un algorithme plus efficace que l'algorithme précédent pour générer des permutations aléatoires uniformes en utilisant le principe "diviser pour régner". Nous allons utiliser l'idée suivante : un tableau contenant les nombres de 1 à n peut être obtenu en *intercalant* un tableau contenant les nombres de 1 à $\lfloor n/2 \rfloor$ dans un tableau contenant les nombres de $(\lfloor n/2 \rfloor + 1)$ à n (voir Figure 1).

Question 1.4 Écrire une fonction `intercale` qui prend en argument deux listes L_1 et L_2 et qui

renvoie une nouvelle liste obtenue en intercalant L_1 dans L_2 (cf. Figure 1). La fonction `intercale` devra satisfaire la propriété suivante. Soient L_1 et L_2 deux listes d'entiers et soient n_1 et n_2 leurs tailles, alors :

— avec probabilité $\frac{n_1}{n_1+n_2}$, la tête de la liste `intercale`(L_1, L_2) est la tête de L_1 et sa queue est obtenue en intercalant la queue de L_1 dans L_2 ;

— avec probabilité $\frac{n_2}{n_1+n_2}$ c'est l'inverse : la tête de la liste `intercale`(L_1, L_2) est la tête de L_2 et sa queue est obtenue en intercalant L_1 dans la queue de L_2.

Remarque : on fera attention à ce que la fonction ne recalcule pas la longueur des deux listes à chaque appel récursif, il pourra être utile pour cela de définir une fonction auxiliaire qui prend en argument la taille des listes en plus.

Question 1.5 Écrire une fonction `gen_permutation2` qui prend en argument deux entiers p et q et calcule une permutation aléatoire des entiers de p à q (inclus) de la façon suivante :

— si $p > q$, renvoyer la liste vide ;

— si $p = q$, renvoyer la liste contenant uniquement p ;

— sinon générer une permutation des entiers de p à $\lfloor \frac{p+q}{2} \rfloor$, une permutation des entiers de $\lfloor \frac{p+q}{2} \rfloor + 1$ à q et intercaler les deux listes.

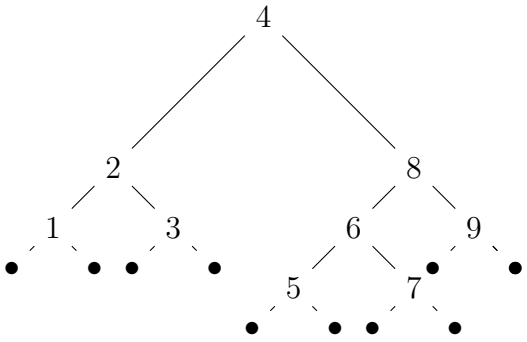
On obtient une permutation aléatoire des nombres de 1 à n en appelant `gen_permutation2` 1 n.

Question 1.6 Quelle est la complexité de cet algorithme en nombre d'appels au générateur de nombres aléatoires (en fonction de n) ? Et en nombre de filtrages de motif (match) ?

1.3 Construction de l'ABR

Pour rappel, un arbre binaire est : (1) soit réduit à une feuille, (2) soit décomposable en une racine qui est un nœud interne et qui pointe vers deux enfants ordonnés, l'enfant gauche et l'enfant droit. On peut dès lors définir un ABR. C'est un arbre binaire dont les nœuds internes sont étiquetés (par des entiers distincts) de telle sorte que la racine possède une étiquette plus grande que toutes les étiquettes de l'enfant gauche, et plus petite que toutes les étiquettes de l'enfant droit. Récursivement, les deux enfants de la racine sont des ABR.

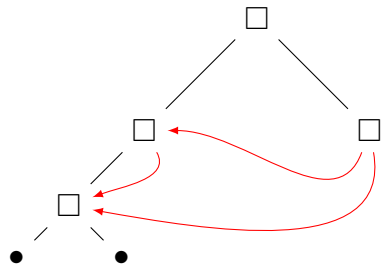
Question 1.7 Étant donnée une liste d'entiers tous distincts, construire l'ABR associé à cette liste. Pour rappel, on insère toujours dans une feuille. Ainsi, si la liste est vide, on renvoie l'arbre réduit à une feuille. Sinon, pour insérer l'entier ℓ en tête de liste, on parcourt la branche partant de la racine de l'arbre actuel allant à une feuille, en utilisant les étiquettes des nœuds internes pour progresser. Par exemple, en un nœud étiqueté par k, si $\ell < k$ on insérera ℓ dans l'enfant gauche de k, et sinon ce sera dans l'enfant droit. Par exemple, l'arbre représenté à droite est construit à partir de la liste $[4, 2, 3, 8, 1, 9, 6, 7, 5]$.



Remarque : plusieurs listes différentes peuvent donner le même ABR.

2 Compression des ABR

Le but de cette section est de présenter une structures afin de représenter l'ABR de manière plus compacte en mémoire. On ne peut rien gagner au niveau des étiquettes des nœuds, il faut les stocker, donc le seul gain possible intervient au niveau de la structure arborescente. L'idée globale consiste à repérer les sous-arbres (non réduit à une feuille) ayant la même structure arborescente (en ou-



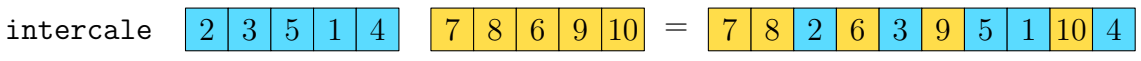


FIGURE 1 – Un tableau de taille 10 obtenu en intercalant deux tableaux de taille 5

bliant l'étiquetage) puis en remplaçant la deuxième occurrence du sous-arbre via un pointeur vers le premier sous-arbre. Par exemple, pour la structure de la question 1.7, on obtiendra l'arbre représenté à droite.

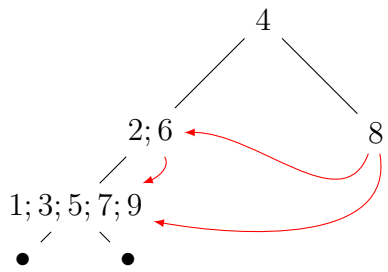
Question 2.8 Afin de reconnaître si deux arbres (ou sous-arbres) non étiquetés sont isomorphes (i.e. identiques), on va associer à chaque arbre une chaîne de caractères construite sur l'alphabet {`(`, `)`}, via la construction suivante. Soit A l'arbre binaire à compresser, et ϕ la fonction suivante :

— Si A est réduit à une feuille, on lui associe le mot vide ε (ainsi $\phi(\bullet) = \varepsilon$) ;

— Si A a un nœud interne et deux enfants G et D qui sont des arbres binaires alors on lui associe : $\phi(A) =$ `(` $\phi(G)$ `)` $\phi(D)$.

On pourrait montrer que ϕ est injective. Ainsi, pour savoir si deux arbres binaires A et B sont isomorphes, il suffit de savoir tester si les mots $\phi(A)$ et $\phi(B)$ sont identiques.
Implémenter la fonction ϕ.
Par exemple, la chaîne de caractères associée à l'arbre de la question 1.7 est : `(((())())(((())())())()`.

Il faut désormais être en mesure de stocker les étiquettes dans la structure compressée, mais il y a des nœuds qui peuvent avoir plusieurs parents. On pourrait y stocker toutes les étiquettes des nœuds qu'ils représentent dans l'arbre non compressé, sur l'exemple on obtient l'arbre ci-contre. Le problème de cette solution est que la recherche en est très pénalisée, voire impossible dans certains cas en raison de l'ensemble de valeurs stocké en chaque nœud.



L'idée mise en œuvre consiste, lorsqu'on remplace un sous-arbre par un pointeur, à stocker les étiquettes contenues dans ce sous-arbre sous forme d'un tableau avec le pointeur. Les étiquettes doivent être insérées dans le tableau dans l'ordre *préfixe*. Nous aurons aussi besoin de stocker la taille des sous-arbres pour la recherche d'élément. Par exemple dans le cas de l'arbre de la question 1.7 cela donne :

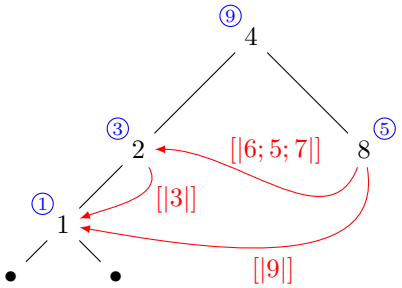


FIGURE 2 – Compression de l'ABR-exemple. En rouge, les pointeurs annotés avec les étiquettes du sous-arbre rangées en ordre préfixe. En bleu la taille des sous-arbres.

Question 2.9 Écrire une fonction `prefixe` prenant en argument un ABR et calculant un tableau contenant les étiquettes de l'arbre rangées en ordre préfixe.

Question 2.10 Implémenter la compression associée à la Figure 2 pour les ABR.

Question 2.11 Implémenter une fonction de recherche de valeur dans un ABR compressé.
La différence avec la recherche dans les ABR classiques est que lorsqu'on franchit un pointeur (en rouge sur le dessin) pour la première fois, on considère le tableau qui y est attaché et on initialise un indice $i = 0$. Puis dans la suite de la recherche on lit les étiquettes des nœuds dans ce tableau plutôt que sur les nœuds, en mettant à jour i.
Par exemple, si l'on cherche 7 dans l'ABR compressé de la Figure 2 :

— 7 est plus grand que la racine 4, on descend dans le sous-arbre droit vers 8 ;

— puis 7 est inférieur à 8 on suit le pointeur gauche et on considère le tableau $t = [|6; 5; 4|]$ et l'entier $i = 0$;

— on arrive sur le nœud 2 qui correspond en fait au nœud $t.(0) = 6$;

— comme $6 < 7$ on doit suivre le fils droit de 2 et on ajoute 2 à i, la valeur 2 correspond à la taille du fils gauche (1) plus 1 pour la racine ;
— on arrive sur le fils droit du nœud 2 qui correspond en fait au nœud $t.(2) = 7$, on a terminé.

Donner la complexité au pire cas de la recherche d'une valeur dans une structure contenant n valeurs.

Question 2.12 (facultative) Quelle est la complexité en moyenne de la recherche dans un ABR compressé ?

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

Question 3.13 Trouver une fonction permettant de calculer le temps pris par l'exécution d'un algorithme sur une donnée particulière. Pour calculer l'espace mémoire occupé par une structure, nous vous proposons la fonction suivante :

```
let sizeof (x: 'a) : int = Obj.reachable_words (Obj.repr x)
```

Si `x` est un pointeur, `(sizeof x)` calcule le nombre de mots mémoire stockés sur le tas accessible depuis ce pointeur. Si `x` n'est pas un pointeur, `(sizeof x)` renvoie 0.

Question 3.14 Effectuer une étude expérimentale de complexité en temps de l'algorithme de recherche dans la structure compressé et dans la structure non compressée. Pour ce faire, on combinera les section 1. et 2. du devoir. Agrémenter l'étude de graphiques et de commentaires.

Question 3.15 Effectuer une étude expérimentale de complexité en espace du gain de la compression des ABR. Agrémenter l'étude de graphiques et de commentaires.