UE Ouverture **Devoir de Programmation**

*Devoir à faire en **trinôme**. Soutenance en séance de TD/TME. Les enseignants décideront d'un ordre de passage pour le mardi 19/11 ou pour le 26/11 (ce n'est pas votre choix).*
Rapport et Code Source à rendre (par mail à vos enseignants) au plus tard le 27/11 à 23h59.
Langage de programmation 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 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).

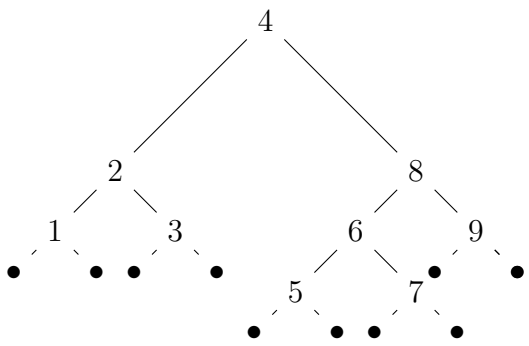
Question 1.2 Implémenter une fonction `gen_permutation` prenant en entrée une valeur entière n. Cette fonction génère une liste L des entiers de 1 à n (L est triée) et une liste vide P puis qui vide entièrement la liste L et remplit la liste P en appelant `extraction_alea`.

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

1.2 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.

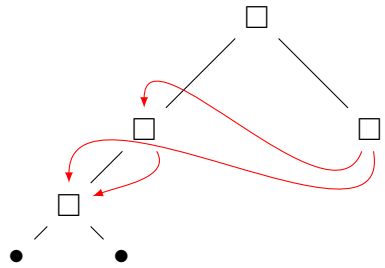
Question 1.3 Étant donnée une liste d'entiers tous distincts, construire l'ABR associé à cette liste. Pour rappel, on insère toujours dans une feuille. Ainsi, si la liste est vide, on renvoie l'arbre réduit à une feuille. Sinon, pour insérer l'entier ℓ en tête de liste, on parcourt la branche partant 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 $[4, 2, 3, 8, 1, 9, 6, 7, 5]$.
Remarques : plusieurs listes différentes peuvent donner le même ABR.

2 Compression des ABR

Le but de cette section est de présenter diverses 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 oubliant 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 1.3, on obtiendra l'arbre représenté à droite.

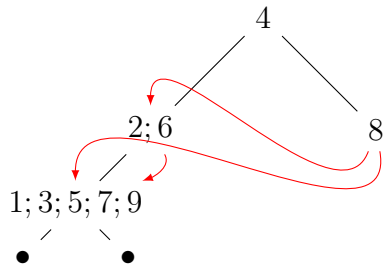


Question 2.4 Afin de reconnaître si deux arbres (ou sous-arbres) non étiquetés sont isomorphes (i.e. identiques), on va associer à chaque arbre une chaîne de caractères construite sur l'alphabet $\{(,)\}$, via la construction suivante. Soit A l'arbre binaire à compresser, et ϕ la fonction suivante :

- 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 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.
Par exemple, la chaîne associée à l'arbre binaire de la question 1.3 est : $((()))((()))()$.

Il faut désormais être en mesure de stocker les informations dans les nœuds internes. Mais, il y a des nœuds qui peuvent avoirs plusieurs parents, et donc qui doivent contenir plusieurs valeurs, sur l'exemple on obtient l'arbre ci-contre.
Le problème de la recherche est très pénalisé, voire impossible dans certains cas en raison de l'ensemble de valeurs stocké en chaque nœud.
L'idée mise en œuvre consiste à ajouter de l'information lorsque l'on passe à travers les arêtes rouges. Par exemple, en associant la clé α à l'arête gauche de l'étiquette 6 et β à l'arête droite, et γ pour l'arête droite de 2, on peut dès lors associer aux valeurs ces clés afin de savoir d'où elles proviennent dans l'arbre initial. Voilà ce que donne l'exemple.



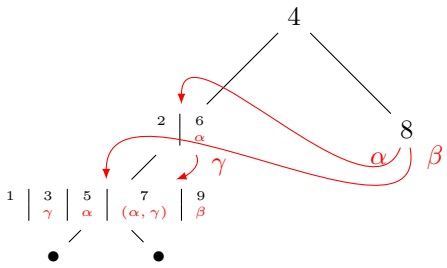


FIGURE 1 – Compression de l'ABR-exemple

Afin d'obtenir une telle compression, on doit avoir un générateur d'identificateur unique pour donner un nom aux arêtes rouges. Cela peut se faire naturellement en donnant un numéro à chaque arête rouge, lors de la compression.

Question 2.5 Implémenter la compression associée à la Figure 1 pour les ABR, en utilisant une liste

pour les couples clé-valeur dans chaque nœud. On appellera cette construction *ABR-compressé-listes.*

Question 2.6 Implémenter une fonction de recherche de valeur dans un ABR-compressé-listes.
Comme on peut s'en douter, la recherche d'un élément risque d'être fortement ralentie.
Donner la complexité au pire cas de le recherche d'une valeur dans une structure contenant n valeurs.

L'idée désormais consiste à stocker de façon plus efficace l'ensemble des clés. Pour ce faire un utilisera une `map` *de OCALM.*

Question 2.7 Implémenter la compression associée à la Figure 1 pour les ABR, en utilisant une `map` pour les couples clé-valeur dans chaque nœud. On appellera cette construction *ABR-compressé-maps.*

Question 2.8 Implémenter une fonction de recherche de valeur dans un ABR-compressé-maps.
Quelle est la structure de données sous-jacente à la structure `map` ? Donner la complexité au pire cas de le recherche d'une valeur dans une structure contenant n valeurs.

Question 2.9 (facultative). Les clés (où k-uplets de clés) des arêtes rouges peuvent odonnés de façon totale. On peut donc remplacer la `map` interne de chaque nœud par un ABR compacté. Implémenter une telle structure et sa fonction de recherche.

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

Question 3.10 Trouver une fonction permettant de calculer le temps pris par d'exécution d'un algorithme sur une donnée particulière.
Trouver une fonction permettant de calculer l'espace mémoire total consommé par une structure de données.

Question 3.11 Effectuer une étude expérimentale de complexité en temps des algorithmes de recherche. Pour ce faire, on combinera les section 1. et 2. du devoir.
Agrémenter l'étude de graphiques et de commentaires.

Question 3.12 Effectuer une étude expérimentale de complexité en espace du gain de la compression des ABR.
Agrémenter l'étude de graphiques et de commentaires.