Concours blanc

OPTION INFORMATIQUE - Devoir n° 5 - Olivier Reynet

Les quatre parties de cet examen sont indépendantes. OCaml est le seul langage nécessaire.

A Sélection du $(k+1)^{\text{e}}$ plus petit élément (CCINP 2023)

La sélection du $(k+1)^{\text{e}}$ plus petit élément d'une liste d'entiers L, non nécessairement triée, consiste à trouver le $(k+1)^{\text{e}}$ élément de la liste obtenue en triant L dans l'ordre croissant.

Par exemple, si $L = [9;1;2;4;7;8]$ le 3^{e} plus petit élément de L est 4 . On pourra remarquer que si la liste L est triée dans l'ordre croissant, le $(k+1)^{\text{e}}$ plus petit élément est l'élément de rang k dans L.

On présente un algorithme permettant de résoudre ce problème de sélection avec une complexité temporelle linéaire dans le pire cas. Celui-ci est basé sur le principe de "diviser pour régner" et sur le choix d'un bon pivot pour partager la liste en deux sous-listes.

Dans cette partie, les fonctions demandées sont à écrire en OCaml et ne doivent faire intervenir aucun trait impératif du langage (références, tableaux ou autres champs mutables ou exception par exemple).

Étant donné un réel a, on note $\lfloor a \rfloor$ le plus grand entier inférieur ou égal à a.

a Fonctions utiles

Dans cette section, on écrit des fonctions auxiliaires qui sont utiles pour la fonction principale.

A1. Écrire une fonction récursive de signature `longueur : 'a list -> int` et telle que `longueur l` est la longueur de la liste `l`.

A2. Écrire une fonction récursive de signature `insertion : 'a list -> 'a -> 'a list` et telle que `insertion l a` est la liste triée dans l'ordre croissant obtenue en ajoutant l'élément a dans la liste croissante `l`.

A3. En déduire une fonction récursive de signature `tri_insertion : 'a list -> 'a list` et telle que `tri_insertion l` est la liste obtenue en triant `l` dans l'ordre croissant.

A4. Écrire une fonction récursive de signature `selection_n : 'a list -> int -> 'a` et telle que `selection_n l n` est l'élément de rang `n` de la liste `l`. Par exemple, `selection_n [4 ; 2 ; 6 ; 4 ; 1 ; 15] 3` est égal à 4 .

A5. Écrire une fonction récursive de signature `paquets_de_cinq : 'a list -> 'a list list` et telle que `paquets_de_cinq l` est une liste de listes obtenue en regroupant les éléments de la liste `l` par paquets de cinq sauf éventuellement le dernier paquet qui est non vide et qui contient au plus cinq éléments. Par exemple :

- `paquets_de_cinq []` est égal à `[]` ,
- `paquets_de_cinq [2 ; 1 ; 2 ; 1 ; 3]` est égal à `[[2 ; 1 ; 2 ; 1 ; 3]]`,
- `paquets_de_cinq [3 ; 4 ; 2 ; 1 ; 5 ; 6 ; 3]` est égal à `[[3 ; 4 ; 2 ; 1 ; 5] ;[6 ; 3]]`.

A6. Écrire une fonction récursive de signature `medians : 'a list list -> 'a list` et telle que `medians l` est la liste `m` obtenue en prenant dans chaque liste l_k apparaissant dans la liste de listes `l` l'élément médian de l_k. On convient que pour une liste A dont les éléments sont exactement $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$, l'élément médian désigne $a_{\lfloor \frac{n}{2} \rfloor}$.
Dans le cas où la liste L n'est pas triée, l'élément médian désigne l'élément médian de la liste obtenue en triant L par ordre croissant. Par exemple : `medians [[3 ; 1 ; 5 ; 3 ; 2] ;[4 ; 3 ; 1] ;[1 ; 3] ;[5 ; 1 ; 2 ; 4]]` est égal à `[3 ; 3 ; 3 ; 4]`.

A7. Écrire une fonction de signature `partage : 'a -> 'a list -> 'a list * 'a list * int * int` telle que `partage p l` est un quadruplet `l1, l2,n1,n2` où `l1` est la liste des éléments de `l` plus petit que `p`, `l2` est la liste des éléments de `l` strictement plus grand que `p`,`n1` et `n2` sont respectivement les longueurs de `l1` et `l2`.

b La fonction de sélection et sa complexité

On détaille la fonction de sélection.

A8. Écrire une fonction récursive de signature : `selection : 'a list -> int -> 'a` telle que `selection l k` est le $(k+1)^{\text{e}}$ plus petit élément de la liste `l` . L'écriture de la fonction sera une traduction en OCaml de l'algorithme 1 présenté en page 4.
On cherche à déterminer la complexité en nombre de comparaisons de la fonction `selection`. Pour tout $n \in \mathbb{N}$, on note $T(n)$ le nombre maximum de comparaisons entre éléments lors d'une sélection d'un élément quelconque dans des listes L sans répétition de taille n.
En analysant l'algorithme 1, il est possible de démontrer que :

$$\forall n \geq 55, T(n) \leq T\left(\left\lfloor \frac{n+4}{5} \right\rfloor\right) + T\left(\left\lfloor \frac{8n}{11} \right\rfloor\right) + 4n. \tag{1}$$

A9. (bonus points) En admettant la proposition 1, montrer que pour tout entier n supérieur à 1 , on a :

$$T(n) \leq (200 + T(55))\, n. \tag{2}$$

Pour l'initialisation, on pourra remarquer que T est une fonction croissante.

Algorithme 1 Sélection du $(k+1)^e$ plus petit élément

```
1 : Fonction SELECTION(L,k)                                   ▷ L est une liste d'entiers, k ∈ ℕ
2 :     n ← LONGUEUR(L)
3 :     si n ≤ 5 alors
4 :         M ← TRI_INSERTION(L)
5 :         renvoyer l'élément de rang k de M
6 :     sinon
7 :         L_Cinq ← PAQUETS_DE_CINQ
8 :         M ← MEDIANS(L_Cinq)
9 :         pivot ← SELECTION(M, ((n+4)//5)//2)   ▷ Le rang correspond au rang du médian de la liste M
10 :        L1, L2, n1, n2 ← PARTAGE(pivot, L)    ▷ L'opérateur // désigne le quotient d'entiers.
11 :        si k < n1 alors
12 :            renvoyer SELECTION(L1, k)
13 :        sinon
14 :            renvoyer SELECTION(L2 (k − n1))
```

B Parcours préfixe d'arbres binaires de recherche (CCINP 2021)

Dans toute la suite, Σ désigne un alphabet fini totalement ordonné. Le symbole ε désigne le mot vide.

■ **Définition 1 — Arbre binaire.** Un arbre binaire T étiqueté par les éléments de Σ est de manière inductive soit :

- l'arbre vide que l'on note $\circ$;
- un triplet (T_g, r, T_d) où r est un élément de Σ, T_g et T_d des arbres binaires. Les éléments r, T_g et T_d sont respectivement appelés racine, sous-arbre gauche et sous-arbre droit de T.

■ **Définition 2 — Arbre binaire de recherche .** Un Arbre Binaire de Recherche (abrégé en ABR) T est inductivement soit :

- l'arbre vide $\circ$;
- un triplet (T_g, r, T_d) où r est un élément de Σ, T_g et T_d des ABR. De plus, toute valeur apparaissant dans T_g est strictement inférieure à r et toute valeur apparaissant dans T_d est supérieure ou égale à r.

L'insertion dans un arbre binaire de recherche T est de manière inductive soit :

- si $T = \circ$, alors $T \leftarrow a = (\circ, a, \circ)$;
- si $T = (T_g, r, T_d)$ et $r \leq a$, alors $T \leftarrow a = (T_g, r, T_d \leftarrow a)$
- si $T = (T_g, r, T_d)$ et $r > a$, alors $T \leftarrow a = (T_g \leftarrow a, r, T_d)$.

On définit récursivement alors l'insertion d'un mot w dans un arbre binaire T noté également $T \leftarrow w$ comme suit :

- si $w = \varepsilon$, alors $T \leftarrow w = T$
- si $w = av$ avec $a \in \Sigma$, alors $T \leftarrow w = (T \leftarrow a) \leftarrow v$

Dans le cas où T est un **ABR**, on admet que $T \leftarrow a$ et $T \leftarrow w$ sont également des **ABR**. Étant donné un mot w sur Σ, **l'arbre binaire de recherche associé à w est l'arbre $\circ \leftarrow w$.**

Étant donné un mot w sur Σ, l'arbre binaire de recherche associé à w est l'arbre $\circ \leftarrow w$.

Dans la suite, les lettres de Σ sont représentées en Ocaml par des char et les mots sur l'alphabet Σ par des char list. Ainsi, le mot "arbre" est représenté par la liste `['a' ; 'r' ; 'b' ; 'r' ; ; e']`.

On représente les arbres binaires en Ocaml à l'aide de la structure arbre suivante :

```
type 'a arbre =   Vide
                | Noeud of ('a arbre) * 'a * ('a arbre)
```

Ainsi, l'arbre $T = (\circ, a, (\circ, b, \circ))$ est représenté en Ocaml par :

```
Noeud(Vide, 'a', Noeud(Vide, 'b', Vide))
```

B10. Représenter le graphe de l'ABR associé au mot FANTASTIQUE, l'ordre sur les lettres étant l'ordre alphabétique.

B11. Écrire une fonction récursive en Ocaml de signature : `insertion_lettre : char -> char arbre -> char arbre` et telle que `insertion_lettre a t` est l'arbre binaire obtenu en insérant la lettre `a` dans l'arbre binaire `t`.

B12. Écrire une fonction récursive en Ocaml de signature `insertion_mot : char list -> char arbre -> char arbre` et telle que `insertion_mot w t` est l'arbre binaire obtenu en insérant le mot `w` dans l'arbre binaire `t`.

■ **Définition 3 — Lecture préfixe .** Soit T un arbre binaire. La lecture préfixe de T que l'on note w_T est le mot défini récursivement par :

- si $T = \circ$, alors $w_T = \varepsilon$;
- si $T = (T_g, r, T_d)$, alors $w_T = r\, w_g\, w_d$ où w_g et w_d sont les lectures préfixes respectives de T_g et de T_d.

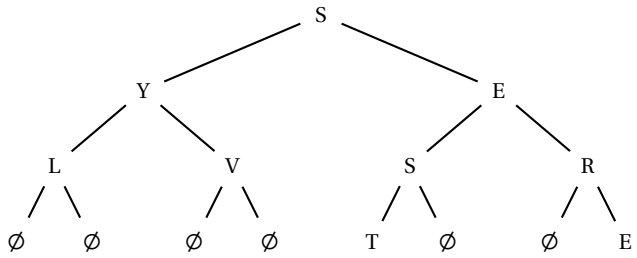

FIGURE 1 – Arbre binaire de la question 4

B13. Expliciter w_T pour l'arbre binaire T représenté sur la figure 1.

B14. Écrire une fonction récursive en Ocaml de signature `prefixe : char arbre -> char list` et telle que `prefixe t` est le mot qui correspond à la lecture préfixe de l'arbre `t`.

B15. Soit T un ABR, soit w_T la lecture préfixe de T. Montrer que T est l'ABR associé à w_T. On pourra utiliser l'application $A : w \to (\circ \leftarrow w)$ qui à un mot fait correspondre l'ABR associé et démontrer la propriété $\mathcal{P} : A(w_T) = T$. Procéder par induction structurelle sur les arbres binaires.

C Implémentation d'un dictionnaire

Un dictionnaire est une structure de données permettant de stocker des éléments en les repérant par des clés. C'est une structure de donnée très utilisée dans la pratique.

■ **Exemple 1 — Mémoriser les variables d'un programme.** Lors de l'exécution d'un programme, celui-ci a souvent besoin d'accéder à la valeur d'une variable à partir de son nom. Un dictionnaire ad-hoc peut être utilisé dans ce cas :

- les clefs sont les noms des variables : des chaînes de caractères
- les valeurs sont des entiers qui peuvent représenter l'adresse de ces variables en mémoire.

```
let a = [|0;1|]
let c = 'z'
and elst = []
and f = 5.7
and i = 3
and lst = [3;4;7]
and s = "arbres";;
```

Par exemple, si on exécute le programme ci-dessous, alors les variables sont stockées en machine à l'aide d'un dictionnaire qui comporte des paires (clef, valeur) :
`("a", 24), ("c", 14568), ("elst", 23), ("f", 42),("i", 7),("lst", 66),("s", 13).`
Les adresses ont ici été choisies arbitrairement pour l'exemple.

Afin d'implémenter la structure de dictionnaire, on se propose d'utiliser des arbres binaires de recherche. Ces arbres binaires de recherche possèdent des nœuds étiquetés par des couples (clé, valeur) et tels que pour tout nœud n de l'arbre :

— La **clef** de n est **strictement** supérieure à toutes les clefs présentes dans le sous-arbre gauche,
— La **clef** de n est **strictement** inférieure à toutes les clefs présentes dans le sous-arbre droit.

R Dans la suite de ce sujet, on suppose donc que les clefs présentes dans un arbre binaire de recherche sont toutes différentes.

R Le dictionnaire de l'exemple précédent est représenté par les arbres de la figure 2 : plusieurs arbres binaires de recherche peuvent représenter le même dictionnaire, celui-ci est équilibré.

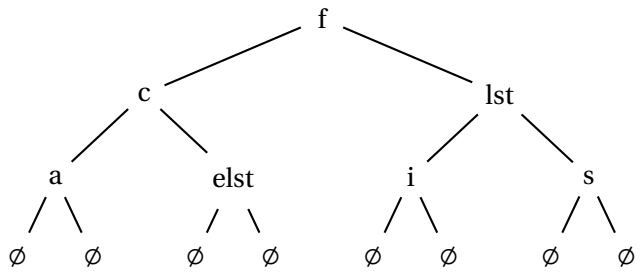

FIGURE 2 – Arbre binaire représentant le dictionnaire de l'exemple 1

En ce qui concerne les éléments OCaml, vous pouvez utiliser les opérateurs `<,=,>,>=, <=` pour comparer deux chaînes de caractères suivant l'ordre lexicographique. Par exemple, toutes les assertions ci-dessous sont vraies :

```
1 "a" < "c";;
2 "c" < "elst";;
3 "f" < "lst";;
4 "elst" < "lst";;
5 "elst" < "s";;
```

Le type `dict` est défini comme suit pour représenter les dictionnaires en OCaml :

```
1 type label = { key: string; value: int;}
2 type dict =   Empty
3             | Node of label * dict * dict
```

R Cette structure de donnée est dite persistante car elle est :

statique c'est-à-dire qu'une fois que la structure de données a été créée en mémoire, la zone mémoire utilisée pour stocker cette structure est figée et de taille fixe.

immuable c'est-à-dire qu'après l'insertion d'une donnée dans la structure lors de l'initialisation, la structure ne peut plus être modifiée.

a Opérations élémentaires sur les dictionnaires

On choisit de représenter un dictionnaire vide par un arbre vide.

C16. Écrire une fonction de signature `create: unit -> dict` qui créé un dictionnaire vide.

C17. En utilisant le filtrage de motifs, écrire une fonction de signature `is_empty: dict -> bool` qui teste si un dictionnaire est vide.

C18. Écrire une fonction de signature `find: dict -> string -> int` qui prend en entrée une clef et renvoie la valeur associée à cette clef. Une exception est levée avec le message `"Key error"` si la clef n'apparait pas dans le dictionnaire.

C19. Écrire une fonction de signature `add: dict -> string -> int -> dict` dont les paramètres d'entrée sont un dictionnaire `d`, une clef `key` ainsi qu'une valeur `val`. Cette fonction ajoute au dictionnaire la paire `key,v`. Une exception est levée avec le message `"Key already recorded"` si la clef apparaît déjà dans le dictionnaire.

C20. Quelles sont les complexités des fonctions `create`, `is_empty`, `find` et `add`? On peut les exprimer en fonction de la hauteur des arbres donnés en entrée.

> ■ **Définition 4 — Arbre binaire parfait.** Un arbre binaire parfait est un arbre dans lequel tous les niveaux sauf le dernier doivent être totalement remplis. Si le dernier n'est pas rempli totalement alors il doit être rempli de gauche à droite.

C21. Montrer qu'un arbre binaire parfait à n nœuds possède une hauteur $h = \lfloor \log_2(n) \rfloor$.

C22. On considère un arbre binaire à n nœuds. Soit f le nombre de feuilles de l'arbre. Montrer la propriété $\mathcal{P} : f \leqslant \frac{n+1}{2}$. On porcèdera par induction structurelle.

D Logique et satisfaisabilité (d'après CCINP 2024)

Une formule propositionnelle est construite à l'aide de constantes propositionnelles, de variables propositionnelles et de connecteurs logiques. Les connecteurs logiques seront notés $\neg$ (négation), $\wedge$ (conjonction), $\vee$ (disjonction). Dans cette partie, on étudie le problème de satisfiabilité d'une formule et son application à la détermination d'une conséquence logique entre deux formules propositionnelles.

Le problème CNF-SAT est défini de la façon suivante : étant donné une formule sous forme normale conjonctive, admet-elle un modèle, c'est-à-dire une valuation des variables, qui rende la formule vraie? On souhaite écrire un programme qui teste si une valuation donnée rend une telle formule vraie.

Dans cette partie, on considère que si une formule contient n variables propositionnelles, elles seront désignées par $x_0, x_1, \ldots, x_{n-1}$.

On définit le type OCaml suivant :

```
type clause =  Var of int
               | Non of clause
               | Ou of clause*clause
```

L'argument du constructeur `Var` correspond au numéro de la variable concernée.

Une formule sous forme normale conjonctive ayant m clauses sera implémentée par une liste de m clauses. Les tableaux seront implémentés par le module `Array` dont les éléments suivants pourront être utilisés :

- `type 'a array`, notations `[| |]`
- création d'un tableau : `Array.make : int -> 'a -> 'a array`
- accès à l'élément d'indice i du tableau `t.(i)`
- modification de l'élément placé à l'indice i du tableau `t.(i)<- v`
- taille du tableau : `Array.length : 'a array -> int`

D23. Donner le code OCaml correspondant à la clause $c = x_0 \vee x_1 \vee \neg x_2$.

D24. Donner le code OCaml permettant de définir la formule : $f = (x_0 \vee x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$.

D25. Démontrer que toute formule logique peut se mettre sous une forme CNF.

D26. Écrire une fonction de signature `evalue_clause : clause -> bool array -> bool` qui prend en paramètre une clause et une valuation représentée par un tableau contenant à l'indice i, la valeur de vérité de la variable x_i et renvoie la valeur de vérité de la clause.

D27. Écrire une fonction de signature `evalue_FNC : clause list -> bool array -> bool` qui prend en paramètre une clause et une valuation représentée par un tableau contenant à l'indice i, la valeur de vérité de la variable x_i et évalue une formule donnée sous forme normale conjonctive.

D28. Quel résultat obtient-on avec la formule F et le tableau de valuations [|false;true;true|] ? Justifier.

On souhaite énumérer toutes les valuations possibles pour un nombre de variables fixé. Étant donné une valuation, on considérera que si la valeur `true` correspond à 1 et la valeur `false` correspond à 0 , la valuation suivante correspond à l'ajout de 1 au nombre binaire associé. Ainsi, la valuation suivante de [|false;true;false|] est [|false;true;true|]. On considère que la valuation suivante de [|true;true;true|] n'existe pas.

D29. Écrire une fonction de signature `suivant : bool array -> bool` qui prend en paramètre un tableau de booléens, lui attribue la valuation "suivante" si possible et renvoie `true`; sinon renvoie `false`.

D30. En déduire une fonction de signature `satisfiable : clause list -> int -> bool` qui prend en paramètre une formule en forme normale conjonctive, son nombre de variables et renvoie `true` s'l existe une valuation qui rend la formule vraie, `false` sinon.

D31. Quelle est la complexité en temps de cette fonction par rapport aux paramètres d'entrée ?

D32. Proposer une stratégie de retour sur trace pour résoudre le problème de satisfiabilité d'une formule.