DS INFO N°5

Ce sujet comporte 6 pages. Il comporte 4 exercices et 1 problème.

Vous numéroterez chaque copie double, en prenant soin de mettre votre nom sur chaque copie.

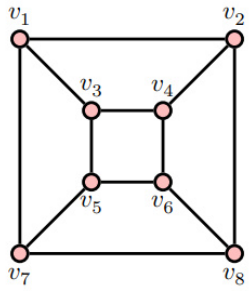
Les stylos effaçables et le crayon à papier sont interdits.

Les algorithmes et les choix d'implémentation devront être expliqués avec clarté et concision.

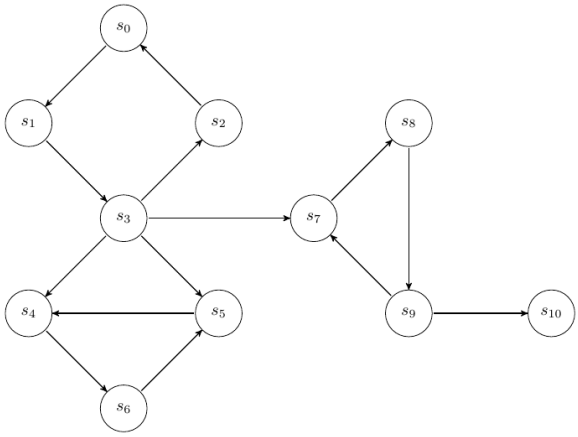
On s'attachera à respecter scrupuleusement les prototypes de fonctions fournis.

Exercice 1 (Questions - Applications directes du cours.).

1. Montrer que le graphe ci-dessous est biparti.

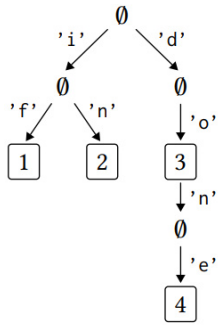


2. Redessiner et entourer (ou colorier) les composantes fortement connexes du graphe ci-dessous. Donner le résultat obtenu en empilant successivement les étiquettes des sommets terminés lors d'un parcours en profondeur, en partant du sommet s_3. On fera l'hypothèse que la liste des successeurs est triée par étiquettes croissantes des successeurs. On pourra détailler l'exécution du tri en profondeur récursif pour ne pas se tromper. S'agit-il d'un tri topologique ?



3. Montrer qu'un graphe non orienté ayant un nombre fini de sommets et dont tout sommet est de degré supérieur ou égal à 2 possède au moins un cycle.

L'exercice suivant présente une structure d'arbre permettant d'implémenter un dictionnaire dont les clés sont des chaînes de caractères.
Ces arbres sont appelés **arbres préfixes**, et plus connu en anglais sous le nom *trie*.[1]
Voici par exemple l'arbre représentant le dictionnaire associant des chaînes de caractères à des entiers et contenant les entrées suivantes : {`"if"` $\mapsto$ 1, `"in"` $\mapsto$ 2, `"do"` $\mapsto$ 3, `"done"` $\mapsto$ 4}
Dans ces arbres, chaque branche est étiquetée par un caractère. Un nœud contient une valeur si la séquence de lettres menant de la racine de l'arbre à ce nœud est une entrée dans le dictionnaire. Si cette séquence de caractères n'est pas une entrée, le nœud peut exister mais n'a pas de valeur associée, ce que l'on note $\emptyset$. Par exemple, dans l'arbre ci-dessus, la clé `"don"` n'est pas une entrée du dictionnaire. Le noeud se trouvant au bout du chemin associé à cette clé existe (car il a été crée lors de l'ajout de la clé `"done"`), mais est vide.



Chaque nœud est repéré de manière unique par son chemin depuis la racine, ce chemin étant représenté par le mot w formé par tous les caractères associés aux branches étiquetées successivement empruntées dans l'arbre pour arriver jusqu'à ce nœud. En particulier, la racine correspond au mot vide. Le nœud contient soit $\emptyset$ s'il n'y a pas de valeur associée à w, soit la valeur associée à w, ici encadrée.
On rappelle les fonctions associées au module `Hashtbl` en OCaml. Seules les fonctions figurant dans cette liste sont autorisées dans la suite.

`Hashtbl.create: int -> ('a, 'b) Hashtbl.t :` constructeur. L'utilisateur doit préciser le nombre de seaux m souhaités.

`Hashtbl.add: ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit :` transformateur, ajout d'un couple clé-valeur dans la table.

`Hashtbl.find_opt: ('a, 'b) Hashtbl.t -> 'a -> 'b :` accesseurs, cherche si une clé est présente dans la table de hachage et retourne la valeur associée sous la forme d'un type `'b option`

`Hashtbl.remove: ('a, 'b) Hashtbl.t -> 'a -> unit :` transformateur, retire le couple clé-valeur associé à une clé k de la table de hachage. Si la clé n'est pas présente, cette fonction ne fait rien.

`Hashtbl.replace: ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit :` transformateur, remplace la valeur associée à une clé par une nouvelle valeur si elle existe, sinon, elle ajoute le couple clé-valeur.

`Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit :` applique une fonction `f: 'a -> 'b -> unit` sur toutes les entrées de la table de hachage.

`Hashtbl.length : ('a, 'b) Hashtbl.t -> int :` donne le nombre d'entrées dans une table de hachage.

1. Le mot *trie* vient du mot re**trie**val, rechercher, extraire.

On autorise également l'utilisation de la fonction `String.length`.

Exercice 2 (Arbres préfixes - Exercice de programmation).
Toutes les fonctions de cet exercice seront écrites en OCaml.
On propose le type OCaml suivant pour représenter de tels arbres :

```
type 'v trie =
 {
    mutable node_val : 'v option;
    branches : (char, 'v trie) Hashtbl.t
 };;
```

et on suppose que l'on a écrit un constructeur `trie_create : unit -> 'v trie` créant un arbre constitué d'un seul nœud racine vide.

1. Expliquer en quelques phrases le type OCaml proposé.
2. Écrire une fonction `trie_add: 'v trie -> string -> 'v -> unit` qui insère une entrée dans notre dictionnaire implémenté avec un arbre préfixe. Si la clé était déjà présente, la fonction se contente de remplacer la valeur associée à la clé par la nouvelle valeur.
3. Écrire une fonction `trie_find_opt: 'v trie -> string -> 'v option` qui cherche une valeur associée à une clé dans notre dictionnaire et gère le cas où cette valeur n'est pas présente en renvoyant un objet construit avec le constructeur `None`.
4. Écrire une fonction `trie_remove: 'v trie -> string -> bool` qui cherche une valeur associée à une clé dans un arbre préfixe et supprime la valeur associée dans l'arbre. Si la clé n'a pas été trouvée, la fonction renvoie la valeur `false`.
5. Citer un algorithme où vous avez utilisé des arbres préfixes. Quel était le type des clés dans ce cas ?

Exercice 3 (Logique propositionnelle (CCINP 2018)).
On définit le connecteur de Sheffer, ou d'incompatibilité, par $x_1 \diamond x_2 = \neg x_1 \vee \neg x_2$.

1. Construire la table de vérité du connecteur de Sheffer.
2. Exprimer ce connecteur en fonction de $\neg$ et $\wedge$.
3. Vérifier que $\neg x_1 \equiv x_1 \diamond x_1$.
4. En déduire une expression des connecteurs $\wedge$, $\vee$ et $\rightarrow$ en fonction du connecteur de Sheffer. Justifier en utilisant des équivalences avec les formules propositionnelles classiques.
5. Démontrer par induction sur les formules propositionnelles que l'ensemble de connecteurs $\mathcal{C} = \{\diamond\}$ est un système complet.
6. Application : soit $\mathcal{F}$ la formule propositionnelle $x_1 \vee (\neg x_2 \wedge x_3)$. Donner une forme logiquement équivalente de $\mathcal{F}$ utilisant uniquement le connecteur de Sheffer.

Exercice 4 (Modélisation).
Un voyageur perdu dans le désert arrive à une bifurcation à partir de laquelle sa piste se sépare en deux. Chaque piste peut soit mener à une oasis, soit se perdre dans un désert profond. Chaque piste est gardée par un sphinx. Les données du problème sont les suivantes :

A. le sphinx de droite dit : " Une au moins des 2 pistes conduit à une oasis "

B. le sphinx de gauche dit : " La piste de droite se perd dans le désert "

C. soit les deux sphinx disent la vérité, soit ils mentent tous les deux

Le voyageur aimerait bien savoir s'il y a une oasis au bout d'un des deux chemins et si oui quelle direction prendre.

1. Introduire deux variables propositionnelles pour modéliser le problème (expliciter ce qu'elles représentent) et traduire les trois données en formules propositionnelles ;
2. Par la méthode de votre choix, résoudre l'énigme en justifiant votre réponse.

Problème - Listes triables par pile (CCINP 2023)

Toutes les fonctions de cet exercice seront écrites en OCaml. On s'interdit d'utiliser les traits impératifs du langage OCaml (références, tableaux, champs mutables, etc.).

On représente en OCaml une permutation σ de $[\![0, n-1]\!]$ par la liste d'entier $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$.

Un arbre binaire étiqueté est soit un arbre vide, soit un nœud formé d'un sous-arbre gauche, d'une étiquette et d'un sous-arbre droit.

On représente un arbre binaire non étiqueté par un arbre binaire étiqueté en ignorant simplement les étiquettes. On étiquette un arbre binaire non étiqueté à n nœuds par $[\![0; n-1]\!]$ en suivant l'ordre infixe de son parcours en profondeur. La permutation associée à cet arbre est donnée par le parcours en profondeur par ordre préfixe. La figure 1 propose un exemple (on ne dessine pas les arbres vides).

1. Définir un type OCaml `arbre` permettant de représenter un arbre binaire étiqueté par des entiers.
2. Redessiner et étiqueter l'arbre (c) de la figure 1 et donner la permutation associée.
3. Écrire une fonction `parcours_prefixe : arbre -> int list` qui renvoie la liste des étiquettes d'un arbre dans l'ordre préfixe de son parcours en profondeur. On pourra utiliser l'opérateur @.
4. Écrire une fonction `etiquette : arbre -> arbre` qui prend en paramètre un arbre dont on ignore les étiquettes et qui renvoie un arbre identique mais étiqueté par les entiers de $[\![0, n-1]\!]$ en suivant l'ordre infixe d'un parcours en profondeur.

Une permutation σ de $[\![0, n-1]\!]$ est triable avec une pile s'il est possible de trier par ordre décroissant la liste $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$ en utilisant uniquement une structure de pile comme espace de stockage interne. On utilise une liste résultat pour accumuler le résultat du tri au cours de l'algorithme.
L'algorithme est le suivant, énoncé ici dans un style impératif :

— Initialiser une pile vide ;

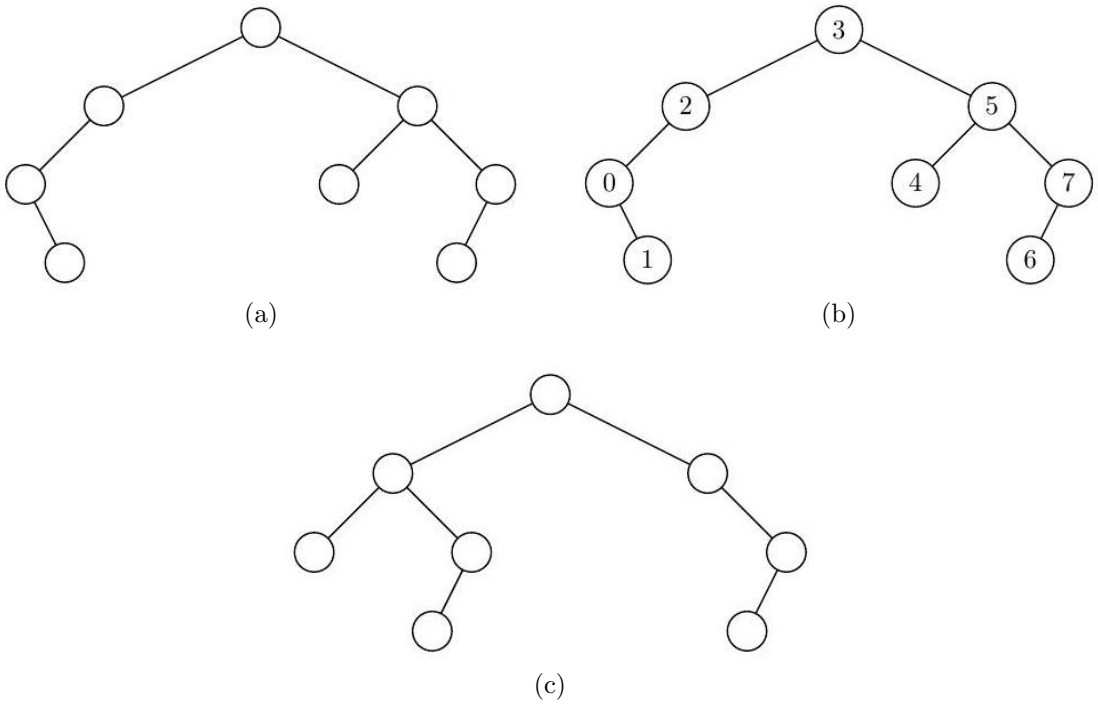


FIGURE 1 – (a) un arbre binaire non étiqueté ; (b) son étiquetage en suivant un ordre infixe, la permutation associée est `[3; 2; 0; 1; 5; 4; 7; 6]` ; (c) un autre arbre binaire non étiqueté

— Pour chaque élément en entrée :

  — Tant que l'élément est plus grand que le sommet de la pile, dépiler le sommet de la pile vers la liste résultat ;

  — Empiler l'élément en entrée dans la pile ;

— Dépiler tous les éléments restant dans la pile vers la liste résultat

Par exemple, pour la permutation `[3; 2; 0; 1; 5; 4; 7; 6]`, on empile $3, 2, 0$, on dépile 0, on empile 1, on dépile $1, 2, 3$, on empile $5, 4$, on dépile $4, 5$, on empile $7, 6$, on dépile $6, 7$.
On obtient la liste triée `[7; 6; 5; 4; 3; 2; 1; 0]` en supposant avoir ajouté en sortie les éléments dans une liste, en insérant à chaque fois en tête de liste comme cela est le cas naturellement en OCaml.

On admet qu'une permutation est triable par pile si et seulement si cet algorithme permet de la trier correctement.

On remarquera que la pile intermédiaire est toujours triée par ordre croissant (de la tête à la queue).

5. Dérouler l'exécution de cet algorithme sur la permutation associée à l'arbre (c) de la figure 1 et vérifier qu'elle est bien triable par pile. On représentera chaque étape de l'algorithme sur une ligne et on décrira, sur chaque ligne : l'état courant de la liste donnée en argument, l'état de la pile intermédiaire et l'état de la liste servant à accumuler le résultat.
6. Écrire une fonction `trier : int list -> int list` qui implémente cet algorithme dans un style fonctionnel. Par exemple, `trier [3; 2; 0; 1; 5; 4; 7; 6]` doit

renvoyer la liste `[7; 6; 5; 4 ; 3 ; 2 ; 1 ; 0]`. On utilisera directement une liste pour implémenter une pile.

7. Montrer que s'il existe $0 \leq i < j < k \leq n-1$ tels que $\sigma_k < \sigma_i < \sigma_j$, alors σ n'est pas triable par une pile.
8. On se propose de montrer que les permutations de $[\![0, n-1]\!]$ triables par une pile sont en bijection avec les arbres binaires non étiquetés à n nœuds.

   (a) Montrer que la permutation associée à un arbre binaire est triable par pile. On pourra remarquer le lien entre le parcours préfixe et l'opération empiler d'une part et le parcours infixe et l'opération dépiler d'autre part.

   (b) Montrer qu'une permutation triable par pile est une permutation associée à un arbre binaire. Indication : on peut prendre σ_0 comme racine, puis procéder récursivement avec les $\sigma_0 - 1$ éléments pour construire le fils gauche, et avec le reste pour le fils droit.

   (c) On note AB_n l'ensemble des arbres binaires non étiquetés à n nœuds et P_n l'ensemble des permutations de $[\![0, n-1]\!]$. On note enfin Ptp_n l'ensemble des permutations de $[\![0, n-1]\!]$ triables par pile. Nous avons étudié dans les questions précédentes une application $f : AB_n \to P_n$.
   Quel est l'ensemble image I de cette fonction ?
   Que reste-t-il à montrer sur la fonction $\tilde{f} : AB_n \to I$ pour prouver que les permutations de $[\![0, n-1]\!]$ triables par une pile sont en bijection avec les arbres binaires non étiquetés à n nœuds ?
   Proposez une démonstration pour cette dernière étape.