Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml. Dans le cas du C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`, `<stdassert.h>`, ...) sont déjà importées.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *Questions de cours*

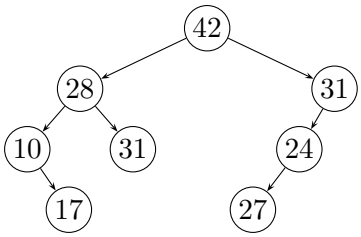
Q1– Donner la définition d'un arbre binaire.

Q2– Donner les définitions de la hauteur et de la taille d'un arbre binaire.

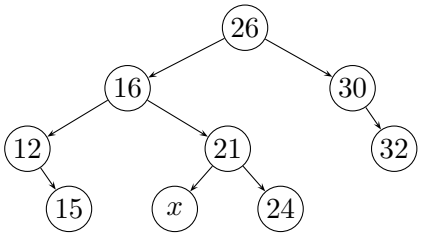
Q3– Donner la définition d'un arbre binaire de recherche.

Q4– Prouver le parcours infixe d'un arbre binaire de recherche fournit les clés dans l'ordre croissant.
Indication : on pourra raisonner par récurrence sur la taille de l'arbre.

Q5– Donner l'ordre des noeuds lors des parcours prefixe, infixe et suffixe de l'arbre suivant :



Q6– On considère l'arbre binaire suivant :



Donner les valeurs de l'étiquette x pour lesquelles cet arbre est un arbre binaire de recherche.

Q7– On implémente les arbres binaires de recherche en OCaml à l'aide du type suivant :

```
type abr =
  | Vide
  | Noeud of  abr * int * abr;;
```

Ecrire une fonction `insere : int -> abr -> abr` qui prend en argument un entier x et un arbre binaire de recherche a et renvoie un arbre binaire de recherche contenant x et tous les éléments de a.

❑ Exercice 2 : *Valeur plus petite la plus proche*

On considère un tableau d'entiers *positifs* et on s'intéresse au problème de la recherche pour chacun de ces entiers de la valeur plus petite la plus proche située à gauche dans le tableau. Dans le cas où aucune valeur située à gauche dans le tableau n'est plus petite que la valeur considérée alors on renverra -1.
Par exemple dans le tableau $\{2, 1, 7, 9, 8, 3\}$:

— Il n'y a aucune valeur à gauche de 2, donc la valeur plus petite la plus proche est -1,

— Pour 1, aucune valeur située à gauche n'est plus petite, donc on renvoie aussi -1,

— Pour 7, la valeur plus petite la plus proche est 1.

— Pour 9, c'est 7.

— Pour 8 c'est 7.

— Pour 3, c'est 1.

Et donc le tableau des valeurs plus petites les plus proches dans cet exemple est $\{-1, -1, 1, 7, 7, 1\}$

Q8– Donner le tableau des valeurs plus petites les plus proches pour le tableau $\{5, 7, 11, 6, 9, 2\}$

Q9– On propose l'algorithme suivant pour résoudre ce problème : pour chaque élément `tab[i]` du tableau on parcourt les valeurs `tab[i-1], ..., tab[0]` *dans cet ordre*, si on trouve un élément strictement inférieur à `tab[i]` alors c'est la valeur plus petite la plus proche, sinon la valeur plus petite la plus proche est -1. Ecrire une implémentation de cet algorithme en C sous la forme d'une fonction de signature `int *vpp_naif(int tab[], int size)` qui prend en argument un tableau d'entiers `tab` ainsi que sa taille `size` et un renvoie un tableau de taille `size` contenant à l'indice `i` la valeur strictement inférieure la plus proche de `tab[i]`.

Q10– Justifier rapidement que l'algorithme précédent a une complexité quadratique

On considère maintenant l'algorithme suivant qui utilise une pile dotée de son interface usuelle (`est_vide, empiler, depiler`) et de la fonction `sommet` qui renvoie la valeur située au sommet de la pile sans la dépiler.

Algorithme : Valeurs plus petites les plus proches

Entrées : Un tableau t d'entiers positifs de taille n
Sorties : Un tableau s d'entiers positifs de taille n tel que $s[i]$ soit la valeur plus petite la plus proche de $t[i]$

```
1  s ← tableau de taille n
2  p ← pile de taille maximale n
3  pour i ← 0 à p − 1 faire
4      tant que p n'est pas vide et sommet(p) ⩾ t[i] faire
5          depiler(p) ;
6      fin
7      si p est vide alors
8          s[i] ← −1
9      fin
10     sinon
11         s[i] ← sommet(p)
12     fin
13     empiler t[i] dans p
14 fin
15 return s
```

Q11– On fait fonctionncer cet algorithme sur le tableau $\{2, 7, 5, 8, 6, 3\}$. Recopier et compléter le tableau suivant qui indique pour chaque valeur de l'indice `i` de la boucle `for` l'état de la pile et du tableau s après l'exécution de la boucle pour les valeurs de `i` de 0 à 5 (on note une pile avec les extrémités `|` et `>` pour indiquer le sommet de la pile)

i	État de la pile	État du tableau s
Initialement	`|>`	$\{-1, -1, -1, -1, -1, -1\}$
0	`|2>`	$\{-1, -1, -1, -1, -1, -1\}$
1	`|2, 7>`	$\{-1, 2, -1, -1, -1, -1\}$
2	...	...
3	...	...
4	...	...
5	...	...

Q12– On suppose qu'on a *déjà implémentée* en C une structure de donnée de pile qu'on manipule à l'aide des fonctions suivantes :

— `est_vide` de signature `bool est_vide(pile p)`,

— `empiler` de signature `void empiler(pile *p, int v)`,

— `depiler` de signature `int depiler(pile *p)`.

Ecrire en utilisant ces fonctions une fonction `sommet` de signature `int sommet(pile *p)` qui renvoie le sommet de la pile sans le depiler si la pile n'est pas vide et -1 sinon.

Q13– Ecrire une implémentation en C de l'algorithme des valeurs plus petites les plus proches donné ci-dessus et utilisant une pile sous la forme d'une fonction de signature `int *vpp_pile(int tab[], int size)` qui renvoie le tableau des valeurs plus petites les plus proches.

Q14– Prouver que cet algorithme est de complexité linéaire, on pourra vérifier que chaque élément du tableau t est empilé une fois et dépilé au plus une fois.

❑ **Exercice 3** : *Base de données de publications scientifiques*

On utilise le schéma relationnel suivant afin de modéliser une base de données de publications scientifiques. Chaque article publié ayant un ou plusieurs auteurs.

- **Article** (IdArticle, titre, revue, volume, annee)
- **Auteur** (IdAuteur, nom, prenom)
- **Publie** (#Article,#Auteur)

La clé étrangère #Article de la table **Publie** fait référence à la clé primaire de la table **Article** et la clé étrangère #Auteur de la table **Publie** fait référence à la clé primaire de la table **Auteur**. Les attributs titre, revue, nom et prenom sont des chaines de caractères, les autres sont des entiers.

Q15– Justifier que l'attribut #Article de la table **Publie** seul, ne peut pas servir de clé primaire pour cette table.

Q16– Expliquer ce qu'affiche la requête suivante :

```
SELECT nom, prenom
FROM Auteur
JOIN Publie ON Auteur.IdAuteur = Publie.Auteur
WHERE Publie.Article = 42
```

Q17– Ecrire une requête permettant d'obtenir la liste des titres des articles parus en 2022 listé par ordre alphabétique.

Q18– Ecrire une requête permettant d'obtenir les noms des revues listé par ordre alphabétique. On souhaite obtenir cette liste *sans répétition* des noms de revues.

Q19– Ecrire une requête permettant d'obtenir les noms et prénoms des auteurs qui ont publié dans la revue "Nature" en 2000.

Q20– Ecrire une requête permettant d'obtenir les titres et revues des articles écrits (ou co-écrit) par Donald Knuth en 2010.

Q21– Ecrire une requête permettant d'obtenir la liste des volumes de la revue "Nature" en 2020 avec le nombre d'article qu'il contient.

Q22– Ecrire une requête permettant d'obtenir pour chaque revue, son nom et l'année de publication de son article le plus ancien.

❑ **Exercice 4** : *Hachage de chaine de caractères*

Le langage d'implémentation dans cet exercice est le langage C, on suppose déjà importées les librairies `<stdint.h>` et `<string.h>` et on s'intéresse aux fonction de hachages sur des chaines de caractères constituées uniquement des lettres de l'alphabet (minuscules ou majuscules), des chiffres de 0 à 9 et des caractères spéciaux `_` et `*`. On remarquera que cela fait un total de **64** caractères possibles.

Q23– Justifier rapidement qu'il est préférable d'utiliser un type entier non signé du langage C (`uint8_t, uint32_t, uint64_t`) comme type de retour de la fonction de hachage.

Q24– On suppose que les chaines de caractères ont une longueur fixe de huit caractères. Montrer que si la valeur de hachage est stockée sous la forme du type `uint32_t` du langage C, alors il est certain que deux valeurs distinctes entrent en collision.

Q25– Toujours en supposant des chaines de longueur fixe de huit caractères, montrer que si la valeur de hachage est stockée sous la forme du type `uint64_t` du langage C, alors on peut trouver une fonction de hachage ne provoquant aucune collision.

Dans la suite de l'exercice, les chaines de caractères à hacher peuvent être de n'importe quelle longueur et on numérote les 64 caractères possibles de la façon suivante :

— les lettres majuscules portent les numéros 1 à 26,

— les lettres minuscules 27 à 52,

— les chiffres portent les numéros 53 à 62

— la caractère `_` a le numéro 63 et `*` le 64.

Et on propose la fonction de hachage suivant pour une chaine s de longueur n constituée des caractères $c_0, \dots, c_{n-1}$:
$h(s) = \sum_{i=0}^{n-1} N(c_i) * 32^i$
où $N(c_i)$ est le numéro du caractère c_i.
On suppose déjà écrite une fonction de signature `int num(char c)` qui renvoie le numéro attribué à un des 64 caractères possibles, par exemple `num('A')` renvoie 1, `num('a')` renvoie 27.

Q26– Expliquer pourquoi l'implémentation suivante de la fonction de hachage est de complexité quadratique en la longueur de la chaine `s` et proposer une correction afin de rendre cette complexité linéaire.

```
uint64_t hash(char *s)
{
    uint64_t h = 0;
    for (int i = 0; i < strlen(s); i++)
    {
        h = h * 32 + num(s[strlen(s) - 1 - i]);
    }
    return h;
}
```

Q27– Déterminer deux chaines de longueur 2 qui entrent en collision.

Q28– Montrer qu'il est possible de construire des chaines de longueurs arbitraires entrant en collision.

❑ **Exercice 5** : *Détection de cycle*

Le langage d'implémentation dans cet exercice est OCaml.

Etant donné un entier $N \in \mathbb{N}^*$, on considère la fonction :

$$\begin{array}{rccc} f_N : & \mathbb{Z}^2 & \mapsto & \mathbb{Z}^2 \\ & (x,y) & \longrightarrow & ((x+2y^2) \mod N, (3x^2+y) \mod N) \end{array}$$

Pour un couple $(x_0, y_0) \in \mathbb{Z}^2$, on définit la suite $(u_n)_{n\in\mathbb{N}}$ par $u_0 = (x_0, y_0)$ et $u_{n+1} = f_N(u_n)$.

Q29– Vérifier sur l'exemple $N = 10$ et $(x_0, y_0) = (1, 8)$ en calculant manuellement les premiers termes que la suite contient un cycle. Quelle est la longueur de ce cycle ?

Q30– Justifier que pour toute valeur de N fixée et tout couple initial (x_0, y_0), la suite (u_n) contient nécessairement un cycle. On pourra remarquer que l'ensemble des couples possibles est fini.

Q31– Ecrire la fonction f_N en OCaml avec la signature : `let f int -> int -> int-> int * int` en donnant les paramètres dans cet ordre : `x`, `y` et `N`.

Q32– Ecrire une fonction `let appartient 'a -> 'a list -> bool` qui teste si un élément appartient à une liste. Donner la complexité de cette fonction en fonction de la taille de la liste.

Q33– Ecrire une fonction `let cycle_naif int -> int -> int -> int * int` qui prend en argument (dans cet ordre) x_0, y_0 et N et détecte le cycle en maintenant à jour une liste des couples déjà rencontrés, et renvoie le premier couple qui apparaît deux fois.

Q34– Quelle est la complexité dans le pire cas de la fonction `cycle_naif` en fonction du nombre de termes à parcourir avant de trouver le premier couple apparaissant deux fois ?

On propose maintenant d'utiliser une table de hachage afin de stocker les termes déjà rencontrés, les clés sont les couples rencontrées et la valeur associée est l'indice dans la suite de ce couple. Par exemple si les trois premiers termes de la suite sont $(1,5),(4,3),(7,7)$ alors la table de hachage doit contenir les clés $(1,5),(4,3)$, et $(7,7)$ associées respectivement aux valeurs $0,1$ et 2. On rappelle ci-dessous les fonctions de manipulation d'une table de hachage en OCaml :

- `Hashtbl.create : int -> ('a,'b) Hashtbl.t` renvoie une table de hachage dont on donne la taille initiale.
- `Hashtbl.mem ('a, 'b) Hashtbl.t -> 'a -> bool` renvoie `true` si la clé donnée en argument apparait dans la table de hachage.
- `Hashtbl.add ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` pour ajouter un couple (clé, valeur) à une table de hachage.
- `Hashtbl.find ('a, 'b) Hashtbl.t -> 'a -> 'b` renvoie la valeur associée à une clé.

Q35– Justifier rapidement que l'utilisation d'une table de hachage pour stocker les couples déjà rencontrés permet d'obtenir une complexité meilleure que celle de la fonction `cycle_naif`.

Q36– En utilisant une table de hachage de OCaml, écrire une version de la fonction de recherche de cycle. Cette fonction devra renvoyé le premier élément rencontré à deux reprises *ainsi que la longueur du cycle*.