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*

1. Donner la définition d'un arbre binaire.

   Un arbre binaire est une structure de données hiérarchique composée de noeuds définie récursivement, en effet un arbre binaire est :
   - soit vide, on le note alors $\varnothing$
   - soit un noeud c'est à dire un triplet (g, v, d) où g et d sont deux arbres binaires et v l'étiquette.

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

   - Le nombre de noeuds d'un arbre binaire A, noté $n(A)$, se définit récursivement par :
   $$\begin{cases} n(A) = 0 & \text{si } A \text{ est vide} \\ n(A) = 1 + n(g) + n(d) & \text{si } A = (g, a, d) \end{cases}$$
   - La hauteur d'un arbre binaire A, noté $h(A)$, se définit récursivement par :
   $$\begin{cases} h(A) = -1 & \text{si } A \text{ est vide} \\ h(A) = 1 + \max(h(g), h(d)) & \text{si } A = (g, a, d) \end{cases}$$

3. Donner la définition d'un arbre binaire de recherche.

   Un arbre binaire de recherche (noté ABR), est un arbre binaire tel que :
   - Les étiquettes des noeuds, appelées clés sont toutes comparables entre elles.
   - Pour tous les noeuds (g, v, d) l'ensemble des clés présentes dans le sous arbre gauche g (resp. droit d) sont strictement inférieures (resp. supérieures) à v.
   - Les clés sont uniques.

4. 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.

Pour tout $n \in \mathbb{N}$, on note $\mathcal{P}(n)$ la propriété : « le parcours infixe d'un ABR de taille n fournit les clés dans l'ordre croissant ».
• Initialisation : $\mathcal{P}(0)$ est vraie puisque le parcours infixe d'un arbre vide est vide et donc rangé dans l'ordre croissant.
• Hérédité : soit $n \in \mathbb{N}$, tel que pour tout $k \leqslant n$, $\mathcal{P}(k)$ est vrai, montrons alors que $\mathcal{P}(n+1)$ est vraie. Soit un arbre binaire de taille $n+1$, alors cet arbre n'est pas vide et donc c'est un triplet (g, e, d) où g et d sont des arbres binaires de taille inférieure ou égale à n. En notant $p(a)$ le parcours prefixe d'un arbre a, on a par définition du parcours infixe : $p((g, e, d)) = (p(g), e, p(d))$, on considère maintenant x, y deux éléments apparaissant *dans cet ordre* dans $(p(g), e, p(d))$ et on raisonne par disjonction de cas :

— $x \in p(g)$ et $y \in p(g)$, par hypothèse de récurrence le parcours infixe de g est rangé dans l'ordre croissant et donc $x < y$.

— $x \in p(g)$ et $y = e$, par la propriété des ABR les clés du sous arbres gauche sont strictement inférieur à la racine donc $x < y$.

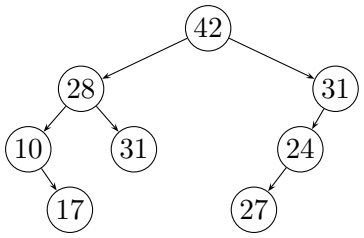
— $x \in p(g)$ et $y \in p(d)$, par la propriété des ABR, $x < e$ et $e < y$, donc $x < y$.

— $x = e$ et $y \in p(d)$, par la propriété des ABR, $y > e$ donc $x < y$.

— $x \in p(d)$ et $y \in p(d)$, par hypothèse de récurrence le parcours infixe de d est rangé dans l'ordre croissant et donc $x < y$.

Donc $P(n+1)$ est vraie.

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



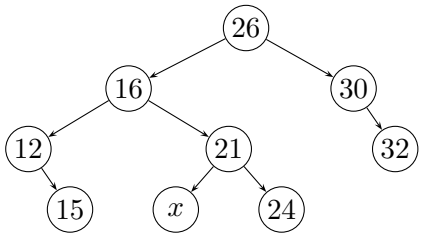
Voici l'ordre des noeuds dans chacun des parcours :

— Prefixe : $42, 28, 10, 17, 31, 31, 24, 27$

— Infixe : $10, 17, 28, 31, 42, 27, 24, 31$

— Suffixe : $17, 10, 31, 28, 27, 24, 31, 42$

6. On considère l'arbre binaire suivant :



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

On peut écrire le parcours infixe de cet arbre : $12, 15, 16, x, 21, 24, 26, 30, 32$ et donc les valeurs de x pour lesquelles cet arbre est un ABR sont celles de l'intervalle $]16, 21[$ en supposant x entier, les valeurs possibles de x sont donc $17, 18, 29$ et 20.

7. 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.

```
let rec insere abr nv =
  match abr with
  | Vide -> Noeud(Vide,nv,Vide)
  | Noeud(g,v,d) -> if nv<v then Noeud(insere g nv, v, d) else Noeud(g, v, insere
  ↪  d nv);;
```

❑ **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 $\{-1, -1, 1, 7, 7, 1\}$

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

   On obtient : $\{-1, 5, 7, 5, 6, -1\}$

2. 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]`.

```c
int *vpp_naif(int tab[], int size)
// Renvoie pour chaque indice i, l'emplacement de plus proche valeur inférieure -1 si inexistant
{
    int *nsv = malloc(sizeof(int) * size);
    int idx;
    nsv[0] = -1;
    for (int i = 1; i < size; i++)
    {
        idx = i - 1;
        while (idx >= 0 && tab[idx] >= tab[i])
        {
            idx--;
        }
        if (idx < 0)
        {
            nsv[i] = -1;
        }
        else
        {
            nsv[i] = tab[idx];
        }
    }
    return nsv;
}
```

3. Justifier rapidement que l'algorithme précédent a une complexité quadratique

> Pour chaque indice `i` du tableau, on parcourt dans le pire cas, le sous tableau `i-1, ..., 0` en effectuant des opérations élémentaires. On effectue donc au plus $1 + 2 + \dots (n-1)$ opérations élémentaires. Et donc la complexité est 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 $t[i]$

```
s ← tableau de taille n
p ← pile de taille maximale n
pour i ← 0 à p − 1 faire
    tant que p n'est pas vide et sommet(p) ⩾ t[i] faire
        depiler(p) ;
    fin
    si p est vide alors
        s[i] ← −1
    fin
    sinon
        s[i] ← sommet(p)
    fin
    empiler t[i] dans p
fin
return s
```

4. 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	`|1, 3>`	[-1, 1, 1, -1, -1, -1]
3	[1, 3, 9]	[-1, 1, 1, 3, -1, -1]
4	[1, 3, 8]	[-1, 1, 1, 3, 3, -1]
5	[1, 4]	[-1, 1, 1, 3, 3, 1]

5. 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.

```c
int sommet(pile *p)
{
    if (est_vide(*p))
    {
        return -1;
    }
    int temp = depiler(p);
    empiler(p, temp);
    return temp;
}
```

6. 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.

```c
int *vpp_pile(int tab[], int size)
{
    pile p = cree_pile(size);
    int *nsv = malloc(sizeof(int) * size);
    for (int i = 0; i < size; i++)
    {
        while (!est_vide(p) && sommet(&p) >= tab[i])
        {
            depiler(&p);
        }
        nsv[i] = sommet(&p);
        empiler(&p, tab[i]);
    }
    return nsv;
}
```

7. 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.

La boucle `for` ne contient que des instructions depiler, empiler et d'affectation dans le tableau `nsv`, chaque élément n'est empilé qu'une seule fois il y en donc `n` opérations `empiler` en tout, on ne peut donc pas dépiler plus de `n` fois et donc l'algorithme est de complexité linéaire.

❑ **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 prénom sont des chaines de caractères, les autres sont des entiers.

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

   L'énonce indique qu'un article peut avoir plusieurs auteurs, par conséquent dans la table publie, plusieurs enregistrements peuvent avoir la même valeur pour le champ Article. Donc cette valeur n'est pas unique pour chaque enregistrement et donc ne peut pas servir de clé primaire.

2. Expliquer ce qu'affiche la requête suivante :

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

   Cet requête affiche les noms et prénoms des auteurs de l'article ayant l'IdArticle 42.

3. Ecrire les requêtes permettant d'afficher les informations suivante :
   a) La liste des titres des articles parus en 2022 listé par ordre alphabétique.

```sql
SELECT titre
FROM Article
WHERE annee = 2022
ORDER BY titre ASC ;
```

   b) Les noms des revues listé par ordre alphabétique, sans répetition.

```sql
SELECT DISTINCT revue
FROM Article
ORDER BY titre ASC ;
```

   c) Les noms et prénoms des auteurs qui ont publié dans la revue "Nature" en 2000.

```sql
SELECT nom, prenom FROM Auteur
JOIN Publie ON Publie.Article = Auteur.IdAuteur
JOIN Article ON Article.IdArticle = Publie.Article
WHERE Article.revue = "Nature" AND Article.annee = 2000
```

   d) Les titres et revues des articles écrits (ou co-écrit) par Donald Knuth en 2010.

```sql
SELECT titre, revues FROM Article
JOIN Publie ON Publie.Article = Auteur.IdAuteur
JOIN Article ON Article.IdArticle = Publie.Article
WHERE Auteur.prenom = "Donald" AND Auteur.nom= "Knuth" AND Article.annee = 2010
```

   e) La liste des volumes de la revue "Nature" en 2020 avec le nombre d'article qu'il contient.

```
SELECT volume, COUNT(*) FROM Article
GROUPE BY volume
WHERE Article.annee = 2020 and Article.revue = "Nature"
```

f) Pour chaque revue, son nom et l'année de publication de son article le plus ancien.

```
SELECT revue, MIN(annee) FROM Article
GROUPE BY revue
```

❑ **Exercice 4** : *Représentations classiques d'ensembles* 🎓 d'après CCSE 2021 - MP (Partie 2)

Les programmes de cet exercice doivent être écrits en OCaml.

On s'intéresse dans cet exercice à des structures de données représentant des ensembles d'entiers naturels. On notera $|E|$ le cardinal d'un ensemble E.

■ **Partie I** : Avec une liste d'entiers triés

Dans cette partie uniquement, on implémente un ensemble d'entiers positifs par la liste des ses éléments rangés *dans l'ordre croissant.*. Par exemple la liste `[2; 7; 11]` représente l'ensemble $\{2, 7, 11\}$.

1. Ecrire la fonction `intersection : int list -> int list -> int list` qui prend en argument deux listes d'entiers triés représentant des ensembles et renvoyant leur intersection sous la forme d'une liste triée d'entiers.

```
let rec intersection l1 l2 =
  match l1, l2 with
  | l1, [] -> []
  | [], l2 -> []
  | h1::t1, h2::t2 -> if h1 = h2 then h1::(intersection t1 t2) else
    if h1 < h2 then intersection t1 l2 else intersection l1 t2;;
```

2. Ecrire une fonction `succ_list` de signature `int list -> int -> int` prenant en arguments une liste d'entiers *distincts* dans l'ordre croissant et un entier x et renvoyant le successeur de x dans la liste, c'est à dire le plus petit entier strictement supérieur à x de la liste (-1 si cela n'existe pas). Par `succ_list [2; 7; 11] 5` doit renvoyer 7.

```
let rec succ_list entiers x =
  match entiers with
  | [] -> -1
  | h::t -> if h>x then h else succ_list t x;;
```

3. Donner la complexité de cette fonction dans le pire des cas.

> Les appels récursif sont en temps constant et la taille de la liste diminue de 1 à chaque appel donc la complexité est linéaire en fonction de la taille de la liste.

■ **Partie II** : Avec un tableau trié

Soit N un entier naturel strictement positif, fixé pour toute cette partie. On choisit de représenter un ensemble d'entiers E de cardinal $n \leq N$ par un tableau de taille $N + 1$ dont la case d'indice 0 indique le nombre n d'éléments de E et les cases d'indices 1 à n contiennent les éléments de E *rangés dans l'ordre croissant*, les autres cases étant non significatives. Par exemple, le tableau `[| 3; 2; 5; 7; 9; 1; 14 |]` représente l'ensemble $\{2, 5, 7\}$. En effet, cet ensemble contient 3 éléments car la case d'indice 0 du tableau contient 3 et ces 3 éléments sont 2, 5, 7 (cases d'indice 1 à 3).

1. Pour une telle implémentation d'un ensemble E, décrire brièvement des méthodes permettant de réaliser chacune des opérations ci-dessous (on ne demande pas d'écrire des programmes) et donner leurs complexités dans le pire cas :
   — déterminer le maximum de E,

— tester l'appartenance d'un élément x à E

— ajouter un élément x dans E (on suppose que $x \notin E$ et que la taille du tableau est suffisante)

On note `tab` le tableau représentant l'ensemble d'entiers

— Pour déterminer le maximum de E, il suffit de renvoyer `tab[tab[0]]` car les éléments sont dans l'ordre croissant et leur indice vont de 1 à `tab[0]`. C'est donc une opération en temps constant.

— On doit parcourir le tableau entre les éléments d'indice 1 et `tab[0]` (complexité linéaire), ou alors (puisque le tableau est trié) effectuer une recherche dichotomique (complexité logarithmique).

— On incrémente `tab[0]` et on place l'élément `x` à l'indice `tab[0]`, ensuite, pour que le tableau reste trié, on peut par exemple échanger cet élément avec son voisin tant qu'il lui est inférieur (et qu'on a pas atteint l'indice 1). L'insertion est alors en complexité linéaire.

2. Par une méthode dichotomique, écrire une fonction `succ_vect` de signature `int array -> int -> int` prenant en arguments un tableau `t` codant un ensemble E comme ci-dessus et un entier x et renvoyant le successeur de x dans E (-1 si cela n'existe pas.)

```
let succ_vect entiers x =
  if x >= entiers.(entiers.(0)) then -1 else
  if x< entiers.(1) then entiers.(1) else
    let rec aux entiers x deb fin =
      let mil = (deb+fin)/2 in
      if entiers.(mil) = x then entiers.(mil+1) else
      if entiers.(mil)<x then aux entiers x mil fin else aux entiers x deb mil
    in
    aux entiers x 1 entiers.(0)
;;
```

3. Calculer la complexité dans le pire cas de la fonction `succ_vect` en fonction de n.

A chaque appel récursif la taille de l'intervalle $[\![\texttt{deb};\texttt{fin}]\!]$ est divisée par 2. Cet intervalle étant de taille $|E|$, il faut au plus $\log(|E|)$ division avant de quitter la boucle, la fonction est donc de complexité logarithmique en la taille de l'ensemble.

4. Ecrire une fonction `union_vect` de signature `int array -> int array -> int array` prenant en arguments deux tableaux `t_1` et `t_2`, de taille N, codant deux ensembles E_1 et E_2 et renvoyant le tableau correspondant à $E_1 \cup E_2$. On supposera que $|E_1 \cup E_2| \leq N$.

```
let union_vect entiers1 entiers2 =
  (* les tailles des deux ensembles *)
  let t1 = entiers1.(0) in
  let t2 = entiers2.(0) in
  let union = Array.make (Array.length entiers1) 0 in
  let rec aux i i1 i2 =
    (* affecte union.(i) et renvoie la taille de l'union*)
    if i1>t1 && i2>t2 then i else
    if (i1>t1 || entiers2.(i2)<entiers1.(i1)) then (union.(i) <- entiers2.(i2);
    ↪  aux (i+1) i1 (i2+1)) else
    if (i2>t2 || entiers1.(i2)<entiers2.(i1)) then (union.(i) <- entiers1.(i1);
    ↪  aux (i+1) (i1+1) i2) else
      (union.(i) <- entiers1.(i1); aux (i+1) (i1+1) (i2+1))
  in
  let t = aux 1 1 1 in
  union.(0) <- (t-1);
  union
;;
```

■ **Partie III** : Avec une table de hachage

Soit K un entier naturel strictement positif. On choisit de représenter un ensemble d'entiers E de cardinal n par une table de hachage de taille K avec résolution des collisions par chainage. La fonction de hachage est $h(i) = i \mod K$.

1. Dans le cas où $K = 10$, représenter la table de hachage qui correspond à l'ensemble $\{2, 5, 7, 15\}$.

		2 -> null			5 -> 15 -> null		7 -> null		

2. A quelle condition, portant sur K et sur n, la fonction h génère-t-elle forcément des collisions ?

   Il y a K alveoles donc dès que $n > K$, on a forcément des collisions (principe des tiroirs)

3. Décrire brièvement (on ne demande pas d'écrire un programme) une fonction permettant de renvoyer le maximum d'un ensemble E représenté par une table de hachage. Donner sa complexité.

   On parcourt les listes chainées contenues dans chaque alvéole, en mettant à jour une variable contenant le maximum. On a donc une complexité en $O(|E|)$.

4. En OCaml, un ensemble est donc représenté par un tableau (de taille K) de listes d'entiers c'est à dire par le type `array int list`. Ecrire la fonction `appartient : int -> int array list -> bool` qui prend en arguments un entier et un ensemble représenté par une table de hachage et renvoie un booléen indiquant si cet élément appartient ou non à l'ensemble.

   Dans ce cas, on doit simplement regarder la tête de chaque liste, on a donc une complexité en $O(K)$.