

Devoir surveillé d'informatique

⚠ Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml suivant l'exercice. Dans le cas du C, on suppose que les bibliothèques standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.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 : Question de cours

1. Rappeler la définition d'un arbre binaire.
2. Rappeler les définitions de la hauteur et de la taille d'un arbre binaire.
3. Soit a un arbre binaire de taille n et de hauteur h . Prouver que $h + 1 \leq n \leq 2^{h+1} - 1$

□ Exercice 2 : Nombres de Hamming

Les programmes de cet exercice doivent être écrits en langage C.

■ Partie I : Implémentation d'une file en C

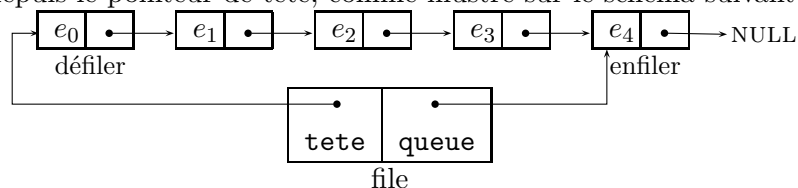
Dans cette partie, on implémente une file en C à l'aide d'une liste chaînée en conservant un pointeur sur le premier élément de la liste (à partir duquel on défile) et un autre pointeur sur le dernier élément de la liste (à partir duquel on enfiler). Pour cela on définit un type structuré `maillon` contenant une valeur et un pointeur sur le maillon suivant, et une file est alors implémentée comme deux pointeurs (un vers le début et l'autre vers la fin de la liste chaînée.) :

```

1  /* Implémentation d'une file avec accès au premier et au dernier*/
2  struct maillon
3  {
4      int valeur;
5      struct maillon *suivant;
6  };
7  typedef struct maillon maillon;
8  struct file
9  {
10     maillon *tete;
11     maillon *queue;
12 };
13 typedef struct file file;

```

⚠ On notera bien que dans cette implémentation, l'ajout d'un élément s'effectue sur le pointeur de queue et le retrait d'un élément depuis le pointeur de tête, comme illustré sur le schéma suivant :



La création d'une file vide s'effectue en initialisant les deux pointeurs à NULL :

```

1 file cree_file()
2 {
3     file f = {.tete = NULL, .queue = NULL};
4     return f;
5 }

```

1. Dans l'implémentation suivante de `defiler`, ajouter une instruction `assert` permettant de vérifier que la file n'est pas vide avant de défiler. Sans cette instruction, quel sera le comportement de `defiler` sur une file vide ?

```

1 int defiler(file *f)
2 {
3     int v = f->tete->valeur;
4     maillon *old = f->tete;
5     f->tete = f->tete->suivant;
6     return v;
7 }

```

2. Un problème subsiste dans l'implémentation proposée pour la fonction `defiler`, lequel et comment le corriger ?
3. Ecrire la fonction `prochain` qui renvoie le prochain élément qui sera défilé mais *sans le retirer de la file*. On traitera par un `assert` le cas de la file vide.

```

1 int prochain(file f)
2 {
3     assert (f.tete != NULL);
4     return (f.tete)->valeur;
5 }

```

4. Ecrire la fonction permettant d'enfiler un élément de prototype `void enfiler(file *f, int val)`

```

1 void enfiler(file *f, int val)
2 {
3     maillon *nm = malloc(sizeof(maillon));
4     nm->valeur = val;
5     nm->suivant = NULL;
6     if (f->queue == NULL)
7     {
8         f->queue = nm;
9     }
10    else
11    {
12        f->queue->suivant = nm;
13        f->queue = nm;
14    }
15    if (f->tete == NULL)
16    {
17        f->tete = nm;
18    }
19 }

```

■ Partie II : Nombre de Hamming

On appelle *nombre de Hamming* un entier naturel strictement positif n'ayant pas d'autres diviseurs premiers que 2, 3 et 5. Les premiers nombres de Hamming, par ordre croissant, sont donc 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, ...

1. Donner (sans justification) les six nombres de Hamming suivants.
2. Ecrire une fonction `est_hamming` de signature `bool est_hamming(int n)` permettant de tester si l'entier passé en paramètre est un nombre de Hamming.

```
1  bool est_hamming(int n)
2  {
3      if (n==1) {return true;}
4      while (n%2==0)
5      { n=n/2;}
6      while (n%3==0)
7      { n=n/3;}
8      while (n%5==0)
9      { n=n/5;}
10     return (n==1);
11 }
```

3. Donner, en la justifiant, la complexité de votre fonction `est_hamming`.

Pour générer les n premiers nombres de Hamming de *façon efficace* et par ordre croissant, on propose l'algorithme suivant :

- On initialise trois files `f2`, `f3` et `f5` contenant au départ un 1 ;
- A chaque étape :
 - On note $x = \min \{x_2, x_3, x_5\}$ où x_2 (resp. x_3 , resp. x_5) est le prochain élément à être défilé de `f2` (resp. `f3`, resp. `f5`)
 - x est le prochain nombre de Hamming, on l'enlève de la file qui le contient (il est forcément le prochain à être défilé)
 - On enfile $2x$ à `f2`, $3x$ à `f3` et $5x$ à `f5`

4. Simuler l'exécution de cet algorithme pour $n=8$ en donnant l'évolution des 3 files.
5. Implémenter cet algorithme en écrivant une fonction `hamming` de signature `int * hamming(int n)` qui renvoie le tableau des n premiers nombres de Hamming. On supposera déjà écrite la fonction `min3` qui renvoie le plus petit des trois entiers passés en paramètres. Votre fonction devra utiliser l'implémentation des files de la partie 1 et ses fonctions `enfiler` et `defiler`
6. Un même nombre peut être présent dans plusieurs des files. Ecrire une nouvelle version de la fonction `hamming` ne présentant plus ce problème.

```

1  int * hamming(int n)
2  {
3      file f2 = cree_file();
4      file f3 = cree_file();
5      file f5 = cree_file();
6      int x2, x3, x5, xm;
7      enfiler(&f2, 1);
8      enfiler(&f3, 1);
9      enfiler(&f5, 1);
10     int *ham = malloc(sizeof(int) * n);
11     for (int i = 0; i < n; i++)
12     {
13         x2 = prochain(f2);
14         x3 = prochain(f3);
15         x5 = prochain(f5);
16         if (x2 == min3(x2, x3, x5))
17         {
18             xm = defiler(&f2);
19         }
20         if (x3 == min3(x2, x3, x5))
21         {
22             xm = defiler(&f3);
23         }
24         if (x5 == min3(x2, x3, x5))
25         {
26             xm = defiler(&f5);
27         }
28         enfiler(&f2, 2 * xm);
29         enfiler(&f3, 3 * xm);
30         enfiler(&f5, 5 * xm);
31         ham[i] = xm;
32     }
33     return ham;
34 }

```

□ 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 chaînes 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.
2. Expliquer ce qu'affiche la requête suivante :

```

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

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

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

```
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.

```
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.

```
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
```

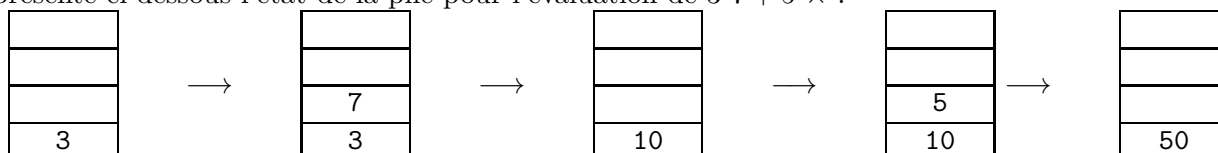
□ Exercice 4 : Piles et notation polonaise inverse

Les programmes de cet exercice doivent être écrits en OCaml.

La notation polonaise inverse (NPI) est une méthode d'écriture des expressions mathématiques qui n'utilise pas de parenthèses et qui de plus se calcule sans règles de priorité. Prenons un exemple, l'expression $(3 + 7) \times 5$, s'écrit en notation polonaise inversée : $3\ 7\ +\ 5\ \times$. C'est à dire qu'on donne *d'abord les deux opérandes puis l'opération*. Le but de l'exercice est d'écrire une fonction OCaml évaluant une expression en NPI passée en paramètre à l'aide d'une pile, on utilisera la méthode suivante :

- ① Parcourir l'expression de gauche à droite
- ② Si on rencontre un nombre l'empiler
- ③ Si on rencontre une opération effectuer cette opération entre les deux valeurs situées au dessus de la pile et empiler le résultat

On représente ci-dessous l'état de la pile pour l'évaluation de $3\ 7\ +\ 5\ \times$:



1. Ecrire l'expression $(4 + 3) \times (5 + 1)$ en NPI.
2. Donner les étapes de l'évaluation de cette expression en faisant figurer à chaque étape le contenu de la pile. Afin de représenter les éléments (c'est à dire soit des `int` soit une opération) d'une expression en NPI, on définit le type union suivant en OCaml :

```
1 type elt = Add | Sub | Mult | Div | Nombre of int
```

Ainsi, l'expression $3 + 5 \times$ est représentée par la liste `[Nombre 3; Nombre 5; Add; Nombre 5; Mult]`.

D'autre part, on utilise le module `Stack` de OCaml afin de disposer d'une structure de pile *mutable*. On rappelle ci-dessous les fonctions principales de ce module :

- `Stack.create` de signature `() -> 'a t` qui crée une pile vide d'éléments de type `'a`. Par exemple `let mapile = Stack.create ()`
- `Stack.push` de signature `'a 'a t -> ()` qui empile un élément. Par exemple `Stack.push 5 mapile` empile l'entier 5 sur `mapile` (le type option `'a` est alors le type `int`).
- `Stack.pop` de signature `'a t -> 'a` qui renvoie l'élément situé au sommet de la pile en le dépilant.

3. Ecrire une fonction `evaluate` de signature `elt list -> int` et qui renvoie l'évaluation d'une expression en npi donné sous la forme d'une liste de type `elt`. Par exemple, `evaluate [Nombre 3; Nombre 7; Sub; Nombre 5; Mult]` doit renvoyer 50.

```
1 let evaluate npi =
2   let rec aux npi pile =
3     match npi with
4     | [] -> Stack.pop pile
5     | h::t -> (match h with
6       | Add -> Stack.push (Stack.pop pile + Stack.pop pile) pile
7       | Sub -> Stack.push (Stack.pop pile - Stack.pop pile) pile
8       | Mult -> Stack.push (Stack.pop pile * Stack.pop pile) pile
9       | Div -> Stack.push (Stack.pop pile / Stack.pop pile) pile
10      | Nombre x -> Stack.push x pile );
11     aux t pile
12   in
13   let temp = Stack.create () in
14   aux npi temp;;
```

□ Exercice 5 : 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 implémente dans cet exercice des ensembles par des structures connues et, 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.

1. 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.).

```
1 let rec succ_list entiers x =
2   match entiers with
3   | [] -> -1
4   | h::t -> if h>x then h else succ_list t x
```

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

■ **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 $\{2, 5, 7\}$.

- 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)
- 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.)

```

1  let succ_vect entiers x =
2    if x >= entiers.(entiers.(0)) then -1 else (
3      let imin = ref 1 in
4      let imax = ref entiers.(0) in
5      let found = ref false in
6      let imid = ref 0 in
7      while (!imax - !imin >= 0 && not !found) do
8        imid := (!imax + !imin)/2;
9        if (entiers.(!imid)=x) then found:=true else
10         if (entiers.(!imid)<x) then imin := !imid + 1 else imax := !imid-1
11        done;
12        if (!found) then entiers.(!imid+1) else entiers.(!imin))
13  ;;

```

- Calculer la complexité dans le pire cas de la fonction `succ_vect` en fonction de n .
- 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 $|E_1 \cup E_2| \leq N$.

```

1  let union t1 t2 =
2      (* size est la variable N de l'énoncé*)
3      let t = Array.make size 0 in
4      let size1 = t1.(0) in
5      let size2 = t2.(0) in
6      let i1 = ref 1 in
7      let i2 = ref 1 in
8      let i = ref 1 in
9      while (!i1 <= size1 || !i2 <= size2) do
10         Printf.printf "i1 = %d, i2 = %d, i = %d\n" !i1 !i2 !i;
11         (* prendre dans t1 si t2 est vide OU si il reste des éléments dans les 2 et le
           ↳ plus petit est dans t1*)
12         if (!i2 > size2) || (!i1 <= size1 && !i2 <= size2 && t1.(!i1) <= t2.(!i2)) then
13             (
14                 t.(!i) <- t1.(!i1);
15                 (* en cas d'égalité on avance aussi dans t2*)
16                 if (!i2 <= size2 && t1.(!i1) = t2.(!i2)) then (i2 := !i2 + 1);
17                 i1 := !i1 + 1;
18             ) else
19             (* sinon prendre dans t2 *)
20             (
21                 t.(!i) <- t2.(!i2);
22                 (* en cas d'égalité on avance aussi dans t1*)
23                 if (!i1 <= size1 && t1.(!i1) = t2.(!i2)) then (i1 := !i1 + 1);
24                 i2 := !i2 + 1;
25             );
26             i := !i + 1;
27         done;
28         t.(0) <- !i-1;
29         t;;

```

■ 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 chaînage. La fonction de hachage est $h(i) = i \bmod K$.

1. Dans le cas où $K = 10$, représenter la table de hachage qui correspond à l'ensemble $\{2, 5, 7, 15\}$.
2. A quelle condition, portant sur K et sur n , la fonction h génère-t-elle forcément des collisions ?
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.
4. Peut-on améliorer la complexité de la fonction précédente si on suppose que les listes chaînées contenues dans chacune des alvéoles de la table de hachage sont maintenues triées par ordre décroissant ?