Devoir Surveillé 2

7 octobre 2023

INFORMATIQUE MPI*

DURÉE DE L'ÉPREUVE : **4 heures**

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Ce sujet comporte huit pages numérotées de 1/18 à 18/18

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 4 parties indépendantes, utilisant les langages de programmation C et OCaml ainsi que du pseudo-code.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

- La partie I contient des démonstrations sur des propriétés de connexité dans les graphes.
- La partie II s'intéresse aux arbres couvrants, et à l'algorithme de Prim, en pseudo-code.
- La partie III s'intéresse aux tas de Fibonacci, une structure de données permettant d'implémenter efficacement des files de priorités dans lesquelles on peut modifier la clé associée à une valeur, à leur implémentation en C et à leur étude.
- La partie IV s'intéresse à l'utilisation de l'algorithme A* pour la résolution du jeu de taquin et son implémentation en OCaml.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

Partie I. Connexité – 10 pts

☐ 1 – Soit G un graphe à $2n$ sommets, de degré minimal supérieur ou égal à n. Montrer que G est connexe.
Indication : *On pourra s'intéresser à la plus petite composante connexe.*

Réponse 1. Supposons que $G = (S, A)$ possède au moins deux composantes connexes et notons C la plus petite d'entre elles. $|C| \leq \frac{|S|}{2} = n$. Donc le degré maximal d'un sommet de C est $n-1$, on a une contradiction.

Pour un graphe G, on note $\bar{G}$ le graphe ayant le même ensemble de sommet de G tel que pour tout couple de sommets (x, y), x et y sont adjacents dans $\bar{G}$ si et seulement si ils ne sont pas adjacents dans G.

☐ 2 – Soit G un graphe ayant au moins 5 sommets. Montrer qu'au moins l'un des deux graphes G ou $\bar{G}$ admet un cycle.
Indication : *On pourra compter le nombre d'arêtes.*

Réponse 2. Notons n le nombre de sommets de G et m son nombre d'arêtes. $\bar{G}$ possède $m' = \frac{n(n+1)}{2} - m$ arêtes. Supposons que $m < n-1$ et $m' < n-1$, alors $m+m' = \frac{n(n+1)}{2} < 2n-2$ avec $n+1 > 5$ donc $5n < 4n-4$, ce qui est absurde. Nécessairement, on a $m \geq n-1$ ou $m' \geq n-1$, on un graphe sur n sommets qui possède au moins n arêtes possède un cycle.

Un sommet x d'un graphe non orienté connexe G est dit *point d'articulation* de G si $G - x$ est non connexe.

☐ 3 – Montrer que tout graphe connexe contient au moins deux sommets qui ne sont pas des points d'articulation.
Indication : *On pourra considérer un chemin élémentaire maximal, c'est-à-dire un chemin qui ne contient pas deux fois le même sommet, et qui n'est sous-chemin d'aucun chemin élémentaire.*

Réponse 3. Soit $s_0 - s_1 - \ldots - s_k$ un chemin élémentaire. s_0 ne peut pas être un point d'articulation, car tous ses voisins sont dans $\{s_i \mid i \in [\![1; k]\!]\}$ (par maximalité du chemin élémentaire). Il en est de même pour s_k.

☐ 4 – On définit inductivement une classe de graphes non orientés $\mathcal{C}$ par :

1. tout sommet isolé est dans $\mathcal{C}$;
2. si $G \in \mathcal{C}$, tout graphe H obtenu en ajoutant à G un sommet x et une ou plusieurs arêtes adjacentes à x est dans $\mathcal{C}$.

Montrer que $\mathcal{C}$ est exactement la classe des graphes connexes.
Indication : *On pourra utiliser le résultat de la question précédente.*

Réponse 4. Il est rapide de vérifier que $\mathcal{C}$ est inclus dans la classe des graphes connexes :

- tout sommet isolé est un graphe connexe ;
- si on ajoute un sommet à un graphe connexe et qu'on lui ajoute une ou plusieurs arêtes adjacentes, on obtient un graphe connexe.

Dans l'autre direction, on montre par récurrence que tout n, tout graphe connexe sur n sommets est dans $\mathcal{C}$.

- Le cas $n = 1$ est direct ;
- Soit G un graphe connexe sur $n + 1$ sommets. Il admet au moins un sommet x qui n'est pas un point d'articulation. $G - x$ est connexe sur n sommets donc il appartient à $\mathcal{C}$, et G aussi.

Partie II. Arbres couvrants – 20 pts

On considère la variante VK suivante de l'algorithme de Kruskal sur un graphe non orienté connexe pondéré $G = (S, A)$:

```
[Variante de l'algorithme de Kruskal : VK]
T = A
trier les arêtes de A dans l'ordre décroissant de leur poids
Pour chaque arête a :
    si T \ {a} est connexe, alors T = T \ {a}
renvoyer T
```

☐ 5 – Proposer un algorithme permettant de déterminer si un graphe $G = (S, A)$ est connexe en temps $O(|A|)$ dans le pire des cas. On supposera que le graphe est représenté par listes d'adjacence.

Réponse 5. On peut effectuer un parcours en profondeur ou un parcours en largeur à partir d'un sommet s quelconque en marquant comme vus les sommets rencontrés et vérifier si tous les sommets sont vus.

☐ 6 – Quelle est la complexité de l'algorithme VK ?

Réponse 6. Si on utilise l'algorithme proposé à la question précédente, on effectur une opération de complexité $O(|A|)$ pour chaque arête du graphe. On a donc une complexité $O(|A|^2)$.

☐ 7 – Montrer que VK renvoie toujours un arbre couvrant.

Réponse 7.

- La connexité est directe par définition de l'algorithme. On a l'invariant suivant : (S, T) est un sous-graphe connexe.
- Supposons que l'ensemble d'arêtes T renvoyé par l'algorithme contiennt un cycle C. Alors n'importe quelle arête de C aurait pu être retiré en préservant la connexité, ce qui est absurde car aucune ne l'a été.

☐ 8 – Cet arbre couvrant est-il toujours de poids minimal ? justifier votre réponse.

Réponse 8. Il est toujours de poids minimal. En effet, on peut montrer l'invariant suivant :
Il existe un arbre couvrant de poids minimal qui ne contient aucune arête de $A \setminus T$.
Au départ, $A \setminus T = \emptyset$.
Pour une itération considérant l'arête a : si il existe un arbre couvrant F ne contenant aucune arête de $A \setminus T$, on a deux possibilités :

- Si $a \notin F$ ou $T \setminus \{a\}$ n'est pas connexe, F convient à l'étape suivante.
- Sinon, a est sur un cycle C de T (sinon, $T \setminus \{a\}$ ne serait pas connexe), et est de poids maximal sur ce cycle (sinon, l'autre arête du cycle aurait été retiré avant a). Soit b une autre arête de ce cycle. $F \setminus \{a\} \cup \{b\}$ est de poids au plus égal à celui de F et convient à l'étape suivante.

Dans un graphe non orienté $G = (S, A)$. Si P est une partie non triviale de S ($P \neq \emptyset$ et $P \neq S$), le couple $(P, S \setminus P)$ est appelé *coupure* du graphe et noté $(P, \neg P)$. On dit qu'une arête $x - y$ *traverse* la coupure $(P, \neg P)$ si $x \in P$ et $y \in \neg P$ ou $x \in \neg P$ et $y \in P$.

☐ 9 – Montrer que si une arête est sur un cycle $\mathcal{C}$ et traverse une coupure $(P, \neg P)$, alors il existe une autre arête de $\mathcal{C}$ traversant cette même coupure.

Réponse 9. Soit $s_0 - s_1 - \ldots - s_k$ le cycle $\mathcal{C}$, avec $s_0 \in P$ et $s_k \notin P$, c'est-à-dire que (s_0, s_k) traverse la coupure $(P, \neg P)$. Soit i le plus petit entier tel que $s_i \in P \wedge s_{i+1} \notin P$ (si cet ensemble est vide, on montre par récurrence que tout les s_i sont dans P, ce qui est absurde car s_k n'y est pas). Alors (s_i, s_{i+1}) traverse également $(P, \neg P)$.

Dans un graphe non orienté pondéré connexe $G = (S, A)$, pour n'importe quel sommet s, l'algorithme de Prim se base sur ce principe pour construire un arbre couvrant de poids minimal enraciné [1] en s.

1. En théorie des graphes, un arbre enraciné et la donné d'un sous-graphe connexe acyclique (un arbre), et d'un sommet particulier qui est la racine de l'arbre. On retrouve alors les propriétés rencontrés avec la définition des arbres d'arité quelconque par induction.

```
[Algorithme de Prim]
P = {s}, T = {}

Tant que P est différent de S :
  trouver une arête minimale (s1,s2) pour la traversée de (P,S\ P)
  P = P U {s1,s2}
  T = T U {{s1,s2}}
```

☐ 10 – Quelle structure de données abstraite pourrait être utilisée pour trouver efficacement l'arête (s_1, s_2) ? Justifier

Réponse 10. Une file de priorité peut être utilisé pour stocker les sommets de $S \setminus P$ qui sont adjacents à un sommet de P, avec comme priorité le poids minimal d'une arête le reliant à un sommet de P. On peut même directement y stocker l'arête qui possède ce poids minimal, ou alors stocker le sommet parent dans un tableau à part. Dans tous les cas, cette structure de données est adaptée pour trouver l'arête de poids et donc de priorité minimale.

☐ 11 – Proposer un invariant qui permet de montrer la correction de l'algorithme de Prim et montrer qu'il est vérifié.

Réponse 11.

Il existe un arbre couvrant de poids minimal contenant T.

— Au départ, $T = \emptyset$
— Pour une itération donnée, soit F un arbre couvrant minimal contenant T. Considérons la coupure $(P, \neg P)$. L'arête a qui est ajoutée à T est minimale pour la coupure $(P, \neg P)$. Si $a \notin F$, alors $F \cup \{a\}$ contient un cycle C. Il existe donc une autre arête b du cycle C qui traverse la coupure $(P, \neg P)$, et $F \setminus \{b\} \cup \{a\}$ est un arbre couvrant de poids inférieur ou égal à celui de F et convient pour l'étape suivante.

Partie III. Tas de Fibonacci (C) – 40 pts

Cette partie traite de la construction de tas de Fibonacci, une structure de données implémentant les files de priorité qui permet de diminuer efficacement la clé d'une valeur donnée. Cette structure de donnée est fortement liée aux listes circulaires doublement chaînées.
Commençons par des listes doublement chaînées. Voici un exemple de structure en C qui définit des listes doublement chaînées. Chaque maillon possède un pointeur vers le maillon suivant, et un pointeur vers le maillon précédent. Le maillon de tête possède un pointeur `NULL` comme maillon précédent et le maillon de queue possède un pointeur `NULL` comme maillon suivant.

Doubly Linked Lists — C

```c
struct maillon {
  int value;
  struct maillon *next;
  struct maillon *prev;
};
typedef struct maillon* dll;
```

Un avantage des listes doublement chaînées est qu'il est aisé d'ajouter ou de retirer un élément de la liste sans avoir à la parcourir.
Voici un exemple de fonction pour ajouter un élément v juste avant le maillon pointé par ℓ :

C

```c
void insert(dll l, int v){
  struct maillon *m = malloc(sizeof(struct maillon));
  assert(m != NULL);
  m->value = v;
  m->next = l;
  l->prev->next = m;
  m->prev = l->prev;
  l->prev = m;
}
```

☐ 12 – Que fait la ligne `assert(m != NULL);` ?

Réponse 12. Elle vérifie que la mémoire demandée a bien été allouée et que le pointeur `m` n'est pas `NULL`. Si ce n'est pas le cas, le programme renvoie une erreur.

☐ 13 – Dans quels cas cette fonction peut-elle renvoyer une erreur ? Proposer une version corrigée de cette fonction.

Réponse 13. On peut avoir une erreur si on essaye de déréférencer un pointeur `NULL`, c'est-à-dire si `l`, ou `l->prev` vaut `NULL`. On peut aussi remarquer qu'on ne peut pas insérer d'élément dans une liste vide, il faudrait renvoyer une nouvelle liste pour gérer ce cas.

```c
dll insert(dll l, int v){
  struct maillon *m = malloc(sizeof(struct maillon));
  assert(m != NULL);
  m->value = v;
  m->next = l;
  if (l == NULL) {return m;}
  if (l->prev != NULL) {l->prev->next = m;}
  m->prev = l->prev;
  l->prev = m;
  return l; // ou return m;
}
```

☐ 14 – Avec une liste doublement chaînée, on peut accéder à tous les éléments de la liste à partir de n'importe quel maillon, même si ce n'est pas celui de tête.
Écrire une fonction `void print_dll(dll l)` qui affiche toutes les valeurs présentes dans la liste.

Réponse 14.

```c
void print_dll(dll l){
  if (l == NULL) { return; }
    while (l->prev != NULL) { l = l->prev; }
    while (l->next != NULL) {
    print_int(l->value);
    l = l->next;
  }
}
```

☐ 15 – Écrire une fonction `void suppr_maillon(dll l)` qui supprime le maillon pointé par ℓ de la liste dans laquelle il se trouve. On prendra soin de libérer la mémoire.

Réponse 15.

```c
void suppr_maillon(dll l){
  if (l == NULL) { return; }
  if (l->prev != NULL) {l->prev->next = l->next; }
  if (l->next != NULL) {l->next->prev = l->prev; }
  free(l);
}
```

Le parcours d'une liste doublement chaînée à partir d'un maillon quelconque peut nécessité dans le pire des cas de parcourir entièrement deux fois la liste ou d'utiliser une variable auxiliaire. Pour rendre plus efficace ce parcours, on peut rendre la liste circulaire : c'est-à-dire que le champ `prev` du maillon de tête pointe maintenant vers le maillon de queue, et le champ `next` du maillon de queue pointe vers le maillon de tête.

☐ 16 – Une liste circulaire doublement chaînée doit toujours vérifier l'invariant suivant :

Pour tout maillon m, le maillon précédent le maillon suivant m doit être le maillon m, et le maillon suivant le maillon précédent m doit être m.

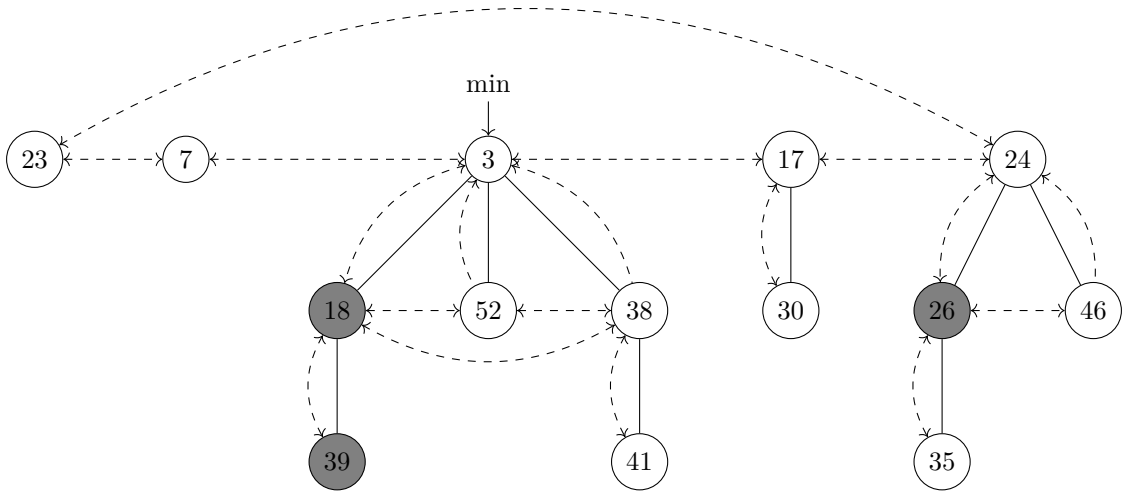
Écrire une fonction `void check(dll l)` qui vérifie cet invariant et renvoie une erreur si il n'est pas vérifié.

Réponse 16.

C

```c
void check(dll l) {
  if (l == NULL) { return; }
  dll tmp = l->next;
  while (tmp != l) {
    assert(tmp->next->prev == tmp && tmp->prev->next == tmp);
  }
  assert(l->next->prev == l && l->prev->next == l);
}
```

Un tas de Fibonacci est une structure de données constituée d'un ensemble d'arbres ayant la propriété de tas. Les racines de ces arbres sont stockés dans une liste circulaire doublement chaînée, et on y accède par un pointeur vers le nœud dont la clé est minimale. Voici un exemple de représentation graphique d'un tas de Fibonacci. Le marquage des nœuds (grisés ou non grisés) n'interviendra que pour décroître une clé et ne sera donc pas utile pour le moment. Les nombres inscrits dans les nœuds sont les clés. Les traits pleins représentent les liens au sein des arbres, alors que les pointillés représentent les pointeurs utilisés dans la structure C donnée sous la figure.



Exemple nř1

Fibonacci Heaps C

```c
struct node{
  int key;
  int deg;
  bool mark;
  struct node* left;
  struct node* right;
  struct node* son;
  struct node* father;
};
typedef struct node* Fib_heap;
```

Par souci de clarté, le champ `value` qui serait nécessaire à l'implémentation concrète d'une file de priorité est omis. Comme pour l'implémentation d'arbres d'arités quelconques, chaque nœud pointe vers l'un de ses fils (`son`), on peut accéder aux autres en parcourant la liste circulaire doublement chaînée à laquelle il appartient grâce à ses champs `left` et `right`. Le champ `son` d'une feuille vaut `NULL`. Une des fonctions nécessitant l'accès au père d'un nœud, on a ajouté un champ `father` qui désigne le parent du nœud, ou `NULL` si celui-ci est une racine. Le champ `deg` correspond au degré du nœud, et servira à la consolidation du tas, pour éviter que la liste des racines devienne trop longue. Dans la suite du sujet, on utilisera le type `Fib_heap` pour désigner uniquement des pointeurs vers des nœuds dont la clé est minimale. Les pointeurs vers d'autre nœuds seront notés `struct node*`.

☐ 17 – Rappeler la propriété de tas.

Réponse 17. Un arbre vérifie la propriété de tas si chaque nœud (excepté la racine) possède une clé plus grande que celle de son parent.

☐ 18 – Écrire une fonction `Fib_heap Fib_new(int k)` qui créé un nouveau nœud dont la clé est l'entier k.

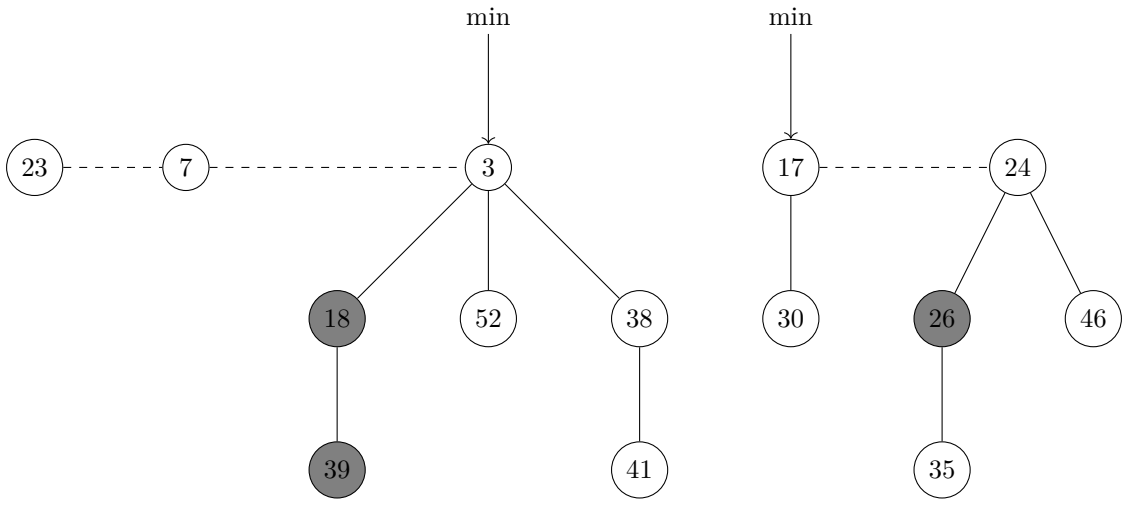
Réponse 18.

C
```c
Fib_heap Fib_new(int k){
  Fib_heap f = malloc(sizeof(struct node));
  f->key = k; f->deg = 0; f->mark = false;
  f->left = f; f->right = f;
  f->son = NULL; f-> father = NULL;
  return f;
}
```

☐ 19 – Écrire une fonction

`Fib_heap Fib_union(Fib_heap f1, Fib_heap f2)`

qui renvoie l'union des deux tas f_1 et f_2, c'est-à-dire le tas dont la liste des racines est la concaténation des listes des racines de f_1 et f_2.



Exemple nř2 : l'union de ces tas donne l'exemple nř1

Réponse 19.

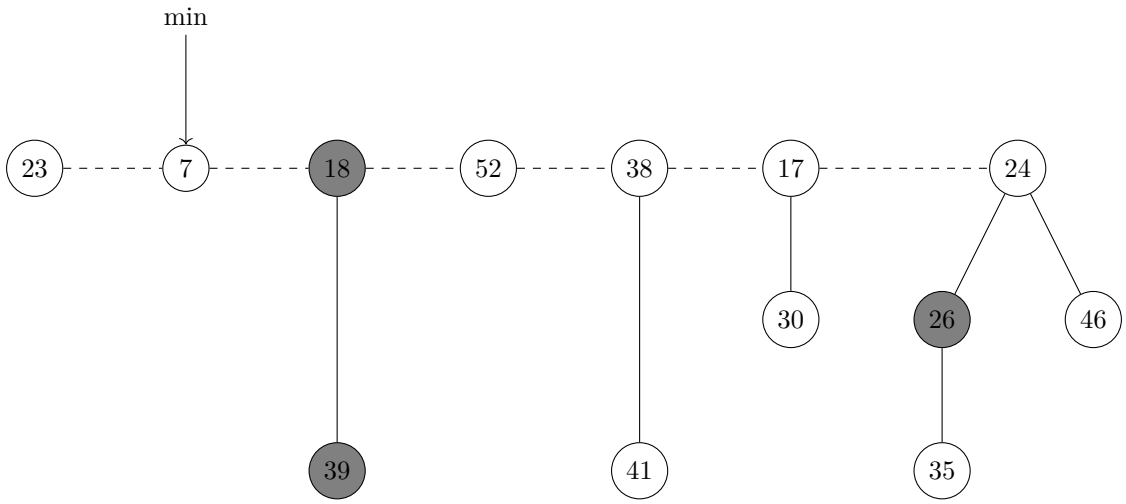
C

```
Fib_heap Fib_union(Fib_heap f1, Fib_heap f2){
  if (f1 == NULL) { return f2; }
  if (f2 == NULL) { return f1; }
  f1->right->left = f2->left;
  f2->left->right = f1->right;
  f1->right = f2;
  f2->left = f1;
  if (f2->key < f1->key) { return f2; }
  return f1;
}
```

Pour extraire le minimum d'un tas, on procède en deux étapes :

1. Enlever le minimum de la liste des racines, et rajouter ses fils comme nouvelles racines.
2. Consolider le tas : Tant qu'il y a des racines de même degré, regrouper les deux arbres, et augmenter le degré de la nouvelle racine.
   Cette étape n'est pas nécessaire pour extraire le minimum, mais c'est elle qui garantie la complexité amortie des tas de Fibonacci en réduisant la taille de la liste circulaire doublement chaînée.

☐ 20 – Écrire une fonction `Fib_heap Fib_remove_min(Fib_heap f)` qui enlève le minimum du tas et place ses fils dans la liste des racines.



Exemple nř3 : résultat de remove_min sur l'exemple nř1

Réponse 20.

C

```c
void Fib_remove_father(struct node *n){
  if (n == NULL) { return n; }
  struct node *tmp = n->right;
  while (tmp != n) {
    tmp->father = NULL;
  }
  n->father = NULL;
}

Fib_heap Fib_remove_min(Fib_heap f){
  struct node *n = f->son;
  Fib_remove_father(n);
  if (f->right != f) { // f possède au moins 2 éléments
    f->right->left = f->left;
    f->left->right = f->right;
    n = Fib_union(f->right, n);
    // Attention, le pointeur renvoyé peut ne pas être le minimum
  }
  struct node *min = n;
  struct node *tmp = n->right;
  while (tmp != n) {
    if (tmp->key < min->key) { min = tmp; }
    tmp = tmp->right;
  }
  return min;
}
```

Pour repérer les arbres qui ont le même degré, nous allons créer un tableau contenant pour chaque indice i un pointeur vers un nœud de degré i si il en existe un ou `NULL` sinon.

☐ 21 – Écrire une fonction `int Fib_max_degree(Fib_heap f)` qui renvoie le plus grand degré parmi les nœuds dans la liste des racines.

Réponse 21.

C
```c
int Fib_max_degree(Fib_heap f){
  assert (f != NULL);
  struct node *tmp = f->right;
  int max = f->deg;
  while (tmp != f){
  ^^Iif (tmp->deg > max) { max = tmp->deg;}
  ^^Itmp = tmp->right;
  }
  return max;
}
```

☐ 22 – Écrire une fonction `void Fib_consolidate(Fib_heap f)` qui consolide le tas. Chaque fois que l'on trouve deux nœuds n_1 et n_2 de même degré, on place n_2 comme fils de n_1 et on augmente le degré de n_1 de 1 (ou vice-versa).

Réponse 22.

C
```c
void Fib_consolidate(Fib_heap f){
  if (f == NULL) { return; }
  int d = Fib_max_degree(f);
  struct node **tab = malloc(d * sizeof(struct node*));
  for (int i = 0; i < d; i = i + 1) { tab[i] = NULL; }
  tab[f->deg] = f;
  struct node *tmp = f->right;
  while (tmp != f){
    struct node *n = tab[tmp->deg];
    if (n == NULL){
      tab[tmp->deg] = tmp;
      tmp = tmp->right;
    } else {
      tmp->son = Fib_union(n, tmp->son);
      n->father = tmp;
      tab[tmp->deg] = NULL;
      tmp->deg = tmp->deg + 1;
    }
  }
  free(tab);
}
```

Pour décroître la clé d'un nœud, on effectue directement le changement, puis on met à jour la structure si la nouvelle clé du nœud est plus petite que la clé de son père en coupant ce nœud : c'est-à-dire qu'on le place en tant que racine d'un nouvel arbre, on enlève sa marque si il est marqué, et on marque son père. Si son père était déjà marqué, on le coupe à son tour (il peut y avoir un effet de cascade qui coupe plusieurs nœuds).

☐ 23 – Écrire une fonction récursive `void Fib_cut(Fib_heap f, struct node* n)` qui coupe le nœud n dans le tas f.

Réponse 23.

```C
void Fib_cut(Fib_heap f, struct node* n){
  struct node *p = n->father;
  if (p == NULL) { return; }
  if (p->son == n && n->right != n)
  ^^I{ p->son = n->right; }
  n->father = NULL; n->mark = false;
  n->right->left = n->left;
  n->left->right = n->right;
  n->left = n; n->right = n;
  Fib_union(f, n); // La valeur de retour est nécessairement f, on peut donc l'ignorer
  p->deg = p->deg - 1;
  if (p->mark) { Fib_cut(f, p); }
  else { p->mark = true; }
}
```

☐ 24 – Écrire une fonction

```
Fib_heap Fib_decrease_key(Fib_heap f, struct node *n, int k)
```

qui décroît la clé du nœud n pour la mettre à k. On supposera sans le vérifier que le nœud n appartient bien au tas f.

Réponse 24.

```C
Fib_heap Fib_decrease_key(Fib_heap f, struct node *n, int k){
  n->key = k;
  if (k < n->father->key) { Fib_cut(f, n); }
  if (k < f->key) { return n; }
  return f;
}
```

Analysons l'efficacité de cette structure de données.

On pose F_n le n-ième terme de la suite de Fibonacci. On a pour tout $n \in \mathbb{N}$:

— $F_{n+2} \geq \varphi^n$, avec $\varphi = \frac{1+\sqrt{5}}{2}$;

— $F_{n+2} = 1 + \sum_{i=0}^{n} F_i$.

☐ 25 – Montrer que le degré d'un tas de Fibonacci est au plus logarithmique en la taille du tas.
Indication : On pourra montrer qu'un tas de degré k possède au moins φ^k éléments.

Réponse 25. Le degré d'un nœud est le nombre de fils qu'il possède.
On montre par induction qu'un nœud de degré d est racine d'un sous-arbre d'au moins F_{d+2} éléments :
au départ, un tas est de degré 0 et possède 1 élément ;
étant donné un nœud n de degré d, il possède d fils $m_1, \ldots, m_d$. Au moment ou son fils m_i a été ajouté, il avait pour degré $i-1$ car on ne peut ajouter comme fils qu'un nœud de même degré. Depuis, il a perdu au plus un fils (grâce au marquage des nœuds), donc son degré est au moins $i-2$. Par induction, on a donc le nœud n qui est racine d'un arbre qui possède au moins $1 + \sum_{i=1}^{d} F_{(i-2)+2}$ soit F_{d+2} éléments.

La complexité amortie des tas de Fibonacci $\mathcal{F}$ est calculée en prenant en compte le potentiel $p(\mathcal{F})$ du tas,

$$p(\mathcal{F}) = a(\mathcal{F}) + 2m(\mathcal{F})$$

où $a(\mathcal{F})$ est le nombre d'arbres dans $\mathcal{F}$ et $m(\mathcal{F})$ le nombre de sommets marqués dans $\mathcal{F}$.
L'idée est d'augmenter artificiellement le coût des opérations les plus simples, et de stocker ce coût en tant que potentiel, pour le déduire du coût des opérations les plus complexes.
La complexité amortie d'une opération est définie comme la complexité de l'opération, à laquelle on ajoute la différence de potentiel entre le tas d'arrivé et le tas de départ. (cette différence peut être négative)

☐ 26 – Analyser l'évolution de $p(\mathcal{F})$ lors des opérations définies précédemment : union, extraction du minimum, décroissement d'une clé.

Réponse 26. $p(\mathcal{F}_1 \cup \mathcal{F}_2) = p(\mathcal{F}_1) + p(\mathcal{F}_2)$
Lors de l'extraction du minimum, le potentiel du tas augmente de $d-1$ où d est le degré du nœud dont la clé est minimale.
Lorsqu'on l'on décroît une clé d'un nœud n : Si la nouvelle clé n'est pas plus petite que celle du parent de n, le potentiel ne change pas.
Sinon, si on note k le nombre de nœuds coupés : au moins $k-1$ nœuds perdent leur marque. donc le potentiel décroît d'au moins $2(k-1) - k = k - 2$.

☐ 27 – Pour chacune des opérations, donner sa complexité amortie : temps d'exécution + différence de potentiel (nouveau potentiel - ancien potentiel, pouvant être négatif). On se contentera d'une notation asymptotique.

Réponse 27.

- Union : $O(1)$ pour le temps d'exécution, pas de changement de potentiel. La complexité amortie est constante ;
- Extraction du min : Le temps d'exécution est $O(d)$ et le potentiel augmente de $d-1$, où d est le nombre de fils du nœud de clé minimale. $d = O(\log n)$ donc la complexité amortie est logarithmique en la taille du tas ;
- Décroître une clé : Le changement de potentiel est $-k+2$ et la complexité est $O(k)$. Donc la complexité amortie est constante.

Partie IV. Taquin (OCaml) – 30 pts

Le jeu du taquin consiste à déplacer des tuiles adjacentes à la case vide jusqu'à arriver à mettre tous les nombres dans l'ordre. Voici un exemple de positions pour un taquin de dimension 3 :

3	1	2
7	5	
4	6	8

Position initiale

	1	2
3	4	5
6	7	8

Position finale

On représente une position de taquin en OCaml par un `int array`, en représentant la case vide par 0, et on généralise le problème à une grille de dimensions $n \times n$. La case numéro i représente le nombre en position $(i/n, i \bmod n)$. Une position est dite *valide* si chaque nombre entre 0 et $n^2 - 1$ apparaît une et une seule fois dans le tableau. La position gagnante correspond à une position valide triée.

☐ 28 – Écrire une fonction `est_finale : int_array -> bool` qui détermine si une position correspond à la position finale.

Réponse 28.

```
let est_finale pos =
  let flag = ref true in
  for i = 0 to Array.length pos - 1 do
    if pos.(i) <> i then flag := false
  done; !flag
```

☐ 29 – Écrire une fonction `case_vide : int array -> int` qui renvoie l'indice de la case vide.

Réponse 29.

```
let case_vide pos =
  let nn = Array.length pos in
  let rec loop i =
    if i >= nn then raise Not_found
    else if pos.(i) = 0 then i
    else loop (i+1)
  in loop 0
```

☐ 30 – Écrire une fonction `est_valide : int array -> bool` qui détermine si une position est valide ou non et qui possède une complexité dans le pire des cas $O(n)$, où n est la taille du tableau.

Réponse 30.

```
let est_valide pos =
  let nn = Array.length pos in
  let effectif = Array.make nn 0 in
  for i = 0 to nn - 1 do
    effectif.(pos.(i)) <- effectif.(pos.(i)) + 1
  done;
  let flag = ref true in
  for i = 0 to nn - 1 do
    if effectif.(i) <> 1 then flag := false
  done;
  !flag
```

☐ 31 – Écrire une fonction `distance : int -> int -> int` qui à partir de deux nombres i et j calcule le nombre de déplacements horizontaux ou verticaux nécessaires pour aller de la position i à la position j.

Réponse 31. On suppose l'existence d'une variable globale n donnant la dimension de la grille (les tableaux sont de taille $n \times n$).

```
(* fonction existante en OCaml, recopiée ici par souci de complétion *)
let abs x = if x < 0 then -x else x

let distance i j =
  let xi, yi = i mod n, i / n in
  let xj, yj = j mod n, j / n in
  abs (xi - xj) + abs (yi - yj)
```

On cherche maintenant à trouver la solution la plus courte au problème.

☐ 32 – Soit h la fonction qui à partir d'une position, donne le nombre de cases qui sont mal placées. Montrer que la fonction h est une heuristique admissible.

Réponse 32. Erreur dans l'énoncé (corrigée ici) : h donne le nombre de tuiles mal placées (sans compter la case vide), pas le nombre de cases bien placées.
$h(p) = 0$ si p est la position finale. Chaque coup déplace une seule tuile, donc le nombre de coups à jouer pour terminer est nécessairement supérieur ou égal au nombre de tuiles mal placées.

☐ 33 – Proposer une autre heuristique admissible f qui domine l'heuristique h, c'est-à-dire que pour toute position x, $f(x) \geq h(x)$.

Réponse 33. Si la tuile i se trouve en position j, il faut au moins `distance i j` déplacements pour amener cette tuile à sa place. On peut donc choisir comme heuristique f la somme des `distance i j` pour toutes les tuiles i. On remarque que $f(x) \geq h(x)$ car si `distance i j` vaut 0, alors la tuile est bien placée.

☐ 34 – Écrire une fonction `heuristique : int array -> int` qui calcule l'heuristique f.

Réponse 34.

```
let heuristique pos =
  let nn = Array.length pos in
  assert (nn = n * n);
  (* On pourrait recalculer n mais ce n'est
     pas la partie intéressante de l'exercice *)
  let dist = ref 0 in
  for i = 0 to nn - 1 do
    dist := !dist + distance i pos.(i)
  done; !dist
```

☐ 35 – Écrire une fonction `voisins : int array -> int array list` qui à partir d'une position du taquin, renvoie la liste des positions atteignables à partir de celle-ci en un déplacement. On prendra soin de créer des copies du tableau initial à l'aide de la fonction `Array.copy : int array -> int array`.

Réponse 35.

```
let voisins pos =
  let nn = Array.length pos in
  let z = case_vide pos in
  let res = ref [] in
  let copy_and_change i j =
    let tab = Array.copy pos in
    tab.(i) <- pos.(j); tab.(j) <- pos.(i); tab
  in
  if z > 0 then res := copy_and_change z (z-1)::!res;
  if z < nn-1 then res := copy_and_change z (z+1)::!res;
  if z >= n then res := copy_and_change z (z-n)::!res;
  if z < nn-n then res := copy_and_change z (z+n)::!res;
  !res
```

On suppose disposer d'une file de priorité de type `('a, 'b) file_prio` disposant des fonctions suivantes :

- `creer_FP : unit -> ('a, 'b) file_prio` : ne prend pas d'argument et crée une file de priorité vide ;
- `ajouter_FP : ('a, 'b) file_prio -> 'a -> 'b -> unit` : prend en argument une file de priorité, un élément et une priorité et ajoute l'élément dans la file de priorité avec la priorité donnée, ou met à jour sa priorité si l'élément est déjà dans la file ;
- `extraire_FP : ('a, 'b) file_prio -> 'a` : prend en argument une file de priorité et extrait l'élément de priorité maximale et renvoie sa valeur.

☐ 36 – Écrire une fonction `solve : int array -> int array list` qui à partir d'une position du taquin, renvoie la liste des positions successives permettant de résoudre le problème, en utilisant l'algorithme A*.

Réponse 36.

```
(* Il faut une structure de données persistante pour servir de clé dans une table de hachage *)
let cle pos = Array.to_list pos

let solve pos =
  let nn = Array.length pos in
  let dist = Hashtbl.create 1 in
  let parent = Hashtbl.create 1 in
  let f = creer_FP () in
  let rec ajoute_liste f d p = function
    | [] -> ()
    | t::q -> ajouter_FP f (t,d,p) (d+heuristique t); ajoute_liste f d p q
  and traite (x,d,p) =
    if not (Hashtbl.mem parent (cle x)) then
      begin
        Hashtbl.add dist (cle x) d;
        Hashtbl.add parent (cle x) p;
        if est_finale x then raise Exit;
        ajoute_liste f (d+1) (Some x) (voisins x)
      end
  and chemin x = match Hashtbl.find parent (cle x) with
    | None -> [x]
    | Some p -> x::chemin p
  in
  ajouter_FP f (pos, 0, None) 0;
  try
    while true do
      traite (extraire_FP f)
    done; assert false
  with
    Exit -> List.rev (chemin (Array.init nn (fun x -> x)))
```

Fin de l'épreuve