Algorithmique III (I41) - Licence d'Informatique

Contrôle terminal (Session 1 - Mai 2023)

Le sujet précisait que les explications, commentaires et justifications des résultats étaient exigés. L'absence de brèves explications préalables en français et de commentaires pour les algorithmes était sanctionnée par la perte de la moitié des points accordés pour leur écriture. L'absence de justification à un résultat non trivial était sanctionnée par 0.

EXERCICE 1. (Recherche d'un mot) Soit Σ un alphabet fini et $(\Sigma^*, \cdot)$ le monoïde constitué de l'ensemble des *mots* Σ^* définis sur l'alphabet Σ et de la loi $\cdot$ de concaténation. Si $u = u_1u_2 \dots u_n$ est un mot de Σ^*, l'entier n est sa *longueur*, on la note $|u|$. L'élément neutre du monoïde est le *mot vide* ε de longueur 0. Un couple constitué d'un mot u non-vide et d'un entier p est une *portion* d'un mot v si et seulement s'il existe deux mots x et y tels que $v = x.u.y$, auquel cas p est la *position* de u dans v.

Exemples : $(p, 3)$, $(re, 1)$, $(érer, 4)$ et $(re, 5)$ sont des portions du mot *repérer*, entre autres.

Soit $v \in \Sigma^*$. On définit une relation binaire $\shortparallel$ sur l'ensemble $P(v)$ des portions de v par

$$(u, p) \shortparallel (u', q) \;\Leftrightarrow\; |u| = |u'|.$$

Exemples : pour $v :=$ *repérer*, $(epé, 2) \shortparallel (ére, 4)$ et $(re, 1) \shortparallel (re, 5)$.

(1) [1.0] Donnez une définition formelle en logique des prédicats d'une portion (u, p) d'un mot v.

Réponse. Voilà une définition *possible* : un couple $(u, p) \in \Sigma^* \times \mathbb{N}$ est appelé *portion* d'un mot $v \in \Sigma^*$ s'il satisfait la proposition suivante :

$$(\underbrace{p + |u| - 1 \leqslant |v|}_{(G)}) \;\wedge\; (\underbrace{\forall i \in [\![1, |u|]\!] \quad u_i = v_{p+i-1}}_{(D)}). \tag{1}$$

Le terme (G) de la conjonction (1) exprime qu'il reste au moins $|u|$ symboles dans le mot v à partir de la position p, le terme (D) que les $|u|$ symboles de u coïncident avec ceux de v à partir du p-ème symbole de v.

(2) [0.5] Démontrez que $\shortparallel$ est une relation d'équivalence.

Réponse. Il faut montrer que cette relation est réflexive, symétrique et transitive. C'est "évident" puisque la relation $\shortparallel$ repose exclusivement sur l'égalité des longueurs des mots et l'égalité est une relation d'équivalence.

Pour s'en convaincre : une portion $(u, p) \in P(v)$ est en relation avec elle même car $|u| = |u|$ donc $\shortparallel$ est réflexive. Si $(u_1, p_1) \shortparallel (u_2, p_2)$ alors $|u_1| = |u_2|$ et par symétrie de l'égalité $(u_2, p_2) \shortparallel (u_1, p_1)$ et $\shortparallel$ est donc symétrique. Si $(u_1, p_1) \shortparallel (u_2, p_2)$ et $(u_2, p_2) \shortparallel (u_3, p_3)$, alors $|u_1| = |u_2|$ et $|u_2| = |u_3|$ et la transitivité de l'égalité donne $|u_1| = |u_3|$, soit $(u_1, p_1) \shortparallel (u_3, p_3)$, prouvant la transitivité de $\shortparallel$.

(3) [0.5] Calculez le cardinal de chaque classe d'équivalence.

Réponse. Deux portions sont en relation si et seulement si elles ont même longueur, le nombre de classes d'équivalence est celui des longueurs de portions possibles, à savoir $n := |v|$. Notons $P_\ell \subseteq P(v)$ la classe des portions de longueur ℓ pour $\ell \in [\![1, n]\!]$. Comme $u \neq \varepsilon$, si $(u, p) \in P_\ell$, la condition (G) dans (1) permet d'obtenir un encadrement des positions p :

$$1 \leqslant p \leqslant n - \ell + 1,$$

et cet intervalle est de cardinal

$$|P_\ell| = (n - \ell + 1).$$

(4) [0.5] En déduire le nombre de portions possibles d'un mot v de longueur $n > 0$. Justifiez.

Réponse. Les classes d'équivalence pour la relation $\shortparallel$ forment une partition de $P(v)$, on peut donc appliquer la formule de sommation :

$$\begin{aligned}
|P(v)| &= \sum_{\ell=1}^{n} |P_\ell| \\
&= \sum_{\ell=1}^{n} n - \ell + 1 \\
&= n(n+1) - \sum_{\ell=1}^{n} \ell \\
&= n(n+1) - \frac{n(n+1)}{2} \\
&= \frac{n(n+1)}{2}.
\end{aligned}$$

(5) [3.5] Écrivez un algorithme `Chercher(u,v)` qui renvoie la plus petite des positions p des portions (u,p) d'un mot v si elle existe, et 0 sinon. Exemple : si u = ***che*** et v = *re**cher**cher*, la fonction doit renvoyer la valeur 3.

Réponse. On utilise une variable p pour calculer la position de la portion (u,p) que l'on cherche et une variable i pour indexer les symboles de u et ceux de v à partir de la position p, l'algorithme compare donc les symboles u_i et $v_{p+(i-1)}$. On commence aux positions $(i,p) \leftarrow (1,1)$ et à chaque passage dans la boucle, l'une exclusivement des deux variables i ou p est incrémentée, i si les deux symboles coïncident, p sinon et dans ce cas on réinitialise i à 1 pour faire repartir l'analyse au premier symbole de u.

Comme u est indexé par i et v par $p+i-1$, il faut s'assurer avant d'entrer dans la boucle que $i \leqslant |u|$ et que $p+i-1 \leqslant |v|$. Cette dernière condition est remplacée par $p+|u|-1 \leqslant |v|$ qui permet d'arrêter l'incrémentation de p s'il reste moins de $|u|$ symboles dans v à partir de la position p, évitant de commencer une nouvelle série de comparaisons inutiles (cf. Algo. 1).

(6) [0.5] Faites la preuve d'arrêt de votre algorithme.

Réponse. La condition d'entrée dans la boucle est

$$(i \leqslant |u|) \ \wedge \ (p+|u|-1 \leqslant |v|).$$

On sort donc de la boucle si

$$\underbrace{(i > |u|)}_{(I)} \ \vee \ \underbrace{(p > |v|-|u|+1)}_{(P)}. \tag{2}$$

À chaque passage dans la boucle, l'une des deux variables i ou p est incrémentée, l'une au moins des deux suites constituées de leurs valeurs successives est donc strictement croissante et satisfaira (2) puisqu'une suite strictement croissante d'entiers naturels n'est jamais majorée.

(7) [1.5] Donnez les éléments de la preuve de correction partielle de votre algorithme.

Réponse. On s'assure tout d'abord de ne jamais sortir des bornes d'indexation de structures énumérées, ici les mots u et v. Les valeurs initiales des variables i et p sont égales à 1. Comme l'égalité $u_i = v_{p+i-1}$ n'est testée que si la condition d'entrée dans la boucle est satisfaite, on a toujours

```
ALGORITHME Chercher(u,v):entier
DONNEES
 · u, v: mots
VARIABLES
 · i, p: entiers
DEBUT
 · p ← 1              # position cherchée dans v
 · i ← 1              # avancement sur u et v
 · TQ ((i <= |u|) ET (p + |u| - 1 <= |v|) FAIRE
 ·   · SI (u[i] = v[p + (i - 1)]) ALORS
 ·   ·   · i ← i + 1  # symbole suivant de u et v
 ·   · SINON
 ·   ·   · p ← p + 1  # on repart du symbole suivant
 ·   ·   · i ← 1      # de v et du début de u
 ·   · FSI
 · FTQ
 · SI (i > |u|) ALORS
 ·   · RENVOYER p     # on a trouvé (u,p)
 · SINON
 ·   · RENVOYER 0     # u n'apparaît pas dans v
 · FSI
FIN
```

Algo. 1. Recherche de la sous-chaîne u dans v.

$i \leqslant |u|$ et $p+|u|-1 \leqslant |v|$ et en additionnant on en déduit

$$(p+|u|-1+i \leqslant |v|+|u|) \ \Leftrightarrow \ (p+i-1 \leqslant |v|)$$

D'autre part, $1 \leqslant p+i-1$ puisque $i \geqslant 1$ et $p \geqslant 1$, par conséquent, on a toujours $i \in [\![1,|u|]\!]$ et $p+i-1 \in [\![1,|v|]\!]$.

Il reste justifier que la valeur renvoyée par l'algorithme est correcte. Supposons que l'algorithme renvoie une valeur p qui est nécessairement strictement positive puisque nous avons prouvé que $p \in [\![1,|v|-|u|+1]\!]$. Dans ce cas i a nécessairement été incrémentée $|u|$ fois *consécutivement*, sans quoi sa valeur aurait été réinitialisée à 1 et, lors de ces itérations, la variable p est restée invariante. Cela montre que tous les symboles de u ont été comparés positivement à ceux de v à partir de la position p pour v.

Nous venons de montrer que si l'algorithme renvoie une valeur strictement positive p, alors (u,p) est une portion de v.

Réciproquement, montrons que si (u,p) est une portion de v où p est minimal, alors l'algorithme renvoie la valeur p. Supposons que (u,p) soit la première portion de v. La condition `SINON` à l'intérieur de la boucle assure que si l'on n'a pas atteint la fin la reconnaissance de u dans v à la position p, la future comparaison se fera une position plus loin dans v et sur le premier symbole de u. Ainsi, puisque p a été initialisé à la première position de v, si $i > |u|$ en sortant de la boucle, on a nécessairement trouvé u à la première position p possible.

Comme l'algorithme renvoie 0 si et seulement s'il ne renvoie pas la position p et qu'il renvoie p si et seulement s'il a trouvé la première portion (u,p) de v, alors il renvoie 0 si et seulement s'il n'a pas trouvé de portion (u,p) dans v.

NB. Ces justifications ne constituent *pas* une preuve formelle, seulement des éléments de preuve.

(8) [2.0] En supposant que $|u| \leqslant |v|$, calculez la complexité de votre algorithme dans le meilleur des cas et dans le pire des cas.

Réponse. Les fonctions de complexité dépendent de la taille des instances en entrée, le meilleur des cas n'est donc *jamais* obtenu en considérant des instances de taille minimale, sinon il n'y a *évidemment* rien à faire...

Nous allons exprimer les complexités en fonction de $n := |v|$ et $m := |u|$ en comptant le nombre de comparaisons entre les symboles de u et v. Les instructions hors de la boucle `TQ` sont en effet en $\Theta(1)$ et la boucle ne contient qu'une instruction conditionnelle `SI` qui, satisfaite ou non, oriente vers deux groupes d'instructions en $\Theta(1)$. On suppose que $|\Sigma| \geqslant 2$, sinon en notant a l'unique symbole de Σ, on aurait $u = a^m$ et $v = a^n$ et $(u,1)$ serait toujours la portion recherchée. Pour le meilleur des cas, deux situations différentes sont à considérer :

a) Si u_1 est différent de tous les symboles de v, c'est la condition (P) de (2) qui fait sortir de la boucle après $n-m+1$ comparaisons.

b) Si u est un *préfixe* de v, i.e. $(u,1)$ est une portion de v, c'est la condition (I) qui fait sortir de la boucle après avoir comparé les m symboles de u avec les m premiers symboles de v.

La complexité dans le meilleur des cas est donc

$$\check{T}(n,m) = \Theta(\min\{m, n-m+1\}).$$

Notons a et b deux symboles distincts de l'alphabet Σ et supposons que $v := a^n$ et $u := a^{m-1}b$. Il faut alors attendre la dernière comparaison entre $u_m = b$ et $v_{p+m-1} = a$ pour décider que (u,p) n'est pas une portion de v et incrémenter p, il y aura donc un nombre total de comparaisons

$$\hat{T}(n,m) = \Theta(m \times (n-m+1)).$$

EXERCICE 2. (File de priorité) Un hopital doit gérer la prise en charge des patients qui entrent aux urgences. L'urgentiste code chaque nouveau patient avec un couple (a,p) où a est son ordre d'arrivée et p sa priorité. Plus la valeur de p est faible plus la priorité est importante, un patient en priorité 0 doit être soigné immédiatement.

Le stagiaire du service informatique propose d'utiliser une file initialement vide et d'y insérer les patients au fur et à mesure de leur arrivée. Quand un médecin peut soigner un nouveau patient, on cherche le premier patient avec la plus grande priorité et on le sort de la file.

Une cellule C est un enregistrement à trois champs, deux entiers `C.a` et `C.p` et une adresse `C.suiv` vers une cellule, et sert à coder un patient. Une file de priorité est codée par un enregistrement `F` à deux champs : une liste chaînée `F.lc` de cellules et une adresse `F.fin` vers la dernière cellule. On écrit `(F.lc)»x` pour désigner un champ `x` de la cellule adressée par `F.lc` et on note `[]` la liste vide.

(1) [1.0] Écrivez un algorithme `AjouterPatient(@F,a,p)` qui ajoute un patient à la fin de la file `F`. Indication : utilisez *sans l'écrire* l'algorithme `Allouer(a,p)` qui alloue une cellule, initialise ses trois champs à `a`, `p` et `[]` et renvoie son adresse. Voir Algo. 2.

Réponse. On crée tout d'abord une nouvelle liste `aux` constituée d'une unique cellule avec `Allouer(a,p)`. Si la liste `F.lc` n'est pas vide, on accroche `aux` à la fin grâce au pointeur `F.fin`, sinon on initialise la liste `F.lc` à `aux` et dans les deux cas, on met à jour `F.fin`.

(2) [1.0] Donnez les éléments de la preuve de correction de votre algorithme `AjouterPatient` puis calculez sa complexité.

```
ALGORITHME AjouterPatient(@F,a,p)
DONNEES
 · F: {lc, fin: listes}
 · a, p: entiers
VARIABLES
 · aux: liste
DEBUT
(1) · aux ← Allouer(a,p)
    · SI (F.lc = []) ALORS
(2) ·   · F.lc ← aux
    · SINON
(3) ·   · (F.fin)»suiv ← aux
    · FSI
(4) · F.fin ← aux
FIN
```

Algo. 2. Ajout d'un nouveau patient en fin de liste.

Réponse. L'algorithme s'arrête nécessairement puisqu'il n'y a pas de boucle. Après l'allocation, on dispose d'une liste `aux` constituée d'une unique cellule initialisée à $(a, p, [])$. Après l'instruction conditionnelle `SI`, si la liste `F.lc` était initialement vide, elle pointe désormais sur la même cellule que `aux`, sinon sa cellule de fin dont le champ `suiv` pointait sur la liste vide pointe sur la même cellule que `aux`. Dans les deux cas le pointeur sur la dernière cellule est mis à jour. Les figures 2 et 1 illustrent les mécanismes.

(3) [2.5] Écrivez un algorithme `ExtrairePatient(@F)` qui cherche le premier patient de la liste avec la priorité maximale, l'élimine de la liste et renvoie le couple (a, p) que contenait sa cellule.

Réponse. Il faut parcourir la liste `F.lc` des patients et trouver la position de la cellule de priorité maximale, donc de valeur p minimale. Pour pouvoir décrocher cette cellule de la liste `F.lc`, il faut se situer sur la cellule *qui précède* et gérer le cas particulier où le minimum est dans la première cellule auquel cas il n'y a pas de cellule précédente. Pour faciliter la compréhension de l'algorithme `ExtrairePatient(@F)`, on utilise un algorithme auxiliaire `ChercherMin(F)`, qui renvoie l'adresse de la cellule qui *précède* la cellule contenant la valeur minimale, ou la liste vide si c'est la

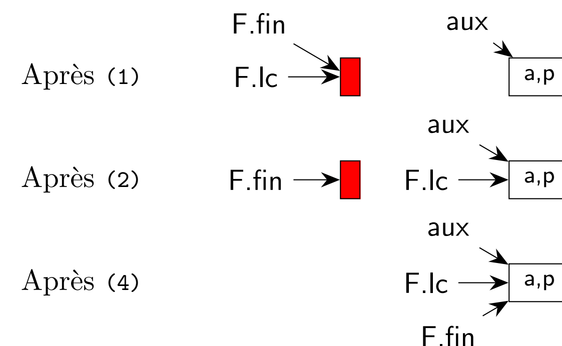

Figure 1. Insertion d'une cellule en queue de liste vide.

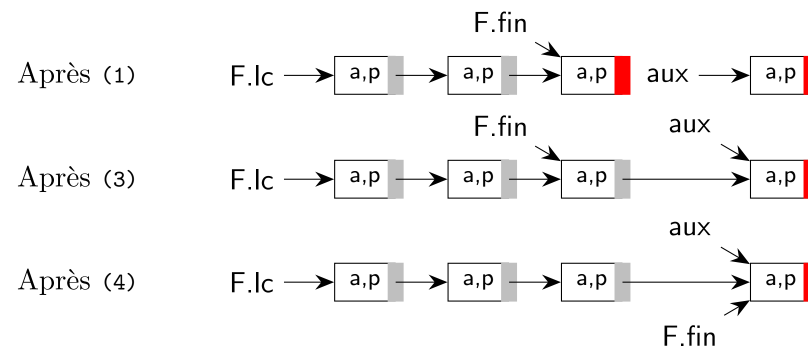

Figure 2. Insertion d'une cellule en queue de liste non-vide.

première de la liste `F.lc`. On suppose que la liste `F.lc` contient au moins une cellule quand on appelle `ChercherMin(F)`, le cas où la liste `F.lc` est vide est traité en amont dans l'algorithme `ExtrairePatient`.

L'algorithme `ChercherMin(F)` compare la valeur de priorité minimale, initialisée à celle de la première cellule, à celles des cellules suivantes de la liste `F.lc` en restant toujours une cellule en amont. Comme il n'y a pas de cellule en amont dans le cas où la valeur minimale est dans la première cellule, si l'adresse `lmin` est le début de la liste `F.lc`, c'est soit parce que la valeur minimale est dans la première cellule ou parce quelle est dans la deuxième, si cette deuxième cellule existe. On distingue les deux cas en

comparant les valeurs de priorité de ces deux cellules avec une inégalité large afin que le premier patient arrivé soit traité en cas d'égalité des priorités. Voir Algo. 3.

```
ALGORITHME ChercherMin(F)
DONNEES
 · F: {lc, fin: listes}
VARIABLES
 · l, lmin: listes
DEBUT
 · lmin ← F.lc       # init. adresse vers cellule de priorité "min"
 · l ← F.lc          # départ de la recherche en début de liste
 · TQ (l»suiv != []) FAIRE
 ·  · SI (l»suiv»p < lmin»p) ALORS
 ·  ·  · lmin ← l    # valeur p dans la cellule suivante plus faible
 ·  · FSI
 ·  · l ← l»suiv     # on passe à la cellule suivante
 · FTQ
 · SI (lmin = F.lc) ET (lmin»suiv != []) ET (lmin»p <= lmin»suiv»p) ALORS
 ·  ·  RENVOYER []   # la priorité "min" est dans la première
 · SINON             # cellule => pas de cellule en amont
 ·  ·  RENVOYER lmin
 · FSI
FIN
```

Algo. 3. Recherche de la cellule qui précède celle de priorité maximale.

L'algorithme `ExtrairePatient` n'a plus qu'à décrocher la cellule contenant la priorité maximale grâce à l'adresse de la cellule précédente renvoyée par l'algorithme `ChercherMin` en traitant le cas particulier où c'est la première cellule. Si la liste `F.lc` est vide on renvoie le couple $(0,0)$, ce qui ne peut être confondu avec une valeur valide pour un patient puisque l'ordre d'arrivée est supposé commencer à 1. La libération de la mémoire n'était pas demandée dans le sujet. Voir Algo. 4.

(4) [1.0] Donnez les éléments de la preuve de correction de votre algorithme `ExtrairePatient` puis calculez sa complexité.

```
    ALGORITHME ExtrairePatient(@F):(entier,entier)
    DONNEES
     · F: {lc, fin: listes}
    VARIABLES
     · a, p: entiers
     · l,lmin: listes
    DEBUT
     · SI (F.lc = []) ALORS
     ·  · (a,p) ← (0,0)          # code de retour si liste vide
     · SINON
(0)  ·  · lmin ← ChercherMin(F)
     ·  · SI (lmin = []) ALORS   # la cellule min est en début de
(1)  ·  ·  · l ← F.lc            # liste, on copie son adresse
(2)  ·  ·  · F.lc ← (F.lc)»suiv  # puis on la décroche
     ·  · SINON
(3)  ·  ·  · l ← lmin»suiv       # on copie l'adresse de la cellule
(4)  ·  ·  · lmin»suiv ← l»suiv  # min puis on la décoche
     ·  · FSI
     ·  · (a, p) ← (l»a, l»p)    # on copie les valeurs dans la cellule
     ·  · Liberer(l)             # min puis on libère la mémoire
     · FSI
     · RENVOYER (a,p)
    FIN
```

Algo. 4. Extraction du patient de priorité maximale.

Réponse. La variable `l` de l'algorithme `ChercherMin` est initialisée au début de la liste `F.lc` et passe à la cellule suivante uniquement si celle-ci existe. La dernière cellule pointant sur la liste vide par hypothèse sur l'algorithme d'allocation, on sort de la boucle. L'adresse `lmin` est initialisée au début de la liste `F.lc` et n'est mise à jour que si la priorité dans la cellule suivante pointée par `l` (elle existe, c'est la condition d'entrée dans la boucle) est strictement inférieure. Noter que la condition stricte est essentielle, sinon de tous les patients de même priorité maximale, c'est le dernier arrivé qui sera traité et pas le premier. Le test après la boucle permet de distinguer le cas où le minimum est dans la deuxième cellule, auquel cas l'adresse `lmin` est bien placée sur la cellule qui précède, à savoir

la première de la liste, ou alors le minimum est dans la première cellule et il faut traiter ce cas particulier.

L'algorithme `ExtrairePatient` s'arrête puisqu'il n'y a pas de boucle. Les deux schémas des figures 3 et 4 explicitent la gestion des différentes adresses et justifient l'algorithme.

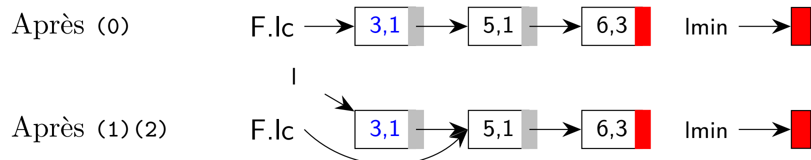

Figure 3. Extraction du patient de priorité maximale s'il est au début de la liste.

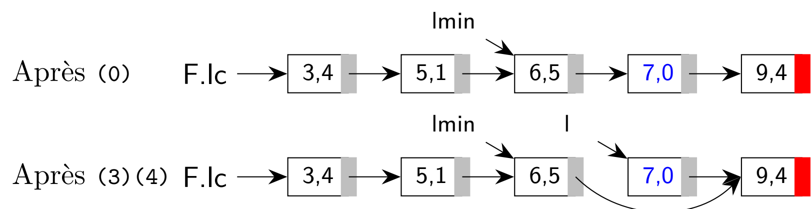

Figure 4. Extraction du patient de priorité maximale s'il n'est pas au début de la liste.

La complexité de l'algorithme `ChercherMin` est celle de la recherche d'un minimum dans une liste quelconque qui impose d'avoir parcouru toutes les valeurs de la structure, donc en $\Theta(n)$ si n désigne le nombre de termes dans cette structure. Si l'on excepte l'appel à l'algorithme `ChercherMin`, le nombre d'instructions exécutées par l'algorithme `ExtrairePatient` est borné par deux constantes, donc avec un coût en $\Theta(1)$. La complexité de l'algorithme `ExtrairePatient` est donc

$$T(n) = \Theta(n) + \Theta(1) = \Theta(n).$$

Un ancien étudiant de la licence d'informatique est recruté dans le même hopital et propose de faire mieux en gérant la file de priorité avec un *tas*, de manière à ce que le patient avec la plus grande priorité soit toujours en première position (la valuation d'un nœud de l'APO associé est plus *petite* que celles de ses fils).

La file de priorité est codée par un enregistrement `FP` à 2 champs, le tas `FP.T` indexé à partir de 1, et l'index `FP.n` du dernier patient, initialement nul pour indiquer que la file de priorité est vide. On suppose que le tas `FP.T` est de taille suffisante pour que $n \leqslant |\texttt{FP.T}|$ sans avoir à le vérifier.

(5) [2.5] Écrivez un algorithme `AjouterPatient(@FP,a,p)` qui ajoute un patient dans la file de priorité. Indication : utilisez, *sans l'écrire* l'algorithme `Tamiser(@T,ip,ifin)` qui tamise le sous-arbre de racine `ip` sans descendre plus bas que `ifin`.

```
ALGORITHME AjouterPatient(@FP,a,p)
DONNEES
 · FP: {T: tableau, n: entier}
VARIABLES
 · ip: entier
DEBUT
 · FP.n ← FP.n + 1
 · FP.T[FP.n] ← {a,p}    # ajout du nouveau patient
 · ip ← FP.n DIV 2       # calcul de l'indice du père
 · TQ (ip > 1) FAIRE
 ·  · Tamiser(FP.T, ip, FP.n)
 ·  · ip ← ip DIV 2      # calcul de l'indice du père
 · FTQ
FIN
```

Algo. 5. Ajout d'un nouveau patient dans la file de priorité.

Réponse. On dispose donc d'un APO avec la valeur de priorité minimale à la racine de l'arbre et au début du tas. Le patient prioritaire est donc le premier élément du tas et on peut coder un patient avec la même structure `{a,p}` que dans la première partie mais sans l'adresse devenue inutile. L'ajout d'un nouveau patient dans ce tas, peut lui faire perdre la propriété APO, il faudra donc la rétablir. On peut être tenté de placer le nouveau

patient à la racine, i.e. en `FP.T[1]`, puis d'appliquer l'algorithme de tamisage à la racine, mais cette valeur prendrait la place de la valeur minimale qu'il faudrait replacer correctement dans le tas, alors qu'elle a de bonnes raisons d'y rester statistiquement.

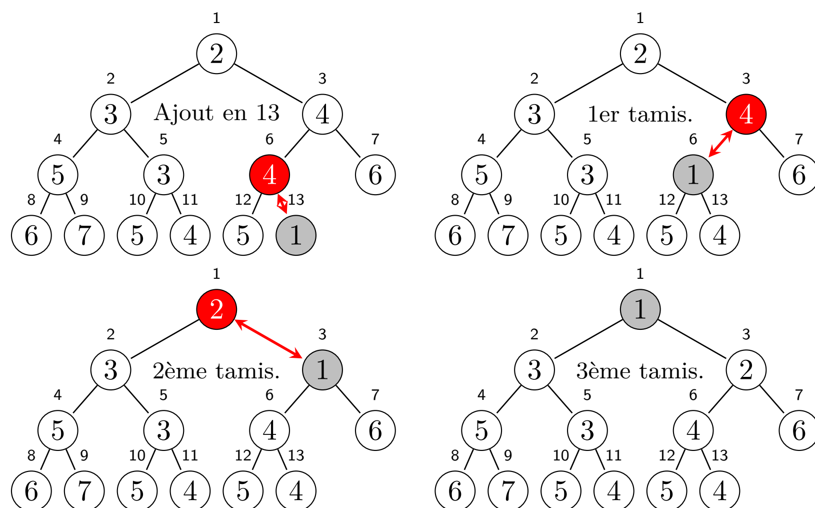

Figure 5. Ajout d'un patient de priorité 1 à l'indice 13 dans le tas puis rétablissement de la propriété apo.

On laisse donc la valeur minimale à sa place en début de tableau et on insère le nouveau patient à la "fin" du tableau `FP.T`. On s'inspire de l'algorithme `Entasser` qui transforme un tableau quelconque en tas pour rétablir la propriété apo. On applique donc l'algorithme de tamisage en partant du nœud père de la feuille nouvellement introduite à la dernière position n (on a mis à jour `FP.n`), et dont le père est d'indice $\lfloor \frac{n}{2} \rfloor$. À la différence d'un tableau quelconque, il est inutile de tamiser tous les nœuds à partir du nœud père puisque l'arbre était apo avant l'introduction de la feuille. Ainsi, au lieu de tamiser successivement tous les nœuds de l'arbre dans l'ordre décroissant à partir du nœud père, on tamise de père en père jusqu'à la racine. On suppose que l'algorithme de tamisage utilise pour valuation la valeur de priorité p d'un patient codé par l'enregistrement `{a,p}`. La figure 5 illustre le fonctionnement de l'algorithme 5 en ne mentionnant que la priorité dans les nœuds.

(6) [2.0] Donnez les éléments de la preuve de correction de votre algorithme `AjouterPatient` puis calculez sa complexité.

Réponse. On suppose bien entendu que l'algorithme `Tamiser` est correct, la suite des valeurs entières de la variable `ip` initialisée à une valeur positive est strictement décroissante et toutes ses valeurs sont positives, elle ne peut donc être minorée et fait échouer à terme la condition `ip > 1`.

La correction partielle repose sur la celle de l'algorithme de tamisage et la propriété suivante (rappel du cours) : si les sous-arbres gauche et droit d'un arbre binaire sont des apo, alors l'arbre obtenu après tamisage de sa racine est un apo. Notons F la feuille d'indice n où est ajouté un nouveau patient (a,p) et $P := \Gamma^{-1}(F)$ son père d'indice $\lfloor \frac{n}{2} \rfloor$. Comme l'arbre associé au tas était un apo avant l'ajout de F, seul le sous-arbre de racine P peut perdre cette propriété si $\nu(P) > p$ avec $p = \nu(F)$. Après tamisage de P, s'il y a eu échange des valuations de F et P, alors c'est le sous-arbre de racine $G := \Gamma^{-1}(P)$ (le grand père de F) qui peut perdre le statut d'apo si $\nu(G) > \nu(P)$, propageant le cas échéant le problème de père en père jusqu'à la racine de l'arbre.

Si la propriété d'apo n'est pas satisfaite à cause des valuations d'un père P et celle p de son fils F associée au nouveau patient, l'algorithme `Tamiser` réalise un unique échange entre ces deux valuations et s'arrête. En effet, exceptée la valuation p de F, les valuations des autres nœuds de l'arbre de racine P sont toutes supérieures à $\nu(P)$ puisque l'arbre était un apo avant l'ajout du nouveau patient. Ceci montre que la complexité de chaque tamisage est en $\Theta(1)$. Le nombre total de tamisages est évidemment égal à la profondeur de l'arbre, à savoir $\lfloor \log_2(n) \rfloor$, autrement dit :

$$T(n) = \Theta(1).\Theta(\log(n)) = \Theta(\log(n)).$$

Noter qu'il est inutile de remonter jusqu'à la racine si l'algorithme `Tamiser` ne fait pas d'échange entre les valuations du père et du fils, on peut donc améliorer le procédé en modifiant l'algorithme `Tamiser` pour qu'il renvoie un booléen indiquant s'il y a eu échange ou non. La complexité serait alors en $O(\log(n))$.