Structure de données arborescentes

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

2 mars 2025

1 Arbres

Les arbres permettent de représenter des situations et des objets hiérarchiques. Par exemple :

— Les résultats d'un tournoi

— Les dossiers d'un ordinateur

— La structure d'un document HTML

— Les expressions arithmétiques

— Les programmes OCaml

Les arbres permettent également d'implémenter de manière efficace certaines structures de données comme les dictionnaires.

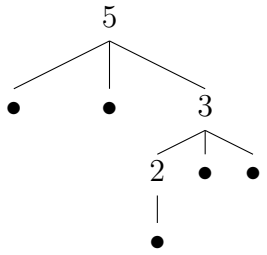
Définition 1. Soit E un ensemble d'éléments appelés ***étiquettes***. On défini par induction l'ensemble des arbres étiquetés par E, noté $\mathcal{A}_E$ comme suit :

— L'arbre vide $\bullet$ est un arbre étiqueté par E : $\bullet \in \mathcal{A}_E$.

— Pour $e \in E, r \in \mathbb{N}^*$, et $A_1, ...A_r$ des arbres étiquetés par E, $N(e, [A_1, ..., A_n]) \in \mathcal{A}_E$. On dit que c'est un arbre d'arité r.

Autrement dit, $\mathcal{A}_E$ est l'ensemble construit par induction à partir des constructeurs $\bullet$ et N.

Par exemple, $N(5, [\bullet, \bullet, N(3, [N(2, [\bullet]), \bullet, \bullet])])$ est un arbre. On peut le représenter graphiquement sous la forme suivante :



Définition 2. On définit la relation "***est un sous arbre de***" inductivement :

— Pour $A \in \mathcal{A}_E$, A est un sous-arbre de lui-même

— Pour $A, A' \in \mathcal{A}_E$, si $A = N(e, [A_1, ..., A_n])$ avec $A_1, \ldots, A_r \in \mathcal{A}_E$, si A' est un sous-arbre d'un des $A_i, i \in [\![1, r]\!]$, alors c'est un sous-arbre de A.

Autrement dit, A' est un sous arbre de A s'il apparaît dans A. Si de plus, $A' \neq A$, on dit que c'est un sous-arbre strict.

Définition 3. Soit $k \in \mathbb{N}^*$ On dit qu'un arbre A est k-aire si tous ses sous-arbres non vides sont d'arité k.

Quelques cas particuliers : pour $k = 2$ on parle d'arbre binaire, pour $k = 3$ d'arbre ternaire.

Exemple 1. Quelques exemples d'arbres k-aires :

A Chemins, noeuds

On pose $k \in \mathbb{N}$, et on considère dans cette partie des arbres k-aires. On pose $\Sigma = \{1, 2, \ldots, k\}$. On rappelle qu'un ***mot*** sur Σ est une suite finie d'éléments de Σ. On note Σ^* l'ensemble des mots sur Σ.

Définition 4. Pour deux mots $u, v \in \Sigma^*$, on note $u.v$ la concaténation de u et v.

Si $u \in \Sigma^*$ et $S \subseteq \Sigma^*$, on note $u.S$ l'ensemble des mots obtenus en concaténant un élément de S à u :

$$u.S = \{u.v | v \in S\}$$

Exemple 2. Sur l'alphabet $\{0, 1\}$, $01.\{0, 1, 10\} = \{010, 011, 0110\}$.
Si $u \in \Sigma^*$, $u.\emptyset = \emptyset$.

Un **noeud** d'un arbre est une position dans l'arbre. Formellement, cela va correspondre à un chemin dans l'arbre :

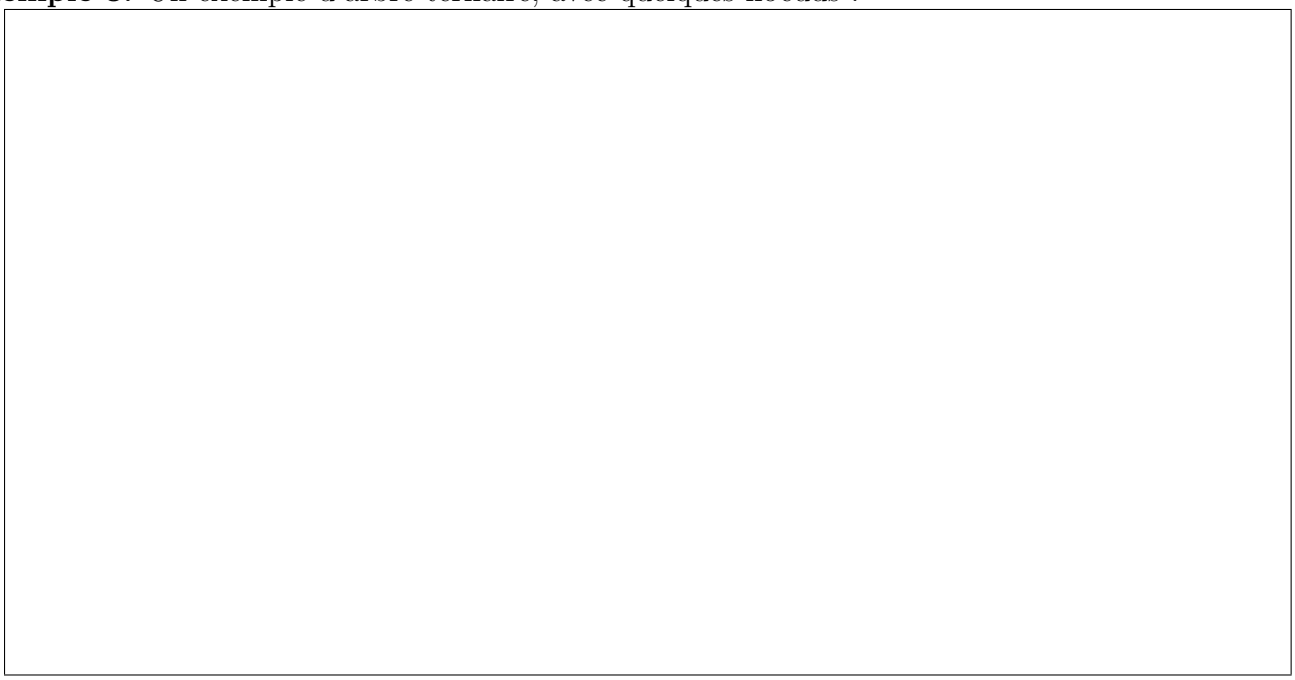
Définition 5. Soit $A \in \mathcal{A}_E$. On note $\mathbf{ch}(A) \subseteq \Sigma^*$ l'ensemble des chemins admissibles de A, que l'on définit par induction sur $\mathcal{A}_E$:

— $\mathbf{ch}(\bullet) = \emptyset$: l'arbre vide n'admet aucun chemin

— $\mathbf{ch}(N(e, [A_1, \ldots, A_k])) = \{\varepsilon\} \cup 1.\mathbf{ch}(A_1) \cup 2.\mathbf{ch}(A_2) \cup \cdots \cup k.\mathbf{ch}(A_k)$

On appelle ***noeud*** de A tout élément $n \in \mathbf{ch}(A)$. La ***profondeur*** d'un noeud est sa longueur en tant que mot sur Σ. La ***racine*** d'un arbre est le chemin vide ε.

Exemple 3. Un exemple d'arbre ternaire, avec quelques noeuds :



On définit ensuite l'étiquette d'un noeud, qui est l'étiquette notée à cette position de l'arbre :

Définition 6. Soit A un arbre non vide, et n un noeud de A. On définit l'***étiquette*** de n dans A, notée $\mathbf{et}(A, n)$, par récurrence sur la profondeur de n :

- $\mathbf{et}(N(e, [...]), \varepsilon) = e$
- Pour $i \in \Sigma$ et $u \in \Sigma^*$, $\mathbf{et}(N(e, [A_1, \ldots, A_k], i.u) = \mathbf{et}(A_i, u)$

Définition 7. On appelle ***taille*** de A le nombre de noeuds de A.
On appelle ***hauteur*** de A la profondeur de son noeud le plus profond.
La hauteur de l'arbre vide est -1.

Exemple 4. Quelques arbres, avec leur hauteur et leur taille :

Définition 8. Soit A un arbre, n, n' des nœuds de A. Alors :

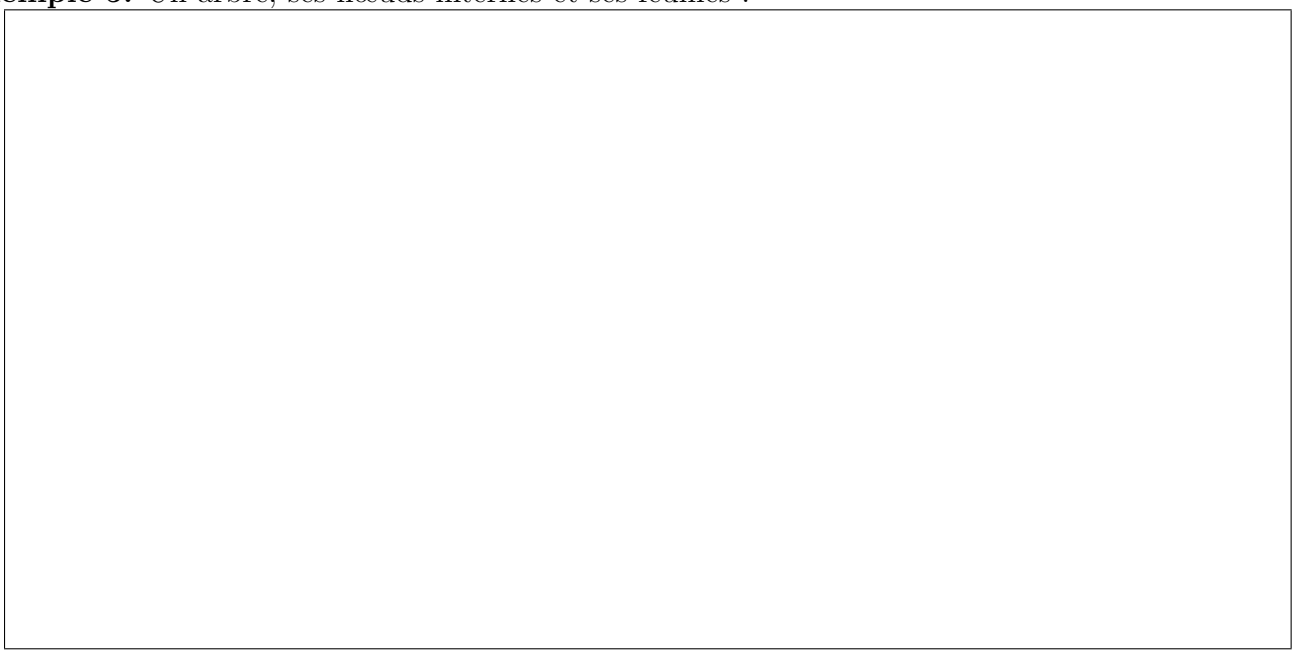
— n' est un ***enfant*** de n si il existe $i \in \Sigma$ tel quel $n' = n.i$. On dit alors que n est le ***parent*** de n'.

— n' est un ***descendant*** de n si il existe $u \in \Sigma^*$ tel quel $n' = n.u$.

Pour les arbres binaires, on parle d'enfant gauche et d'enfant droit pour parler des deux enfants d'un noeud.

On dit qu'un nœud est une ***feuille*** s'il n'a pas d'enfants. Sinon, on dit que c'est un ***nœud interne***.

Exemple 5. Un arbre, ses nœuds internes et ses feuilles :



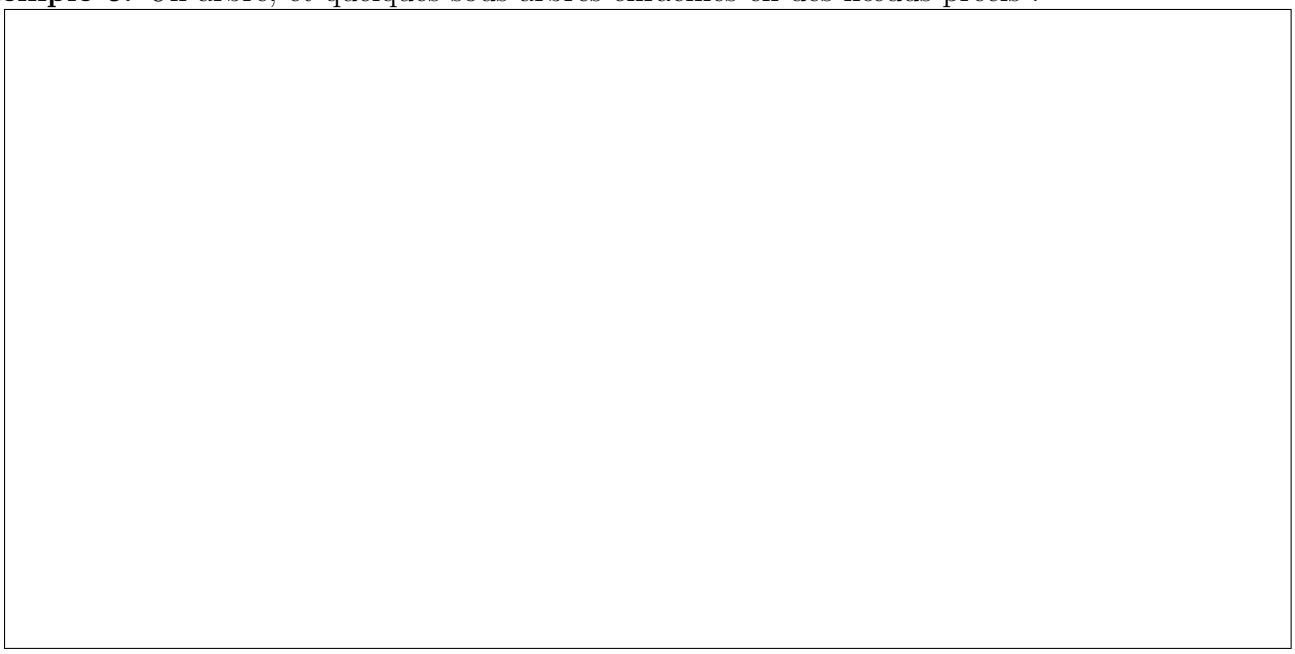
Dans un arbre A, chaque sous-arbre non-vide correspond à un nœud de A :

Définition 9. On définit par récurrence sur la profondeur des nœuds les arbres enracinés :

— Pour A un arbre non vide, de racine ε, l'arbre enraciné en ε dans A est A

— Si A est un arbre et $n = i.n'$ un nœud de A différent de la racine, alors A a au moins i fils $A_1, \ldots, A_i$, et l'arbre enraciné en n dans A est l'arbre enraciné en n' dans A_i.

Exemple 6. Un arbre, et quelques sous-arbres enracinés en des nœuds précis :



Exercice 1. Pour A un arbre et n un nœud de A, on considère A_n l'arbre enraciné en n. Soit n_1 un nœud de A_n. Montrer que n_1 dans A_n et $n.n_1$ dans A ont la même étiquette.

Solution. On procède par récurrence sur la longueur de n : Montrons :

$$\forall l \in \mathbb{N}, \forall A \text{ arbre}, \forall n \text{ nœud de } A \text{ de longueur } l, \forall n_1, \mathbf{et}(A, n.n_1) = \mathbf{et}(A_n, n_1)$$

où A_n signifie l'arbre enraciné en n dans A.

— Si $n = \varepsilon$ est la racine de A, alors $A_n = A$ et $n.n_1 = n_1$: la propriété est vraie

— Si $n = i.n'$ avec $i \in \Sigma$ alors A_n est l'arbre enraciné en n' dans A_i, et $\mathbf{et}(A, n.n_1) = \mathbf{et}(A_i, n'.n_1)$. Par hypothèse de récurrence, $\mathbf{et}(A_i, n'.n_1) = \mathbf{et}((A_i)_{n'}, n_1)$. De plus, $\mathbf{et}((A_i)_{n'}, n_1) = \mathbf{et}(A_{i.n'}, n_1) = \mathbf{et}(A_n, n_1)$ par définition des arbres enracinés.

Définition 10. Soit A un arbre.

— On appelle ***branche*** de A toute suite de nœuds $(n_1, \ldots, n_k)$ tels que pour $i \in [\![1, k-1]\!]$, n_{i+1} est un enfant de n_i.

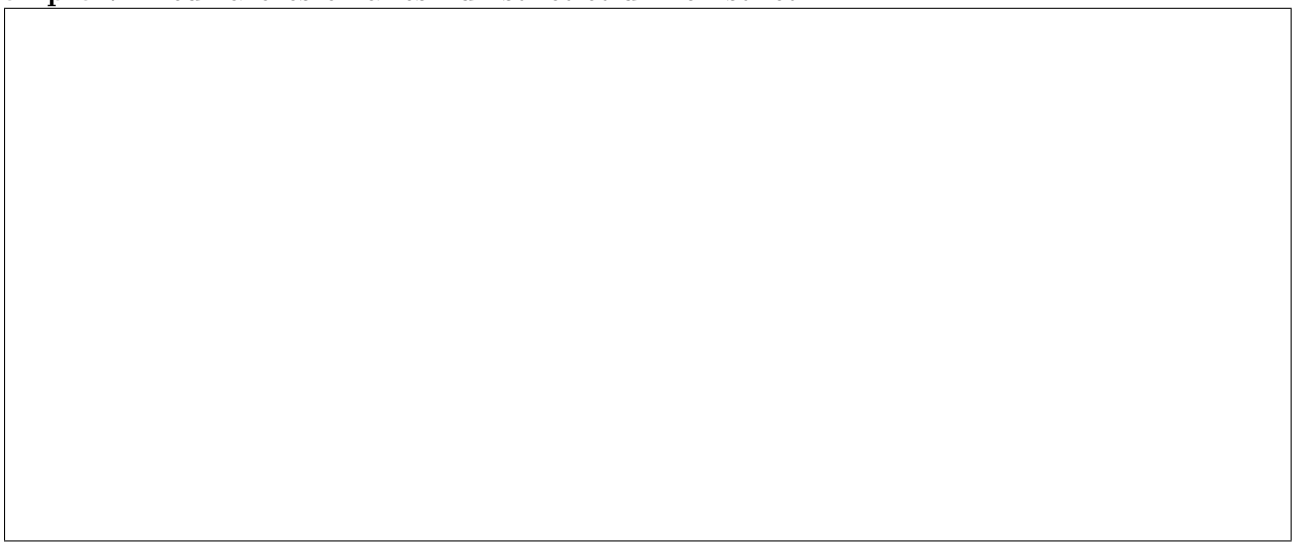
— On appelle ***chemin*** de A toute suite de nœuds $(n_1, \ldots, n_k)$ tels que pour $i \in [\![1, k-1]\!]$, n_{i+1} est un enfant de n_i ou n_i est un enfant de n_{i+1}.

— On appelle **chemin strict** dans A tout chemin dont les nœuds sont deux à deux distincts (i.e. ne repassant jamais par un même nœud).

B Arbre strict

Définition 11. On dit qu'un arbre est k-aire strict si tous ses nœuds ont exactement 0 ou k enfants, i.e. si tous ses nœuds internes ont exactement k enfants.

Exemple 7. Deux arbres binaires : un strict et un non-strict :



Remarque 1. Un arbre k-aire strict non vide est de la forme $N(e, [\bullet, \ldots, \bullet])$ ou de la forme $N(e, [A_1, \ldots, A_k])$ avec les A_i tous non-vides.

En général, lorsque l'on considère des arbres stricts, on les définit directement par induction, en utilisant un constructeur pour les feuilles et un pour les nœuds internes :

Exercice 2.

On définit $\mathcal{AS}_E^k$ l'ensemble des arbres k-aires stricts étiquetés par E par induction :

— Pour $e \in E$, $\mathbf{F}(e) \in \mathcal{AS}_E^k$: c'est une feuille étiquetée par e.

— Pour $e \in E$ et $a_1, \ldots a_k \in \mathcal{AS}_E^k$, $\mathbf{N}(e, a_1, \ldots, a_k) \in \mathcal{AS}_E^k$: c'est un nœud interne étiqueté par e et ayant $a_1, \ldots a_k$ comme enfants.

Quelques propriétés générales sur les arbres stricts.

Question 1. Définir une fonction inductive permettant de calculer la taille d'un arbre strict, et donner sa complexité en fonction de n le nombre de nœuds dans l'arbre. Faire de même pour compter le nombre de feuilles et de nœuds internes.

Question 2. Définir une fonction inductive permettant de calculer la hauteur d'un arbre strict, et donner sa complexité en fonction de n le nombre de nœuds dans l'arbre.

Question 3. On considère un arbre k-aire strict, a. On note h sa hauteur, i son nombre de nœuds internes, f son nombre de feuilles et n son nombre de nœuds. Montrer les relations suivantes entre h, f, i et n :

1. $n = f + i$
2. $(k-1)i + 1 = f$
3. $f \leq k^h$
4. $(k-1)n + 1 \leq k^{h+1}$

En particulier, pour un arbre binaire strict, on a $f = i + 1$ et $n + 1 \leq 2^{h+1}$.

Question 4. Montrer que pour tout $h \in \mathbb{N}$, il existe un arbre binaire strict pour lequel $n + 1 = 2^{h+1}$.

On considère dans la suite des arbres binaires stricts. Pour un arbre A, on appelle ***diamètre*** de A la longueur maximale d'un chemin entre 2 nœuds quelconques de A. Un chemin étant ici une suite de nœuds tels que deux nœuds successifs sont liés par parenté, dans un sens ou dans l'autre.

Question 5. Un chemin de longueur maximale passe-t-il forcément par la racine ?

Question 6. Donner une fonction inductive permettant de calculer le diamètre d'un arbre.

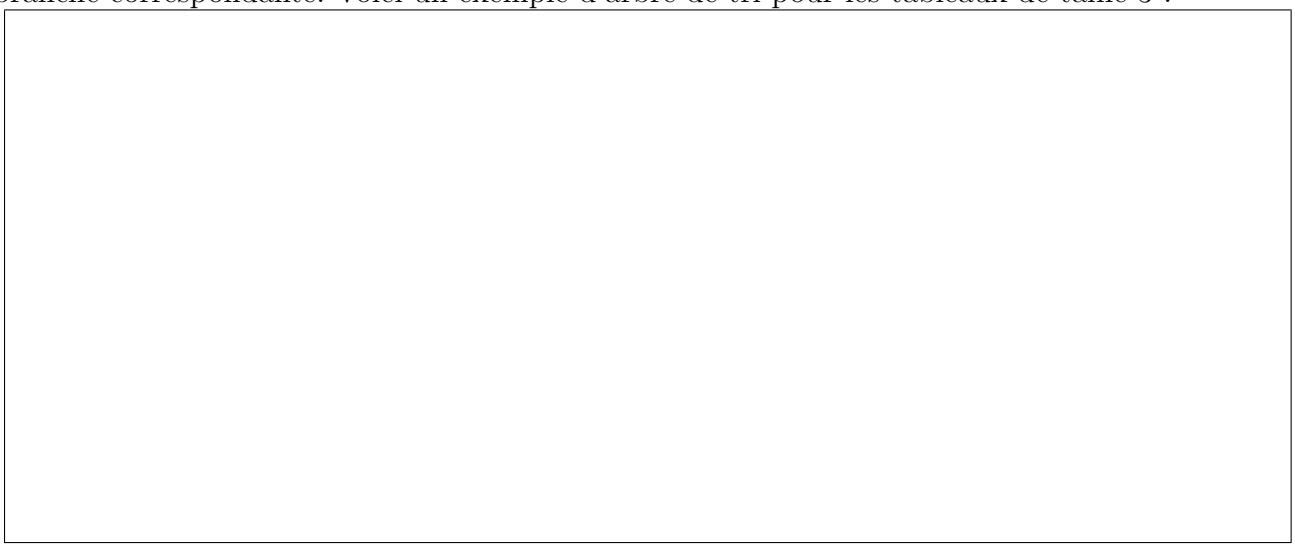
Question 7. Évaluer la complexité de cette fonction. Proposer une solution en $O(n)$ avec n la taille de l'arbre.

L'inégalité $n + 1 \leq 2^{h+1}$ est une inégalité fondamentale des arbres binaires. On peut la réécrire $h \geq \log_2(n+1)$, et elle exprime alors qu'un arbre ayant n nœuds doit avoir une hauteur au moins logarithmique en n. De même, on a l'inégalité $h \geq \log_2(f)$.

Exercice 3. Nous avons vu que le tri fusion s'effectue en $\mathcal{O}(n \log n)$. Montrons que c'est la meilleure complexité asymptotique possible.

On appelle ***tri par comparaison*** un algorithme de tri qui ne peut que comparer les valeurs du tableau à trier, sans connaître la structure même des valeurs. Tentons de minorer le nombre de comparaisons effectuées par un algorithme de tri par comparaison dans le pire cas.

On peut représenter tout algorithme de tri par comparaison par un arbre de tri : chaque nœud interne est une comparaison de la forme "$T[i] \leq T[j]$?", avec deux branches selon que la réponse soit oui ou non. Une feuille correspond à l'ordre déterminé par l'algorithme en suivant la branche correspondante. Voici un exemple d'arbre de tri pour les tableaux de taille 3 :



Question 1. Combien un arbre de tri doit-il y avoir de feuilles au moins ?

Question 2. En déduire la hauteur minimale d'un arbre de tri.

On peut même montrer qu'un tri par comparaison effectuera **en moyenne** au moins de l'ordre de $n \log n$ opérations :

Question 3. Montrer que la hauteur moyenne $h_m(A)$ d'un arbre A, définie comme la moyenne des profondeurs des feuilles, vérifie $h_m(A) \geq \log_2(f(A))$.

2 Parcours d'arbre

Un parcours d'arbre est un algorithme qui énumère les nœuds d'un arbre, un à un. Dans la suite, on identifiera les arbres et leur racine pour alléger les notations. Par exemple, les "enfants de A" seront les arbres enracinés en les enfants de la racine de A.

Parcours en profondeur

Les algorithmes de parcours en profondeur ont le schéma suivant :

Algorithme 1 : Parcours

Entrée(s) : A un arbre
1 **si** A ***est vide*** **alors**
2 **retourner**
// pré-traitement
3 $A_1, \ldots, A_r \leftarrow$ enfants de A;
4 **pour** $i = 1$ **à** r **faire**
5 **Parcours**(A_i);
// post-traitement

Les phases de pré/post-traitement dépendent de ce que l'on est en train de faire. Par exemple, écrivons une fonction qui affiche les étiquettes d'un arbre binaire :

```
1  type arbre =
2    Vide | Noeud of int * arbre * arbre;;
3
4  let rec lister1 a = match a with
5    | Vide -> ()
6    | Noeud(n, g, d) ->
7      print_int n;
8      lister1 g;
9      lister1 d
10
11 let rec lister2 a = match a with
12   | Vide -> ()
13   | Noeud(n, g, d) ->
14     lister2 g;
15     lister2 d;
16     print_int n
```

Selon l'ordre dans lequel on effectue les opérations, les nœuds ne sont pas affichés dans le même ordre. On parle de parcours ***préfixe*** lorsque l'on traite les nœuds avant leurs enfants, et de parcours ***postfixe*** lorsque l'on traite les nœuds après leurs enfants.

De plus, pour les arbres binaires, il existe la possibilité de faire des parcours ***infixe***, qui consistent à traiter chaque nœud entre ses deux enfants :

```
1  let rec lister3 a = match a with
2    | Vide -> ()
3    | Noeud(n, g, d) ->
4      lister2 g;
5      print_int n;
6      lister2 d
7  ;;
```

Exemple 8. Un arbre, et ses trois ordres préfixe, infixe et postfixe :

Exemple 9. On considère un arbre binaire strict, étiqueté par $\{+, -, \times, /\} \cup \mathbb{N}$, tel que les feuilles sont étiquetées par $\mathbb{N}$, et les nœuds internes par les opérateurs. Un tel arbre représente une expression arithmétique : si on l'affiche selon les ordres préfixes, infixes ou postfixes, on obtient respectivement la notation polonaise, la notation standard et la notation polonaise inversée :

Parcours en largeur

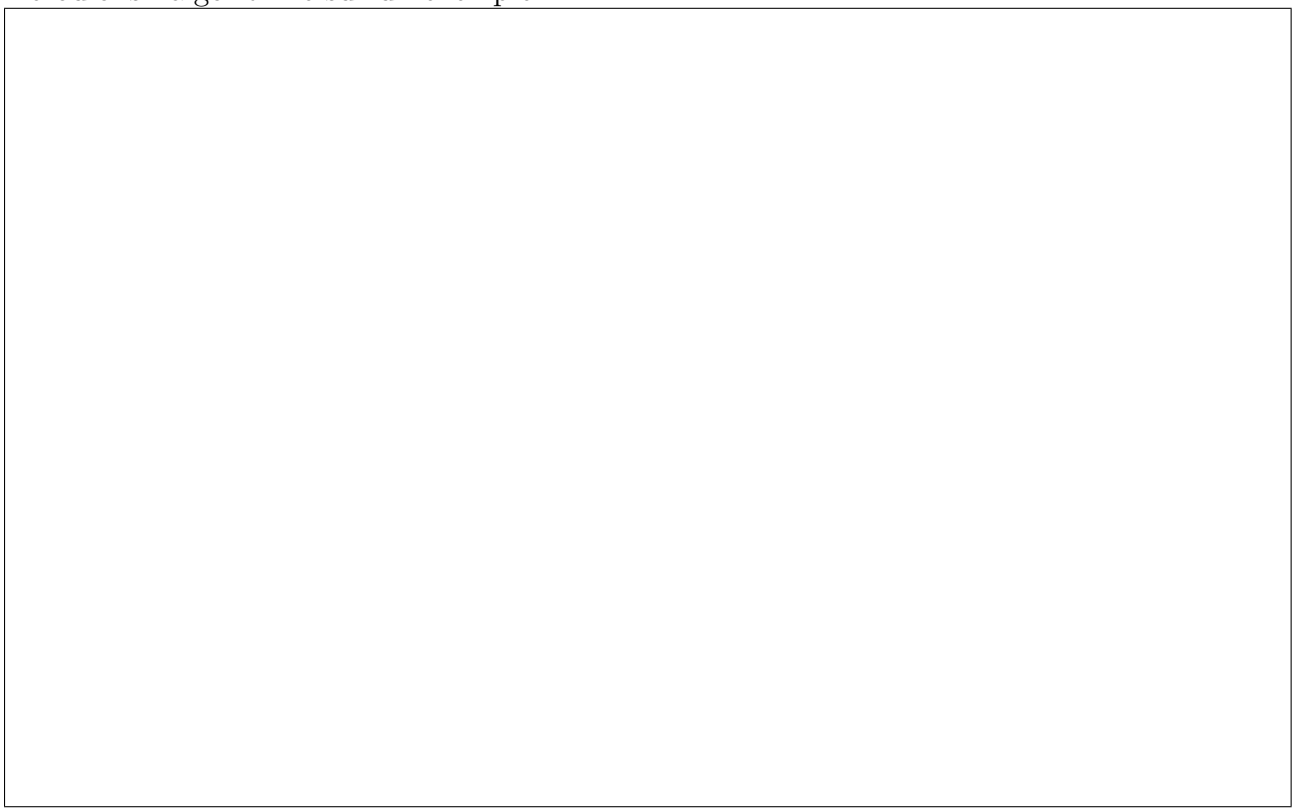
Le parcours en largeur consiste à traiter successivement tous les nœuds de profondeur p avant de traiter tous ceux de profondeur $p+1$. Il est moins naturel d'écrire un tel parcours de manière récursive. Les parcours en profondeur s'écrivent facilement en utilisant une structure de ***file*** :

Algorithme 2 : Parcours en largeur

Entrée(s) : A un arbre

```
1 r ← la racine de A ;
2 F ← une file vide ;
3 F.enfiler(r);
4 tant que F ≠ ∅ faire
5 |   u ← F.defiler();
6 |   Traiter u;
7 |   pour v enfant de u faire
8 |   |   F.enfiler(v);
```

Déroulons l'algorithme sur un exemple :



On peut montrer l'invariant suivant, qui permet de garantir que les nœuds sont traités par ordre de profondeur :

F est ordonnée par profondeurs croissantes, et l'écart entre les profondeurs des nœuds dans F est de 0 ou de 1.

Remarque 2. En remplaçant la file par une pile, on retrouve un parcours en profondeur :

3 Arbres binaires de recherche

Une application essentielle des arbres est l'implémentation des structures d'ensemble et de dictionnaire.

Définition 12. Le structure abstraite d'ensemble permet de stocker des éléments uniques. Ses opérations de bases sont :

— Créer un ensemble vide.

— Ajouter un élément à l'ensemble. Si l'élément est déjà présent, ça n'a aucun effet.

— Supprimer un élément à l'ensemble. Précondition nécessaire : l'élément est présent dans l'ensemble.

— Déterminer si l'ensemble contient un élément donné.

Si l'on implémente cette structure en OCaml, on aura donc :

```
1  type 'a set = ...
2
3  (* ensemble vide *)
4  let empty_set : 'a set = ...
5
6  (* add s x est l'ensemble s auquel x est rajouté *)
7  let add (s: 'a set) (x: 'a) : 'a set = ...
8
9  (* delete s x est l'ensemble s dans lequel x est supprimé
10    (hypothèse: x est dans s *)
11 let delete (s: 'a set) (x: 'a) : 'a set = ...
12
13 (* contains s x = true si x est dans s, false sinon *)
14 let contains (s: 'a set) (x: 'a) : bool = ...
```

A Principe général

On stocke les éléments de l'ensemble dans les étiquettes d'un arbre binaire. L'ensemble représenté par un arbre binaire est donc l'ensemble des étiquettes de ses nœuds.

Exemple 10. Implémentons une fonction permettant de rechercher un élément dans un arbre binaire quelconque : On définit inductivement la fonction **recherche**(A, x) où A est un arbre binaire et x un élément à chercher. Cette fonction renvoie T (true) ou F (false) selon si x est dans A ou non. Notons $\vee$ le OU booléen :

— **recherche**$(V, x) = F$

— **recherche**$(N(y, g, d), x) = (x = y) \vee$ **recherche**$(g, x) \vee$ **recherche**(d, x)

La complexité est linéaire en la taille de l'arbre car dans le pire cas, on inspecte chaque nœud de l'arbre une et une seule fois.

Le principe des arbres binaires de recherche va être de stocker les éléments de manière ordonnée, de telle sorte que pour rechercher un élément, on n'aura qu'à explorer un seul des deux sous-arbres à chaque étape. Les arbres binaires de recherche vont donc fonctionner selon un principe proche de la dichotomie, en permettant de diviser par deux l'espace de recherche à chaque étape.

Définition 13. Soit E un ensemble d'étiquettes munies d'un ordre $\leq$ ***total***. Un arbre A étiquetté par E est un ***arbre binaire de recherche*** si il est vide, ou si il est de la forme $N(x, g, d)$ avec $x \in E$ et g, d deux arbres binaires de recherche tels que :

— Pour toute étiquette x_g dans g, $x_g < x$

— Pour toute étiquette x_d dans d, $x_d > x$

On notera ABR pour Arbre Binaire de Recherche.

Exemple 11. Pour chaque arbre, dire si c'est un ABR ou non, en justifiant :

On remarque que par définition des ABR, un sous-arbre d'un ABR est un ABR. On peut donc raisonner par induction structurelle sur les ABR. Par exemple :

Proposition 1. Un arbre binaire est un ABR si et seulement si la liste de ses étiquettes dans l'ordre infixe est croissante.

Démonstration. Commençons par montrer le sens direct. Montrons par induction sur la structure d'ABR que pour tout ABR A, la liste dans l'ordre infixe des étiquettes de A est croissante.

— Si A est vide : A n'a pas d'étiquettes, donc son parcours infixe donne une liste vide, qui est croissante.

— Si $A = N(x, g, d)$ avec $x \in E$ et g, d des ABR, alors par hypothèse d'induction, L_g et L_d les listes des étiquettes de g et d dans l'ordre infixe sont croissantes. Or, le parcours infixe de A donne la liste $L_g@[x]@L_d$. De plus, toutes les étiquettes de g sont inférieures à x, et toutes les étiquettes de d sont supérieures à x. Donc, $L_g@[x]@L_d$ est croissante.

Montrons maintenant le sens indirect. Montrons par induction sur les arbres binaires que : pour tout arbre binaire A, si le parcours infixe de A est croissant, alors A est un ABR.

— Si A est vide, A est un ABR.

— Si $A = N(x, g, d)$ avec $x \in E$, g, d des arbres binaires, supposons que le parcours infixe de A est croissant. Alors, les parcours infixes de g et d sont également croissantes, et donc par hypothèse d'induction, g et d sont des arbres binaires. De plus, le parcours infixe de A liste toutes les étiquettes de g, puis x, puis toutes les étiquettes de d. Donc, les étiquettes de g sont inférieures à x, et celles de d sont supérieures à x : A est bien un ABR.

□

B Recherche

On définit la fonction **contient** : $\mathcal{A}_E \times E \to \mathbb{B}$ par induction sur les arbres binaires de recherche comme suit :

— Pour $x \in E, \mathbf{contient}(\mathbf{V}, x) = F$

— Pour $x \in E, g, d \in \mathcal{A}_E, \mathbf{contient}(\mathbf{N}(x, g, d), x) = T$

— Pour $x, y \in E$ avec $x < y$, pour $g, d \in \mathcal{A}_E, \mathbf{contient}(\mathbf{N}(y, g, d), x) = \mathbf{contient}(g, x)$

— Pour $x, y \in E$ avec $x > y$, pour $g, d \in \mathcal{A}_E, \mathbf{contient}(\mathbf{N}(y, g, d), x) = \mathbf{contient}(d, x)$

En d'autres termes, pour chercher x dans A, on regarde si x est plus grand ou plus petit que la racine, et on chercher récursivement à gauche ou à droite, jusqu'à atteindre un arbre vide ou un nœud étiqueté par l'élément recherché.

Terminaison Les appels récursifs causés par la fonction **contient** se font sur un des deux enfants de l'arbre, et la fonction est bien définie sur les arbres vides : elle termine.

Correction Montrons que pour A un arbre et $x \in E$, $\mathbf{contient}(A, x)$ vaut T si et seulement si A contient x. Procédons par induction sur les arbres binaires :

- Si $A = V$ est l'arbre vide, alors $\mathbf{ajout}(A, x) = F$ et A ne contient pas x
- Si $A = N(y, g, d)$, alors :
  — Si $x = y$: $\mathbf{recherche}(A, x) = T$ et A contient bien x.
  — Si $x < y$: $x \in A \Leftrightarrow x \in g$. Et par hypothèse d'induction, $\mathbf{contient}(g, x) = T \Leftrightarrow x \in g$. Donc, $x \in A \Leftrightarrow \mathbf{contient}(A, x) = T$.
  — Si $x > y$: C'est analogue

Complexité Remarquons que les appels récursifs se font systématiquement sur des arbres de hauteurs strictement décroissantes. De plus, chaque appel a un coût constant en dehors des appels récursifs. Ainsi, en notant $C(h)$ le coût de la recherche d'un élément dans un arbre de hauteur h, on a $C(h) \leq C(h-1) + \mathcal{O}(1)$. Donc, la fonction est en $\mathcal{O}(h)$ avec h la hauteur de l'arbre, là où la fonction de recherche que l'on avait écrit pour les arbres binaires quelconque était en $\mathcal{O}(n)$.

C Ajout

Lorsque l'on ajoute un nouvel élément dans un arbre binaire, le plus simple est de créer une nouvelle feuille, en suivant le chemin que l'on obtient lorsqu'on le recherche.

Exemple 12. Schéma tableau

On définit donc inductivement la fonction **ajout** : $\mathcal{A}_E \times E \to \mathcal{A}_E$ comme suit :

— Pour $x \in E$, $\mathbf{ajout}(x, V) = N(x, V, V)$

— Pour $x \in E, g, d \in \mathcal{A}_\mathcal{E}, \mathbf{ajout}(\mathbf{N}(x, g, d), x) = N(x, g, d)$

— Pour $x, y \in E$ avec $x < y$, pour $g, d \in \mathcal{A}_\mathcal{E}, \mathbf{ajout}(\mathbf{N}(y, g, d), x) = N(y, \mathbf{ajout}(g, x), d)$

— Pour $x, y \in E$ avec $x > y$, pour $g, d \in \mathcal{A}_\mathcal{E}, \mathbf{ajout}(\mathbf{N}(y, g, d), x) = N(y, g, \mathbf{ajout}(d, x))$

Terminaison Les appels récursifs causés par la fonction **ajout** se font sur un des deux enfants de l'arbre, et la fonction est bien définie sur les arbres vides : elle termine.

Correction Montrons que pour A un arbre et $x \in E$, $\mathbf{ajout}(A, x)$ est un ABR contenant les étiquettes de A et x. On procède par induction structurelle sur les arbres :

- Si $A = V$ est l'arbre vide, alors $\mathbf{ajout}(A, x) = N(x, V, V)$ contient bien x ainsi que les étiquettes de A.
- Si $A = N(y, g, d)$, alors :
  - Si $x = y$: $\mathbf{ajout}(A, x) = A$ et A contient déjà x.
  - Si $x < y$: Par hypothèse d'induction, $\mathbf{ajout}(g, x)$ est un ABR contenant les étiquettes de g ainsi que x. Donc, toutes les étiquettes de g sont inférieures à y, et donc $N(y, \mathbf{ajout}(g, x), d)$ est un ABR et contient bien les étiquettes de A, i.e. y et les étiquettes de g et d, ainsi que x.
  - Si $x > y$: C'est analogue

Complexité Comme pour la recherche, la fonction **ajout** cause à chaque fois un seul appel récursif sur un arbre de hauteur strictement inférieure. Donc la complexité est en $\mathcal{O}(h)$.

D Suppression

Intéressons nous maintenant à la suppression.

Exemple 13. Considérons l'arbre binaire de recherche suivant et essayons de supprimer sa racine :

On se rend compte que l'on crée un trou dans l'arbre, qu'il faut remplir, en décalant en série d'éléments, ou bien en y mettant directement un élément adéquat.

Remarque 3. Si l'on remplace la racine d'un ABR par le plus grand élément de son sous arbre gauche, alors la structure d'ABR est conservée. Idem pour le plus petit élément de son sous-arbre droit.

Cette remarque va nous permettre d'extraire un schéma pour la suppression : pour supprimer une étiquette, on va chercher et supprimer récursivement la plus petite étiquette de son sous-arbre droit et remplacer l'étiquette supprimée par celle ci. Si le sous-arbre droit ne contient aucune étiquette, alors supprimer la racine de $N(x, g, d)$ revient à prendre l'arbre g.

On définit donc inductivement la fonction **supprimer** : $\mathcal{A}_E \times E \to \mathcal{A}_E$:

- Pour $x \in E$, $g \in \mathcal{A}_E$, **supprimer**$(N(x, g, V), x) = g$
- Pour $x \in E$, $g, d \in \mathcal{A}_E$ avec d non vide, **supprimer**$(N(x, g, d), x) = N(m, g, d')$ où :
  - m est l'étiquette min contenue dans d
  - $d' =$ **supprimer**(d, m)
- Pour $x, y \in E$ avec $x < y$ et $g, d \in \mathcal{A}_E$, **supprimer**$(N(y, g, d), x) = N(y,$ **supprimer**$(g, x), d)$
- Pour $x, y \in E$ avec $x > y$ et $g, d \in \mathcal{A}_E$, **supprimer**$(N(y, g, d), x) = N(y, g,$ **supprimer**$(d, x))$

Exemple 14. Une fois que l'on a trouvé l'étiquette minimale m du sous-arbre droit d, l'appel **supprimer**(d, m) n'est pas réellement récursif, car m n'a pas d'enfant droit, et se supprime donc facilement :

Donc, comme trouver l'étiquette min d'un arbre prend un temps $\mathcal{O}(h)$, la fonction s'effectue en $\mathcal{O}(h)$.

4 Arbres de recherche équilibrés

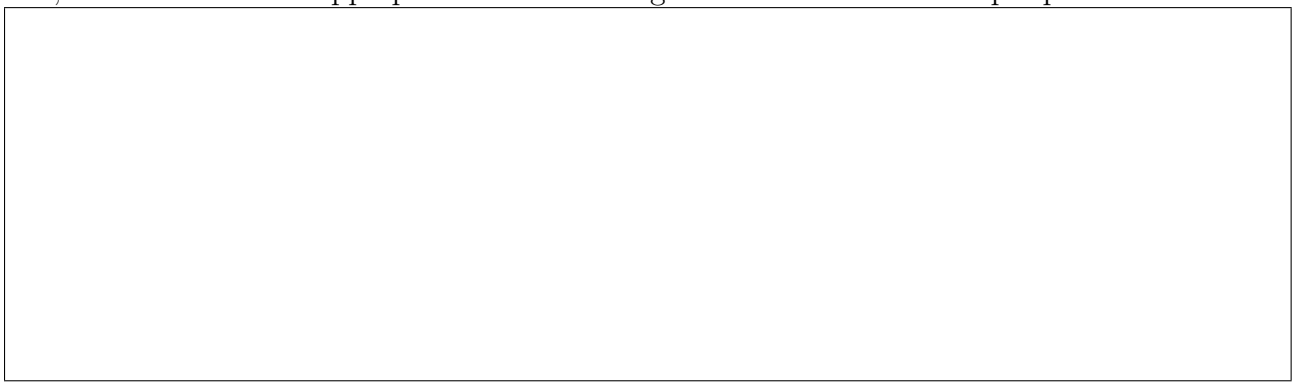
Dans cette partie, on considère des arbres dont toutes les étiquettes sont uniques, ne pouvant donc pas contenir de doublons.

Les opérations données précédemment s'effectuent en $\mathcal{O}(h)$, mais cela ne donne aucune information réelle sur la complexité si l'on a aucune contrainte sur la forme de l'arbre. En effet, nous avons vu qu'un arbre binaire de taille n a une hauteur h comprise entre $\log n$ et n.
L'objectif de cette partie est de mettre en place un type d'arbre où l'insertion et la suppression ont des mécanismes de rééquilibrage intégrés, garantissant que la hauteur est en $\mathcal{O}(\log n)$, et donc que les différentes opérations le sont aussi.

A Rotation d'arbre

Il existe plusieurs types d'arbres auto-équilibrants. Nous allons en étudier une famille classique, assez simple à implémenter : les arbres ***rouge-noir***. Ces arbres utilisent un mécanisme de rééquilibrage appelé ***rotation d'arbre***.

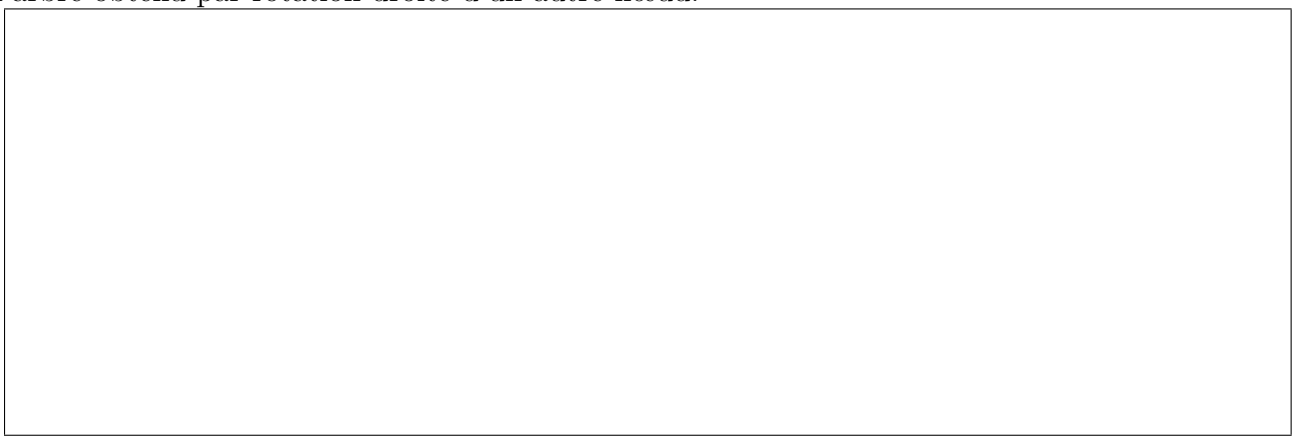
Définition 14. Soit $A = N(x, N(y, g', d'), d)$ un arbre dont l'enfant gauche est non-vide. L'arbre obtenu en appliquant une rotation droite sur A est $A' = N(y, g', N(x, d', d))$. Inversement, l'arbre obtenu en appliquant une rotation gauche sur A' est A. Graphiquement :



Notons $R_g : \mathcal{A}_E \to \mathcal{A}_E$ la rotation gauche et $R_d : \mathcal{A}_E \to \mathcal{A}_E$ la rotation droite.

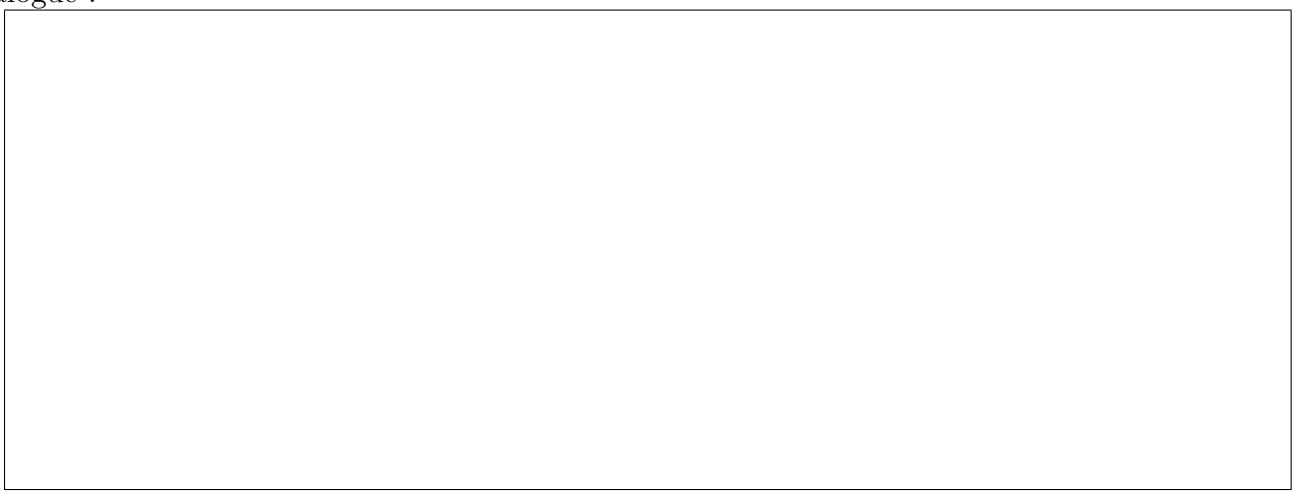
On peut appliquer les rotations gauche et droite au sein d'un arbre, sur un nœud interne quelconque. On notera $R_d^x(A)$ l'arbre obtenu en effectuant une rotation droite sur le sous-arbre enraciné en x dans A, lorsque c'est possible. On note de même $R_g^x(A)$ pour une rotation gauche.

Exemple 15. Un exemple d'arbre binaire, puis l'arbre obtenu par rotation gauche de sa racine, et l'arbre obtenu par rotation droite d'un autre nœud.



Proposition 2. Les rotations gauche et droite conservent la structure d'ABR.

Démonstration. Montrons le pour la rotation gauche, la preuve pour la rotation droite étant analogue :



□

Exercice 4. Combien de rotations gauche ou droite distinctes peut-on appliquer sur un arbre binaire à n nœuds ?

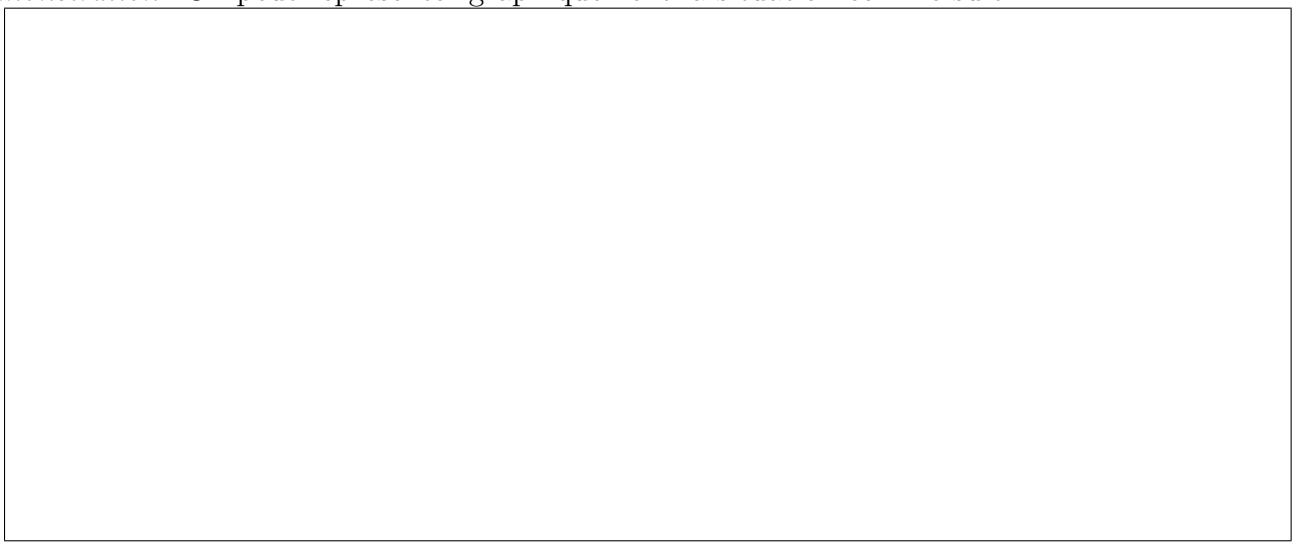
Proposition 3. Soit $A = N(x, g, d)$ avec $g = N(y, g', d')$, tel que :

— $h(g) > h(d)$ (A est déséquilibré vers la gauche)

— $h(g') > h(d')$ (g est déséquilibré vers la gauche)

Alors $h(R_d(A)) = h(A) - 1$.

Démonstration. On peut représenter graphiquement la situation comme suit :



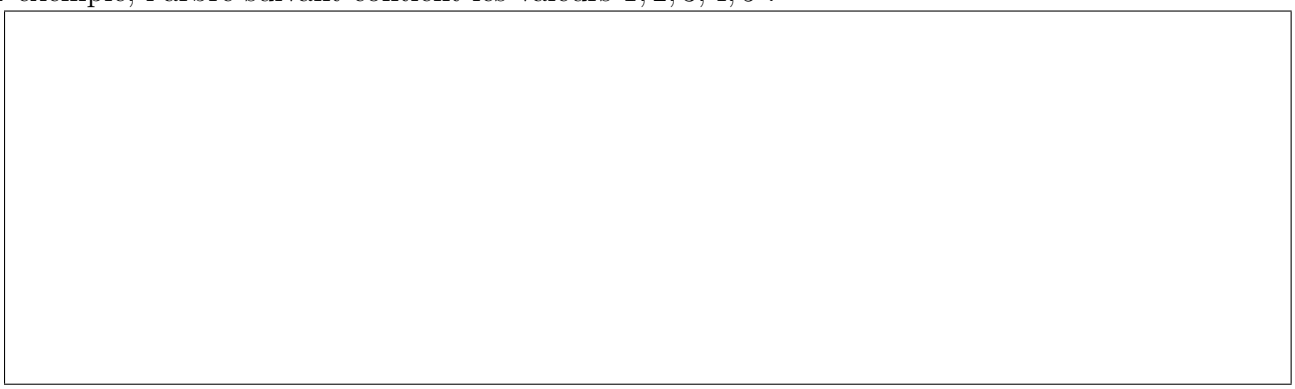
On a $h(A) = 2 + h(g')$, et $h(R_d(A)) = 1 + h(g') = h(A) - 1$. □

Les rotations servent donc à rééquilibrer les arbres, en transférant le surplus de hauteur d'un enfant vers l'autre.

Exercice 5. Aller sur perso.ens-lyon.fr/guillaume.rousseau/mp2i/demos/tree_rotation/ et utilisez des rotations d'arbres pour équilibrer les arbres binaires proposés. Cliquez sur un nœud pour le sélectionner puis utilisez les touches gauche et droite du clavier pour appliquer une rotation.

B Arbres rouge-noir

On considère maintenant des arbres de recherche binaires stricts, où les informations sont stockées uniquement dans les feuilles. Les nœuds internes serviront donc seulement d'aiguillage. Par exemple, l'arbre suivant contient les valeurs $1, 2, 3, 4, 5$:



On considère donc des arbres binaires stricts. On note $\mathbf{ABS}_E$ l'ensemble des arbres binaires stricts étiquetés par E. On rappelle que cet ensemble est construit inductivement comme suit :

— $\mathbf{F}(e) \in \mathbf{ABS}_E$ pour $e \in E$: on appelle ces arbres les "feuilles"

— $\mathbf{N}(e, g, d) \in \mathbf{ABS}_E$ pour $e \in E$ et $g, d \in \mathbf{ABS}_E$: on appelle ces arbres les "nœuds internes"

Dans un arbre binaire de recherche implémenté avec ce type d'arbres, les éléments de l'arbre sont les étiquettes des feuilles, et pour un arbre $\mathbf{N}(x, g, d)$, les éléments de g sont inférieurs où égaux à x, et les éléments de d sont strictement supérieurs à x.

Exercice 6. Montrer qu'un arbre de recherche binaire strict a forcément des nœuds internes tous distincts, et des feuilles toutes distinctes.

Voyons comment la procédure d'insertion s'écrit sur ces arbres :

— $\mathbf{insert}(F(x), y) = N(x, F(x), F(y))$ si $x < y$

— $\mathbf{insert}(F(x), y) = N(y, F(y), F(x))$ si $y < x$

— $\mathbf{insert}(F(x), x) = F(x)$ (car alors x est déjà dans A).

— $\mathbf{insert}(N(x, g, d), y) = N(x, \mathbf{insert}(g, y), d)$ si $y \leq x$

— $\mathbf{insert}(N(x, g, d), y) = N(x, g, \mathbf{insert}(d, y))$ si $x < y$

Définition 15. Un ***arbre rouge-noir*** (ou ARN) est un arbre binaire de recherche dont les nœuds ont été additionnellement étiquetés par une couleur, rouge ou noir, de telle sorte que :

(i) Si un nœud est rouge, ses deux enfants sont noirs

(ii) Tous les chemins entre la racine et les feuilles ont le même nombre de nœuds noirs.

(iii) La racine est noire

(iv) Les feuilles sont noires

On dit qu'un arbre est un arbre rouge-noir relaxé s'il vérifie les conditions (i), (ii) et (iv). Remarquons qu'un sous-arbre d'un ARN relaxé est aussi un ARN relaxé, car les propriétés autres que la (iii) se transmettent inductivement aux enfants.

On notera $N(c, x, g, d)$ un nœud interne de couleur c.

Exemple 16. Quelques ARN valides, et quelques arbres qui ne vérifient pas les 4 conditions de la définition :

Le principe des ARN réside dans les observations suivantes :

— La condition (i) assure qu'il n'y a pas trop de nœuds rouges dans l'arbre : sur un chemin donné, au plus la moitié des nœuds sont rouges.

— La condition (ii) est une condition d'équilibre : elle restreint la position des feuilles en imposant un nombre minimum de nœuds entre les feuilles et la racine.

Les conditions (iii) et (iv) sont moins essentielles, mais simplifient la manipulation et les preuves.

La propriété clé des ARN est que leur hauteur est forcément logarithmique en leur taille :

Proposition 4. Soit A un ARN de hauteur h, de taille n. Alors, $h \leq 2\log_2(n+1)$

Pour montrer cette propriété, il faut introduire quelques notions et lemmes.

Définition 16. Soit A un ARN relaxé. On appelle ***hauteur noire*** de A le nombre de nœuds internes noirs entre la racine de A et toute feuille. On la note $\mathbf{bh}(A)$.

Exemple 17. Quelques ARN relaxés, et leur hauteur noire à chaque fois :

Lemme 1. Tout ARN relaxé A contient au moins $2^{1+\mathbf{bh}(A)} - 1$ nœuds.

Montrons-le par induction sur les ARN relaxés.

— Si $A = F(e)$ est une feuille : Alors $2^{1+\mathbf{bh}(A)} - 1 = 1$, et A contient 1 nœud.

— Si $A = N(c, x, g, d)$: $\mathbf{bh}(A)$ vaut soit $\mathbf{bh}(g)$ soit $\mathbf{bh}(g) + 1$ selon si c vaut Rouge ou Noir. Donc, $\mathbf{bh}(g) \geq \mathbf{bh}(A) - 1$, et donc par hypothèse d'induction, g contient au moins $2^{\mathbf{bh}(A)-1+1} - 1$ nœuds. Il en va de même pour d, d'où A contient au moins $1+2(2^{\mathbf{bh}(A)-1+1} - 1) = 2^{\mathbf{bh}(A)+1} - 1$ nœuds.

Montrons maintenant la propriété énoncée : si A est un ARN de hauteur h et de taille n, alors $h \leq 2\log_2(n+1)$

Démonstration. Comme tout nœud rouge doit avoir uniquement des enfants noirs, et que la racine et les feuilles sont noires, au moins la moitié des nœuds d'un chemin entre la racine de A et une feuille sont noirs. Autrement dit, $h \leq 2\mathbf{bh}(A)$. Donc, $n \geq 2^{\mathbf{bh}(A)+1} - 1 \geq 2^{\frac{h}{2}+1} - 1$. En prenant le logarithme, on obtient : $\frac{h}{2} + 1 \leq \log_2(n+1)$ et donc $h \leq 2\log_2(n+1) - 2 \leq 2\log_2(n+1)$. □

Exercice 7. Quel est le plus grand nombre de nœuds d'un arbre rouge noir avec une hauteur noire de k ? Et le plus petit ?

Il reste à voir comment modifier les opérations d'insertion et de suppression pour pouvoir conserver la structure des ARN, tout en gardant une complexité logarithmique.

C Insertion

Pour insérer un nouvel élément dans un ARN, le principe est d'effectuer une insertion classique sur les arbres binaires stricts de recherche, en colorant le nouveau nœud créé en rouge. Cette insertion peut créer une anomalie dans l'arbre : on peut en effet se retrouver dans une des deux situations suivantes :

— La racine est rouge (si l'arbre initial n'avait qu'une feuille)

— Deux nœuds consécutifs sont rouges

Les règles (i) et (iii) peuvent donc être brisées. En revanche, après l'insertion, les règles (ii) et (iv) sont toujours valides : on n'a pas ajouté de feuille rouge, et l'on n'a pas ajouté de nœud noir donc la hauteur noire est toujours bien définie. Nous allons montrer qu'avec des bons choix de rotations, on peut régler l'anomalie. Plus précisément, les rotations vont permettre de transférer l'anomalie d'un nœud vers son parent, et ce jusqu'à atteindre la racine.

Exemple 18. Faisons à la main quelques insertions et tentons de corriger l'arbre obtenu pour le rendre à nouveau rouge-noir. Cf annexe A

Ces différents exemples donnent une idée plus précise de la procédure d'insertion dans les arbres rouge-noir. Plus précisément, on aura une procédure **correctionARN** qui servira à faire remonter un problème de nœuds rouges vers la racine, et une procédure **insertion** qui sera définie comme :

— $\textbf{insertion}(N(c, x, g, d), y) = \textbf{correctionARN}(N(c, x, \textbf{insertion}(g, y), d))$ si $y < x$

— $\textbf{insertion}(N(c, x, g, d), y) = \textbf{correctionARN}(N(x, g, \textbf{insertion}(d, y)))$ si $x < y$

— $\textbf{insertion}(F(y), x) = N(\textbf{Rouge}, x, F(x), F(y))$ si $x < y$

— $\textbf{insertion}(F(y), x) = N(\textbf{Rouge}, y, F(y), F(x))$ si $y < x$

— $\textbf{insertion}(A, x) = A$ dans les autres cas (car alors x est déjà dans A)

On peut voir cette définition comme ayant deux phases : une phase de descente pour trouver l'endroit où insérer le nouvel élément, et une phase de remontée qui corrige les nœuds jusqu'à atteindre la racine.

La fonction **correctionARN** aura comme spécification :

— Pré-condition : A est un ARN relaxé, dont un enfant est rouge et a un enfant rouge. Tous les autres nœuds de A sont corrects du point de vue des règles des ARN.

— Post-condition : $\textbf{corretionARN}(A)$ est un ARN relaxé de même hauteur noire que A.

Notons que comme à part l'enfant et le petit-enfant, l'arbre ne possède aucune anomalie, la racine est forcément noire, puisqu'elle a un enfant rouge. On considère donc un arbre A, et on a 4 cas à considérer : l'enfant de la racine peut être à gauche ou à droite, et le petit-enfant problématique peut être à gauche ou à droite également. Dans chaque cas, on rééquilibre à l'aide d'une ou deux rotations l'arbre, puis on recolore les nœuds pour obtenir un ARN :

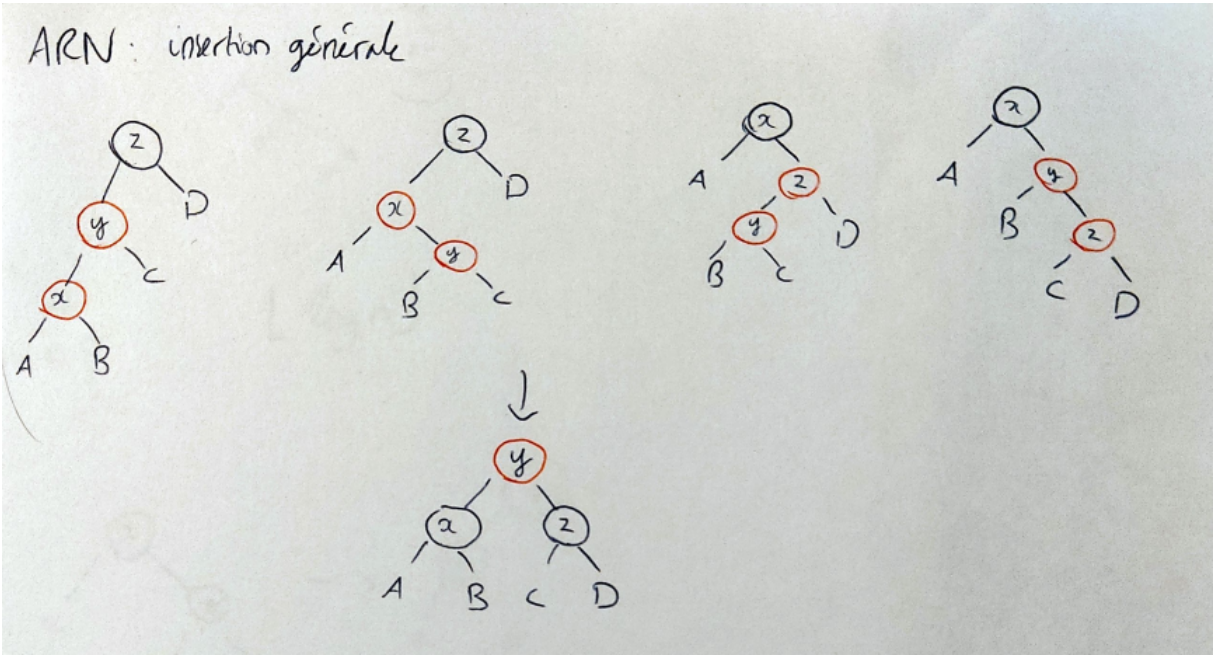


FIGURE 1 – Définition de **correctionARN**

La hauteur noire ne change dans aucun cas.

A la toute fin de la procédure d'insertion, on obtient un ARN relaxé : si la racine est rouge, il faut la recolorer en noir. C'est à ce moment **uniquement** que l'on augmente potentiellement la hauteur noire.

Complexité La fonction **correctionARN** s'effectue en $\mathcal{O}(1)$, donc la fonction **insertion** fait un nombre constant d'opérations à part l'appel récursif. Donc, comme pour l'insertion dans un arbre binaire de recherche quelconque, la complexité est en $\mathcal{O}(h) = \mathcal{O}(\log n)$ car on considère des arbres rouge-noir : l'insertion se fait en temps logarithmique !

D Suppression

Le principe pour la suppression est le même : on commence par effectuer une suppression d'ABR classique, puis on corrige les éventuelles anomalies.

Commençons par voir comment supprimer un élément dans un ABR dont les informations sont stockées aux feuilles, sans rééquilibrage. La procédure sera plus simple que dans les ABR dont les informations sont stockées sur tous les nœuds. En effet, pour supprimer une feuille, il suffit de remplacer le parent de cette feuille, p, par l'autre enfant de p :

— $\mathbf{deleteABR}(N(y, F(x), d), x) = d$
— $\mathbf{deleteABR}(N(y, g, F(x)), x) = g$
— $\mathbf{deleteABR}(N(y, g, d), x) = N(y, \mathbf{deleteABR}(g, x), d)$ si $x \leq y$
— $\mathbf{deleteABR}(N(y, g, d), x) = N(y, g, \mathbf{deleteABR}(d, x))$ si $x > y$

Dans les autres cas, la fonction n'est pas définie. En particulier, cette fonction ne peut pas supprimer l'unique étiquette d'un arbre-feuille. En effet, on ne peut pas représenter l'arbre vide directement avec les constructeurs que l'on utilise pour les arbres stricts. On ne peut pas simplement rajouter un constructeur **V** pour l'arbre vide, car on ne veut pas pouvoir écrire par exemple $N(\mathbf{Rouge}, x, V, V)$. En revanche, on peut diviser le type des ARN en deux étages :

```
type 'a noeud_arn =
  | Feuille of 'a
  | NoeudInterne of couleur * 'a * 'a noeud_arn * 'a noeud_arn

type 'a arn = noeud_arn option (* None pour l'arbre vide, Some a pour l'arbre non-vide a *)
```

Le type `'a arn` signifie donc qu'un arn est soit un arbre vide (représenté par `None`), soit un arbre non vide dont la racine est un nœud. En hiérarchisant ainsi la situation, on évite ainsi de pouvoir avoir des arbres vides apparaissant au sein d'un ARN.

Intéressons-nous maintenant à la correction des anomalies. Après avoir supprimé un élément d'un ARN, il peut s'être passé deux choses : soit on a supprimé un nœud rouge, auquel cas les conditions d'ARN sont encore vérifiées, soit on a supprimé un nœud noir, auquel cas on peut avoir une anomalie au niveau du nombre de nœuds noirs sur un chemin vers une feuille. On dira qu'il y a un **défaut de nœud noir**.

Exemple 19. Au tableau : quelques suppressions dans des ARN.

Finalement, le schéma pour la suppression est le même que pour l'insertion : dans une première phase, on descend dans l'arbre pour supprimer un noeud, puis on remonte en appliquant une procédure de correction. Ici, cette fonction de correction aura pour spécification :

— Précondition : $A = N(x, g, d)$ est un arbre tels que g et d sont des ARN relaxés, mais dont les hauteurs noires diffèrent de 1.

— Postcondition : **correctionARN2**(A) est un ARN relaxé. Sa hauteur noire vaut $\mathbf{bh}(A)$ ou $\mathbf{bh}(A) - 1$.

Notons qu'avec cette spécification, on ne corrige pas forcément l'anomalie, car on peut avoir diminué la hauteur noire, auquel cas on a simplement transféré le problème vers le parent.

Regardons la fonction de correction. Par symétrie, en considère un arbre $N(x, g, d)$ tel que la hauteur noire de g est inférieure à celle de d. Nous allons utiliser les nœuds de d pour corriger le défaut. On raisonne selon la couleur des différents nœuds de d. On sait que d est de hauteur noire au moins 1, donc contient au moins 1 nœud interne. On peut donc jouer sur 4 nœuds : la racine, la racine de d, et les deux enfants de d. On a donc potentiellement $2^4 = 16$ cas possible. En réalité, on en aura 9 (dont un avec 4 sous-cas), car certains cas sont interdits par les règles des ARN (ex : tous les nœuds sont rouges)

Voir Annexe B : suppression dans les ARN.

Tout comme l'insertion, la suppression se fait en $\mathcal{O}(\log n)$.

5 Tas

On cherche à implémenter la SDA de ***file de priorité***, qui permet d'extraire les éléments non pas dans leur ordre d'arrivée mais par ordre de priorité.

Définition 17. On fixe un ensemble E totalement ordonné. Une file de priorité stockant des éléments de E a comme opérations :

— Création d'une file de priorité vide

— Ajouter un élément à une file de priorité

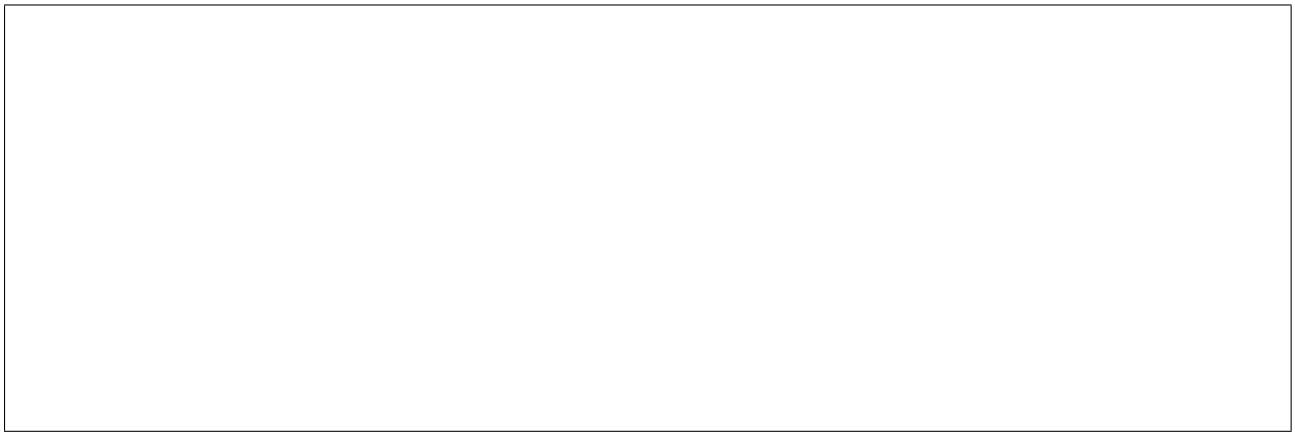
— Extraire l'élément minimal d'une file de priorité

— Déterminer si une file de priorité est vide.

Une implémentation classique de cette SDA est la structure de **tas**.

Définition 18. Un arbre binaire est dit ***complet à gauche*** si tous les niveaux de l'arbre sont pleins excepté éventuellement le dernier, et si les feuilles sont le plus à gauche possible. On dit qu'un arbre binaire est ***parfait*** si tous les niveaux de l'arbre sont pleins.

Exemple 20. Un arbre complet à gauche avec 10 noeuds, et un arbre parfait de même hauteur :



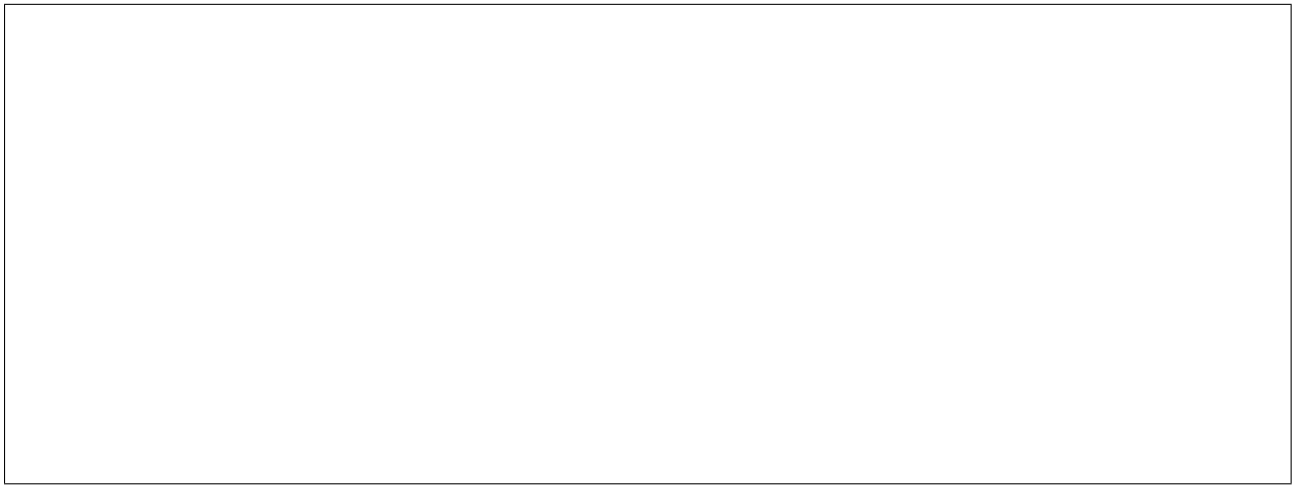
Exercice 8. Donner une relation entre la taille et la hauteur d'un arbre parfait. Donner un encadrement dans le cas d'un arbre complet à gauche.

Définition 19. Un arbre binaire A étiqueté par E est un **tas-min** s'il est complet à gauche, et si pour tout noeud $N(x, g, d)$, toutes les étiquettes de g et toutes les étiquettes de d sont plus grandes que x.

On définit de manière analogue les **tas-max**.

Autrement dit, dans un tas-min, chaque étiquette de noeud est le minimum de l'arbre enraciné correspondant. En particulier, le minimum d'un tas-min se trouve à la racine.

Exemple 21. Exemple de tas-min. Dans la suite on utilisera ce tas pour les exemples, on le notera T_0.

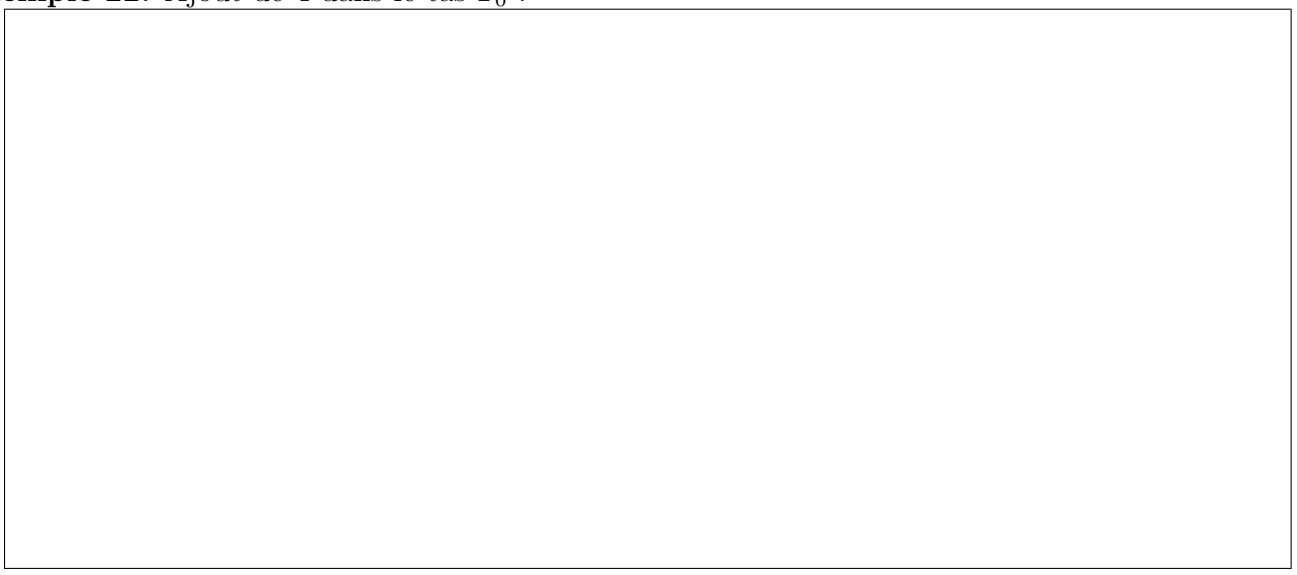


A Opérations

Dans toute la suite, le terme "tas" désigne les tas-min. Rappelons qu'un tas doit être complet à gauche, et donc que l'insertion et la suppression doivent préserver cette structure. Pour cela, les opérations vont toujours s'effectuer au niveau de la dernière feuille (la plus à droite), et échangeront les étiquettes entre cette feuille et la racine afin de réparer les éventuelles anomalies.

Ajout Soit T un tas, et x un élément à insérer. L'idée est de mettre x directement après la dernière feuille de T, puis de faire remonter x vers la racine tant que x est supérieur à l'étiquette de son parent.

Exemple 22. Ajout de 4 dans le tas T_0 :



Donnons maintenant l'algorithme correspondant. Dans la suite, on notera $f.e$ l'étiquette d'un noeud f. De plus, on notera **parent**(f) le parent de f, et si f est la racine alors on posera par convention que **parent**$(f) = f$.

Algorithme 3 : Ajout dans un tas

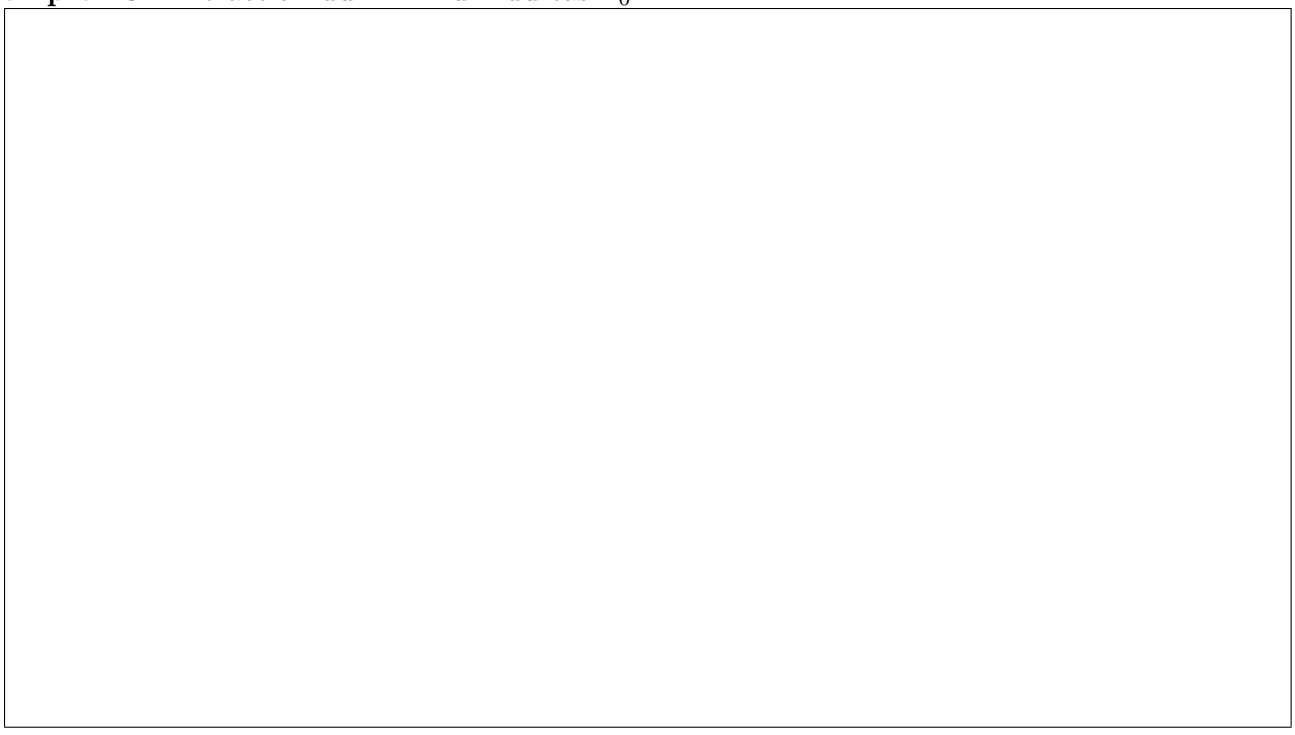
Entrée(s) : T un tas, x un élément à insérer
Sortie(s) : x a été inséré dans T, et T reste un tas

```
1 p ← premier noeud de T n'ayant pas 2 enfants;
2 si p n'a pas d'enfant gauche alors
3 |   Ajouter à T une nouvelle feuille f, enfant gauche de p;
4 sinon
  |   // p n'a pas d'enfant droit
5 |   Ajouter à T une nouvelle feuille f, enfant droit de p;
6 tant que f n'est pas la racine et f.e < p.e faire
7 |   Échanger f.e et p.e;
8 |   f ← p;
9 |   p ← parent(f)
```

Le coût de cette opération va dépendre de l'efficacité avec laquelle on peut trouver le premier nœud n'ayant pas 2 enfants. Remarquons qu'après avoir trouvé ce nœud, le reste de l'algorithme s'exécute en $\mathcal{O}(h) = \mathcal{O}(\log n)$, car la boucle while s'exécute au plus $\lfloor \log n \rfloor$ fois. En effet, à chaque tour de boucle, le nœud f remonte d'un niveau de profondeur dans l'arbre, et au départ f est de profondeur h.

Extraction Pour extraire le minimum d'un tas, il faut supprimer sa racine. Pour cela, on remplace l'étiquette de la racine par celle de la dernière feuille, que l'on supprime. On corrige ensuite les anomalies en faisant redescendre la nouvelle racine vers son bon emplacement.

Exemple 23. Extraction du minimum du tas T_0 :



Voyons l'algorithme correspondant. Dans la suite, on notera $n.g$ l'enfant gauche d'un noeud n, et $n.d$ son enfant droit.

Algorithme 4 : Extraction de la racine

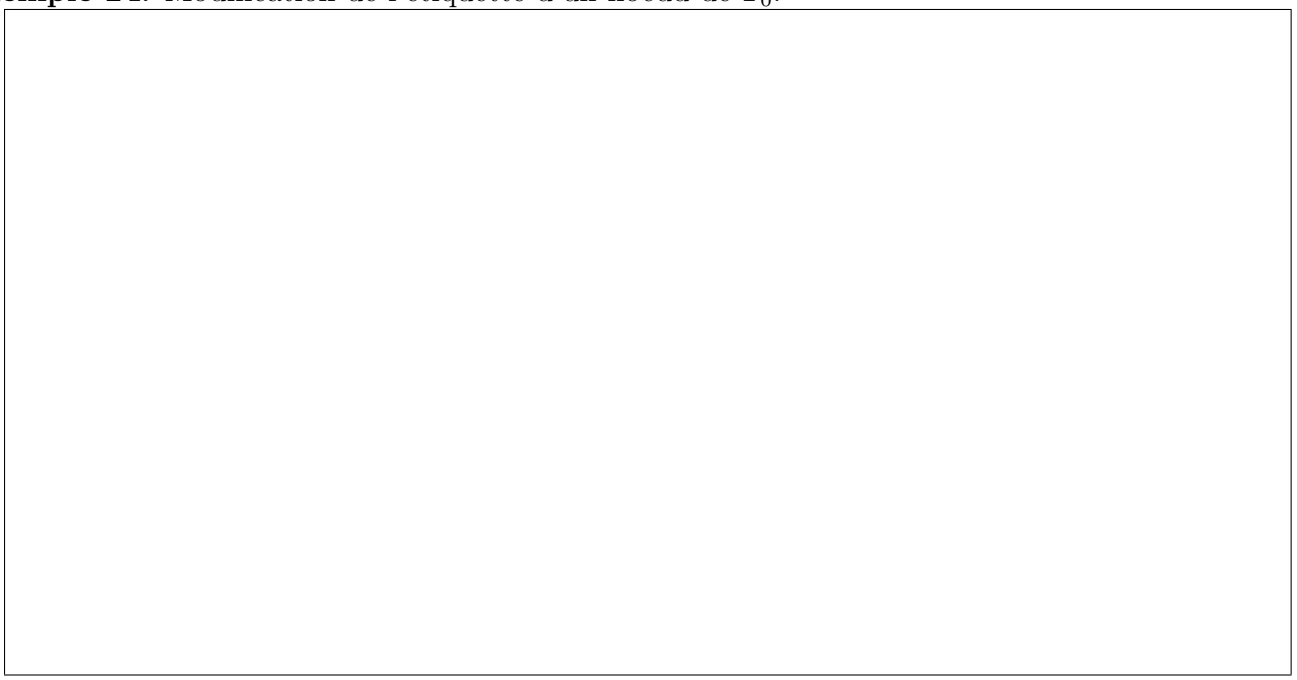
Entrée(s) : T un tas
Sortie(s) : e étiquette de la racine de T. La racine a été supprimée, et T reste un tas

```
1  r ← la racine de T ;
2  e ← r.e;
3  n ← dernière feuille de T ;
4  r.e ← n.e;
5  Supprimer n de T;
6  tant que r.e > r.g.e ou r.e > r.d.e faire
7      si r.g.e > r.d.e alors
8          Échanger r.e et r.d.e;
9          r ← d;
10     sinon
11         Échanger r.e et r.g.e;
12         r ← d;
13 retourner e
```

La complexité dépend à nouveau de l'efficacité avec laquelle on peut récupérer la dernière feuille. Si l'on est capable de le faire en $\mathcal{O}(\log n)$, alors la complexité totale est $\mathcal{O}(\log n)$, car la boucle while s'exécute au plus $\lfloor \log n \rfloor$ fois, puisque le noeud r a une profondeur qui augmente de 1 à chaque fois.

Augmentation de priorité Dans une file de priorité, on ajoute parfois l'opération "augmenter la priorité d'un élément", i.e. diminuer l'étiquette d'un noeud. Avec l'implémentation par tas, pour diminuer l'étiquette d'un noeud, on peut réutiliser le même principe que pour la procédure d'insertion, et faire remonter le noeud vers la racine jusqu'à ce qu'il soit bien placé.

Exemple 24. Modification de l'étiquette d'un noeud de T_0.

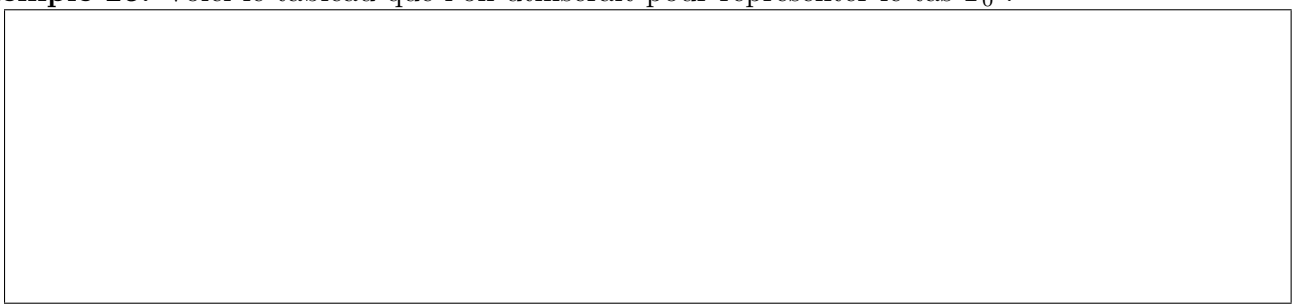


B Implémentation des tas par tableau

Pour implémenter efficacement ces opérations, il faudra donc pouvoir accéder facilement à la dernière feuille, ainsi qu'aux parents des noeuds. Avec les implémentations des arbres que nous avons vu, ça sera compliqué à mettre en place. On introduit donc une nouvelle manière d'implémenter les arbres, qui sera efficace dans le cas des arbres complets à gauche : l'implémentation par tableau.

On stocke les étiquettes de l'arbre dans un tableau. On stocke la racine dans la case d'indice 0, les deux enfants de la racine dans les cases 1 et 2, les quatre petits-enfants de la racine dans les cases 3, 4, 5 et 6, etc... De manière générale, on stocke les noeuds de profondeur k entre les indices $2^k - 1$ et $2^{k+1} - 2$ inclus. De cette manière, on accède aux enfants du noeud à la case i en allant aux cases $2i + 1$ et $2i + 2$. De même, on accède au parent du noeud d'indice i en allant à la case $\lfloor \frac{i}{2} \rfloor$.

Exemple 25. Voici le tableau que l'on utiliserait pour représenter le tas T_0 :



En C, on pourra donc utiliser la structure suivante :

```
1 // T est un type quelconque, celui des éléments àstocker
2 typedef struct tas {
3   T* tab;
4   int taille_max;
5   int taille_actuelle;
6 } HEAP;
```

L'attribut `taille_max` sera le nombre maximal de cases du tableau `tab`, autrement dit le nombre de cases mémoires réservées, et l'attribut `taille_actuelle` sera la taille effective du tas, c'est à dire le nombre d'éléments stockés à un moment donné. Ainsi, la case d'indice `taille_actuelle` sera la case à remplir en cas d'ajout.

Remarque 4. On peut adapter toute cette partie avec le principe des tableaux redimensionnables, afin de faire varier la taille de la zone allouée en fonction du remplissage réel du tas. Par souci de simplicité, on se contentera de donner ici le code dans le cas d'un tableau statique.

Exemple 26. Voici la représentation en mémoire du tableau contenant T_0, et son évolution après une insertion

Opérations Voyons l'ajout maintenant, et le reste en TP. On traduit simplement l'algorithme donné dans la première partie, en traduisant les notions sur les arbres en notions sur les indices du tableau. Par exemple, la case d'indice `t->taille_actuelle` est celle de la prochaine feuille à ajouter.

Nous allons séparer la procédure d'ajout en deux. D'abord, une fonction qui permettra de faire remonter un nœud mal placé vers la racine, et que l'on pourra réutiliser pour coder l'augmentation de priorité. On appellera cette fonction **tasser**.

```
1  /* Corrige t en le tassant àpartir du noeud d'indice i pour en faire un tas
2  Préconditions: - l'arbre dont i est la racine est un tas
3              - l'étiquette du parent de i est inférieure
4                à toutes les étiquettes de l'arbre enraciné
5                en i, sauf éventuellement celle de i lui-même
6  Postcondition: t est un tas                                  */
7  void tasser(HEAP* t, int i){
8    p = i/2; // pere de i
9    while (i > 0 && t->tab[i] < t->tab[p]){
10     T tmp = t->tab[i];
11     t->tab[i] = t->tab[p]
12     t->tab[p] = tmp;
13     i = p;
14     p = i/2; // division entière
15   }
16 }
```

Cette procédure correspond donc à la boucle while de l'algorithme d'ajout décrit plus haut.

On utilise ensuite cette procédure pour l'ajout : on crée une nouvelle feuille contenant l'élément à rajouter, et on tasse à partir de cette feuille.

```
1  /* Ajoute x au tas t. T ne doit pas être plein */
2  void ajout(HEAP* t, T x){
3    assert(t->taille_actuelle < t->taille_max);
4
5    // cas particulier: tas vide
6    if (t->taille_actuelle == 0){
7      t->tab[0] = x;
8      t->taille_actuelle++;
9      return;
10   }
11
12   // nouveau noeud et son parent
13   int f = t->taille_actuelle;
14   int p = f/2;
15
16
17   // ajout du nouvel élément
18   t->tab[f] = x;
19   t->taille_actuelle++;
20
21   tasser(t, f);
22 }
```

Pour l'extraction, on suivra le même schéma : on crée une fonction auxiliaire (que l'on appelle généralement **tamiser**) qui permettra ensuite de faire remonter la racine afin de la placer au bon endroit :

```
1  /* Corrige t en le tamisant àpartir du noeud d'indice i pour
2     en faire un tas
3  Précondition: Les deux sous-arbres de i sont des tas
4  Postcondition: t est un tas                              */
5  void tamiser(struct tas* t, int i){
6    ...
7  }
8
9  /* Extrait le minimum de t et retransforme t en tas. Renvoie l'étiquette extraite. */
10 T extraire_min(struct tas* t){
11   assert(t->taille_actuelle > 0);
12
13   // récupérer la dernière feuille et l'enlever
14   f = t->taille_actuelle - 1;
15   t->taille_actuelle--;
16
17   // échanger les étiquettes de la racine et de la feuille enlevée
18   T res = t->tab[0];
19   t->tab[0] = t->tab[f];
20   t->tab[f] = res;
21
22   tamiser(t, 0);
23   return res;
24 }
```

Il peut sembler étrange d'échanger l'étiquette de la racine avec celle de la feuille enlevée plutôt que de simplement écraser la valeur de la racine, mais cela permet de conserver tous les éléments, et nous allons voir que cette propriété permet d'écrire un algorithme de tas très élégant.

Complexité L'ajout, l'extraction et la modification de priorité sont en $\mathcal{O}(\log n)$. C'est mieux qu'une implémentation naïve des files de priorité avec les listes, où les opérations sont en $\mathcal{O}(n)$.

Il existe une implémentation concrète des files de priorité appelée les **tas de Fibonacci** permettant l'insertion en $\mathcal{O}(1)$ et l'extraction en $\mathcal{O}(\log n)$ en complexité amortie, et permettant d'augmenter la priorité d'un élément en $\mathcal{O}(1)$ en complexité amortie.

C Tri par tas

La procédure d'ajout sert à étendre un tas : si $A[0..i[$ est un tas alors après avoir appelé **ajout**(A, x), $A[0..i+1[$ sera un tas. Si l'on appelle simplement **tasser**(A, i) à la place, cela revient à ajouter $A[i]$ au tas stocké dans $A[0..i[$.

Inversement, la procédure d'extraction de min permet de réduire la taille d'un tas, sans détruire d'élément : si $A[0..i[$ est un tas, alors après avoir appelé **extraire_min**(A), $A[0..i-1[$ est un tas, et $A[i]$ contient l'ancienne valeur de $A[0]$, qui était le minimum de $A[0..i[$.

On peut donc utiliser ces deux procédures pour implémenter un algorithme de **tri par tas**, dont le principe est assez proche du tri par insertion : On sépare le tableau A à trier en deux parties : la partie gauche de A servira à stocker les éléments d'un tas, et la partie droite de A servira à stocker les autres éléments du tableau. Au départ, le tas est vide, et l'on insère dedans tous les éléments de A un par un, en appelant **tasser**(A, i) pour i allant de 0 à $|A|-1$. Après

cette étape, A tout entier est un tas. On peut donc extraire le minimum du tas $|A|$ fois d'affilée, ce qui aura pour effet de stocker les éléments dans l'ordre décroissant dans la partie droite de A.

Exemple 27. On considère $A = [5, 2, 7, 3, 4, 1]$. On suppose que l'on utilise A pour stocker un tas, et qu'initialement le tas est vide. Donc, les cases de A, bien que remplies de données, ne font pas partie du tas. Si l'on appelle **tasser**$(A, 1)$, alors tout se passe comme si on insérait $A[1] = 2$ dans le tas $A[0..1[$. Après cette étape, $A = [2, 5, 7, 3, 4, 1]$. Ensuite, si l'on appelle **tasser**$(A, 2)$, tout se passe comme si l'on insérait $A[2] = 7$ dans le tas $A[0..2[$, et donc après cette étape on a $A = [2, 5, 7, 3, 4, 1]$. Voici les valeurs successives de A après les appels successifs à **tasser**(A, i) :

	État de A	Taille du tas
Initialement	$[5, 2, 7, 3, 4, 1]$	0
Après **tasser**$(A, 0)$	$[5, 2, 7, 3, 4, 1]$	1
Après **tasser**$(A, 1)$	$[2, 5, 7, 3, 4, 1]$	2
Après **tasser**$(A, 2)$	$[2, 5, 7, 3, 4, 1]$	3
Après **tasser**$(A, 3)$	$[2, 3, 7, 5, 4, 1]$	4
Après **tasser**$(A, 4)$	$[2, 3, 7, 5, 4, 1]$	5
Après **tasser**$(A, 5)$	$[1, 3, 2, 5, 4, 7]$	6

Après le i-ème tassage, $A[0..i[$ contient un tas, et $A[i..6[$ contient les éléments restants de A. Donc, après le dernier tassage, A est un tas. A partir de ce nouveau tableau, observons ce qu'il se passe si l'on effectue 5 extractions :

	État de A	Taille du tas
Initialement	$[1, 3, 2, 5, 4, 7]$	6
Après **tamiser**$(A, 0)$	$[2, 3, 7, 5, 4, 1]$	5
Après **tamiser**$(A, 0)$	$[3, 4, 7, 5, 2, 1]$	4
Après **tamiser**$(A, 0)$	$[4, 5, 7, 3, 2, 1]$	3
Après **tamiser**$(A, 0)$	$[5, 7, 4, 3, 2, 1]$	2
Après **tamiser**$(A, 0)$	$[7, 5, 4, 3, 2, 1]$	1

Ainsi, après tamisé i fois, $A[0..i[$ contient un tas, et $A[i..6[$ contient les plus petits éléments de A dans l'ordre décroissant. Donc, après le dernier tamisage, A est trié dans l'ordre décroissant.

Le tri par tas peut donc s'exprimer ainsi :

Algorithme 5 : Tri par tas

Entrée(s) : A un tableau de taille n
Sortie(s) : A est trié, en place

```
1 T ← tas vide, stocké dans le tableau A;
2 tant que T.taille < n faire
3 |   tasser(T, T.taille);
4 |   T.taille ← T.taille + 1;
5 tant que T.taille > 0 faire
6 |   T.taille ← T.taille − 1;
7 |   Échanger A[T.taille] et A[0];
8 |   tamiser(T, T.taille);
```

Terminaison Les deux boucles sont des boucles for cachées : l'algorithme termine.

Correction Pour montrer la correction de l'algorithme, on peut commencer par montrer qu'après la première boucle while, A contient un tas. Pour cela, si l'on admet que les éléments de A ne sont jamais supprimés ou modifiés, mais uniquement échangés, l'invariant suivant convient : "T est un tas contenant $T.taille$ éléments". En effet :

— En entrée de boucle, T est un tas vide, il contient bien 0 éléments.

— Si au début d'un passage de boucle, T est un tas et contient bien $T.taille$ éléments, alors il est stocké dans les $T.taille$ premières cases de A. Donc, **tasser**$(T, T.taille)$ revient exactement à ajouter $A[T.taille]$ dans T : T est toujours un tas en fin de passage et sa taille a bien augmenté de 1.

En particulier à la fin de la boucle, T est un tas de taille n. On montre de même que la deuxième boucle a comme invariant "T est un tas et $A[T.taille..n[$ contient les $n - T.taille - 1$ plus petits éléments de A dans l'ordre décroissant.", et donc qu'à la fin de l'algorithme, A est trié par ordre décroissant.

Complexité La complexité de ce tri est en $\mathcal{O}(n \log n)$ car les deux boucles s'exécutent n fois chacune, et chaque passage coûte $\log n$.

Une remarque : la première partie de l'algorithme, consistant à construire le tas, peut s'effectuer en $\mathcal{O}(n)$ (c'est plus difficile à faire). Pour le tri par tas cela n'a aucun impact sur la complexité théorique car la deuxième boucle reste en $\mathcal{O}(n \log n)$, mais cette construction peut être utile dans d'autres situations. Par exemple, si l'on veut obtenir les k plus petits éléments d'un tableau, alors on peut transformer le tableau en tas min en $\mathcal{O}(n)$, puis extraire le min k fois d'affilée en $\mathcal{O}(\log n)$. Ceci donne une complexité en $\mathcal{O}(n + k \log n)$, plus efficace que la méthode consistant à trier le tableau puis récupérer les k premiers éléments, qui est en $\mathcal{O}(n \log n + k) = \mathcal{O}(n \log n)$.