Structures à l'aide d'arbres

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Rappel sur les structures abstraites

Rappel sur les structures abstraites

Rappel : structure abstraite

On rappelle qu'une **structure abstraite** est la donnée d'un **type** et des **opérations** que l'on peut effectuer dessus.

La définition d'une structure abstraite est indépendante d'une réalisation (**implémentation**) **concrète**.

Les file de priorité

File de priorité

La structure de **file de priorité** (abrégé **FP**) : elle doit gérer des couples de la forme (clé, élément).

Les **clés** sont à valeurs dans un ensemble ordonné (en pratique ce sont des entiers), qui définisse la "priorité" des éléments associés.

Les **éléments** présents dans la **FP** sont tous distincts, par contre ce n'est pas nécessairement le cas des clés.

Il y a une distinction à faire, suivant si les éléments prioritaires sont ceux de plus grande clé (on parle de **FP max**) ou de plus petite clé (**FP min**).

Les file de priorité

File de priorité

Une **file de priorité** doit garantir les opérations suivantes :

- **création** d'une **FP vide** ;
- test d'**égalité** d'une **FP** au **vide** ;
- **insertion** d'un nouvel élément, avec sa clé ;
- **retrait** de l'élément de plus grande priorité ;
- **modification** de la priorité d'un élément (augmentation ou diminution).

Les file de priorité

Exemple

Une **file de priorité** peut être utilisée pour :

- gérer un agenda (les tâches les plus urgentes doivent être effectuées en premier) ;
- dans une imprimante réseau (gestion d'impressions prioritaires) ;
- gérer les processus d'un ordinateur (**système d'exploitation**) ;
- algorithme de **Dijkstra** : on utilisera une **file de priorité (min)** pour l'implémentation efficace d'un algorithme de recherche de plus courts chemins dans un graphe pondéré.

Les dictionnaires

Dictionnaires

La structure de **dictionnaire** : elle doit également gérer des couples (clé, élément).

Les **clés** sont aussi à valeurs dans un ensemble ordonné (entiers, chaînes de caractères), par contre elles sont maintenant supposées toutes distinctes, contrairement aux **éléments**.

Les dictionnaires

Dictionnaires

Une structure de **dictionnaire** doit garantir les opérations suivantes :

- **création** d'un **dictionnaire vide** ;
- test d'**égalité** d'un dictionnaire au **vide** ;
- **recherche** de la présence d'un couple (clé, élément) à partir de la **clé** ;
- **insertion** d'un couple (clé, élément), si la clé n'est pas déjà présente ;
- **suppression** d'un couple (clé, élément) à partir de la **clé** ;
- **remplacement** de l'élément d'un couple (clé, élément) par un autre de même clé.

Les dictionnaires

Exemple

Un **dictionnaire** peut être utilisé pour :

- gérer un dictionnaire (au sens usuel) ;
- gérer des listes d'utilisateurs d'un site web ;
- implémenter efficacement la fonction de modification de la priorité d'un élément dans une file de priorité.

Rappel sur les structures abstraites

Implémentation

Une fois donnée une **structure abstraite**, le but du jeu est d'en donner une **implémentation efficace**, ce que l'on va faire maintenant.

Tas et file de priorité

Une file de priorité à l'aide d'une liste

Utilisation d'une liste

Une première idée pour implémenter une **file de priorité** est d'utiliser une liste.

Une file de priorité à l'aide d'une liste

Utilisation d'une liste non triée

Avec une **liste non triée**, quelques-unes des opérations sont faciles (notamment l'insertion d'un nouvel élément), par contre l'opération consistant à retirer l'élément prioritaire n'est pas évidente car il faut parcourir toute la liste pour calculer le maximum.

Utiliser une liste non triée n'est donc pas une bonne idée d'implémentation.

Une file de priorité à l'aide d'une liste

Utilisation d'une liste triée

Avec une **liste triée** (dans l'ordre décroissant pour une **FP-max**), il est maintenant facile de retirer l'élément de plus grande priorité d'une d'une file de priorité.

Par contre, l'insertion d'un nouvel élément nécessite de chercher où insérer le nouvel élément dans la liste, ce qui prend un temps linéaire en la taille de la liste dans le pire cas.

Utiliser une liste triée n'est donc pas non plus une bonne idée.

La structure de tas

Remarque

Dans la suite, on se concentre sur l'implémentation d'une **FP-max**, pour laquelle on va utiliser un **tas-max** (on ferait usage de manière symétrique d'un **tas-min** pour une **FP-min**).

La structure de tas

Définition (tas)

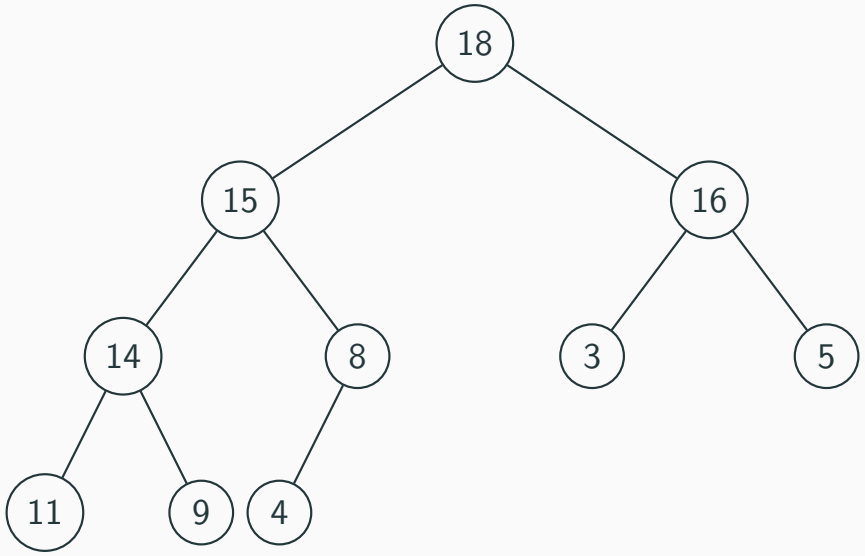
Un **tas-max** (resp. **tas-min**) est un **arbre binaire** complet à gauche, tel que l'étiquette d'un nœud quelconque de l'arbre soit supérieure (resp. inférieure) ou égale à celles de ses fils.

Remarque

La propriété de **tas-max** (resp. **tas-min**) implique que l'étiquette d'un nœud s est supérieure (resp. inférieure) ou égale à celles de tous les nœuds appartenant au sous-arbre enraciné en s.

En particulier, la **racine** de l'arbre possède la plus grande (resp. petite) étiquette.

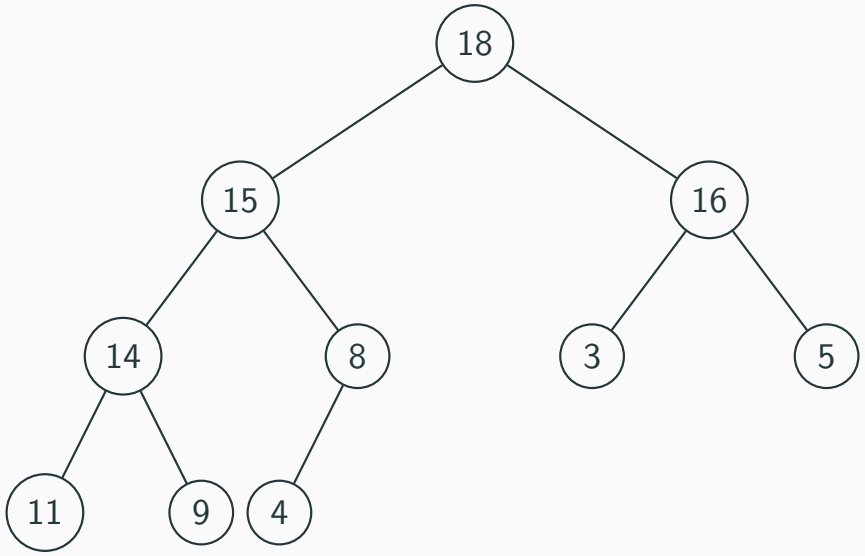
La structure de tas



Exemple

L'arbre ci-dessus est un **tas-max**.

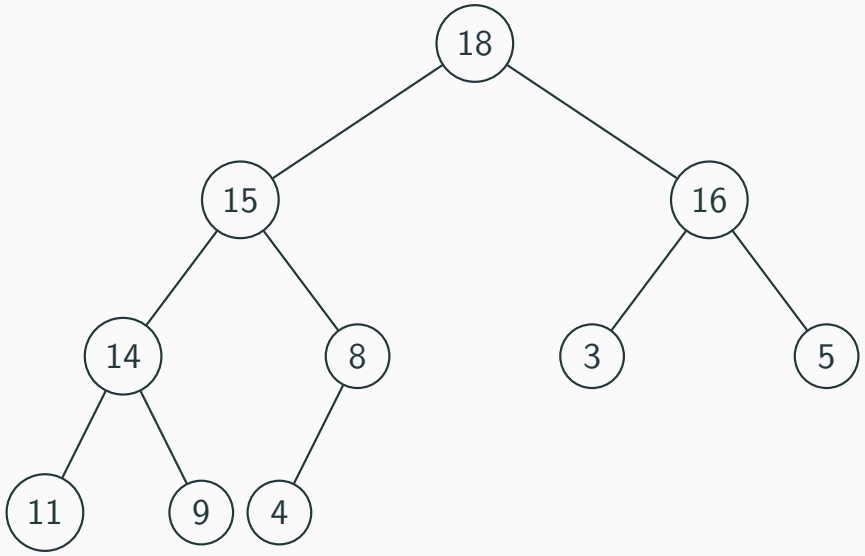
La structure de tas



Définition

Pour mémoire, un **arbre** est dit **binaire** lorsque tous ses nœuds ont au plus deux fils.

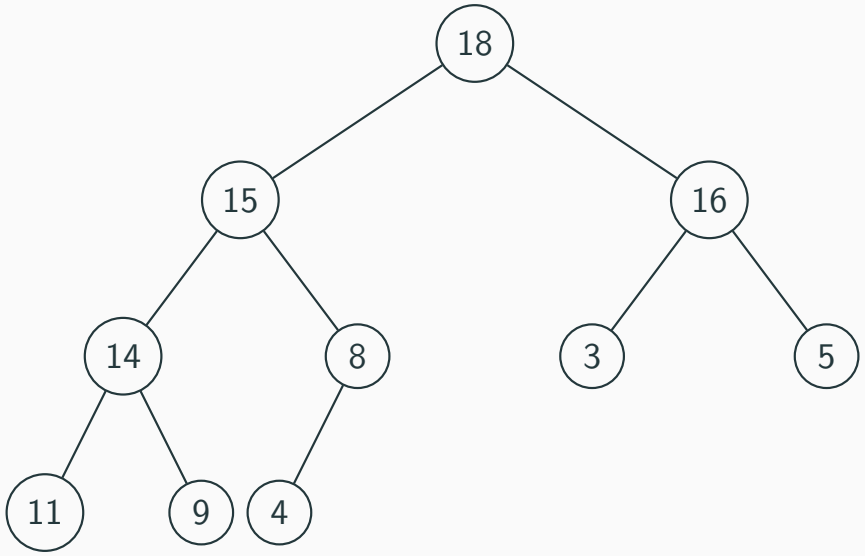
La structure de tas



Définition

Un **arbre** est dit **complet** si toutes ses feuilles se trouvent à la même **profondeur** (i.e. tous les niveaux de l'arbre sont **totalement remplis**).

La structure de tas



Définition

Un **arbre binaire** est dit **complet à gauche** si :

- toutes ses feuilles se situent sur les deux derniers niveaux ;
- l'avant dernier niveau est totalement rempli ;
- le dernier niveau est rempli le plus à gauche possible.

La structure de tas

Définition

La **profondeur** d'un nœud dans un arbre est définie **inductivement** :

- la **racine** est de **profondeur** 0 ;
- la hauteur d'un nœud qui n'est pas la racine est la **profondeur de son parent** $+ 1$.

Définition

La **hauteur** d'un arbre est la profondeur maximale de ses feuilles.

La structure de tas

Proposition

Un **arbre binaire complet à gauche** de hauteur h possède entre 2^h et $2^{h+1} - 1$ nœuds.

Preuve

On numérote les niveaux de l'arbre par profondeur croissante.

L'arbre possède $h + 1$ niveaux. Les h premiers étant remplis, cela représente $1 + 2 + \cdots + 2^{h-1} = 2^h - 1$ nœuds.

Le dernier niveau possède entre 1 et 2^h nœuds, d'où l'encadrement ci-dessus.

La structure de tas

Corollaire

La **hauteur** d'un **arbre binaire complet à gauche** à n nœuds est $\lfloor \log_2(n) \rfloor$.

Preuve

$$
\begin{aligned}
& 2^h \leq n < 2^{h+1} \\
\Rightarrow \quad & h \leq \log_2(n) < h+1 \\
\Rightarrow \quad & h = \lfloor \log_2(n) \rfloor
\end{aligned}
$$

Opérations sur un tas

Remarque

Ainsi, la hauteur d'un **arbre binaire complet à gauche** à n nœuds est en $O(\log n)$.

Toutes les fonctions que l'on écrira dans la suite auront cette complexité, ce qui mène à une implémentation de la structure de file de priorité bien plus intéressante qu'avec des listes !

Opérations sur un tas

On va maintenant décrire de manière théorique comment maintenir une structure de **tas-max** lorsqu'on rajoute ou supprime un nœud.

Essentiellement, il y a deux mécanismes à décrire, dont les autres vont découler.

Il s'agit de rétablir la **structure de tas** lorsqu'on diminue ou augmente l'étiquette d'un nœud.

On suppose que **l'échange** des étiquettes contenues dans deux nœuds peut se faire en **temps constant** dans le tas.

Opérations sur un tas

Augmentation de l'étiquette

Imaginons que l'on augmente l'étiquette d'un nœud.

Alors la propriété de **tas-max** n'est plus nécessairement vérifiée, mais le seul endroit où elle pourrait être violée est entre ce nœud et son parent : ile se peut que l'étiquette du parent soit maintenant **strictement inférieure**.

Dans ce cas, on peut **échanger** l'étiquette du nœud avec celle de son parent, et **réitérer** l'opération : l'étiquette augmentée va progressivement remonter jusqu'à ce que la structure de tas soit rétablie.

Augmentation d'une étiquette

Algorithme 1 : `monter_noeud`

Données : Un (presque) tas-max t, un nœud s

si s *n'est pas la racine* ***et*** $E(parent(s)) < E(s)$ **alors**

    échanger les étiquettes de s et $parent(s)$;

    `monter_noeud`$(t, parent(s))$

Augmentation de l'étiquette

Essentiellement, on applique l'algorithme récursif ci-dessus pour rétablir la structure de tas ($E(s)$ désigne l'étiquette du nœud s).

Augmentation d'une étiquette

Algorithme 1 : `monter_noeud`

Données : Un (presque) tas-max t, un nœud s

si s *n'est pas la racine* ***et*** $E(parent(s)) < E(s)$ **alors**

échanger les étiquettes de s et $parent(s)$;

`monter_noeud`$(t, parent(s))$

Terminaison

la **terminaison** est immédiate, car si un appel récursif est effectué, c'est avec un nœud dont la **profondeur** est strictement inférieure à celle de s.

Augmentation d'une étiquette

Algorithme 1 : `monter_noeud`

Données : Un (presque) tas-max t, un nœud s

si s *n'est pas la racine* ***et*** $E(parent(s)) < E(s)$ **alors**

échanger les étiquettes de s et $parent(s)$;

`monter_noeud`$(t, parent(s))$

Correction

La **correction** est immédiate, pour peu que l'on précise ce qu'est un "presque tas-max", dont un sommet s est marqué : il s'agit d'un arbre vérifiant toutes les propriétés de **tas-max**, sauf peut-être $E(parent(s)) \geq E(s)$.

Il est clair que si l'algorithme s'arrête alors t est un **tas-max**, et si un appel récursif est effectué, c'est clairement sur un presque tas-max et sur le sommet pouvant violer la condition de tas.

Augmentation d'une étiquette

Algorithme 1 : `monter_noeud`

Données : Un (presque) tas-max t, un nœud s

si s *n'est pas la racine* ***et*** $E(parent(s)) < E(s)$ **alors**

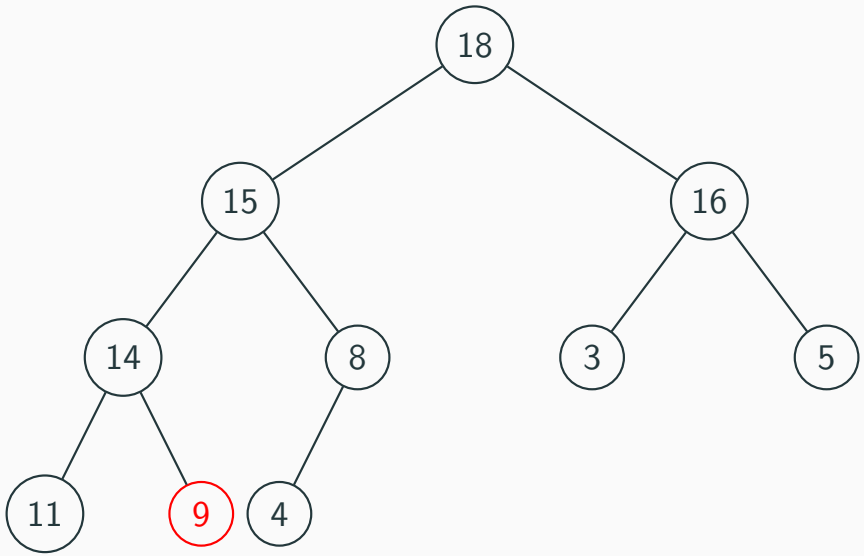
échanger les étiquettes de s et $parent(s)$;

`monter_noeud`$(t, parent(s))$

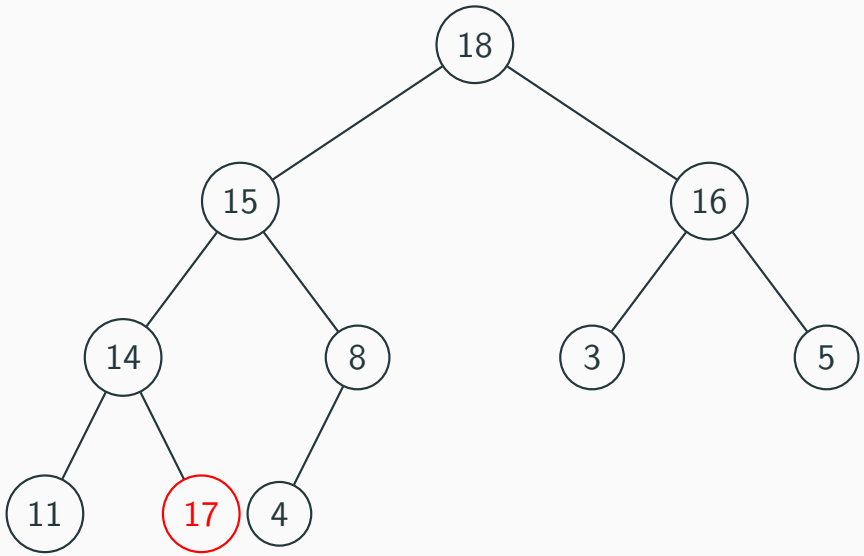
Complexité

La **complexité** est clairement linéaire en la profondeur de s, donc majorée par la hauteur du tas, soit en $O(\log n)$ où n est le nombre d'éléments dans le tas.

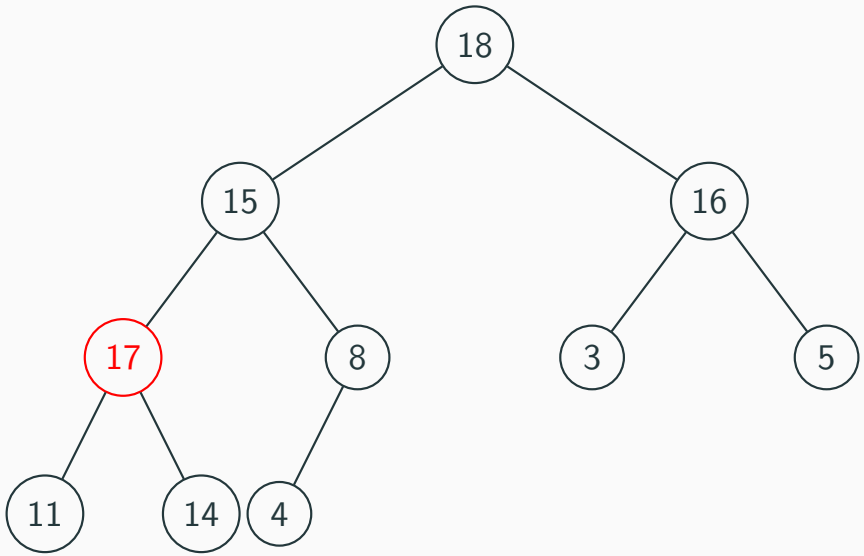
Augmentation d'une étiquette



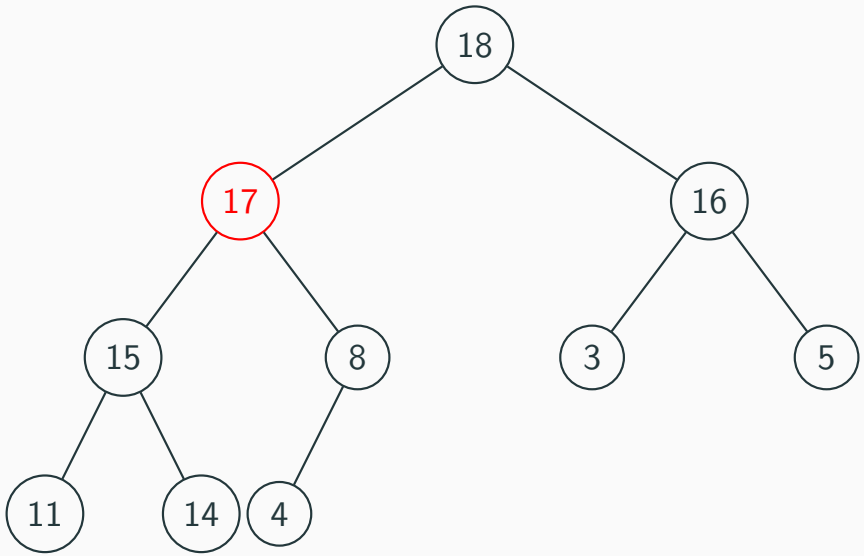
Augmentation d'une étiquette



Augmentation d'une étiquette



Augmentation d'une étiquette



Diminution d'une étiquette

Diminution de l'étiquette

À l'inverse, lorsqu'on diminue l'étiquette d'un nœud, la propriété de **tas** peut être violée car l'étiquette du nœud devient plus grande que l'un (au moins) de ces **enfants**.

Dans ce cas, il suffit d'échanger le nœud avec son **fils** ayant la **plus grande** étiquette pour que le problème descende d'un cran.

Diminution d'une étiquette

Algorithme 2 : `descendre_noeud`

Données : Un (presque) tas-max t, un nœud s

$u \leftarrow s$;

si s *a un fils gauche* g ***et*** $E(g) > E(s)$ **alors**

  $u \leftarrow g$;

si s *a un fils droit* d ***et*** $E(g) > E(u)$ **alors**

  $u \leftarrow d$;

si $u \neq s$ **alors**

  échanger les étiquettes de s et u ;

  `descendre_noeud`(t, u)

Terminaison

La **terminaison** est évidente : la **profondeur** de s augmente strictement à chaque appel récursif, et on s'arrête si on arrive sur une feuille.

Diminution d'une étiquette

Algorithme 2 : `descendre_noeud`

Données : Un (presque) tas-max t, un nœud s

$u \leftarrow s$;

si s *a un fils gauche* g ***et*** $E(g) > E(s)$ **alors**

  $u \leftarrow g$;

si s *a un fils droit* d ***et*** $E(g) > E(u)$ **alors**

  $u \leftarrow d$;

si $u \neq s$ **alors**

  échanger les étiquettes de s et u ;

  `descendre_noeud`(t, u)

Correction

La **correction** se montre facilement par récurrence décroissante sur la profondeur de s.

Diminution d'une étiquette

Algorithme 2 : `descendre_noeud`

Données : Un (presque) tas-max t, un nœud s

$u \leftarrow s$;

si s *a un fils gauche* g ***et*** $E(g) > E(s)$ **alors**

  $u \leftarrow g$;

si s *a un fils droit* d ***et*** $E(g) > E(u)$ **alors**

  $u \leftarrow d$;

si $u \neq s$ **alors**

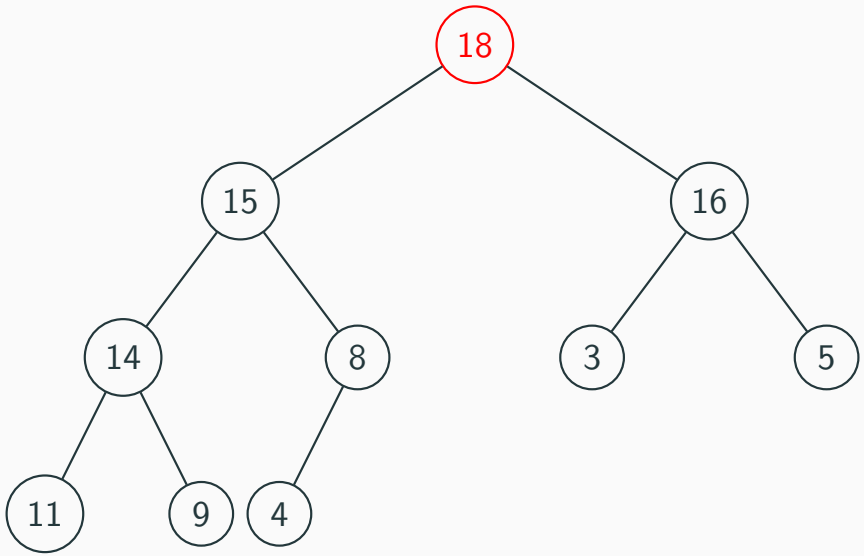
  échanger les étiquettes de s et u ;

  `descendre_noeud`(t, u)

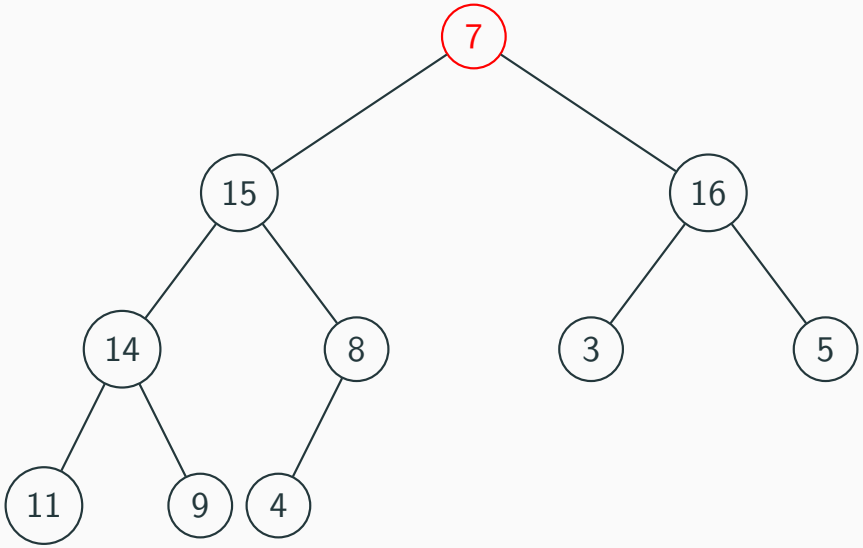
Complexité

La **complexité** est linéaire en la distance (maximale) entre s et une feuille, donc majorée par la hauteur du tas, donc en $O(\log n)$ où n est le nombre d'éléments dans le tas.

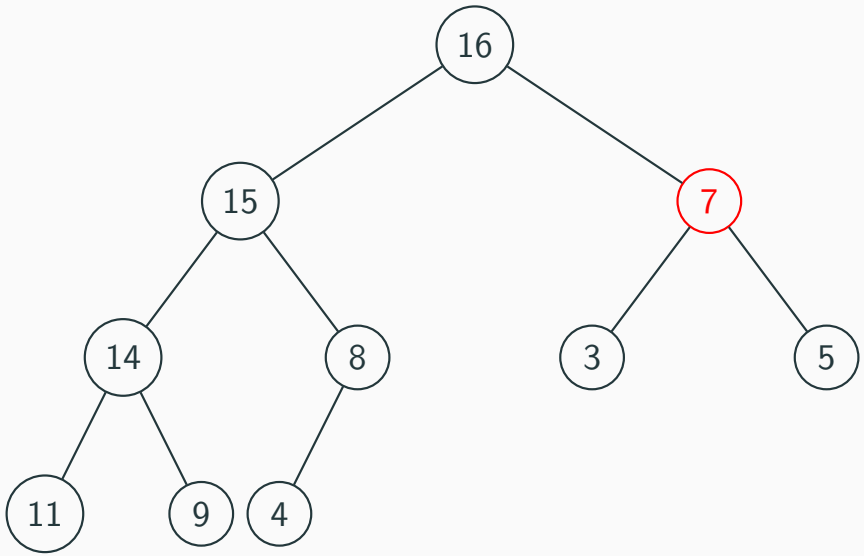
Diminution d'une étiquette



Diminution d'une étiquette



Diminution d'une étiquette



Opérations sur un tas

Rajout d'un nœud

Pour **rajouter** un nœud dans le tas, il suffit de placer le nœud de façon à maintenir la structure d'arbre binaire complet à gauche, et d'appeler `monter_noeud` pour rétablir la structure de tas.

Suppression de la racine

Pour rétablir la structure de tas si l'on **supprime** la **racine** du tas, il suffit de placer à la racine le nœud situé en dernière feuille, et d'appeler `descendre_noeud`.

Stockage d'un arbre binaire complet à gauche dans un tableau

Stockage dans un tableau

On peut facilement stocker un arbre binaire complet à gauche dans un tableau (ce qui rendra facile les échanges d'étiquettes) : il suffit de stocker les éléments de l'arbre dans l'ordre du parcours en largeur.

Proposition

Si i est l'indice d'un élément du tableau, correspondant au i-ème nœud dans l'énumération du parcours en largeur (initialisée à partir de la racine d'indice 0) :

- si $i \neq 0$ (donc i n'est pas la racine), le père de i est $\lfloor \frac{i-1}{2} \rfloor$;
- si i possède un fils gauche, alors c'est $2i + 1$;
- si i possède un fils droit, alors c'est $2i + 2$.

Stockage d'un arbre binaire complet à gauche dans un tableau

Preuve

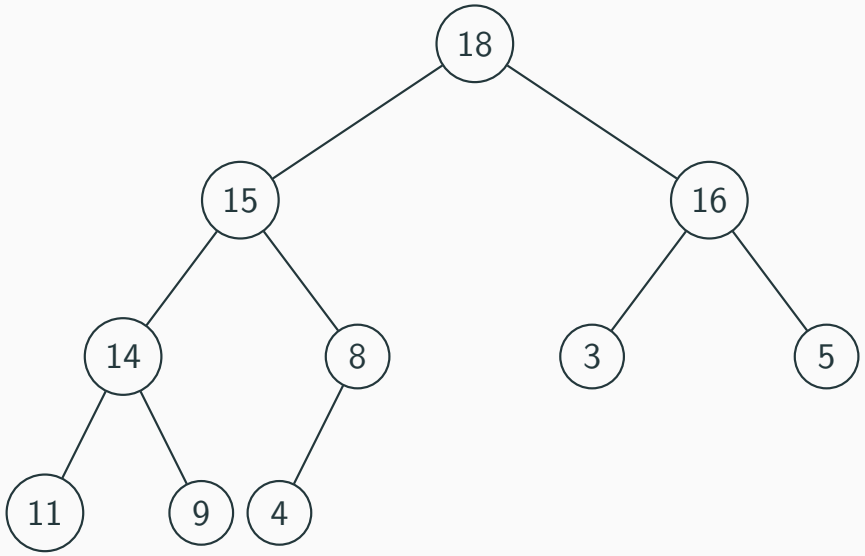
Il suffit de montrer la proposition sur les indices des fils, l'assertion sur l'indice du parent en résulte.

On vérifie (par récurrence immédiate) que les nœuds à profondeur h ont pour indices $2^h - 1, 2^h, \cdots, 2^{h+1} - 2$.

Ainsi le nœud le plus à gauche à la profondeur $h+1$ est $2^{h+1} - 1$, le suivant 2^{h+1}, qui s'obtiennent bien comme $2(2^{h-1} - 1)$ auquel on ajoute 1 ou 2.

Il en va donc de même des suivants.

Stockage d'un arbre binaire complet à gauche dans un tableau



18	15	16	14	8	3	5	11	9	4

Stockage d'un arbre binaire complet à gauche dans un tableau

Remarque

Il est assez courant lorsqu'on implémente un tas dans une tableau, de commencer à remplir le tableau à partir de l'indice 1 en ignorant la première case, ce qui donne $\lfloor \frac{i}{2} \rfloor$, $2i$, et $2i+1$ dans la proposition précédente.

On ne le fera pas dans ce cours, mais ce sera peut être la convention prise dans certaines épreuves de concours.

Implémentation de la structure de file de priorité

```
1  type 'a fp = { mutable n : int ; tab : 'a array } ;;
```

Implémentation dan un tableau

Pour la réalisation d'une **file de priorité**, il est nécessaire d'utiliser un tableau qui n'est pas entièrement rempli, pour laisser de la place aux éléments que l'on va rajouter dans la file.

On présente ici une implémentation où la taille du tableau est fixée une fois pour toutes à la création, et l'on maintient une variable indiquant le nombre d'éléments du tableau qui font effectivement partie du tas.

Implémentation de la structure de file de priorité

Remarque

Une implémentation plus générale, sans hypothèse sur la **capacité** de la **file de priorité**, consiste à utiliser un **tableau redimensionnable** comme on l'a vu plus tôt dans l'année : il n'est pas très compliqué d'implémenter une telle structure, il suffit de doubler la taille du tableau utilisé lorsque l'on manque de place.

Il faut alors recopier tous les éléments de l'ancien tableau vers le nouveau, ce qui est coûteux, mais cette opération est faite suffisamment peu souvent pour que l'ajout d'un élément à la structure se fasse en **complexité amortie constante**.

Implémentation de la structure de file de priorité

Couples (clé, élément)

En pratique, pour implémenter une structure de file de priorité, il faut stocker dans le tableau des couples (clé, élément).

La **comparaison** d'éléments en OCaml se faisant avec l'**ordre lexicographique**, toutes les fonctions que l'on va écrire compareront les couples comme il se doit.

Implémentation de la structure de file de priorité

```
1  let fg i = 2*i+1 ;;
2  let fd i = 2*i+2 ;;
3  let pere i = (i-1)/2 ;;
4  let echanger t i j = let a=t.(i) in t.(i) <- t.(j) ; t.(j) <- a ;;
5
6  let creer_fp taille x = {n = 0 ; tab = Array.make taille x} ;;
7
8  let est_vide_fp f = f.n = 0 ;;
```

Fonctions basiques

- Les fonctions `fg`, `fd`, `pere`, et `echanger` nous serviront à manipuler facilement le tableau associé au tas.
- La fonction `creer_fp` prend en paramètres la taille du tableau (la capacité de la file), ainsi qu'un élément x utile pour fixer le type de la file de priorité, même si la file est vide à la création.
- Enfin, pour tester si une file de priorité est vide, il suffit de vérifier que le champ n est nul.

Implémentation de la structure de file de priorité

```
1  let rec monter_noeud t i =
2    if i <> 0 && t.(pere i) < t.(i) then
3      begin
4        echanger t i (pere i) ;
5        monter_noeud t (pere i)
6      end
7  ;;
```

```
1  let rec descendre_noeud t n i =
2    let j = ref i in
3    if fg i < n && t.(fg i) > t.(i) then j := fg i ;
4    if fd i < n && t.(fd i) > t.( !j) then j := fd i ;
5    if i <> !j then
6      begin
7        echanger t i !j ;
8        descendre_noeud t n !j
9      end
10 ;;
```

Monter et descendre

On implémente maintenant les algorithmes `monter_noeud` et `descendre_noeud` vus précédemment.

Les deux fonctions prennent en arguments le tableau `t` et l'indice i du nœud en question.

La fonction `descendre_noeud` prend aussi le nombre n d'éléments dans le tas.

Implémentation de la structure de file de priorité

```
1  let enfiler_fp f x =
2    f.tab.(f.n) <- x ;
3    f.n <- f.n + 1 ;
4    monter_noeud f.tab (f.n - 1)
5  ;;
```

```
1  let supprimer_max_fp f =
2    f.n <- f.n - 1 ;
3    echanger f.tab 0 f.n ;
4    descendre_noeud f.tab f.n 0 ;
5    f.tab.(f.n)
6  ;;
```

Fonctions de file de priorité

On donne maintenant l'implémentation des deux principales opérations de file de priorité max : ajouter un élément et supprimer le maximum.

Implémentation de la structure de file de priorité

```
1  let enfiler_fp f x =
2    f.tab.(f.n) <- x ;
3    f.n <- f.n + 1 ;
4    monter_noeud f.tab (f.n - 1)
5  ;;
```

```
1  let supprimer_max_fp f =
2    f.n <- f.n - 1 ;
3    echanger f.tab 0 f.n ;
4    descendre_noeud f.tab f.n 0 ;
5    f.tab.(f.n)
6  ;;
```

Fonctions de file de priorité

Pour enfiler un élément, on suppose qu'il y a assez de place dans le tableau, et on rajoute un élément à la suite des autres éléments du tas.

On incrémente le nombre d'éléments contenus dans la file, et on appelle `monter_noeud` sur le dernier élément du tas.

Implémentation de la structure de file de priorité

```
1  let enfiler_fp f x =
2    f.tab.(f.n) <- x ;
3    f.n <- f.n + 1 ;
4    monter_noeud f.tab (f.n - 1)
5  ;;
```

```
1  let supprimer_max_fp f =
2    f.n <- f.n - 1 ;
3    echanger f.tab 0 f.n ;
4    descendre_noeud f.tab f.n 0 ;
5    f.tab.(f.n)
6  ;;
```

Fonctions de file de priorité

À l'inverse, pour retirer le maximum (la racine du tas), on choisit de l'échanger avec le dernier élément du tas, en décrémentant le nombre d'éléments contenus dans la file.

On appelle descendre_noeud sur la nouvelle racine, et on renvoie l'élément supprimé.

Implémentation de la structure de file de priorité

Complexité

Ainsi implémentée, la structure de **file de priorité** possède une complexité $O(\log n)$ pour toutes ses opérations, où n est le nombre d'éléments présents dans la file de priorité, exceptée la création qui est **linéaire** en la capacité choisie, et le test d'égalité au vide qui s'exécute en **temps constant**.

Intermède : le tri par tas

Tri pas tas

Avec toutes ces fonctions, et notre structure de tas, on peut facilement implémenter un tri efficace appelé “tri par tas” :

- on se sert du tableau `t` à trier comme contenu d'une **file de priorité** ;
- on enfile tous les éléments de `t` à l'aide de `enfiler_fp` pour obtenir un **tas** ;
- on supprime tous les éléments de la file de priorité, en remarquant que lorsque `supprimer_max_fp` est appelée :
  - les éléments ne faisant plus partie de la file de priorité sont les plus grands éléments du tableau, et sont triés (à la fin de `t`) ;
  - l'élément supprimé par `supprimer_max_fp` est déplacé “au bon endroit” pour maintenir cet invariant.

Intermède : le tri par tas

```
1  let tri_par_tas t =
2    let n = Array.length t in
3    let fp = { n = 0 ; tab = t } in
4    for i = 0 to n-1 do
5      enfiler_fp fp t.(i)
6    done ;
7    while fp.n > 0 do
8      let _ = supprimer_max_fp fp in ()
9    done
10 ;;
```

Terminaison

La fonction `tri_par_tas` termine, car :

- les fonctions `enfiler_fp` et `supprimer_max_fp` terminent ;
- à chaque passage dans la boucle **`while`**, la valeur de `fp.n` décroit strictement.

Intermède : le tri par tas

```
1  let tri_par_tas t =
2    let n = Array.length t in
3    let fp = { n = 0 ; tab = t } in
4    for i = 0 to n-1 do
5      enfiler_fp fp t.(i)
6    done ;
7    while fp.n > 0 do
8      let _ = supprimer_max_fp fp in ()
9    done
10 ;;
```

Correction

La correction repose sur les invariants de boucle suivants :

- boucle **for** : “`t[0:i]` est un tas” ;
- boucle **while** : “`t[fp.n:n]` est trié, et contient les plus grands éléments de t”.

Intermède : le tri par tas

```
1  let tri_par_tas t =
2    let n = Array.length t in
3    let fp = { n = 0 ; tab = t } in
4    for i = 0 to n-1 do
5      enfiler_fp fp t.(i)
6    done ;
7    while fp.n > 0 do
8      let _ = supprimer_max_fp fp in ()
9    done
10 ;;
```

Complexité

La fonction `tri_par_tas` a une complexité en $O(n \log n)$ car :

- les fonctions `enfiler_fp` et `supprimer_max_fp` fonctionnent en $O(\log n)$;
- on passe n fois dans la boucle **`for`** et dans la boucle **`while`**.

Modification d'une clé

Dans une **file de priorité**, les éléments stockés sont des couples (clé, élément).

L'une des opérations possibles de la structure est la **modification** de la clé associée à un élément présent dans la file.

Il faut donc être capable de retrouver le nœud du tas à partir de l'élément.

On indique ici deux solutions.

Modification d'une clé

Utilisation d'un tableau

Un cas qui se produit souvent est celui où les éléments sont à valeurs dans un ensemble de la forme $[\![0, B-1]\!]$, avec B un entier pas trop grand.

$\hookrightarrow$ On peut alors utiliser un **tableau** de taille B pour stocker les positions des éléments présents dans la file de priorité dans le tableau associé au tas.

Modification d'une clé

Utilisation d'un tableau

- Lorsqu'on utilise la fonction echanger dans une des fonctions précédentes, il ne faut pas oublier de répercuter le changement sur ce tableau des positions.
- De même, il faut gérer l'**insertion** et la **suppression** d'un élément dans la file de priorité avec ce tableau.
- La **complexité** des opérations de file de priorité restent alors **logarithmique** en le nombre d'éléments, de même que la modification de la priorité : il suffit de monter ou descendre l'élément dont on a modifié la priorité.

Modification d'une clé

Exemple : Utilisation d'un tableau

Un exemple où cette approche est très efficace est celle de l'algorithme de Djikstra sur un graphe dont les sommets sont numérotés de 0 à $N-1$.

Modification d'une clé

Utilisation d'un dictionnaire

Plus généralement, on peut utiliser une structure de **dictionnaire** dans laquelle on stocke des couples (élément, position dans le tableau), ce qui amène à la section suivante.

La complexité des opérations de **file de priorité** s'en trouve modifiée selon la structure choisie ; signalons qu'avec une **table de hachage** elle est quasiment inchangée.

$\hookrightarrow$ Bien implémentée, une **table de hachage** effectue ses opérations en **temps constant amorti**, sous des hypothèses naturelles de répartition "aléatoire" des clés de hachage.

Arbres binaires de recherche, Arbres AVL

Implémentation d'un dictionnaire avec une liste chaînée

Liste chaînée

Pour les mêmes raisons que pour une file de priorité, utiliser une structure de liste chaînée pour implémenter une structure de dictionnaire n'est pas efficace du point de vue de la complexité ; il faut stocker des couples (clé, élément) où les clés sont des éléments distincts, et rechercher si une clé est présente se fait en $O(n)$ dans le pire des cas.

On va là aussi proposer une structure efficace à l'aide d'arbres, plus précisément d'**arbres binaires de recherche**.

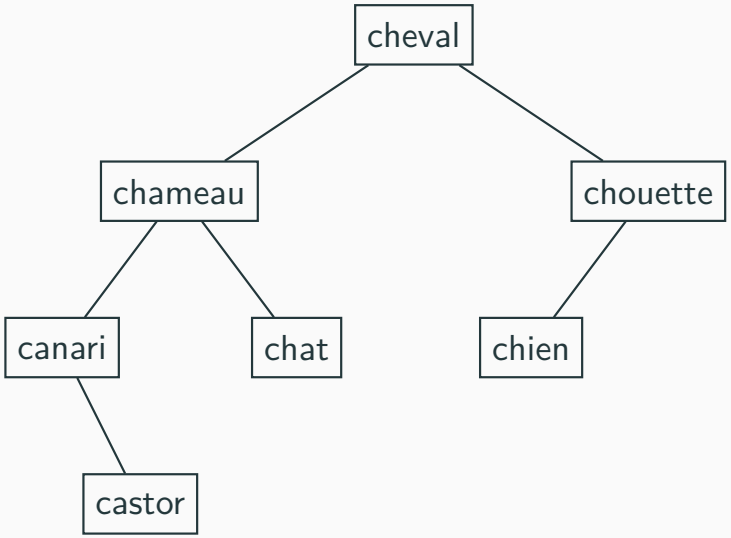
Structure d'arbre binaire de recherche

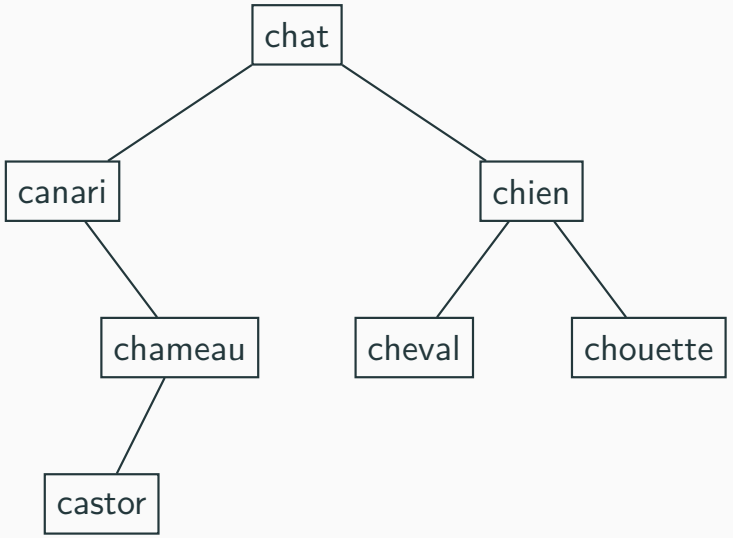
Définition (Arbre binaire de recherche)

Un **arbre binaire de recherche** (ABR) est un arbre binaire éventuellement vide dont les nœuds sont des couples (clé, valeur), tel que :

- deux nœuds n'ont **jamais** la **même clé** ;
- les clés sont à valeur dans un ensemble **totalement ordonné** ;
- pour tout nœud de l'arbre, les clés du **sous-arbre gauche** sont **strictement inférieures** à celle du nœud, et les clés du **sous-arbre droit** sont **strictement supérieures** à celle du nœud.

Structure d'arbre binaire de recherche





Exemple

Voici deux exemples d'ABR dont l'ensemble des clés est : {canari, castor, chameau, chat, cheval, chien, chouette}.

L'ordre utilisé est l'ordre alphabétique.

On n'a pas mis les valeurs associées, qui pourraient être les définitions de ces mots dans la langue française.

Structure d'arbre binaire de recherche

Proposition

Un arbre binaire est un **ABR** si et seulement si l'**énumération infixe** de ses nœuds est **strictement croissante**.

Remarque

L'**énumération infixe** n'est pas unique : elle correspond à plusieurs **ABR**.

C'est le cas par exemple pour les ABR précédents : les clés sont les mêmes donc les deux arbres ont la même **énumération infixe**.

Structure d'arbre binaire de recherche

Définition (Hauteur)

La **hauteur** d'un ABR est définie inductivement comme suit :

- la hauteur de l'arbre vide est -1 ;
- pour un arbre non vide, elle est égale au maximum des hauteurs de ses sous-arbres gauche et droit, $+\ 1$.

Remarque

On peut considérer qu'un ABR non vide est un arbre binaire **entier**, les "**feuilles**" de l'arbre étant des arbres **vides**.

C'est cohérent avec la définition de la hauteur, et pratique avec l'implémentation OCaml que l'on va voir dans ce chapitre.

Structure d'arbre binaire de recherche

Proposition

Le nombre de nœuds n d'un ABR (non vide) de hauteur h vérifie : $h + 1 \leq n \leq 2^{h+1} - 1$.

Preuve

La borne $2^{h+1} - 1$ est obtenue pour un arbre binaire **complet**, comme dans le cas des tas.

Un **chemin** de la racine a une feuille de profondeur h comporte $h + 1$ nœuds, ce qui démontre l'autre borne.

Corollaire

La hauteur h d'un ABR (non vide) à n nœuds vérifie :
$\lfloor \log_2(n) \rfloor \leq h \leq n - 1$.

Preuve

En passant au $\log$ dans l'encadrement précédent, on obtient :
$\log_2(n+1) - 1 \leq h \leq n - 1$.

Or $h \geq \log_2(n+1) - 1 > \log_2(n) - 1 \geq \lfloor \log_2(n) \rfloor \geq \lfloor \log_2(n) \rfloor - 1$.

Donc $h \geq \lfloor \log_2(n) \rfloor$.

Implémentation des ABR en Ocaml

Implémentation persistante

On choisit une implémentation **persistante** des **ABR**, identique à celle classique des arbres binaires.

Contrairement à l'implémentation des tas de la section précédente, les fonctions qui travailleront sur les arbres ne modifieront pas la structure mais renverront de nouveaux arbres.

En pratique, on implémente les ABR comme des **arbres binaires entiers** (tout nœud interne a 2 fils), les feuilles correspondent à un **arbre vide**.

Implémentation des ABR en Ocaml

```
1  type 'a abr = Vide | N of 'a abr * 'a * 'a abr;;
2
3  let rec hauteur a = match a with
4    | Vide -> -1
5    | N(g,_,d) -> 1 + max (hauteur g) (hauteur d)
6  ;;
```

Exemple

On peut facilement calculer la hauteur d'un tel arbre.

Implémentation de la structure de dictionnaire avec des ABR

```
1  type ('a, 'b) noeud = {cle : 'a; mutable valeur: 'b};;
```

Nœuds d'un ABR

En OCaml, on pourrait représenter les nœuds comme ci-dessus.

On a rendu le deuxième champ **mutable**, car dans un **dictionnaire**, on peut vouloir changer la valeur associée à une clé : dans ce cas, il suffit de retrouver le nœud à partir de la clé, et modifier la valeur associée.

Ainsi, la modification d'une valeur est essentiellement une recherche de clé.

Dans la suite, on oubliera les valeurs pour ne considérer que les clés, d'un point de vue algorithmique les opérations sont les mêmes.

Implémentation de la structure de dictionnaire avec des ABR

```
1  let creer_abr () = Vide ;;
2  let est_vide_abr a = a = Vide ;;
```

Fonctions basiques

Créer un ABR vide, et tester si un ABR est vide est facile.

Implémentation de la structure de dictionnaire avec des ABR

```
1  let rec rechercher a x = match a with
2    | Vide -> false
3    | N(g,y,_) when y>x -> rechercher g x
4    | N(_,y,d) when y<x -> rechercher d x
5    | _ -> true
6  ;;
```

Recherche

Si l'on **cherche** x dans un ABR de racine y, on est dans l'un des 3 cas suivants :

- soit $y = x$, auquel cas on a trouvé x ;
- soit $y < x$, auquel cas x ne peut se trouver que dans le sous-arbre droit du nœud étiqueté par y ;
- soit $y > x$, auquel cas x ne peut se trouver que dans le sous-arbre gauche du nœud étiqueté par y.

Implémentation de la structure de dictionnaire avec des ABR

```
1  let rec inserer a x = match a with
2    | Vide -> N(Vide,x,Vide)
3    | N(g,y,d) when y>x -> N(inserer g x, y, d)
4    | N(g,y,d) when y<x -> N(g, y, inserer d x)
5    | _ -> a
6  ;;
```

Insertion

Pour l'**insertion** de x dans l'ABR, on chemine dans l'arbre jusqu'à arriver à une feuille **`Vide`**, que l'on remplace par un arbre contenant simplement x.

On choisit de renvoyer l'arbre à l'identique si x est déjà présent.

Implémentation de la structure de dictionnaire avec des ABR

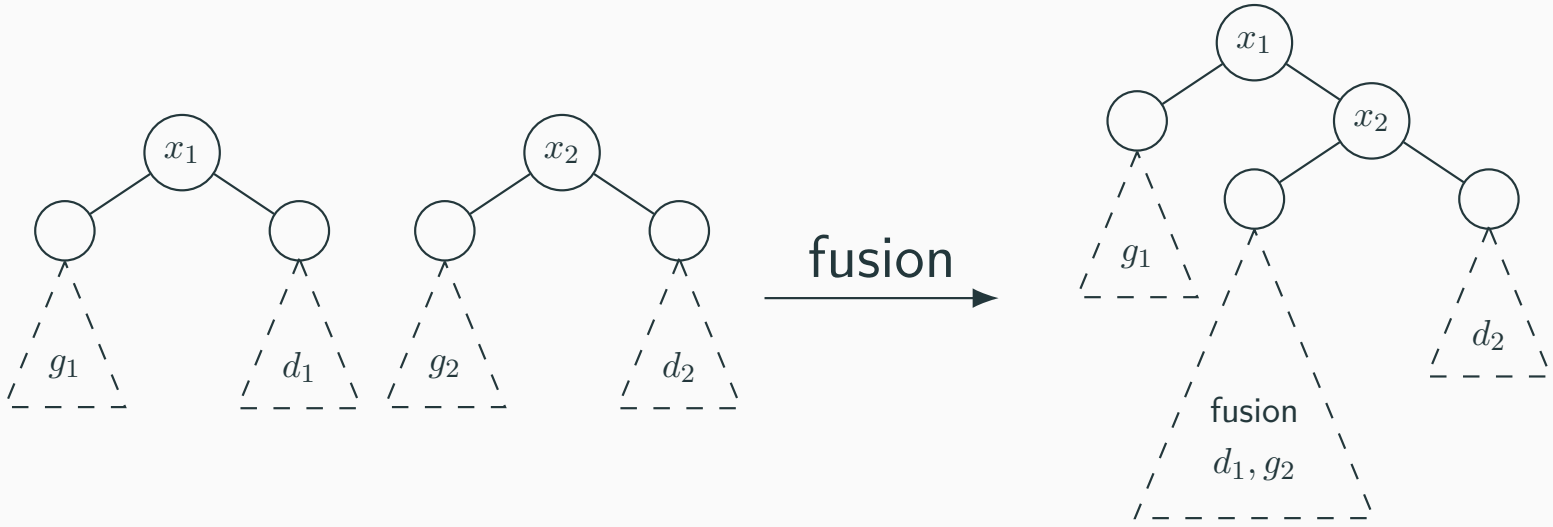
Suppression d'un élément dans un ABR

La **suppression** d'un élément est légèrement plus complexe.

Comme pour l'insertion, on va **descendre** dans l'arbre jusqu'à tomber sur l'élément en question, et on sera ramené à la suppression de la **racine** d'un ABR.

On propose alors deux solutions.

Implémentation de la structure de dictionnaire avec des ABR



Suppression par fusion

Si x est la racine de l'ABR et l'élément à supprimer, on peut **fusionner** ses deux sous-arbres gauche et droit.

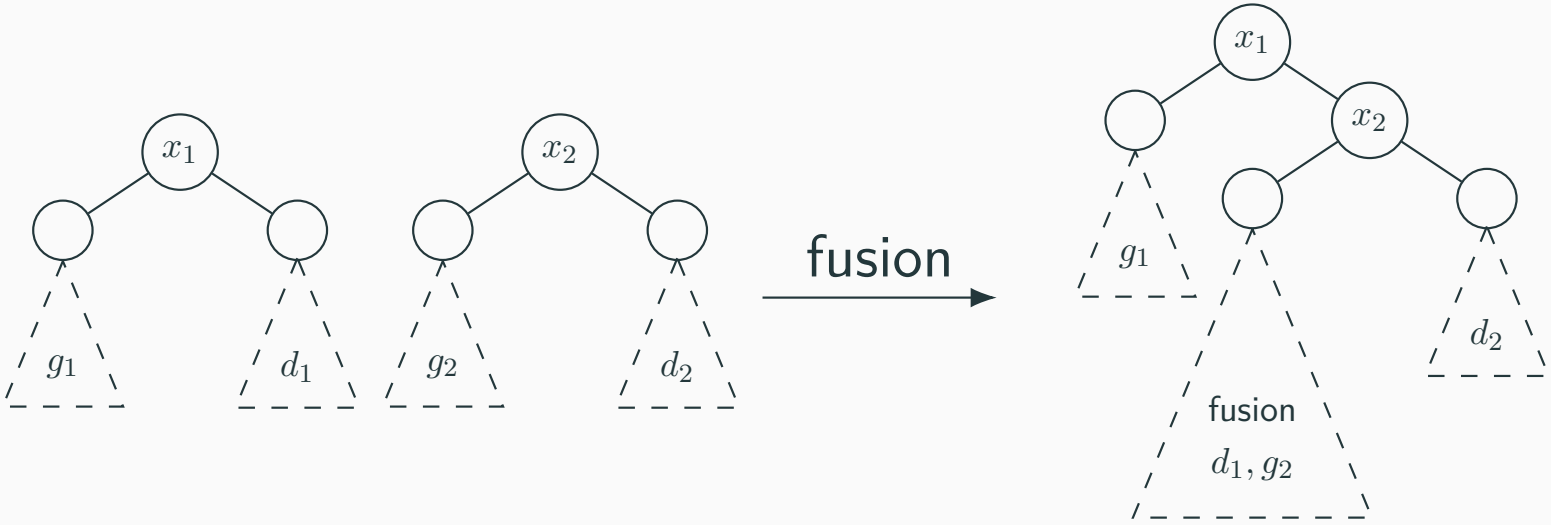
- Cette fusion est immédiate si l'un des deux sous-arbres est vide.
- Sinon, elle peut se faire selon le schéma ci-dessus, où x_1 et x_2 sont les fils de x.

Implémentation de la structure de dictionnaire avec des ABR

Remarques

- Cette méthode donne bien un ABR si les deux arbres concernés sont bien des ABR et que les clés du premier sont strictement inférieures aux clés du second.
- Le choix de x_1 comme nouvelle racine est arbitraire, et on pourrait choisir symétriquement x_2.
- De même que pour l'insertion, si le nœud n'est pas présent dans l'arbre, celui-ci est renvoyé à l'identique.

Implémentation de la structure de dictionnaire avec des ABR




```
1   let rec fusion a1 a2 = match a1,a2 with
2     | Vide,_ -> a2
3     | _,Vide -> a1
4     | N(g1,x1,d1), N(g2,x2,d2) -> N(g1,x1,N(fusion d1 g2,x2,d2))
5   ;;
6
7   let rec supprimer a x = match a with
8     | Vide -> Vide
9     | N(g,y,d) when y>x -> N(supprimer g x, y, d)
10    | N(g,y,d) when y<x -> N(g, y, supprimer d x)
11    | N(g,y,d) -> fusion g d
12  ;;
```

Implémentation de la structure de dictionnaire avec des ABR

```
1  let rec max_abr a = match a with
2    | Vide -> failwith "vide"
3    | N(_,x,Vide) -> x
4    | N(_,_,d) -> max_abr d
5  ;;
```

```
1  let rec supprimer a x = match a with
2    | Vide -> Vide
3    | N(g,y,d) when y>x -> N(supprimer g x, y, d)
4    | N(g,y,d) when y<x -> N(g, y, supprimer d x)
5    | N(Vide,y,d) -> d
6    | N(g,y,d) -> let z=max_abr g in N(supprimer g z, z, d)
7  ;;
```

Suppression du maximum du sous-arbre gauche

Supposons encore que la racine x soit l'élément à **supprimer**.

- Si le sous-arbre gauche est **vide**, l'arbre obtenu après suppression est simplement le sous-arbre droit.
- Sinon, on peut remplacer la racine par le maximum de son sous-arbre gauche (préalablement supprimé).

$\hookrightarrow$ On maintient ainsi bien la structure d'ABR.

Pour trouver le **maximum** d'un **ABR**, il suffit de descendre le plus à **droite** possible.

Implémentation de la structure de dictionnaire avec des ABR

Complexité

Toutes les opérations d'insertion, de suppression, et de recherche prennent un temps $O(h)$, où h est la **hauteur** de l'arbre.

Malheureusement, h peut être proche du nombre de nœuds, ce qui est mauvais en terme de complexité.

Exemple

En partant d'un arbre vide, si on insère successivement n nœuds dans l'ordre croissant, on obtient un arbre de hauteur $n-1$, et la construction se fait en temps $O(n^2)$ (on construit un "**peigne**").

Implémentation de la structure de dictionnaire avec des ABR

Solutions

Il y a plusieurs solutions pour remédier à ce problème.

- Si on se donne un ensemble de clés que l'on insère dans un ordre aléatoire (avec distribution uniforme sur les $n!$ permutations possibles), on peut montrer que l'espérance de la hauteur de l'arbre obtenu est $O(\log n)$.
  $\hookrightarrow$ En pratique, les choses se passent bien si on laisse faire le hasard.
- On peut rajouter de l'information dans l'ABR. Ces informations permettent, en utilisant des "rotations" bien choisies, d'équilibrer l'arbre pour garder la hauteur $O(\log n)$.

La suite du chapitre est dévolue à l'étude des arbres **AVL**, qui s'inscrivent dans la deuxième stratégie.

Arbres AVL

Définition

Un arbre **AVL** est un ABR tel que pour tout nœud, ses sous-arbres gauche et droit aient la même hauteur, à 1 près.

Remarque

Cette condition est suffisante pour que l'arbre soit approximativement équilibré.

Le nom des arbres **AVL** vient du nom de ses inventeurs : Georgii Adelson-Velsky et Evguenii Landis.

Théorème

La hauteur h d'un arbre **AVL** à n nœuds vérifie $h = O(\log n)$.

Preuve

Il suffit de voir qu'un arbre **AVL** a toujours un grand nombre de nœuds, relativement à sa hauteur.

Notons N_h le nombre de nœuds minimal d'un arbre **AVL** de hauteur h.

On a $N_0 = 1, N_1 = 2$, et $H_h = N_{h-1} + N_{h-2} + 1$. La suite (N_h+1) vérifie donc la même relation que la suite de Fibonacci.

Théorème

La hauteur h d'un arbre **AVL** à n nœuds vérifie $h = O(\log n)$.

Preuve

Ainsi, on peut montrer que $N_h = C\varphi^h + D(\frac{-1}{\varphi})^h - 1$, avec $\varphi = \frac{1+\sqrt{5}}{2}$, et C et D deux constantes. Donc, asymptotiquement, $N_h \sim C\varphi^h$.

Par suite, un arbre **AVL** à n nœuds possède une hauteur majorée par $\log_\varphi(n)$ (à une constante additive près), d'où le résultat.

Remarque

$\frac{1}{\log_2(\varphi)} \approx 1.44$, donc asymptotiquement un arbre **AVL** à n nœuds a une hauteur majorée par une quantité de l'ordre de $1.44 \times \log_2(n)$.

On n'est pas très loin des arbres binaires **complets** pour lesquels la hauteur est bornée par $\log_2(n)$.

Rotations et maintien de la structure d'arbre AVL

Arbres AVL

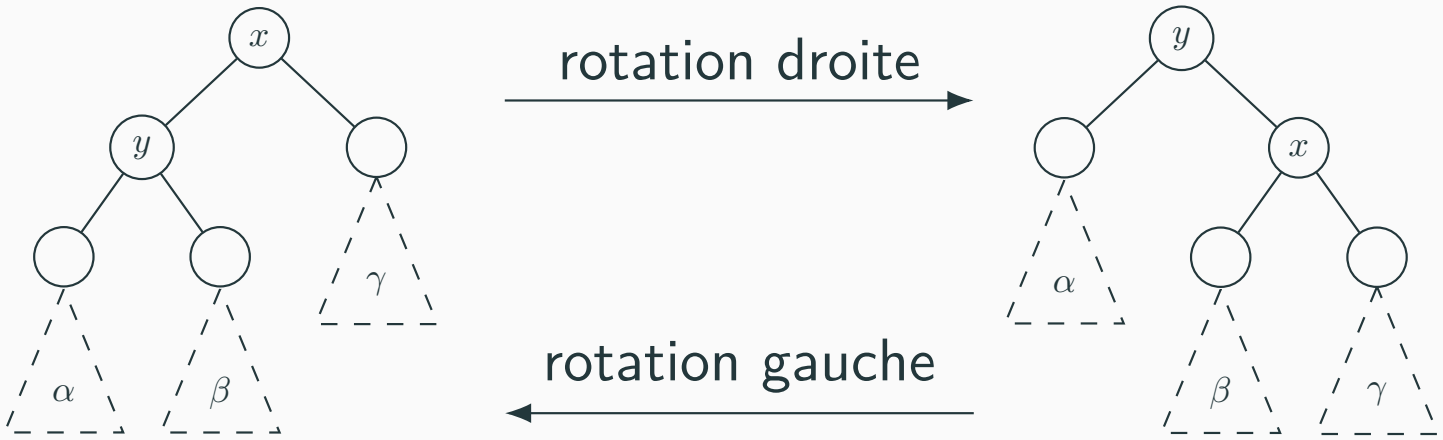
Les opérations de recherche, insertion, et suppression dans un arbre **AVL** à n nœuds se font donc en $O(\log n)$.

Mais il faut modifier les fonctions d'insertion et de suppression pour que la structure d'arbre **AVL** soit maintenue.

On va procéder en utilisant des **rotations** sur l'arbre.

On remarque que ces rotations maintiennent bien la structure d'**ABR**.

Rotations et maintien de la structure d'arbre AVL



Rotations

Rotations dans un **ABR** : x et y sont des nœuds, α, β et γ sont des **ABR**.

Rotations et maintien de la structure d'arbre AVL

Rééquilibrage

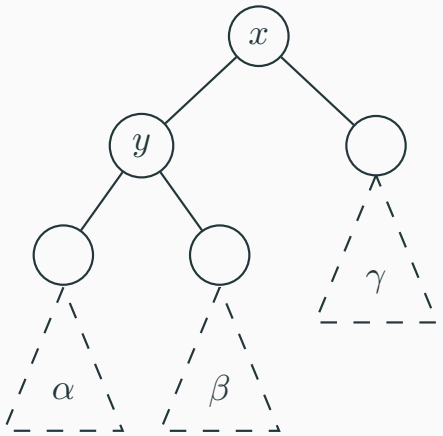
En fait, on va insérer/supprimer les nœuds comme dans les **ABR**, mais on opérera un **rééquilibrage** éventuel "à la remontée", en procédant par des **rotations**.

Lorsqu'on insère/supprime un nœud, il se peut que l'on introduise un déséquilibre qui fait perdre la structure d'**AVL**.

Comme insertion et suppression font varier les hauteurs des sous-arbres d'au plus 1, c'est qu'un nœud (dont le sous-arbre associé a pour hauteur h) possède deux sous-arbres de hauteurs respectives $h-1$ et $h-3$.

On suppose que le sous-arbre de hauteur $h-1$ est celui de gauche, l'autre cas s'en déduit par symétrie.

Rotations et maintien de la structure d'arbre AVL

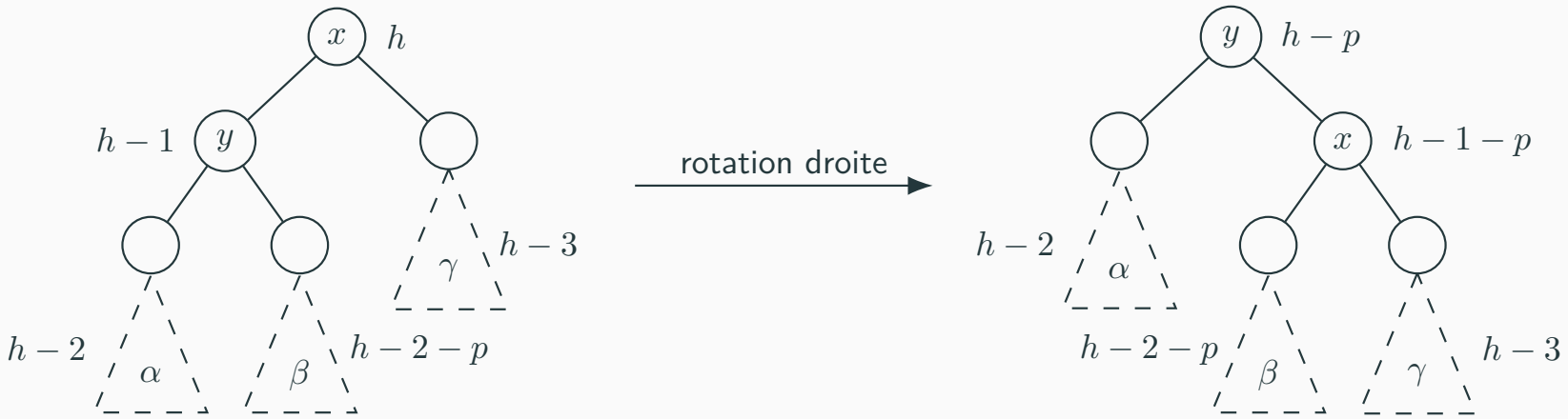


Rééquilibrage

Le sous-arbre de gauche se décompose en une racine y et deux sous-arbres gauche et droit α et β.

On distingue alors deux cas.

Rotations et maintien de la structure d'arbre AVL



Rééquilibrage

Cas 1 : $h(\alpha) \geq h(\beta)$, i.e. $h(\alpha) = h - 2$ et $h(\beta) = h - 2 - p$ avec $p \in \{0, 1\}$.

Dans ce cas, une **rotation droite** suffit, et l'arbre total passe d'une hauteur h à $h - p$: si $p = 1$, le déséquilibre se propage potentiellement plus haut.

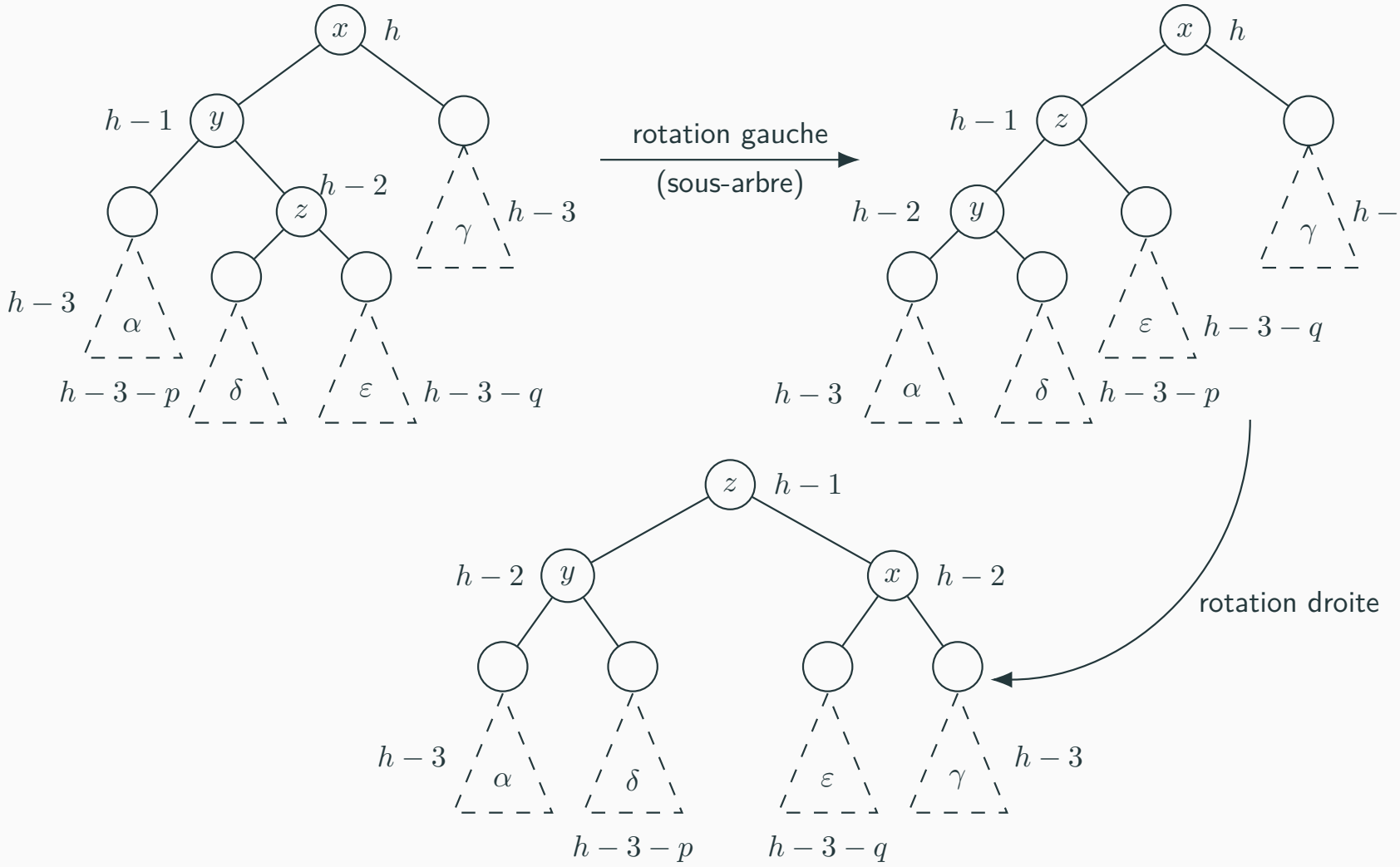
Rééquilibrage

Cas 2 : $h(\alpha) < h(\beta)$. β se décompose en une racine z et deux sous-arbres δ et ε, de hauteur respectives $h-3-p$ et $h-3-q$, avec $p, q \in \{0, 1\}$, l'un au moins étant nul.

On vérifie qu'une **rotation gauche** sur le sous-arbre enraciné en y nous ramène (presque) au cas précédent.

Une **rotation droite** rééquilibre l'arbre total, qui passe d'une hauteur h à une hauteur $h-1$.

Rotations et maintien de la structure d'arbre AVL



Opérations sur les AVL en OCaml

```
1  type 'a avl = Vide | N of int * 'a avl * 'a * 'a avl
```

Idée

On donne simplement l'idée de l'implémentation des opérations sur les **AVL**, qui fera l'objet d'un TP.

Le nouveau champ de type **`int`** indique la **hauteur** du sous-arbre enraciné au nœud.

Ainsi, on n'a pas à recalculer la hauteur des sous-arbres pour savoir s'il faut rééquilibrer l'arbre ou si on est dans le cas 1 ou le cas 2 (cette opération couterait $O(n)$, on perdrait donc la complexité logarithmique des opérations sur les **ABR**).

Opérations sur les AVL en OCaml

Idée

Les schémas précédents permettent d'écrire une fonction `equilibrer : 'a avl -> 'a avl`, qui prend en paramètre un arbre qui est presque un **AVL** : les sous-arbres gauche et droit sont supposés être des **AVL** de hauteur qui diffère d'au plus 2.

Le champ **hauteur** de l'arbre est également possiblement erroné.

La fonction procède si nécessaire à une ou deux rotations, et renvoie un **AVL** (avec champ **hauteur** correct), et ce en **temps constant**.

Opérations sur les AVL en OCaml

Utilisation de la fonction `equilibrer`

Une fois la fonction `equilibrer` écrite, il est facile de réécrire des versions pour arbres **AVL** des fonctions sur les **ABR**.

```
1  let rec inserer a x = match a with
2    | Vide -> N(0,Vide,x,Vide)
3    | N(h,g,y,d) when y=x -> a
4    | N(h,g,y,d) when y<x -> equilibrer (N(h,g,y,inserer d x))
5    | N(h,g,y,d) -> equilibrer (N(h,inserer g x,y,d))
6  ;;
```

Exemple

Voici ci-dessus le code de la fonction d'**insertion**.

Conclusion sur les dictionnaires

Complexité

À l'aide du théorème sur la hauteur d'un **AVL**, on voit que l'on a bien créé une structure de **dictionnaire** où toutes les opérations s'effectuent en $O(\log n)$, où n est le nombre d'éléments stockés.

Tables de hachage

Une autre implémentation classique des **dictionnaires** est celle faisant usage d'une **table de hachage**, que l'on verra en TP.