Chapitre 16 - Structures à l'aide d'arbres binaires

Objectif :

Optimiser l'organisation des étiquettes d'un arbre binaire pour réaliser certaines opérations plus efficacement et pouvoir implémenter d'autres structures abstraites basées sur ces opérations.

I. Arbres binaires de recherche

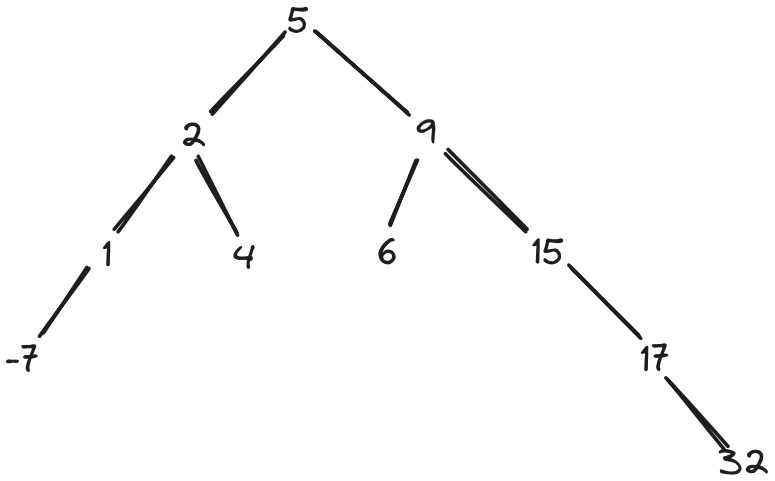
Objectif : Pouvoir rechercher efficacement une étiquette dans l'arbre.

Pour cela, on ne doit avoir qu'une seule branche de l'arbre à explorer.

Idée d'organisations des étiquettes :

Pour tout noeud de l'arbre, toutes les étiquettes de son sous-arbre gauche sont plus petites et celles de son sous-arbre droit sont plus grandes.

Exemple :



Définition inductive des arbres binaires de recherche :

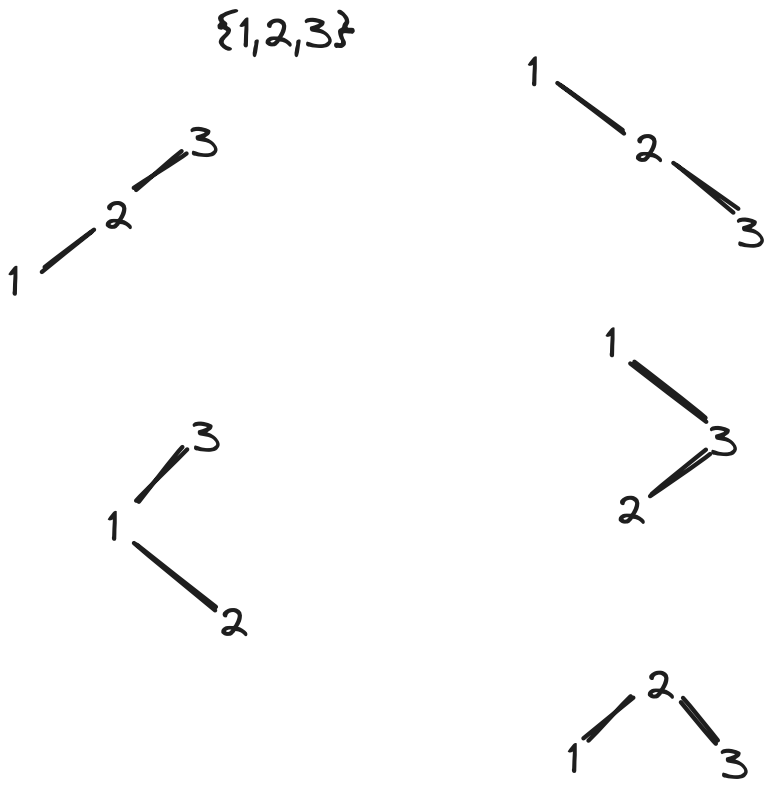
Soit $(E, \prec)$ un ensemble totalement ordonné d'étiquettes.

- Assertion : $\perp$ est un arbre binaire de recherche
- Règle d'inférence : si g et d sont des arbres binaires de recherche, et $e \in E$ une étiquette et $\max e_g \prec e \prec \min e_d$ avec pour convention $max \perp = -\infty$ et

$\min \perp = +\infty$ alors $N(e, g, d)$ est un arbre binaire de recherche.

1. Arbres binaires de recherche

a. Propriétés



Dans le cas général, il existe plusieurs arbres binaires de recherche pour un même ensemble d'étiquettes.

Propriété :

La hauteur h d'un arbre binaire de recherche à n noeuds vérifie :

$$\lfloor \log_2(n) \rfloor \leq h \leq n - 1$$

Propriété :

Un arbre binaire est un arbre binaire de recherche si et seulment si l'énumération de ses étiquettes pour un parcours en profondeur dans l'ordre infixe est croissante.

Preuve par induction structurelle :

Assertion : Trivialement vrai

Règle d'inférence :

Considérons l'arbre A = N(e,g,d).

Sens $\Rightarrow$:

Notons P_g e P_d le parcours infixe de A avec P_g (respectivement P_d) les énumérations du parcours infixe de g (respectivement d). A est un arbre binaire de recherche donc g et d sont des arbres binaires de recherches donc par hypothèse d'induction, P_g et P_d sont croissantes et par définition, e est supérieur à toute étiquette de g donc P_g e croissant et idem e P_d est croissant donc P_g e P_d croissant.

Sens $\Leftarrow$:

Notons P_g e P_d un parcours infixe, supposé croissant.

- P_g est croissant donc par hypothèse d'induction, il existe un arbre binaire de recherche de parcours P_g, noté g.
- De même, on a un arbre binaire de recherche issu de P_ {d}
- Comme P_g e P_d est croissant, on a $\max P_g \prec e \prec \min P_d$ donc l'arbre de parcours infixe de P_g e P_d est un arbre binaire de recherche.

b. Opérations élémentaires

Recherche d'un extremum :

- Minimum : On suit la branche sur la gauche jusqu'au bout
- Maximum : On suit la branche sur la droite jusqu'au bout

Définition inductive du maximum :

- Maximum$(\bot) = -\infty$
- $\mathrm{Maximum(N(e,g,d))} = \begin{cases} \mathrm{e \text{ si } d} = \bot \\ \mathrm{Maximum(d)} \text{ sinon} \end{cases}$

Définition inductive du minimum :

- Minimum$(\bot) = +\infty$
- $\mathrm{Minimum(N(e,g,d))} = \begin{cases} \mathrm{e \text{ si } g} = \bot \\ \mathrm{Minimum(d)} \text{ sinon} \end{cases}$

```
Algorithme minimum(arbre binaire de recherche a):
        si a = ⊥ alors :
                renvoyer + ∞
        sinon si SAG(a) = ⊥ alors
                renvoyer etiq(racine(a))
        sinon
                renvoyer minimum(SAG(a))
```

$C(a)$ est la complexité dans le pire des cas de minimum(a).

$C(\perp) = \mathcal{O}(1)$

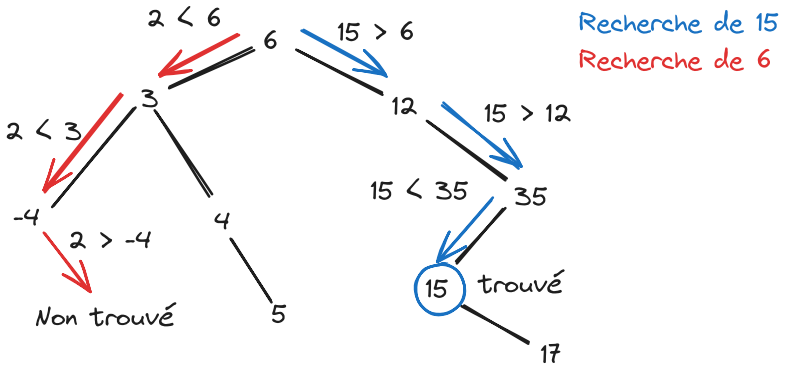
$C(N(e,g,d)) = \mathcal{O}(1) + C(g) \leq \mathcal{O}(1) + \max(C(g), C(d))$

Donc $C(a) = \mathcal{O}(h(a))$ car on reconnaît la définition de la hauteur.

Or $\lfloor \log_2(n) \rfloor \leq h \leq n-1$ avec n le nombre de noeuds de a donc dans le pire des cas, linéaire en nombre de noeuds de l'arbre binaire de recherche.

Idem, la complexité pour le maximum est en $\mathcal{O}(h(a))$.

<u>Recherche d'un élément quelconque :</u>



Recherche(elt,$\perp$) = Faux

$$Recherche(elt, N(e,g,d)) = \begin{cases} \text{Vrai si elt} = \text{e} \\ \text{recherche(elt,g) si elt} < \text{e} \\ \text{recherche(elt,d) sinon} \end{cases}$$

<u>Analyse :</u>

Correction par induction structurelle, complexité en $\mathcal{O}(h(\text{arbre binaire de recherche}))$

Insertion :

→ feuille avec la nouvelle étiquette

→ emplacement trouvé comme pour la recherche

→ souvent, on n'insère pas l'étiquette si elle est déjà présente mais parfois, si c'est utile d'avoir plusieurs fois les mêmes étiquettes, on choisit un sous-arbre (toujours le même) et on continue l'insertion dedans.

Idem, complexité pour max en $\mathcal{O}(h(a))$.

Recherche d'un élément quelconque :

Même schéma que précédent mais quand on tombe sur non trouvé on ajoute l'étiquette.

Définition inductive :

- Insere$(elt, \bot) = N(elt, \bot, \bot)$
- $\text{Insere(elt,N(e,g,d))} = \begin{cases} \text{N(e,g,d) si elt} = \text{e} \\ \text{N(e,Insere(elt,g),d) si elt} < \text{e} \\ \text{N(e,g,Insere(elt,d)) sinon} \end{cases}$

Analyse :

Correction par induction structurelle, complexité en $\mathcal{O}(h(\text{arbre binaire de recherche}))$

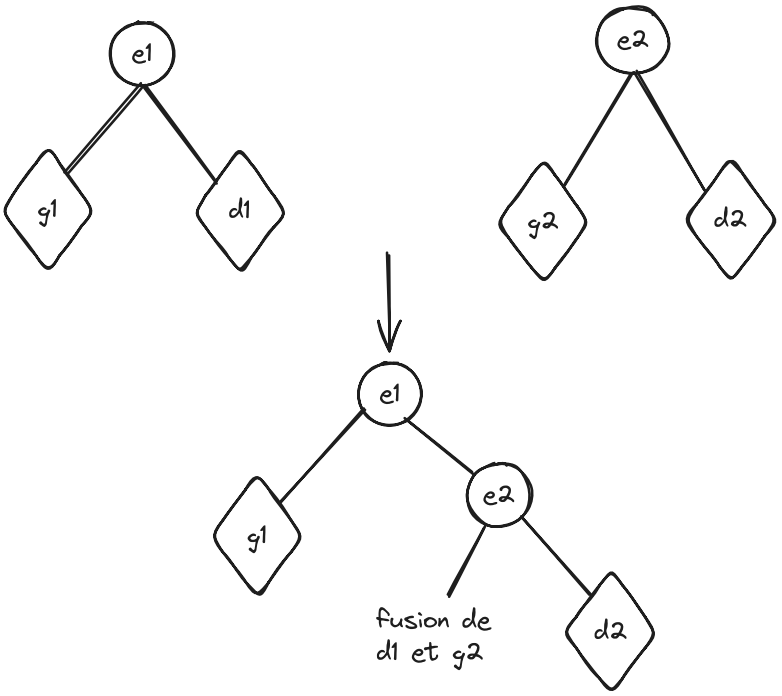
Suppression :

Méthode de la fusion :

→ recherche de l'élément à supprimer dans l'arbre.

→ supprime le noeud trouvé

→ on fusionne ses deux sous-arbres pour les ré-accrocher au père du noeud supprimé.



Définition inductive :

Fusion$(\perp, \perp) = \perp$

Fusion$(\perp, N(e,g,d)) = N(e,g,d)$

Fusion$(N(e,g,d), \perp) = N(e,g,d)$

Fusion$(N(e_1,g_1,d_1,N(e_2,g_2,d_2)) = N(e_1,g_1,N(e_2,Fusion(d_1etg_2),d_2))$

Sup_fusion$(elt, \perp) = \perp$

$$\text{Sup fusion(elt,N(e,g,d))} = \begin{cases} \text{Fusion(g,d) si elt} = \text{e} \\ \text{N(e,sup fusion(elt,g),d) si elt} < \text{e} \\ \text{N(e,g,Sup fusion(elt,d)) sinon} \end{cases}$$

On suppose ici qu'il n'y a pas de doublon.

Analyse : Idem

Méthode de remontée (du minimum/maximum) :

→ recherche de l'élément à supprimer

→ 3 cas possibles

- Si le noeud à supprimer est une feuille, on le supprime simplement
- Si le noeud à supprimer a un seul sous-arbre, on rattache le sous-arbre au père du noeud qu'on supprime.
- Sinon (2 sous-arbres), on ne supprime pas le noeud. Dans ce noeud, on place l'étiquette du maximum du sous-arbre gauche, puis on supprime ce maximum, ou bien le minimum du sous-arbre droit.

Définition inductive :

Sup_min$(elt, \perp) = \perp$

$$\text{Sup min(elt,N(e,g,d))} = \begin{cases} \perp \text{ si elt} = \text{e et g} = \text{d} = \perp \\ \text{d si elt} = \text{e et g} = \perp \neq \text{d} \\ \text{g si elt} = \text{e et d} = \perp \neq \text{g} \\ \text{N(minimum(d),g,Sup min(minimum(d),d)) si elt} = \text{e et d} \neq \perp \text{ et g} \neq \\ \text{N(e,Sup min(elt,g),d) si elt} < \text{e} \\ \text{N(e,g,Sup min(elt,d)) sinon} \end{cases}$$

Analyse : idem

Remarque sur les complexités :

La complexité des opérations dépend de la complexité des comparaisons des éléments de $(E, \prec)$. Ce n'est efficace que si les comparaisons ont un coût constant.

c. Implémentation d'un tableau associatif

Le tableau associatif est une structure de données abstraite, constituée d'associations clé-valeur.

- Créer un tableau associatif vide
- Ajoute une association clé/valeur au tableau associatif
- Accès à une valeur depuis la clé
- Test de présence d'une clé
- Supprimer une association

Arbre binaire de recherche	**Table de hachage**
$\mathcal{O}(1)$ car arbre binaire de recherche vide	$\mathcal{O}(\text{taille de la table})$
insère association en $\mathcal{O}(h(\text{arbre binaire de recherche}))$ Organisation arbre binaire de recherche avec les clés	$\mathcal{O}(1)$ amorti si fonction hachage efficace et collisions bien gérées
Recherche de la clé : $\mathcal{O}(h(\text{arbre binaire de recherche}))$	//

Arbre binaire de recherche	Table de hachage
Recherche : $\mathcal{O}(h(\text{arbre binaire de recherche}))$	//
suppression : $\mathcal{O}(h(\text{arbre binaire de recherche}))$	//

Les arbres binaires de recherche sont moins efficace que la table de hachage, mais nettement plus facile à implémenter.

2. Arbres équilibrés (arbres bicolores)

Définition :

Un arbre binaire de recherche a est dit équilibré si $h(a) = \mathcal{O}(\log(n))$ avec n le nombre de noeuds de a.

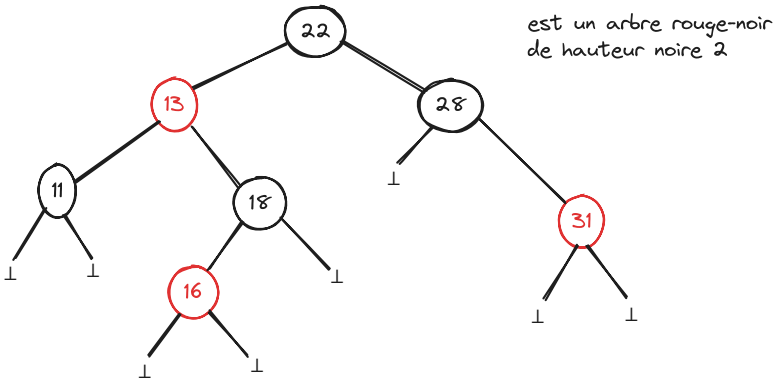
Définition :

Un arbre bicolore (en arbre rouge-noir) est un arbre binaire de recherche dont les noeuds sont colorés en rouge ou en noir en respectant les contraintes :

- La racine est noire.
- Aucun noeud rouge n'a un fils rouge.
- Tous les chemins de la racine à un $\perp$ ont le même nombre de noeuds noirs.

Le nombre de noeud moins de la racine aux $\perp$ est appelé "hauteur noire" de l'arbre rouge-noir, noté $h_N(\text{arbre binaire rouge-noir})$.

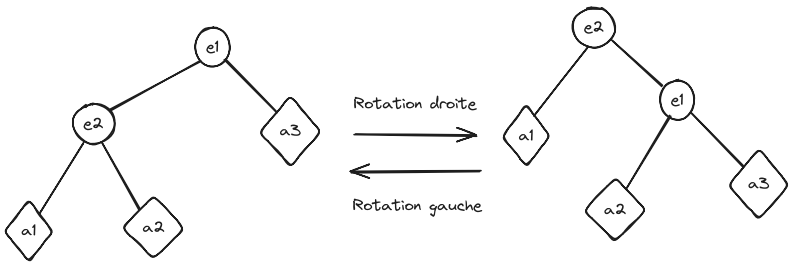
Exemple :



a. Opérations élémentaires

→ Recherches, idem que pour les arbres binaires de recherche

→ Rotations :



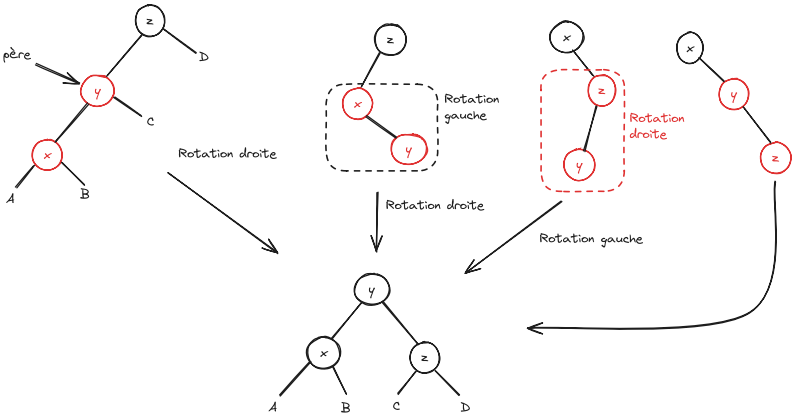
Les couleurs ne changent pas.

Insertion :

Pour ne pas modifier la hauteur noire, la feuille insérée est rouge. On règle les problèmes avec des rotations sur un noeud noir ayant deux noeuds consécutifs rouges, le noeud devient rouge et ses deux fils noirs.

On répercute les corrections de couleurs en remontant vers la racine. A la fin, on remet la racine en noir si nécessaire.

Correction des conflits rouge/rouge :



On corrige un par un tous les conflits. A la fin, on remet la racine en noir.

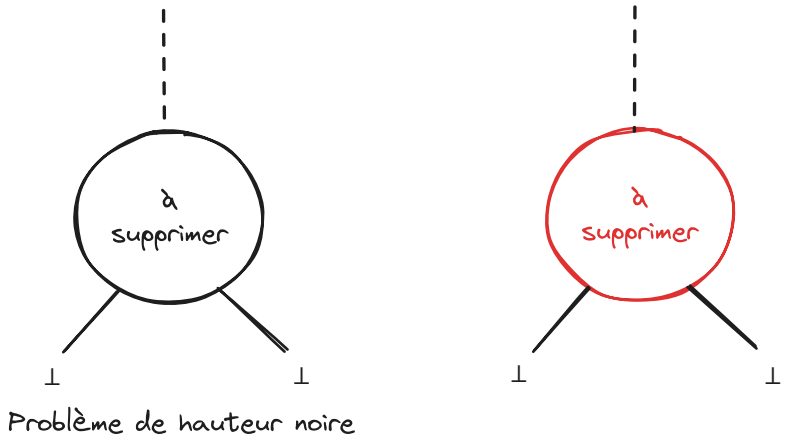
Complexité :

Les rotations sont en $\mathcal{O}(1)$, donc on ajoute uniquement des opérations en $\mathcal{O}(1)$ après chaque appel récursif pour l'insertion, donc la complexité est toujours en $(hauteur(arbre))$

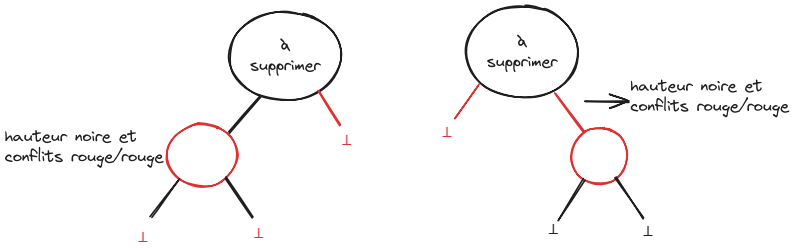
Suppression :

On utilise la méthode de remontée d'un extrémum.

Si le noeud à supprimer est une feuille :



Si le noeud à supprimer a un fils :



Si le noeud à supprimer a 2 fils :

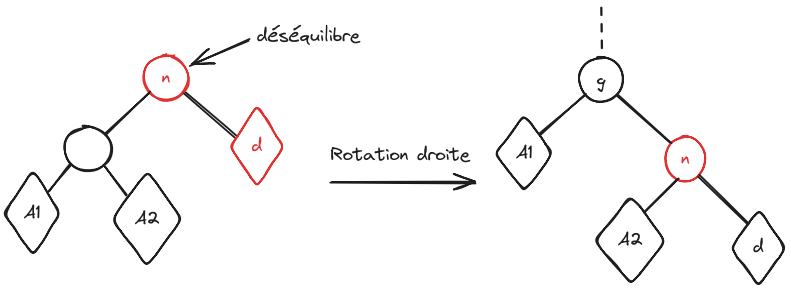
→ aucun problème pour le noeud à supprimer car seule son étiquette change.

→ La suppression du minimum/maximum amène à 1 des 2 cas précédents.

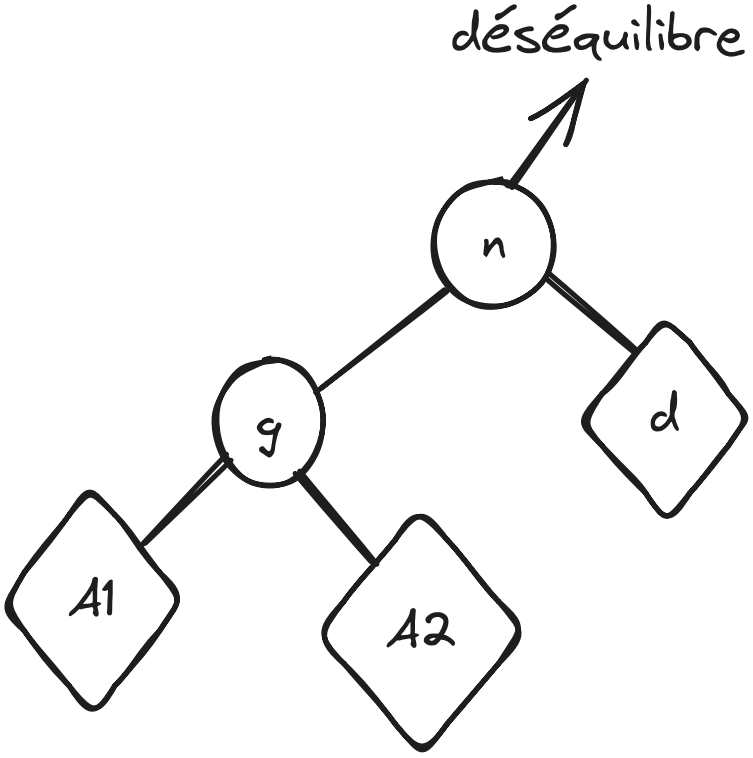
Sinon, on corrige les hauteurs noires. On a une différence de 1 entre les hauteurs des sous-arbres gauche et sous-arbre droit d'un noeud.

On se place dans le cas où la hauteur noire du sous-arbre gauche d'un noeud est plus grande de 1 que celle du sous-arbre droit.

Premier cas :

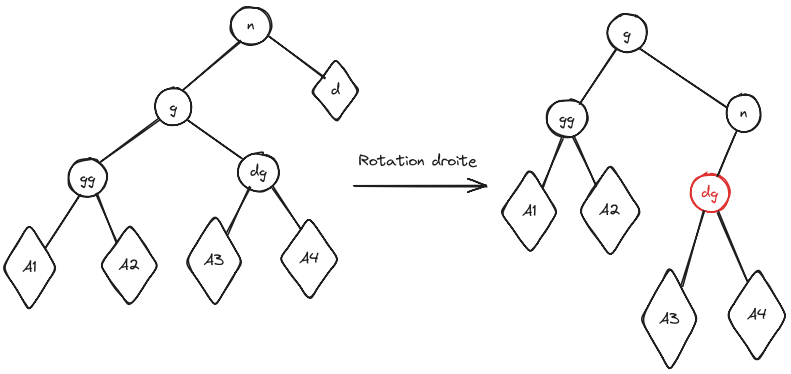


Deuxième cas :



On colorie n en rouge et on résout comme dans le premier cas, puis on répète les corrections de hauteurs noires sur le reste de l'arbre.

Troisième cas :



3 cas symétriques, si la hauteur noire est inférieure dans le sous-arbre gauche, on règle le problème avec une rotation à gauche, et recoloration.

Ensuite, on corrige tous les conflits rouge/rouge. On finit par remettre la racine en noire si nécessaire.

Complexité :

A nouveau, on ajoute à chaque appel récursif des rotations et des changements de couleur en $\mathcal{O}(1)$, donc la complexité est en $\mathcal{O}(hauteur(arbre))$.

b. Propriété des arbres rouges/noirs

Première propriété :

Soit a un arbre bicolore, $h(a) \leq 2 \times h_N(a)$.

Preuve :

Le plus long chemin de la racine à un $\perp$ alterne les noeuds noirs et rouges.

Deuxième propriété :

Soit a un arbre bicolore $2^{h_N(a)} - 1 \leq t(a)$

Preuve :

Par récurrence sur $t(a)$.

Théorème :

Les arbres bicolores sont équilibrés, autrement dit $h = \mathcal{O}(\log(n))$ avec n la taille de l'arbre.

Preuve :

D'après les propriétés 1 et 2, $2^{h(a)/2} - 1 \leq 2^{h_N(a)} - 1 \leq t(a)$. Donc $h(a) = \mathcal{O}(\log(t(a)))$. Donc toutes les opérations sur les arbres rouges/noirs sont logarithmiques en la taille de l'arbre.

II. Tas

Objectif : Obtenir une extraction du minimum/maximum efficace.

Pour avoir un accès rapide (en $\mathcal{O}(1)$) au maximum/minimum, il faut qu'il soit à la racine.

Définition :

Soit $(E, \prec)$ un ensemble totalement ordonné. Soit A un arbre. On dit que A respecte la propriété d'ordre des tas, si pour tout noeud de A, pour tout fils f de n, $etiq(n) \prec etiq(f)$.

Définition :

Un tas est un arbre binaire nécessairement complet à gauche, qui respecte la propriété d'ordre des tas.

Deux tas particuliers sur les ensembles $(\mathbb{Z}, \leq)$ et $(\mathbb{Z}, \geq)$ sont appelés tas-min et tas-max.

Propriétés immédiates :

Si un noeud d'étiquette e est un ancêtre d'un noeud d'étiquette e', alors $e \prec e'$.

Le minimum d'un tas-min est à la racine, le maximum d'un tas-max est à la racine.

Les tas sont équilibrés, c'est-à-dire que $h(tas) = \mathcal{O}(\log(t(tas)))$.

a. Opérations élémentaires (ici sur les tas-max)

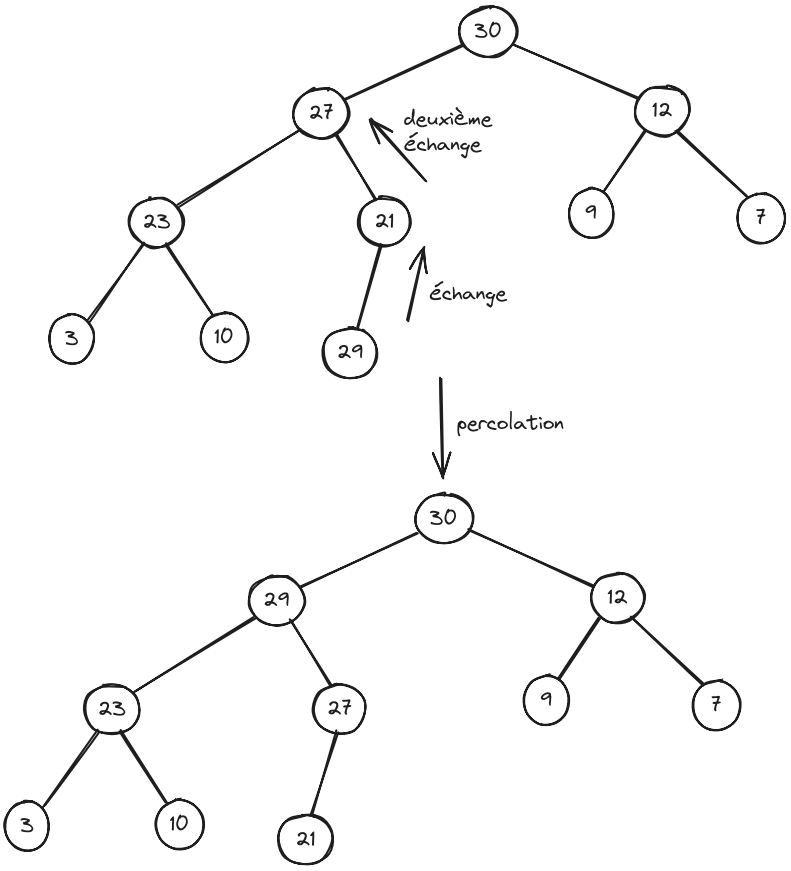
Lecture du maximum (lecture de la racine): $\mathcal{O}(1)$.

Percolations :

C'est une opération qui vise à modifier par échanges successifs un arbre binaire complet à gauche pour lequel un unique noeud ne respecte pas la propriété d'ordre des tas.

On le fait vers le haut si le non respect se fait avec le père, vers le bas si le non respect se fait avec les fils.

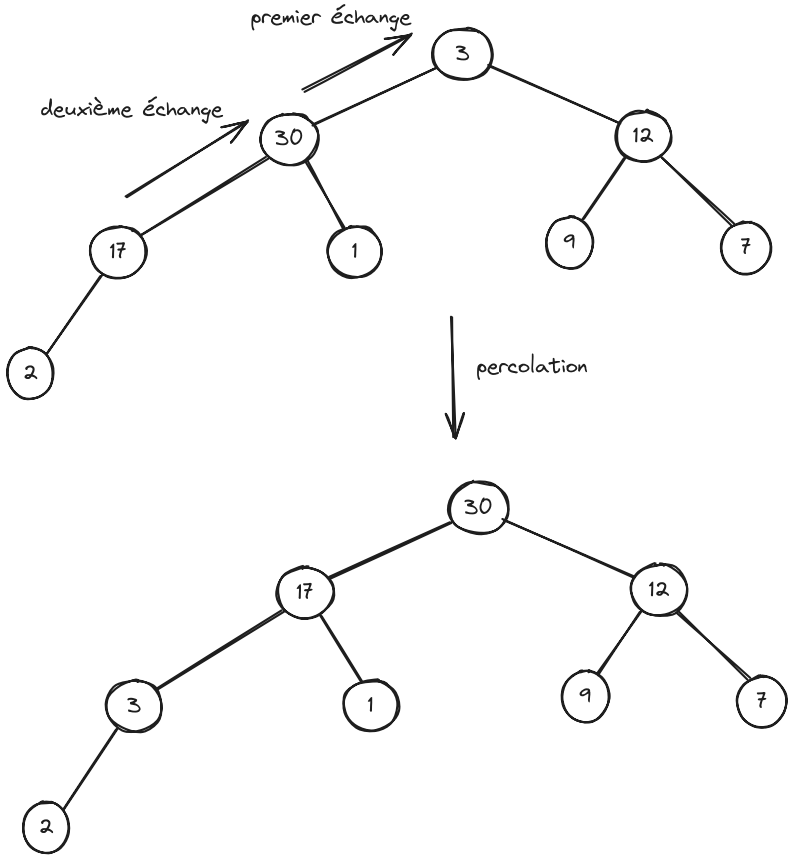
Exemple :



Algorithme :

```
Perco_haut(arbre A, noeud n appartenant à A)
        Si n différent de racine de A et etiq(père(n)) < etiq(n):
                échanger etiq(n), etiq(pere(n))
                perco_haut(A, pere(n))
```

Exemple :



Algorithme similaire avec plusieurs “si” pour distinguer les cas si l’échange se fait avec le fils gauche ou droit.

Complexité :

$C_p =$ complexité de perco_haut pour n qui est à la profondeur p dans A

$C_0 = \mathcal{O}(1)$

$C_p = \mathcal{O}(1) + C_{p-1}$

donc $C_p = \mathcal{O}(p)$

La percolation vers le haut est en $\mathcal{O}(hauteur(tas))$. Même raisonnement pour vers le bas.

Insertion :

On place une feuille au seul endroit possible pour que l’arbre reste complet à gauche. On fait ensuite une percolation vers le haut pour faire respecter la propriété de tas.

Complexité : $\mathcal{O}(\log(\text{taille du tas}))$

Suppression du maximum :

On échange l'étiquette de la racine avec celle de la feuille dont la suppression permet de garder un arbre binaire complet à gauche ("tout en bas à droite"). On supprime simplement cette feuille, l'arbre reste complet à gauche puis on effectue une percolation vers le bas de la racine.

Complexité : $\mathcal{O}(\log(\text{taille du tas}))$

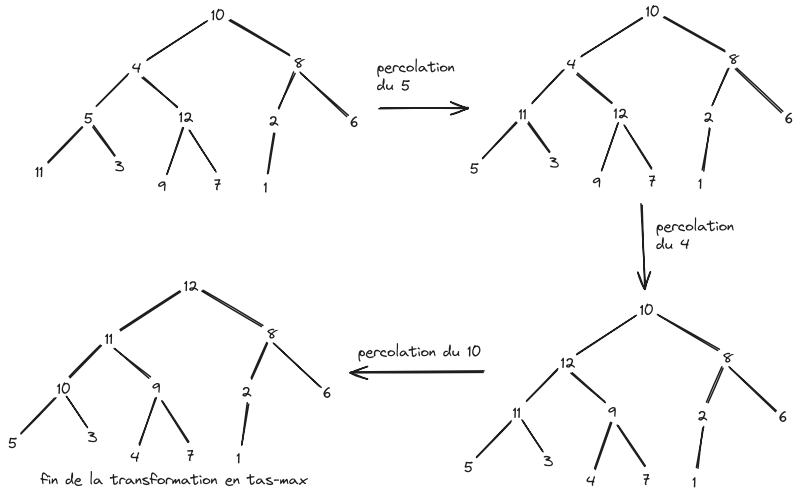
b. Tri par tas

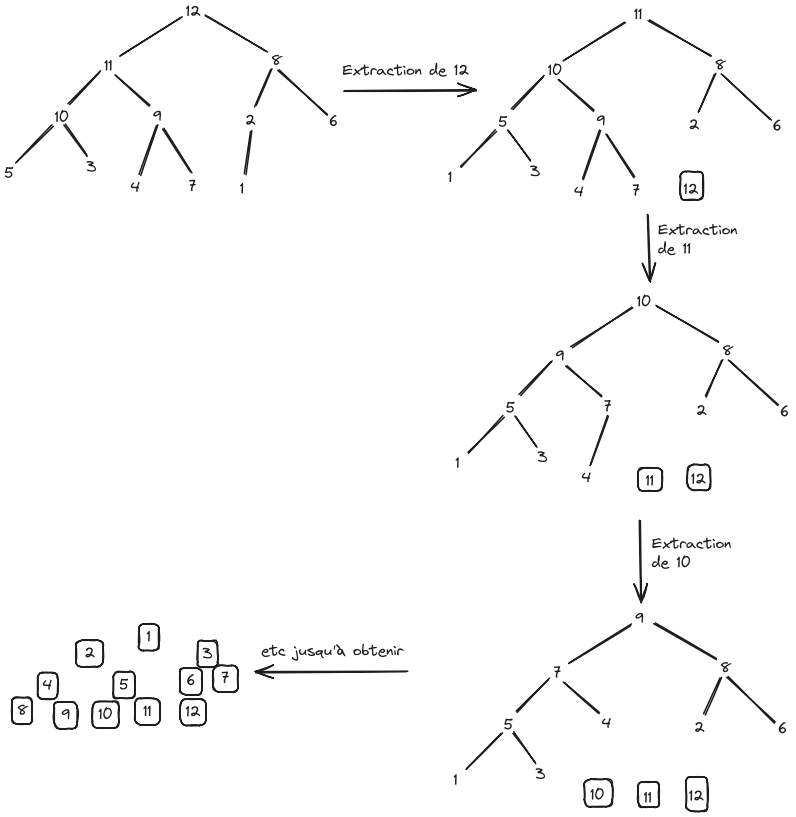
Objectif :

→ Transformer le tableau en tas puis faire des extractions.

→ Généralement, on utilise un tas-max.

Exemple : $10, 4, 8, 5, 12, 2, 6, 11, 3, 9, 7, 1$ (ordre de parcours en largeur)





Algorithme :

```
Tri_par_tas(tableau T de taille n)
// transformer T en tas-max
Pour i allant de partie entieère de n-2/2 à 0 :
        perco_bas(T;i)
// extractions successives du maximum
Pour i allant de n à 1 :
        échange(T,0,i-1)
        perco_bas(T,0)
```

Analyse :

Preuve de correction :

Variants : i pour les deux boucles.

Invariants :

Première boucle : “Tous les arbres enracinés en T_j avec $j > i$, sont des tas-max.”

Tous les arbres sont enracinés en T_j avec $j > \lfloor \frac{n-2}{2} \rfloor$ sont des feuilles, donc des tas. A la fin, l’arbre enraciné en T_i est un tas car ses 2 sous-arbres sont des tas (invariant vrai

au début de l'itération) et on percole le noeud i ce qui le place au bon endroit. Après la boucle, T est un tas-max.

Deuxième boucle : "les éléments d'indice i à $n-1$ sont les plus grands éléments triés."

Complexité spatiale : $\mathcal{O}(1)$

Complexité temporelle :

Première boucle : On percole tous les noeuds aux profondeurs $0 \leq i \leq \text{hauteur}(\text{tas}) - 1$, donc à la profondeur i, on fait $\mathcal{O}(2^i)$ percolations, chacune en $\mathcal{O}(h-i)$ comparaisons.

$\mathcal{O}\left(\sum_{i=0}^{h-1} 2^i(h-i)\right) = \mathcal{O}(\sum_{j=1}^{h} j \times 2^{h-j}) = \mathcal{O}(2^h \sum_{j=1}^{h} \frac{j}{2^j}) = \mathcal{O}(2^h)$ (car somme converge vers 2).

Or $h = \lfloor \log_2(n) \rfloor$ avec n le nombre d'éléments à trier. Donc la première boucle est en $\mathcal{O}(n)$.
Deuxième boucle :

$$\mathcal{O}\left(\sum_{i=0}^{n} (\mathcal{O}(log(n) + \mathcal{O}(1))\right) = \mathcal{O}(n \times log(n))$$

Total : $\mathcal{O}(n) + \mathcal{O}(n \times log(n)) = \mathcal{O}(n \times log(n))$.

c. File de priorité

Une file de priorité est une structure de données abstraite, qui stocke des valeurs prises à des priorités quelconque, associées à des priorités qui appartiennent à un ensemble totalement ordonné.

Interface :

- Créer une file de priorité vide : $() \rightarrow$ File de priorité vide.
- Ajouter un élément à une file de priorité associée à une certaine priorité :

  File de priorité, priorité $\rightarrow ()$
- Extraire l'élément le plus prioritaire d'une file de priorité :

  File de priorité $\rightarrow$ valeur
- Test de vacuité :

File de priorité $\rightarrow$ booléen

Implémentation :

$\rightarrow$ Non efficace avec une liste (au moins 1 opération linéaire)

$\rightarrow$ avec un tas (dans l'ordre avec n le nombre d'éléments) :
$\mathcal{O}(1), \mathcal{O}(log(n)), \mathcal{O}(log(n)), \mathcal{O}(1)$

Applications :

- Ordonnancement des processus
- Algorithme de Dijkstra
- Algorithme de Huffman