



Chap 09 · Arbres binaire de recherche

Prise de note par Léo BERNARD en MP2I au lycée du Parc.



Pour nos amis les arbres à l'envers.

- Chap 09 · Arbres binaire de recherche
- I. Ensembles, dictionnaires
 - 1.1. Type abstrait SET
 - 1.2. Type abstrait MAP
- II. Arbres binaires des recherche
 - 2.1. Définitions
 - 2.2. Algorithmes élémentaires
 - 2.2.a. Recherche dans un ABR
 - 2.2.b. Insertion
 - 2.2.c. Suppression
 - 2.3. Réalisation d'un dictionnaire à partir d'un ABR
- III. Tables de hachage
 - 3.1. Un cas simple
 - 3.2. Principe
 - 3.3. Résolution par chaînage
 - 3.4. Critères pour une bonne fonction de hachage
 - 3.5. Résolution par adressage ouvert
- IV. Arbres auto équilibrés
 - 4.1. Arbres 2-3
 - 4.1.a. Présentation
 - 4.1.b Recherche dans un arbre 2-3
 - 4.2.c. Insertion dans un arbre 2-3
 - 4.2. Arbre rouge-noir
 - 4.2.a Définition
 - 4.2.b. Recherche
 - 4.2.c. Insertion

I. Ensembles, dictionnaires

1.1. Type abstrait SET

La structure de données **SET** correspond à la notion mathématique d'ensemble.

- pas de répétition,

- pas d'ordre

Une signature possible (pour des ensembles *fonctionels*) :

```
* `member: 'a -> 'a set -> bool (test d'appartenance)
* `add: 'a -> 'a set -> 'a set
* `remove: 'a -> 'a set -> 'a set
* `empty_set: 'a set
* `is_empty: 'a set -> bool
* `equal: 'a set -> 'a set -> bool
* `iter: ('a -> unit) -> 'a set -> unit (boucle for)
```

On peut préciser un peu la spécification de ces fonctions. Pour tout objet `s` de type `'a set`, on associe l'ensemble mathématique correspondant $\varphi(s)$.

On exige que :

- $\varphi(\text{empty_set}) = \emptyset$
- $\text{is_empty } s \iff \varphi(s) = \emptyset$
- $\varphi(\text{add } x \ s) = \varphi(s) \cup \{x\}$
- $\varphi(\text{remove } x \ s) = \varphi(s) \setminus \{x\}$
- $\text{member } x \ s \iff x \in \varphi(s)$
- $\text{equal } s \ t \iff \varphi(s) = \varphi(t)$
- si $\varphi(s) = \{x_1, x_2, \dots, x_n\}$, alors `iter f s` a le même effet que `List.iter f [x1, x2, ..., xn]`

 L'ordre dans un SET est arbitraire.

 Ex :

On dispose d'un type `'a set` doté des opérations précédentes.

Écrire une fonction `cardinal_liste : 'a list -> int` qui renvoie le nombre d'éléments *distincts* de la liste donnée en argument.

```
let cardinal_liste li =
  let aux li set n =
    match li with
    | [] -> n
    | x::xs ->
      if member x set then
        aux li set n
      else
        aux xs (add x ens) (n+1)
      (* Ou alors : *)
      (* aux xs (add x set) (if member x set then n else n+1) *)
    in aux li empty_set 0
;;
```

```

let cardinal_liste li =
  let set = ref empty_set in
  let n = ref 0 in
  let f x =
    if not (member x !set) then
      incr n; (*ajoute 1 à n*)
      ens := add x !ens
  in List.iter f li;
  !nb

```

1.2. Type abstrait MAP

Le type abstrait **MAP** (ou **DICT**) correspond à un dictionnaire, i.e. une *application partielle* (pas tous les éléments de l'ensemble d'entrée ont une sortie) d'un ensemble *A* vers un ensemble *B* (on parle aussi parfois de tableau associatif).

Rappel: Type **'a option** en OCaml

```

type 'a option =
| None
| Some of 'a

```

Permet de définir des fonctions qui renvoient un élément de type **'a** (**Some x**) ou rien (**None**)

Signature fonctionnelle :

```

* `get: 'a -> ('a, 'b) map -> 'b option (*None si clé absente*)
* `set: 'a -> 'b -> ('a, 'b) map -> ('a, 'b) map
  (*Ajout d'une association ou mise à jour si la clé
   était déjà présente*)
* `remove: 'a -> ('a, 'b) map -> ('a, 'b) map
* `empty_map: ('a, 'b) map
* `iter: ('a * 'b -> unit) -> ('a, 'b) map -> unit (*boucle for*)
  (* ou ('a * 'b -> unit) si on veut *)

```

Formellement, on associe à tout dictionnaire **d: ('a, 'b) map** une relation fonctionnelle $\varphi(d)$ $\subset A \times B$, i.e. un ensemble de couples vérifiant :

$$((x, y) \in \varphi(d) \text{ et } (x, y') \in \varphi(d)) \implies y = y'$$

Note à soi-même: c'est la définition ensembliste d'une fonction.

On exige :

- $\varphi(\text{empty_map}) = \emptyset$
- $\text{get } x \text{ d} =$
 - **Some y** si $(x, y) \in \varphi(d)$

- `None` s'il n'existe pas de couples de la forme (x, y) dans $\varphi(\boxed{d})$.
- $\varphi(\text{remove } x \boxed{d}) = \varphi(\boxed{d}) \setminus \{(x, y) ; y \in B\}$
- $\varphi(\text{set } x y \boxed{d}) = \varphi(\text{remove } x \boxed{d}) \cup \{(x, y)\}$

 Ex: La manière la plus simple de réaliser ce type abstrait est d'utiliser une liste de couples `(clé, valeur)` tels que les clés soient 2 à 2 distinctes.

```
type ('a, 'b dict) = ('a * 'b) list
```

Écrire en OCaml les fonctions `get` et `set` respectant la spécification et donner leur complexité.

```
let rec get key dict =
  match dict with
  | [] -> None
  | (k,v) :: tail ->
    if k = key then Some v
    else get key tail
;;
;

let rec set key value dict =
  match dict with
  | [] -> [(key, value)]
  | (k,v) :: tail ->
    if k = key then (k, value) :: tail
    else (k,v) :: (set key value tail)
;;
;
```

 Ex:

On considère un `'a array` comme une application $f : \llbracket 0, n \rrbracket \rightarrow \boxed{a}$

Écrire une fonction antécédent : `'a array -> ('a, int list) dict` qui renvoie un dictionnaire `m` dont les clefs sont les $y \in f(\llbracket 0, n \rrbracket)$ et où la valeur associée à un y est la liste des $x \in \llbracket 0, n \rrbracket$ tel que $f(x) = y$.

```
0 1 2 3 4 5 6 7
[| a; a; b; c; b; c; a; c |]

a -> [0; 1; 6]
b -> [2; 4]
c -> [3; 6; 7]
```

```
(* itératif *)
let antecedent tab =
  let dict = ref empty_map in
  for i = 0 to Array.length tab - 1 do
    let e = tab.(i) in
```

```

match get e dict with
| None -> d := set e [i] !d
| Some l -> d := set e (i::l) !d
done;
!d
;;

(* fonctionnel *)
let antecedent tab =
  let rec aux dict i =
    if i = Array.length tab - 1 then (
      dict
    ) else (
      let e = tab.(i) in
      match get tab.(i) dict with
      | None ->
          let new_dict = set e [i] dict in
          aux new_dict (i+1)
      | Some li ->
          let new_dict = set e (i::li) dict in
          aux new_dict (i+1)
    )
  in
  aux empty_map 0
;;

```

II. Arbres binaires des recherche

Dans la suite, on pourra abréger "arbre binaires des recherche" à "ABR" ("BST" pour Binary Search Tree en anglais).

2.1. Définitions

On travaille ici avec un ensemble d'étiquettes muni d'une relation d'ordre (totale).

On travaille aussi avec des arbres binaires non stricts :

$$\begin{aligned} \perp &\in \mathcal{T}(A) \\ (x, g, d) &\in \mathcal{T}(A) \text{ si } \begin{cases} x \in A \\ (g, d) \in \mathcal{T}(A)^2 \end{cases} \end{aligned}$$

Initialement, un ABR est un arbre binaire où l'étiquette de chaque nœud est :

- strictement plus grande que toutes les étiquettes de son sous-arbre gauche.
- strictement plus petite que toutes les étiquettes de son sous-arbre droit.

 Définition plus formelle :

- Si $T \in \mathcal{T}(A)$, on définit $\varphi(T)$ par :
 - $\varphi(\perp) = \emptyset$
 - $\varphi(x, G, D) = \{x\} \cup \varphi(G) \cup \varphi(D)$

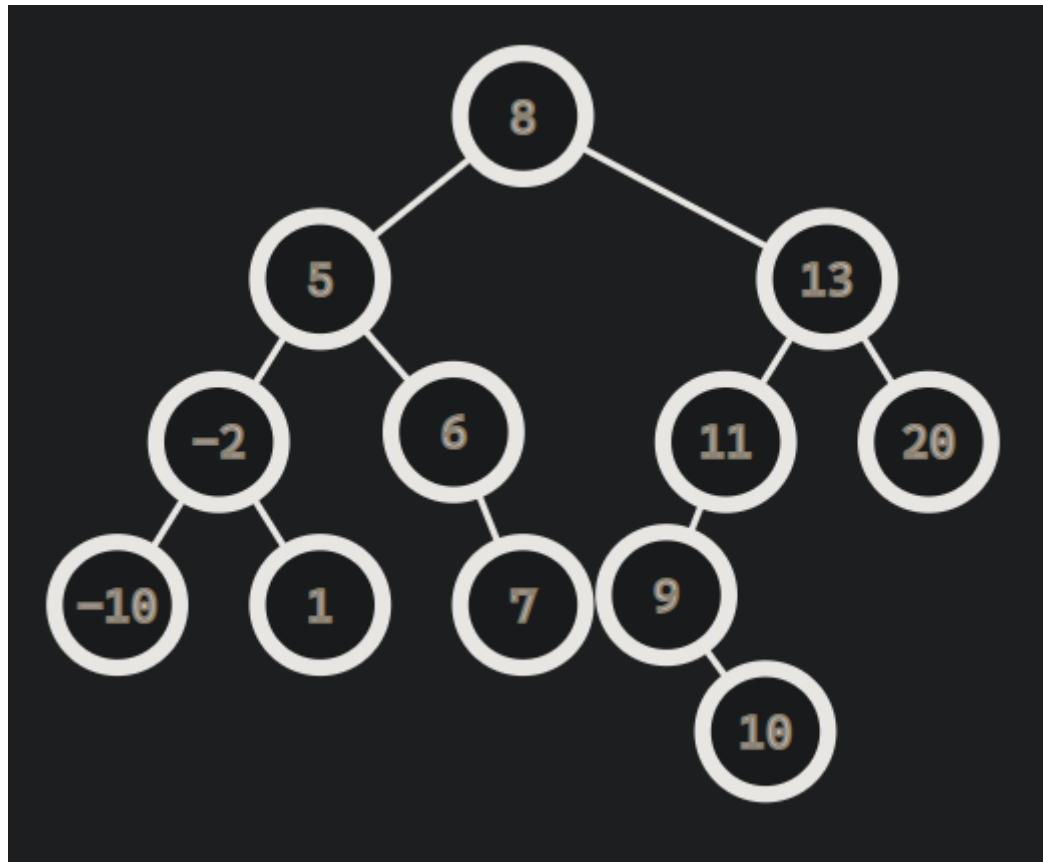
L'ensemble $\text{ABR}(A)$ (ou $\text{BST}(A)$) des arbres binaires de recherche sur A est défini par

- $\perp \in \text{ABR}(A)$
- $(x, g, d) \in \text{ABR}(A)$ si et seulement si :
 - g et d sont dans $\text{ABR}(A)$
 - $\max(\varphi(g)) < x < \min(\varphi(d))$

Remarque : Une feuille est de la forme (x, \perp, \perp) . Par convention :

$$\begin{aligned}-\infty &= \max(\emptyset) = \max(\varphi(\perp)) < x \\ +\infty &= \min(\emptyset) = \min(\varphi(\perp)) > x\end{aligned}$$

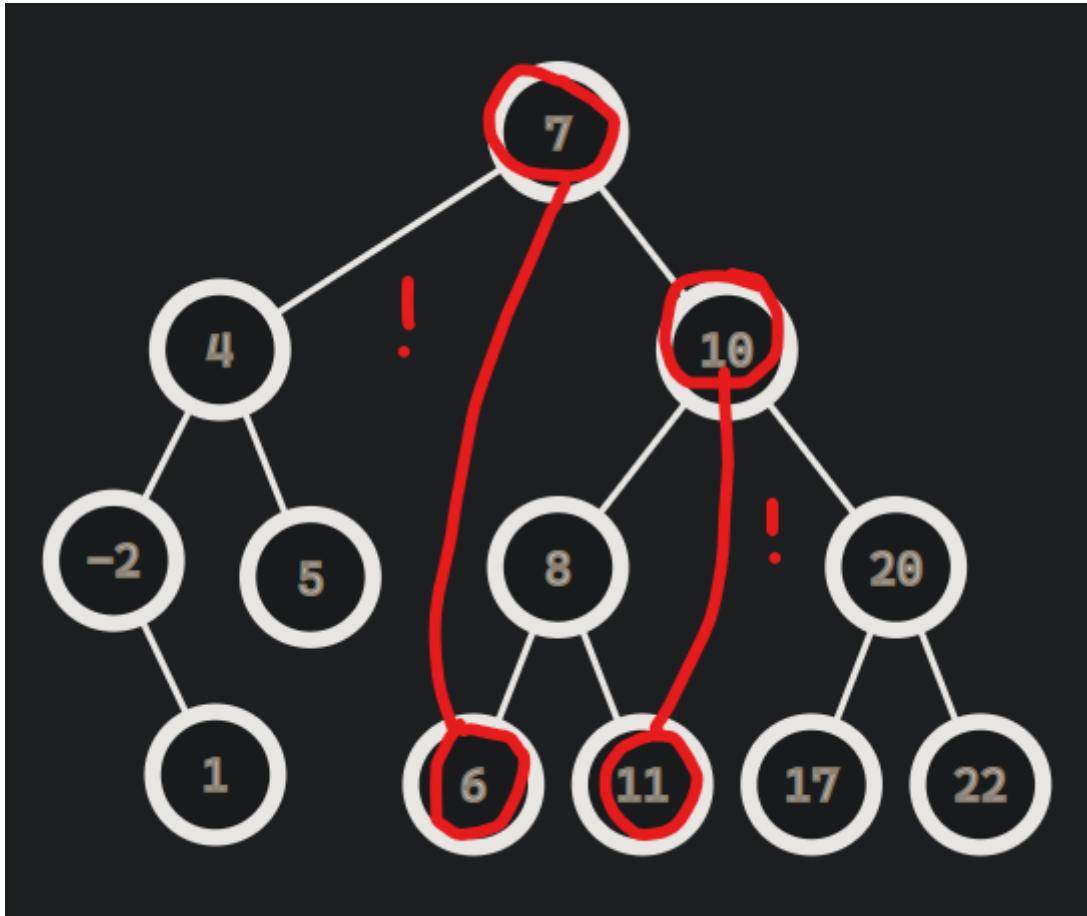
 Ex :



C'est un ABR pour l'ensemble $\{-10, -2, 1, 5, 6, 7, 8, 9, 10, 11, 13, 20\}$.

⚠ Attention : La propriété d'ABR n'est pas *locale* : il ne suffit pas de vérifier que chaque nœud a une étiquette plus grande que son fils gauche et plus petite que celle de son fils droit.

 Exemple :



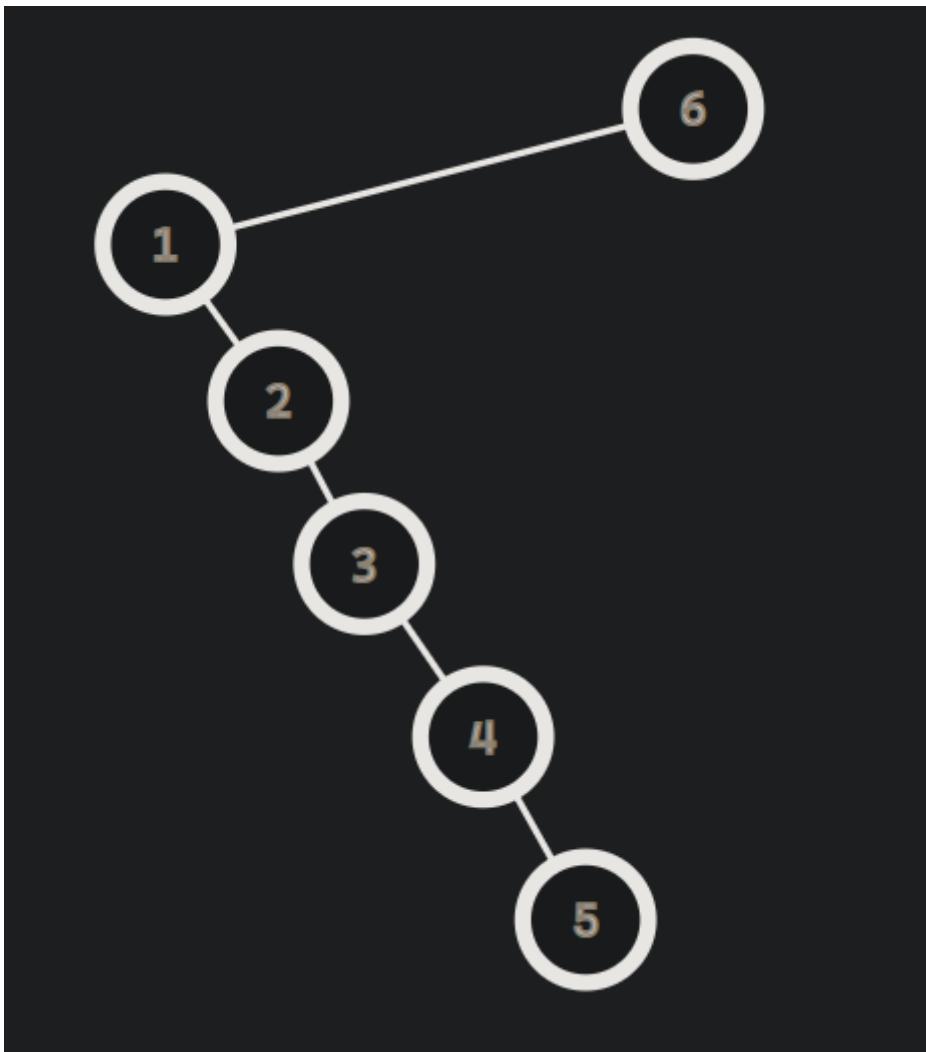
La valeur des fils gauche et droit respectent l'inégalité $g < r < d$ localement, mais ce n'est pas un ABR.

👉 Remarque : Pour un ensemble donné, on peut construire plusieurs ABR.

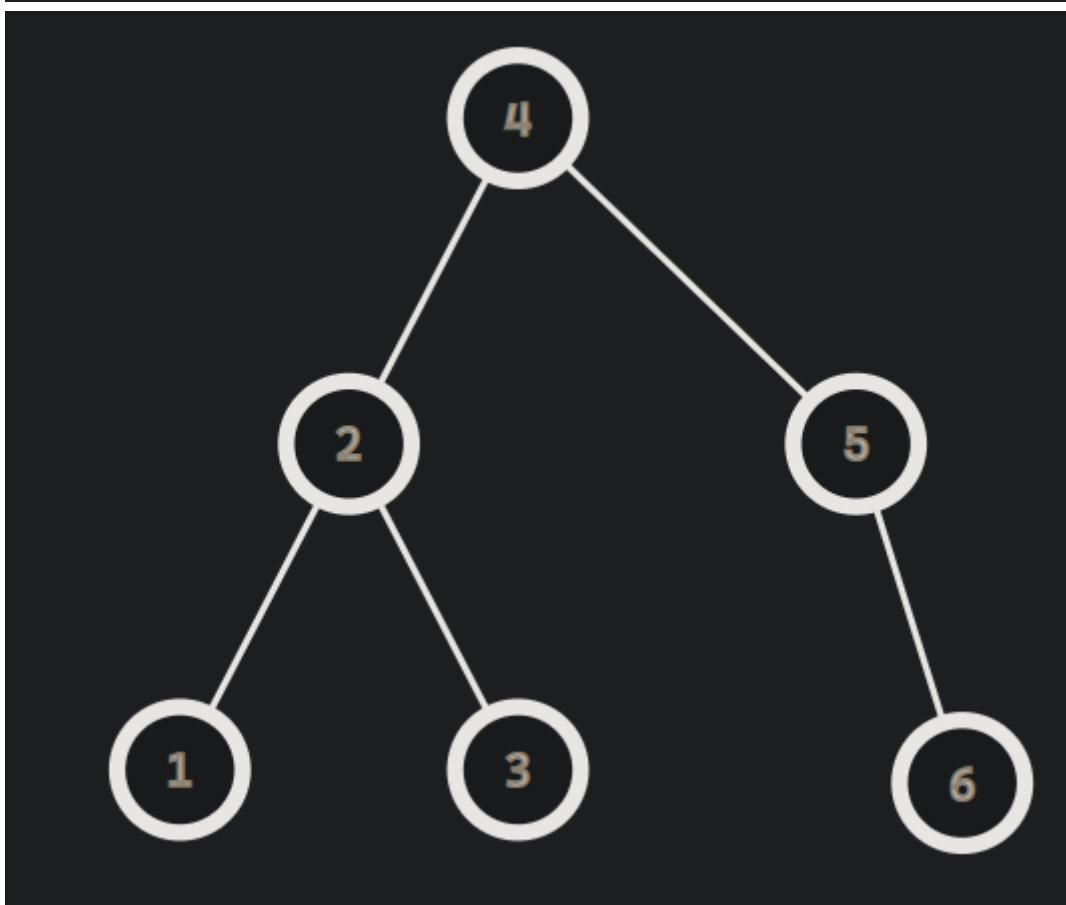
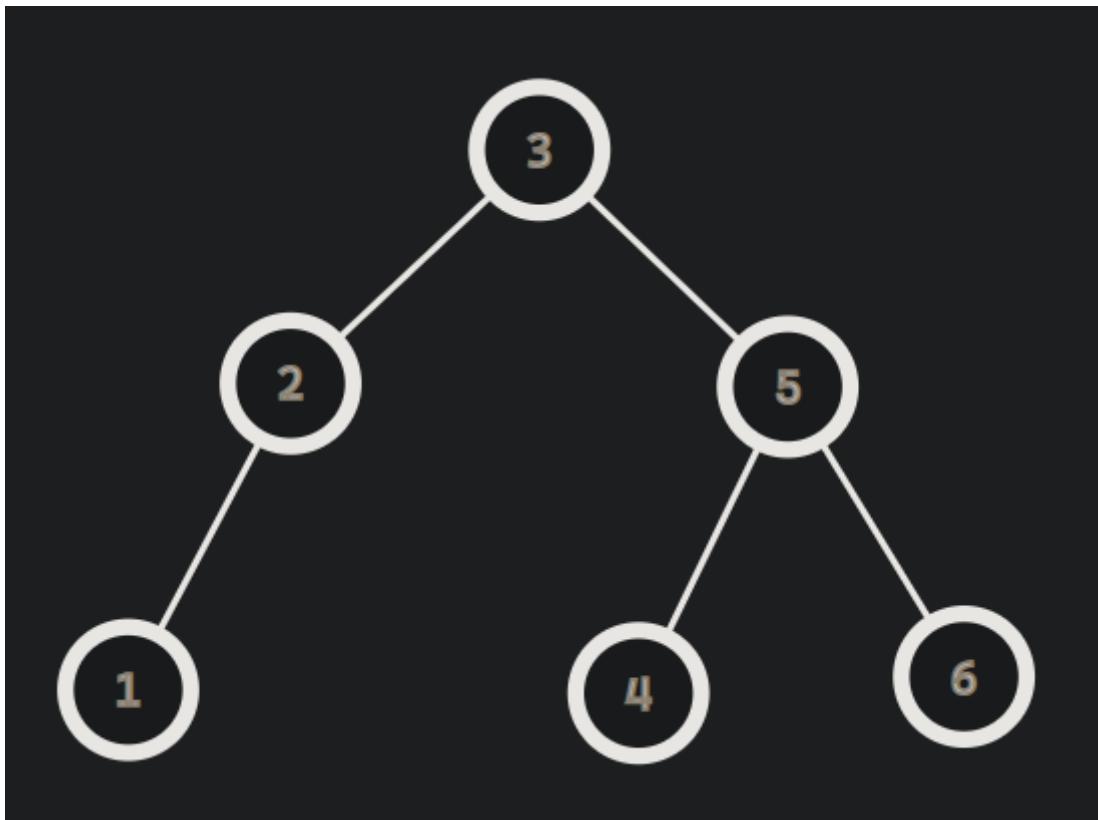
📚 Exemple :

$$E = \{1, 2, 3, 4, 5, 6\}$$

Donner un ABR de hauteur maximale et deux de hauteur minimale pour E .



Hauteur maximale : hauteur 5



Hauteur minimale : hauteur 2

Théorème :

T est un ABR si et seulement si, $\text{infixe}(T)$ est triée par ordre croissant.

Preuve : Par induction structurelle.

- si $T = \perp$, rien à prouver lol
- sinon, T est de la forme (x, G, D) . On note l_G, l_D, l_T la liste des étiquettes de G, D , et T lus en ordre infixé.

Par définition, on a $l_T = l_G, x, l_D$ (les virgules sont des concaténations).

- Supposons que T soit un ABR.
 - G et D sont des ABR par définition. Ainsi, l_G et l_D sont triés par hypothèse d'induction.
 - $\max(\varphi(G)) < x < \min(\varphi(D))$ donc $\max(l_G) < x < \min(l_D)$
- Ainsi, l_T est croissante.
- Supposons l_T croissante.
 - Donc l_G et l_D (qui sont des sous-listes de l_T) sont croissantes. Par hypothèse d'induction, G et D sont des ABR.
 - $\max(l_G) < x < \min(l_D)$ donc $\max(\varphi(G)) < x < \min(\varphi(D))$.

Donc T est un ABR.

On peut conclure l'induction. \square

Remarque :

- le minimum d'un ABR se lit sur le nœud le plus à gauche (ce n'est pas forcément une feuille, il peut avoir un fils droit).
- le maximum se lit sur le nœud le plus à droite.

2.2. Algorithmes élémentaires

Blague de Solène: il est où le d'Algorithm ?

Rebout, qui a pas compris la blague et après qu'on lui a expliqué : bon, elle est pas nulle cette blague, mais...

En OCaml :

```
type 'a arbre =
| V
| N of 'a * arbre * arbre
```

En C :

```
struct noeud {
    int etiquette;
    struct noeud *g;
    struct noeud *d;
};

typedef struct noeud arbre;
```

L'arbre vide est représenté par le npointeur nul `NULL`.

La création d'un nouveau nœud peut se faire avec la fonction:

```
arbre *nouveau_noeud(int x) {
    arbre *noeud = malloc(sizeof(arbre));
    noeud->etiquette = x;
    noeud->g = NULL;
    noeud->d = NULL;
    return noeud;
}
```

Exemple :

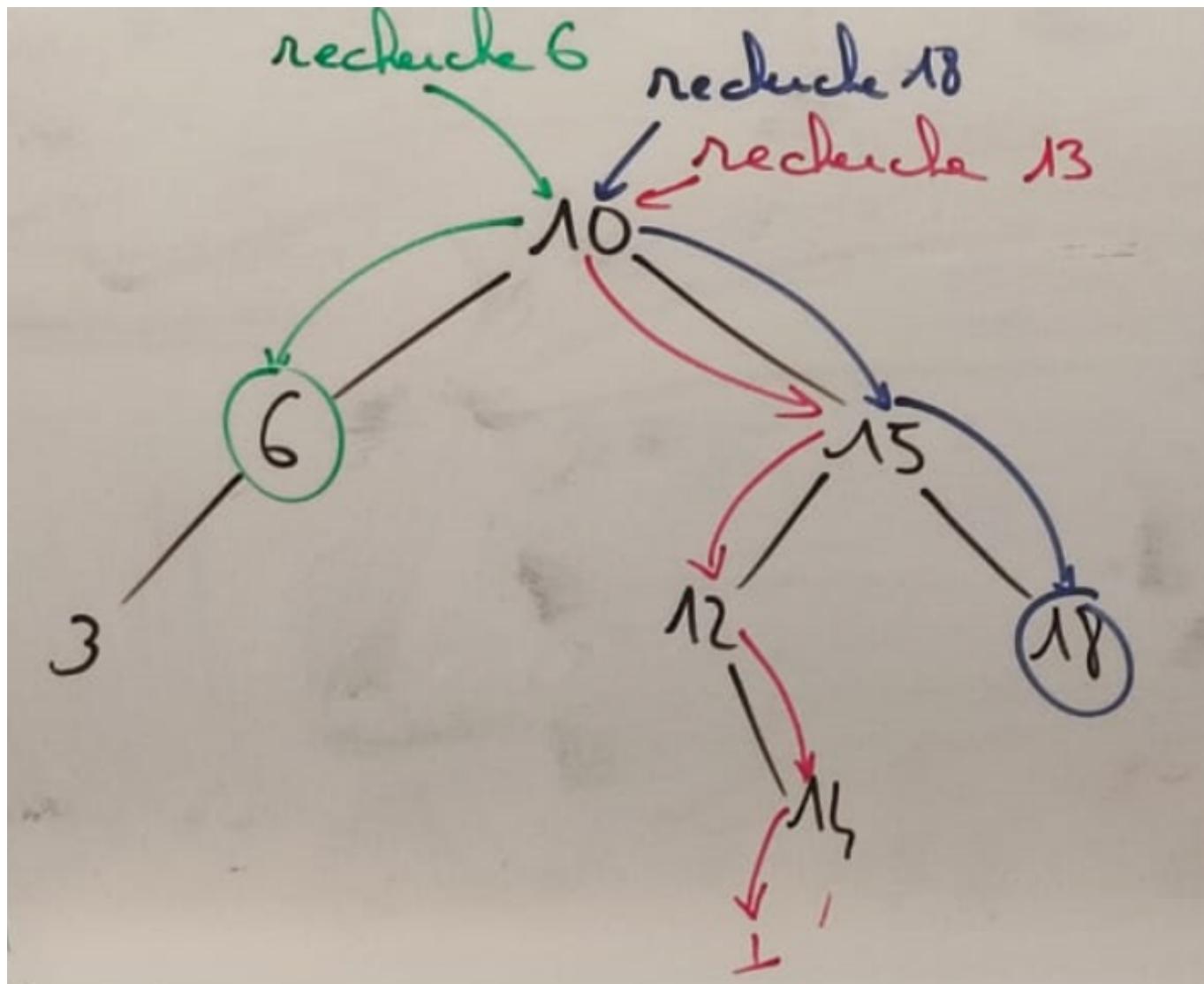
Écrire une fonction `libere_arbre` qui libère la mémoire accapée par un arbre.

```
void libere_arbre(arbre *t) {
    if (t != NULL) { // ou: if (t) { ... }
        libere_arbre(t->g);
        libere_arbre(t->d);
        free(t);
    };
}
```

2.2.a. Recherche dans un ABR

Pour rechercher un élément dans un ABR, on descend le long d'une branche jusqu'à trouver l'élément ou arriver à un sous-arbre vide.

Exemple : On recherche `6`.



Ca rappelle une recherche dichotomique.

Exercice :

1. Écrire une fonction : `apartient 'a abr -> 'a -> bool`
2. Prouver sa correction.

1.

```
let appartient abr x =
  match abr with
  | V -> false
  | N (r, _, _) when r = x -> true
  | N (r, g, d) when r > x -> appartient g x
  | N (r, g, d) when r < x -> appartient d x
```

2.

$$x \in \varphi(T) \iff \text{apartient } t \ x \text{ renvoie true}$$

EXO A LA MAISON : À prouver par induction structurelle

```
// Rappel :  
struct noeud {  
    int etiquette;  
    struct noeud *g;  
    struct noeud *d;  
};  
typedef struct noeud abr;  
//////////  
  
// Version impérative :  
bool appartient(abr *t, int x) {  
    abr *courant = t;  
    while (courant != NULL) {  
        if (t->etiquette == x)  
            return true;  
        else if (t->etiquette > x)  
            courant = t->g;  
        else  
            courant = t->d;  
    }  
    return false  
}  
  
// Version récursive :  
bool apparteint(abr *t, int x) {  
    if (t == NULL) return false;  
    else if (abr->etiquette == n) return true;  
    else if (abr->etiquette > n) return (appartenit(abr->g, x));  
    else return appartenit(abr->d, n);  
}
```

```
let rec maximum abr =  
  match abr with  
  | V -> failwith "vide"  
  | N (x, _, V) -> x  
  | N (_, _, d) -> maximum d
```

2.2.b. Insertion

Les insertions se font en bas de l'arbre (à la place d'une feuille vide) : on descend le long d'une branche de manière à trouver le bon endroit.

Si l'élément est déjà présent, il sera rencontré lors de la descente de la branche et on renvoie alors l'arbre inchangé.

Exercice :

```
let insere : 'a abr -> 'a -> 'a abr;;
```

```

let rec insere abr x =
  match abr with
  | V -> N (x, V, V)
  | N (r, _, _) when x = r -> abr
  | N (r, g, d) when x < r -> N (r, insere g x, d)
  | N (r, g, d) when x > r -> N (r, g, insere d x)

```

```

// Rappel :
struct noeud {
  int etiquette;
  struct noeud *g;
  struct noeud *d;
};

typedef struct noeud abr;
///////////

```



```

abr *insere(abr *t, int x) {
  if (t == NULL)
    return nouveau_noeud(x);
  if (x < t->etiquette)
    t->g = insere(t->g, x);
  else { if (x > t->etiquette)
    t->d = insere(t->d, x);
  };
  return t;
}

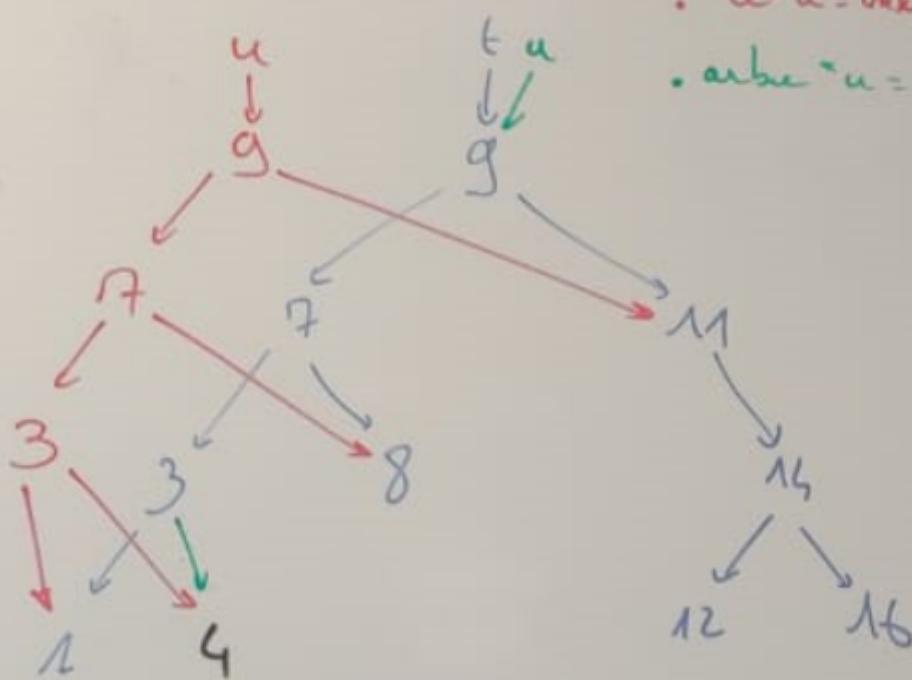
```

Version impérative ?

Version renvoyant `void` ? -> accepte en argument un `abr **t`

 En mémoire :

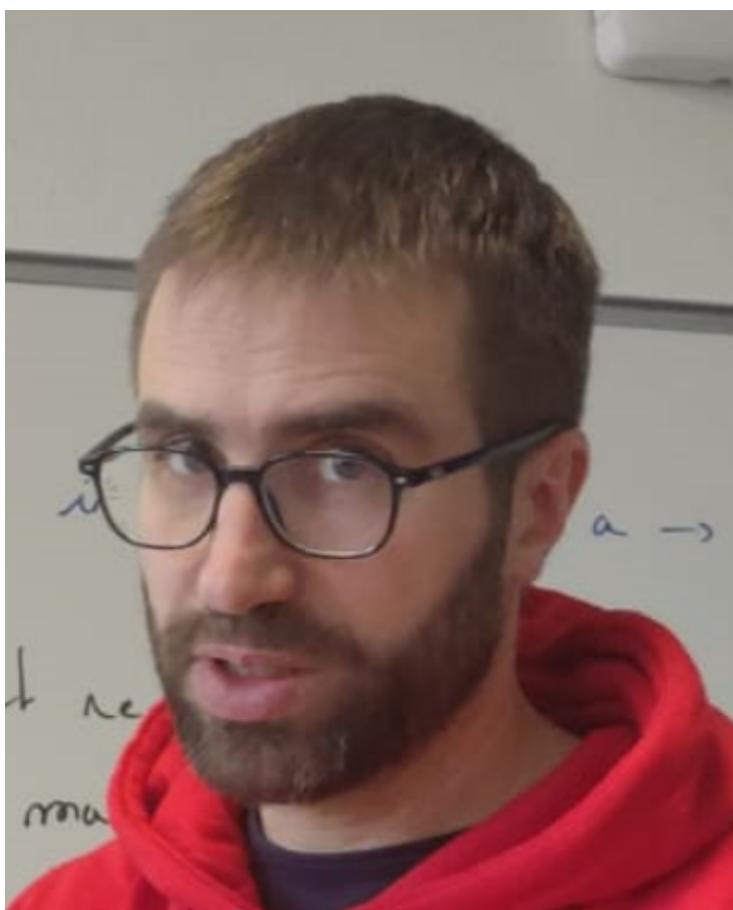
En mémoire



- let $u = \text{inserer } t \text{, } 4$
- arbre = $u = \text{inserer}(t, 4)$

On a un problème d'aliasing : on a des noms différents pour la même chose.

- 🐾 En OCaml, il y a un partage de la mémoire sur la parties non problématique de l'arbre
- 🎲 En C, il y a de l'aliasing : $t = a$.



Chad Bébout qui est sur le point de t'explorer

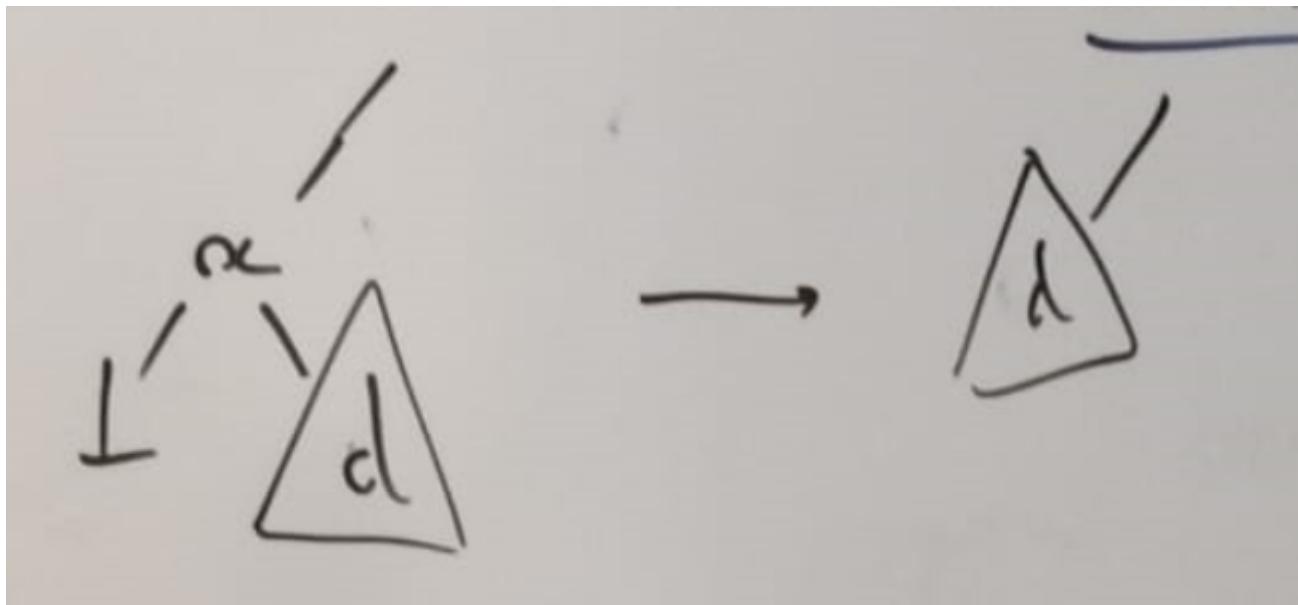
2.2.c. Suppression

Soient un ABR T qui représente l'ensemble $\varphi(T)$ et un élément x .

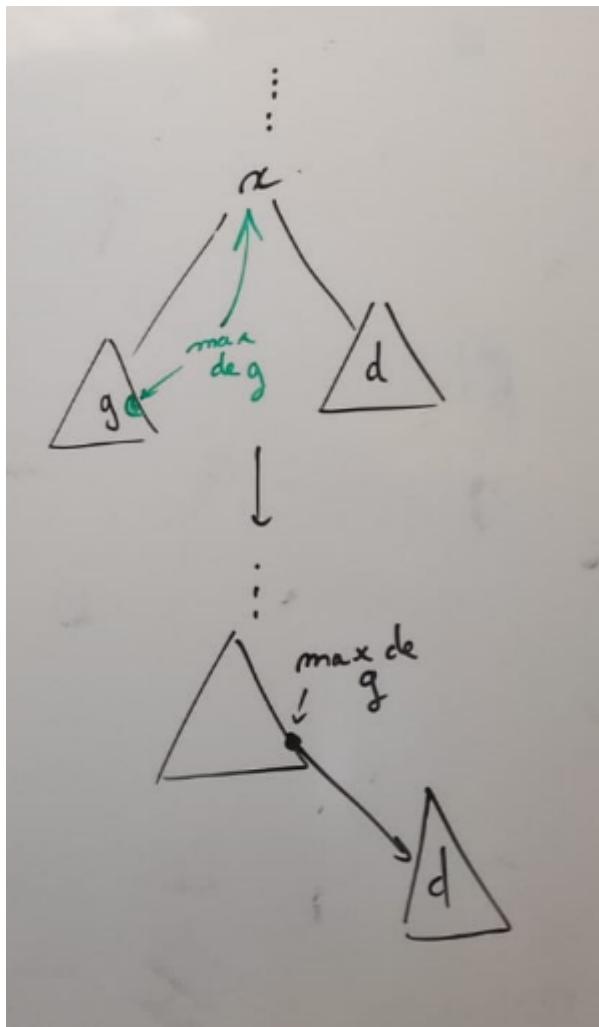
On souhaite renvoyer un ABR représentant l'ensemble $\varphi(T \setminus \{x\})$

Cas simple :

- Si $x \notin \varphi(T)$, il n'y a rien à faire.
- Si x est sur une feuille, on remplace cette feuille par \perp .
- Si x est sur un nœud interne avec 1 des 2 enfants vide, on remplace x par ce fils.



```
let rec supprime abr x =
  match abr with
  | V -> V
  | N (r, V, d) when x=r -> d
  | N (r, g, V) when x=r -> g
  | N (r, g, d) when x<r -> N (r, supprime g x, d)
  | N (v, g, d) when x>y -> N (r, g, supprime d x)
  | N (r, g, d) -> ???
```



Stratégie pour le dernier cas:

- On considère le sous-arbre gauche g de l'élément à supprimer
- On récupère le maximum m de g et on calcule $g' = \text{supprime } g \ m$ (*le maximum ne peut pas avoir de fils droit donc on est dans un cas simple*)
- On revoie l'arbre (m, g', d)

Exercice :

1. Ecrire une fonction :

```
supprime_max: 'a abr -> 'a abr * 'a
```

```
(* Rappel *)
type 'a arbre =
| V
| N of 'a * arbre * arbre

let rec supprime_max abr =
  match abr with
  | N (r, g, d) ->
    let nv_d, el = supprime_max d in
    N (r, g, nv_d), el
  | N (r, g, V) ->
```

```
g, r  
;;
```

2. Compléter la ligne 8 de la fonction précédente.

Complexité des opérations :

Proposition:

Soit t un ABR de hauteur h .

Une opération de recherche, d'insertion ou de suppression effectue au plus $h + 1$ comparaisons entre clés.

Démonstration :

On effectue au plus une comparaison par niveau de l'arbre...

Théorème :

Soit t un ABR de hauteur h .

En supposant que la comparaison de deux clés se faire en temps constant, les opérations sur un ABR peuvent s'effectuer en temps $O(h)$.

Rappel: $\lfloor \log_2 n \rfloor \leq h \leq n + 1$

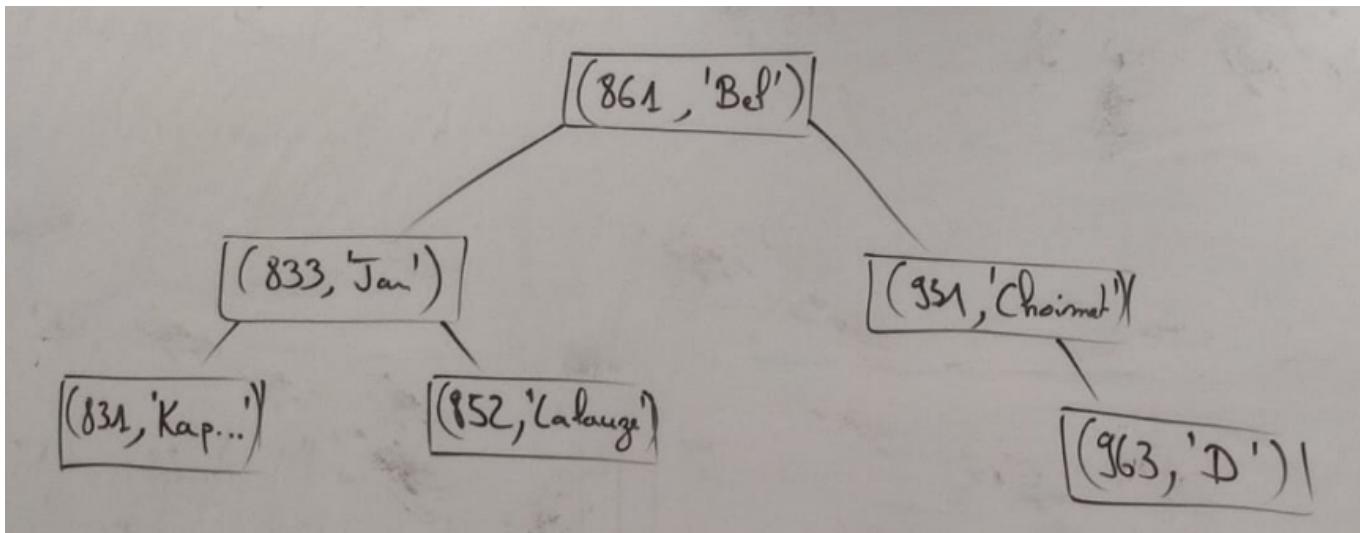
Remarque :

- La comparaison entre entier/flottants/caractères (machine) se fait en temps constant.
- La comparaison entre chaînes de caractères ne se fait pas en temps constant (*les tries sont plus adaptées pour représenter un ensemble de mots*).

2.3. Réalisation d'un dictionnaire à partir d'un ABR

Les ABR permettent de réaliser de manière fonctionnelle la structure de dictionnaire *à condition que les clés soient choisies dans un ensemble totalement ordonné*.

Pour se faire, on stocke dans les nœuds les couples `(clé, valeur)` et toutes les comparaisons se font uniquement sur les clés.



Exercice :

```
type ('k, 'v) dict =
| Vide
| Noeud of 'k * 'v * ('k, 'v) dict * ('k, 'v) dict
```

Écrire les fonctions get et set correspondantes.

```
get: 'k -> ('k, 'v) dict -> 'v option
set: 'k -> 'v -> ('k, 'v) dict -> 'v option
```

```
let rec get key dict =
  match dict with
  | Vide ->
  | Noeud (k, v, g, d) -> (*TODO*)
```

```
let rec supprime_max t =
  match t with
  | Vide -> failwith "vide"
  | Noeud (x, g, Vide) -> x, g
  | Noeud (x, g, d) ->
    let m, d' = supprime_max d in
    m, Noeud (x, g, d')
```

TODO: compléter:

```
let rec supprime x t =
... (*Un truc "hyper long" qu'on a déjà marqué askip*)
| Noeud (y, g, d) (*avec x=y*)
-> let m, g' = supprime_max g in
  Noeud (m, g', d)
```

III. Tables de hachage

→ [Voir poly sur les tables de hachage](#)

Les tables de hachage fournissent une réalisation *fondamentellement impérative* des structures abstraites de dictionnaire et d'ensemble. Pour des raisons partiellement mathématiques et partiellement liées aux spécificités des machines actuelles, elles sont souvent plus rapides que les variantes d'arbres binaires de recherche, mais un peu moins flexibles.

3.1. Un cas simple

Supposons que l'on souhaite stocker un ensemble d'associations $(\text{clé}, \text{valeur})$, et supposons de plus que l'on sache que :

- il y aura environ 5 000 clés;
- toutes les clés seront des entiers entre 0 et 9 999.

Dans ce cas, la solution la plus efficace (et de loin!) est d'utiliser un tableau de longueur 10 000, et de stocker l'association (k, v) , si elle existe, dans la case d'indice k du tableau. En OCaml, on pourrait par exemple prendre le type suivant :

```
type 'v table = 'v option array
```

Les fonctions `get`, `set` et `remove` sont alors extrêmement simples, et s'exécutent, de manière immédiate, en temps constant :

```
let get table k = table.(k)
let set table k v = table.(k) <- Some v
let rem table k = table.(k) <- None
```

3.2. Principe

Le principe d'une table de hachage est de conserver une solution proche de celle décrite ci-dessus tout en s'affranchissant des contraintes sur les clés, qui sont désormais librement choisies dans un ensemble K . On va donc fixer une valeur N pour la taille de la table (la longueur du tableau sous-jacent) et ramener les clés dans l'ensemble $\llbracket 0, N \rrbracket$. Typiquement, il y aura deux étapes pour cela :

- on se dote d'une fonction $h : K \rightarrow N$, dite fonction de hachage;
- pour obtenir un indice dans le tableau à partir d'une clé k , on calcule $h_N(k) \stackrel{\text{d\'ef.}}{=} h(k) \bmod N$.

 **Exemple.** Nous verrons plus loin les propriétés que l'on recherche pour une « bonne » fonction de hachage. Cependant, nous pouvons dès à présent donner quelques exemples de fonctions de hachage possibles (sans préjuger de leur qualité) :

- si $K = \mathbb{N}$ (ou $K = \mathbb{Z}$), on peut tout simplement prendre l'identité pour h , et l'on a alors $h_N(k) = k \bmod N$;
- si K est l'ensemble des chaînes de caractères, on peut voir chaque octet de la chaîne comme un entier (entre 0 et 255) et définir $h(k)$ comme la somme de ces entiers;
- si K est l'ensemble des nombres flottants (disons sur 64 bits), on peut juste interpréter la représentation binaire de k comme un entier et se ramener au premier cas.

L'idée est alors de créer un tableau un tableau de taille N et de stocker l'association (k, v) dans la case numéro $h_N(k)$ de ce tableau. Si i est l'indice d'une case du tableau, on aura alors trois cas :

- aucune association ne vérifie $h_N(k) = i$: la case du tableau est « vide »;
- une seule association vérifie $h_N(k) = i$: la case du tableau contient cette association;
- on a une collision : plusieurs associations vérifient $h_N(k) = i$. On a alors un problème : les différents types de tables de hachage correspondent aux différentes manières de le régler.

L'idée fondamentale est que la recherche d'une association devient beaucoup plus rapide : si l'on cherche une clé k , il est inutile de parcourir tout le tableau. On calcule $h_N(k)$ et l'on regarde dans la case correspondante du tableau : si la clé n'y apparaît pas, c'est qu'elle n'apparaît nulle part dans le tableau.

 **Remarque.** Il est important de bien comprendre que l'endroit où l'on stocke une association (k, v) ne dépend que de la clé k et pas du tout de la valeur v .

3.3. Résolution par chaînage

Une idée simple et couramment utilisée pour gérer les collisions est celle du *chaînage*. Au lieu d'utiliser un tableau de couples (k, v) , on utilise un tableau de listes de couples (k, v) . La case i du tableau contiendra la liste (éventuellement vide) des associations (k, v) vérifiant $h_N(k) = i$: en anglais, on appelle cette liste un *bucket* (*seau*, littéralement, mais on parle plutôt d'*alvéole* en français).

On pourrait donc imaginer le type suivant en OCaml :

```
type ('k, 'v) dict =
  { hash : 'k -> int
  ; table : ('k * 'v) list array
  }
```

- Le champ `hash` contient la fonction de hachage h .

- Le champ `table` contient le tableau à proprement parler.

À l'intérieur d'une alvéole, les opérations de recherche, ajout, remplacement se font comme dans la réalisation naïve de l'exercice vu auparavant.

Globalement, on a donc deux étapes pour chaque opération :

- quand on veut ajouter un couple (k, v) , on calcule $h_N(k)$ et l'on ajoute (k, v) à la liste se trouvant dans la case $h_N(k)$ du tableau (ou l'on remplace la valeur associée à k , s'il y en avait déjà une);
- quand on cherche la valeur associée à une clé k , on calcule $h_N(k)$ et on la cherche dans la liste contenue dans la case $h_N(k)$ du tableau (et uniquement dans cette liste).

Ville	Paris	Lyon	Toulouse	Rennes	Nancy	Lille	Nice
h_5	2	0	2	4	1	0	2

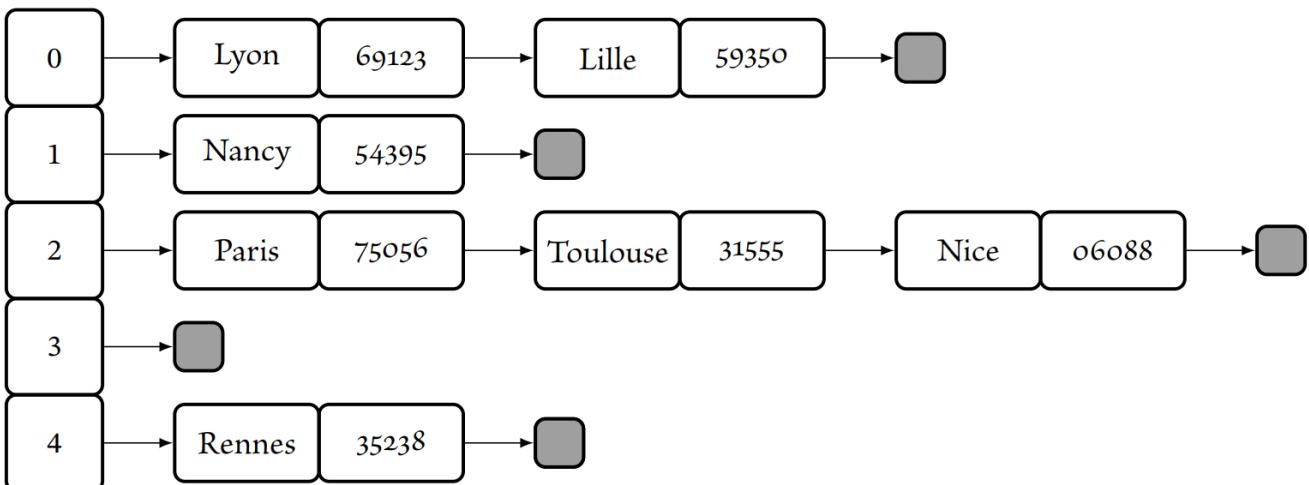


Figure 9.1 – Principe d'une table de hachage (chaînée), on associe à chaque ville son code INSEE

Cookie Exercices sur les tables de hachage du poly :

- Écrire trois fonctions `get_list`, `set_list` et `remove_list` agissant sur une liste d'associations. On pourra supposer que la liste contient au plus une association pour une clé donnée (et on maintiendra cet invariant).

```

let rec get_list li key =
  match li with
  | [] -> None
  | (k, v)::xs ->
    if key = k then Some v
    else get_list xs key
  
```

```

let rec set_list li key val =
  match li with
  
```

```

| [] -> [(key, val)]
| (k, v) :: xs ->
  if key = k then (key, val) :: xs
  else (k, v) :: set_list xs key val

let rec remove_list li key =
  match li with
  | [] -> []
  | (k, v) :: xs ->
    if key = k then xs
    else (k, v) :: remove_list xs key

```

2. Écrire les fonctions get, set et remove agissant sur un dictionnaire.

```

let get di key =
  let i = di.hash key in (* h(a) *)
  let i_mod = i mod (Array.length di.table) in (* h_N(a) *)
  get_list (di.table.(i_mod)) key

let set di key val =
  let i = (di.hash key) mod (Array.length di.table) in
  di.table.(i) <- set_list di.table.(i) key val

let remove di key =
  (*TODO*)

```

TODO

En suivant cette méthode, la complexité de la recherche, de l'ajout ou de la suppression d'une clé k ne dépend plus du nombre total n d'associations du dictionnaire, mais seulement du nombre de clés en collision avec k . Plus précisément, on a :

- une évaluation de $h_N(k)$, dont le coût peut dépendre de k (si notre clé est par exemple une chaîne de caractères arbitrairement longue), mais pas de n ;
- un nombre de comparaisons entre clés égal à la longueur de la chaîne présente dans l'alvéole.

Idéalement, on souhaiterait répartir équitablement les valeurs de h_N , pour qu'il y ait (environ) le même nombre d'associations dans chaque alvéole.

3.4. Critères pour une bonne fonction de hachage

Une fonction de hachage « idéale » ne provoquerait jamais de collisions : c'est bien évidemment impossible, puisque l'ensemble K des clés possibles est en général beaucoup plus grand que $\llbracket 0, N \rrbracket$ (voire infini). En étant un peu plus raisonnable, on souhaiterait que h_N se comporte comme un tirage aléatoire uniforme dans l'ensemble $\llbracket 0, N \rrbracket$. Il n'est pas très difficile de construire des générateurs de nombre pseudo-aléatoires approchant raisonnablement bien

cette propriété, mais il y a une difficulté supplémentaire : il est indispensable que deux appels $h_N(k)$ avec le même k renvoient la même valeur! Si l'on tire au hasard une empreinte à chaque fois, ce ne sera bien sûr pas le cas.

Quand on analysera le coût des opérations sur les tables de hachage, on supposera toujours que la fonction de hachage répartit les clés uniformément et indépendamment dans l'ensemble $[0, N]$.

👉 **Remarque.** Autrement dit, on supposera :

- pour tout $0 \leq i < N$, pour une clé k choisie aléatoirement, $\mathbb{P}(h_N(k) = i) = \frac{1}{N}$;
- si k et k' sont choisies aléatoirement, alors $\mathbb{P}(h_N(k) = h_N(k')) = \frac{1}{N}$.

Ces propriétés ne sont pas réellement vérifiées par les fonctions de hachage utilisées en pratique (elles n'ont même pas forcément de sens), et de plus les clés n'ont aucune raison d'être choisies au hasard (ce qui invalidera la plus grande partie de notre analyse).

Cependant :

- il est possible de rendre cela rigoureux (en tirant la fonction de hachage au hasard dans une famille de fonctions bien choisies);
- les performances pratiques sont conformes à l'analyse simplifiée, pour peu que la fonction de hachage soit raisonnablement bien choisie.

Remarques.

- On peut montrer que si l'on répartit aléatoirement n boules dans n urnes, le nombre de boules dans l'urne la plus « chargée » sera en moyenne $\frac{\ln n}{\ln(\ln n)}$ (environ). Cela signifie que dans une table de hachage chaînée de taille n avec un facteur de charge de 1, le *pire cas* pour la recherche d'une clé sera en $(\frac{\ln n}{\ln(\ln n)})$ (en supposant que la hachage se fasse uniformément).
- En OCaml, on utilisera la fonction `Hashtbl.hash : 'a -> int`, qui à un objet quelconque associe un entier positif ou nul de manière « pseudo-uniforme ».

3.5. Résolution par adressage ouvert

On verra en TP cette autre manière de résoudre les problèmes de collision.

IV. Arbres auto équilibrés

4.1. Arbres 2-3

4.1.a. Présentation

Définition : Un **arbre 2-3** est:

- soit l'arbre vide \perp ,
- soit de la forme $N(g, x, d)$ où x est une étiquette, et g, d sont des arbres 2-3 vérifiant la condition:

$$\max g < x < \min d$$

- soit de la forme $N(g, x, m, y, d)$ où x, y sont des étiquettes et g, m, d des arbres 2-3 vérifiant la condition:

$$\max g < x < \min m \leq \max m < y < \min d$$

De plus, on impose une condition *d'équilibrage parfait* : tous les nœuds vides sont à la même profondeur.

 **Remarque** : À nouveau, on a pris des inégalités strictes ici (sauf pour $\min m$ et $\max m$, l'arbre m pouvant avoir zéro ou un élément), ce qui impose l'unicité des clés et correspond typiquement à une structure d'ensemble.

On a également gardé la convention $\min \perp = +\infty$ et $\max \perp = -\infty$.

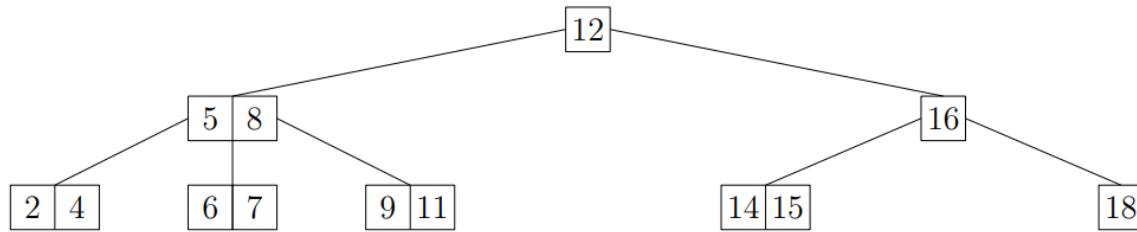
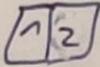
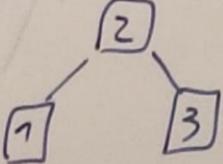


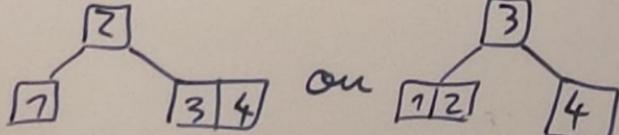
FIGURE 9.1 – Un exemple d'arbre 2-3; les nœuds vides ne sont pas représentés.

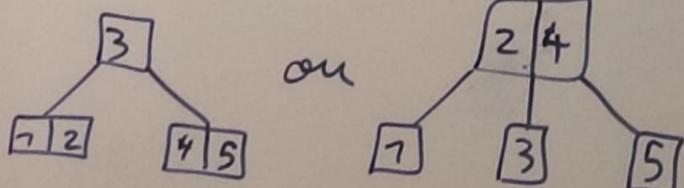
 **Exercice** : Dessiner tous les arbres 2-3 ayant comme ensemble d'étiquettes :

1. {1, 2};
2. {1, 2, 3};
3. {1, 2, 3, 4};
4. {1, 2, 3, 4, 5}.

$\{1, 2\}$: 

$\{1, 2, 3\}$: 

$\{1, 2, 3, 4\}$: 

$\{1, 2, 3, 4, 5\}$: 

💡 Proposition : En notant $h(t)$ la hauteur de t et $|t|$ son nombre d'étiquettes, on a :

$$h(t) = \mathcal{O}(\log |t|)$$

🛠 Démonstration : Tous les nœuds vides sont à distance $h(t) + 1$ de la racine (en prenant $h(V) = -1$), donc il y a au moins 2^k nœuds à profondeur k . Il y a donc au moins 2^h étiquettes à profondeur h , d'où $2^h \leq n$ et donc $h \leq \log_2 n$.

□

👉 Remarque : Plus précisément, on a $\lfloor \log_3(n+1) \rfloor - 1 \leq h \leq \lfloor \log_2(n+1) \rfloor - 1$.

4.1.b Recherche dans un arbre 2-3

Imaginons que l'on représente en OCaml un arbre 2-3 à l'aide du type suivant :

```
type 'a arbre23 =
| V
| B of 'a arbre23 * 'a * 'a arbre23
| T of 'a arbre23 * 'a * 'a arbre23 * 'a * 'a arbre23
```

On peut alors très facilement adapter l'algorithme de recherche dans un ABR :

```
let rec appartient x t =
  match t with
  | V -> false
  | B (g, y, d) when x = y -> true
  | _ -> appartient x g || appartient x d
```

```

| B (g, y, d) -> if x < y then appartient x g else appartient x d
| T (g, y, m, z, d) when x = y || x = z -> true
| T (g, y, m, z, d) ->
  if x < y
    then appartient x g
  else if x < z
    then appartient x m
  else appartient x d

```

Proposition : La recherche d'une clé dans un arbre 2-3 contenant n clés se fait en temps $\mathcal{O}(\log n)$ (sous l'hypothèse que les comparaisons entre clés se font en temps constant).

 **Démonstration :** On descend le long d'une branche de l'arbre en faisant des opérations en temps constant (comparaison de clés, déconstruction) sur chaque nœud traversé. La complexité est donc en $\mathcal{O}(h)$, c'est-à-dire en $\mathcal{O}(\log n)$ d'après la proposition précédente.

□

4.2.c. Insertion dans un arbre 2-3

L'insertion d'un nouvel élément dans un arbre 2-3 débute comme l'insertion dans un arbre binaire : on descend dans l'arbre jusqu'à trouver l'endroit approprié. Par exemple, pour insérer 17 dans

l'arbre de la figure 9.1, on commence par effectuer la descente figurée en gras sur le schéma suivant :

[...]

$A_1 \quad A_2 \quad A_3 \quad A_4$

$\text{infix}(A_1), a, \text{infix}(A_2), b, \text{infix}(A_2), c$
 $\text{infix}(A_3), d, \text{infix}(A_3), e, \text{infix}(A_3)$
 $e, \text{infix}(A_6)$

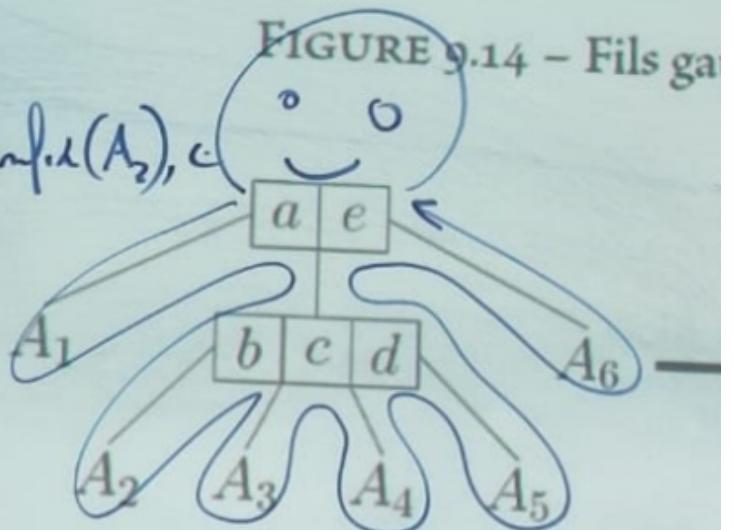
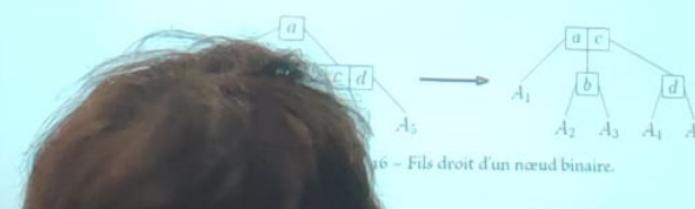
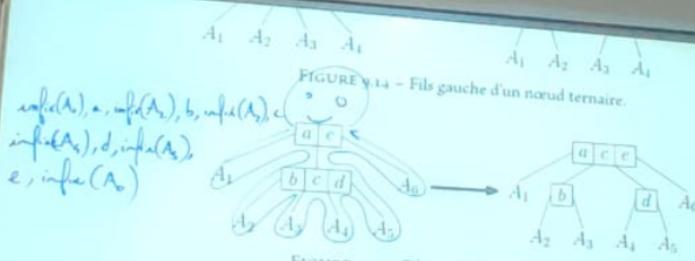


FIGURE 9.15 – Fils central d'un nœud ternaire.



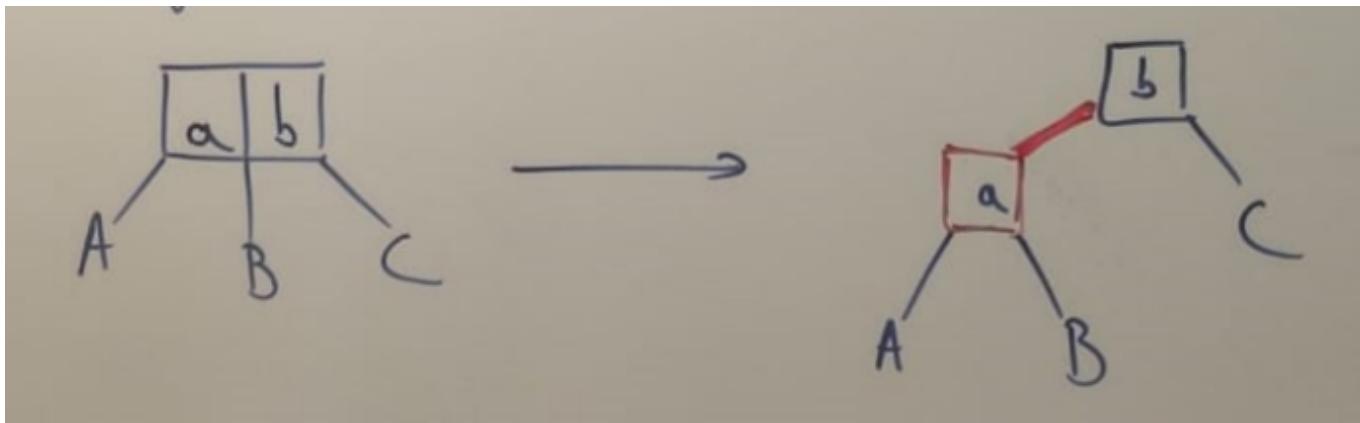
M. Bebout tout fier de son magnifique poulpe

4.2. Arbre rouge-noir

4.2.a Définition

Les arbres rouge-noir (ou arbres bicolores) sont essentiellement une *version binarisée des arbres 2-3*.

L'idée est de remplacer tous les nœuds ternaires par 2 nœuds binaires, en marquant en rouge l'arête nouvellement créée (ainsi que le nœud auquel l'arête aboutit).

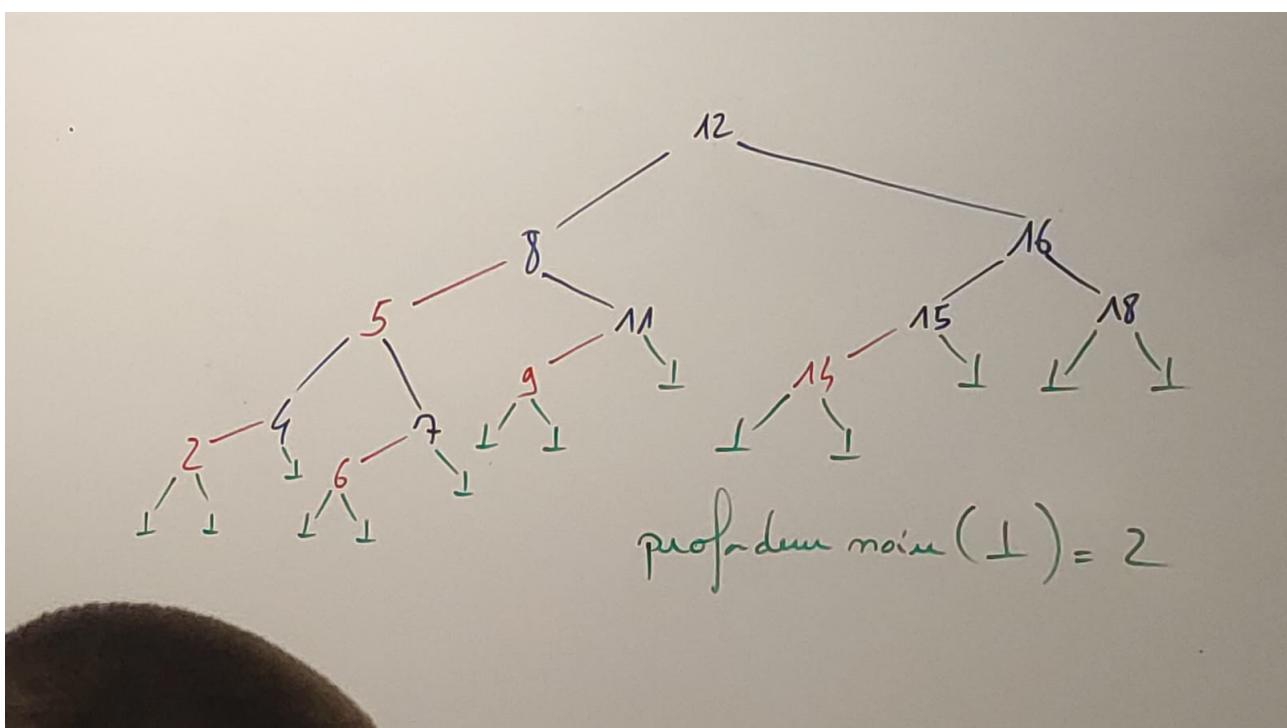
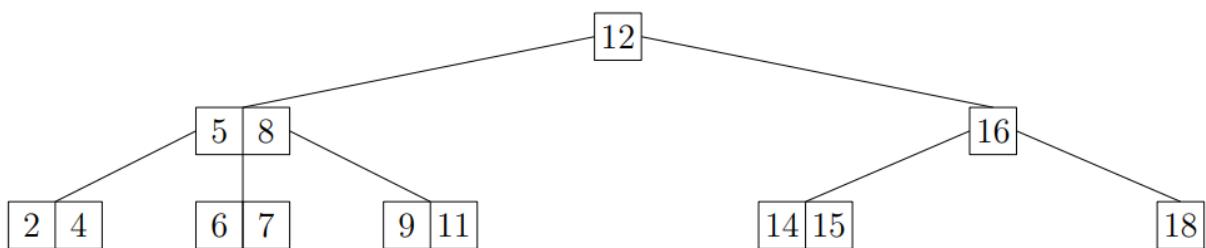


Propriété d'ordre pour un arbre 2-3 → propriété d'ordre pour un ABR.

La propriété d'équilibrage totale est un peu modifiée. On définit la notion de profondeur noire d'un nœud : c'est le nombre d'arêtes noires sur le chemin de la racine à ce nœud.
→ Tous les nœuds vides doivent partager la même profondeur noire.

Enfin : il est impossible d'avoir 2 arêtes rouges consécutives : cela correspondrait à un nœud quaternaire.

Exemple : On binarise l'arbre suivant :



Définition : Un arbre rouge-noir (gauche) est un ABR dans lequel chaque nœud possède une couleur (rouge ou noir, les sous-arbres vides étant considérés noirs) et qui vérifient les

conditions suivantes :

- aucun nœud rouge ne possède de fils rouge (*sinon, ça serait un arbre quaternaire*).
- tous les chemins reliant la racine à un nœud vide contiennent le même nombre de nœuds noirs.
- le fils droit d'un nœud est nécessairement noir.
- la racine est noire.

💡 **Proposition :** Si t est un arbre rouge-noir, alors $h(t) = \mathcal{O}(\log |t|)$

🛠️ **Démonstration :** Si t est un arbre rouge noir de taille n et de hauteur h , on peut le transformer en arbre 2-3, de taille n et de hauteur $h' \geq \frac{h}{2}$ (les liens rouges sont supprimés, et au plus la moitié des liens sont rouges...). Or, $h' = \mathcal{O}(\log_2 n)$, donc $h \leq 2h' = \mathcal{O}(\log_2 n)$

4.2.b. Recherche

La recherche dans un arbre rouge-noir est identique à celle dans un ABR.

💡 **Proposition :** La recherche se fait en temps $\mathcal{O}(\log_2 |t|)$ (si la comparaison entre clés se fait en $\mathcal{O}(1)$).

4.2.c. Insertion

On traduit l'insertion pour les arbres 2-3 vers les arbres rouge-noir.

- On commence par insérer tout un bas, en transformant un nœud binaire en un nœud ternaire ou un nœud ternaire en un nœud quaternaire (dans l'arbre 2-3).

Cela se traduit par la création d'un nœud rouge.

- La création de ce nœud a de fortes chances de violer les invariantes d'un arbre rouge-noir.
- Ces violations sont déplacées vers le haut de l'arbre, jusqu'à disparaître...

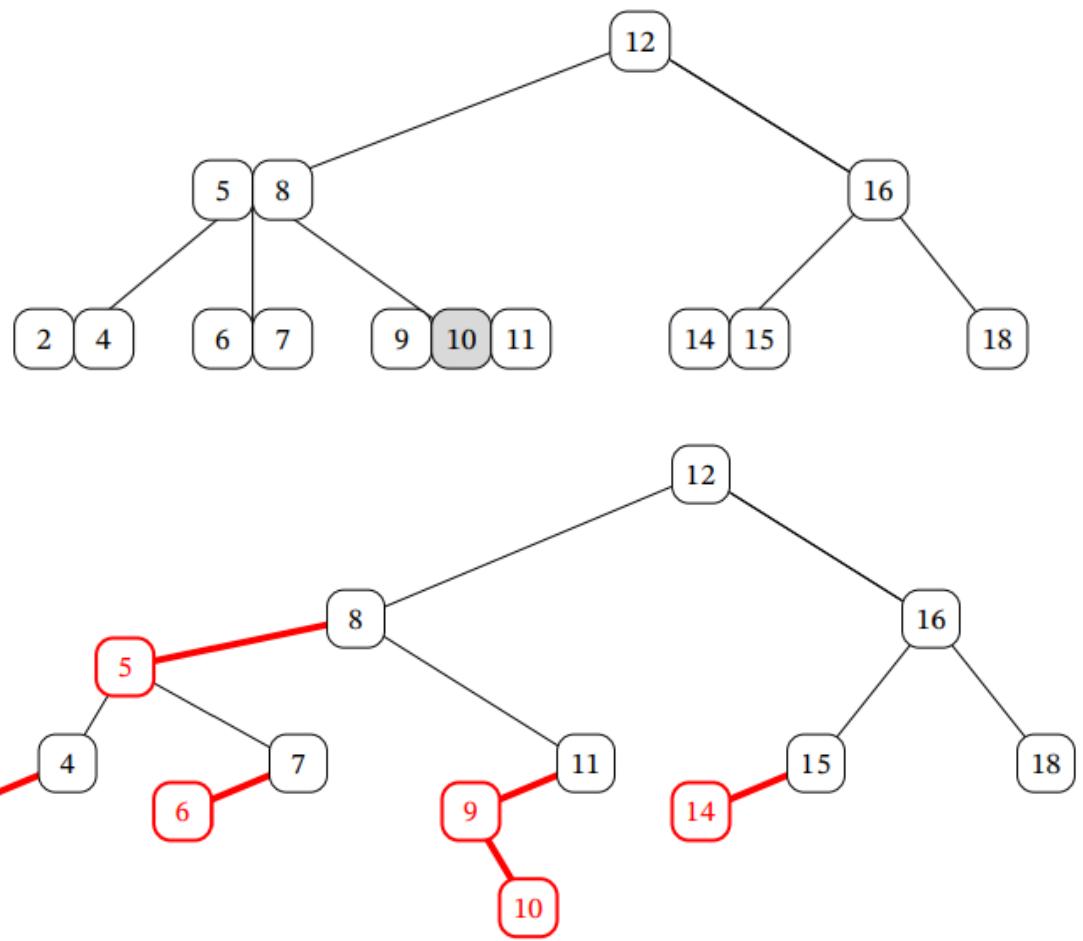


FIGURE 9.1 - Insertion comme une feuille - création d'un nœud quaternaire

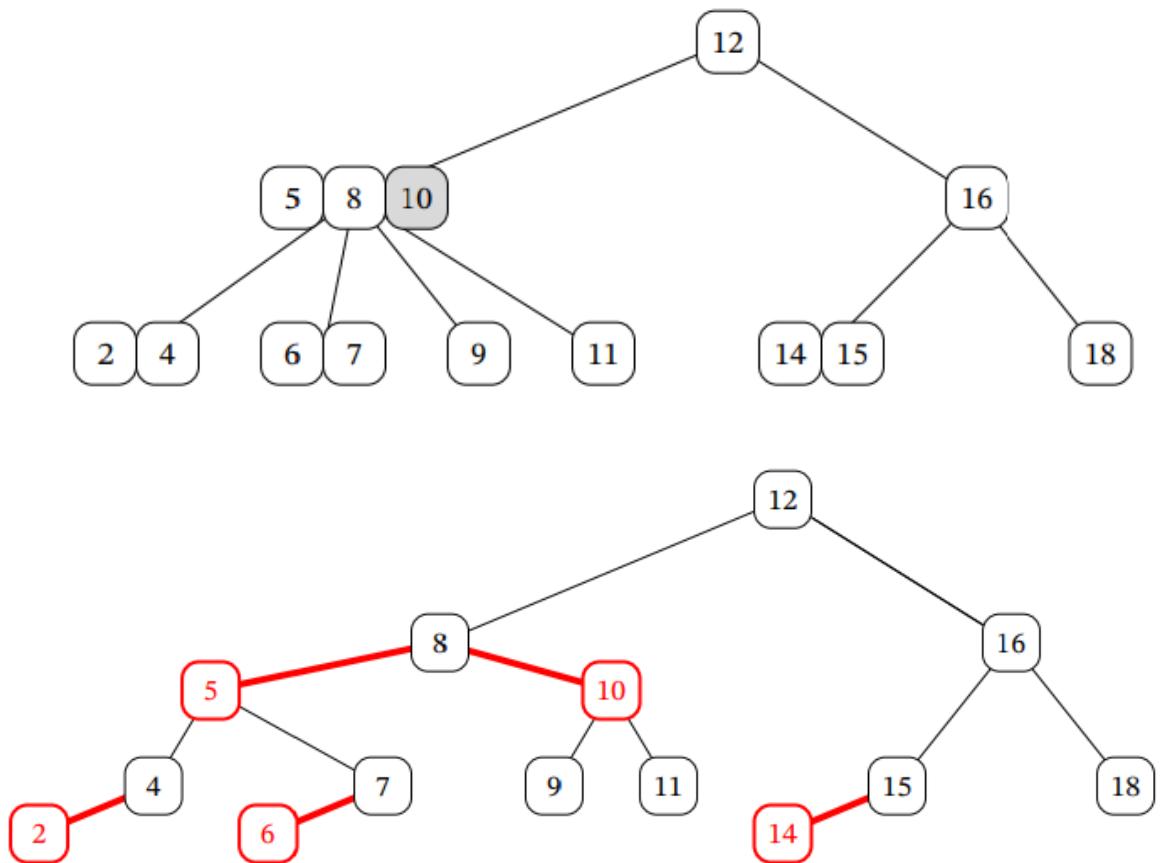


FIGURE 9.2 - Le problème remonte et persiste

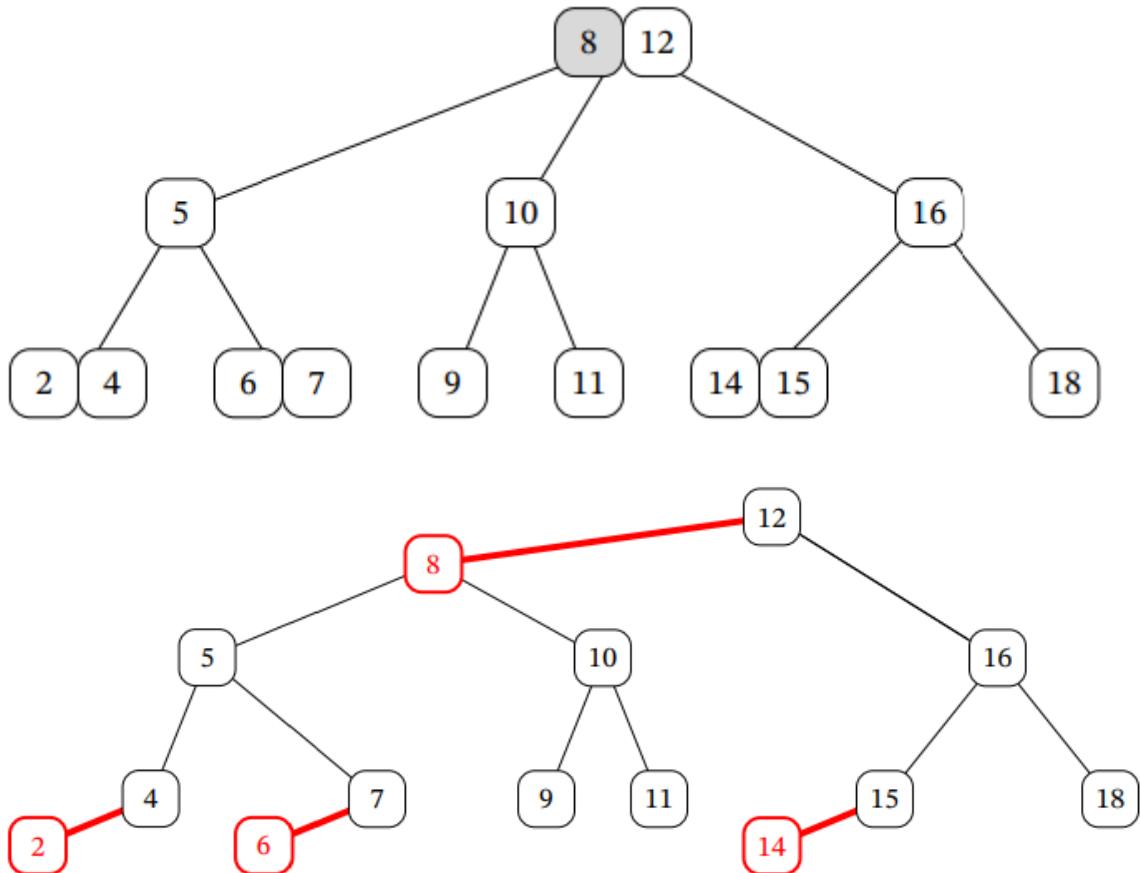
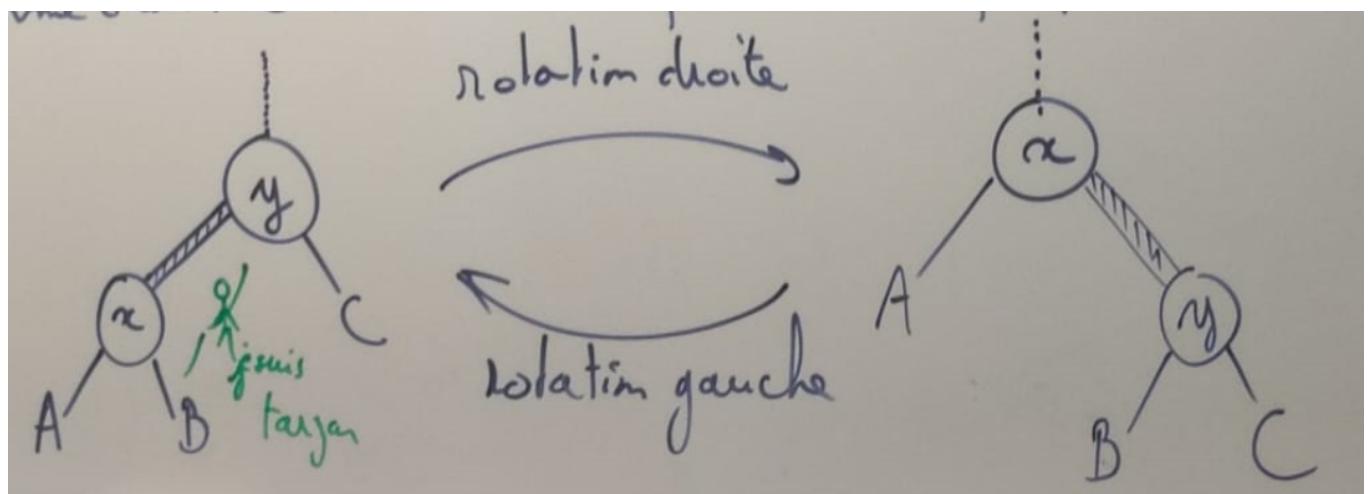


FIGURE 9.3 - Le problème remonte et disparaît

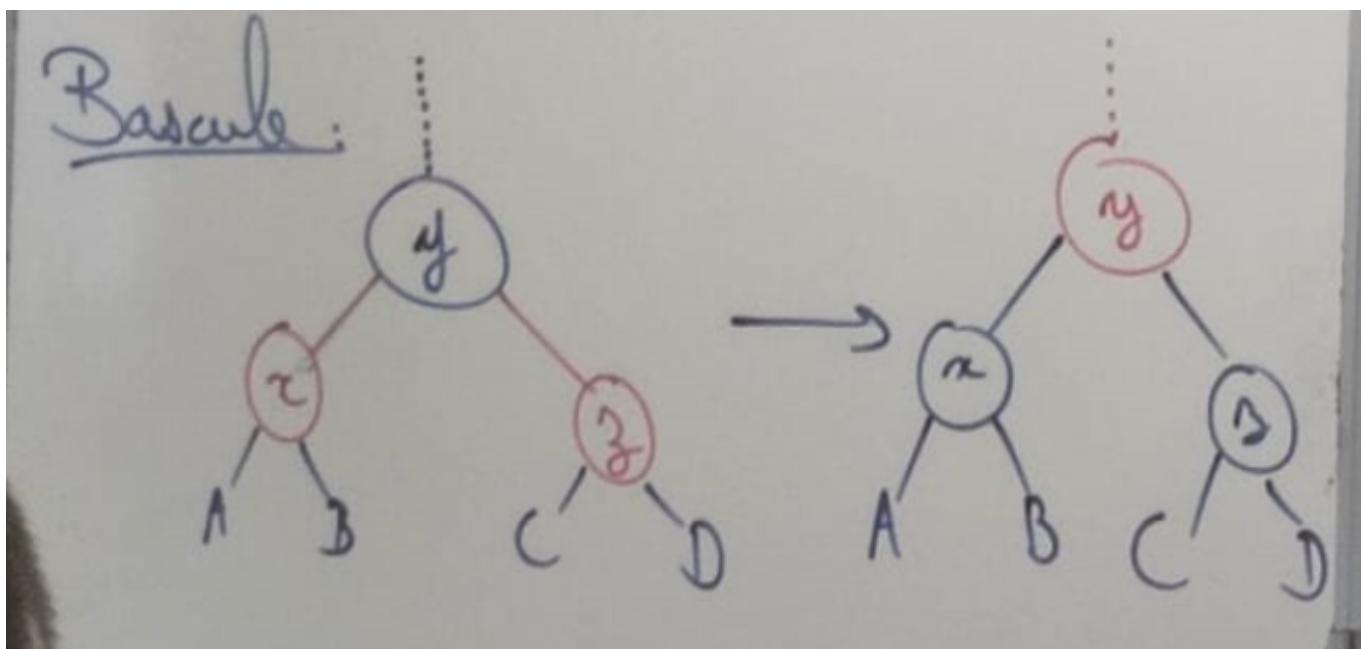
Rotation d'un ABR

La rotation autour d'une arête est l'opération fondamentale permettant de modifier la forme d'un ABR tout en respectant la propriété d'ordre.

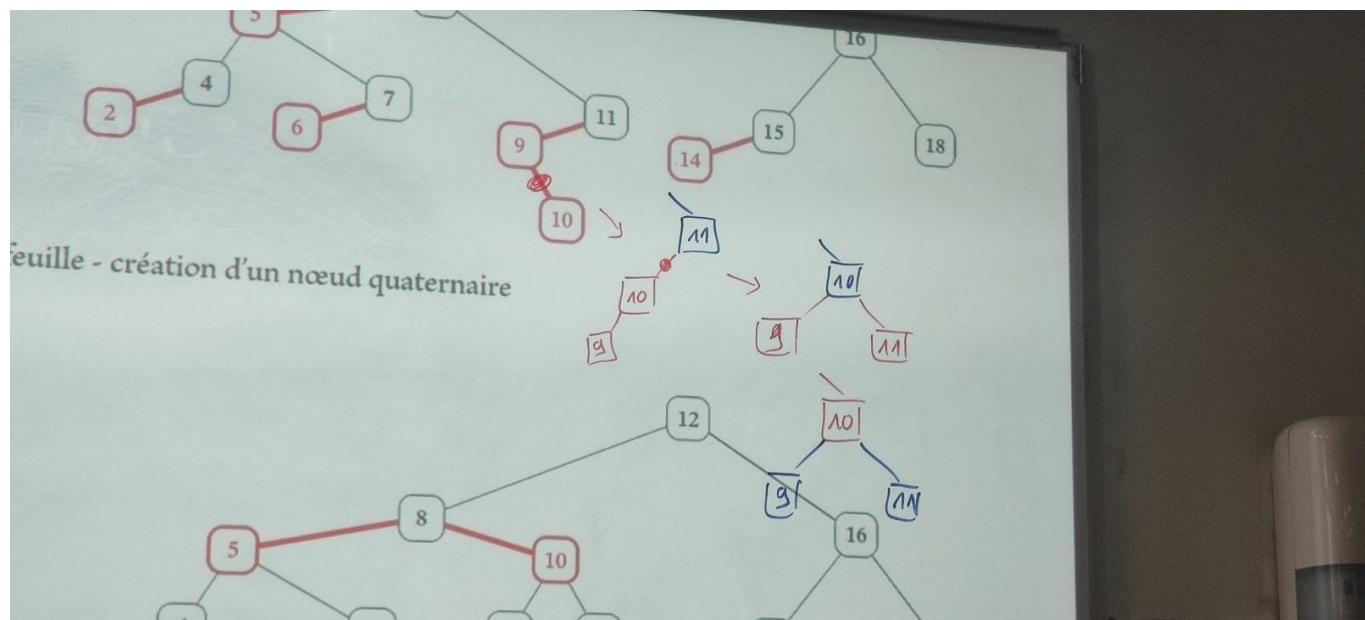


Dans un arbre rouge-noir, la rotation doit également gérer la couleur des nœuds : l'idée est que l'arête tourne et c'est elle qui détermine la couleur du nœud.

Dans l'exemple, x et y échangent leur couleur (l'opération respecte les profondeurs noires).



Exemple :



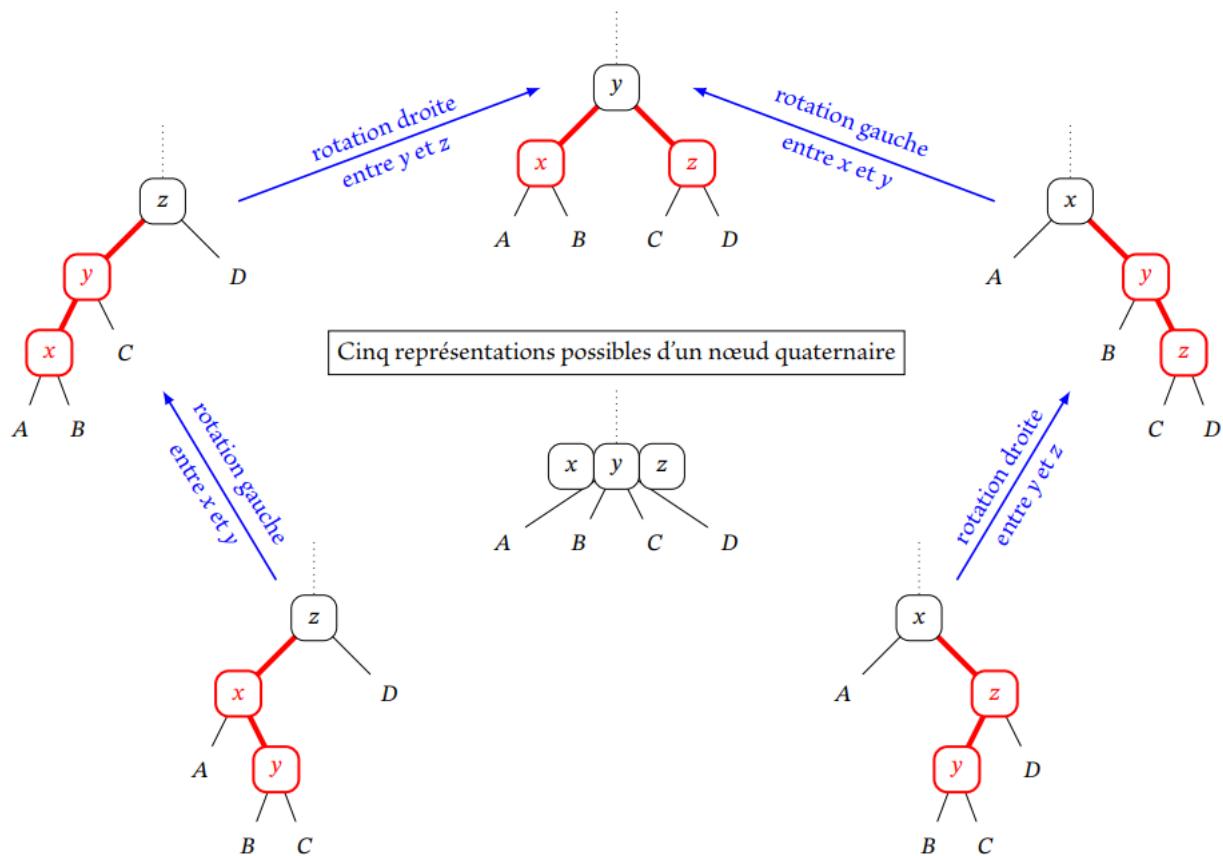


FIGURE 9.4

FIGURE 9.4

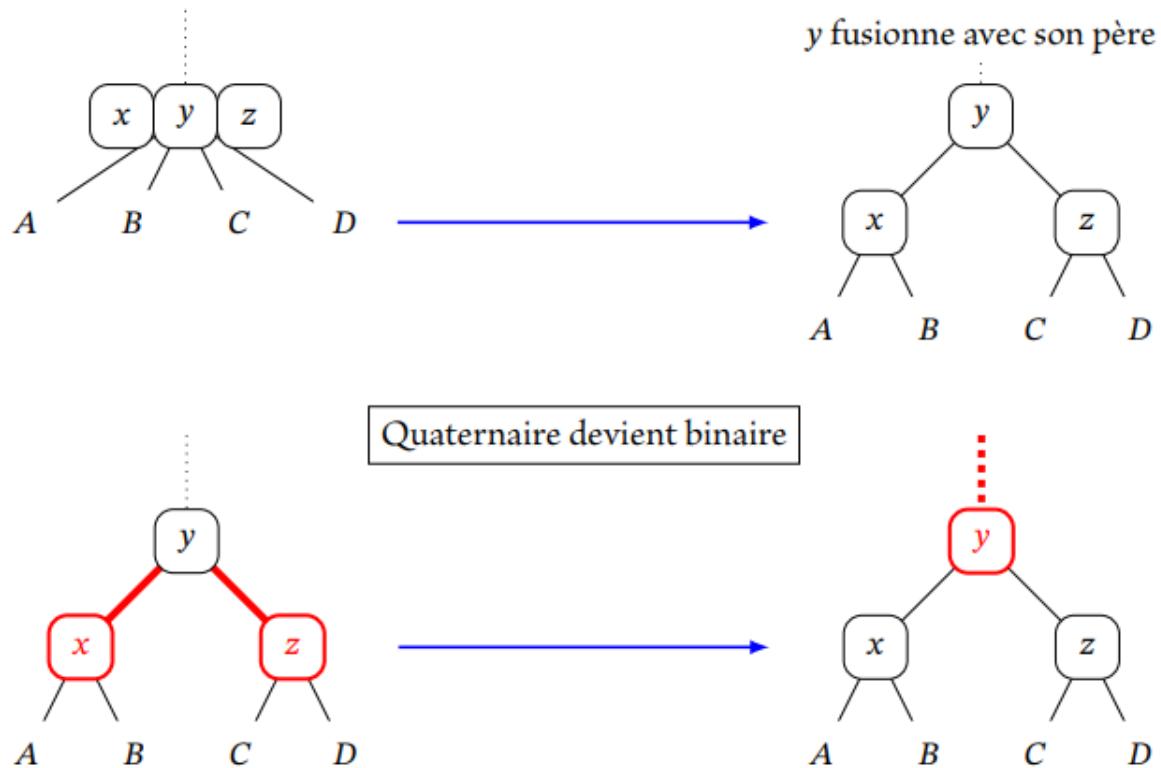
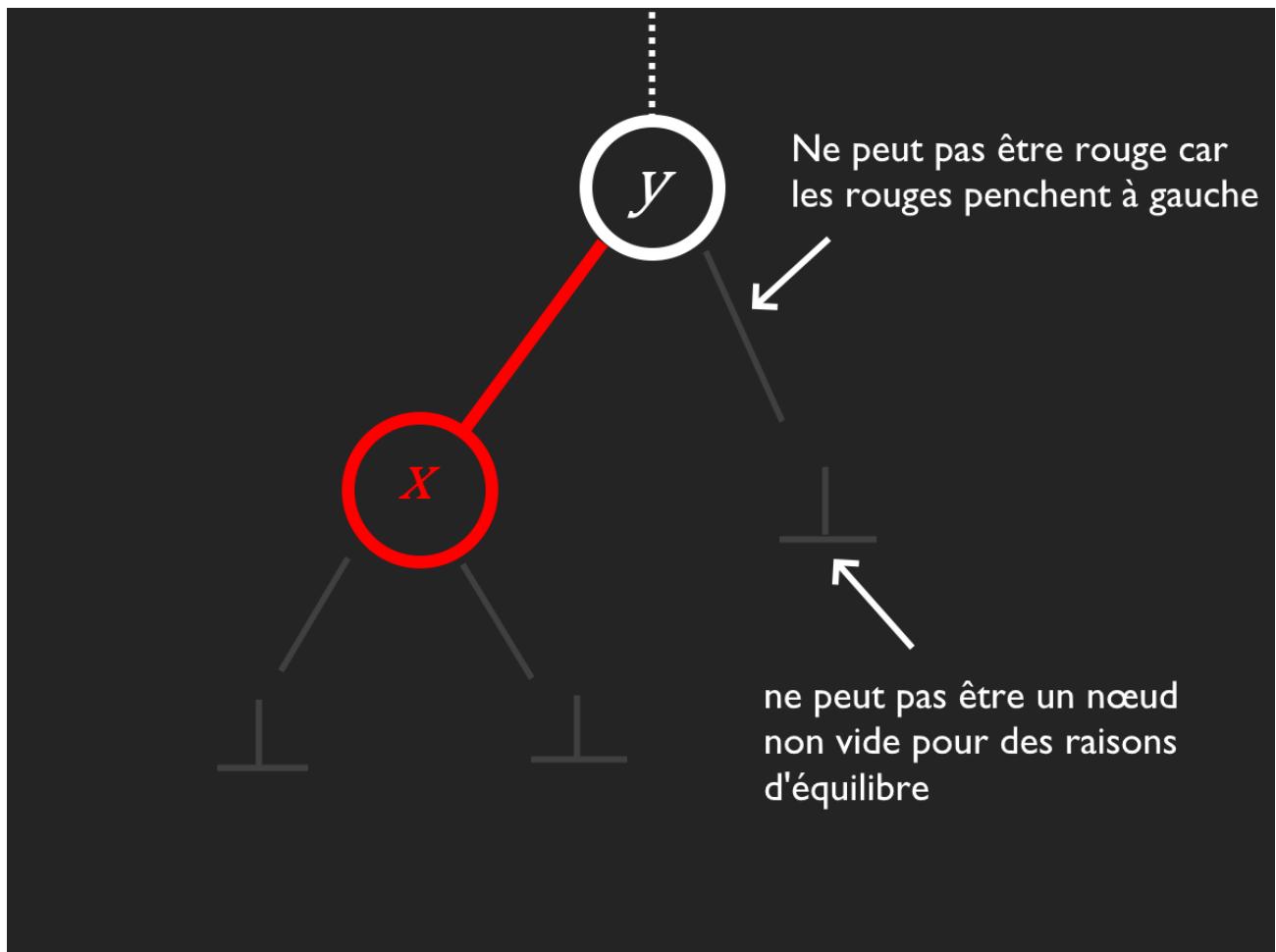


FIGURE 9.5

Différents cas pour l'insertion d'un élément.

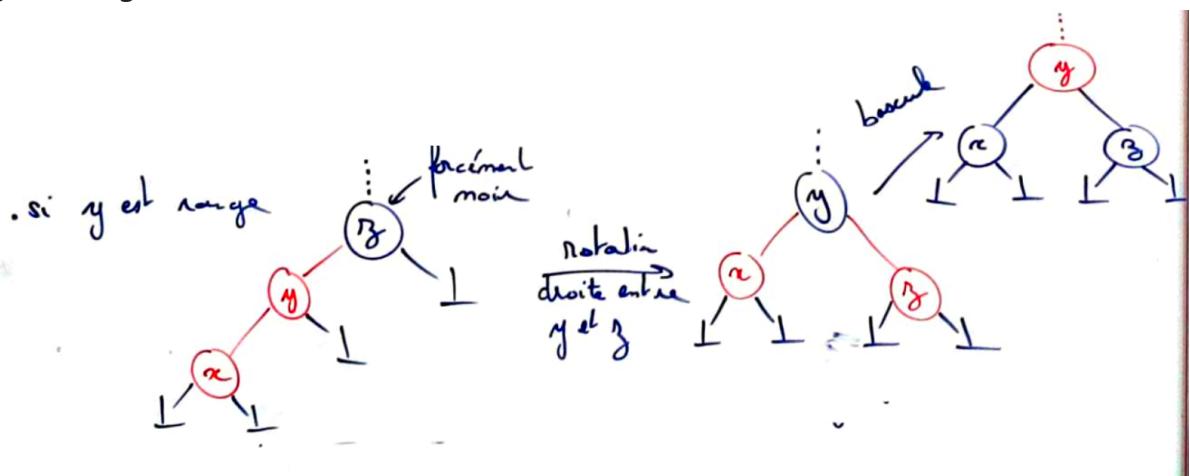
- Si $x < y$:

- si y est noir



→ Rien à faire dans ce cas là.

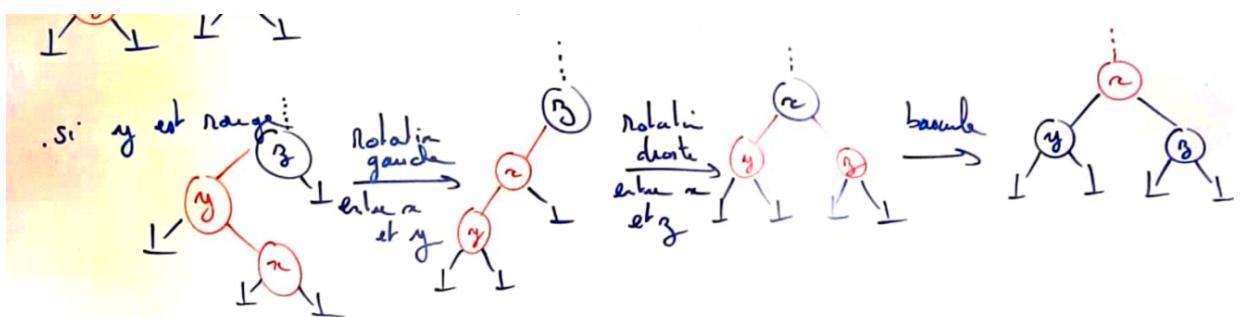
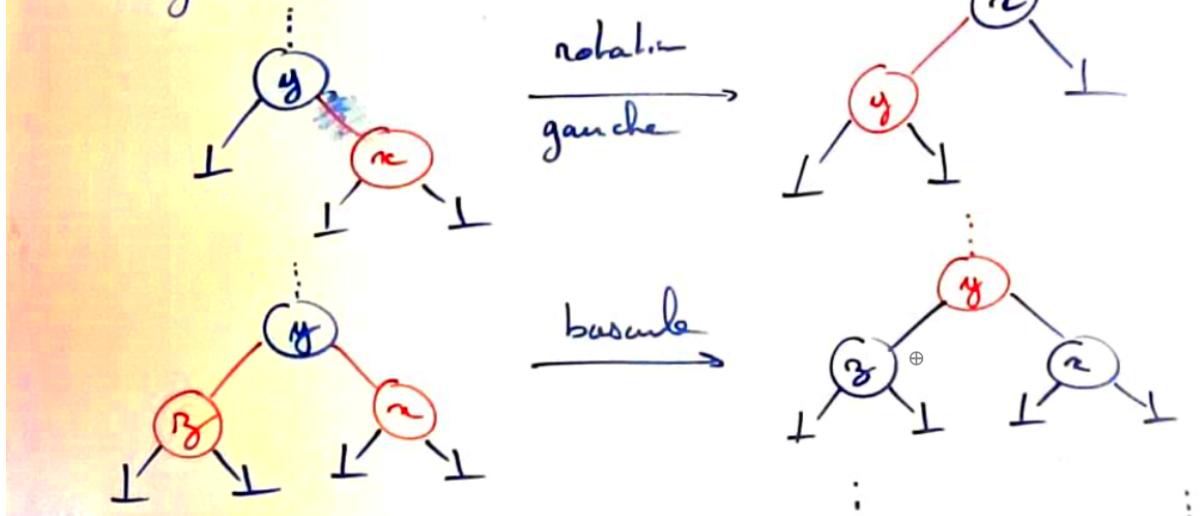
- si y est rouge



↑ Correspond à l'insertion dans un nœud ternaire d'un arbre 2-3

Si $x > y$:

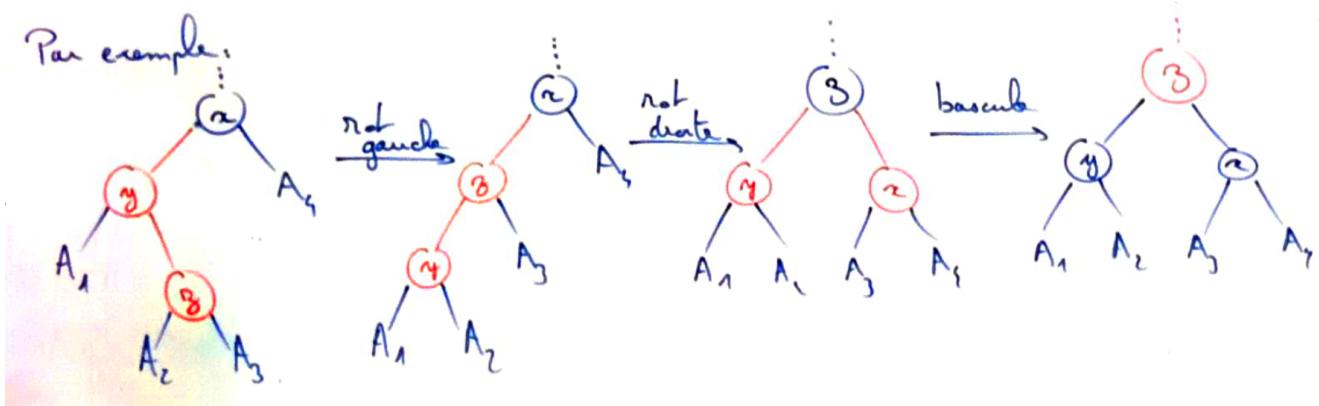
si y est noir



Après l'insertion de l'élément comme une feuille, on remonte le chemin vers la racine, pour éliminer les possibles violations des propriétés de la structure.

Les cas sont exactement identiques : il suffit de remplacer les \perp par des sous arbres (de racine noire).

Exemple :



On remarque qu'on effectue essentiellement les mêmes opérations dans le même ordre quitte à inhiber quand on ne peut pas les appliquer :

Pour insérer x dans l'arbre a , on le rejouté récursivement à gauche ou à droite, on obtient ainsi a' et on effectue alors :

$(\text{bascule}^? \circ \text{rotation droite}^? \circ \text{rotation gauche}^?)(a')$

Ici, **op?** signifie qu'on effectue l'opération uniquement si elle s'applique :

- pour la **rotation gauche** : le fils gauche et le fils droit du fils gauche doivent être rouges.
- pour la **rotation droite** : le fils gauche et le fils gauche du fils gauche doivent être rouges.
- pour la **bascule** : les deux fils doivent être rouges.

À la fin de ce processus, il reste éventuellement une étape : noircir *la racine* si elle est rouge.

🚧 Implémentation en TP.

📚 Exemple :

