Fonctionnalité et Modularité

Cours 4

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-fm

Plan

- arbres
- représentation on Ocaml)
- types inductifs
- filtrage et test d’exhaustivité
- fonction inductives
- arbres binaires de recherche
- arbres de recherche équilibrés

télécharger Ocaml en `http://www.ocaml.org`

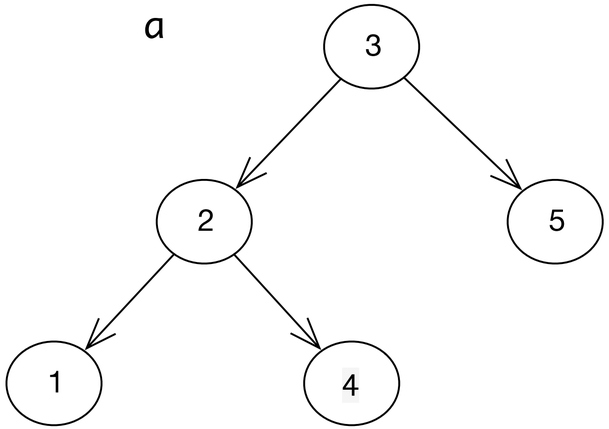
Arbres

- on définit le type `arbre`

```
type 'a arbre =
  | Feuille of 'a
  | Noeud of 'a * 'a arbre * 'a arbre ;
```

- et on construit des arbres

```
let a = Noeud (3, Noeud (2, Feuille (1), Feuille (4)), Feuille (5))
```



- calculs de la hauteur et de la taille de l'arbre

```
let rec hauteur = function
  | Feuille _ -> 0
  | Noeud (_, a1, a2) -> 1 + max (hauteur a1) (hauteur a2) ;;

let rec taille = function
  | Feuille _ -> 1
  | Noeud (_, a1, a2) -> 1 + (taille a1) + (taille a2) ;;
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Arbres

- impression d'un arbre

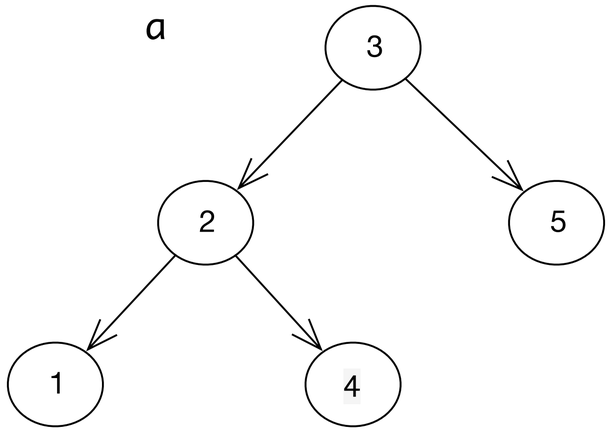
```
let rec string_of_arbre = function
  | Feuille x -> Printf.sprintf "Feuille %d" x
  | Noeud (x, a, b) ->
      Printf.sprintf "Noeud (%d, %s, %s)"
        x (string_of_arbre a) (string_of_arbre b) ;;

let print_arbre a = Printf.printf "%s\n" (string_of_arbre a) ;;
```

- on construit et imprime des arbres

```
let a = Noeud (3, Noeud (2, Feuille (1), Feuille (4)), Feuille (5)) ;;

print_arbre a ;;
Noeud (3, Noeud (2, Feuille (1), Feuille (4)), Feuille (5))
```



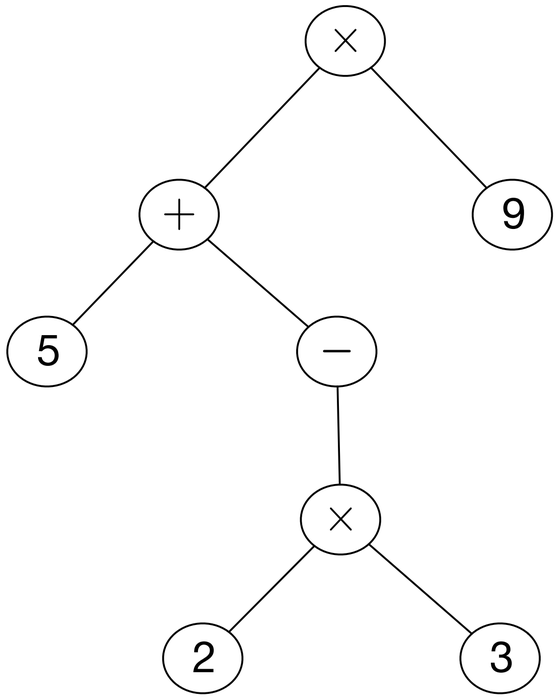
Arbres

- On peut distinguer les noeuds binaires et les noeuds unaires

```
type 'a arbre =
  | Feuille of 'a
  | Noeud_Un of 'a * 'a arbre
  | Noeud_Bi of 'a * 'a arbre * 'a arbre ;;
```

- et on construit l'arbre par:

```
let a = Noeud_Bi ('*',
  Noeud_Bi ('+', Feuille ('5'),
                 Noeud_Un ('-',
                         Noeud_Bi ('*', Feuille ('2'), Feuille ('3')))),
  Feuille ('9')) ;;
```



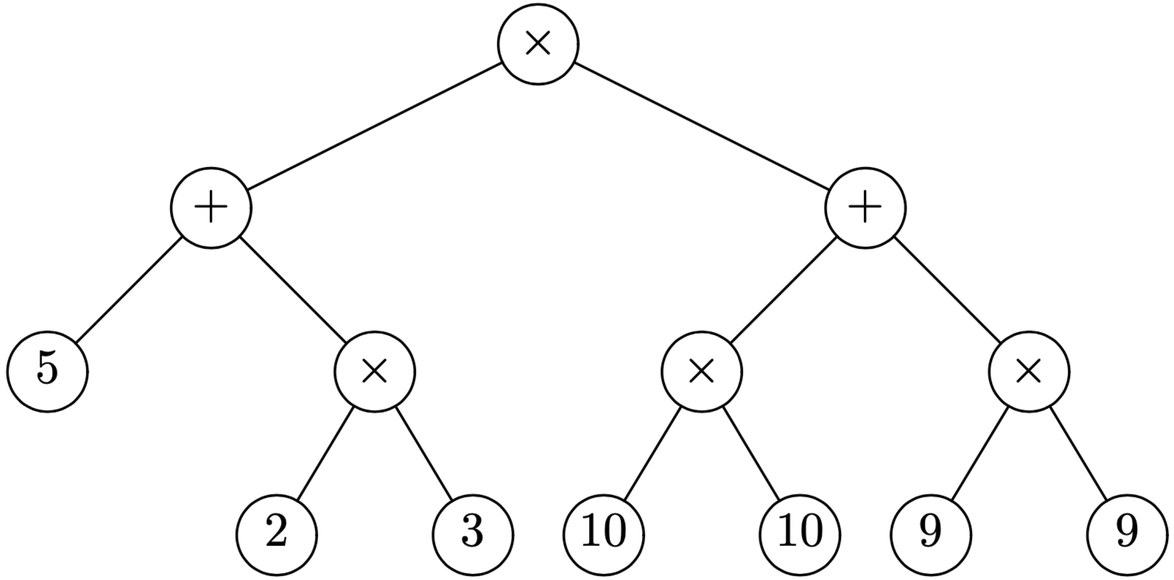
Arbres

- On parcourt ou calcule sur les arbres avec des fonctions récursives

induction structurelle

```
let rec hauteur = function
  | Feuille _ -> 0
  | Noeud (_, a1, a2) -> 1 + max (hauteur a1) (hauteur a2) ;;

let rec taille = function
  | Feuille _ -> 1
  | Noeud (_, a1, a2) -> 1 + (taille a1) + (taille a2) ;;
```



- et on calcule les hauteur et taille

```
let a = Noeud ("*", Noeud ("+", Feuille ("5"), Noeud ("*", Feuille ("2"),
  Feuille ("3"))), Noeud ("+", Noeud ("*", Feuille ("10"), Feuille ("10")), Noeud ("*",
  Feuille ("9"), Feuille ("9")))) ;;
hauteur (a)
3

taille (a)
13
```

Arbres binaires de recherche

- recherche en table organisée en **arbre binaire**

- chaque paire (clé, valeur) est stockée dans les noeuds et feuilles

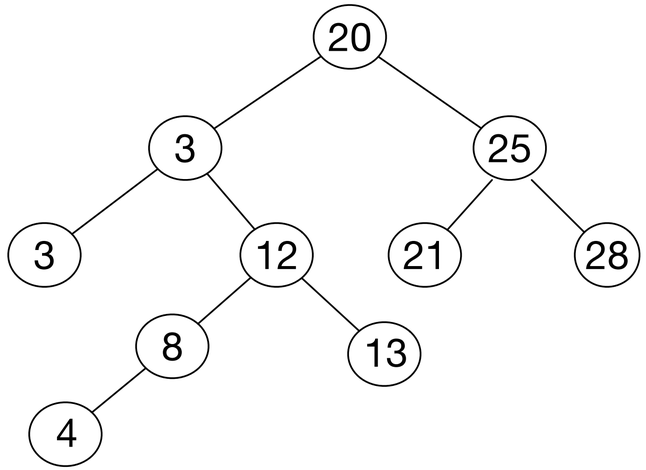
  [**on simplifie ici en ne considérant que les clés**]

- les clés sont stockées dans **l'ordre préfixe**:

  la clé d'un noeud est plus grande que les clés de son fils gauche

  la clé d'un noeud est plus petite que les clés de son fils droit

  [ici, on met les clés égales vers la gauche]




```
type 'a arbre =
  | Feuille
  | Noeud of 'a * 'a arbre * 'a arbre ;;

let a =
  let feuille x = Noeud (x, Feuille, Feuille) in

  Noeud (20, Noeud (3, (feuille 3),
                    Noeud (12, Noeud (8, (feuille 4), Feuille),
                           feuille 13)),
              Noeud (25, (feuille 21), (feuille 28))) ;;
```

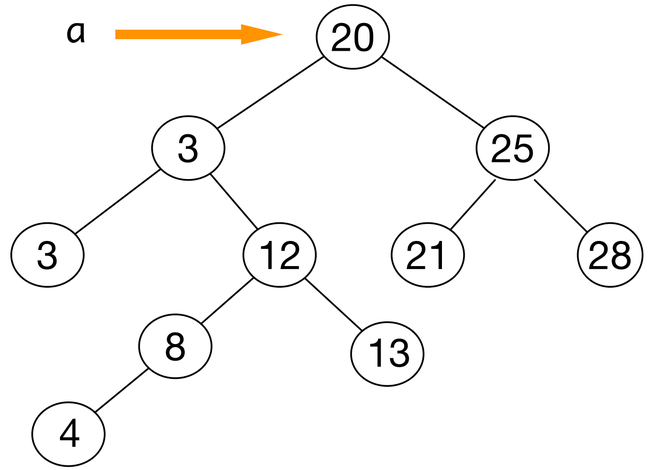
Arbres binaires de recherche

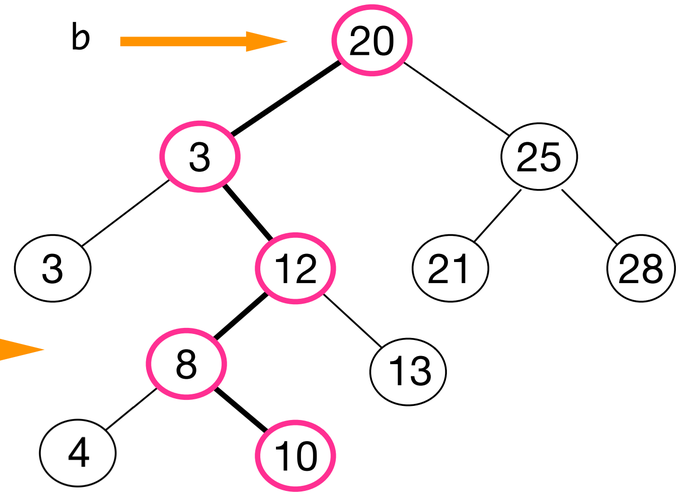
- ajouter une clé (style: **programmation fonctionnelle**)

```
let rec rechercher x a = match a with
  | Feuille -> false
  | Noeud (y, g, d) -> x = y ||
        if x < y then rechercher x g else rechercher x d ;;


let rec ajouter x = function
  | Feuille -> Noeud (x, Feuille, Feuille)
  | Noeud (y, g, d) -> if x <= y then
      Noeud (y, ajouter x g, d) else Noeud (y, g, ajouter x d) ;;


let b = ajouter (10, a) ;;
```





on ne modifie pas l'arbre `a`, les noeuds **rouges** sont nouveaux

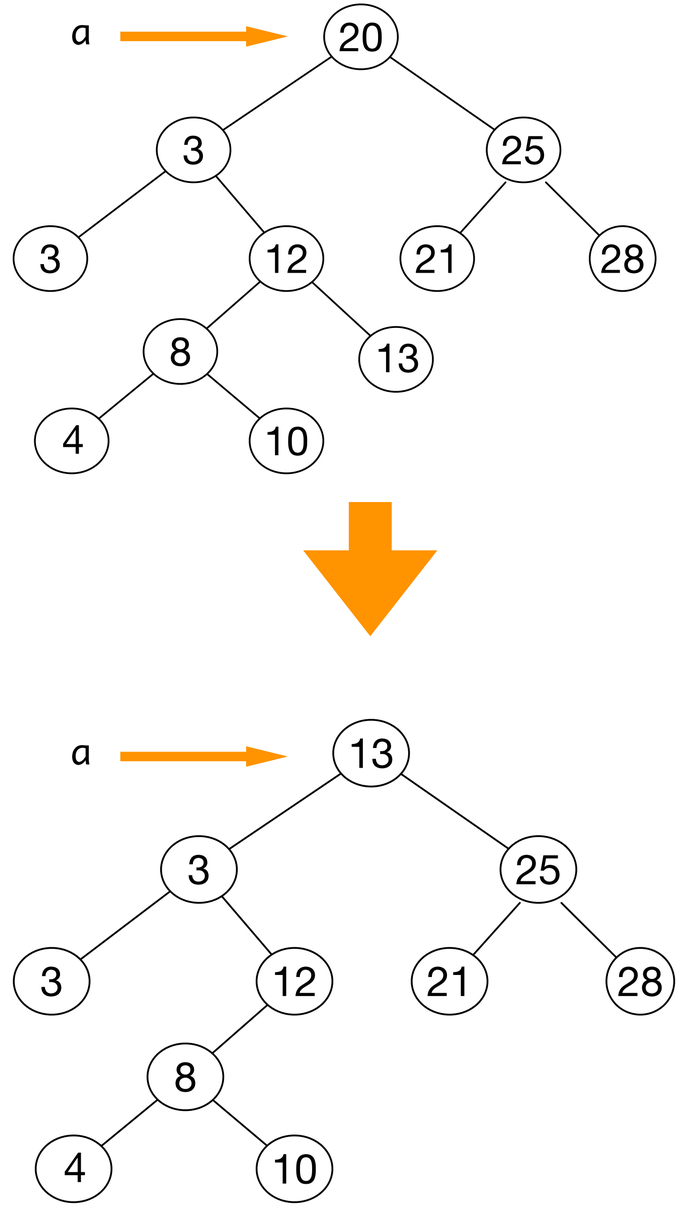
Arbres binaires de recherche

• supprimer une clé

on la supprime simplement si la clé est dans une feuille

sinon on la remplace par la plus grande dans le sous-arbre de gauche ou la plus petite dans le sous-arbre de droite

le programme est plus compliqué

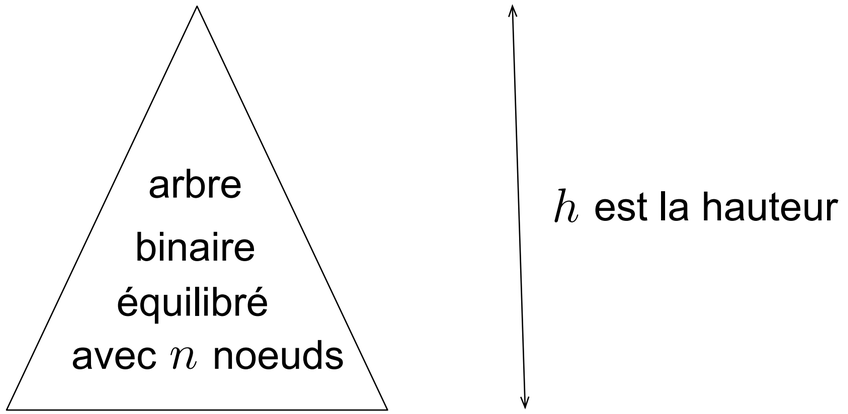


Exercice écrire la fonction `supprimer x a`

```
let b = supprimer 20 a ;;
```

Arbres binaires de recherche

- l'ajout d'une clé se fait sur une feuille
- la recherche et l'ajout dans un arbre binaire de recherche fait moins de `h` opérations où `h` est la **hauteur** de l'arbre
- la hauteur est `log(n)` pour un arbre de taille `n` si l'arbre binaire est **parfait**
- il faut donc veiller à ce que l'arbre de recherche soit **bien équilibré** pour que la recherche fasse `log(n)` opérations
- comment faire des arbres bien équilibrés ?



$$h \simeq \log n$$

$$n \simeq 2^h$$

Arbres de recherche équilibrés (AVL)

[Adelson-Velsky & Landis, 1962]

```
type 'a arbreAVL =
  | Feuille
  | Noeud of 'a arbreAVL * 'a * 'a arbreAVL * int ;;

let hauteur = function
  | Feuille -> 0
  | Noeud (_, _, _, h) -> h ;;

let noeud a x b = Noeud (a, x, b, 1 + max (hauteur a) (hauteur b)) ;;
```

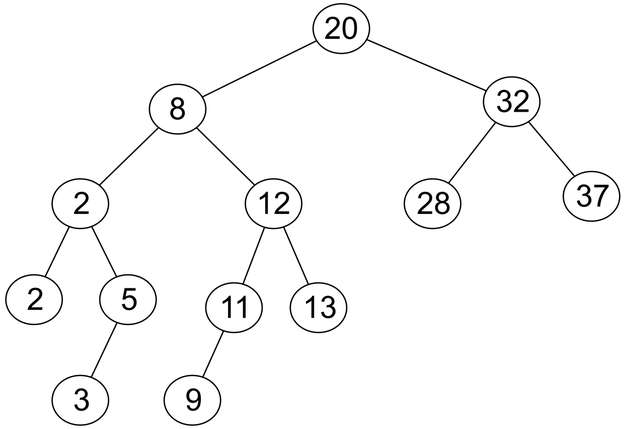
- on s'intéresse à la différence de hauteur entre fils gauche et droit

```
let bal = function
  | Feuille -> 0
  | Noeud (g, _, d, _) -> (hauteur d) - (hauteur g) ;;
```

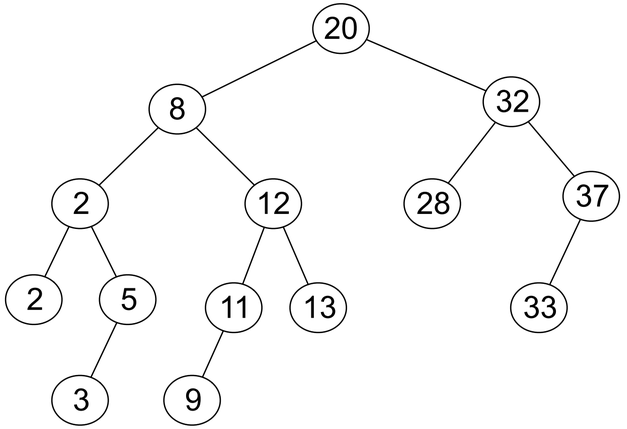
- les arbres AVL équilibrent les hauteurs des fils gauche et droit à une unité près

```
-1 <= bal a <= 1
```

Arbres de recherche équilibrés (AVL)

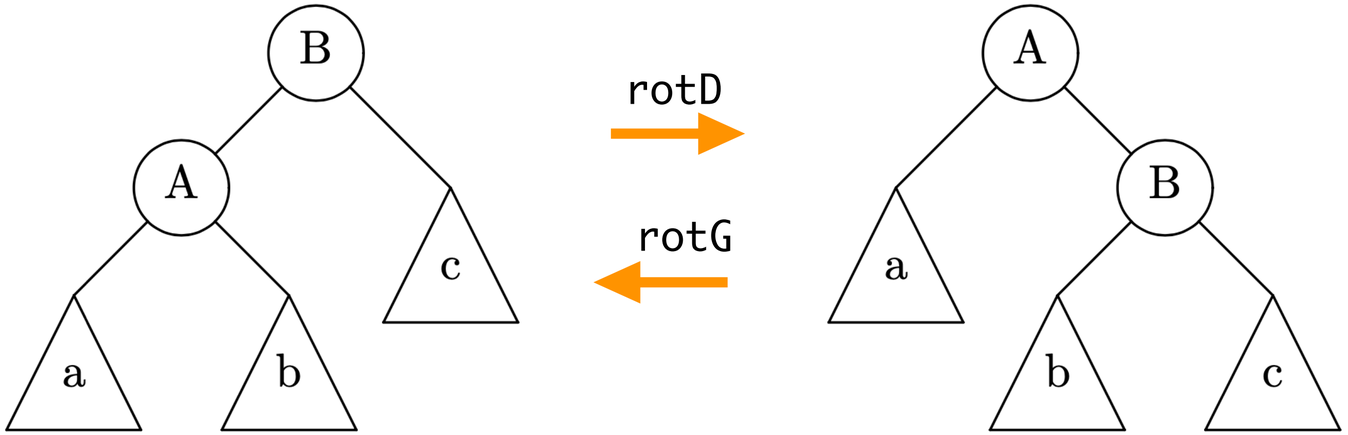


AVL ✗



AVL ✓

Arbres de recherche équilibrés (AVL)

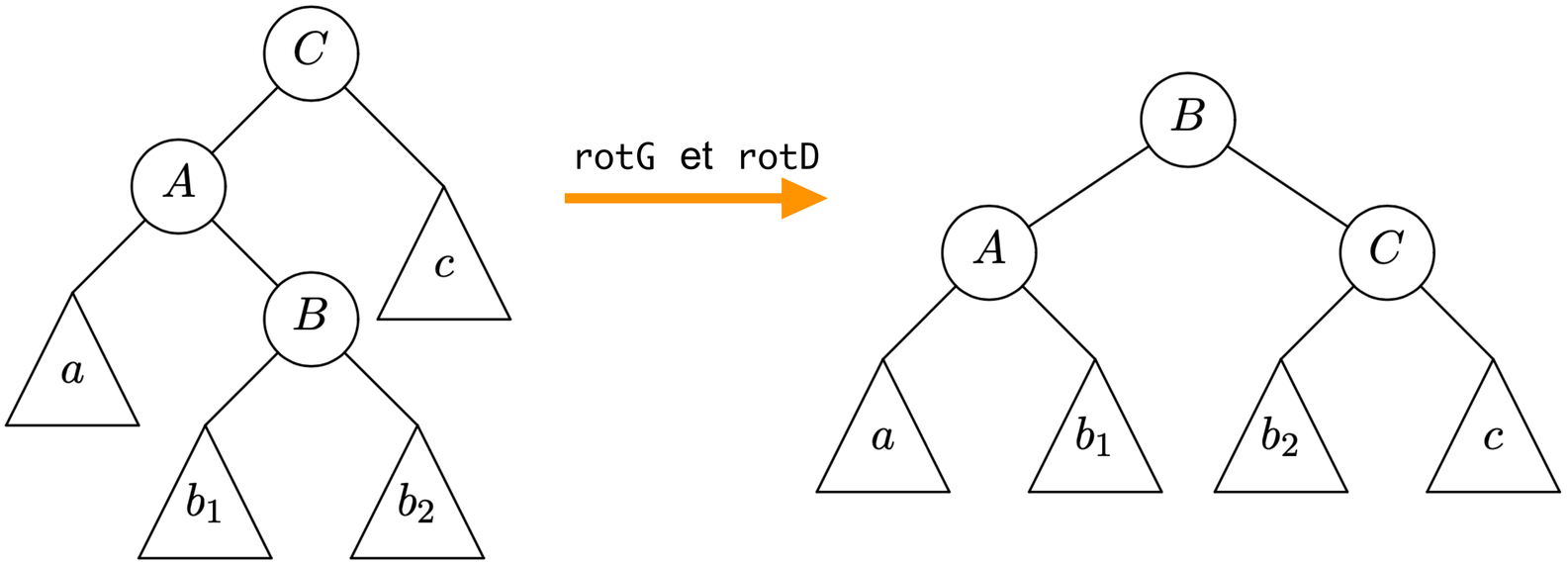



```
let rotD = function
  | Noeud (Noeud (a, x, b, _), y, c, _) ->
      noeud a x (noeud b y c)
  | _ -> raise Error;;
```

```
let rotG = function
  | Noeud (a, x, Noeud (b, y, c, _), _) ->
      noeud (noeud a x b) y c
  | _ -> raise Error;;
```

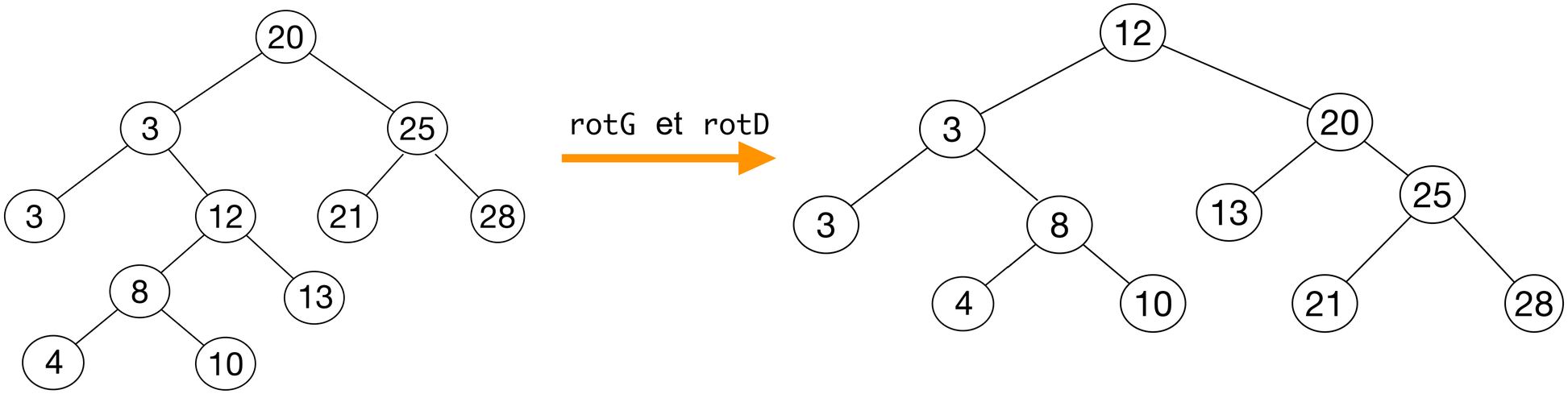
Arbres de recherche équilibrés (AVL)

- double rotation (gauche puis droite)



Arbres de recherche équilibrés (AVL)

- exemple de double rotation



Arbres de recherche équilibrés (AVL)

- ajouter une clé à un arbre AVL

```
let rec ajouter x a =
  let res = match a with
  | Feuille -> Noeud (Feuille, x, Feuille, 1)
  | Noeud (g, y, d, _) ->
      if x < y then noeud (ajouter x g) y d
      else if y < x then noeud g y (ajouter x d)
      else a in
  match res with
  | Feuille -> failwith "Impossible"
  | Noeud (g, y, d, _) ->
      if bal res < (-1) then
        if bal g < 0 then rotD res
        else rotD (noeud (rotG g) y d)
      else if bal res > 1 then
        if bal d > 0 then rotG res
        else rotG (noeud g y (rotD d))
      else
        res ;;
```

Exercice écrire la fonction `supprimer (x, a)` pour enlever une clé d'un arbre AVL

- ces fonctions sont bien compliquées

Arbres de recherche équilibrés

- on peut rendre **plus flexible** la loi d'équilibre des arbres AVL

- arbres 2-3

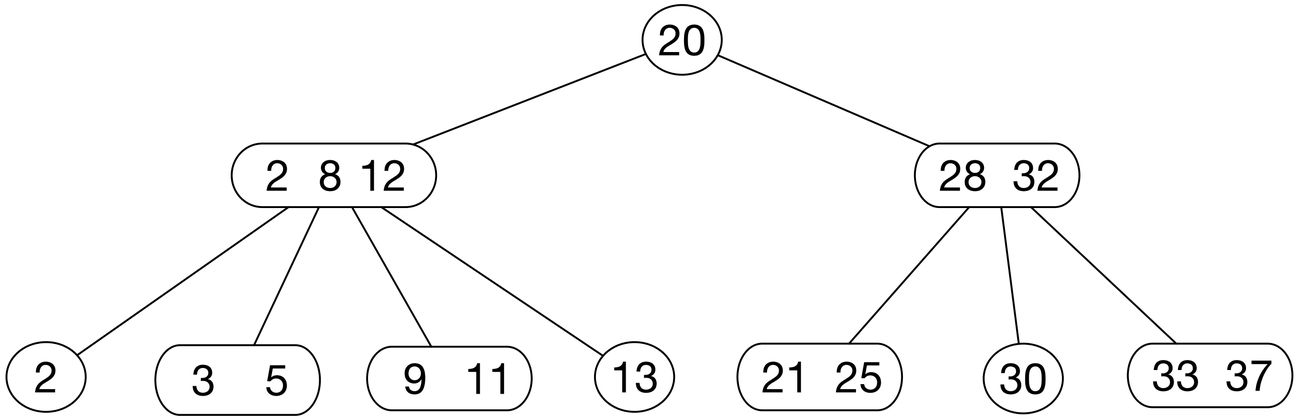
- arbres 2-3-4 ou plus généralement arbres-B *(B-trees*)

- arbres bicolores rouge-noir

Arbres de recherche équilibrés (2-3-4)

[Bayer & McCreight, 1970]

- les noeuds peuvent contenir 1, 2 ou 3 clés et donc 2, 3 ou 4 fils

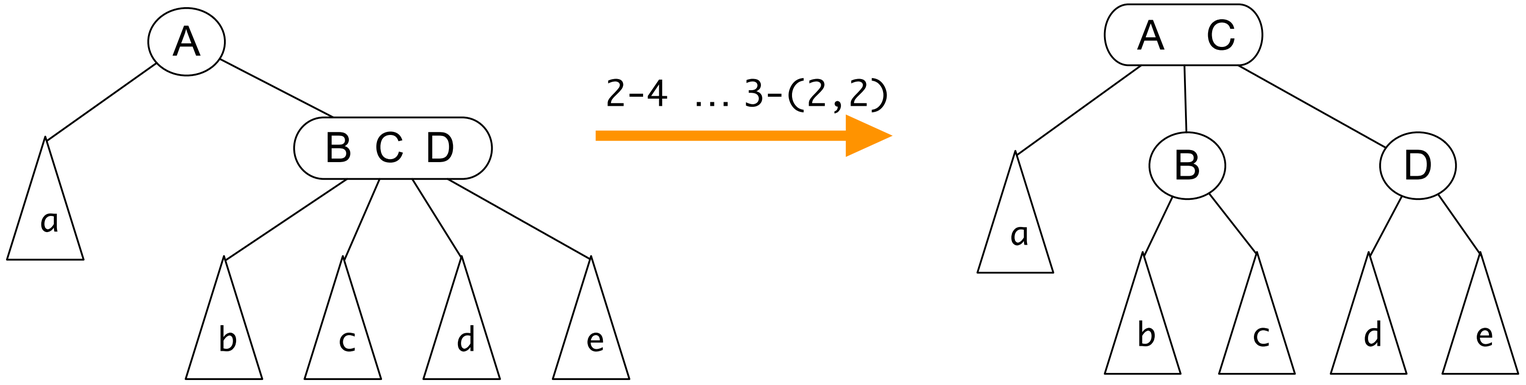


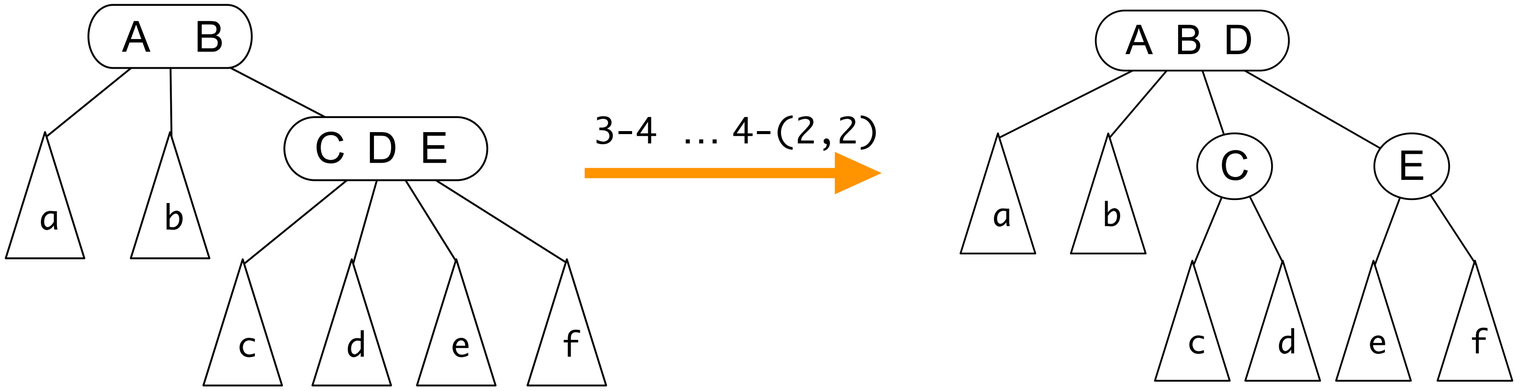
- on peut insérer une nouvelle clé dans tout noeud non quaternaire

- si impossible, on éclate le noeud quaternaire

Arbres de recherche équilibrés (2-3-4)

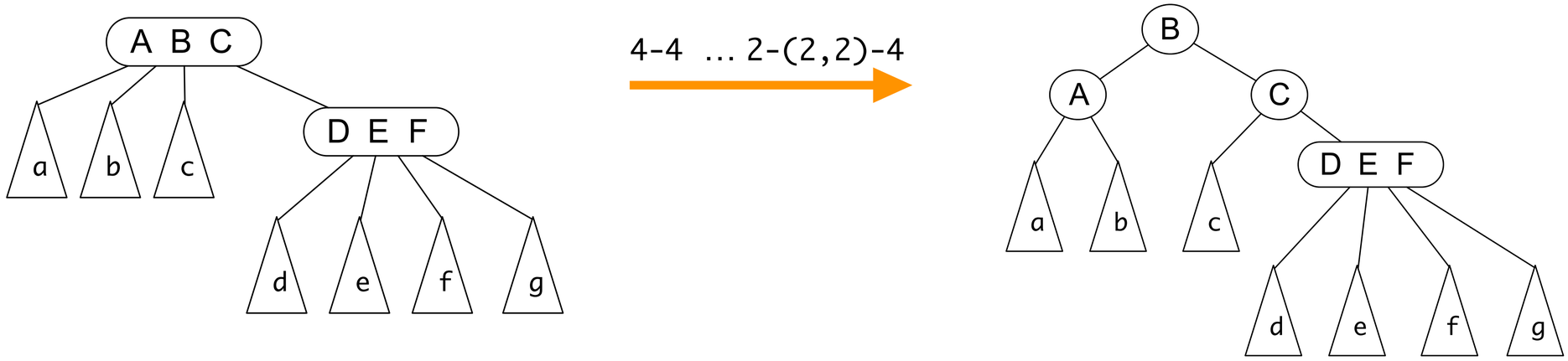
- on éclate les noeuds 4 sans augmenter la hauteur





Arbres de recherche équilibrés (2-3-4)

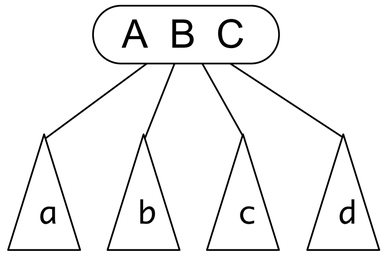
- on éclate un noeud 4 racine en augmentant la hauteur de 1 sans déséquilibrer les sous-arbres

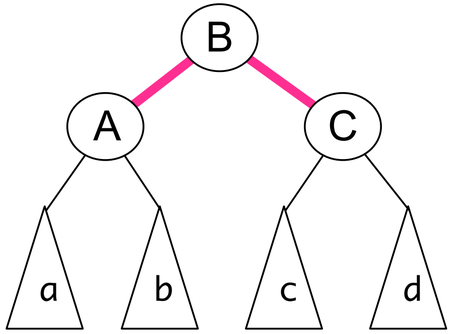


Arbres bicolores (rouge-noir)

[Guibas & Sedgewick, 1978]

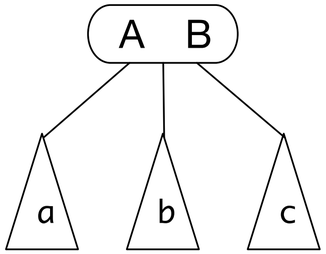
• on peut implémenter les arbres 2-3-4 par des arbres binaires bicolores

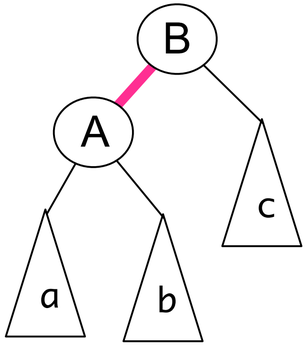




arbres 2-3-4

arbres bicolores

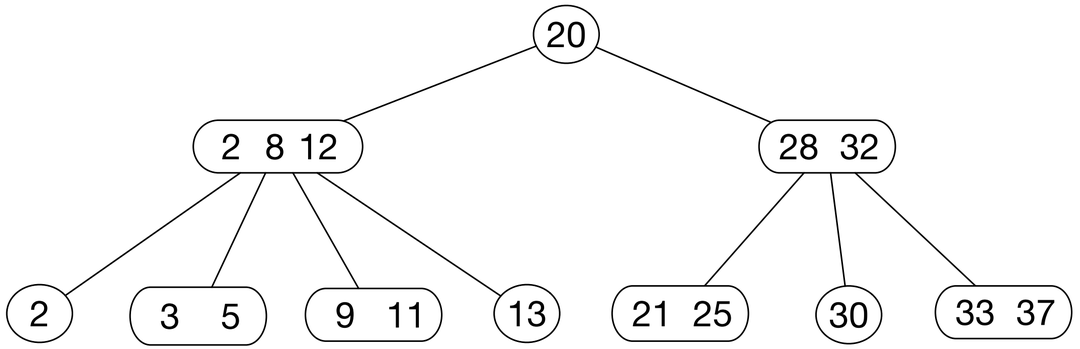


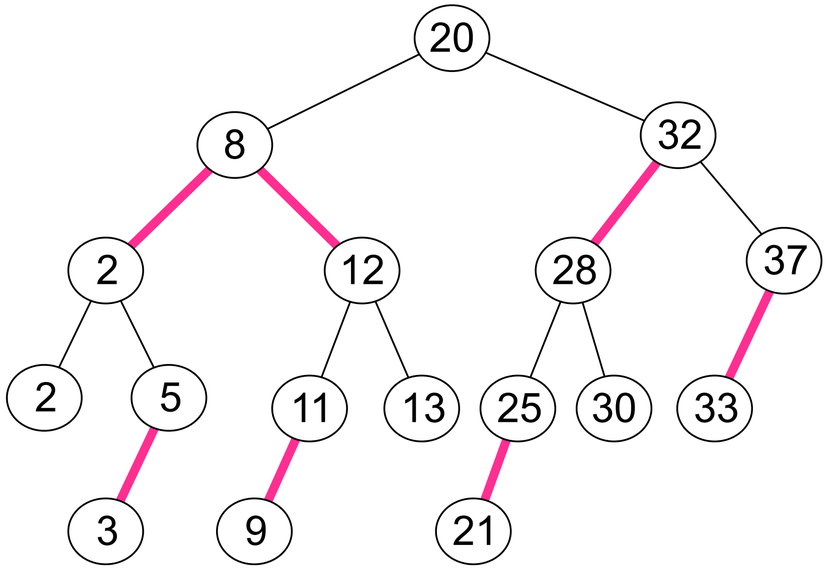


[on choisit
arbitrairement
vers la gauche]

Arbres bicolores (rouge-noir)

• on peut implémenter les arbres 2-3-4 par des arbres binaires bicolores





arbres 2-3-4 ⟷ arbres bicolores

• dans un arbre rouge-noir, le nombre de branches noires sur tout chemin d'un noeud à ses feuilles est le même

[tout noeud est équilibré par rapport à sa hauteur noire]

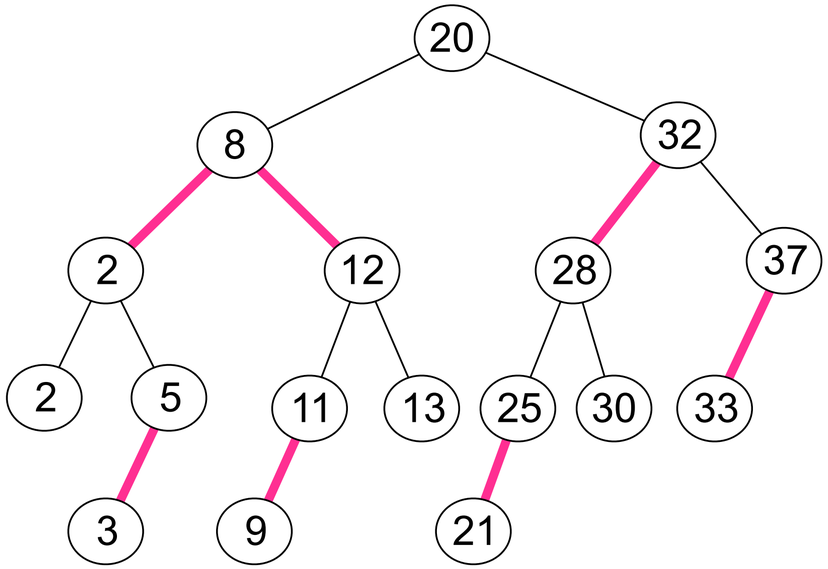
Arbres bicolores (rouge-noir)

• la couleur d'un noeud est la couleur de la branche qui la relie à son père

• le champ `couleur` est un simple booléen

```
type couleur = ROUGE | NOIR ;;

type 'a arbreRN =
  | Feuille
  | Noeud of couleur * 'a arbreRN * 'a * 'a arbreRN ;;
```



• dans un arbre rouge-noir, le nombre de branches noires sur tout chemin d'un noeud à ses feuilles est le même

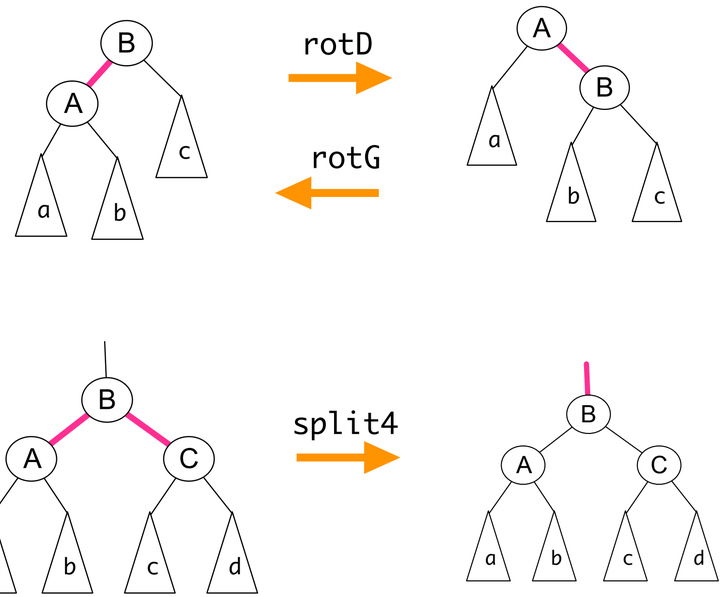
[tout noeud est équilibré par rapport à sa hauteur noire]

Arbres bicolores (rouge-noir)

- le programme complet pour les arbres rouge-noir

```
let balance a = match a with
  | Noeud (NOIR, Noeud (ROUGE, Noeud (ROUGE, a, x, b), y, c), z, d)
  | Noeud (NOIR, Noeud (ROUGE, a, x, Noeud (ROUGE, b, y, c)), z, d)
  | Noeud (NOIR, a, x, Noeud (ROUGE, Noeud (ROUGE, b, y, c), z, d))
  | Noeud (NOIR, a, x, Noeud (ROUGE, b, y, Noeud (ROUGE, c, z, d))) ->
      Noeud (ROUGE, Noeud (NOIR, a, x, b), y, Noeud (NOIR, c, z, d))
  | _ -> a ;;
```

```
let ajouter x a =
  let rec add x = function
  | Feuille -> Noeud (ROUGE, Feuille, x, Feuille)
  | Noeud (col, g, y, d) as a ->
     if x < y then balance (Noeud (col, add x g, y, d))
     else if x > y then balance (Noeud (col, g, y, add x d))
     else a in
  match add x a with
    | Noeud (_, g, y, d) -> Noeud (NOIR, g, y, d)
    | Feuille -> failwith "Impossible" ;;
```



Arbres bicolores (rouge-noir)

- on rajoute une fonction pour l'impression

```
let rec string_of_arbre = function
  | Feuille -> Printf.sprintf "_"
  | Noeud (ROUGE, Feuille, x, Feuille) ->
      Printf.sprintf "ROUGE %d" x
  | Noeud (NOIR, Feuille, x, Feuille) ->
       Printf.sprintf "NOIR %d"  x
  | Noeud (ROUGE, a, x, b) ->
       Printf.sprintf "ROUGE (%s, %d, %s)"
          (string_of_arbre a) x (string_of_arbre b)
 | Noeud (NOIR, a, x, b) ->
        Printf.sprintf "NOIR (%s, %d, %s)"
          (string_of_arbre a) x (string_of_arbre b) ;;

let print_arbre a = Printf.printf "%s\n" (string_of_arbre a) ;;
```

- exemple d'exécution

```
let a = Array.init 15 (fun i -> Feuille) ;;
for i = 1 to 10 do
  a.(i) <- ajouter (5 + i) a.(i-1) ;
  print_arbre a.(i)
done ;;
```

```
                                                                NOIR 6
                                                        NOIR (_, 6, ROUGE 7)
                                                    NOIR (NOIR 6, 7, NOIR 8)
                                            NOIR (NOIR 6, 7, NOIR (_, 8, ROUGE 9))
                                        NOIR (NOIR 6, 7, ROUGE (NOIR 8, 9, NOIR 10))
                                NOIR (NOIR 6, 7, ROUGE (NOIR 8, 9, NOIR (_, 10, ROUGE 11)))
                        NOIR (NOIR (NOIR 6, 7, NOIR 8), 9, NOIR (NOIR 10, 11, NOIR 12))
                NOIR (NOIR (NOIR 6, 7, NOIR 8), 9, NOIR (NOIR 10, 11, NOIR (_, 12, ROUGE 13)))
NOIR (NOIR (NOIR 6, 7, NOIR 8), 9, NOIR (NOIR 10, 11, ROUGE (NOIR 12, 13, NOIR 14)))
```

Au-delà des arbres de recherche

- algorithmes Diviser pour Régner (*divide and conquer*)
- géométrie (*computational geometry*)
- analyse syntaxique

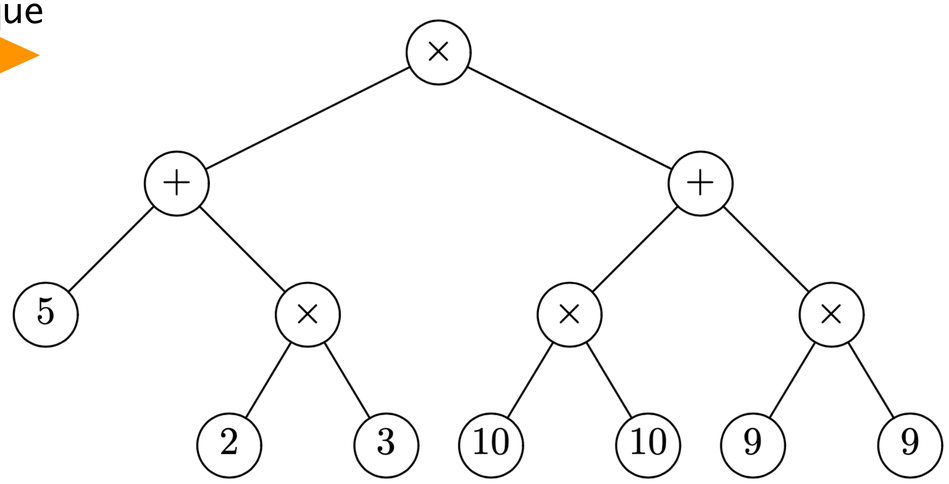
chaîne de caractères

(5 + 2 * 3) * (10 * 10 + 9 * 9)

analyse syntaxique

belle impression

arbre de syntaxe abstraite



- structure arborescente des systèmes de fichiers
- les arbres sont à la base des algorithmes de l'informatique

Conclusion

VU:

- récursivité
- listes
- filtrage
- exemple de structures fonctionnelles
- arbres

TODO list

- références
- données modifiables
- enregistrements
- modules