Colle 14

Arbres 2-3

MPII : Ocaml

On se propose ici de un type d'arbre qui permet une structure d'arbre de recherche (il n'est plus binaire) en garantissant une hauteur logarithmique en la taille de manière structurelle, contrairement aux arbres rouges-noir ou AVL qui ne parviennent à cette hauteur qu'au prix d'une correction de la structure à chaque opération.
Plus précisément les nœuds auront 2 fils ou 3 fils et cette souplesse permet d'obtenir des arbres pour lesquels les feuilles seront toutes à la même profondeur : les arbres seront parfaits.
On choisit aussi une structure hétérogène : l'information utile est située dans les feuilles, les nœuds se servent qu'à orienter le cheminement.
Nous nous contenterons de clés entières pour simplifier l'étude mais la généralisation est immédiate.

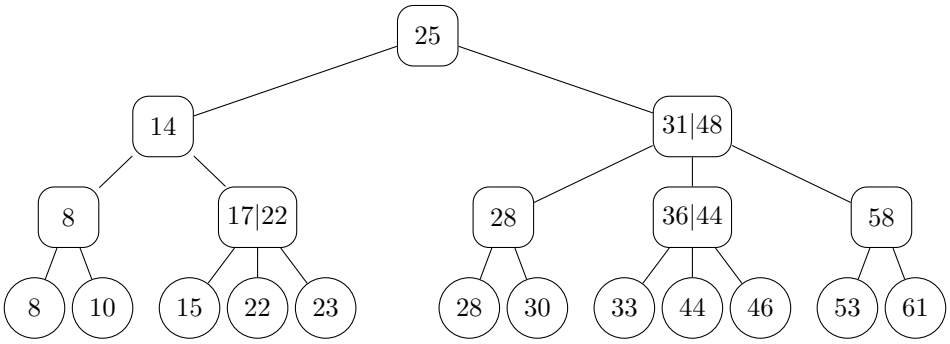
I Définition

Nos arbres 2-3 sont définis par le type :

```
type arbre23 = |F of int
               |N2 of arbre23 * int * arbre23
               |N3 of arbre23 * int * arbre23 * int * arbre23;;
```

Cette structure sera utilisée avec des conditions supplémentaires.

- Toutes les feuilles sont à la même profondeur
- Les valeurs des nœuds et feuilles des descendants d'un nœud `N2` vérifient les mêmes conditions que pour un arbre binaire de recherche.
- Pour un nœud `N3(g, a, m, b, d)` les valeurs des nœuds et feuilles sont bornées.
  - Les valeurs des nœuds et feuilles de `g` sont inférieures ou égales à `a`.
  - Les valeurs des nœuds et feuilles de `m` appartiennent à $\{a+1, a+2, \dots, b\}$ avec $a < b$.
  - Les valeurs des nœuds et feuilles de `d` sont strictement supérieures à `b`.



On remarque qu'un tel arbre contient au moins une feuille : il n'y a pas d'arbre vide.

Exercice 1
Écrire une fonction `chercher n a` qui cherche si une valeur n est une feuille de l'arbre a.

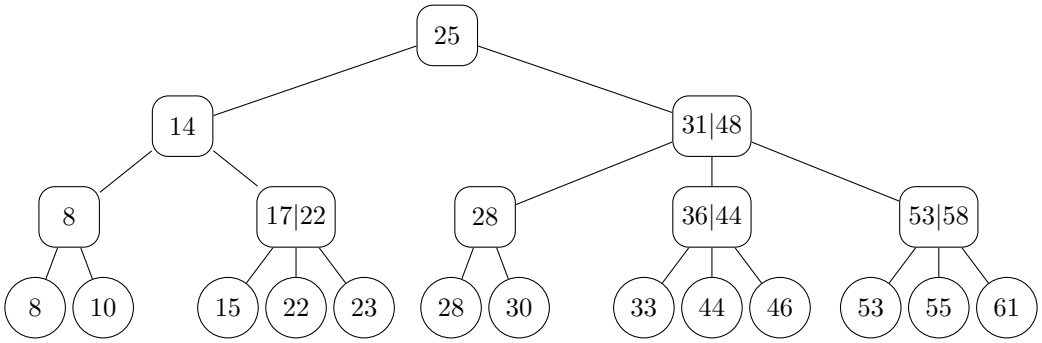
II Adjonction

L'adjonction se fait en ajoutant une feuille au nœud qui doit la contenir. par deux feuille : le père doit avoir un fils de plus.

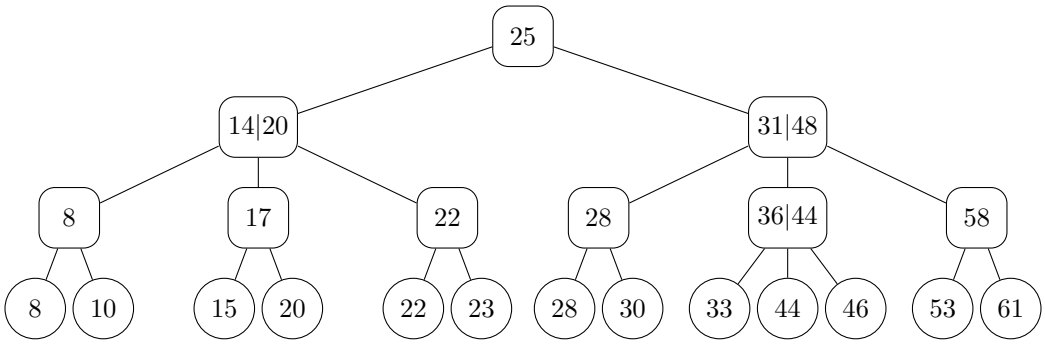
- Si le père est un nœud `N2`, on le transforme en nœud `N3` et le travail s'arrête.
- Si le père est un nœud `N3`, on le transforme en 2 nœuds `N2` et le père de ce nœud doit avoir un fils de plus. On recommence alors le processus pour le père.

Voici quelques exemple.

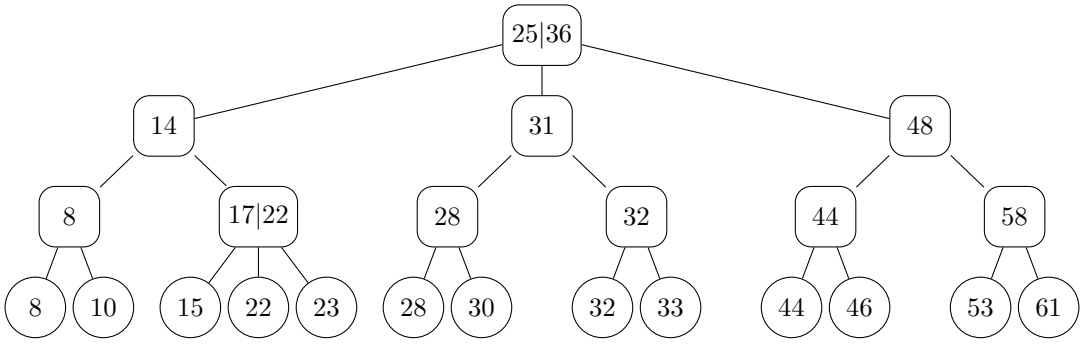
Adjonction de 55 à l'exemple



Adjonction de 20 à l'exemple



Adjonction de 32 à l'exemple



On voit donc qu'on a besoin de "remonter" de l'information lorsqu'on descend récursivement dans l'arbre.
Pour gérer cette remontée d'information, on peut enrichir la réponse de la fonction et utiliser cette réponse pour adapter les cas, c'est ce qui a été fait dans le cas des arbres rouge-noir.

Une autre possibilité est d'envoyer une exception qui signifie le besoin d'ajouter un fils et gérer cette exception avec `try ... with`. C'est cette méthode que nous allons utiliser ici.
Une exception peut envoyer des paramètres : ici, lors de l'ajout d'un fils, on va renvoyer les deux fils et la valeur de séparation d'où la définition

```
exception Eclatement of arbre23 * int * arbre23;;
```

Ainsi une fonction d'ajout pourra commencer par

```
let rec aux_add n arbre =
  match arbre with
  |F k when n = k -> F k
  |F k when n < k -> raise (Eclatement (F n, n, F k))
  ...
```

La récursivité deviendra

```
  ...
  |N2 (g, k, d) when n <= k
     -> begin try N2 (aux_add n g, k, d)
        with Eclatement (a1, p, a2) -> N3(a1, p, a2, k, d) end
  ...
```

Dans le cas des nœuds `N3`, le traitement de l'exception consistera à lever encore une exception.

```
  ...
  |N3 (g, k, m, l, d) when n <= k
    -> begin try N2 (aux_add n g, k, m, l, d)
       with Eclatement (a1, p, a2)
         -> raise (Eclatement (N2 (a1, p, a2), k, N2 (m, l, d)))
            end
  ...
```

La fonction est décrite comme une fonction auxiliaire ; en effet il se peut que l'appel initial renvoie une exception, il faudra alors rattraper celle-ci dans le corps de la fonction principale. Ce sera le cas si le parcours jusqu'à la feuille à ajouter ne comporte que des nœuds `N3` ou si on ajoute une valeur à un arbre réduit à une feuille (on passe de 1 à 2 éléments). C'est la seule possibilité d'augmenter la hauteur de l'arbre.

Exercice 2
Écrire une fonction `ajouter n a` qui ajoute une valeur n dans un arbre a.
Si la valeur existait déjà on ne l'ajoute pas.

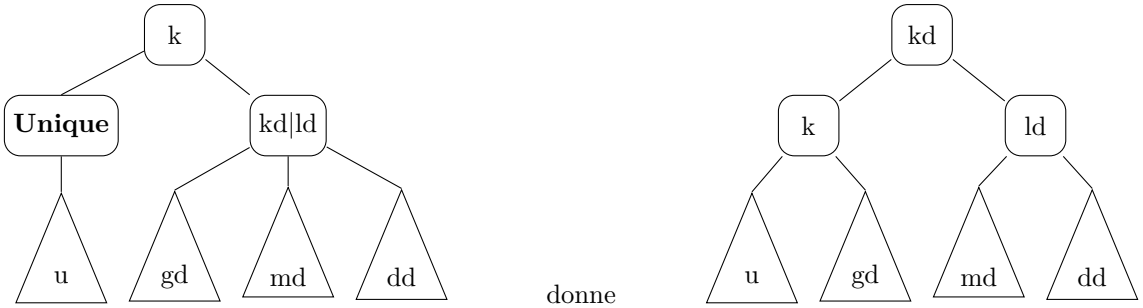
Exercice 3
Écrire une fonction `trier23 :list -> list` qui trie une liste en utilisant un arbre 2-3.
Quelle est sa complexité ?

III Suppression

Si on veut supprimer une feuille dans un nœud `N3`, il suffit de le transformer en nœud `N2`.
Par contre, dans un nœud `N2`, on arrive à un nœud avec un fils unique, ce qui n'est pas possible.
On gère ce problème avec une exception

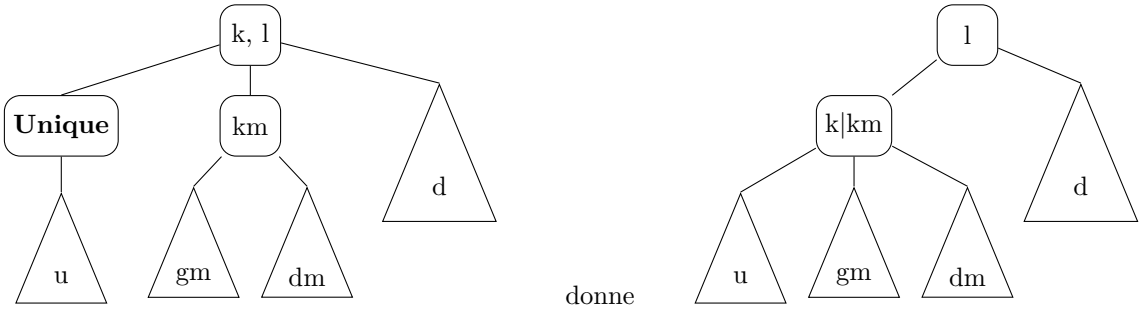
```
exception Unique of arbre23;;
```

Si cette exception parvient comme fils d'un nœud dont un frère est un nœud `N3`, on peut transformer en 2 nœuds `N3` (on représente ici le cas d'un père `N2`, le cas `N3` est semblable).



Si cette exception est voisine d'un arbre de `N2`, on fusionne en un nœud `N3`.
Pour un père de type `N2`, cela fait prolonger l'exception

Pour un père de type `N3`, on arrive à un nœud de type `N2`.



Exercice 4
Écrire une fonction `supprimer n a` qui supprime une valeur n dans un arbre a.
Si la valeur n'existait déjà on renvoie un arbre semblable.

Solutions

Solution de l'exercice 1

```
let rec chercher n a =
  match a with
  |F k -> k = n
  |N2 (g, k, d) when n <= k ->  chercher n g
  |N2 (g, k, d)             ->  chercher n d
  |N3 (g, k, m, l, d) when n <= k ->  chercher n g
  |N3 (g, k, m, l, d) when n <= l ->  chercher n m
  |N3 (g, k, m, l, d)             ->  chercher n d;;
```

Solution de l'exercice 2

```
let ajouter n a =
  let rec aux_add n a =
    match a with
    |F k when n < k -> raise (Eclatement (F n, n, F k))
    |F k when n = k -> F k
    |F k  -> raise (Eclatement (F k, k, F n))
    |N2 (g, k, d) when n <= k
      -> begin try N2 (aux_add n g, k, d)
          with Eclatement (a1, p, a2) -> N3 (a1, p, a2, k, d) end
    |N2 (g, k, d)
      -> begin try N2 (g, k, aux_add n d)
          with Eclatement (a1, p, a2) -> N3 (g, k, a1, p, a2) end
    |N3(g, k, m, l, d) when n <= k
      -> begin try N3(aux_add n g, k, m, l, d)
          with Eclatement(a1, p, a2)
            -> raise (Eclatement (N2(a1, p, a2), k, N2 (m, l, d)))
               end
    |N3(g, k, m, l, d) when n <= l
      -> begin try N3(g, k, aux_add n m, l, d)
          with Eclatement(a1, p, a2)
            -> raise (Eclatement (N2(g, k, a1), p, N2 (a2, l, d)))
               end
    |N3(g,k,m,l,d)
      -> begin try N3(g, k, m, l, aux_add n d)
          with Eclatement(a1, p, a2)
            -> raise (Eclatement (N2(g, k, m), l, N2 (a1, p, a2)))
               end
  in try aux_add n a
     with Eclatement(a1, p, a2) -> N2 (a1, p, a2);;
```

Solution de l'exercice 3

```
let rec vers23 liste =
  match liste with
  |[] -> failwith "Un arbre vide est impossible"
  |[k] -> F k
  |t::q -> ajouter t (vers23 q);;
```

```
let parcours arbre =
  let rec aux_par a fait =
    match a with
    |F k -> k :: fait
    |N2(g, k, d) -> aux_par g (aux_par d fait)
    |N3(g, k, m, l, d) -> aux_par g (aux_par m (aux_par d fait))
  in aux_par arbre [];;
```

```
let trier23 l = parcours (vers23 l);;
```

Solution de l'exercice 4

```
let supprimer n a =
  let rec aux_rem n a =
    match a with
    |F k when n = k -> failwith "Ceci ne devrait pas arriver"
    |F k -> F k
    |N2 (g, k, d) when g = F n -> raise (Unique d)
    |N2 (g, k, d) when d = F n -> raise (Unique g)
    |N3 (g, k, m, l, d) when g = F n -> N2(m, l, d)
    |N3 (g, k, m, l, d) when m = F n -> N2(g, k, d)
    |N3 (g, k, m, l, d) when d = F n -> N2(g, k, m)
    |N2 (g, k, d) when n <= k
      -> begin try N2 (aux_rem n g, k, d)
          with Unique u -> match d with
            |N2 (gd, kd, dd) -> raise (Unique (N3 (u, k, gd, kd,
               dd)))
            |N3 (gd, kd, md, ld, dd) -> N2 (N2 (u, k, gd), kd, N2
               (md, ld, dd))
            |_ -> failwith "Ceci ne devrait pas arriver" end
    |N2 (g, k, d)
      -> begin try N2 (g, k, aux_rem n d)
          with Unique u -> match g with
            |N2 (gg, kg, dg) -> raise (Unique (N3 (gg, kg, dg, k,
               u)))
            |N3 (gg, kg, mg, lg, dg) -> N2 (N2 (gg, kg, mg), lg,
               N2 (dg, k, u))
            |_ -> failwith "Ceci ne devrait pas arriver" end
    |N3(g, k, m, l, d) when n <= k
      -> begin try N3(aux_rem n g, k, m, l, d)
          with Unique u -> match m with
            |N2 (gm, km, dm) -> N2 (N3 (u, k, gm, km, dm), l, d)
            |N3 (gm, km, mm, lm, dm) -> N3 (N2 (u, k, gm), km, N2(
               mm, lm, dm), l, d)
            |_ -> failwith "Ceci ne devrait pas arriver" end
    |N3(g, k, m, l, d) when n <= l
      -> begin try N3(g, k, aux_rem n m, l, d)
          with Unique u -> match d with
            |N2 (gd, kd, dd) -> N2 (g, k, N3 (u, l, gd, kd, dd))
            |N3 (gd, kd, md, ld, dd) -> N3 (g, k, N2 (u, l, gd),
               kd, N2(md, ld, dd))
            |_ -> failwith "Ceci ne devrait pas arriver" end
    |N3(g,k,m,l,d)
      -> begin try N3(g, k, m, l, aux_rem n d)
          with Unique u -> match m with
            |N2 (gm, km, dm) -> N2 (g, k, N3 (gm, km, dm, l, u))
            |N3 (gm, km, mm, lm, dm) -> N3 (g, k, N2 (gm, km, mm),
                lm, N2(dm, l, u))
            |_ -> failwith "Ceci ne devrait pas arriver" end
  in match a with
    |F k when n = k -> failwith "Un arbre vide est impossible"
    |a -> try aux_rem n a
     with Unique u -> u;;
```