Cours Informatique MP.

Arbres binaires de recherche (ABR).

Antoine MOTEAU
antoine.moteau@wanadoo.fr

. . . /arbreabr.tex (2003)
. . . /arbreabr.tex Compilé le vendredi 09 mars 2018 à 16h 26m 29s avec LaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

Arbres binaires de recherche (ABR).

Table des matières

Arbres binaires de recherche (ABR).

1 Introduction

Lorsque l'on veut définir un ensemble de données où chaque donnée (information) est associée à une clé d'accès (adresse), fonctionnant sur le modèle d'un dictionnaire, il est souhaitable de mettre en place une structure de données qui, entre autres,

1. réduit les temps d'accès : recherche, insertion ou suppression de l'information associée à une clé,
2. soit capable d'autoriser la présence d'homonymes (si on le souhaite), c'est à dire d'associer des informations distinctes à des valeurs de clé identiques,
3. permette l'accès à des informations de taille variable,
4. n'occupe que le volume nécessaire aux informations stockées,
5. etc ...

Une structure de tableau simple (indicé par une valeur de clé), si elle permet un accès rapide (direct) à l'information (condition 1), oblige à utiliser des clés consécutives, en nombre limité, avec des trous pour les clés inutilisées et ne satisfait pas aux conditions 2 et 4 ci-dessus.

Parmi les structures qui ont été proposées pour satisfaire aux conditions 1, 2, 3 et 4, on trouve en particulier

- Les tables de hachage
  - à adressage indirect, avec liste de collisions des données associées à une même valeur de clé,
  - à adressage ouvert (données repoussées).
- Les arbres binaires de recherche (avec une vue particulière de la condition 2).

Ce sont les arbres binaires de recherche qui font l'objet de ce chapitre.

2 Définitions

Définition 2.1.

Un arbre binaire de recherche (**ABR**) est un arbre binaire homogène tel que

- les étiquettes des nœuds sont constituées de deux parties :
  - la clé, qui appartient à un ensemble totalement ordonné $(E, \leqslant)$,
  - une information complémentaire (valeur associée à la clé) ;
- la clé de chaque nœud est
  - supérieure ou égale à celle de tout nœud de l'arbre gauche,
  - inférieure ou égale à celle de tout nœud de l'arbre droit.

Un arbre binaire de recherche sera dit **"strict"** si la clé d'un nœud est strictement supérieure à celle de tout nœud de l'arbre gauche et strictement inférieure à celle de tout nœud de l'arbre droit (un arbre binaire de recherche est strict si et seulement si les clés y sont uniques).

Dans un arbre binaire de recherche, pour une valeur de clé donnée,

- la recherche s'effectue uniquement dans les sous-arbres susceptibles de contenir la clé : sous-arbre gauche ou (exclusivement) sous-arbre droit ;
- la recherche s'arrête à la première rencontre d'un nœud satisfaisant à la valeur de clé.

Dans un arbre non strict (**"large"**), plusieurs nœuds pouvant contenir la même valeur de clé, seul le premier nœud trouvé est accessible, les autres nœuds, de même clé, sont **masqués** (ou non accessibles).
Un nœud masqué ne redevient accessible qu'après suppression des nœuds de même clé qui étaient atteints avant lui.

3 Exemples

3.1 Variables globales et variables locales dans un programme

On pourrait imaginer que les variables d'un programmes soient enregistrées dans un **ABR** (**large**, de type dictionnaire), avec le nom de variable comme clé.

- A l'entrée d'un sous programme, ses variables locales sont introduites dans l'arbre, au plus près de la racine (**devant** des variables globales de même nom).
- Pendant toute la portée du sous-programme, les variables globales de même nom que les variables locales sont masquées (inaccessibles) et seules les variables locales sont accessibles.
- Lorsque le sous-programme se termine, ses variables locales sont détruites (supprimées de l'arbre) et les variables globales qui étaient masquées sont alors à nouveau accessibles.

3.2 Ensembles de Caml (ABR au sens strict)

Extraits de l'aide Caml :

set : sets over ordered types
This module implements the set data structure, given a total ordering function over the set elements. All operations over sets are purely applicative (no side-effects). The implementation uses balanced binary trees, and is therefore reasonably efficient : insertion and membership take time logarithmic in the size of the set, for instance.

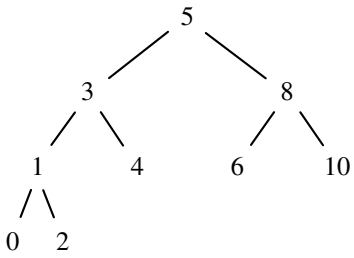
`type 'a t` (The type of sets containing elements of type 'a.)
...
`value iter : ('a -> 'b) -> 'a t -> unit`
iter f s applies f in turn to all elements of s, and discards the results. The elements of s are presented to f in a non-specified order.

`value choose : 'a t -> 'a`
Return one element of the given set, or raise Not_found if the set is empty. Which element is chosen is not specified, but equal elements will be chosen for equal sets.
...
Remarque.

- La définition précise du type "ensemble" n'est pas publique.
- La signature est réduite au minimum ;

Illustration :

```
let rec set_of_list = function
  | []   -> set__empty eq__compare      (* ordre générique *)
  | a::q -> set__add a (set_of_list q)
;;
(* set_of_list : 'a list -> 'a set__t = <fun> *)
```




```
let print_int_set = set__iter (function x -> print_int x; print_char ' ');;
(* print_int_set : int set__t -> unit = <fun> *)

let S = set_of_list [1;2;10;4;3;6;5;8;4;3;0];;   (* --> S : int set__t = <abstr> *)
print_int_set S;;              (* --> 0 1 2 3 4 5 6 8 10 - : unit = () *)
```

Remarques.

- Bien que la notion d'ensemble (fini) ne soit pas associée (mathématiquement) à un ordre sur les éléments, la mise en place matérielle, sous forme d'un arbre binaire de recherche, nécessite l'utilisation d'une relation d'ordre. Apparemment, Caml (light) utilise la relation d'ordre "naturelle" implicite (ou générique), dénommée `eq__compare` (et même <= dans les versions $\geqslant$ 0.74), qui est construite d'après les relations d'ordre usuelles sur les types simples (`int`, `float`, `char`, `string`), étendues à l'aide de l'ordre lexicographique sur les produits.
- La structure d'arbres binaires équilibrés utilisée ici par Caml (Light) n'est pas précisée (**AVL** ? **Arbres Rouge et Noir** ? autre ?) et c'est souhaitable : cette structure peut être remplacée par une structure plus performante dans l'avenir, la signature gardant rigoureusement le même comportement, ce qui assure la pérénité des anciens programmes (à condition qu'ils n'utilisent que la signature officielle, sans bidouilles à travers les failles de protection de confidentialité de la structure).

3.3 Le module map de Caml (ABR au sens large)

Extraits de l'aide Caml :

map : association tables over ordered types
This module implements applicative association tables, also known as finite maps or dictionaries, given a total ordering function over the keys. All operations over maps are purely applicative (no side-effects). The implementation uses balanced binary trees, and therefore searching and insertion take time logarithmic in the size of the map.

`type ('a, 'b) t`
The type of maps from type 'a to type 'b.

`value empty : ('a -> 'a -> int) -> ('a, 'b) t`
The empty map. The argument is a total ordering function over the set elements. This is a two-argument function f such that f e1 e2 is zero if the elements e1 and e2 are equal, f e1 e2 is strictly negative if e1 is smaller than e2, and f e1 e2 is strictly positive if e1 is greater than e2. Examples : a suitable ordering function for type int is prefix −. You can also use the generic structural comparison function eq__compare.

`value add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t`
add x y m returns a map containing the same bindings as m, plus a binding of x to y. Previous bindings for x in m are not removed, but simply hidden : they reappear after performing a remove operation. (This is the semantics of association lists.)

`value find : 'a -> ('a, 'b) t -> 'b`
find x m returns the current binding of x in m, or raises Not_found if no such binding exists.

`value remove : 'a -> ('a, 'b) t -> ('a, 'b) t`
remove x m returns a map containing the same bindings as m except the current binding for x. The previous binding for x is restored if it exists. m is returned unchanged if x is not bound in m.

`value iter : ('a -> 'b -> 'c) -> ('a, 'b) t -> unit`
iter f m applies f to all bindings in map m, discarding the results. f receives the key as first argument, and the associated value as second argument. The order in which the bindings are passed to f is unspecified. Only current bindings are presented to f : bindings hidden by more recent bindings are not passed to f.
...

Illustration :

```
let rec map_of_list = function
  | []       -> map__empty eq__compare                (* utiliser l'ordre générique *)
  | (a,b)::q -> map__add a b (map_of_list q)
;;
(* map_of_list : ('a * 'b) list -> ('a, 'b) t = <fun> *)

let list_of_map m = let L = ref [] in map__iter (fun a b -> L := (a,b)::!L ) m;  !L;;
(* list_of_map : ('a, 'b) t -> ('a * 'b) list = <fun> *)
```

Exemple de masquage et de dé-masquage :

```
let m = map_of_list [ 1,"info1"; 2,"info2"; 3, "info3"; 4, "info4"; 1, "info1bis" ];;
(* m : (int, string) map__t = <abstr> *)
list_of_map m;; (* (int * string) list = [4,"info4"; 3,"info3"; 2,"info2"; 1,"info1"] *)
```

• ⟼ On constate que l'élément `(1, "info1bis")` est masqué.

```
let m1 = map__remove 1 m;;                        (* m1 : (int, string) map__t = <abstr> *)
let m2 = map__add 2 "info2bis" m1;;               (* m2 : (int, string) map__t = <abstr> *)
list_of_map m2;; (* (int * string) list=[4,"info4"; 3,"info3"; 2,"info2bis"; 1,"info1bis"] *)
```

• ⟼ l'élément `(1, "info1bis")` est à nouveau accessible puisqu'il est le premier rencontré de clé 1.
• ⟼ l'élément `(2, "info2")` a été masqué par l'introduction d'un nouvel élément de même clé 2.

Remarque. Cela semble indiquer que l'insertion d'un nouvel élément se fait à la racine de l'arbre de recherche, comme le laisse supposer la documentation de `add` .

4 Signature d'un ABR

Pour décrire un **ABR**, on utilise une structure d'arbre binaire homogène normal, l'état "de recherche" n'étant qu'une qualité de l'arbre, non décrite par la structure arborescente, assurée uniquement par l'utilisation de fonctions qui conservent la qualité d'**ABR** (pour la relation d'ordre utilisée).
Cela conduit en général à la définition d'une signature dont l'écriture est privée (**type de données abstrait**), comme dans le cas des modules `set` et `map` de Caml, selon la stratégie suivante :

- Type inconnu pour l'utilisateur (type abstrait) : cela empêche l'utilisateur d'écrire des fonctions qui ne fonctionnerait plus en cas d'amélioration de la structure.
- Signature minimale (d'écriture privée), comportant toutes les fonctions vitales :
  - Constructeurs :
    - Arbre vide (avec éventuellement la description de la relation d'ordre à utiliser),
    - insertion d'une information attachée à une une valeur de clé unique,
    - insertion d'une information attachée à une une valeur de clé non unique.
  - Destructeur : suppression d'un élément selon une valeur de clé.
  - Prédicat : est vide .
  - Requêtes :
    - recherche d'information selon une valeur de clé,
    - minimum et maximum de valeur de clé,
    - clé successeur, clé prédécesseur d'une valeur de clé.
  - ...
- Signature étendue : par exemple des parcours de l'arbre (selon un ordre précisé ou non).

Remarque. La relation d'ordre à utiliser, sur le type des clés,
- peut être passée comme argument systématique aux fonctions de la signature,
- peut faire partie de la définition d'instances du type (cf. module **map** de Caml),
- peut éventuellement être la relation d'ordre par défaut (générique) (cf. module **set** de Caml).

Dans la suite, sauf avis contraire, toutes les illustrations utiliseront
- le type d'arbre binaire homogène :

```
type 'a arbre =                    (* type simplifié pour illustration *)
  | Vide
  | Noeud of ('a arbre) * 'a * ('a arbre)
```

  où une étiquette `e` de type `'a` est un couple (clé, valeur) :

```
let cle e = fst e and valeur e = snd e;;
```

- avec l'ordre par défaut (générique) sur le type de clé : `<=` (identique à `eq_compare`).

A priori, dans la définition d'**ABR**, la clé de chaque nœud est
- supérieure ou égale à celle de tout nœud de l'arbre fils gauche,
- inférieure ou égale à celle de tout nœud de l'arbre fils droit.

Lors de l'insertion on pourra faire en sorte que, en cas d'égalité, l'insertion se fasse du coté gauche, ce qui donnerait des clés strictement supérieures à celle des pères dans les fils droits, **à condition qu'une autre fonction (par exemple, suppression ou équilibrage) ne crée pas des cas d'égalité entre la clé d'un père et une clé de son fils droit !**

Par sécurité, les arbres qui présentent des cas d'égalité dans le fils droit devront être acceptés (et traités conformément à la définition).

Test d'un ABR. Pour assurer la conformité de la signature, le concepteur doit pouvoir vérifier qu'un arbre obtenu est bien un ABR :

Un arbre A est un **ABR** si et seulement si il est l'arbre vide (convention) ou si
- ses sous-arbres (fils) gauche (A_g) et (fils) droit (A_d) sont des **ABR**,
- le maximum des clés dans A_g est inférieur ou égal à la clé de la racine,
- le minimum des clés dans A_d est supérieur ou égal à la clé de la racine.

Dans un **ABR** strict, où l'étiquette est un couple (clé, valeur), la clé étant entière dans]min_int,max_int[:

```
let is_strict_int_ABR t = fst (test t)   (* test ABR à clé entière, ordre naturel *)
  where rec test = function
    | Vide         -> true, (max_int, min_int)
    | Noeud(g,e,d) -> let c = cle e in
                      let (bg, (ming, maxg)) = test g and (bd, (mind, maxd)) = test d in
                      bg & bd & (maxg < c) & (mind > c), (min ming c, max maxd c)
;;
(* is_strict_int_ABR : (int * 'a) arbre -> bool = <fun>*)

let a =      Noeud(Noeud(Noeud(Vide,(1,0),Vide),(2,0),Vide),
       (4,0),Noeud(Vide,(6,0),Noeud(Vide,(7,0),Vide)));;
is_strict_int_a_ABR a;;  (*- : bool = true *)
```

Dans un **ABR** large, où l'étiquette est un couple (clé, valeur), la clé n'étant pas forcément entière :

```
let is_large_ABR = function        (* test ABR à clé quelconque, ordre naturel *)
 | Vide -> true
 | t    -> fst (test t)
 where rec test = function
  | Vide               -> failwith "Cas impossible"
  | Noeud(Vide,e,Vide) -> (true, (c, c))
  | Noeud(Vide,e,d) -> let c = cle e in let (bd, (mind,maxd)) = test d in bd & (c <= mind),(c,maxd)
  | Noeud(g,e,Vide) -> let c = cle e in let (bg, (ming,maxg)) = test g in bg & (maxg <= c),(ming,c)
  | Noeud(g,e,d)    -> let c = cle e in
                       let (bg, (ming, maxg)) = test g and (bd, (mind, maxd)) = test d in
                       bg & bd & (maxg <= c) & (c <= mind), (ming, maxd)
;;
(* is_large_ABR : ('a * 'b) arbre -> bool = <fun> *)

let a = Noeud(Noeud(Noeud(Vide,(1,1),Vide),(2,2),Vide),(4,4),
              Noeud(Vide,(6,6),Noeud(Vide,(7,7), Noeud(Vide,(7,7), Vide))));;
let b = Noeud(Noeud(Noeud(Vide,("un",1),Vide),("deux",2),Vide),("quatre",4),
              Noeud(Vide,("six",6),Noeud(Vide,("sept",7),Noeud(Vide,("sept",7), Vide))));;
is_large_ABR a;;  (*- : bool = true *)    is_large_ABR b;;  (*- : bool = false *)
```

5 Requêtes dans un ABR

Les requêtes, selon une valeur de clé, ne fournissent que le premier élément trouvé. Si l'arbre n'est pas strict, certains éléments peuvent être **masqués** (inaccessibles par une recherche normale).

5.1 Recherche selon une valeur de clé

```
let rec cherche_ABR x = function
  | Vide                          -> raise Not_found
  | Noeud(g, e ,d) when x =  cle e -> valeur e
  | Noeud(g, e ,d) when x <= cle e -> cherche_ABR x g
  | Noeud(g, _ ,d)                 -> cherche_ABR x d   (* when x > cle e *)
;;
(* cherche_ABR : 'a -> ('a * 'b) arbre -> 'b = <fun> *)

let a = Noeud(        Noeud( Noeud(Vide, (2, "canon") , Vide), (2,"arc"), Vide),
        (3,"javelot") ,Noeud(Vide, (5,"fusil") , Vide));;
cherche_ABR 2 a;; (* : string = "arc" *) cherche_ABR 6 a;; (* Uncaught exception: Not_found *)
```

Complexité maximale de la recherche : (l'unité de coût est la comparaison de clés).
Lorsque la racine ne correspond pas à la clé, on recherche dans un seul des sous-arbres gauche ou droit, ce qui fait rechercher dans un sous-arbre de hauteur diminuée de 1. C'est donc la hauteur h qui est le paramètre significatif pour le coût C de recherche : $C(h) = 1 + C(h-1)$, d'où $\boxed{C(h) = h \text{ comparaison de clés.}}$

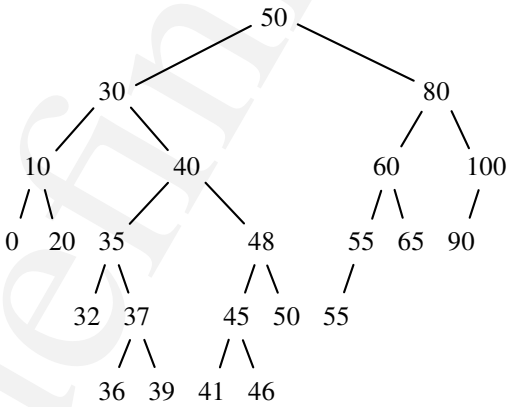
5.2 Minimum et maximum de clé

Pour effectuer une telle recherche, on n'a pas besoin de connaître la relation d'ordre ...

```
let rec mini_ABR  = function
  | Vide            -> failwith "Arbre vide"
  | Noeud(Vide,e,_) -> cle e
  | Noeud(g,_,_)    -> mini_ABR g
and     maxi_ABR  = function
  | Vide            -> failwith "Arbre vide"
  | Noeud(_,e,Vide) -> cle e
  | Noeud(_,_,d)    -> maxi_ABR d
;;
(* mini_ABR : ('a * 'b) arbre -> 'a = <fun>
   maxi_ABR : ('a * 'b) arbre -> 'a = <fun>  *)
```

Dans le cas particulier d'arbres à clés entières
dans] min_int, max_int [$=]-\infty,+\infty[$, on pourrait écrire :

```
let rec mini_int_ABR  = function
 | Vide             -> max_int
 | Noeud(g,(c,v),_) -> min c (mini_int_ABR g)
and     maxi_int_ABR  = function
 | Vide             -> min_int
 | Noeud(_,(c,v),d) -> max c (maxi_int_ABR d)
;;
(* mini_int_ABR : (int * 'a) arbre -> int = <fun>
   maxi_int_ABR : (int * 'a) arbre -> int = <fun> *)
```



Exemple. Soit l'arbre dessiné ci-dessus (graphe réduit aux clés)
et construit "automatiquement" :

```
let a = list_it (fun x -> insert_feuille_ABR (x,x))  (* VOIR insert_feuille_ABR ci-après *)
        [36;39;65;55;46;90;55;50;37;41;45;48;32;35;20;0;100;60;40;10;80;30;50] Vide;;
mini_int_ABR a;;  (* --> - : int = 0   *)  mini_ABR a;;  (* --> - : int = 0   *)
maxi_int_ABR a;;  (* --> - : int = 100 *)  maxi_ABR a;;  (* --> - : int = 100 *)
```

> La complexité maximale, en fonction de la hauteur h, est $C(h) = h$ unité de coût
> l'unité de coût étant un appel de fonction, sans comparaison de clé.

5.3 Successeur ou prédécesseur (strict) d'une valeur de clé, dans un ABR

Pour effectuer une telle recherche, on n'a pas besoin de connaître la relation d'ordre ...
On notera l'utilisation pertinente de la programmation par exception (`try ... with ... -> ...`):
"essayer" (`try`) ... "et si échec, selon la nature de l'échec (`with ...`) , faire" (`->`)' ...

```
let rec succ_ABR x = function
  | Vide                        -> failwith "Pas de successeur strict"
  | Noeud(g, e, d) when x < cle e  -> (try succ_ABR x g with Failure s -> cle e)
  | Noeud(g, e, d)                 ->  succ_ABR x d
and     pred_ABR x = function
  | Vide                        -> failwith "Pas de prédécesseur strict"
  | Noeud(g, e, d) when x > cle e  -> (try pred_ABR x d with Failure s -> cle e)
  | Noeud(g, p, d)                 ->  pred_ABR x g
;;
(* succ_ABR : 'a -> ('a * 'b) arbre -> 'a = <fun>
   pred_ABR : 'a -> ('a * 'b) arbre -> 'a = <fun> *)
```

```
let a = list_it (fun x -> insert_feuille_ABR (x,x))  (* Voir insert_feuille_ABR ci-après *)
        [36;39;65;55;46;90;55;50;37;41;45;48;32;35;20;0;100;60;40;10;80;30;50] Vide;;
succ_ABR 40 a;;  (* --> - : int = 41 *)       pred_ABR 40 a;;  (* --> - : int = 39 *)
pred_ABR 00 a;;  (* --> Uncaught exception: Failure "Pas de prédécesseur strict."  *)
```

> La complexité maximale, en fonction de la hauteur h, est $C(h) = h$ unité de coût
> l'unité de coût étant un appel de fonction, sans comparaison de clé.

5.4 Complexité des requêtes dans un ABR

5.4.1 Complexité maximale

Théorème 5.1.

Dans un arbre binaire de recherche de hauteur h, le coût maximal d'accès à un nœud est un $O(h)$ (comparaisons de clés ou appels récursifs, selon que la requête nécessite ou non des comparaisons).

Preuve. Au pire on recherche dans un seul fils, de hauteur $h-1$, donc $C_{max}(h) = 1 + C_{max}(h-1)$.

5.4.2 Réduction de la complexité maximale

On a vu que,

- dans un arbre binaire à n nœuds, strictement équilibré, la hauteur h est un $O(\log_2(n))$, et plus précisément $h \underset{+\infty}{\sim} \log_2(n)$ et même $\log_2(n+1) \leqslant h \leqslant \log_2(n)+1$;
- dans un arbre binaire qui n'est que H-équilibré (où la différence de hauteur entre deux fils est au plus 1), la hauteur h est aussi un $O(\log_2(n))$, ce $O(\log_2(n))$ étant (légèrement) moins favorable que le $O(\log_2(n))$ de l'équilibre strict (on a obtenu la majoration $h \leqslant 1.45\log_2(n+2) - 0.32$).

Le coût maximal des requêtes sera donc minimal dans des **ABR** strictement équilibrés, quasi minimal dans des **ABR** H-équilibrés et sera maximal dans des **ABR** complètement déséquilibrés (arbres peignes où $h = n$).

Remarque. La transformation en **ABR** (brut), d'une liste déjà triée de n éléments, conduit à un arbre peigne, de hauteur maximale $h = n$.

5.4.3 Complexité moyenne

La complexité moyenne (hauteur moyenne) s'évalue dans un **ABR** (strict) quelconque de hauteur n, construit aléatoirement, par tirage équiprobable des n valeurs de clé (parmi n valeurs).

Théorème 5.2.

Dans un arbre binaire de recherche à **n nœuds**, *le coût moyen $P_{moy}(n)$, d'accès à un* **nœud**, *vérifie :*

$$P_{moy}(n) = O(\ln(n)) \text{ unités de coût} \qquad \left(\text{plus précisément, } P_{moy}(n) \underset{+\infty}{\sim} 2\ln(n)\right)$$

Remarques. Pour la recherche d'un élément particulier, c'est rigoureusement équivalent à la complexité moyenne d'accès à un nœud dans un arbre binaire quelconque à n nœuds (voir cours précédent) !
Le coût réel maximal de la recherche peut être au mieux $\log_2(n)+1$ (si équilibre strict), $2.08\log_2(n)$ (pour le H-équilibre) , ... , et au pire n (cas des arbres peignes).

Preuve. On suppose, sans nuire à la généralité, que les clés sont les entiers de 1 à n, ordonnés naturellement et que l'arbre est un **ABR** strict (les éléments masqués d'un **ABR** large sont inaccessibles en recherche normale, et un **ABR** large se comporte comme un **ABR** strict).

Si $M(n)$ est la (valeur) moyenne de la somme de toutes les profondeurs de nœuds dans des **ABR** à n nœuds, la (valeur) moyenne de profondeur de nœuds sera $P_m(n) = \dfrac{M(n)}{n}$.
On a $M(0) = 0$ (et $M(1) = 0$) .
Pour $n > 1$, construisons un **ABR** $\mathscr{A}$, par tirage au hasard dans $[\![1..n]\!]$, avec équiprobabilité et sans remise :
La première valeur insérée est k, tiré au hasard dans $[\![1..n]\!]$ avec une probabilité de $\dfrac{1}{n}$ et, ensuite

- les $k-1$ valeurs de $[\![1..n]\!]$ strictement inférieures à k sont insérées à gauche suivant l'ordre de tirage
  - si $k > 1$, il y a $(k-1)!$ ordres de tirage possibles, d'où $(k-1)!$ sous-arbres gauches possibles
  - si $k = 1$, l'arbre gauche est vide de nœuds, d'où $1 = (1-1)!$ sous-arbres gauches possibles
- les $n-k$ valeurs de $[\![1..n]\!]$ strictement supérieures à k sont insérées à droite, suivant l'ordre de tirage
  - si $k < n$, il y a $(n-k)!$ ordres de tirages possibles, d'où $(n-k)!$ sous-arbres droits possibles
  - si $k = n$, l'arbre droit est vide de nœuds, d'où $1 = (n-n)!$ sous-arbres droits possibles

1. Etude des sous-arbres gauches, à $k-1$ nœuds :
   - Pour $k=1$, la moyenne de la somme des profondeurs de nœuds des sous-arbres gauches vaut 0
   - Pour $k>1$, chaque élément i de $[\![1..k-1]\!]$ apparaît comme racine d'un sous-arbre gauche avec une probabilité égale à $\dfrac{1}{k-1}$ et contribue à $M(n)$ par
     - la longueur de la racine k de l'arbre à la racine i de son arbre gauche, qui vaut 1,
     - la moyenne (de la moyenne) de la somme des profondeurs de nœuds dans les arbres gauches à $k-1$ nœuds, de sommet i :
     $$\frac{1}{(k-1)!}\sum_{s\in\sigma([\![1..k-1]\!]\setminus\{i\})} M(k-1) = \frac{1}{k-1}M(k-1).$$

   La moyenne de la somme des profondeurs des nœuds appartenant aux sous-arbres gauches à $k-1$ nœuds est donc :

$$M_g(k-1) = \frac{1}{k-1}\sum_{i=1}^{k-1}\left(1+\frac{M(k-1)}{k-1}\right) = \frac{1}{k-1}\big((k-1)+M(k-1)\big) \quad \text{(et 0 si } k=1)$$

2. Etude des sous-arbres droits, à $n-k$ nœuds :
   De même, la moyenne de la somme des profondeurs des nœuds appartenant aux sous-arbres droits à $n-k$ nœuds est :
$$M_d(n-k) = \frac{1}{n-k}\sum_{i=k+1}^{n}\left(1+\frac{M(n-k)}{n-k}\right) = \frac{1}{n-k}\big((n-k)+M(n-k)\big) \quad \text{(et 0 si } k=n)$$

3. Etude de l'arbre (à n nœuds) :
   Pour l'arbre $\mathscr{A}$, la clé k de la racine prenant toute valeur comprise entre 1 et n, de façon équiprobable, on a :

$$\begin{aligned} M(n) &= \frac{1}{n}\sum_{k=1}^{n}\Big((k-1)M_g(k-1)+(n-k)M_d(n-k)\Big) \qquad \text{(barycentre)}\\ &= n-1+\frac{1}{n}\sum_{k=1}^{n}\Big(M(k-1)+M(n-k)\Big) \quad \text{pour } n\geqslant 1, \text{ avec } M(0)=0 \quad \text{(et on a bien } M(1)=0)\end{aligned}$$

   Il ne reste plus qu'à chercher une formule explicite pour M(n) :
   Comme $nM(n) = n(n-1)+2\sum_{k=0}^{n-1}M(k)$ et $(n+1)M(n+1) = (n+1)n+2\sum_{k=0}^{n}M(k)$, par différence, on obtient : $(n+1)M(n+1)-nM(n) = 2n+2M(n)$ ou encore

$$\boxed{(n+1)M(n+1) = (n+2)M(n)+2n.}$$

   On en déduit :

$$\begin{aligned}\frac{M(n+1)}{n+2} &= \frac{M(n)}{n+1}+\frac{4}{n+2}-\frac{2}{n+1} \qquad \text{et, par sommation télescopique (et en revenant en } n)\\ \frac{M(n)}{n+1} &= \frac{M(1)}{2}+4\sum_{k=1}^{n-1}\frac{1}{k+2}-2\sum_{k=1}^{n-1}\frac{1}{k+1}\\ &= \frac{M(1)}{2}+4\left(-\frac{3}{2}+\sum_{k=1}^{n+1}\frac{1}{k}\right)-2\left(-1-\frac{1}{n+1}+\sum_{k=1}^{n+1}\frac{1}{k}\right) = -4+\frac{2}{n+1}+2\sum_{k=1}^{n+1}\frac{1}{k}\end{aligned}$$

   d'où $M(n) = -4(n+1)+2+2(n+1)\sum_{k=1}^{n+1}\frac{1}{k}$ avec $\sum_{k=1}^{n+1}\frac{1}{k}-\ln n \xrightarrow[n\to+\infty]{} \gamma \approx 0.5772156649.$
   On a donc $M(n) \underset{n\to+\infty}{\sim} 2n\ln(n)$ et $P_m(n) = \dfrac{M(n)}{n} \underset{n\to+\infty}{\sim} 2\ln(n).$

6 Insertion et suppression dans un ABR

Il s'agit ici d'insérer ou de supprimer un élément (nœud) dans un ABR, tout en conservant la structure d'ABR.

6.1 Insertion

L'insertion d'un nouveau nœud dans un **ABR** se fait essentiellement selon deux stratégies :

- Insertion en tant que feuille (ce qui est le plus simple).
  L'élément inséré en dernier est donc enfoui profondément dans l'arbre, ce qui peut être
  - un **avantage** lorsque les éléments les moins récemment introduits sont consultés fréquemment,
  - un **inconvénient** si les éléments récemment insérés sont consultés fréquemment.
- Insertion à la racine : le nouvel élément est placé à la racine de l'arbre.
  Les derniers éléments entrés sont donc ceux qui sont accessibles le plus rapidement, avec les avantages et inconvénients contraires de l'insertion en feuille.

6.1.1 Insertion en feuille

Il s'agit simplement de parcourir l'**ABR**, selon la valeur de clé de l'étiquette à insérer, jusqu'à ce que l'on arrive à une feuille que l'on remplace par le nœud à créer :

```
let rec insert_feuille_ABR x = function
   |Vide                                -> Noeud( Vide, x, Vide)
(* |Noeud(g,e,d) as a when cle x = cle e  -> a  (* cas des ABR stricts *) *)
   |Noeud(g,e,d) when cle x <= cle e      -> Noeud(insert_feuille_ABR x g, e, d)
   |Noeud(g,e,d) (* when cle x > cle e *) -> Noeud(g, e, insert_feuille_ABR x d)
;;
(* insert_feuille_ABR : 'a * 'b -> ('a * 'b) arbre -> ('a * 'b) arbre = <fun> *)
```

Dans cette stratégie, pour un **ABR** large, le dernier entré peut être masqué par un élément plus ancien et de même clé, ce qui fait que, pour une même clé, les premiers visibles sont les plus anciens dans l'arbre.

La complexité maximale, en fonction de la hauteur h est $C(h) = h$ unité de coût (comparaison de clé).

6.1.2 Insertion avec repoussement en feuille, dans un ABR large

On peut utiliser une variante de l'insertion en feuille, pour les **ABR** larges, qui inverse l'ancienneté en cas d'égalité de clé :

```
let rec insert_repousse_feuille_ABR x = function                (* pour ABR larges *)
   |Vide                              -> Noeud( Vide, x, Vide)
   |Noeud(g,e,d) when cle x = cle e   -> Noeud(insert_repousse_feuille_ABR e g, x, d)
   |Noeud(g,e,d) when cle x <= cle e  -> Noeud(insert_repousse_feuille_ABR x g, e, d)
   |Noeud(g,e,d)                      -> Noeud(g, e, insert_repousse_feuille_ABR x d)
;;
(* insert_repousse_feuille_ABR : 'a * 'b -> ('a * 'b) arbre -> ('a * 'b) arbre =  <fun> *)

let a = Noeud(        Noeud( Noeud(Vide, 1.0, Vide), 2.0, Vide),
               4.0 ,  Noeud( Vide, 6.0, Noeud(Vide, 7.0, Vide)));;
insert_repousse_feuille_ABR (2.0,1);;
```

Dans cette variante, réservée aux **ABR** larges, en cas d'égalité de clé, l'élément nouveau prend la place de l'ancien et l'ancien est réinséré, selon le même principe, plus profondément, ce qui fait que,

pour une même clé, les premiers visibles sont les plus récents dans l'arbre.

La complexité maximale, en fonction de la hauteur h est $C(h) = h$ unité de coût (comparaison de clé).

6.1.3 Insertion à la racine

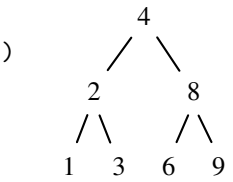
Principe : Pour insérer un élément x de clé c dans l'arbre a, on partage l'arbre a en deux sous-**ABR** :

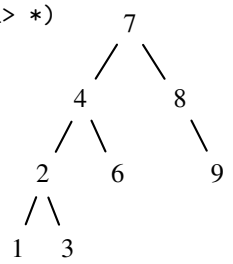
- un **ABR** g , des éléments de a dont la clé est inférieure ou égale à c,
- un **ABR** d , des éléments de a dont la clé est supérieure (strictement) à c,

et on renvoie l'arbre **ABR** (g, x, d).

Selon les cas, **ABR** large ou strict, l'insertion à la racine s'écrit :

```
let rec insert_racine_ABR x t = Noeud(g, x, d) where (g,d) = coupe x t
  where rec coupe x = function
    | Vide                            -> Vide, Vide
    | Noeud(g,e,d) when cle x < cle e -> let g',d' = coupe x g in g', Noeud(d', e, d)
    | Noeud(g,e,d) when cle x > cle e -> let g',d' = coupe x d in Noeud(g, e, g'), d'
(* (4). égalité : cas des ABR stricts *)
    | _  -> failwith "Clé déjà présente."
(* (4). égalité : cas des ABR larges : option 1 (aggrave le déséquilibre) *)
(*  | Noeud(g,e,d) -> Noeud(g, e, Vide), d         *)
(* (4). égalité : cas des ABR larges : option 2 (moins de déséquilibre) *)
(*    | Noeud(g,e,d) -> insert_racine_ABR e g, d   *)
;;
(* insert_racine_ABR : 'a * 'b -> ('a * 'b) arbre -> ('a * 'b) arbre = <fun> *)

let a = Noeud( Noeud( Noeud(Vide, (1,1), Vide), (2,1),
  Noeud(Vide, (3,1), Vide)), (4,1) , Noeud( Noeud(Vide, (6,1), Vide),
 (8,1), Noeud(Vide, (9,1), Vide)));;
insert_racine_ABR 7.0 a;;
```



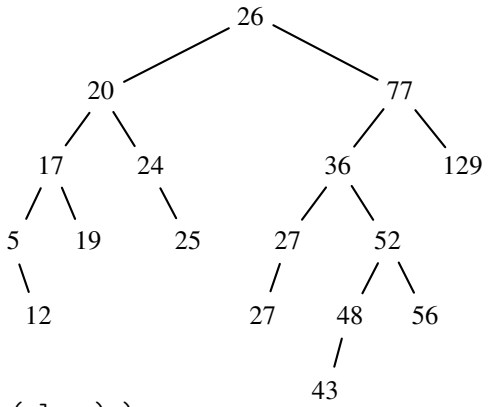


La complexité maximale, en fonction de la hauteur h est $C(h) = h$ unité de coût (comparaison de clé).

6.1.4 Illustration : (mauvais) tri d'une liste, par insertion dans un ABR

Remarque. Tri croissant ou décroissant selon l'ordre générique, et ce n'est pas, en l'état, un tri performant !

```
let rec abr_of_list = function
 | [] -> Vide
 | a::q -> insert_feuille_ABR (a,0) (abr_of_list q)
;;
(* abr_of_list : 'a list -> ('a * int) arbre = <fun> *)

let rec it_gpd_ABR f accu = function
 | Vide         -> accu
 | Noeud(g,e,d) ->
      let accu' = it_gpd_ABR f accu g
      in it_gpd_ABR f (f accu' (cle e)) d
and     it_dpg_ABR f accu = function
 | Vide -> accu
 | Noeud(g,e,d) -> it_dpg_ABR f (f (it_dpg_ABR f accu d) (cle e) ) g
;;
(* it_gpd_ABR : ('a -> 'b -> 'a) -> 'a -> ('b * 'c) arbre -> 'a = <fun> *)
(* it_dpg_ABR : ('a -> 'b -> 'a) -> 'a -> ('b * 'c) arbre -> 'a = <fun> *)

let d = abr_of_list [27;12;43;27;5;25;129; 48; 56; 52; 19;36;77;17;24;20;26];;
it_gpd_ABR (fun x y -> y::x) [] d;;
(* int list = [129; 77; 56; 52; 48; 43; 36; 27; 27; 26; 25; 24; 20; 19; 17; 12; 5] *)
it_dpg_ABR (fun x y -> y::x) [] d;;
(* int list = [5; 12; 17; 19; 20; 24; 25; 26; 27; 27; 36; 43; 48; 52; 56; 77; 129] *)
```



Exercice : Calculer la complexité de ce tri.

6.2 Suppression

6.2.1 Remplacement par un noeud terminal

Pour supprimer le (premier) nœud de clé c dans un **ABR** a =Arbre(g, e, d) :

- si c $<$ cle e, alors on renvoie Arbre(g', e, d) où g' est l'arbre g privé de l'élément de clé c (récursif),
- si c $>$ cle e, alors on renvoie Arbre(g, e, d') où d' est l'arbre d privé de l'élément de clé c (récursif),
- si c $=$ cle e, alors on renvoie l'arbre déduit de a par suppression de la racine de a.

Pour supprimer la racine d'un arbre,

- si l'un des fils est Vide, alors on renvoie l'autre fils (qui peut être Vide),
- sinon, la suppression d'un nœud terminal conservant une structure d'**ABR**, il suffit de remplacer l'étiquette de la racine par celle d'un nœud terminal bien choisi, que l'on supprime de l'arbre :
  - **Stratégie 1** : On supprime l'élément le plus à droite dans le fils gauche (il s'agit d'un plus grand élément du fils gauche) et on remplace l'étiquette de la racine par celle de l'élément supprimé.
  - **Stratégie 2** : On supprime l'élément le plus à gauche dans le fils droit (il s'agit d'un plus petit élément du fils droit) et on remplace l'étiquette de la racine par celle de l'élément supprimé.

  Remarque. Cela se fait sans comparaisons.

Preuve : On démontre facilement, par induction, que l'on obtient bien un **ABR**.

Inconvénient : Dans le cas d'arbres à clés non uniques, cette méthode **bouleverse l'ordre de visibilité**, le nœud terminal utilisé pour le remplacement, qui était dernier dans l'ordre de visibilité pour sa clé, prend la première place. Si cela s'avère problématique, on peut envisager ensuite de repousser la nouvelle racine en dernière position de visibilité, par échange de proche en proche des étiquettes des nœuds de même clé, mais pour que cela soit efficace, il faut s'assurer que les étiquettes contenues dans un fils droit sont toujours strictement supérieures à celle du père (cela pourrait ne pas être vrai si l'arbre a subit des rotations de ré-équilibrage).

Illustration :

```
let rec supprime_ABR c = function
  | Vide                          -> Vide
  | Noeud(g,e,d) when c < cle e   -> Noeud(supprime_ABR c g , e, d)
  | Noeud(g,e,d) when c > cle e   -> Noeud(g, e, supprime_ABR c d)
  | Noeud(g,e,d) as a             -> supprime_racine_ABR a

where rec supprime_racine_ABR = function
  | Vide            -> Vide
  | Noeud(g,_,Vide) -> g
  | Noeud(Vide,_,d) -> d
  | Noeud(g,x,d)    -> let (x',g') = maxi_et_sousABR g
                       in Noeud(g',x',d)                (* remplacement par maxi gauche *)

and       maxi_et_sousABR = function
  | Vide              -> failwith "y a comme un défaut"
  | Noeud(g',x',Vide) -> (x',g')
  | Noeud(g,e,d)      -> let (x',d') = maxi_et_sousABR d in (x', Noeud(g,e,d'))
;;
(* supprime_ABR : 'a -> ('a * 'b) arbre -> ('a * 'b) arbre = <fun*)
```

Pour effectuer le remplacement par le minimum extrait du fils droit, on écrira à la place :

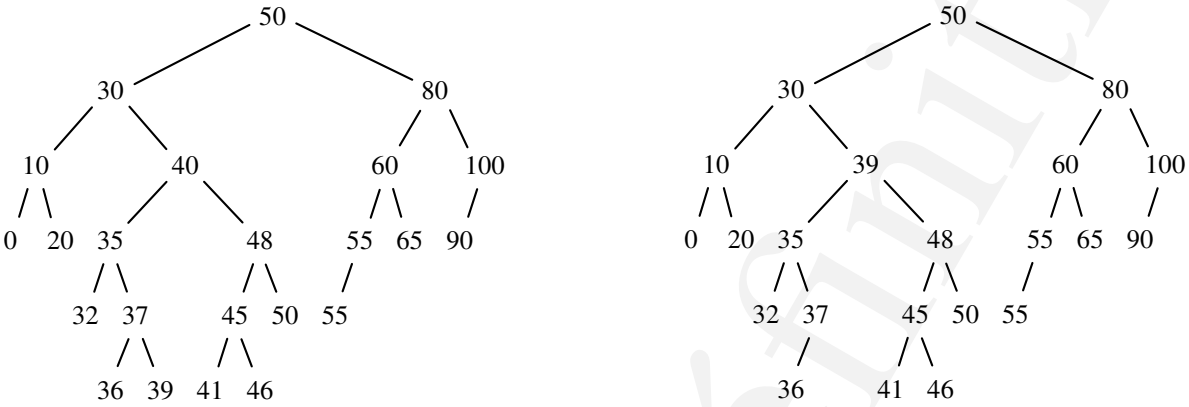
```
   ....
  | Noeud(g,x,d)    -> let (x',d') = mini_et_sousABR d
                       in Noeud(g, x',d')             (* remplacement par mini droite *)

and       mini_et_sousABR = function
  | Vide            -> failwith "y a comme un défaut"
  | Noeud(Vide,x',d') -> (x', d')
  | Noeud(g,e,d)      -> let (x',g') = mini_et_sousABR g in (x', Noeud(g',e,d))
;;
```

Complexité maximale : (le paramètre est la hauteur h et l'unité de coût est la comparaison de clés)

- suppression du minimum : $C_m(0)=0$; $C_m(h)=1+C_m(h-1)$; d'où $C_m(h)=h$
- suppression de la racine (arbre non vide) : $C_r(h)=C_m(h-1)=h-1$
- suppression : $C(0)=0$; $C(h)=1+\max(h-1,C(h-1))$; d'où $C(h)=h$

> La complexité maximale, en fonction de la hauteur h est $C(h)=h$ unité de coût (comparaison de clé).

Exemple : suppression de l'élément de clé 40, remplacé par le maximum de son fils gauche




```
let L = [36;39;65;55;46;90;55;50;37;41;45;48;32;35;20;0;100;60;40;10;80;30;50] ;;
let a = list_it (fun x -> insert_feuille_ABR (x,x) ) L Vide;;
let a1 = supprime_ABR 40 a;;
```

6.2.2 Autres stratégies de suppressions ...

6.3 Défauts de l'insertion et de la suppression

L'insertion et la suppression ont tendance à aggraver le déséquilibre des arbres ABR, ce qui augmente la hauteur, donc le coût des requêtes.
En pratique, on utilise des **ABR** particuliers (**AVL**, **Arbres Rouges et Noirs**, **Arbres déployés**, etc ...) dans lesquels l'insertion et la suppression sont accompagnés de traitements visant à réduire (au moindre coût) le déséquilibre des arbres (ce qui entraîne quand même des coûts supplémentaires pour l'insertion et surtout pour la suppression).
Comme en général les insertions et les suppressions sont beaucoup plus rares que les requêtes, avec des arbres qui maintiennent presque l'équilibre, on arrive à gagner largement en temps moyen de traitement.

7 Complexité (résumé)

7.1 Complexité maximale

Théorème 7.1.

> *Dans un ABR de hauteur h ,*
>
> *les opérations de recherche, insertion, suppression, s'effectuent avec un coût maximal en $O(h)$ comparaisons de clés.*
>
> *les opérations de parcours simples : minimum, maximum, prédécesseur, successeur s'effectuent avec un coût maximal en $O(h)$ appels récursifs, sans comparaisons de clés.*

- Dans le pire des cas (arbre peigne à n nœuds), $h=n$.
- Dans le meilleur des cas (arbre strictement équilibré à n nœuds), $h \underset{+\infty}{\sim} \log_2 n \quad \big(h=O(\log_2 n)\big)$.
- Pour des arbres seulement H-équilibrés à n nœuds, on a aussi $h=O(\log_2 n)$, plus précisément $h \underset{+\infty}{\sim} 2.08\log_2 n$, ce qui fait que ce $O(\log_2 n)$ est moins favorable que celui des arbres strictement équilibrés.

7.2 Complexité moyenne pour l'accès à un nœud

On a également vu que la complexité moyenne d'accès à un nœud dans un ABR quelconque était un $O(\ln n)$, plus précisément équivalente à $2\ln n$, identique à la complexité moyenne d'accès à un nœud dans un arbre binaire quelconque : le choix d'une racine à clé non médiane déséquilibre fortement un ABR.

8 Arbres binaires de recherche équilibrés (information)

Le coût de recherche dans un arbre binaire de recherche (**ABR**) à n nœuds, de hauteur h, étant un $O(h)$, ce coût sera minimal pour un arbre de hauteur minimale.
On cherche donc à avoir des **ABR** qui soient le plus équilibrés possible et pour cela on peut,

- après chaque modification, reconstruire entièrement l'arbre de façon équilibrée, ce qui est très coûteux en temps (éventuellement, on pourrait faire l'équilibrage seulement de façon périodique, mais cela reste coûteux, surtout pour des arbres qui subissent régulièrement de nombreuses modifications),
- soit s'arranger pour que chaque modification garde (à peu près) l'équilibre de l'arbre, sans que cela transforme la totalité de l'arbre (donc au moindre coût).

On choisit bien évidemment la deuxième solution, mais pour qu'elle soit réaliste (au moindre coût), il faut abandonner l'espoir d'un équilibre strict et se contenter d'un "quasi"-équilibre, selon des définitions d'équilibre et des méthodes de traitement spécifiques, où les coûts d'insertion et de suppression avec équilibrage restent de l'ordre d'un $O(\log_2(n))$.

Différentes méthodes ont été développées dans ce but, en particulier :

- Arbres **AVL** (**A**del'son, **V**el'skii, **L**andis, 1962), utilisant le H-équilibre vu précédemment.
  Arbre binaire de recherche H-équilibré (où la différence de hauteur entre deux fils est au plus 1).
  La hauteur h d'un **AVL** à n nœuds est un $O(\log_2(n))$, plus précisément $h \underset{+\infty}{\sim} 2.08\log_2(n)$, ce qui reste raisonnable, sachant que le mieux est $h \underset{+\infty}{\sim} \log_2(n)$ pour un arbre strictement équilibré idéal.
  Dans un **AVL** à n nœuds, la recherche, l'insertion et la suppression (avec minimisation de la hauteur h) se font en $O(h)$, donc en $O(\log_2(n))$.
- **Arbres Rouge et Noir** (Bayer, Sleator, Tarjan).
  Arbre binaire de recherche dont les nœuds sont colorés d'une couleur parmi deux (classiquement Rouge et Noir), et qui doit satisfaire à des contraintes de coloration :
  (A-1) Un nœud est soit blanc, soit noir.
  (A-2) Une feuille est noire (convention)
  (A-3) La racine est noire
  (A-4) Le père d'un nœud blanc est noir.
  (A-5) Tous les chemins partant d'un nœud donné et se terminant à une feuille contiennent le même nombre de nœuds noir (sans prendre en compte le nœud de départ).
  La *hauteur noire* d'un nœud n appartenant à un arbre bicolore (on la note $h_n(n)$) est le nombre de nœuds noirs sur les chemins partant de ce nœud et qui aboutissent à une feuille (sans prendre en compte n). Cette fonction est bien définie grâce à (A-5).
  La hauteur noire ($\mathscr{H}$) d'un arbre bicolore à n nœuds, qui est la hauteur noire de sa racine, est un $O(\log_2(n))$ (à préciser), et les coûts de recherche, d'insertion et de suppression (avec minimisation de la hauteur noire) se font en $O(\mathscr{H})$, donc en $O(\log_2(n))$.
- **Arbres 2-3** (Hopcroft 1970). L'équilibre est maintenu en manipulant le degré des nœuds.
- **B-Arbres** (**B**ayer, McCreight, 1972) généralisation des arbres 2-3.
- **Arbres déployés** (Sleator, Tarjan, 1983).
- ...

Pour maintenir l'équilibre (quasi-équilibre), on fait passer des nœuds d'un fils dans l'autre par des rotations, de façon à réduire la différence de hauteur entre les deux fils. Selon les cas, on aura deux types de rotations : les rotations simples (excédent à l"extérieur" de l'arbre) et les rotations doubles (excédent à l'"intérieur" de l'arbre), vers la gauche (excédent sur le fils droit) ou vers la droite (excédent sur le fils gauche).
Pour connaître la nature du déséquilibre, il faut introduire une information complémentaire δ_i dans les nœuds, par exemple

- la différence de hauteur entre le fils gauche et le fils droit (arbres **AVL**),
- la couleur attribuée au nœud (**Arbres Rouges et Noir**),
-

$< \mathscr{F}\mathscr{I}\mathscr{N} >$