Cours de Programmation MIM1, 2000-2001, Devoir 2
Arbres binaires : ABR et AVL

Laure DANTHONY, Sylvain FOUR

pour le 10 novembre 2000

Commentaires généraux

Dans ce D.M., nous étudions différents types d'arbres et leurs caractéristiques : dans un premier temps, nous introduirons les arbres binaires (chaque nœud a 0, 1 ou 2 fils), ainsi que les arbres binaires de recherche (que nous abrégerons en ABR) : ce sont des arbres binaires dans lesquels chaque étiquette d'un sous-arbre gauche est inférieure à l'étiquette du père, et chaque étiquette du sous-arbre droit est supérieure à l'étiquette du père. Dans un deuxième temps, nous introduirons les AVL, qui sont un cas particulier d'ABR "équilibrés".

Nous utiliserons plusieurs fonctions graphiques adaptées à chaque type d'arbre et à leurs caractéristiques (largeur, profondeur, nombre de nœuds) et visualiserons notamment des fonctions de parcours et de recherche. Enfin, nous confronterons modèle et théorie sur des propriétés classiques des arbres.

Remarque Tous les codes sont regroupés dans une annexe.

1 Affichage Graphique

1.1 Définition des `binary_tree`

1.1.1 Définition

Définissons tout d'abord un type d'arbre de la façon suivante :

```
type binary_tree = Empty_b | Node_b of node_b
     and node_b = {mutable key_b : int ; mutable colour_b : color ;
     mutable left_son_b : binary_tree  ; mutable right_son_b : binary_tree};;
```

Les enregistrements sont plus faciles à gérer que des produits, et les types mutables permettront, lorsque cela sera possible, de travailler en place. Notons que l'on dispose d'un champ `colour_b` qui permettra d'afficher les nœuds en différentes couleurs.

1.1.2 Constructeurs et destructeurs

On dispose des fonctions simples de construction et de destruction suivantes :
`empty_b()is_empty_b, make_b, get_node_b, get_key_b, get_colour_b`
`get_left_son_b, get_right_son_b`.

1.2 Une première version d'outil graphique

1.2.1 Objectifs et idée de l'algorithme

La première routine graphique que nous avons écrite avait pour but de résoudre la plupart des problèmes que rencontraient les applications graphiques usuelles. Ceci nous a permis de mettre au jour certains critères auxquels devait répondre notre programme. Les voici :

– l'ensemble doit être un ensemble relativement symétrique et harmonieux ;

– il faut *absolument* éviter que les arêtes ou les clés se chevauchent.

La solution qui consistait à tracer le graphe "en partant de la racine" a alors été immédiatement rejetée. Elle a certes l'avantage de simplifier l'écriture du programme, mais elle ne laisse aucun contrôle quant au comportement des feuilles : face à un tel outil graphique, on pourra facilement trouver un arbre ayant suffisamment de feuilles à une certaine profondeur pour que sa représentation fasse apparaître des chevauchements.

Nous avons donc choisi de "prendre le problème à l'envers", c'est-à-dire de programmer une fonction graphique qui commence à tracer les feuilles pour terminer à la racine. Puisque le graphe a nécessairement moins de nœuds à chaque étage lorsqu'on remonte vers la racine, cette méthode nous permet de contrôler avec efficacité le comportement de la base de l'arbre sans pour autant délaisser sa tête. Il était alors clair qu'une version récursive était totalement inadaptée et nous avons opté pour une version itérative.

1.2.2 D'un point de vue plus pratique

Comment construire le graphe d'un arbre en partant de ses feuilles ? C'est assez simple pour un arbre complet. Il suffit en effet de construire toutes les feuilles de sa base puis de remonter d'étage en étage en construisant chaque nœud grâce à ses deux nœuds-fils comme le montre la Figure 1 :

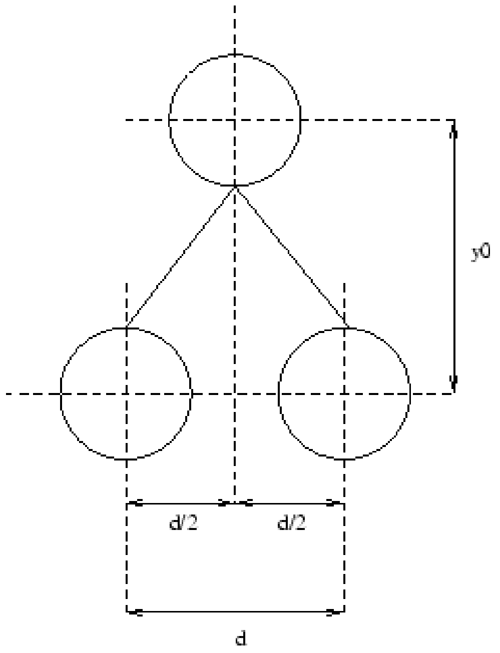


Fig. 1: *Règle de construction d'un père à partir de ses fils*

L'abscisse du père est l'isobarycentre des abscisses de ces fils et quant à son ordonnée, c'est l'ordonnée de ces fils à laquelle on ajoute une constante y_0 que l'on déterminera par la suite.

Le cas d'un arbre non complet pose certains problèmes. En effet, on ne peut, a priori, appliquer la méthode précédente. On pourra certes tracer les branches dont les feuilles sont maximales (c'est-à-dire les feuilles qui se trouvent à la plus grande profondeur de l'arbre). Mais, comment tracer les autres branches ? En d'autres mots, comment choisir les coordonnées de feuilles non maximales étant donné qu'il n'existe aucun nœud tracé à l'étage du dessous pour s'y appuyer ? La solution pour laquelle nous avons opté consiste à se ramener au cas d'un arbre complet (que l'on sait traiter) en "complétant" l'arbre considéré. Il suffit en effet de calculer les coordonnées de tous les nœuds de l'arbre complet construit et on aura ensuite plus qu'à tracer les nœuds qui existent réellement. Le problème des branches aux feuilles non maximales ne se posera plus puisqu'on pourra trouver pour chaque feuille non maximale deux "fils virtuels" sur lesquels on pourra s'appuyer pour calculer les coordonnées souhaitées.

Il se pose encore le problème de l'accès aux feuilles d'un arbre. Notre construction commencera par les feuilles pour ensuite remonter vers la racine en construisant successivement tous les nœuds de chaque étage : il faut donc travailler par étage. La méthode que nous avons choisie est ainsi d'utiliser un "parcours en largeur renversé" de l'arbre placé en argument. En effet, si on renverse la liste des clés obtenue par un parcours en largeur, on obtient successivement les feuilles maximales (rangées de droite à gauche) puis chaque étage (avec les nœuds apparaissant de la droite vers la gauche).

Cette méthode a l'avantage de simplifier la completion de l'arbre. En effet, pour faire un parcours en largeur préfixe, on utilise une pile (cf la section 2.2). A chaque fois que l'on dépile, on parcourt la clé de l'arbre dépilé et on empile ses sous-arbres. Pour compléter un arbre, il suffit alors d'ajouter l'instruction suivante : dès qu'un fils est un arbre vide, on n'empile pas l'arbre vide mais un "arbre fantôme" auquel on donne arbitrairement la clé -1 (clé non utilisée par un arbre standard) et dont les deux fils sont `Empty_b`. Bien-sûr, on ne peut appliquer cette instruction pour chaque nœud sinon on construirait un arbre infini. Mais on connaît le moment où il faut s'arrêter puisque l'on connaît le nombre total de nœuds d'un arbre complet.

Remarque : Au lieu de créer la liste associée au parcours en largeur préfixe de l'arbre complet virtuel puis de la renverser, on ajoute chaque clé de la liste à sa tête si bien qu'on obtient le résultat souhaité sans à avoir à renverser une liste.

1.2.3 Calcul effectif des coordonnées et impression des nœuds

Dans notre approche itérative du problème, les coordonnées des différents nœuds (qui sont en fait les coordonnées des centres des cercles qui contiennent ces nœuds) sont stockées à l'intérieur d'un tableau de taille le nombre total de nœuds d'un arbre binaire complet. Le tableau a en effet l'avantage de nous donner facilement accès à son i-ème élément, opération qui nous est indispensable pour calculer les coordonnées d'un père grâce à celles de ses fils (virtuels ou non).

Le remplissage de ce tableau se fait en deux étapes :

– dans un premier temps, on calcule les coordonnées des feuilles maximales (virtuelles ou non) tout à fait simplement. Leurs abscisses sont calculées par les règles suivantes :

  1. deux feuilles voisines ayant le même père sont séparées par une distance `d` que l'on définira par la suite ;
  2. deux feuilles voisines n'ayant pas le même père sont séparées par la distance $2d$.

  Les feuilles sont bien-sûr placées au bas de l'écran graphique. Après le calcul des coordonnées de chaque feuille, on teste si la feuille est réellement présente dans l'arbre en vérifiant sa clé et si c'est le cas on trace un cercle aux coordonnées calculées, on y inscrit la clé et on le colore.

– dans un deuxième temps, on calcule les coordonnées de chaque nœud à partir de celles de ses fils (si l'on souhaite remplir les coordonnées du i-ème élément du tableau, on se sert de celles de son fils gauche et de son fils droit se trouvant respectivement à la $2i$-ème et $(2i+1)$-ème position du tableau) avec les règles que l'on s'est donné précédemment. Après avoir tracé le cercle correspondant et son contenu, il reste à tracer les arêtes qui relient le nœud à ses fils. Ici apparaît la nécessité de garder en mémoire dans le tableau l'existence ou non de chaque nœud. En effet, puisque l'on "détruit" le premier élément de la liste de parcours préfixe à chaque boucle, le tableau est le seul support adéquat. C'est pourquoi on a choisi de mettre les éléments du tableau au type `coord` défini par :

```
type coord = {mutable x : int ; mutable y : int ; mutable exist : bool}.
```

Le champ `exist` est initialisé à `true` si le nœud est bien réel, à `false` sinon. Ainsi, si le champ `exist` d'un des fils vaut `true`, on trace une arête du père vers ce fils ...

1.2.4 Choix des paramètres

Comme nous avons pu le voir précédemment, il nous faut initialiser de nombreux paramètres. Tout d'abord paramétrons la fonction `make_circle` prenant respectivement en arguments :

- l'abscisse `x1` du centre du cercle ;
- l'ordonnée `y1` du centre du cercle ;
- la couleur `c` du nœud à imprimer ;
- le rayon `r` du cercle ;
- la clé `n` du nœud.

L'écriture de cette fonction est très simple : on effectue les opérations suivantes :

1. on trace un disque de couleur `c` et de rayon `r` aux coordonnées (`x1`, `y1`) ;
2. on le borde d'un cercle noir (dans cet ordre-ci pour ne pas que la couleur "morde" sur le cercle) ;
3. on affiche la clé `n`. Pour cela, on déplace le point courant de sorte que l'entier affiché se trouve centré horizontalement et verticalement dans le cercle (on se sert de l'outil `text_size` qui renvoie une paire d'entiers représentant les dimensions de la chaîne de caractères placée en argument).

En ce qui concerne le paramètre `d` défini en 1.2.3., on a choisi de le prendre égal à 4 pixels ou à moins si l'impression dépasse la fenêtre graphique en abscisse. De même, on a choisi de prendre `y` égal à `6*r_int` ou à moins si l'impression dépasse la fenêtre graphique en ordonnée.

Il se pose enfin le problème du choix du rayon des cercles, rayon qui sera (par souci d'harmonie) commun à tous les nœuds de l'arbre. Il nous paraîssait naturel de paramétrer `r_int` (`int` pour "integer") en fonction du nombre maximal de chiffres des nombres figurant dans les clés. C'est la raison pour laquelle on définit `max_number_of_digits` à partir de la liste des clés en appliquant successivement (grâce aux outils `map` et `List.fold_left`) les fonctions `string_of_int` (afin d'avoir une liste de chaînes de caractères), `fst text_size` (afin d'obtenir la liste du nombre de chiffres de chaque clé) et enfin `fst text_size`.

Remarque : En fait le "`-`" d'un nombre négatif doit aussi compter pour un chiffre. Ceci est assuré par le passage aux chaînes de caractères et l'application de `fst text_size`. Cette possibilité a aussi l'avantage de s'adapter à un changement de police par défaut.

Le rayon `r_int` est alors calculé à partir de `max_number_of_digits` par de simples calculs géométriques (la propriété géométrique étant que le rectangle virtuel bordant la chaîne de caractères imprimées soit inscrit dans le cercle de rayon `r_int`).

1.2.5 Résultats et interprétation

La Figure 2 illustre ce que retourne la fonction graphique précédente sur un arbre très petit. Ce résultat nous satisfait entièrement.

La Figure 3 illustre quant à elle ce que retourne la fonction pour un arbre très profond. Ce résultat est moins satisfaisant. Certes, le test répond à toutes nos attentes, mais il se profile ici le problème d'une impression trop large. En fait, la grande profondeur de l'arbre donné en argument se propage dans la grande largeur de l'arbre imprimé. La Figure 4 illustre ces débordements.

Remarque Même en réduisant au maximum le paramètre `d` des feuilles maximales (qui sont à l'extérieur de la fenêtre graphique), l'impression sort inexorablement de la fenêtre. Afin d'améliorer ceci, on réalise une deuxième version qui a pour but de minimiser la largeur de l'impression tout en conservant une certaine esthétique.

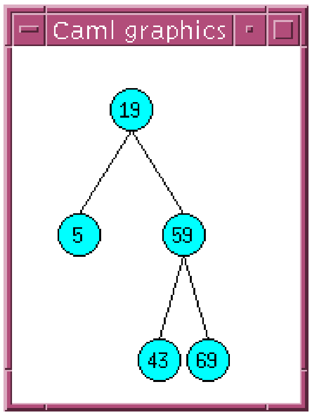


FIG. 2: *Test de la première version sur un arbre très petit*

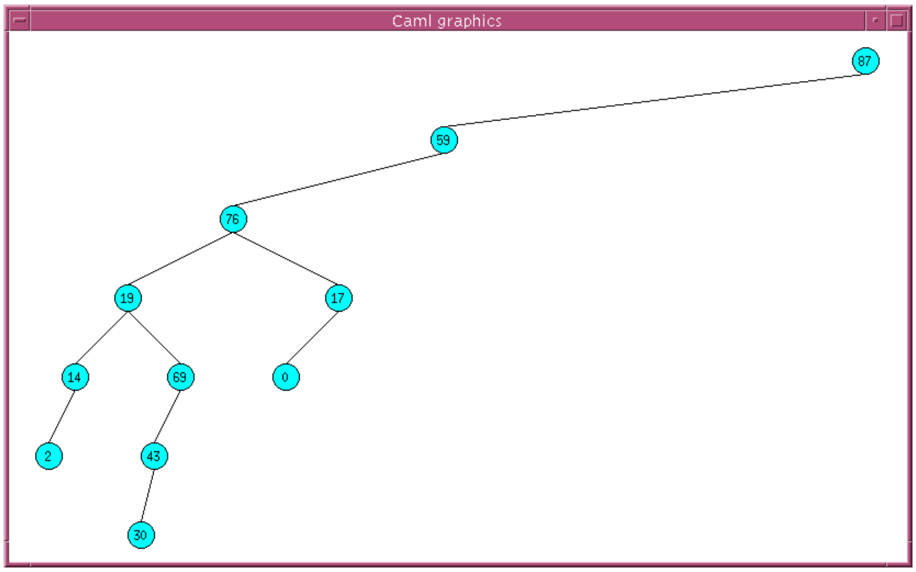


FIG. 3: *Test de la première version sur un arbre très profond*

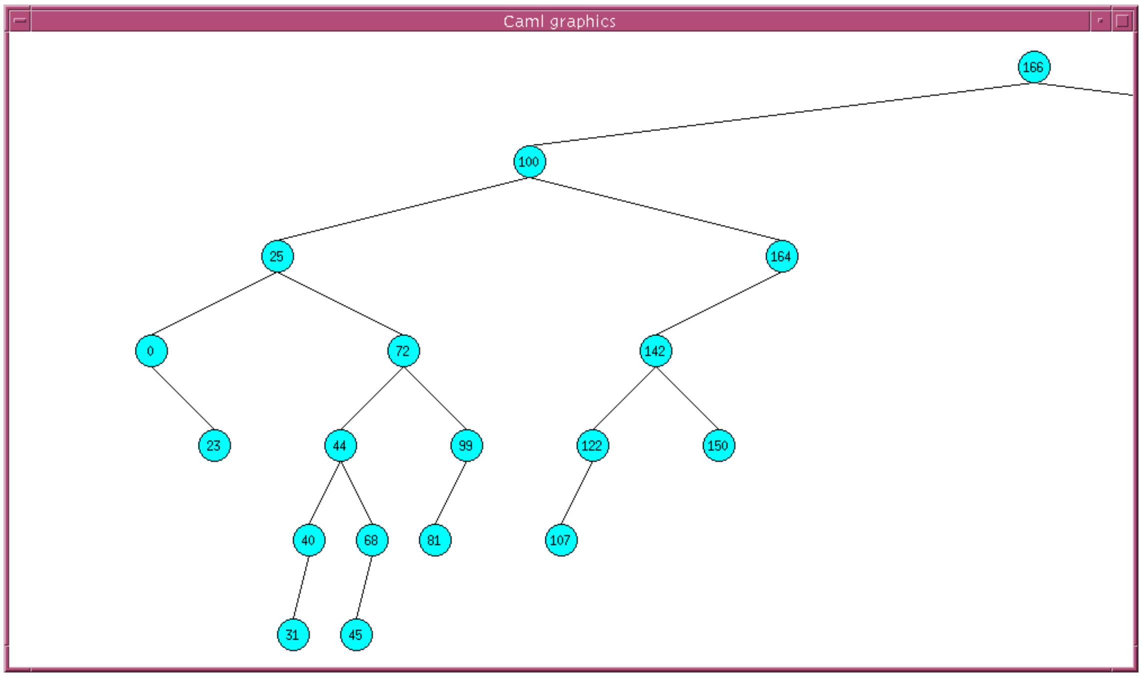


FIG. 4: *Test de la première version sur un arbre très large*

1.3 Une deuxième version graphique

1.3.1 Objectifs et idée de l'algorithme

L'objectif premier de cette deuxième version sera de "concaténer" les deux fils d'un père lorsque cela sera possible. Etudions justement les cas dans lesquelles cette concaténation ne risque pas de créer des chevauchements. Les Figures 5, 6, 7, 8 illustrent les différents cas pour lesquels l'algorithme "collera les deux fils".

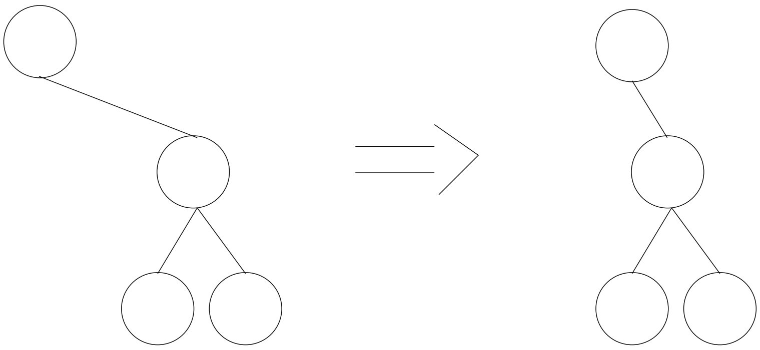


FIG. 5: *Cas où le fils gauche est vide*

Ensuite, comment minimiser la largeur du graphe dans les autres cas ? La solution pour laquelle nous avons opté consiste à construire récursivement l'arbre. A partir d'un nœud, on construit récursivement ses fils puis, suivant la "largeur" des deux sous-arbres construits, on adapte la distance entre les deux racines des deux sous-arbres. En ce qui concerne les "paramètres géométriques" de la construction, on a préféré garder ceux de l'ancienne version puisqu'ils remplissaient très bien leur tâche. On se servira ainsi des mêmes `make_circle`, `d`, `y`, `r_int`...

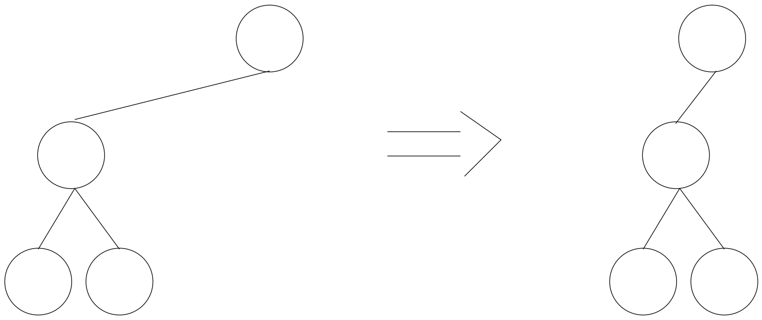


Fig. 6: *Cas où le fils droit est vide*

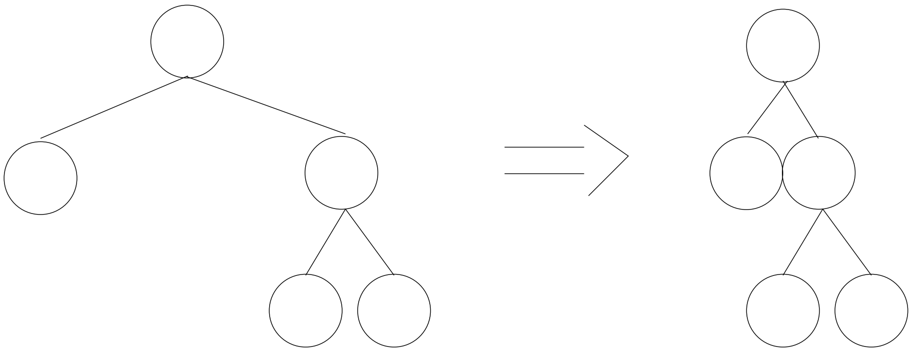


Fig. 7: *Cas où le fils gauche n'a pas de fils*

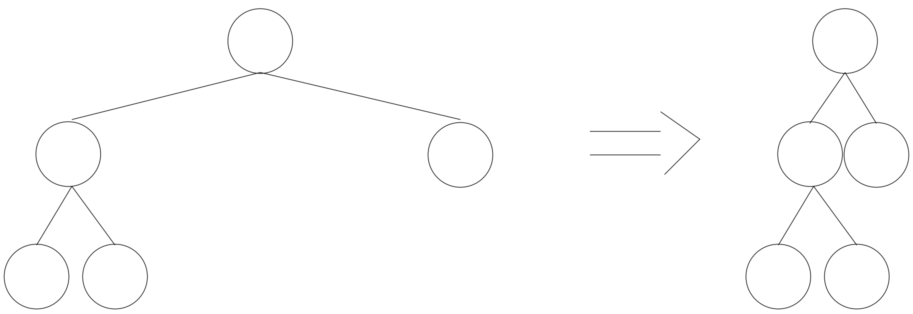


Fig. 8: *Cas où le fils droit n'a pas de fils*

1.3.2 D'un point de vue plus pratique

En fait, lorsqu'on a construit récursivement les deux fils, on a juste besoin de connaître la "largeur gauche" du fils droit imprimé ainsi que la "largeur droite" du fils gauche imprimé. On définit la "largeur gauche" (respectivement "largeur droite") d'un arbre imprimé comme la différence positive entre l'abscisse du centre du cercle représentant le nœud le plus à gauche (respectivement le plus à droite) et celle du centre du cercle représentant la racine.

Ces deux fonctions ont été implémentées sous les noms respectifs de `left_length` et `right_length`. Elles prennent en argument un arbre binaire et renvoie un entier. Expliquons par exemple l'écriture de `left_length`, celle de `right_length` s'en déduisant immédiatement. `left_length` est bien-entendu définie récursivement par :

- si l'arbre placé en argument est vide, on renvoie 0 ;
- sinon si l'arbre placé en argument n'a pas de fils, même conclusion ;
- sinon si l'arbre placé en argument a un fils droit mais pas de fils gauche, le résultat est le maximum de 0 et de la largeur gauche du fils droit auquel on soustrait la différence positive D d'abscisse entre la racine de l'arbre et le nœud de ce fils droit comme le montre la Figure 9.

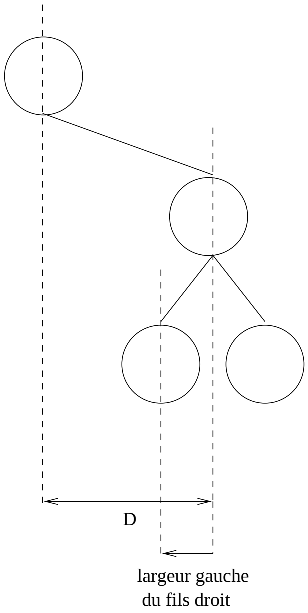


FIG. 9: *Cas où le père a un fils droit mais pas de fils gauche*

- sinon si l'arbre placé en argument a un fils gauche mais pas de fils droit, le résultat est la largeur gauche du fils gauche auquel on ajoute la différence positive D d'abscisse entre la racine de l'arbre et le nœud de ce fils gauche comme le montre la Figure 10.
- sinon l'arbre placé en argument possède deux fils. Le résultat est alors le maximum de :
  - 0 ;
  - la largeur gauche du fils droit auquel on soustrait la différence positive D d'abscisse entre la racine de l'arbre et le nœud de ce fils droit ;
  - la largeur gauche du fils gauche auquel on ajoute la différence positive D d'abscisse entre la racine de l'arbre et le nœud de ce fils gauche.

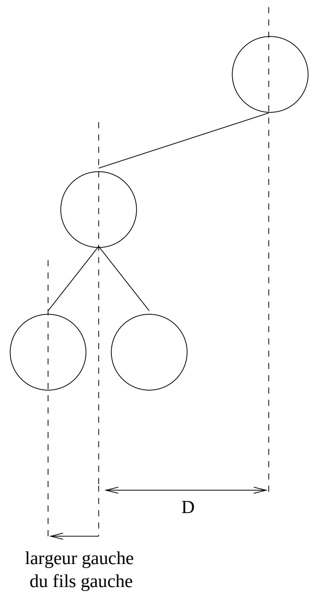


FIG. 10: *Cas où le père a un fils gauche mais pas de fils droit*

1.3.3 Arbres de coordonnées

On choisit ici de modéliser les coordonnées des arbres manipulés récursivement comme pour la construction. C'est pourquoi on introduit un nouveau type intitulé `coordinates_tree`. Il est très semblable au type `binary_tree` excepté le fait qu'on lui a ajouté deux nouveaux champs : l'abscisse `x` et l'ordonnée `y` du centre du cercle représentant le nœud. On crée, de la même manière que pour un type arbre ordinaire, des fonctions de manipulation `empty_c`, `is_empty_c`, `make_c`, `get_x`, `get_key_c`, `get_left_son_c` ("c" pour "coordinates" est utile pour éviter les conflits avec le type `binary_tree`) ...

Le problème de l'affichage d'un arbre revient donc à associer à un arbre de type `binary_tree` un arbre "isomorphe" de type `coordinates_tree`. La fonction qui aura pour but cette opération a pour nom `make_tr_c`. Elle crée l'arbre associé par récurrence structurelle. Notons `tr` l'arbre donné en argument.

- Si `tr` est l'arbre vide, on renvoie `empty_c ()`
- Sinon si `tr` n'a pas de fils, on renvoie un arbre de coordonnées ne comportant qu'un seul nœud, ayant les mêmes propriétés que `tr` (clé, couleur) et on lui affecte les coordonnées d'un point situé en bas à gauche de la fenêtre tel que le cercle n'en touche pas le bord.
- Sinon si `tr` a un fils droit mais pas de fils gauche, on appelle récursivement `make_tr_c` sur ce fils droit. On renvoie un arbre dont le père a les mêmes caractéristiques que la racine de `tr`, ses coordonnées sont explicitées par la Figure 11 suivante et il a bien-sûr un fils droit qui n'est autre que `make_tr_c (get_right_son_b tr)`. Notons qu'il n'y a pas besoin de déplacer `make_tr_c (get_right_son_b tr)` pour créer `make_tr_c tr`.
- Sinon si `tr` a un fils gauche mais pas de fils droit, on appelle récursivement `make_tr_c` sur ce fils gauche. On renvoie un arbre dont le père a les mêmes caractéristiques que la racine de `tr`, ses coordonnées sont explicitées par la Figure 12 et il a bien-sûr un fils gauche qui n'est autre que `make_tr_c (get_left_son_b tr)`. Cette fois-ci, si les abscisses des coordonnées de `make_tr_c (get_left_son_b tr)` sont trop faibles, on n'a pas la place dans la fenêtre graphique pour y ajouter le père. Il suffit alors d'augmenter suffisament les abscisses de ce sous-arbre. On va donc créer, à l'extérieur de la fonction `make_tr_c`, une fonction `push_right` qui prend en argument un arbre

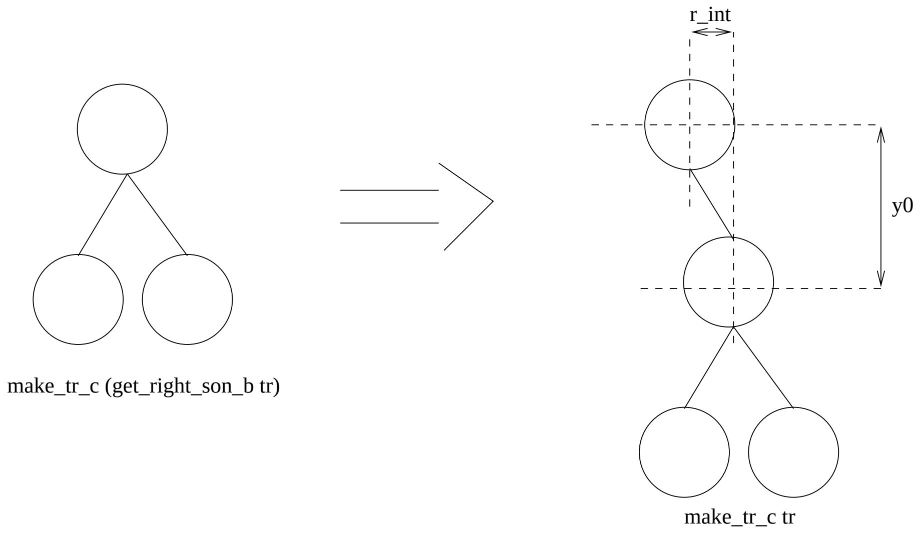


FIG. 11: *Cas où le père n'a pas de fils gauche*

de coordonnées et un entier et qui modifie cet arbre en ajoutant à toutes ses abscisses l'entier placé en argument.

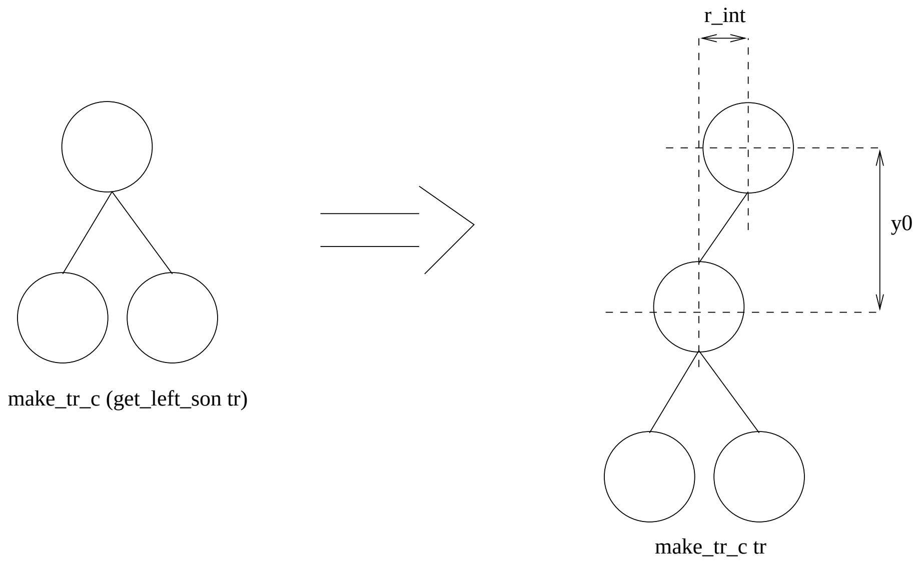


FIG. 12: *Cas où le père n'a pas de fils droit*

– Sinon si `tr` a un fils gauche et un fils droit, on appelle récursivement `make_tr_c` sur ses fils gauche et droit et on obtient respectivement `tr_c_left` et `tr_c_right`.

  La première opération va consister à modifier les ordonnées d'un des deux arbres si leurs racines n'ont pas la même ordonnée. On a donc créé, à l'extérieur de la fonction `make_tr_c`, une fonction `push_up` qui prend en argument un arbre de coordonnées et un entier et qui modifie cet arbre en ajoutant à toutes ses ordonnées l'entier placé en argument.

Ensuite deux cas se présentent.

– Soit on est dans le cas où on peut concaténer les deux cercles représentant les deux racines des sous-arbres (c'est-à-dire le cas où l'un des deux fils n'a pas de fils : cf Figures 7 et 8). Alors, on applique la fonction `push_right` sur l'arbre de droite `tr_c_right` avec la distance adéquate permettant la concaténation des deux cercles.

  Mais, dans le cas où c'est le fils gauche `tr_c_left` qui n'a pas de fils, il ne faut pas, lorsqu'on rapproche le fils droit `tr_c_right`, qu'une partie de son corps sorte de la fenêtre graphique comme sur la Figure 13

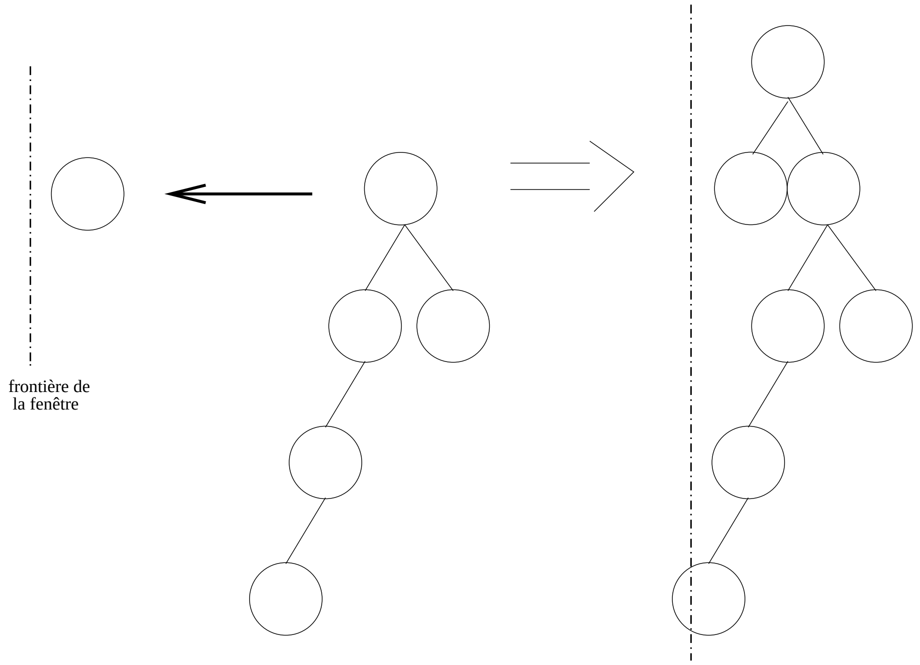


Fig. 13: *Attention au débordement possible !*

  La solution est simple. On calcule quand même l'arbre résultat qui devrait sortir de la fenêtre (en ajoutant un nœud père dont les coordonnées sont calculées d'après les règles de la figure 1) puis on lui applique la fonction `push_right` avec pour second argument une distance suffisante (cette dernière fait intervenir la largeur gauche du fils droit). Enfin, on retourne ce résultat.

– Soit on est dans le cas où il faut évaluer les "largeurs droites et gauches". En fait on a seulement besoin de la "largeur droite" du fils gauche `tr_c_left`. En effet, le problème des ordonnées ayant été réglé, il suffit de déplacer les abscisses de `tr_c_left` d'une distance suffisante vers la droite - distance dépendant uniquement des coordonnées de `tr_c_right` et de sa "largeur droite" comme le montre la Figure 14. Puis on a plus qu'à ajouter le père grâce à la règle de la Figure 1.

Une fois que l'arbre de coordonnées a été programmé, ce n'est plus qu'un jeu d'enfant pour le tracer (on ne détaillera donc pas la procédure de tracé).

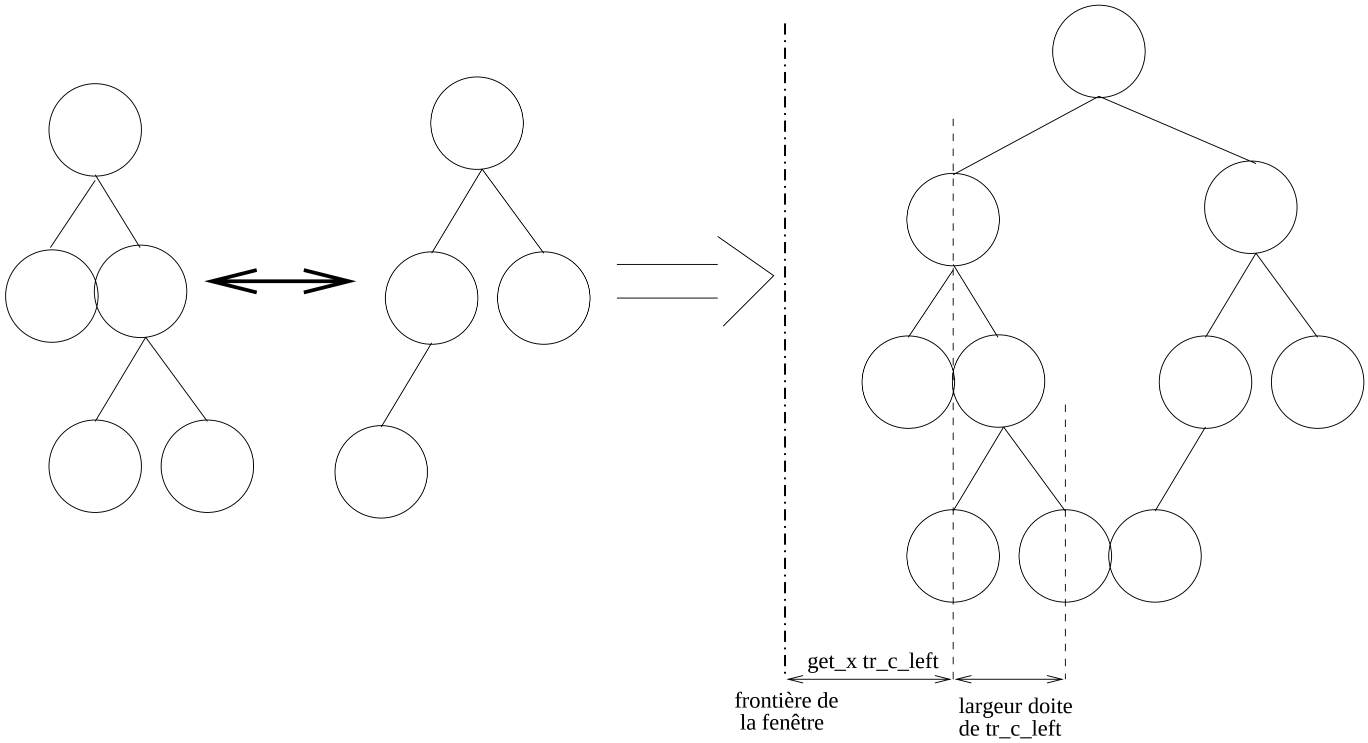


FIG. 14: *On utilise seulement la largeur droite*

1.3.4 Résultats et interprétation

Les Figures 15, 16, 17 montrent ce que retourne la fonction graphique précédente sur les mêmes arbres que ceux des tests du paragraphe 1.2 (on a pris soin de respecter la taille des fenêtres graphiques).

Les résultats sont époustouflants. La largeur gagnée pour les deux derniers tests est impressionnante, d'autant plus que l'on a gardé une certaine symétrie entre fils gauches et fils droits.

Un dernier petit test pour se faire plaisir a été effectué (voir la Figure 18) : nos objectifs sont pleinement atteints.

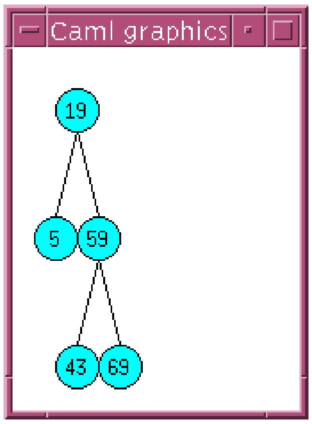


FIG. 15: *Test de la deuxième version sur un arbre très petit*

FIG. 16: *Test de la deuxième version sur un arbre très profond*

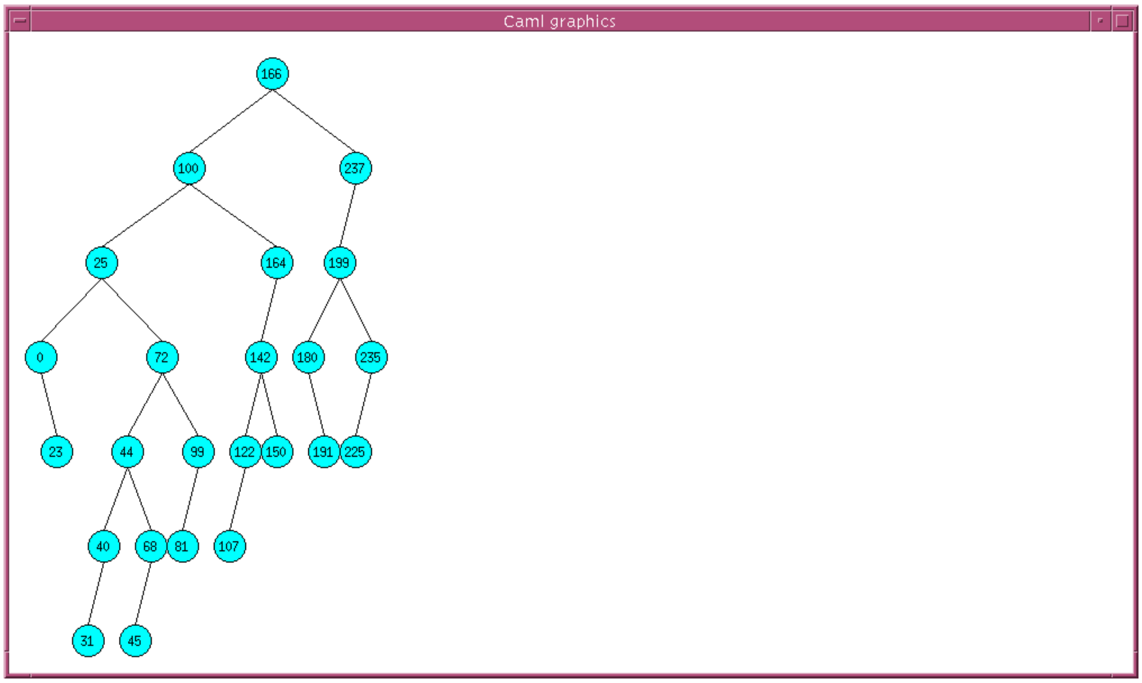


FIG. 17: *Test de la deuxième version sur un arbre très large*

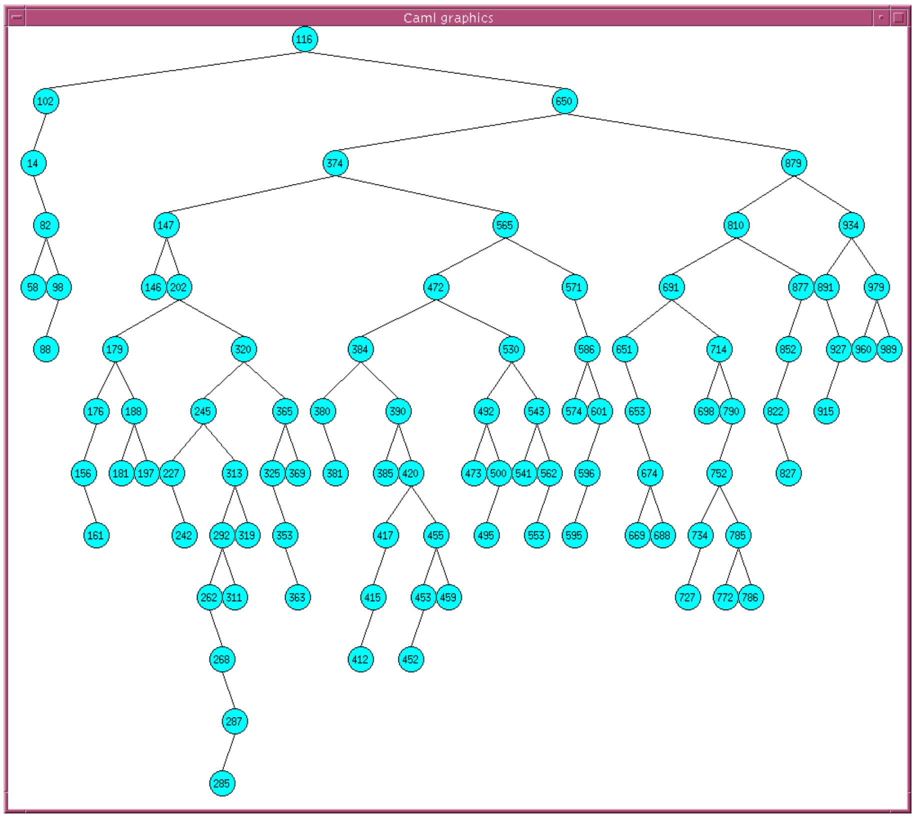


FIG. 18: *Test de la deuxième version sur un arbre à 101 feuilles*

1.3.5 Remarque finale

Les esprits critiques remarqueront que cette deuxième version n'est pas des plus esthététiques pour des arbres de petite taille. Pour répondre à cette dernière objection, on a créé une fonction `draw_tree` qui imprime un arbre par la première version si sa représentation entre dans la fenêtre graphique et qui utilise la deuxième version sinon (grâce à une exception lancée par la première version).

2 Parcours d'arbres

2.1 Parcours en profondeur préfixe

La récursivité Caml est utilisée afin d'imprimer les étiquettes dans l'ordre préfixe : l'étiquette du nœud, puis les étiquettes du fils gauche, puis les étiquettes du fils droit. Pour la visualisation graphique, on décide d'afficher à chaque coloration l'arbre entier, même si cela "clignote" un peu, au lieu de recolorer uniquement le nœud visité. La fonction a le typage suivant :

```
deep_first_graphic : binary_tree -> unit = <fun>
```

2.2 Parcours en largeur préfixe

On utilise la librairie de file (FIFO) donnée en annexe (on dispose des fonctions usuelles suivantes : `make_empty_queue, push, pop`). L'idée est la suivante : on imprime l'étiquette du nœud, et on empile le sous-arbre gauche, puis le sous-arbre droit, ensuite on dépile un arbre, dont on imprime l'étiquette du nœud, on empile les deux sous-arbres et ainsi de suite ... Là aussi, on choisit de redessiner à chaque fois l'arbre entier, en temporisant pour que le parcours visuel ne s'effectue pas trop vite. La fonction a le typage suivant :

```
bread_first_graphic : binary_tree -> unit = <fun>
```

3 Recherche d'une clef dans un ABR

3.1 Principe

La recherche dans un ABR utilise sa spécificité : si la clef cherchée est inférieure à la clef de la racine, on recherche récursivement dans le sous-arbre gauche ; si elle est supérieure à la clef de la racine, on recherche récursivement dans le sous-arbre droit. Ici, afin de visualiser graphiquement cette recherche, on marque les nœuds visités en vert, et on lève une exception lorsque la clef recherchée est trouvée, qui a pour effet de colorier le nœud trouvé en rouge. Cette fonction de recherche fait appel à la fonction auxilliaire :

```
search_bis : int -> binary_tree -> color-> unit = <fun>
```

(appelée sur `green`) qui a juste pour effet de colorier les nœuds visités en vert, et de lever une exception `Found(node)` si la clef cherchée est trouvée. Ensuite, la fonction :

```
search : int -> binary_tree -> color -> color -> unit = <fun>
```

appelée sur `green red` marque le nœud en rouge si l'exception est levée, et affiche le résultat dans la fenêtre graphique.

Remarque Ici se pose un problème de coloriage : la fonction de recherche marque certains nœuds en vert ou en rouge, mais l'arbre retourné a subi ces modifications. L'appel de la même fonction avec la couleur `blue` pour le marquage des n
oe uds visités et aussi pour l'éventuel n
oe ud trouvé permet de résoudre ce (petit) problème.

3.2 Résultats

On choisit dans cette section de compter le nombre de nœuds visités lors de la recherche dans un ABR. On ajoute tout d'abord un compteur du nombre de nœuds visités à la fonction de recherche dont le typage devient :

```
search : int -> binary_tree -> color -> color -> int = <fun>
```

On code aussi une fonction qui génère automatiquement des ABR (`big_brt`) par insertions successives (à l'aide de la fonction `brt_insertion`).

Passons aux tests :

- La fonction `stat_brt` de type

  ```
  stat_brt : int -> int -> float = <fun>
  ```

  crée nt ABR de nombre de nœuds nn et fait la moyenne de leurs hauteurs.

- De même, la fonction de test `make_n_search_brt` de type :

  ```
  make_n_search_brt : int -> int -> float = <fun>
  ```

  permet de calculer la profondeur "expérimentale" d'un nœud dans un ABR (on crée nt ABR à nn nœuds, et l'on recherche 100 clefs quelconques à l'intérieur de chacun de ces ABR)

Voici quelques résultats obtenus en évaluant la hauteur $h(n)$ et la profondeur $p(n)$ d'une clef quelconque d'un ABR à n nœuds avec le maximum de tests possibles (1000 lorsque cela est possible):

n	h(n)	p(n)
100	12,3	8,1
1000	21,1	12,6
10000	29,9	17,8
100000	35,0	22,0

Conclusion La hauteur moyenne et la profondeur moyenne d'un nœud dans un ABR sont "expérimentalement" logarithmiques en n (comme le prévoit la littérature), même si au pire la hauteur d'un ABR peut être linéaire en son nombre de nœuds (cas d'un peigne).

4 Définition des `AVL`

4.1 Définition Caml des AVL

La définition Caml des AVL reprend celle des `binary_tree`, en rajoutant une information supplémentaire dans le champ `node.delta`, celle du déséquilibre du nœud considéré, c'est à dire la différence de taille entre son fils gauche et son fils droit :

```
node.delta = height (node.left_son) - height (node.right_son)
```

On obtient donc :

```
type avl = Empty | Node of node
      and node = {mutable delta : int ; mutable key : int ; mutable colour : color ;
      mutable left_son : avl ; mutable right_son : avl};;
```

4.2 Fonctions de base, constructeurs et destructeurs

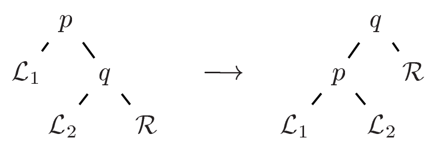
On dispose des fonctions de construction : `make_empty_avl()`,`make_avl`, de destruction : `get_node, get_key, get_delta` ... et enfin de la fonction `is_empty`.

5 Insertion d'une clef dans un AVL

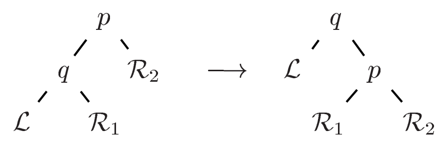
5.1 Fonctions auxilliaires

Afin de pouvoir modifier le "déséquilibre" d'un arbre, on introduit, lorsqu'elles sont possibles, les rotations suivantes :

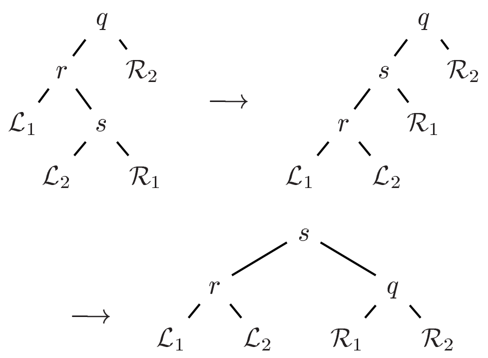
5.1.1 Rotation gauche



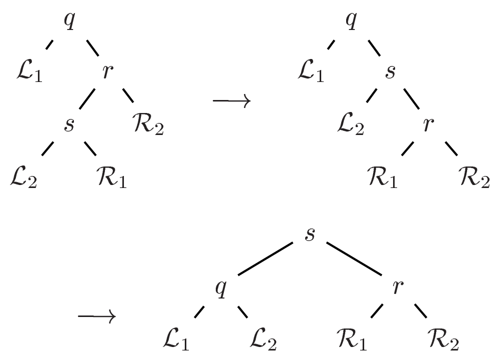
5.1.2 Rotation droite



5.1.3 Rotation gauche-droite



5.1.4 Rotation droite-gauche



Remarque : Une induction structurelle permet de montrer facilement que les opérations de rotations préservent les propriétés des arbres binaires de recherche (en remarquant qu'une caractérisation des ABR est que leur parcours infixe gauche-droite est croissant : après la rotation ce parcours est inchangé ...)

Les codes de ces rotations :

`avl_left_rotation, avl_right_rotation, avl_left_right_rotation,avl_right_left_rotation`

sont donnés en annexe. Notons qu'elles ne sont pas spécifiques aux AVL, aucune information concernant les déséquilibrages n'étant utilisée (les nouvelles valeurs de `delta` seront mises à jour par la fonction `avl_insertion`)

5.2 Idées de l'algorithme

Ajouter une clef dans un AVL, c'est avant tout ajouter une clef dans un ABR. On insère donc récursivement une feuille dans le sous-arbre gauche ou le sous-arbre droit suivant la valeur de la clef. Le problème est qu'alors le déséquilibre peut avoir varié, l'arbre peut donc ne plus être un AVL. La solution réside donc en des rotations, qui, comme on peut le voir sur les schémas, modifient le déséquilibre (et aussi quelquefois la hauteur !). On s'aperçoit rapidement qu'il faut véhiculer l'information "la hauteur du sous-arbre dans lequel on a inséré la clef a été augmentée" (`true`) ou "n'a pas été augmentée" (`false`).

5.3 Différents cas à traiter

Considérons l'arbre $\mathcal{T}(k, t', t'')$ et notons t_1 le sous-arbre résultat de l'insertion récursive de la clef à insérer dans le sous-arbre gauche t' (les cas d'insertions dans le sous-arbre droit sont symétriques) : la fonction auxilliaire `insert2` a renvoyé le couple $(t_1, bool)$ où bool est un booléen.

- si $bool = false$ (c'est à dire que la hauteur de t_1 est la même que celle de t'), alors on modifie $\mathcal{T}$ qui devient (k, t_1, t''), et on renvoie $(\mathcal{T}, false)$
- si $bool = true$ (la hauteur du sous-arbre gauche a été modifiée : la hauteur de t_1 est supérieure d'une unité à celle de t'). Alors deux cas sont possibles :
  - cas $\delta(\mathcal{T}) = 0$ ou -1. Comme la hauteur gauche a augmenté, la valeur de delta varie aussi : $\delta \leftarrow \delta+1$, et le nouvel arbre $\mathcal{T}(k, t_1, t'')$ est bien encore un AVL. On renvoie donc le couple $(\mathcal{T}, true)$ si le "nouveau" $\delta(\mathcal{T})$ est égal à 1 (la hauteur gauche a alors augmenté, i.e. $h(\mathcal{T}') = h(\mathcal{T}) + 1$), et $(\mathcal{T}, false)$ sinon.
  - cas $\delta(\mathcal{T}) = 1$ L'arbre (k, t_1, t'') n'est alors plus un AVL, son déséquilibre étant 2. Les rotations à effectuer dépendent alors des caractéristiques du sous-arbre t_1 :
    - si $\delta(t_1) = 1$, alors on se trouve dans la situation de la Figure 19 où l'on effectue une rotation droite.Il s'agit ensuite de modifier les déséquilibres : $\delta(a) = \delta(b) = 0$ (on remarque ici que la hauteur de l'arbre n'a finalement pas été modifiée).

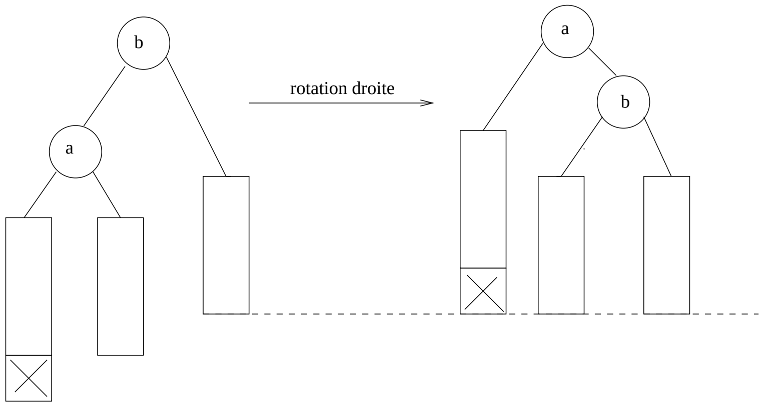


FIG. 19: *Cas où l'on effectue une rotation droite*

– si $\delta(t_1) = -1$ on est dans l'un des 2 cas des Figures 20 et 21 où l'on effectue une rotation gauche-droite.Traitons les cas de la Figure 20, pour l'insertion (a) : $\delta(a) = 0$, $\delta(c) = -1$, $\delta(b) = 0$ et pour l'insertion (b) : $\delta(a) = 0$, $\delta(c) = -1$, $\delta(b) = 0$. Dans les deux cas, la hauteur de l'arbre n'a finalement pas été modifiée. Pour le cas de la Figure 21, $\delta(a) = \delta(b) = \delta(c) = 0$

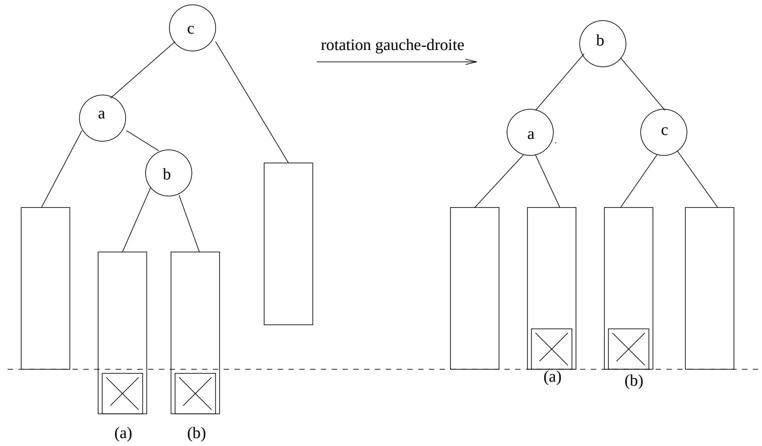


FIG. 20: *Cas où l'on effectue une rotation gauche-droite*

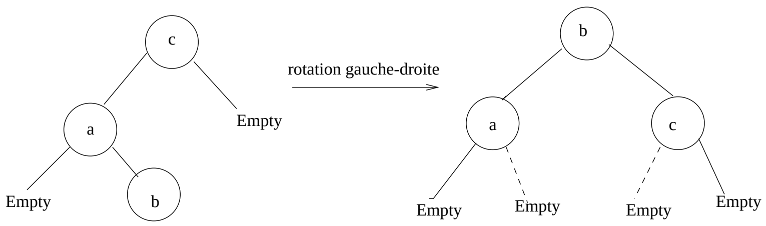


FIG. 21: *Cas "oublié" : si b (élément inséré) est une feuille avant l'insertion*

Notons que ce dernier cas "oublié" dans l'énoncé (et souvent dans la littérature) était la cause du mauvais comportement du programme sur certaines entrées. L'utilisation de la fonction de debuguage (impression du déséquilibre à côté du nœud) a permis de détecter cette erreur plus rapidement.

La fonction de recherche dans un AVL : `avl_insertion` est donnée en annexe. On décide ici d'afficher en rouge la partie de l'arbre modifiée.

```
avl_insertion : int -> avl -> avl = <fun>
```

5.4 Un exemple caractéristique

La fonction `lovely_example` de type

```
lovely_example : unit -> unit = <fun>
```

effectue la création d'un AVL (et sa visualisation graphique) par insertion successives des clefs 12, 3, 2, 5, 4, 7, 9, 11, 14, 10 à l'aide de la fonction `avl_insertion`. Sur cet exemple, tous les cas de rotation sont illustrés. Les différentes étapes de cette création (ainsi que les mises à jour des déséquilibres) sont données sur une feuille annexe .

6 Remarques sur les différents types d'arbres

Si l'avantage des ABR sur les arbres binaires est claire (les fonctions de parcours récursives ne sont pas plus longues à écrire, et l'on ne visite à chaque niveau qu'un seul sous-arbre, le sous-arbre gauche ou le sous-arbre droit), celle des AVL nécessite un peu de calcul. En effet, il est clair que dans le pire des cas, un arbre binaire de recherche à n nœuds peut avoir une hauteur $n-1$ (cas d'un peigne) alors qu'on la le résultat :

PROPOSITION 1 *Dans le pire des cas, la hauteur d'un AVL est logarithmique.*

PREUVE : Parmi les AVL de hauteur h, le plus petit nombre de nœuds possibles est m_h vérifiant :

- $m_0 = 1$ (par convention, un arbre de hauteur 0 a une clef)
- $m_1 = 2$
- $m_{h+2} = 1 + m_{h+1} + m_h$: on construit l'arbre de hauteur $h+2$ à l'aide des arbres aux nombres de nœuds minimaux précédents de hauteur $h+1$ et h (puisque l'on veut un arbre au nombre minimal de nœuds, mais respectant l'équilibrage des AVL), et on rajoute un nœud à la racine pour obtenir un arbre de hauteur supérieure.

Ainsi la suite $(1+m_h)$ est une suite de Fibonnaci, et les calculs usuels permettent d'obtenir l'expression suivante :

$$1 + m_h = \frac{2\varphi_2 - 3}{\sqrt{5}}\ {\varphi_1}^h + \frac{3 - 2\varphi_1}{\sqrt{5}}\ {\varphi_2}^h$$

où

$$\varphi_1 = \frac{1-\sqrt{5}}{2} \quad et \quad \varphi_2 = \frac{1+\sqrt{5}}{2}.$$

D'où, comme $|\varphi_1| < 1$, l'inégalité : $m_h \geq k_2\ {\varphi_2}^h + c$ où k_2 et c sont des constantes. Donc étant donné un arbre à n nœuds : $n \geq m_h$, on récupère :

$$h \leq C \log_2 n\ + D,$$

où C et D sont des constantes (on peut obtenir $C \leq 1,5$). La hauteur d'un AVL est donc bien logarithmique au pire. ■

Remarques

- Comme un arbre complet a une hauteur logarithmique en le nombre de nœuds, la hauteur d'un AVL est toujours logarithmique en son nombre de nœuds.
- La complexité d'une insertion étant comptée comme la profondeur du nœud inséré dans l'arbre, les AVL présentent donc un avantage certain sur les ABR *uniquement* dans le pire des cas (on a vérifié *expérimentalement* que la hauteur moyenne et la profondeur moyenne d'un nœud dans un ABR est logarithmique en le nombre de nœuds).