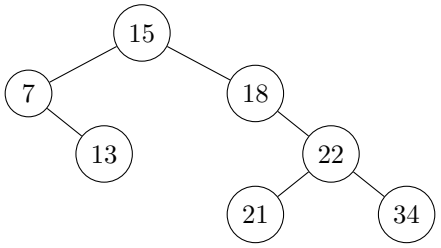
Fiche TD13 : Arbres binaires de recherche

Exercice 1 *Opérations sur un ABR*

On considère l'arbre binaire de recherche A :



1. Ajouter successivement les valeurs $9, 16, 23, 15, 0, 30, 20, 5$ à l'arbre A (en cas d'égalité, on insèrera à gauche).
2. Dessiner l'arbre A après suppression du noeud 22 puis du noeud 15 le moins profond. Dans le cas où le noeud à supprimer possède deux fils, on supprimera préférentiellement à gauche.
3. La suppression de x puis de y dans un ABR produit-elle le même arbre que la suppression de y puis de x ? Si oui, justifier, si non, donner un contre-exemple.
4. Rappeler où se trouve la plus grande valeur stockée dans un ABR. Ecrire un pseudo-code permettant de supprimer cette plus grande valeur de l'ABR en entrée. Faire de même pour la plus petite valeur.

Exercice 2 *Recherche approximative*

Lorsque la recherche d'un élément dans un ABR échoue, l'exécution du programme principal est généralement contrainte à s'arrêter et provoquer une erreur. Dans certains contextes, on préfère poursuivre l'exécution avec une valeur "proche" de la valeur recherchée. Le successeur d'un noeud x dans un ABR est par définition le noeud de l'ABR possédant la plus petite étiquette supérieure à celle de x. Similairement, le prédecesseur d'un noeud x est le noeud de l'ABR possédant la plus grande étiquette inférieure à celle de x. En particulier, si x est dans l'ABR, x est son propre successeur et prédecesseur. On considère que toutes les clés dans l'ABR sont différentes et qu'on peut avoir facilement accès au père d'un noeud.

1. Identifier le prédecesseur de 12 dans l'ABR de l'exercice 1 et le successeur de 19.
2. Proposer un algorithme permettant le calcul du prédecesseur d'un noeud dans un ABR et faire de même pour le successeur.
3. Quelle est la complexité de ces algorithmes ? Dans le pire cas ? Dans le meilleur cas ?

Remarque : La possibilité de déterminer un élément ayant une clé "proche" de celle recherchée est un des avantages des ABR par rapports à d'autres structures implémentant des tables d'associations.

Exercice 3 *Trier avec un ABR*

On note $\mathcal{A}$ l'algorithme suivant : $\mathcal{A}$ prend en entrée un tableau de taille n, insère tous les éléments de ce tableau dans un ABR initialement vide puis renvoit le parcours infixe de cet ABR.

1. Ecrire un pseudo-code pour l'algorithme $\mathcal{A}$ (on pourra le décomposer en plusieurs sous algorithmes de sorte à augmenter la clarté de l'ensemble).
2. Que fait l'algorithme $\mathcal{A}$? Justifier sa correction.
3. Etudier la complexité pire cas et la complexité meilleur cas de l'algorithme $\mathcal{A}$. Comment ces complexités sont-elles modifiées si l'insertion se fait dans un arbre rouge-noir plutôt qu'un ABR ? Comparer ces complexités à celles d'algorithmes connus pour résoudre le même problème que $\mathcal{A}$.

Exercice 4 *Insertion à la racine et rotations*

L'insertion dans un ABR se fait classiquement aux feuilles. Cela implique que les valeurs récemment insérées dans la structure sont celles pour lesquelles le temps de recherche est le plus long. Si tous les éléments ont la même probabilité d'être recherchés, ce n'est pas grave, mais dans certaines applications, les éléments récemment insérés sont recherchés plus fréquemment que les éléments anciens et dans ce cas l'insertion aux feuilles est une mauvaise stratégie en termes de complexité temporelle.

Dans un tel contexte, on préfère insérer les nouveaux éléments "à la racine" (le nouvel élément devient la racine de l'ABR). Evidemment, on souhaite que l'arbre résultant de cette insertion continue d'être un ABR.

1. Si A est un ABR et x un noeud de A, effectuer une rotation droite (respectivement, gauche) sur x produit un arbre contenant les mêmes clés que A et qui reste un ABR : justifier.

2. Insérer l'élément 25 dans l'ABR de l'exercice 1 à l'aide d'une insertion aux feuilles puis effectuer les opérations suivantes : rotation droite sur 34, rotation gauche sur 22, rotation gauche sur 18, rotation gauche sur 15. Que constatez vous ?

3. En vous inspirant des questions précédentes, proposer un algorithme permettant d'insérer à la racine d'un ABR. Justifier sa correction.

4. Quelle est sa complexité ? Comparer avec la complexité de l'insertion aux feuilles et conclure.

Exercice 5 *Calcul de rangs*

Dans cet exercice, on enrichit la structure d'ABR en ajoutant un champ entier n_g à chaque noeud, destiné à contenir le nombre de noeuds de son fils gauche. On suppose que tous les éléments stockés dans l'ABR sont distincts.

1. Dessiner l'ABR obtenu en insérant dans un arbre vide les valeurs suivantes : $12, 5, 24, 3, 37, 49, 25, 17, 8, 29$ dans cet ordre, en indiquant quelle est la valeur de n_g pour chaque noeud.

2. On suppose qu'on dispose de fonctions :

   - val(A), permettant d'accéder à la valeur stockée dans la racine de l'arbre A.
   - ng(A), permettant d'accéder au champ n_g de la racine de A.
   - fg(a) et fd(a), permettant d'accéder aux fils gauche et droit de la racine de A.
   - modifie_ng(A,k), permettant de remplacer la valeur du champ n_g dans la racine de A par k.
   - cons(val,ng,g,d) permettant de construire un arbre dont la racine est étiquetée par val, dont le champ n_g est ng et dont les fils gauche et droit sont respectivement g et d.

   Ecrire un algorithme d'insertion dans un ABR prenant en compte le fait que les champs n_g de certains noeuds doivent être modifiés. Indiquer sa complexité.

Le rang $r(x)$ d'un élément x figurant dans l'ABR A est par définition le nombre d'éléments de A strictement plus petits que x. Le rang du plus petit élément de A est donc 0.

3. Indiquer quels sont les rangs de chacune des valeurs de la question 1.

4. Soit $r \geq 0$ et supposons que l'ABR A contient au moins $r+1$ éléments. On note x l'élément de rang r dans A. Montrer que :

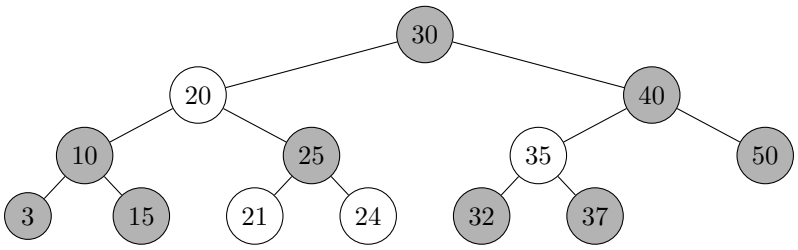
$$\begin{cases} \text{Si } r = \mathsf{ng}(A), \text{ alors } x = \mathsf{val}(A) \\ \text{Si } r < \mathsf{ng}(A), \text{ alors } x \text{ est l'élément de rang } r \text{ dans le fils gauche de } A \\ \text{Si } r > \mathsf{ng}(A), \text{ alors } x \text{ est l'élément de rang } r - 1 - \mathsf{ng}(A) \text{ dans le fils droit de } A. \end{cases}$$

5. En déduire un algorithme permettant de calculer l'élément de rang r dans A. Quelle est sa complexité ?

6. Ecrire un algorithme permettant de calculer le rang d'un élément x dans l'arbre A. On traitera de manière pertinente le cas où x n'est pas présent dans A. Justifier la correction de cet algorithme et établir sa complexité.

Exercice 6 *Insertion dans un arbre rouge-noir*

On considère l'arbre coloré A suivant (dont on omet de dessiner les feuilles, qui sont colorées en noir) :



1. Quelles feuilles doit on colorer en noir de sorte à ce que A soit un arbre rouge-noir ?

2. Indiquer quelles sont les hauteurs noirs de $30, 20, 25$ et 50.

L'algorithme d'insertion dans un arbre rouge-noir suit le principe suivant : on insère l'élément comme dans un ABR classique, on colorie le noeud inséré en rouge puis on rétablit les propriétés rouge-noir si nécessaire.

3. a) Comment corriger la couleur du noeud inséré si l'arbre initial était vide ?

   b) Que faut-il faire si le parent du noeud qu'on vient d'insérer est noir ?

   On suppose désormais que le parent P du noeud inséré I est rouge.

   c) Montrer que I a alors nécessairement un grand parent G et un oncle O.

   d) Dans le cas où O est noir, rappeler comment rétablir les propriétés rouge-noir (en justifiant) à l'aide d'au plus deux rotations et deux recoloriages de noeuds.

   e) Dans le cas où O est rouge, rappeler comment recolorier les noeuds de sorte à déplacer le problème consistant à avoir un noeud rouge ayant un fils rouge plus haut dans l'arbre.

4. Insérer les noeuds $48, 45, 61$ dans l'arbre A ; l'arbre résultant devra respecter les propriétés rouge-noir.

Exercice 7 *B-arbres*

Un B-arbre est un arbre défini comme suit. Chaque noeud x d'un B-arbre stocke les attributs suivants :

- Le nombre n de clés (= étiquettes) stockées dans le noeud.
- Les n clés annoncées $c_1, ..., c_n$ dans l'ordre croissant.
- $n+1$ pointeurs vers ses $n+1$ fils (pour les feuilles, ces pointeurs sont nuls).

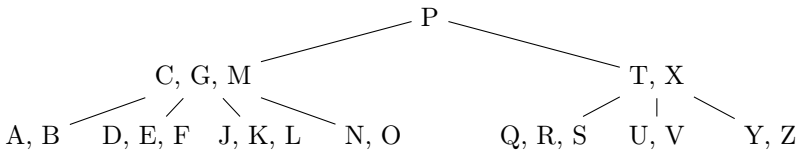
Un B-arbre vérifie de plus les propriétés suivantes :

(1) Toutes ses feuilles sont à la même profondeur.

(2) Si k est une clé dans le i-ème fils d'un noeud avec $i \in [\![1, n+1]\!]$, $c_{i-1} \leq k \leq c_i$ (pour $i \in \{1, n+1\}$, seule l'inégalité ayant du sens est vérifiée).

(3) Il existe un entier $D \geq 2$, appelé degré minimal du B-arbre, tel que tout noeud autre que la racine contient au moins $D-1$ clés et au plus $2D-1$ clés. Si l'arbre n'est pas vide, la racine doit contenir au moins une clé.

Par exemple, l'arbre suivant est un B-arbre de degré minimal $D = 3$ dont les clés sont des lettres et dont on a omis pour chaque noeud de préciser le nombre de clés qu'il stocke :



1. Expliquer comment utiliser les propriétés d'un B-arbre pour y rechercher efficacement un élément.

2. On se donne un B-arbre A de degré minimal D, de hauteur h et contenant n clés au total. Montrer que

$$h \leq \log_D \left(\frac{n+1}{2} \right)$$

   En déduire la complexité pire cas d'une recherche dans un B-arbre. *Indication : On pourra commencer par chercher un minorant du nombre de noeuds à la profondeur p dans A.*

3. Comparer le résultat de la question 2 avec l'inégalité classique liant la hauteur et le nombre de clés stockées dans un arbre rouge-noir. Qu'en penser ?

Remarque : Tout comme les arbres rouge-noir, les B-arbres sont une extension des ABR garantissant un équilibre de la structure. Le "B" de B-arbre n'a pas de signification confirmée : il pourrait vouloir dire "balanced" (équilibré en anglais) ou dériver du nom d'un de ses inventeurs, Rudolf Bayer.