

Exercice : arbres binaires de recherche et B tree

27 mai

1. Rappeler la définition d'un arbre binaire de recherche stockant des entiers munis de la relation d'ordre usuelle.

On utilisera le type suivant pour représenter un tel arbre de recherche :

```
type arbre = Vide | N of (int*arbre*arbre)
```

2. Ecrire une fonction `recherche` : `arbre->int->bool` qui prend en entrée un arbre binaire de recherche et un entier et renvoie `true` si et seulement si l'élément passé en argument est stocké dans l'arbre binaire de recherche.
3. Donner la complexité de votre fonction. Discuter de celle-ci dans le meilleur cas, le pire cas suivant la forme de l'arbre passé en argument. Quelle solution, au programme de Mp2i/MPI, permettrait de garantir une complexité efficace?
4. Ecrire une fonction `collecte` : `arbre-> int list` qui prend en argument un arbre binaire de recherche et renvoie la liste des entiers qui le composent triée par valeurs croissantes. Quelle est la complexité de votre fonction?
5. Nous proposons le programme suivant pour déterminer si un arbre binaire est bien un arbre binaire de recherche :

```
let rec est_ABR a = match a with
  |Vide->true
  |N(x,Vide,Vide)->true
  |N(x,Vide,(N(y,gd,dd) as d))-> (x< y) && (est_ABR d )
  |N(x,(N(y,gg,dg) as g),Vide)-> (y<=x) && (est_ABR g)
  |N(x,(N(y,gg,dg) as g),(N(z,gd,dd) as d))-> (x>=y)&&(x<z)&&(est_ABR g)&&(est_ABR d);;
```

A l'aide d'une exemple bien choisi, montrer que cette fonction est incorrecte.

6. Pour proposer une version correcte, nous allons associer un intervalle à chaque noeud dans lequel doit se trouver la clé du noeud de la manière suivante : lorsqu'un arbre a pour racine k alors son sous arbre gauche a toutes ses clés inférieures ou égales à k et les clés du sous arbre droit sous toutes strictement supérieures à k . Ainsi la racine est associée à l'intervalle `[min_int,max_int]` et les autres noeuds correspondent à des intervalles dont les bornes dépendent des valeurs des noeuds qui composent le chemin depuis la racine de l'arbre. Donner les intervalles pour chacun des noeuds de l'arbre suivant :

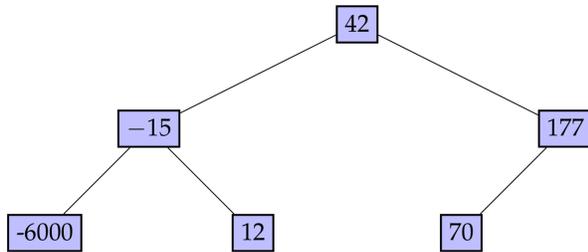


FIGURE 1 – Un arbre binaire de recherche

7. On utilisera donc `max_int` et `min_int` pour écrire une fonction `est_abr_correcte : arbre -> bool` qui détermine si un arbre passé en argument est bien un arbre binaire de recherche. On pourra écrire une fonction auxillaire qui prend en arguments les bornes qui encadrent la valeur de la racine.

Nous allons maintenant travailler sur les B-tree qui consistent en une généralisation de la notion d'arbre binaire de recherche et sont utilisés dans les gestionnaires de bases de données. Un *B-tree* est un arbre d'arité quelconque défini de la manière suivante :

Un B-tree d'ordre t est un arbre qui satisfait les propriétés suivantes :

- (S1) Tous les chemins de la racine à une feuille ont la même longueur, appelée hauteur de l'arbre.
- (S2) La racine est une feuille ou bien a au moins 2 enfants.
- (S3) Chaque noeud qui n'est ni racine ni feuille possède au moins $t+1$ enfants.
- (S4) Chaque noeud a au plus $2t + 1$ enfants.
- (C1) Les noeuds contiennent des clés : la racine contient de 1 à $2t$ clés et les autres noeuds, entre t et $2t$ clés.
- (C2) Dans chaque noeud, les éléments sont stockés de manière croissante.
- (C3) Chaque noeud qui contient ℓ clés a $\ell + 1$ enfants.
- (C4) Soit P un noeud contenant ℓ clés $x_0, \dots, x_{\ell-1}$. Soient P_0, \dots, P_ℓ ses enfants et $K(P_i)$ ($0 \leq i \leq \ell$) l'ensemble des clés de chacun des sous arbres P_i . On a alors :

- $\forall y \in K(P_0), y \leq x_0$
- $\forall 1 \leq y \leq \ell - 1, \forall y \in K(P_i), x_{i-1} \leq y \leq x_i$
- $\forall y \in K(P_\ell), y \geq x_{\ell-1}$

8. Pour chacun des arbres suivants , dire si c'est bien un B-tree d'ordre 2.
9. Dans le B-tree de la figure suivante, expliquer comment chercher la clés de valeur 8.

Pour représenter un B-tree, on va utiliser la type suivant :

```
type btree = Feuille of (int list) | Noeud of (int list * btree array)
```

Ainsi, chaque noeud contient la liste triée de ses c clés et un tableau de taille $c + 1$ contenant ses sous-arbres.

Les arbres de la Figure 3 sont-ils des B-trees d'ordre 2?

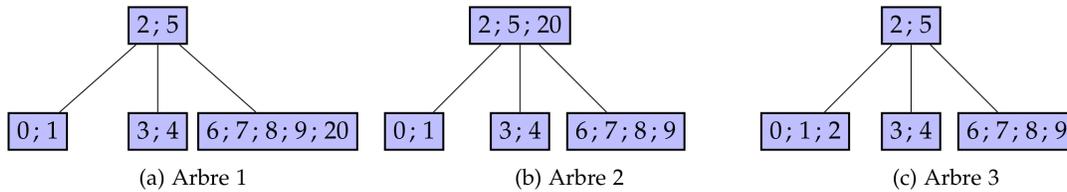


FIGURE 3 – Quelques arbres.

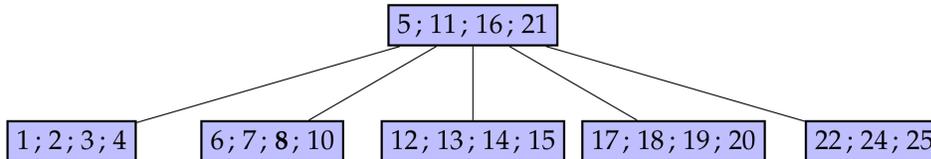


FIGURE 4 – Exemple de B-tree (d'ordre 2) pour la recherche

10. Ecrire une fonction `recherche_b : btree -> int -> bool` qui détermine si un élément est contenu dans un B-tree.
11. On va maintenant analyser la complexité de cette recherche. Soit a un B-tree non vide d'ordre $t > 1$ et de hauteur $h \geq 1$. Nous allons d'abord compter le nombre noeuds maximal N_{max} (resp. minimal N_{min}) qu'il contient.
 - (a) Montrer que $N_{min} = 1 + \frac{2}{t}((t+1)^{h-1} - 1)$ pour $h \geq 2$.
 - (b) Montrer vque $N_{max} = \frac{1}{2t}((2t+1)^h - 1)$ pour $h \geq 1$.
 - (c) En déduire un encadrement du nombre de clés contenues dans un B-tree d'ordre t et de hauteur $h \geq 2$.
 - (d) Conclure sur la complexité de la recherche dans un B-tree en fonction de son nombre de noeuds.

Nous allons maintenant déterminer comment insérer un élément dans un B-tree tout en respectant sa structure.

L'algorithme d'insertion d'une clé réalise des étapes similaires à celui de l'insertion dans un arbre binaire de recherche : l'arbre est parcouru récursivement à partir de la racine, jusqu'à trouver une feuille dans laquelle insérer l'élément. L'insertion se réalise donc **dans une feuille**. Les B-trees ont néanmoins une contrainte supplémentaire : un nombre minimum et maximum de clés par noeud. Avant d'insérer une clé dans un noeud, il faudra s'assurer qu'un noeud "n'est pas complet".

12. Expliquer comment insérer la clés 23 dans l'arbre suivant :

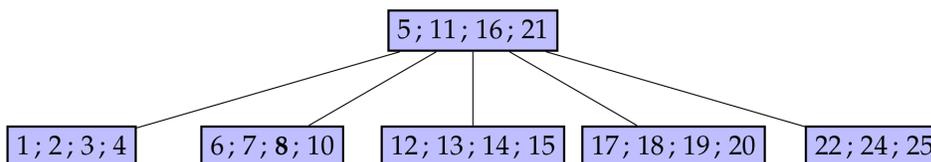


FIGURE 4 – Exemple de B-tree (d'ordre 2) pour la recherche

13. Ecrire une fonction `insere_pos : btree->int->int` qui prend en entrée un B-tree et un entier et renvoie l'indice du sous-arbre dans lequel il faudrait insérer la nouvelle clé passée en argument si l'arbre est une feuille ou bien l'indice où il faudrait insérer l'élément dans la liste si celui-ci est une feuille non complète.
14. En utilisant la fonction précédente, écrire une fonction `insertion_noncomplet : btree->int->btree` qui réalise l'insertion d'une nouvelle clé dans un B-tree sous l'hypothèse que la feuille dans laquelle on fera l'insertion n'est pas pleine.
15. Donner la condition qui permet de déterminer qu'un noeud est complet.

Dans le cas où l'endroit d'insertion est complet, le noeud va être "scindé en deux" (split), afin de récupérer deux noeuds de tailles plus petites. La Figure 5 explique le processus. On considère un B-tree d'ordre $t = 2$ et un sous-arbre enraciné en x , dont l'un des enfants y , doit être scindé, car il contient trop de clefs (ici 5 clefs). Le noeud y est donc scindé en deux noeuds y et z de taille égale à 2 et les sous-arbres associés sont "distribués" en conséquence. L'élément "en trop", ici 8, est alors éjecté, pour être inséré par la suite dans l'ensemble des clefs du noeud parent, x (qui lui même peut être complet, comme nous le verrons par la suite).

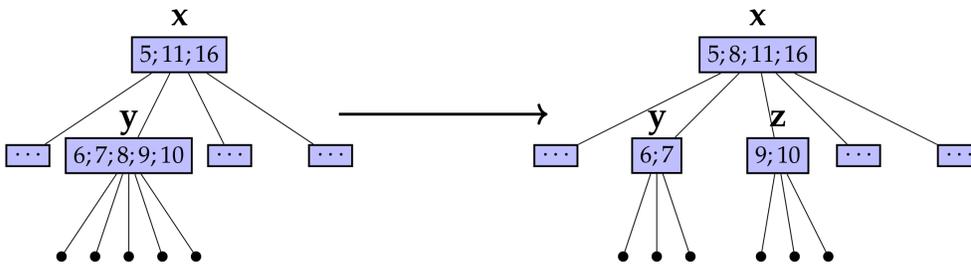


FIGURE 5 – Illustration (partielle) de l'insertion de 8 dans le noeud y avec « split » : lors de l'insertion, le noeud y possède temporairement 5 clefs, ce qui n'est pas permis pour un B-tree d'ordre 2. La procédure de découpage ("split") va donc créer le nouveau noeud z et éjecter la valeur 8. Cette valeur sera retournée au parent à la fin de l'appel récursif d'insertion.

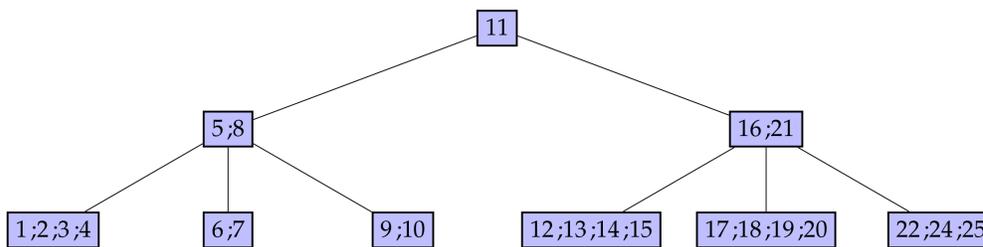


FIGURE 6 – B-tree résultant de l'insertion de la clef 9 dans le B-tree de la Figure 4

On considérera que t est une variable globale.

16. Ecrire une fonction `split : btree-> (int*btree*btree)` qui prend en entrée un B-tree (quelconque) dont le nombre de clés de la racine est supposé (temporairement) égal à $2t + 1$. La fonction renvoie la valeur "ejectée" ainsi que le B-tree enraciné en y actualisé et le nouveau B-tree enraciné en z . Cette fonction, non récursive, ne réalise pas l'insertion dans le noeud parent. On pourra utiliser `Array.sub`.

La procédure d'insertion récursive peut réaliser de multiples scindages, si la valeur éjectée d'un nœud doit être insérée dans un nœud déjà complet. Le résultat final de la procédure d'insertion est donc possiblement un B-tree très différent du B-tree initial. Ainsi, l'insertion de la clé 9 dans le B-tree de la Figure 4 a pour résultat le B-tree de la Figure 6.

17. Que se passe-t-il lors de l'insertion si la racine est complète ?

Notre fonction d'insertion aura la forme suivante :

```
let insertion a k =  
  let valeur, a_maj, anew = insert a k in  
  if (valeur <> (-1) ) then (make_new_root a_maj valeur anew ) else a_maj;;
```

18. Ecrire la fonction `make_new_root: btree->int->btree->btree` qui gère le cas final de la racine.

19. Terminer l'implémentation de l'insertion en implémentant la fonction :

```
insert : btree->int->int*btree*btree.
```