TD 6

Les arbres binaires de recherche

Type en C des arbres binaires (également utilisé pour les ABR) :

```
typedef struct noeud_s {
  struct noeud_s * parent;
  struct noeud_s * gauche;
  struct noeud_s * droite;
  element_t e;
} * ab_t, * abr_t;
```

Exercice 1.
Combien y a t'il de formes d'arbres binaires différents à (respectivement) 1, 2, 3, 4 nœuds ? Les dessiner (sans donner de contenu aux nœuds).

Pour n fixé quelconque donner un exemple d'arbre binaire de hauteur majorée par $\log n + 1$, et un exemple d'arbre binaire de hauteur n.

Exercice 2.
Écrire une fonction $\mathsf{Taille}(x)$ prenant un arbre binaire et rendant le nombre de ses éléments.

Exercice 3.
Écrire une fonction $\mathsf{Hauteur}(x)$ prenant un arbre binaire et rendant sa hauteur, c'est à dire le nombre d'éléments contenus dans la plus longue branche.

Exercice 4.
Dans les deux exemples d'arbres binaires de recherche de la figure 1 :

1. où peut-on insérer un élément de clé 13 ?
2. comment peut-on supprimer l'élément de clé 14 ?

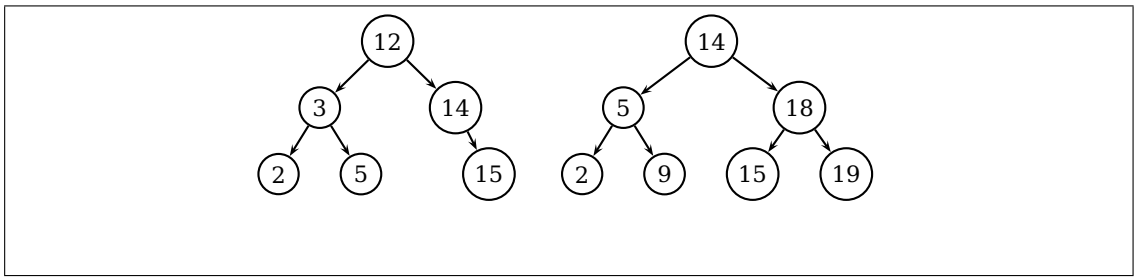


Figure 1: Deux arbres binaires de recherche

Les rotations des arbres binaires sont données dans la figure 2. Elles se lisent comme ceci. Une rotation à droite de centre x consiste en : prendre l'élément a de x, le sous-arbre droite E de x, la racine y du sous-arbre gauche de x, l'élément b contenu dans y, et C et D les deux sous-arbres gauche et droite de y ; remplacer a par b dans x, remplacer le sous-arbre gauche de x par C, et le sous-arbre droite de x par un arbre de racine contenant a (on utilise y pour des raisons d'implantation) et dont les sous-arbres gauche et droite sont respectivement D et E. La rotation à gauche de centre x est l'opération réciproque.

Avec cette manière d'écrire les rotations la référence de x à son parent n'a pas besoin d'être mise à jour. Il est assez standard de trouver une version qui échange la place de x et de y plutôt que d'échanger leurs éléments mais nous ne l'utilisons pas ici.

3. Sur le premier exemple de la figure 1, faire : une rotation à droite de centre le nœud de clé 3 ; puis une rotation à gauche de centre le nœud de clé 14.

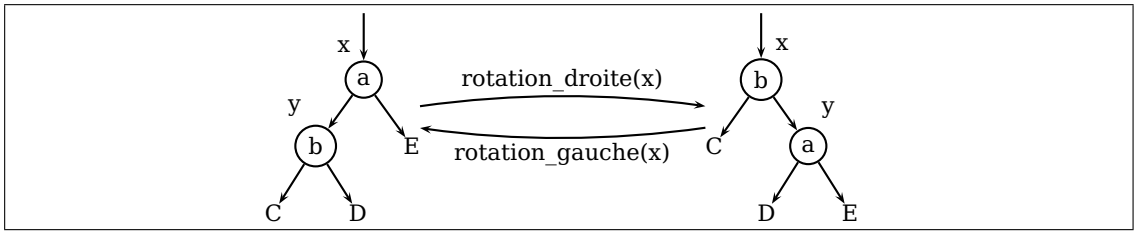


Figure 2: Rotations gauche et droite. Dans cette version les éléments a et b sont échangés entre les nœuds x et y mais x garde sa place dans l'arbre qui le contient.

4. Sur le second exemple de la figure 1, à l'aide de rotations dont vous préciserez le sens et le centre, amener le nœud de clé 9 à la racine.
5. Démontrer que toute rotation préserve la propriété d'être un arbre de recherche.
6. Écrire l'algorithme de rotation à droite en `C`.
7. Écrire un algorithme permettant de remonter à la racine n'importe quel nœud d'un arbre binaire de recherche, à l'aide de rotations.

Exercice 5 (Insertion / suppression ABR)**.**
Former un arbre binaire de recherche en insérant successivement et dans cet ordre les éléments : $10, 7, 3, 9, 11, 5, 6, 4, 8$ (répondre en représentant l'arbre obtenu). Supprimer l'élément 7. (répondre en représentant le nouvel arbre. Il y a deux réponses correctes possibles, selon la variante choisie pour l'algorithme de suppression, n'en donner qu'une seule).

1,5 pt
13 min

Exercice 6 (À la fois ABR et tas ?)**.**
Un tas est nécessairement un arbre binaire quasi-parfait. Est-il toujours possible d'organiser un ensemble de n clés (n quelconque) en tas max de manière à ce que cet arbre binaire soit aussi un arbre binaire de recherche ? (Justifier par un raisonnement ou un contre-exemple).

1,5 pt
9 min

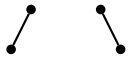
Exercice 7.
On peut afficher les éléments d'un ABR de taille n en ordre trié en un temps $O(n)$.

1. Expliquer comment et argumenter sur le temps d'exécution.
2. Est-ce que la propriété de tas permet d'afficher en ordre trié et en temps $O(n)$ les éléments d'un tas de taille n ? Argumenter. Indication : on peut *planter* (former) un tas de n éléments en un temps $\Theta(n)$.

Corrigé

Correction de l'exercice 1.

Un seul arbre à un nœud, deux à deux nøeuds :



Cinq à trois nœuds :

Quatorze arbres à quatre nœuds (non dessinés). On peut le calculer en trouvant la récurrence :

$$C_{n+1} = \sum_{k=0}^{n} C_k \times C_{n-k}$$

Qui donne ici $C_4 = C_0 \times C_3 + C_1 \times C_2 + C_2 \times C_1 + C_3 \times C_1 = 5 + 2 + 2 + 5 = 14$. (nombres de Catalan).

Pour n fixé l'arbre quasi-parfait à n nœuds est tel que si sa hauteur est h alors :

$$2^{h-1} - 1 < n \leq 2^h - 1$$

D'où $h - 1 \leq \log n < h$, donc $\log n + 1$ majore h.

L'arbre peigne est quand à lui exactement de hauteur n.

Correction de l'exercice 2.

```
int taille(ab_t x) {
  if (estVide(x)) {
    return 0;
  }
  return 1 + taille(gauche(x)), taille(droite(x));
}
```

Correction de l'exercice 3.

```
void hauteur(ab_t x) {
  if (estVide(x)) {
    return 0;
  }
  return 1 + max(hauteur(gauche(x)), hauteur(droite(x)));
}
```

Correction de l'exercice 4.

L'insertion est donnée par les figures 3 et 4.
La suppression est donnée par les figures 5 et 6.

1. Figure 7.
2. rotation à gauche de centre 5, puis rotation à droite de centre 14.

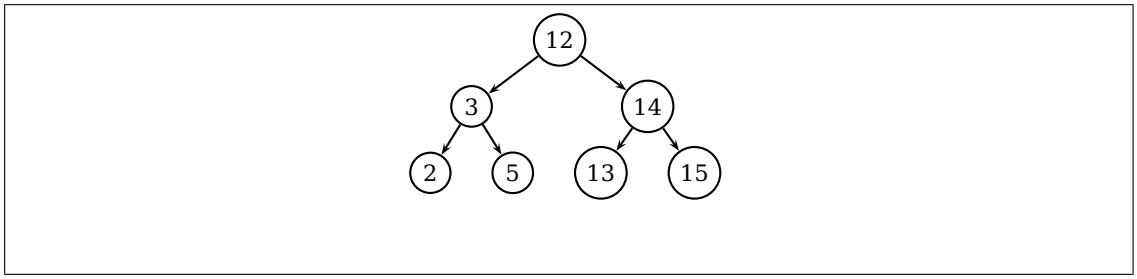


Figure 3: Un arbre binaire de recherche - corrigé

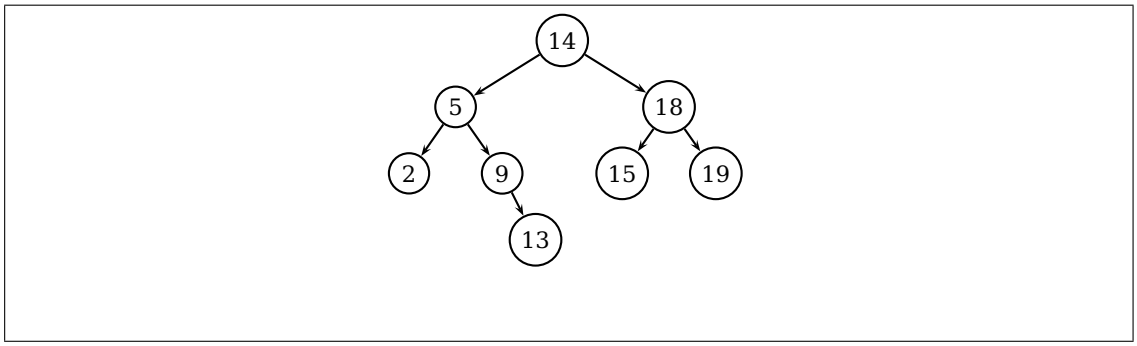


Figure 4: Un autre arbre binaire de recherche - corrigé

3. Il faut montrer la préservation de la propriété d'arbre de recherche. On se contente de montrer que si l'arbre à gauche de la figure est un arbre binaire de recherche alors l'arbre à droite en est un,et réciproquement. En effet, pour l'arbre qui contient l'un de ces deux arbres comme sous-arbre, le fait de remplacer l'un par l'autre ne fait aucune différence : les deux sous-arbres ont mêmes ensembles d'éléments. Pour chacun des deux arbres de la figure on montre qu'être un arbre de recherche, sous l'hypothèse que C, D et E en sont, est équivalent à la propriété : $\forall c \in C$, $\forall d \in D$, $\forall e \in E$, $\text{cle}(c) \leq \text{cle}(b) \leq \text{cle}(d) \leq \text{cle}(a) \leq \text{cle}(e)$, ce qui conclue.

4.
```
/* Rotations :

 x -> a       ---rot. droite de centre x-->     b <- x
     / \                                       / \
    b   E     <--rot. gauche de centre x---   C   a
   / \                                           / \
  C   D                                         D   E

*/

void rotation_droite(ab_t x){
  element_t tmp;
  ab_t y;

  assert(x && x->gauche);

  y = x->gauche;

  /* échange des éléments */
  tmp = x->e;
  x->e = y->e;
```

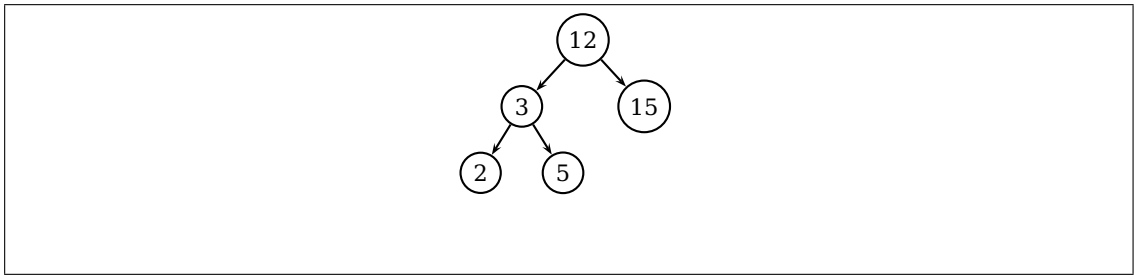


Figure 5: Un arbre binaire de recherche - `corrigé B`

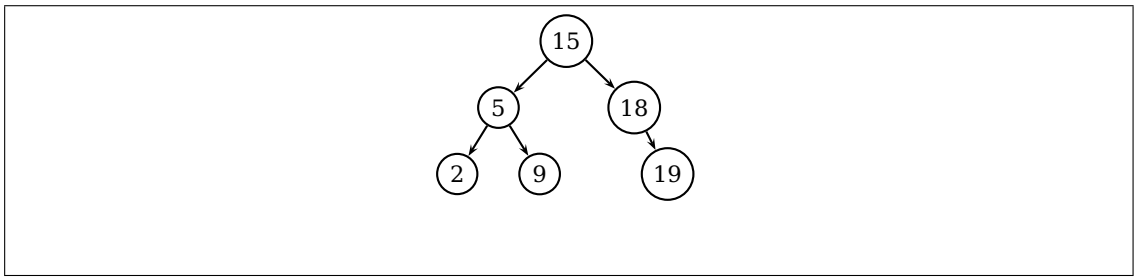


Figure 6: Un autre arbre binaire de recherche - `corrigé B`

```
  y->e = tmp;

  /* déplacement des sous-arbres */
  x->gauche = y->gauche;
  y->gauche = y->droite;
  y->droite = x->droite;
  x->droite = y;

  /* mise à jour des parents */
  (x->gauche)->parent = x;
  (y->droite)->parent = y;
}


void rotation_gauche(ab_t x){
  element_t tmp;
  ab_t y;

  assert(x && x->droite);

  y = x->droite;

  /* échange des éléments */
  tmp = x->e;
  x->e = y->e;
  y->e = tmp;

  /* déplacement des sous-arbres */
  x->droite = y->droite;
  y->droite = y->gauche;
  y->gauche = x->gauche;
```

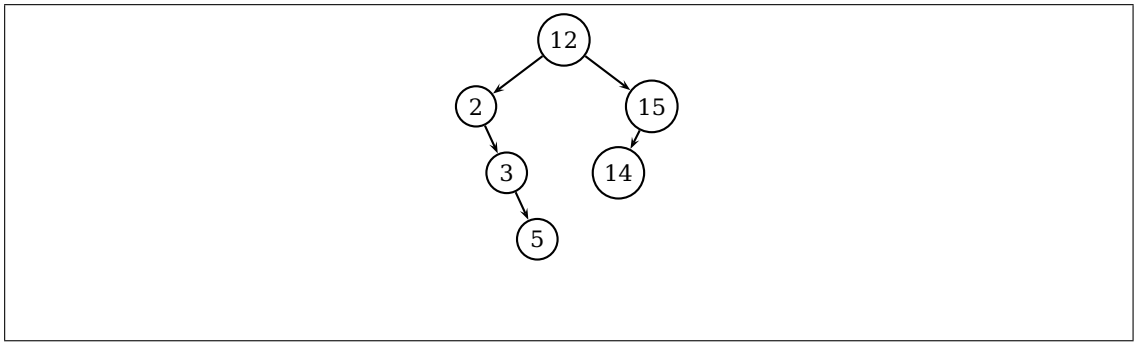


Figure 7: Question 3.1 - corrigé

```
  x->gauche = y;

  /* mise à jour des parents */
  (x->droite)->parent = x;
  (y->gauche)->parent = y;
}
```

5. Voici l'algo en pseudo-code.

Fonction Remonter(x)

```
y = Parent(x);
si y ≠ NULL alors
    /* y existe, x n'est pas la racine                                  */
    si x == Gauche(y) alors
        /* x est fils gauche de y                                       */
        RotationDroite (y);
    sinon
        /* x est fils droit de y                                        */
        RotationGauche (y);
    /* L'élément qui était dans le nœud x est désormais dans le nœud y  */
    Remonter (y);
```

Et en C :

```
void remonter(abr_t x) {
  y = x->parent;
  if (y) { /* x n'est pas encore à la racine */
    if (x == y->gauche) {
      rotation_droite(y);
    }
    else {
      rotation_gauche(y);
    }
    /* l'élément qui était contenu dans x est maintenant dans y */
    remonter(y);
  }
}
```

Correction de l'exercice 5.

Pas de correction.

Correction de l'exercice 6.

Dès que n vaut 3 on est confronté à un problème si a est l'élément à la racine de l'arbre quasi-parfait et que b est son fils gauche et c sont fils droit alors par la propriété des tas on doit avoir $a \geq c$ et par la propriété des ABR $a < c$. Impossible.

Correction de l'exercice 7.

On peut le faire avec la fonction de parcours infixe. Cette fonction est appelée une fois sur chaque nœud de l'arbre et une fois et chacun de ses appels génère au maximum 2 appels sur des nœuds vides (NULL) ne générant aucun nouvel appel. Ainsi il y a au maximum $3 * n$ appels à cette fonction au cours d'un parcours (on pourrait être plus précis et trouver $2n + 1$) et chaque appel se faisant en temps constant (pas de boucle), le temps total du parcours est

```
void parcours_infixe(abr_t x) {
  if (x) {
    parcours_infixe(x->gauche);
    affiche_element(x->e);
    parcours_infixe(x->droite);
  }
}
```

Si on pouvait parcourir les éléments d'un tas en ordre trié en temps $O(n)$, comme on plante un tas en $O(n)$, on aurait un tri par comparaison en $O(n)$. Impossible. Le parcours du tas en ordre trié est ainsi nécessairement en $\Omega(n \log n)$.