Algorithmique en Bioinformatique
Fouille de Données

Master de Bioinformatique

M1 Bioinformatique et Biostatistiques (BIBS)

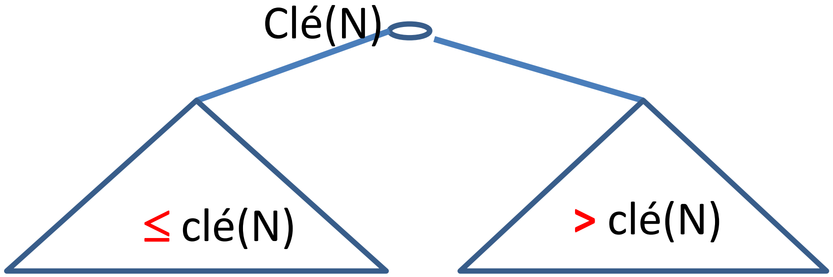
Alain DENISE

CHAPITRE 4 : ARBRES (DEUXIÈME PARTIE)

Arbres binaires de recherche

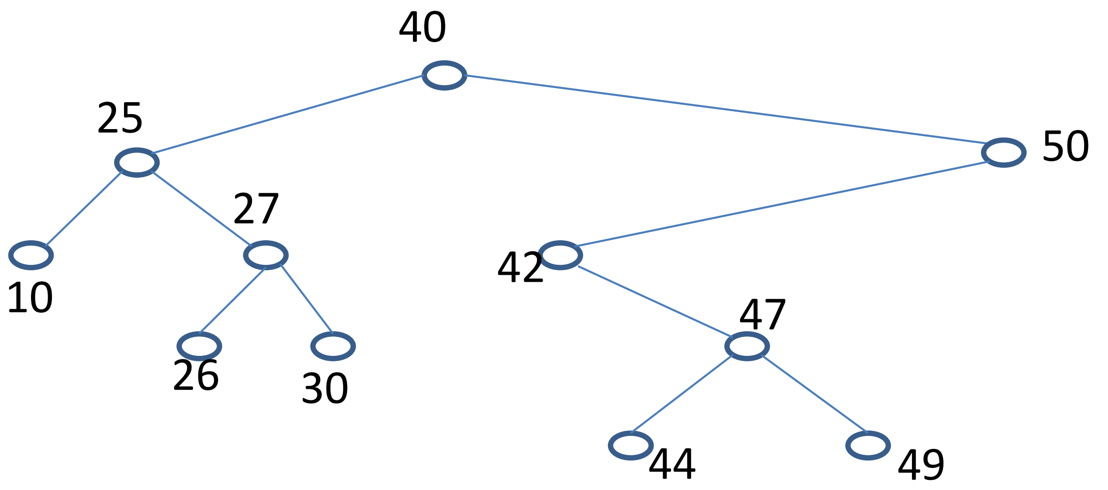
- **But :** stocker efficacement une collection d'éléments dont les clés sont comparables deux à deux.

- **Déf** : Un ***Arbre Binaire de Recherche*** (ABR) est un arbre binaire étiqueté B tel que pour tout noeud N de l'arbre B, les clés des nœuds du sous-arbre gauche de N sont inférieures ou égales à la clé de N et les clés des nœuds du sous-arbre droit de N sont strictement supérieures à la clé de N.
- En tout nœud N on a :

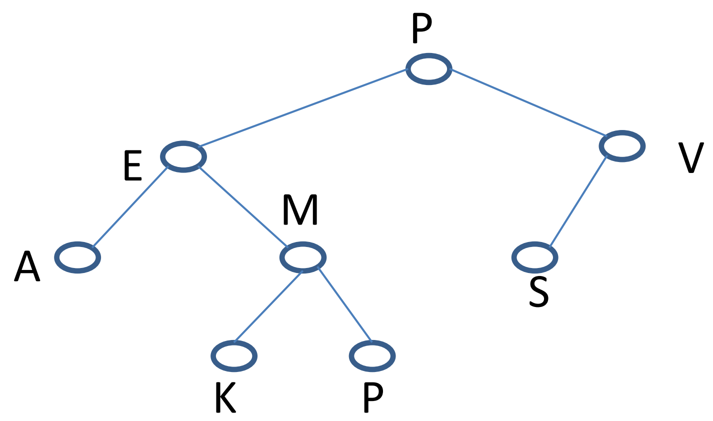


Exemples

B1 ABR ?

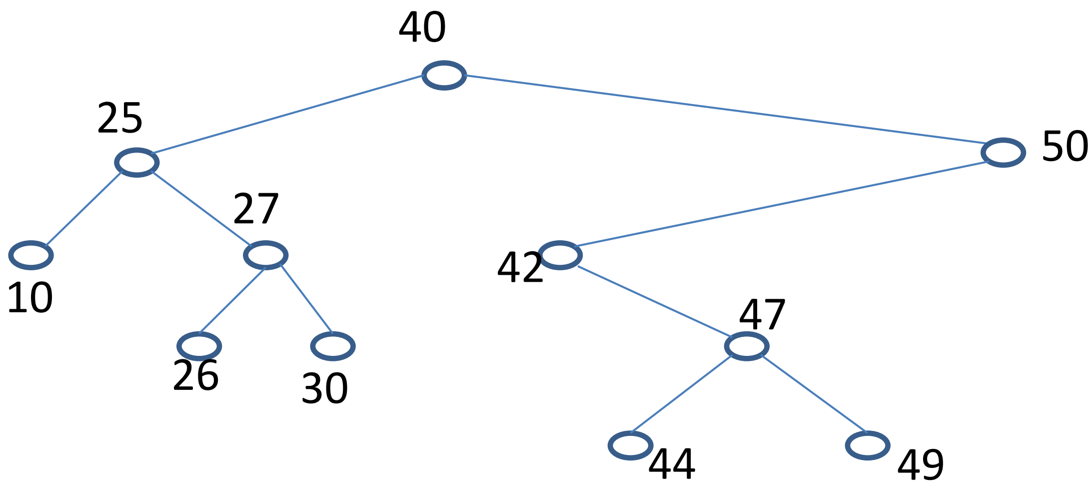


B2 ABR ?



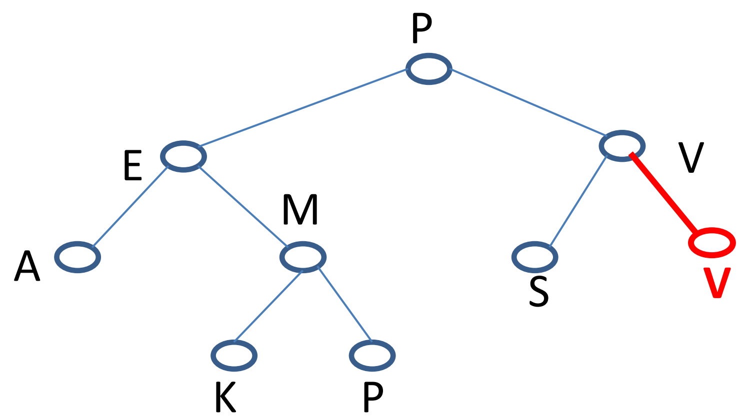
Exemples

B1 ABR ?



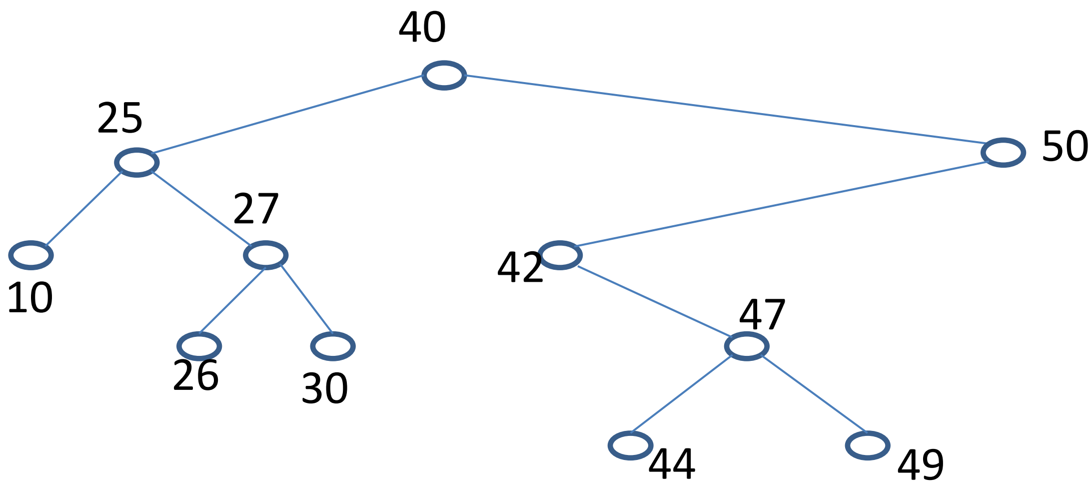
B2 ABR ?

B2' ABR ?
(B2' = B2
plus feuille
rouge
étiquetée V)



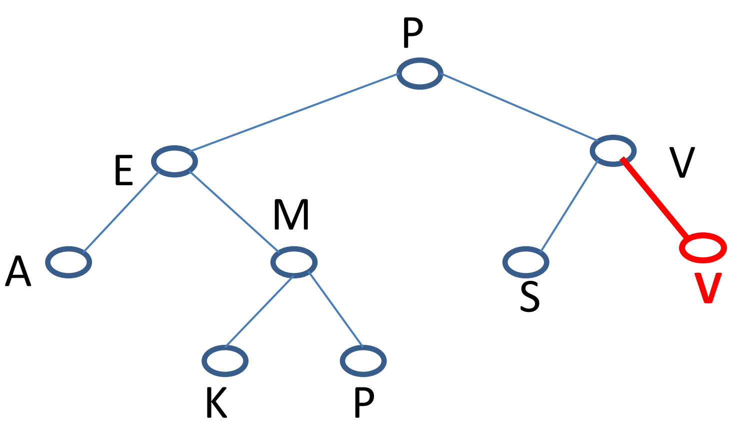
Exemples

B1 ABR ?
OUI.



B2 ABR ?
OUI.

B2' ABR ?
(B2' = B2 plus feuille rouge étiquetée V)
NON.



Propriété

- **Propriété** : La lecture en ***ordre symétrique (ou infixe)*** d'un ABR respecte l'ordre, cad, donne la liste des éléments en ordre croissant.
- **Exercice 1** : pour B1, l'ordre symétrique donne :

Recherche d'un élément dans un ABR

- **But:** trouver une occurrence quelconque d'un élément dont la clé est x dans un ABR B

- **Principe** : on procède **récursivement**
  - Si B est vide, alors arrêt (échec)
  - Sinon on compare x à la clé de la racine :
    - Si égalité alors succès
    - Si x < clé(racine) on cherche à gauche
    - Si x > clé(racine) on cherche à droite

- On peut aussi rechercher **itérativement**, selon un principe similaire.

Exercice 2 : Recherche récursive d’un élément dans un ABR

Fonction rech-ABR(B : ARBRE (IN); x : entier (IN)) → ARBRE ;

// la fonction récursive retourne null s’il n’y a pas d’occurrence de x dans B et sinon un pointeur sur un nœud dont la clé vaut x

Exercice 3 : Recherche itérative d'un élément dans un ABR

Fonction rech-ABR(B : ARBRE (IN); x : entier (IN)) → ARBRE ;

// la fonction itérative retourne null s'il n'y a pas d'occurrence de x dans B et sinon un pointeur sur un nœud dont la clé vaut x

Complexité

- **Complexité en temps :** deux facteurs
    - L'endroit où se situe la clé cherchée si elle est présente
    - La configuration de l'ABR : dégénérée ou bien équilibrée

Opération fondamentale : **comparaison entre clé(B) et x**

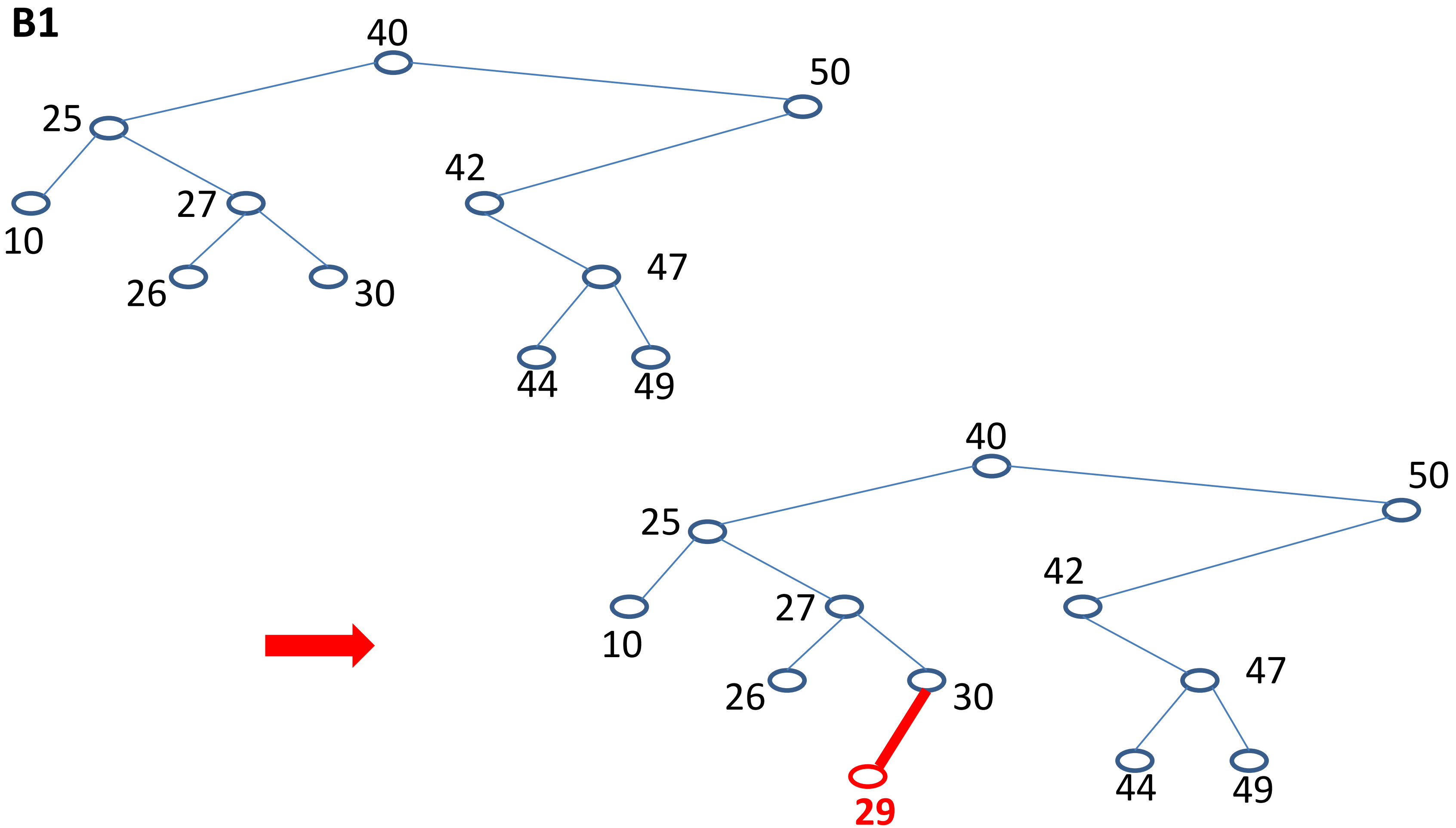
- <u>Au mieux</u> :

- <u>Au pire</u> :

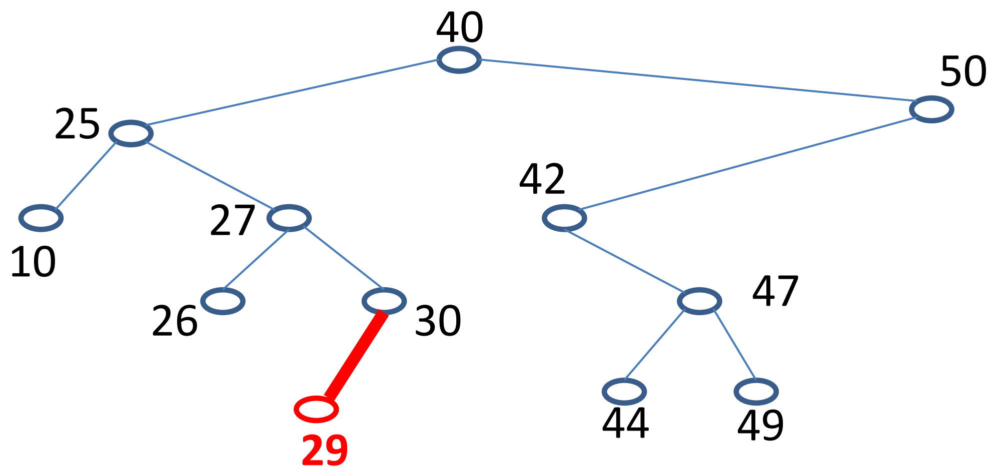
- <u>En moyenne</u> :

Adjonction aux feuilles d'un élément dans un ABR

Exemple : ajouter un nœud de **clé 29** à B1



Adjonction aux feuilles d'un élément dans un ABR



Principe : (1) Recherche négative de la feuille où il faudrait trouver la clé à ajouter dans l'ABR si elle existait (attention une même clé peut avoir plusieurs occurrences)
(2) Insertion proprement dite avec raccrochage au père comme fils droit ou fils gauche

Question : où faudrait-il rajouter une nouvelle occurrence de 47 ?

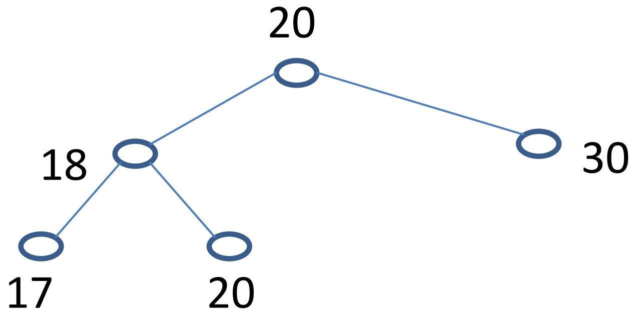
Exercice 4 : adjonction aux feuilles

Construire l'ABR (arbre binaire de recherche) obtenu par adjonctions successives aux feuilles des entiers suivants : 5, 1, 4, 9, 7, 20, 17, 13.

Exercice 5 : adjonction aux feuilles

Indiquer toutes les listes possibles d'éléments telles que l’adjonction aux feuilles itérée donne l’ABR A :

A :

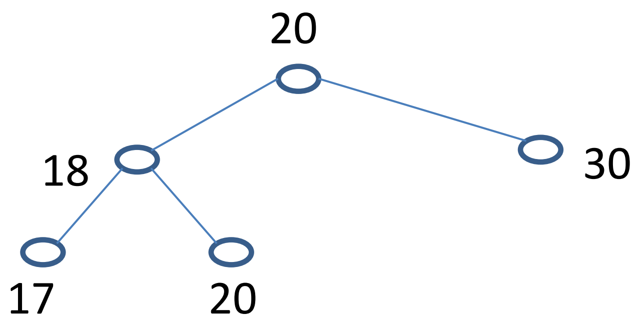


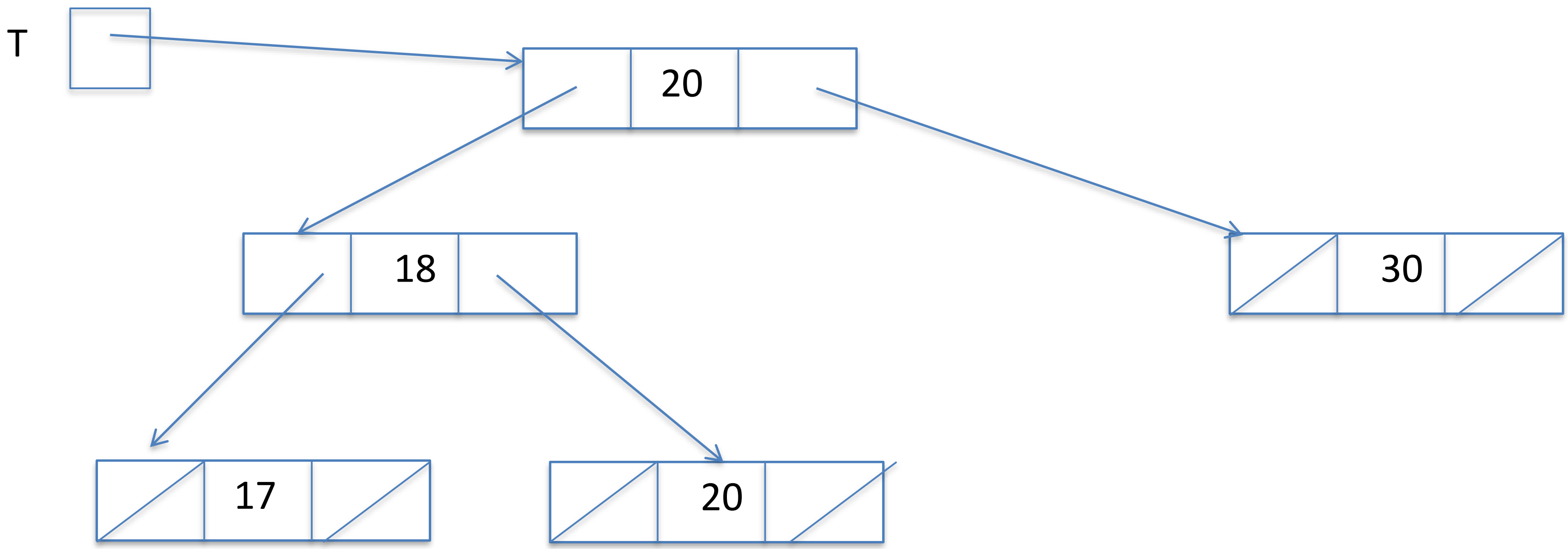
Procédure ajout-feuille-ABR(T : ARBRE (IN/OUT); x : entier (IN))

```
// procédure itérative d'adjonction aux feuilles de T d'un nouveau nœud dont la clé est x
Lexique : Z, A, B : ARBRE ;
Début
    (1) //création du nouveau nœud ; Z pointe sur ce nouveau nœud
        allouer(Z) ; clé(Z) ← x ; g(Z) ← null ; d(Z) ← null ;
    (2) //initialisation des pointeurs A et B tels (par la suite B sera père de A dans T)
        A ← T ; B ← null ;
    (3) //recherche négative de x en partant de la racine
        tant que A ≠ null faire {
                B ← A;
                si clé(A) < x alors A ← d(A)
                sinon A ← g(A)
        }
        // A = null et B pointe sur la feuille à laquelle raccrocher Z
    (4) si B = null alors T ← Z                // nouvel arbre T réduit au nœud Z
        sinon si x > clé(B) alors d(B) ← Z
              sinon g(B) ← Z
fin
```

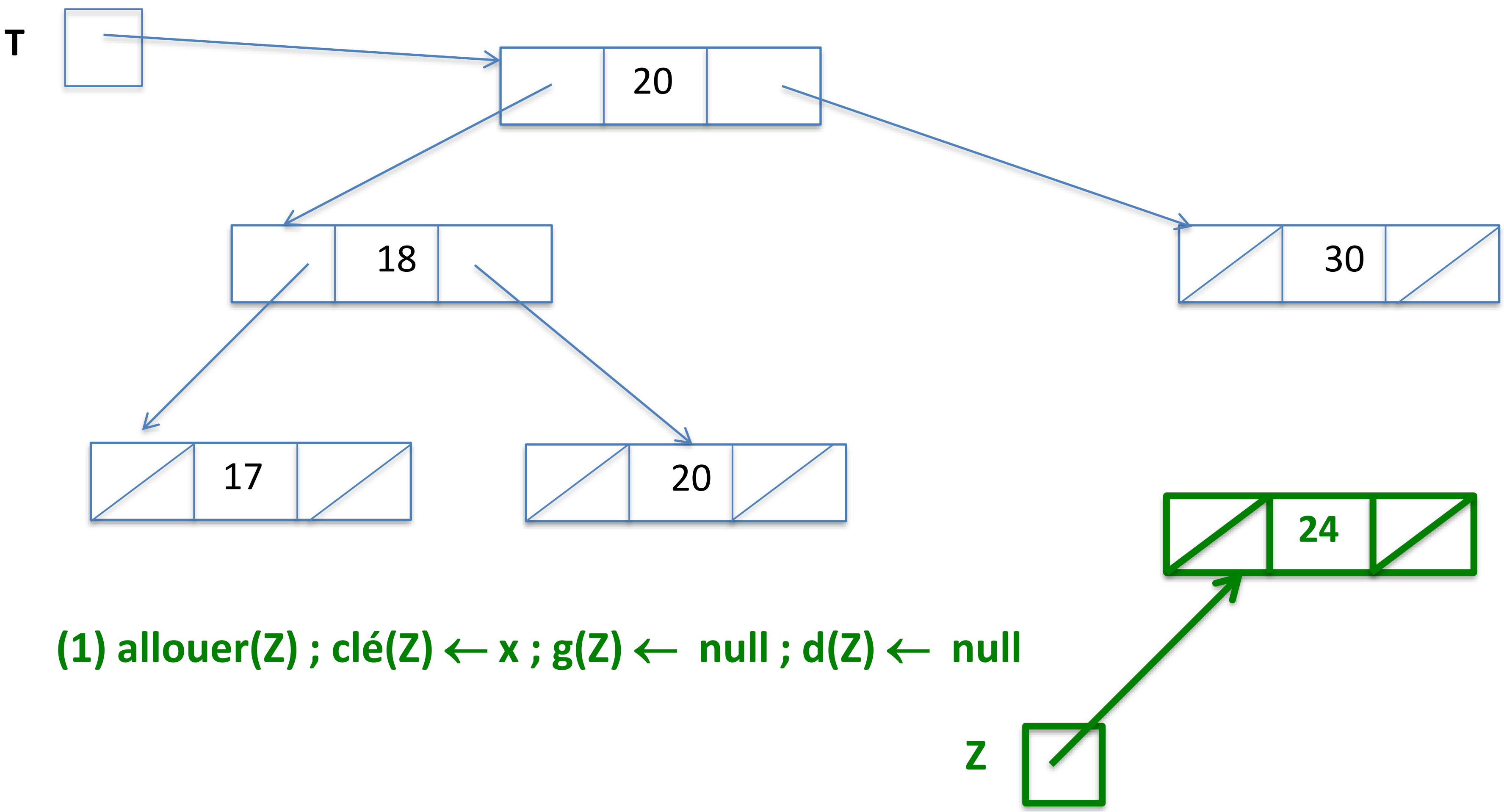
Insertion de **24** dans T

T :





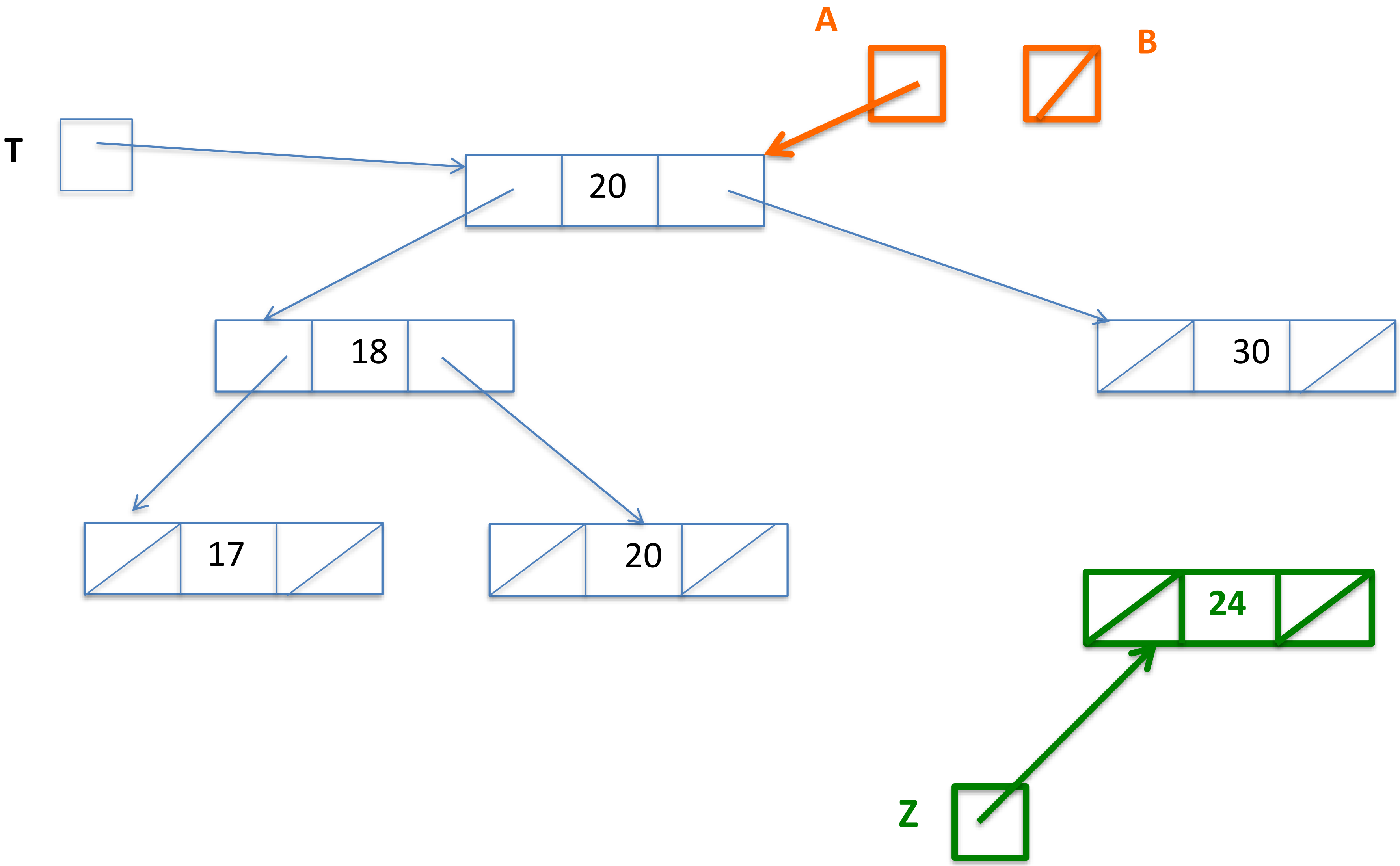
Insertion de **x = 24** dans T



(1) allouer(Z) ; clé(Z) ← x ; g(Z) ← null ; d(Z) ← null

Insertion de **x = 24** dans T

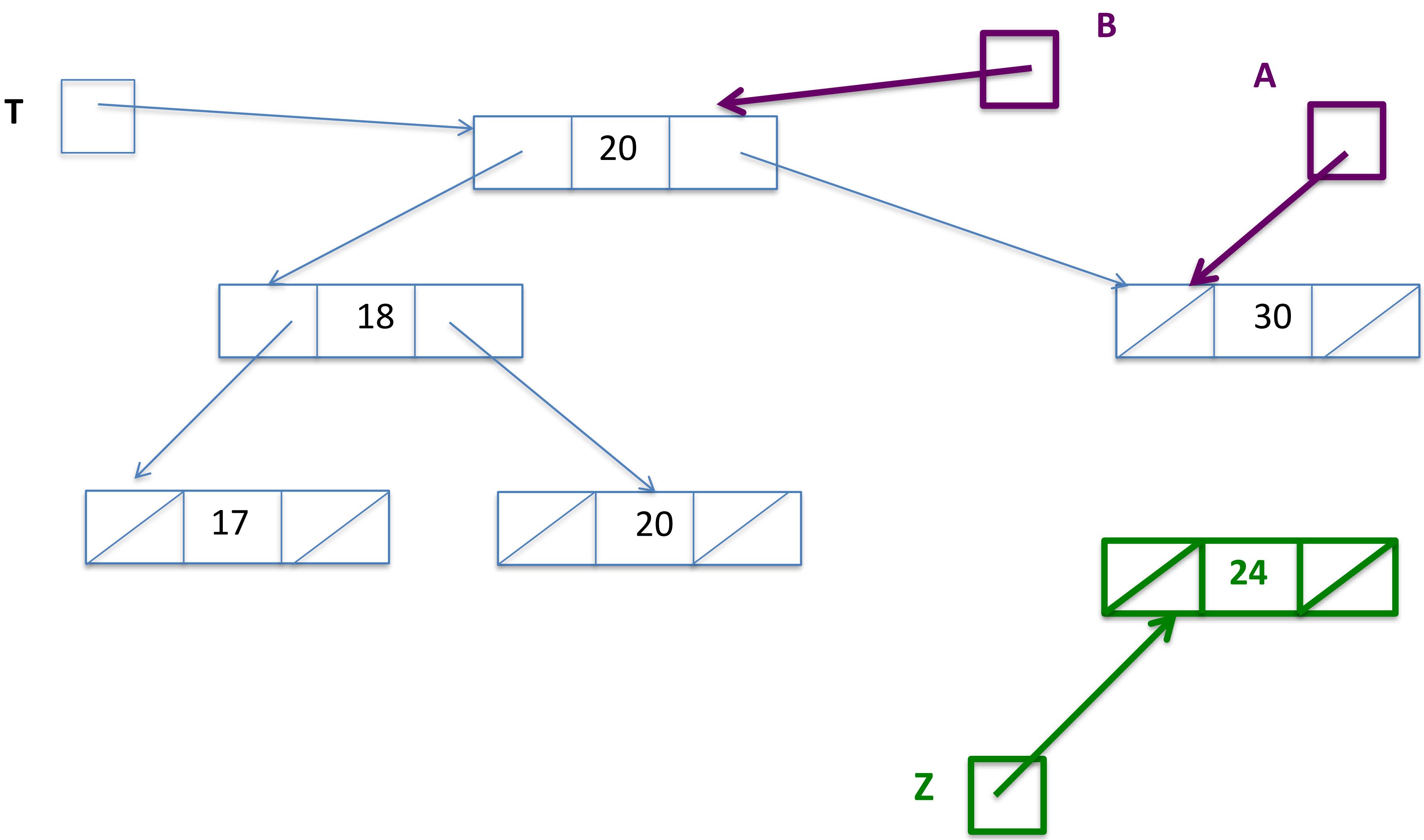
(2) A ← T ; B ← null ;



Insertion de **x = 24** dans T

(3) premier passage dans le tantque :

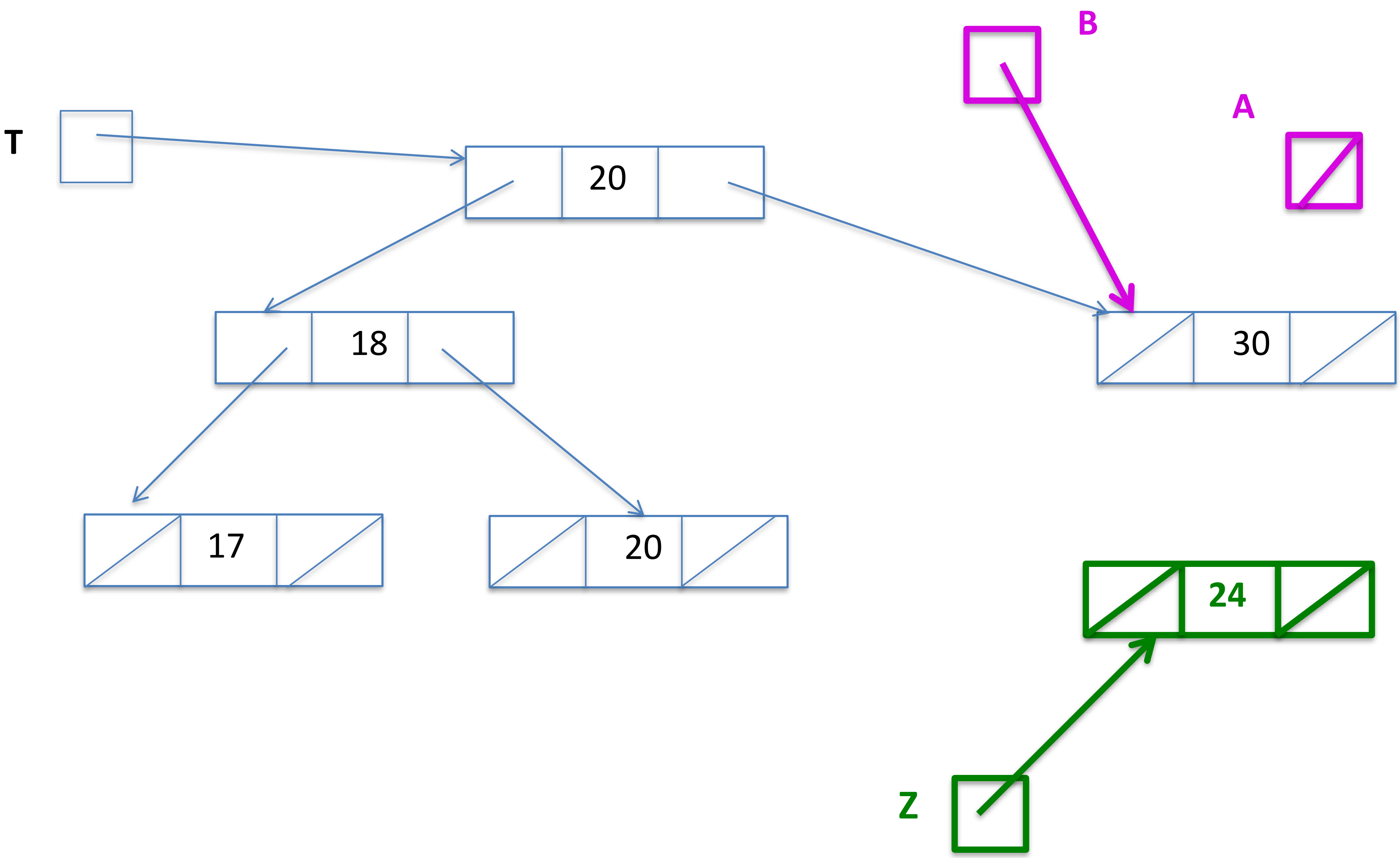
B ← A; si clé(A) < x alors A ← d(A) sinon A ← g(A) finsi



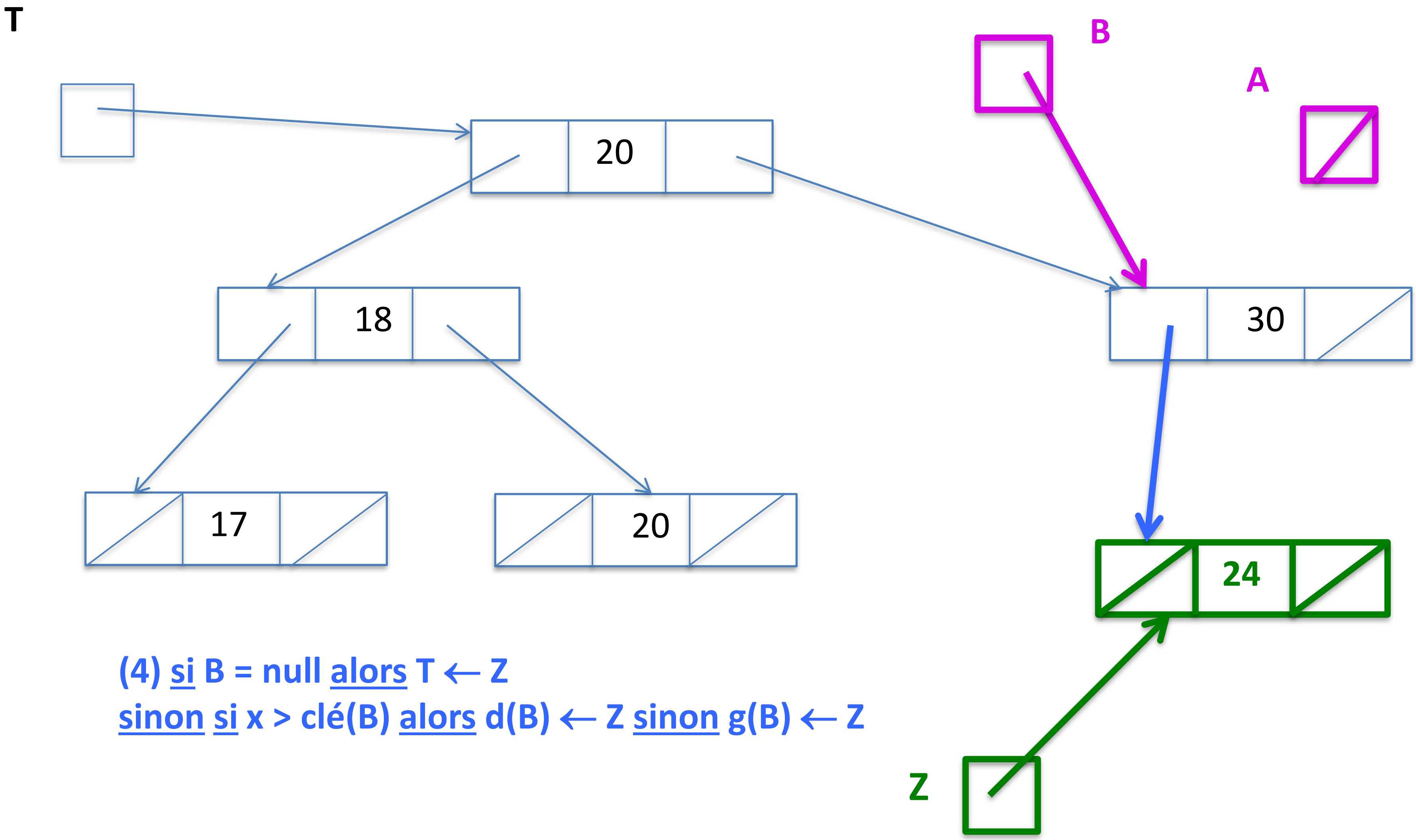
Insertion de **x = 24** dans T

(3) 2ème passage dans le tantque :

B ← A; si clé(A) < x alors A ← d(A) sinon A ← g(A) finsi



Insertion de **x = 24** dans T



(4) si B = null alors T ← Z
sinon si x > clé(B) alors d(B) ← Z sinon g(B) ← Z

Exercice 6 : taille d'un arbre binaire

Ecrire une fonction récursive `compt(A)` qui calcule **le nombre d'éléments d'un arbre binaire** de recherche ou non A.

N.B. Cette fonction ne doit pas utiliser de liste des éléments de l'arbre.

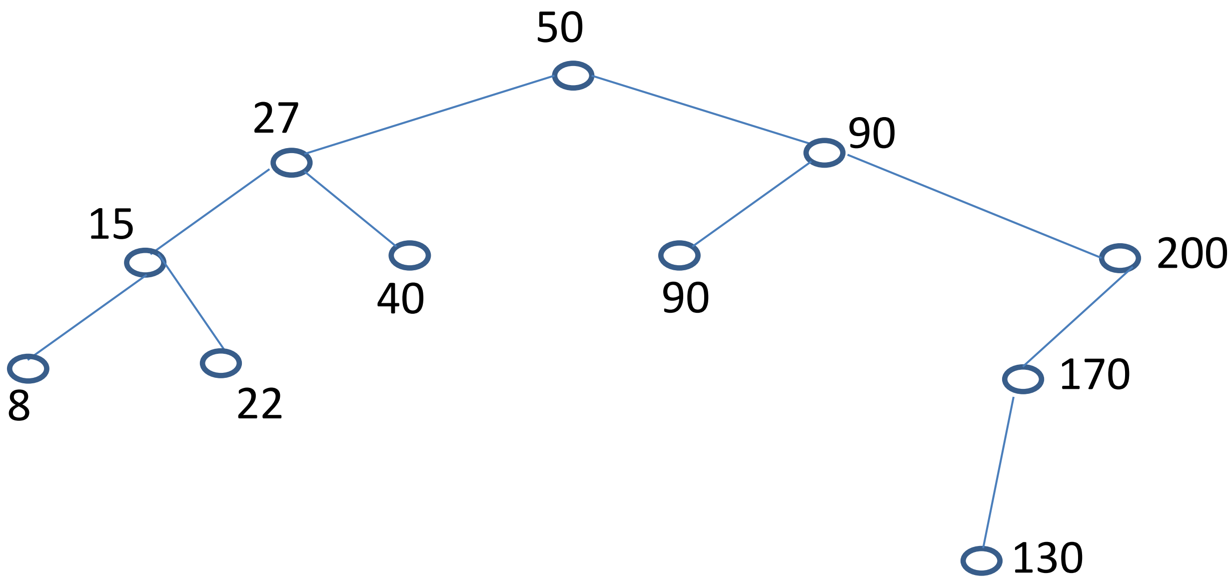
Exercice 7

Nombre d'éléments supérieurs à un élément donné dans un arbre binaire de recherche

2) a) Soit A un **arbre binaire de recherche** et soit X un élément de A, situé à un nœud quelconque de A. Ecrire une fonction récursive qui, étant donnés A et X **retourne le nombre d'éléments de A strictement supérieurs à X**.

(N.B. : cette fonction ne doit pas utiliser de liste des éléments de l'arbre; on compte les répétitions éventuelles ; on peut utiliser la fonction taille écrite précédemment).

Indication : étudier différentes situations pour X sur un exemple d'arbre binaire



1er cas : X = 50

2^{e} cas : X = 90

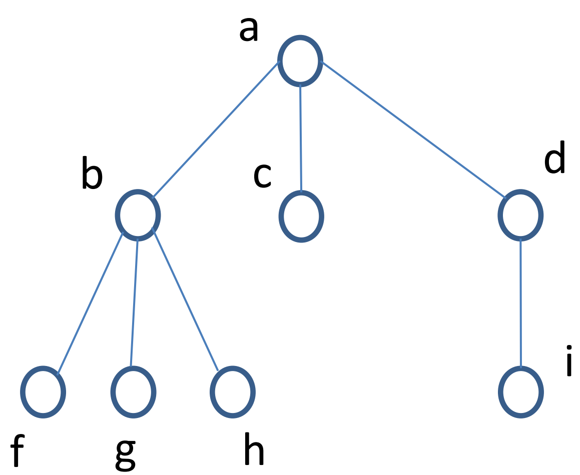
3^{e} cas : X = 22

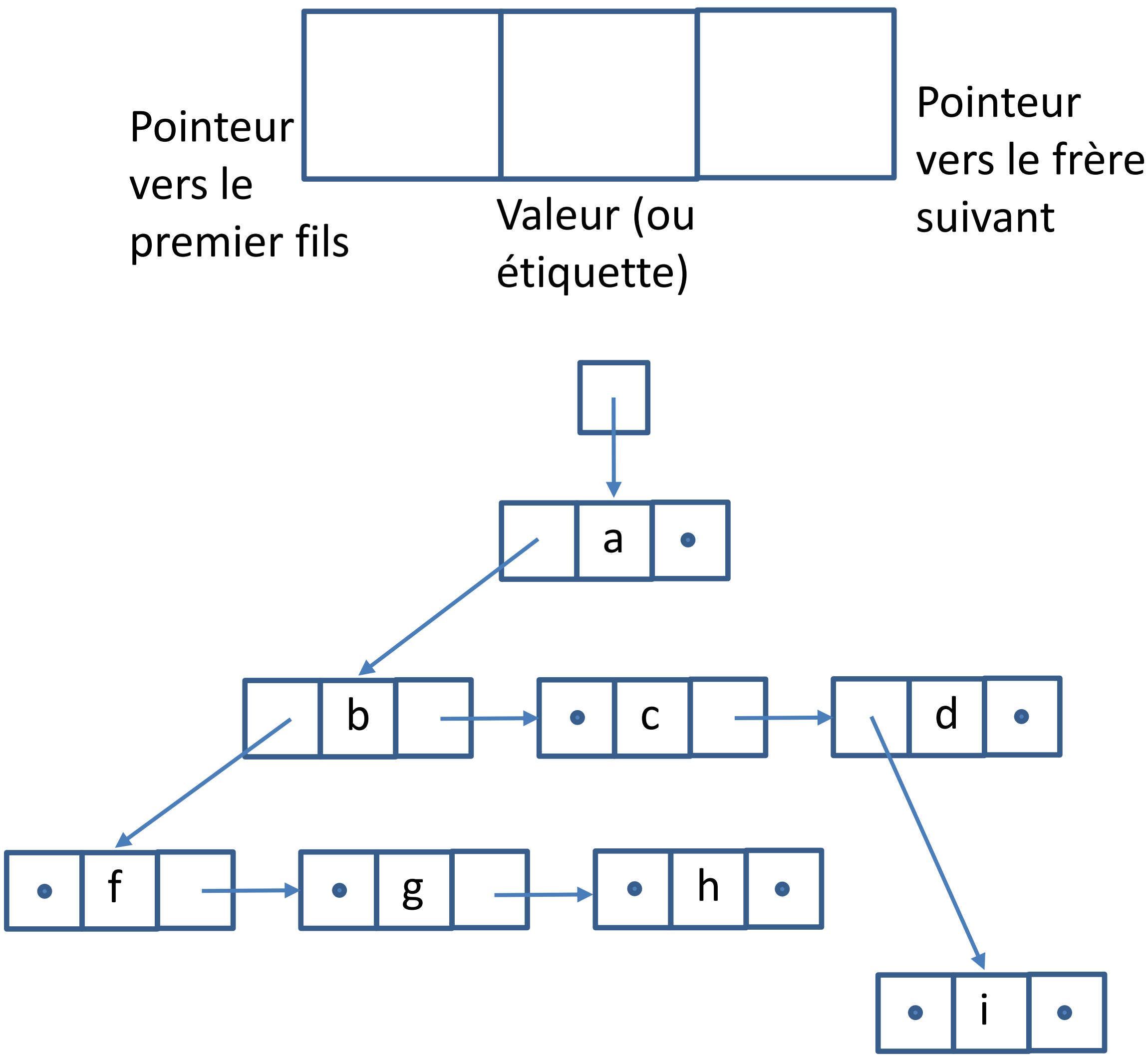
Arbres généraux

- Un ***arbre planaire enraciné*** *est* soit un arbre vide, noté ∅, soit un couple de la forme A = < o, L >, o est un ***nœud*** appelé ***racine***, et L est une **liste ordonnée** d'arbres planaires enracinés disjoints.
  - Vocabulaire : premier fils, frères droits, frère droit suivant, frères gauches, frère gauche précédent.

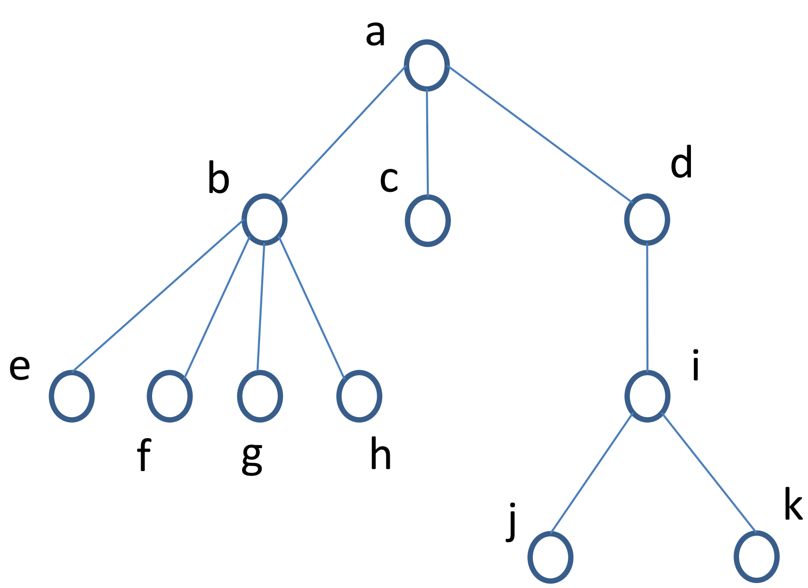
- Un ***arbre libre enraciné*** *est* soit un arbre vide, noté ∅, soit un couple de la forme A = < o, E >, o est un ***nœud*** appelé ***racine***, et E est un **ensemble** d'arbres libres enracinés disjoints.

Représentation chaînée des arbres généraux





Parcours des arbres généraux



En profondeur à main gauche, préfixe :

a, b, e, f, g, h, c, d, i, j, k.

En profondeur à main gauche, suffixe :

e, f, g, h, b, c, j, k, i, d, a.

En largeur à main gauche :

a, b, c, d, e, f, g, h, i, j, k.

Parcours en profondeur

Procédure parcours_prof(A (I) : ARBRE)
// parcours en profondeur à main gauche
<u>Début</u>
    <u>si</u> A ≠ NULL <u>alors</u> {
        traitement 1 */* préfixe */*
        pour tout fils B de A faire
            parcours_prof(B) ;
        traitement 2 */*suffixe */*
    }
<u>Fin</u>

Parcours en profondeur

```
Procédure parcours_prof(A (I) : ARBRE)
// parcours en profondeur à main gauche
Début
        si A ≠ NULL alors {
                traitement 1 /* préfixe */
                B := premier_fils(A)
                faire
                        parcours_prof(B)
                        B := frère_suivant(B)
                tant que B ≠ NULL
                traitement 2 /*suffixe */
                }
Fin
```

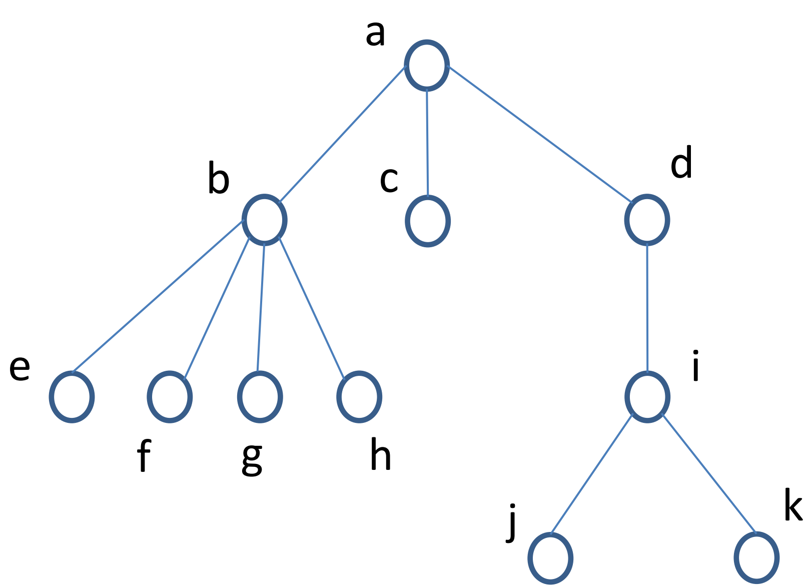
Parcours en largeur

Procédure parcours_largeur(A (I) : ARBRE)
// parcours en largeur à main gauche
<u>Lexique</u> F : FILE
<u>Début</u>

```
        CreerFileVide(F)
        Enfiler(A,F)
        tant que non EstVide(F) faire {
                X := Tête(F)
                Défiler(F)
                traitement
                pour tout fils Y de X faire Enfiler (Y,F)
                }
```

<u>Fin</u>

Exercice 8



Exécuter l'algorithme de parcours en largeur sur l'arbre ci-contre, en montrant l'évolution de la file.

On considèrera que le traitement consiste à afficher l'étiquette du sommet courant (X).

Un peu de combinatoire des arbres… et de bioinformatique

CORRIGÉS DES EXERCICES

Propriété

- **Propriété** : La lecture en ***ordre symétrique (ou infixe)*** d'un ABR respecte l'ordre, cad, donne la liste des éléments en ordre croissant (au sens large).
- **Exercice 1** : pour B1, l'ordre symétrique donne :

  10, 25, 26, 27, 30, 40, 42, 44, 47, 49, 50

Exercice 2 : Recherche récursive d'un élément dans un ABR

Fonction rech-ABR(B : ARBRE (IN); x : entier (IN)) → ARBRE ;

// la fonction récursive retourne null s'il n'y a pas d'occurrence de x dans B et sinon un pointeur sur un nœud dont la clé vaut x

Début

si B est null ou cle(B) = x alors retourner(B) // ou séquentiel

sinon // B ≠ null **et** cle(B) ≠ x

{

si clé(B) < x alors retourner rech-ABR(d(B), x)

sinon retourner rech-ABR(g(B), x)

}

Fin

Exercice 3 : Recherche itérative d'un élément dans un ABR

Fonction rech-ABR_iter(B : ARBRE (IN); x : entier (IN))→ ARBRE

// la fonction itérative retourne null s'il n'y a pas d'occurrence de x dans B et sinon un pointeur sur un nœud dont la clé vaut x

Lexique : A : ARBRE

Début

A ← B ;

Tant que A ≠ null et cle(A) ≠ x faire // et séquentiel

si clé(A) < x alors A ← d(A)

sinon A ← g(A)

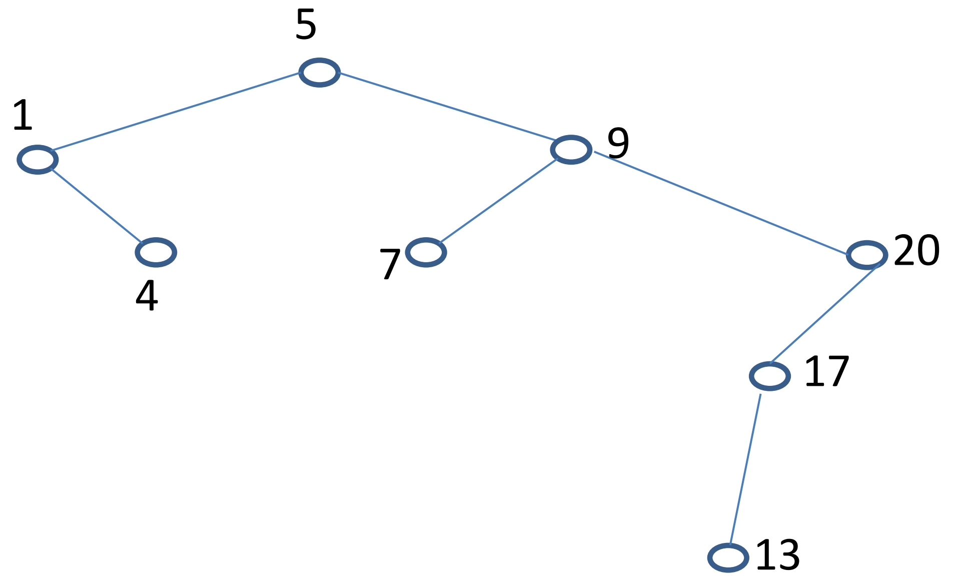
fintq //A = null ou cle(A) = x

Retourner (A)

Fin

Exercice 4 : adjonction aux feuilles

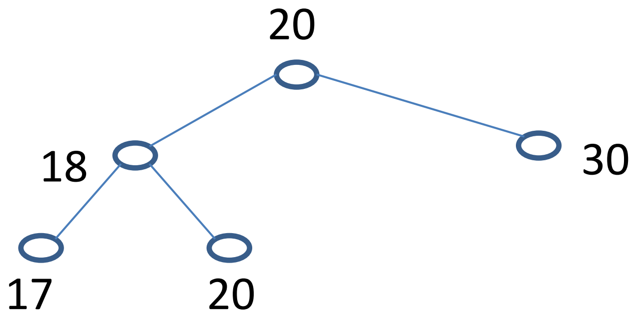
ABR (arbre binaire de recherche) obtenu par adjonctions successives aux feuilles des entiers suivants : 5, 1, 4, 9, 7, 20, 17, 13 :



Exercice 5 : adjonction aux feuilles

Indiquer toutes les listes possibles d'éléments telles que l'adjonction aux feuilles itérée donne l'ABR A :

A :



Indications :

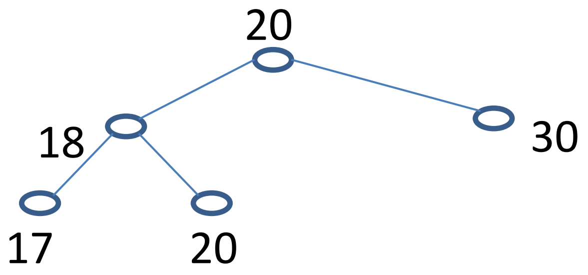
1) la racine de l'ABR A étant 20, toutes les listes doivent commencer par 20

2) Le 2ème élément de la liste donne le fils gauche ou le fils droit selon que sa clé est plus grande ou plus petite que la clé de la racine

Exercice 5 : adjonction aux feuilles

Indiquer toutes les listes possibles d'éléments telles que l'adjonction aux feuilles itérée donne l'ABR A:

A :



On obtient 8 listes :

(20, 30, 18, 17, 20) // une fois 20 et 30 insérés, 18 doit être inséré

(20, 30, 18, 20, 17)

//une fois 20 et 18 insérés, peu importe l'ordre dans lequel on range 17, 20 et 30)

(20, 18, 30, 17, 20)

(20, 18, 30, 20, 17)

(20, 18, 17, 20, 30)

(20, 18, 17, 30, 20)

(20, 18, 20, 30, 17)

(20, 18, 20, 17, 30)

Exercice 6 : taille d'un arbre binaire

```
Fonction taille(A : ARBRE (IN)) retourne entier ;
Début
        Si A = null alors retourner 0
        sinon retourner (1 + taille(g(A)) + d(A))
Fin ;
```

Exercice 7

Nombre d’éléments supérieurs à un élément donné dans un arbre binaire de recherche

```
Fonction nbeltsup(A: ARBRE (IN), x : entier (IN)) retourne entier ;
//On suppose que A est un arbre binaire de recherche dont les clés sont des
entiers ; on utilise la fonction taille de l’exo 1
Début
        si A = null alors retourner 0 ;
        si x = clé(A) alors retourner taille(d(A)) ;
        si x > clé(A) alors retourner (nbeltsup(d(A), x)
        sinon  // x < clé(A)
                retourner (1 + taille(d(A)) + nbeltsup(g(A), x) ;
Fin ;
```