Chapitre 3
Les arbres

Michaël Krajecki

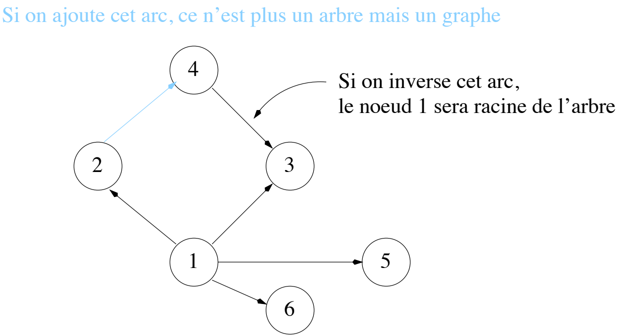
Université de Reims Champagne-Ardenne
michael.krajecki@univ-reims.fr
http://www.univ-reims.fr/crestic

Graphes et algorithmes

Les arbres

- Les arbres de recherche sont des structures de données adaptées à la représentation des ensembles dynamiques.
- les opérations *Rechercher*, *Inserer*, *Supprimer*, *Successeur* ont un coût proportionnel à la hauteur de l'arbre.
- les arbres sont une généralisation des listes linéaires ;
- il sont également un cas particulier de graphe : c'est un graphe connexe, sans cycle.
- s'il existe un nœud d'où partent des chemins en direction de tous les nœuds, alors ce nœud est appelé *racine*.

Exemple



Un arbre muni d'une racine est parfois appelé *arborescence*.

Quelques définitions sur les arbres

Arbre n-aire : tout père a au plus n fils.

Arbre binaire : tout père a au plus 2 fils.

Niveau d'un nœud : 1+ le nombre d'arcs jusqu'à la racine.

Taille d'un arbre : nombre de nœuds.

Une feuille est un nœud qui n'a pas de fils.

Hauteur d'un nœud : 1+ le nombre d'arcs jusqu'à la feuille la plus éloignée.

Hauteur d'un arbre : hauteur de la racine.

Définition (Définition récursive des arbres binaires)

$A_2(V) = \{$ *arbres binaires sur* $V\} = \{()\} \cup \{a : (Fg, Fd)$ *pour* $a \in V$ *et* $Fg, Fd \in A_2(V)\}$.

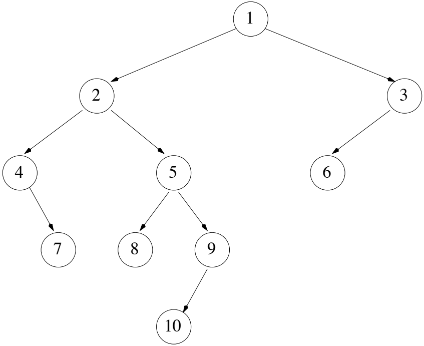
Les différents parcours d'un arbre binaire

Remarque : toute arbre n-aire peut être représenté par un arbre binaire.

Il existe 4 parcours différents pour un arbre binaire :

1. Parcours préfixé : étude du nœud, puis parcours préfixé du fils gauche et enfin du fils droit.
2. Parcours postfixé : parcours postfixé du fils gauche, puis du fils droit et enfin évaluation du nœud.
3. Parcours infixé : parcours infixé du fils gauche, puis évaluation du nœud et enfin parcours infixé du fils droit.
4. Parcours en largeur : les nœuds sont visités par niveaux (hauteur décroissante des nœuds) de gauche à droite.

Exemple



Parcours préfixé : 1, 2, 4, 7, 5, 8, 9, 10, 3, 6.

Parcours postfixé : 7, 4, 8, 10, 9, 5, 2, 6, 3, 1.

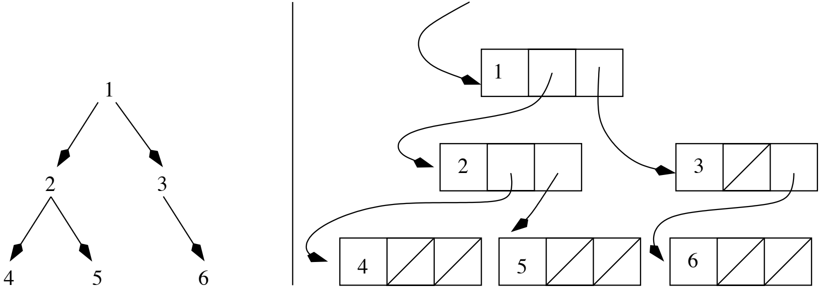
Parcours infixé : 4, 7, 2, 8, 5, 10, 9, 1, 6, 3.

Parcours en largeur : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Plan

Les arbres binaires

Rappel : tout nœud d'un arbre binaire a, au plus, deux fils.
Nous nous proposons de représenter dans cette partie les arbres binaires par chaînage.



Définition d'un nœud

- La structure *noeud* contiendra les informations suivantes :
  - une valeur (appelée *clé*) du type de base (ici un entier),
  - un lien vers le fils gauche
  - un lien vers le fils droit

```
typedef struct noeud {
   int valeur;
   struct noeud *fg, *fd;
} noeud;
typedef noeud* Arbre;
```

- Un arbre est alors défini comme un lien vers un nœud.
- Remarque : cette représentation est minimaliste : il est parfois utile de prévoir un lien vers le père...

Les primitives d'accès

Pour manipuler proprement votre TDA *arbres binaires*, vous devez définir au minimum les actions en consultation et en création suivantes :

Consultation :

1. *EstVide(Arbre) : Booléen* ;
2. *Clé(Arbre) : type_base* ;
3. *FilsGauche(Arbre) : Arbre* ;
4. *FilsDroit(Arbre) : Arbre*.

Création :

1. *ArbreVide() : Arbre* ;
2. *ConsArbre(int, Arbre, Arbre) : Arbre.*

Les primitives d'accès

Vous pouvez également prévoir des actions en modification et en destruction :

Modification :

1. *ModifClé(int, Arbre) : Arbre* ;
2. *ModifFilsGauche(Arbre, Arbre) : Arbre* ;
3. *ModifFilsDroit(Arbre,Arbre) : Arbre*.

Destruction :

DetruireArbre(Arbre) ;

Parcours préfixé

Parcours préfixé : étude du nœud, puis parcours préfixé du fils gauche et enfin du fils droit.

Procédure *ParcoursPrefixe(Arbre A).*
Début
    Si *non EstVide(A)* ***alors***
      ecrire*(Cle(A))*
      ParcoursPrefixe(FilsGauche(A))
      ParcoursPrefixe(FilsDroit(A))
    Fsi
Fin.

Parcours postfixé

Parcours postfixé : parcours postfixé du fils gauche, puis du fils droit et enfin évaluation du nœud.

Procédure *ParcoursPostfixe(Arbre A).*
Début
  Si *non EstVide(A)* ***alors***
    ParcoursPostfixe(FilsGauche(A))
    ParcoursPostfixe(FilsDroit(A))
    ecrire*(Cle(A))*
  Fsi
Fin.

Parcours infixé

Parcours infixé : parcours infixé du fils gauche, puis évaluation du nœud et enfin parcours infixé du fils droit.

Procédure *ParcoursInfixe(Arbre A).*
Début
  Si *non EstVide(A)* ***alors***
    ParcoursInfixe(FilsGauche(A))
    ecrire*(Cle(A))*
    ParcoursInfixe(FilsDroit(A))
  Fsi
Fin.

Évaluation de la taille d'un arbre

Taille d'un arbre : nombre de nœuds.

Fonction *Taille(Arbre A) : entier.*
Début
  Si *EstVide(A)* ***alors*** *Taille* $\leftarrow$ *0*
  sinon
    Taille $\leftarrow$ *1+ Taille(FilsGauche(A))+*
    Taille(FilsDroit(A))) ;
    Fsi
Fin.

Évaluation de la hauteur d'un arbre

Hauteur d'un nœud : 1+ le nombre d'arcs jusqu'à la feuille la plus éloignée.

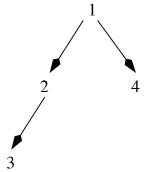
Fonction *Hauteur(Arbre A) : entier.*
variable *: hg, hd : entier*
Début
  Si *EstVide(A)* ***alors*** *Hauteur* $\leftarrow$ *0*
  sinon
    hg $\leftarrow$ *Hauteur(FilsGauche(A))*
    hd $\leftarrow$ *Hauteur(FilsDroit(A))*
    Si *hg>hd* ***alors*** *Hauteur* $\leftarrow$ *1+hg*
    sinon *Hauteur* $\leftarrow$ *1+hd*
    Fsi
  Fsi
Fin.

Exemple

Algorithme *Arbre.*
déclaration :
variable : *a* : *Arbre*
Début
a ← *ConsArbre(1,ConsArbre(2,*
ConsArbre(3,ArbreVide(),ArbreVide()),
ArbreVide()),
ConsArbre(4,ArbreVide(),ArbreVide())) *ParcoursPrefixe(a)*
ParcoursInfixe(a)
ParcoursPostfixe(a)
ecrire(*"Taille =",Taille(a)*)
ecrire(*"Hauteur=",Hauteur(a)*)
Fin.

Exemple

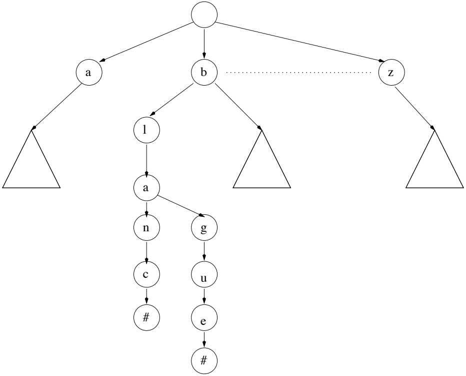
- Voici l'arbre construit :



- Et voici le résultat :
  1 2 3 4
  3 2 1 4
  3 2 4 1
  Taille(a)=4, Hauteur(a)=3

Intérêt des arbres

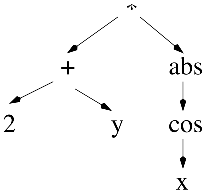
Le partage des données : les mots consécutifs d'un dictionnaire ont le même préfixe $\rightsquigarrow$ éviter de stocker plusieurs fois les lettres communes.



Expressions arithmétiques

Représentation des expressions arithmétiques par un arbre syntaxique :

- les nœuds intérieurs représentent des opérateurs (binaires ou unaires) ;
- les feuilles sont des valeurs.



Plan

Expression arithmétique et arbre binaire

- Nous supposons que seuls les opérateurs + - * / sont définis.
- Il existera deux types de nœuds :
  - un nœud peut contenir la valeur d'une opérande gauche ou droite ;
  - il peut également contenir un opérateur plus deux liens vers les opérandes.
- Une expression sera définie comme un pointeur vers un nœud.

La structure de données

Certains champs seront invalides suivant le type du nœud.

```
typedef struct noeud{
  BOOLEEN type;
  /* VRAI= opérateur, FAUX = opérande */
  char operateur; /* valide si type=VRAI */
  float valeur; /* valide si type=FAUX */
  struct noeud *fg, *fd;
} noeud;
typedef noeud* Expression;
```

Actions en consultation et création

- *EstVide(Expression E) : Booléen*
- *EstOperateur(Expression E) : Booléen*
- *OperandeGauche(Expression E) : Expression*
- *OperandeDroite(Expression E) : Expression*
- *Valeur(Expression E) : Réel*
- *Operateur(Expression E) : Caractère*
- *CreerOperande(float v) : Expression*
- *CreerOperateur(char o, Expression g, Expression d) : Expression*

Notation infixée

- Les parenthèses sont indispensables, pourquoi ?

```
Procédure NotationInfixe(Expression E).
Début
 Si non EstVide(E) alors
  Si EstOperateur(E) alors
   ecrire("(")
   NotationInfixe(OperandeGauche(E))
   ecrire(Operateur(E))
   NotationInfixe(OperandeDroite(E))
   ecrire(")")
  sinon ecrire(Valeur(E))
  Fsi
 Fsi
Fin.
```

La notation postfixée

Procédure *NotationPostfixe(Expression E).*
Début
  Si *non EstVide(E)* ***alors***
    Si *EstOperateur(E)* ***alors***
      NotationPostfixe(OperandeGauche(E))
      NotationPostfixe(OperandeDroite(E))
      ecrire*(Operateur(E))*
    sinon ecrire*(Valeur(E))*
    Fsi
  Fsi
Fin.

Evaluation d'une expression

Fonction *Evalue(Expression E) : réel.*
variable *: g,d : réel*
Début
  Si *EstVide(E)* ***alors*** *Evalue* $\leftarrow$ *0*
  sinon Si *non EstOperateur(E)* ***alors*** *Evalue* $\leftarrow$ *Valeur(E)*
  sinon
    g $\leftarrow$ *Evalue(OperandeGauche(E))*
    d $\leftarrow$ *Evalue(OperandeDroite(E))*
    Cas *Operateur(E)* **parmi**
      '+' : Evalue $\leftarrow$ *g+d*
      '-' : Evalue $\leftarrow$ *g-d*
      '' : Evalue* $\leftarrow$ *g*d*
      '/' : Evalue $\leftarrow$ *g/d*
    Fcas
  Fsi
Fin.

Exemple

- Voici un exemple :

Algorithme *Expression.*
déclaration *:*
variable *: a, b, e : Expression*
Début
  a ← CreerOperateur('+',CreerOperande(2.2)
    CreerOperande(3.3))
  b ← CreerOperande(7.7)
  e ← CreerOperateur('',a,b)*
  NotationInfixe(e)
  NotationPostfixe(e)
  ecrire(*Evalue(e)*)
Fin.

Exemple

- Et le résultat :

*((2.20 + 3.30) * 7.70)*
*2.20 3.30 + 7.70 **
42.349999

Plan

1. Introduction
2. Définitions
3. Les arbres binaires
   - Représentation par chaînage
4. Intérêt des arbres
   - Expression arithmétique et arbre binaire
   - Le codage de Huffman
5. Arbres binaires de recherche

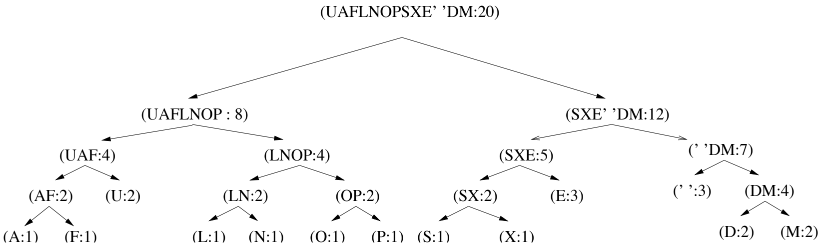
Le codage de Huffman

- Le but du codage de Huffman est de représenter en machine un message de manière plus concise qu'en utilisant la table ASCII par exemple.
- Cette méthode de codage trouve de nombreuses applications en particulier dans le domaine des télécommunications où elle peut être utilisée pour réduire la taille des paquets circulant dans les réseaux.
- Le codage de Huffman consiste à choisir un code dit «préfixe» qui tient compte des fréquences relatives des symboles intervenant dans les messages à coder.

Représentation par un arbre

- Ce code peut être représenté par un arbre binaire dont :
  - les feuilles sont les symboles à coder avec leur fréquence relative ;
  - les nœuds intérieurs sont les ensembles de symboles se trouvant en dessous avec la somme de leur fréquence.
- Soit le message suivant : `EXAMEN DU MODULE PSF`
- Les fréquences associées à chaque symbole sont :
  `E` : 3, `X` : 1, `A` : 1, `M` : 2, `N` : 1, `D` : 2, `U` : 2, `O` : 1, `L` : 1, `P` : 1, `S` : 1, `F` : 1.
  Il manque le caractère ' ' (blanc) qui a une fréquence de 3.
  La somme des fréquences est égales à 20 (la longueur du message).

Représentation par un arbre



Coder un message

- Étant donné un arbre de Huffman, le code de chaque symbole est calculé en partant de la racine de l'arbre et en descendant jusqu'à la feuille contenant ce symbole.
- À chaque fois que l'on descend par une branche gauche on ajoute un 0 au code, par une branche droite un 1.
- Sur l'exemple précédent, le code associé au symbole `U` est `001`, de même `1000` est le code correspondant à `S`.

Décoder une suite de 0 et 1

- Étant donnée une suite de 0,1 à décoder grâce à un arbre de Huffman :
  1. On part de la racine.
  2. On utilise les 0,1 de la suite successivement pour descendre soit à gauche (0), soit à droite (1).
  3. Lorsqu'on a atteint une feuille, un symbole du message est généré et l'on repart de la racine pour décoder le reste de la suite de 0,1.
- Sur l'exemple : la suite 111010110001000 code le message DESS.

Construction d'un arbre de Huffman

- L'arbre de Huffman est construit à partir d'une liste de couples (symbole, fréquence).
- Le principe est de mettre le plus loin de la racine (au plus profond dans l'arbre) les symboles les moins fréquents.
  1. On commence avec la liste des feuilles dans l'ordre croissant de leur fréquence.
  2. On choisit les deux premiers nœuds (*ie* qui ont une fréquence minimale) a_1, a_2.
  3. a_3 est le nouveau nœud ayant a_1 pour fils gauche et a_2 pour fils droit. La fréquence associée à a_3 est la somme des fréquences de a_1 et de a_2.
  4. On insère a_3 au bon endroit dans la liste et on élimine a_1, a_2 de la liste.

Construction d'un arbre de Huffman

- Sur l'exemple, on obtient :
  1. { (`A` : 1) (`F` : 1) (`L` : 1) (`N` : 1) (`O` : 1) (`P` : 1) (`S` : 1) (`X` : 1) (`D` : 2) (`M` : 2) (`U` : 2) (`E` : 3) (' ' : 3) }

Construction d'un arbre de Huffman

- Sur l'exemple, on obtient :
  1. { (A : 1) (F : 1) (L : 1) (N : 1) (O : 1) (P : 1) (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (E : 3) (' ' : 3) }
  2. { (L : 1) (N : 1) (O : 1) (P : 1) (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (AF : 2) (E : 3) (' ' : 3) }

Construction d'un arbre de Huffman

- Sur l'exemple, on obtient :
  1. { (A : 1) (F : 1) (L : 1) (N : 1) (O : 1) (P : 1) (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (E : 3) (' ' : 3) }
  2. { (L : 1) (N : 1) (O : 1) (P : 1) (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (AF : 2) (E : 3) (' ' : 3) }
  3. { (O : 1) (P : 1) (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (AF : 2) (LN : 2) (E : 3) (' ' : 3) }

Construction d'un arbre de Huffman

- Sur l'exemple, on obtient :
  1. { (A : 1) (F : 1) (L : 1) (N : 1) (O : 1) (P : 1) (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (E : 3) (' ' : 3) }
  2. { (L : 1) (N : 1) (O : 1) (P : 1) (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (AF : 2) (E : 3) (' ' : 3) }
  3. { (O : 1) (P : 1) (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (AF : 2) (LN : 2) (E : 3) (' ' : 3) }
  4. { (S : 1) (X : 1) (D : 2) (M : 2) (U : 2) (AF : 2) (LN : 2) (OP : 2) (E : 3) (' ' : 3) }

Construction d'un arbre de Huffman

- On continue...

1. { (D : 2) (M : 2) (U : 2) (AF : 2) (LN : 2) (O' 'P : 2) (SX : 2) (E : 3) (' ' : 3) }

Construction d'un arbre de Huffman

- On continue...

1. { (D : 2) (M : 2) (U : 2) (AF : 2) (LN : 2) (O' 'P : 2) (SX : 2) (E : 3) (' ' : 3) }
2. { (U : 2) (AF : 2) (LN : 2) (OP : 2) (SX : 2) (E : 3) (' ' : 3) (DM : 4) }

Construction d'un arbre de Huffman

- On continue...

1. { (D : 2) (M : 2) (U : 2) (AF : 2) (LN : 2) (O' 'P : 2) (SX : 2) (E : 3) (' ' : 3) }
2. { (U : 2) (AF : 2) (LN : 2) (OP : 2) (SX : 2) (E : 3) (' ' : 3) (DM : 4) }
3. { (LN : 2) (OP : 2) (SX : 2) (E : 3) (' ' : 3) (DM : 4) (UAF : 4) }

Construction d'un arbre de Huffman

- On continue...

1. { (D : 2) (M : 2) (U : 2) (AF : 2) (LN : 2) (O' 'P : 2) (SX : 2) (E : 3) (' ' : 3) }
2. { (U : 2) (AF : 2) (LN : 2) (OP : 2) (SX : 2) (E : 3) (' ' : 3) (DM : 4) }
3. { (LN : 2) (OP : 2) (SX : 2) (E : 3) (' ' : 3) (DM : 4) (UAF : 4) }
4. { (SX : 2) (E : 3) (' ' : 3) (DM : 4) (UAF : 4) (LNOP : 4) }

Construction d'un arbre de Huffman

- On continue...

1. { (D : 2) (M : 2) (U : 2) (AF : 2) (LN : 2) (O’ ’P : 2) (SX : 2) (E : 3) (' ' : 3) }
2. { (U : 2) (AF : 2) (LN : 2) (OP : 2) (SX : 2) (E : 3) (' ' : 3) (DM : 4) }
3. { (LN : 2) (OP : 2) (SX : 2) (E : 3) (' ' : 3) (DM : 4) (UAF : 4) }
4. { (SX : 2) (E : 3) (' ' : 3) (DM : 4) (UAF : 4) (LNOP : 4) }
5. { (' ' : 3) (DM : 4) (UAF : 4) (LNOP : 4) (SXE : 5) }

Construction d'un arbre de Huffman

- Et finalement :

  1. { (UAF : 4) (LNOP : 4) (SXE : 5) (' 'DM : 7) }

Construction d'un arbre de Huffman

- Et finalement :

1. { (UAF : 4) (LNOP : 4) (SXE : 5) (' 'DM : 7) }
2. { (SXE : 5) (' 'DM : 7) (UAFLNOP : 8) }

Construction d'un arbre de Huffman

- Et finalement :

1. { (UAF : 4) (LNOP : 4) (SXE : 5) (' 'DM : 7) }
2. { (SXE : 5) (' 'DM : 7) (UAFLNOP : 8) }
3. { (UAFLNOP : 8) (SXE' 'DM : 12) }

Construction d'un arbre de Huffman

- Et finalement :

1. { (UAF : 4) (LNOP : 4) (SXE : 5) (' 'DM : 7) }
2. { (SXE : 5) (' 'DM : 7) (UAFLNOP : 8) }
3. { (UAFLNOP : 8) (SXE' 'DM : 12) }
4. { (UAFLNOPSXE' 'DM : 20) }

Structure de données

- Pour construire cet arbre, il est nécessaire de proposer une structure de données adaptées à la représentation de listes d'arbres.
- Un nœud de l'arbre aura pour clé une chaîne de caractère et une donnée satellite correspondant à la somme des fréquences.
- Des liens vers les fils gauche et droit sont suffisant pour proposer une solution récursive à la construction de l'arbre ainsi qu'à l'encodage et au décodage d'un message.
- L'écriture d'une version itérative est plus délicate et sera facilitée par la définition d'un lien vers le père.

Arbres binaires de recherche

- Également appelés *arbres ordonnés*
- La complexité d'une recherche dans un arbre quelconque dans le pire des cas est n (où n est le nombre de noeuds de l'arbre).
- Cette recherche n'est donc pas plus efficace qu'une recherche dans un tableau non trié.
- Comme pour les tableaux, pour obtenir une recherche efficace, nous devons «trier» les valeurs présentes dans l'arbre.

Arbres binaires de recherche

Définition (Arbre binaire ordonné)

Un arbre binaire est ordonné si et seulement si la liste infixée de ses valeurs est ordonnée.

- Pour être en mesure d'ordonner un arbre binaire, il est donc nécessaire de disposer d'une relation d'ordre sur les informations de l'arbre (réflexive, antisymétrique et transitive).
- Propriété :
  Si $A \equiv v : (Fg, Fd)$ est ordonné alors

$$\forall v' \in Fg,\ v' \leq v$$

$$\forall v'' \in Fd,\ v \leq v''$$

Minimum et maximum

- Il est facile de calculer le minimum d'un arbre de recherche
- Il suffit de rechercher la feuille la plus à gauche :

Fonction *Minimum(Arbre A) : entier.*
variable *: m : entier*
Début
m $\leftarrow$ *0*
Tant que *non EstVide(A)* ***faire***
m $\leftarrow$ *Cle(A)*
A $\leftarrow$ *FilsGauche(A)*
Ftant *Minimum* $\leftarrow$ *m*
Fin.

Maximum

- De même, nous pouvons calculer le maximum :

Fonction *Maximum(Arbre A) : entier.*
variable *: m : entier*
Début
  m $\leftarrow$ *0*
  Tant que *non EstVide(A)* ***faire***
    m $\leftarrow$ *Cle(A)*
    A $\leftarrow$ *FilsFroit(A)*
  Ftant *Maximum* $\leftarrow$ *m*
Fin.

Recherche d'une valeur

En fonction de la valeur de la clé, on choisit le fils à explorer...

Fonction *Recherche(v : entier, A : Arbre) : Arbre.*
Début
Si *EstVide(A)* ***alors*** *Recherche* $\leftarrow$ *ArbreVide()*
sinon Si *Cle(A)=v* ***alors*** *Recherche* $\leftarrow$ *A*
sinon Si *Cle(A)>v)* ***alors***
Recherche $\leftarrow$ *Recherche(v, FilsGauche(A))*
sinon *Recherche* $\leftarrow$ *Recherche(v, FilsDroit(A))*
Fsi**Fin.**

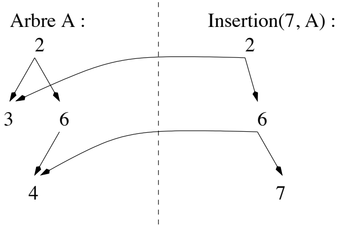
Insertion d'une valeur

Fonction *Insertion (v : entier, A : Arbre) : Arbre.*
Début
  Si *EstVide(A)* ***alors***
    Insertion $\leftarrow$ *ConsArbre(v, ArbreVide(), ArbreVide())*
  sinon Si $v <$ *Cle(A)* ***alors***
    Insertion $\leftarrow$ *ConsArbre(Cle(A), Insertion(v, FilsGauche(A)), FilsDroit(A))*
  sinon *Insertion* $\leftarrow$ *ConsArbre(Cle(A), FilsGauche(A), Insertion(v, FilsDroit(A)))*
  Fsi
Fin.

Insertion d'une valeur

Cette fonction récursive pose quelques problèmes :

- les noeuds du chemin allant de la racine au noeud d'insertion sont recrées, mais pas les autres ;
- l'arbre donné et l'arbre résultat sont donc partiellement superposés ce qui est incohérent.



Une procédure récursive

```
Procédure Insertion2(v : entier, A : Arbre).
variable : Fils : Arbre
Début
  Si EstVide(A) alors    A ←ConsArbre(v, ArbreVide(), ArbreVide())
  sinon Si v < Cle(A) alors
    Fils ←FilsGauche(A)
    Insertion2(v, Fils)
    ModifFilsGauche(A,Fils)
  sinon { v >= Cle(A) }
    Fils ←FilsDroit(A)
    Insertion2(v, Fils)
    ModifFilsDroit(A,Fils)
  Fsi
Fin.
```

Une procédure récursive

- En fait, il faut prévoir 2 fonctions de base pour autoriser la modification des fils gauche et droit d'un arbre...
- Cette procédure évite la reconstruction de certains nœuds.
- La plus grande partie du travail consiste à déterminer le nœud «père».
- Nous pouvons donc écrire une fonction spécifiquement pour cette tâche...

Localisation du père

Fonction *FuturPapa(v : entier, A : Arbre) : Arbre.*
Début
  Si *v<Cle(A)* ***alors***
    Si *EstVide(FilsGauche(A))* ***alors*** *FuturPapa* $\leftarrow$*A*
    sinon *FuturPapa* $\leftarrow$*FuturPapa(v, FilsGauche(A))*
    Fsi
  sinon ***Si*** *EstVide(FilsDroit(A))* ***alors***
  FuturPapa $\leftarrow$*A*
  sinon *FuturPapa* $\leftarrow$*FuturPapa(v, FilsDroit(A))*
  Fsi
Fin.

Encore une procédure d'insertion

- Ensuite, la fonction d'insertion est très simple :

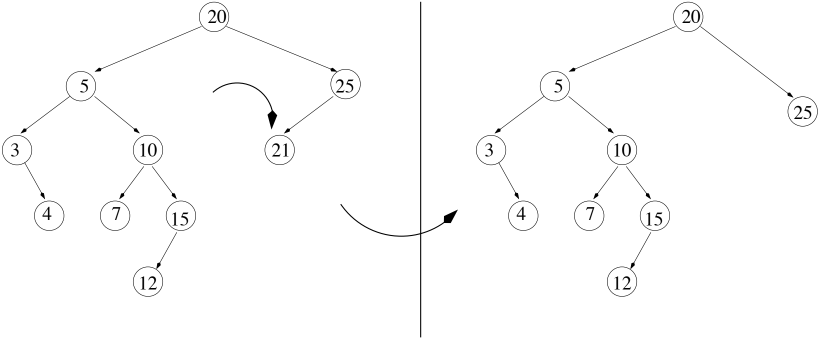
Procédure *Insertion3(v : entier, A : Arbre).*
variable *: bebe, papa : Arbre*
Début
 bebe ←ConsArbre(v, ArbreVide(), ArbreVide())
 Si *EstVide(A)* ***alors*** *A ←bebe*
 sinon
  papa ←FuturPapa(v, A)
  Si *v<Cle(papa)* ***alors*** *ModifFilsGauche(papa, bebe)*
  sinon *ModifFilsDroit(papa, bebe)*
  Fsi
 Fsi
Fin.

Insertion itérative

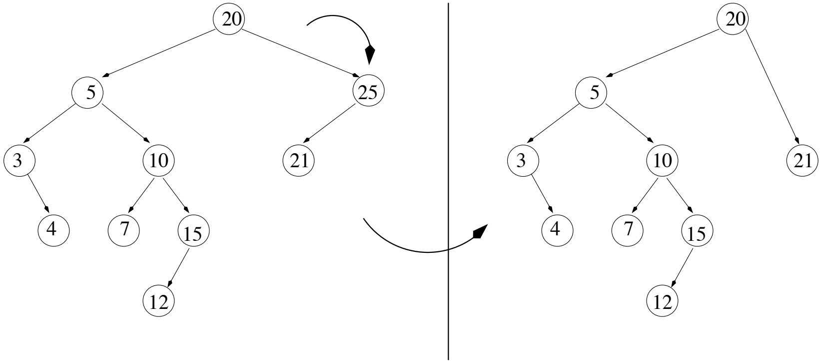
- La remontée induite par la récursivité est inutile

Procédure *Insertion4(v : entier, A : Arbre).*
variable *: pere, n :Arbre ; gauche : booléen*
Début
  Si *EstVide(A)* ***alors*** *A* $\leftarrow$ *ConsArbre(v, ArbreVide(), ArbreVide())*
  sinon
    n $\leftarrow$ *A*
    Tant que non *EstVide(n)* ***faire***
      pere $\leftarrow$ *n*
      Si *v<Cle(n)* ***alors***
        n $\leftarrow$ *FilsGauche(n) ; gauche* $\leftarrow$ *VRAI*
      sinon
        n $\leftarrow$ *FilsDroit(n) ; gauche* $\leftarrow$ *FAUX*
      Fsi
    Ftant
  Fsi
  Si *gauche* ***alors*** *ModifFilsGauche(pere, ConsArbre(v, ArbreVide(), ArbreVide()))*
  sinon *ModifFilsDroit(pere, ConsArbre(v, ArbreVide(), ArbreVide()))*
  Fsi
Fin.

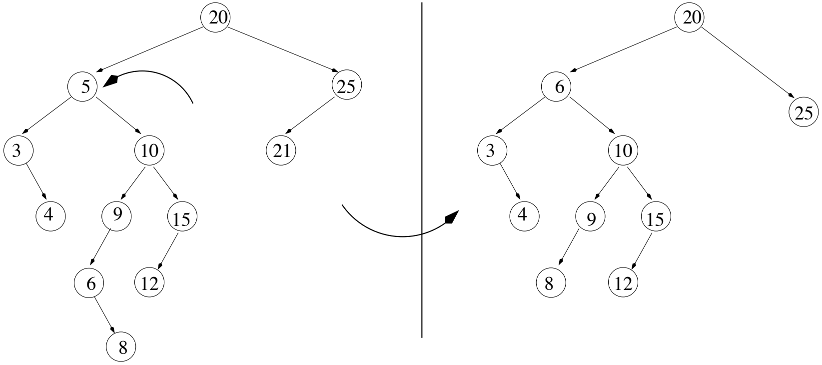
Supprimer une feuille



Supprimer un nœud à un fils



Supprimer un nœud à deux fils



Supprimer une valeur

Procédure *Supprimer(v : entier, A : Arbre).*
variable *: x,y,z : Arbre*
Début
  z ←Recherche(v,A)
  Si** non EstVide(z) **alors
    ***Si** EstVide(FilsGauche(z)) ou EstVide(FilsDroit(z)) **alors** y ←z*
    ***sinon** y ←Successeur(z)*
    Fsi
    ***Si** EstVide(FilsGauche(y)) **alors** x ←FilsDroit(y)*
    ***sinon** x ←FilsGauche(y)*
    Fsi
    Si** non EstVide(x) **alors** ModifPere(x,Pere(y))**Fsi
    ***Si** EstVide(Pere(y)) **alors** A ←x*
    sinon Si** y=FilsGauche(Pere(y)) **alors
      ModifFilsGauche(Pere(y),x)
    ***sinon** ModifFilsDroit(Pere(y),x)*
    Fsi
    Si** y ≠ x **alors** ModifCle(z,Cle(y))**Fsi
    liberer(y)
  Fsi
Fin.