

Cours n° 7 : les arbres et leurs parcours

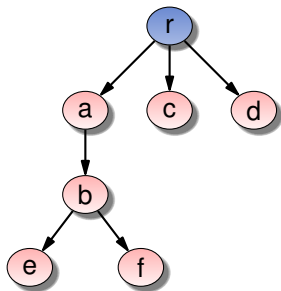
Christophe Gonzales



HUGO3 — Algorithmique

Arbre

- ▶ Ensemble fini A d'éléments, liés entre eux par une relation, dite de "parenté", vérifiant ces propriétés :
 - ▶ "x est le parent de y" ou "y est le fils de x"
 - ▶ Il existe un unique élément r (**racine**) de A sans parent
 - ▶ À part r , tout élément de A possède un **unique** parent



- ▶ Éléments de A = **nœuds**
- ▶ Nœuds sans fils = **feuilles** ou **nœuds terminaux**
- ▶ Les descendants d'un nœud x forment le **sous-arbre** de racine (ou issu de) x
- ▶ Un arbre **n -aire** est un arbre dont les nœuds ont tous au plus n fils.

Arbre binaire

- ▶ C'est un arbre 2-aire.
- ▶ Fils nommés : fils **gauche** et fils **droit**.
On confond souvent le fils et le sous-arbre issu du fils.

Arbre binaire - définition récursive

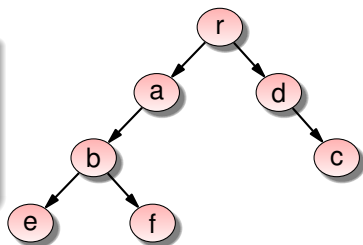
Un arbre binaire A est défini par :

- ▶ L'arbre vide (\emptyset) est un arbre binaire
- ▶ l'arbre de racine r , de fils gauche A_g et de fils droit A_d est un arbre binaire si :
 - ▶ A_g et A_d sont des arbres binaires.

Représentation d'un arbre binaire

```
// structure pour définir les noeuds du graphe
typedef struct node {
    char label;
    struct node* fils_gauche;
    struct node* fils_droit;
} node_t;
```

```
node_t e = {'e', NULL, NULL};
node_t f = {'f', NULL, NULL};
node_t b = {'b', &e, &f};
node_t a = {'a', &b, NULL};
node_t c = {'c', NULL, NULL};
node_t d = {'d', NULL, &c};
node_t r = {'r', &a, &d};
```



Arbre binaire

```
typedef node_t* bintree_t;
```

⇒ permet d'avoir des arbres vides (NULL)

```
// renvoie le label de la racine
char label (bintree_t tree) {
    // si l'arbre est vide, c'est une erreur
    if (tree == NULL) throw std::exception ();
    return tree->label;
}

// renvoie le fils gauche
bintree_t fils_gauche (bintree_t tree) {
    if (tree == NULL) throw std::exception ();
    return tree->fils_gauche;
}

// renvoie le fils droit
bintree_t fils_droit (bintree_t tree) {
    if (tree == NULL) throw std::exception ();
    return tree->fils_droit;
}

// renvoie le nombre de noeuds de l'arbre
int nb_noeuds (bintree_t tree) {
    if (tree == NULL) return 0;
    return 1 + nb_noeuds (tree->fils_gauche)
           + nb_noeuds (tree->fils_droit);
}
```

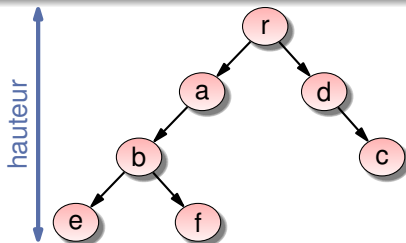
Calcul de la hauteur d'un arbre

$$\text{hauteur } h(A) = \begin{cases} -1 & \text{si } A = \emptyset \\ 1 + \max(h(A_g), h(A_d)) & \text{sinon.} \end{cases}$$

```
int hauteur (bintree_t tree) {
    if (tree == NULL) return -1;

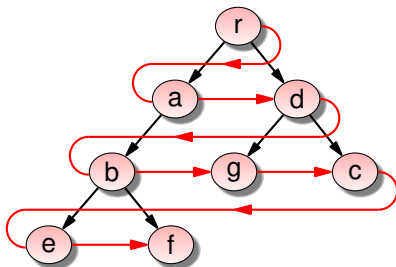
    int hauteur_gauche = hauteur (tree->fils_gauche);
    int hauteur_droite = hauteur (tree->fils_droit);

    if (hauteur_gauche >= hauteur_droite)
        return 1 + hauteur_gauche;
    else
        return 1 + hauteur_droite;
}
```



Parcours en largeur (1/2)

- ▶ **But : parcourir les nœuds selon le tracé rouge**
⇒ parcours niveau par niveau...



- ▶ parcourir *r*
- ▶ puis les fils *a* et *d* de *r*
- ▶ puis le fils *b* de *a* (1er fils de *r*)
- ▶ puis les fils *g* et *c* de *d* (dernier fils de *r*)
- ▶ puis les fils *e* et *f* de *b* (1er fils de *a*)



Utiliser une file !

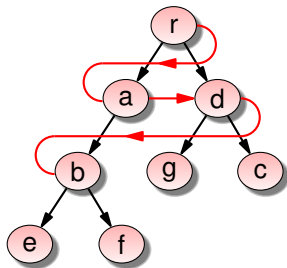
```
void parcours_largeur (node_t* tree) {  
    if (tree == NULL) return;  
  
    fifo_t file = create ();  
    push (&file, tree);  
  
    while (! empty(&file)) {  
        node_t* noeud = pop(&file);  
        printf ("%c ", noeud->label);  
        if (noeud->fils_gauche != NULL)  
            push(&file, noeud->fils_gauche);  
        if (noeud->fils_droit != NULL)  
            push(&file, noeud->fils_droit);  
    }  
    printf ("\n");  
}
```

file =

g	c	e	f
---	---	---	---

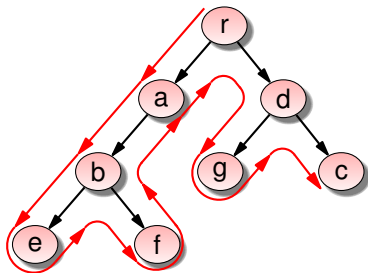
 noeud = b

affichage :



Parcours en profondeur

- **But : parcourir les nœuds selon le tracé rouge**
⇒ parcours en descendant dès qu'on peut...



⇒ affichage : r a b e f d g c

```
void parcours_profondeur (bintree_t tree) {  
    if (tree == NULL) return;  
  
    printf ("%c ", tree->label);  
  
    parcours_profondeur (tree->fils_gauche);  
    parcours_profondeur (tree->fils_droit);  
}
```

Définition d'un dictionnaire (au sens informatique)

Dictionnaire = ensemble de couples clé/valeur où

- ▶ Les clés forment un ensemble **totallement ordonné** ;
 - ▶ Chaque valeur est associée à une unique clé ;
 - ▶ Chaque clé caractérise une unique valeur.
-
- ▶ **Un dictionnaire sert à représenter :**
 - ▶ des dictionnaires (au sens littéraire)
 - ▶ des listes de contacts
 - ▶ des bases de données triées par un identifiant
 - ▶ *etc.*

- ▶ **Problème** : comment représenter un dictionnaire ?
 - ▶ un tableau trié ?
 - ▶ On peut retrouver en $O(\log n)$ une clé (par dichotomie).
 - ▶ Mais insertion et suppression \implies décalages $\implies O(n)$.
 - ▶ une liste triée ?
 - ▶ Recherche trop lente : $O(n)$.
 - ▶ une pile ou une file ?
 - ▶ Recherche trop lente : $O(n)$.
- ▶ **Nouvelle proposition** : Arbre Binaire de Recherche (ABR)

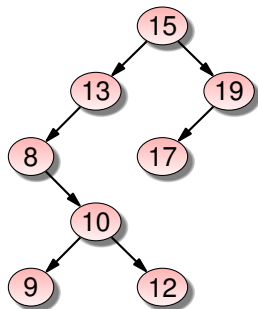
Définition d'un ABR

- ▶ **Notation** : $c(x)$ = clé du nœud x .
- ▶ Pour simplifier, on assimilera la valeur à sa clé.

Définition d'un arbre binaire de recherche

Un Arbre Binaire de Recherche est :

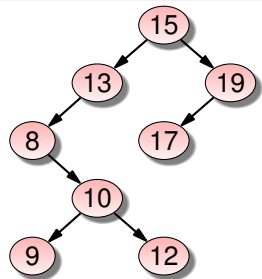
- ▶ un arbre binaire A
- ▶ $\forall x \in A, \forall y \in A_g(x), \forall z \in A_d(x), c(y) < c(x) < c(z)$



```
void parcours_infixe (bintree_t tree) {  
    if (tree == NULL) return;  
  
    parcours_infixe (tree->fils_gauche);  
    printf ("%d ", tree->label);  
    parcours_infixe (tree->fils_droit);  
}
```

Affichage : 8 9 10 12 13 15 17 19

Recherche du plus petit/plus grand élément

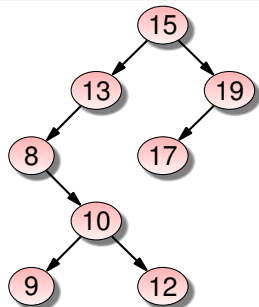


- ▶ Plus petite clé :
dans le nœud le « plus à gauche »
Décalage vers la droite \implies augmentation de la clé
- ▶ Plus grande clé :
dans le nœud le « plus à droite »
Décalage vers la gauche \implies diminution de la clé

```
int get_min (bintree_t tree) {  
    // si l'arbre est vide, c'est une erreur  
    if (tree == NULL) throw std::exception ();  
    if (tree->fils_gauche == NULL) return tree->label;  
    else return get_min(tree->fils_gauche);  
}
```

```
int get_max (bintree_t tree) {  
    // arbre vide = erreur  
    if (tree == NULL) throw std::exception ();  
    if (tree->fils_droit == NULL) return tree->label;  
    else return get_max(tree->fils_droit);  
}
```

Rechercher si un élément existe

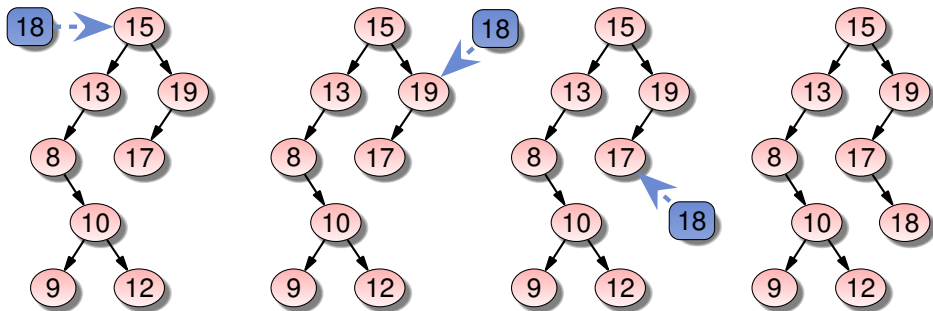


Idée de l'algorithme :

- ▶ Si la clé de la racine = l'élément, on l'a trouvé.
- ▶ Sinon, il est $>$ ou $<$.
- ▶ Clé de la racine $>$ l'élément \implies l'élément ne peut se trouver que dans le sous-arbre gauche.
- ▶ Clé de la racine $<$ l'élément \implies l'élément ne peut se trouver que dans le sous-arbre droit.

```
bool existe_elt (bintree_t tree, int clef) {  
    if (tree == NULL) return false;  
  
    if (tree->label == clef) return true;  
  
    if (tree->label > clef)  
        return existe_elt(tree->fils_gauche, clef);  
    else  
        return existe_elt(tree->fils_droit, clef);  
}
```

Insertion d'un nouvel élément



```
bintree_t insert (bintree_t tree, int elt) {  
    if (tree == NULL) return new_node(elt);  
    if (tree->label > elt)  
        tree->fils_gauche = insert (tree->fils_gauche, elt);  
    else if (tree->label < elt)  
        tree->fils_droit = insert (tree->fils_droit, elt);  
    return tree;  
}
```

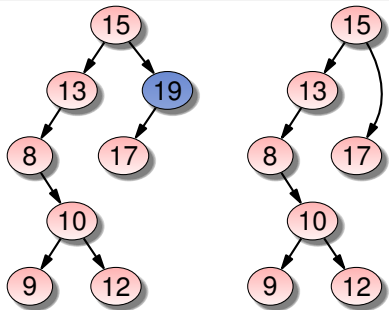


ABR vide \implies la racine de l'arbre peut changer !

- ▶ En principe, nœuds créés « à la volée »
⇒ allocation dynamique

```
node_t* new_node (int elt) {  
    // allocation  
    node_t* node = (node_t*) malloc (sizeof(node_t));  
    if (node == NULL) throw std::exception ();  
  
    // remplissage des champs du nœud  
    node->label = elt;  
    node->fils_gauche = NULL;  
    node->fils_droit  = NULL;  
  
    return node;  
}
```


Suppression de l'élément max



L'élément max ne peut avoir de sous-arbre droit !

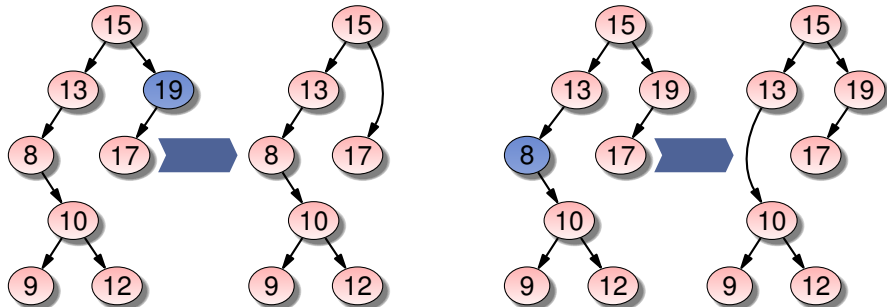
```
bintree_t suppr_max (bintree_t tree) {  
    if (tree == NULL) return NULL;  
    if (tree->fils_droit == NULL) {  
        node_t* fils_gauche = tree->fils_gauche;  
        free (tree);  
        return fils_gauche;  
    }  
    else {  
        tree->fils_droit = suppr_max (tree->fils_droit);  
        return tree;  
    }  
}
```

Suppression d'un élément x quelconque (1/4)

Suppression d'un nœud de l'ABR :



Si le nœud n'a qu'un enfant, rechaîner cet enfant :

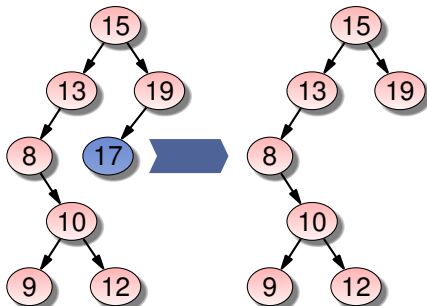


Suppression d'un élément x quelconque (2/4)

Suppression d'un nœud de l'ABR :



Si le nœud n'a aucun enfant, le supprimer simplement :

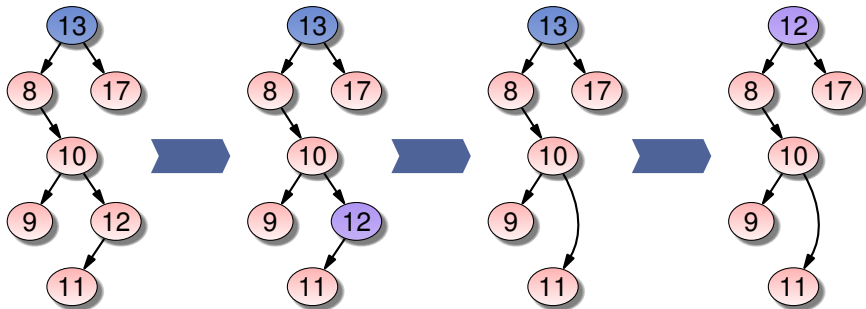


Suppression d'un élément x quelconque (3/4)

- ▶ Si le nœud a 2 enfants :
- ▶ Soit y le max du sous-arbre gauche de x
 - $\implies \forall z$ dans le sous-arbre gauche de $x : z < y$
 - $\implies \forall t$ dans le sous-arbre droit de $x : y < t$



remplacer x par y et rechaîner le sous-arbre gauche de y
(\implies utiliser `suppr_max` pour le rechaînage)



Suppression d'un élément x quelconque (4/4)

```
bintree_t suppr (bintree_t tree, int elt) {  
    // on recherche le noeud à supprimer  
    if (tree == NULL) throw std::exception ();  
    if (tree->label > elt) {  
        tree->fils_gauche = suppr (tree->fils_gauche, elt);  
        return tree;  
    }  
    if (tree->label < elt) {  
        tree->fils_droit = suppr (tree->fils_droit, elt);  
        return tree;  
    }  
  
    // si le noeud a au plus un fils  
    if (tree->fils_gauche == NULL) {  
        node_t* fils_droit = tree->fils_droit;  
        free (tree);  
        return fils_droit;  
    }  
    if (tree->fils_droit == NULL) {  
        node_t* fils_gauche = tree->fils_gauche;  
        free (tree);  
        return fils_gauche;  
    }  
  
    // si le noeud a 2 enfants  
    tree->label = get_max(tree->fils_gauche);  
    tree->fils_gauche = suppr_max(tree->fils_gauche);  
    return tree;  
}
```

- ▶ ABR intéressants pour représenter des ensembles ordonnés de clés.
- ▶ Les alternatives (tableaux ou listes) ont de bien moins bons comportements dans les opérations usuelles : ajout / suppression / recherche.
- ▶ Dans tous les algorithmes des ABRs, le temps de calcul est proportionnel à la hauteur de l'arbre (et non au nombre de clés comme pour les autres alternatives)
- ▶ Mais, dans le pire des cas, la hauteur de l'arbre est le nombre de clés dans l'arbre.
- ▶ En moyenne, les ABRs sont quand même souvent intéressants.
- ▶ Pour éviter les mauvais cas : **équilibrage des arbres.**