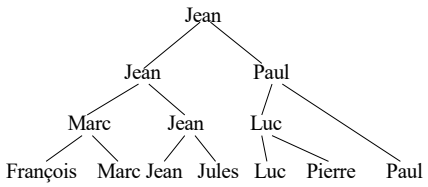
2022-2023, Semestre d'automne
L3, Licence Sciences et Technologies
Université Lyon 1

LIFAPC: Algorithmique, Programmation et Complexité

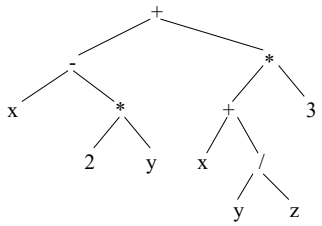
Chaine Raphaëlle (responsable semestre automne)
E-mail : raphaelle.chaine@liris.cnrs.fr
http://liris.cnrs.fr/membres?idn=rchaine

Structures arborescentes

- **Modélise une relation « père de »**
- **Arbres binaires :**
  - au plus 2 fils



- Arbre binaire d'une expression arithmétique :

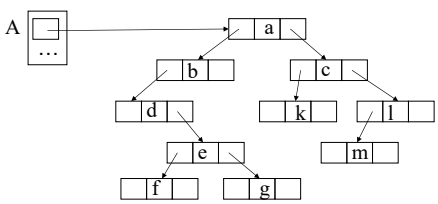


(x-(2*y))+((x+(y/z))*3)

- **Définitions (rappels)**
  - Un **arbre binaire** est soit vide, soit de la forme $B =< o, B_1, B_2 >$ où B1 et B2 sont des arbres binaires disjoints et o un élément stocké dans un nœud **racine** (équivalent de la notion d'«emplacement » ou de « cellule » des listes)
- **module** ArbreBin
  - **importer**
    Module Element
  - **Exporter**
    Type **AB, Noeud**
    procédure initialisation(Résultat a : AB)
    {Préc° : a- non initialisé , Postc° : a+ AB vide}
    procédure initialisation(Résultat a : AB, Donnée b : AB)
    {Préc° : a- non initialisé , Postc° : a+ **COPIE PROFONDE** de b}
    procédure initialisation(Résultat a : AB, Donnée b : AB, p : Nœud *)
    {Préc° : a- non initialisé ,
    Postc° : a+ **COPIE PROFONDE** du ss.arbre de b enraciné en p}

Nœud * **fonction noeudRacine**(a : AB)
{Préc° : a initialisé non vide,
Res : adresse du nœud racine de a}
Nœud * **fonction gauche**(a : AB, p : Nœud *)
{Préc° : p adresse d'un ss. arbre de a ,
Res : adresse du nœud racine
du ss arbre gauche de p}
Nœud * **fonction droite**(a : AB, p : Nœud *)
{Préc° : p adresse d'un ss. arbre de a,
Res : adresse du nœud racine
du ss arbre droit de p}
booléen **fonction testVide**(a : AB, p : Nœud *)
{Préc° : p adresse d'un ss. arbre de a,
Res : vrai si ss.arbre vide, faux sinon}
Element **fonction contenu**(n : Nœud)
{Préc° : n initialisé, Res : elt contenu dans n}
procédure testament(Donnée-Résultat a : AB)
{Préc° : a- initialisé , Postc° : a+ prêt à disparaitre}
FinModule

- **Représentation des arbres binaires**
  - *par utilisation de pointeurs* :
    - A chaque nœud, on associe 2 pointeurs, l'un vers le sous-arbre gauche, l'autre vers le sous-arbre droit.
    - L'arbre est déterminé par l'adresse de sa racine (et éventuellement d'autres infos)



2.2 Par utilisation de tableaux

	sommet	g	d
1			
2	d	-1	10
3	a	5	6
4	g	-1	-1
5	b	2	-1
6	c	13	11
7			
8	f	-1	-1
9	m	-1	-1
10	e	8	4
11	l	9	-1
12			
13	k	-1	-1

266

266

- **Nuance entre arbre et sous-arbre**
  - Dans les procédures et les fonctions, la donnée d'un couple <Arbre, pointeur sur Nœud> modélise généralement un sous-arbre
  - Un sous-arbre peut servir à initialiser un arbre par copie de ses Nœuds et de son organisation… mais un sous-arbre n'est pas un arbre (seulement une sous partie d'un arbre!)
  - La plupart des traitements récursifs s'effectueront sur des sous-arbres
  - Exemple :
    - L'affichage d'un arbre pourra utiliser une procédure récursive d'affichage d'un sous-arbre :
      - commence l'affichage à partir d'un Nœud dont l'adresse est fournie en paramètre.

267

267

- **Vocabulaire familial :**
  - **fils gauche** (resp. **fils droit**) d'un nœud : racine de son sous-arbre gauche (resp. droit)
  - si un nœud n_i a pour fils gauche (resp. droit) un nœud n_j, n_i est le **père** de n_j
  - deux nœuds qui ont le même père sont dits **frères**
  - le nœud n_i est un **ascendant** (ou un **ancêtre**) de n_j si et seulement si n_i est le père de n_j ou un ascendant du père de n_j
  - le nœud n_i est un **descendant** de n_j si et seulement si n_i est un fils de n_j ou un descendant d'un fils de n_j
  - un nœud est :
    - **interne** s'il a exactement 2 fils,
    - **simple** s'il a exactement un fils,
    - **feuille** s'il est sans fils

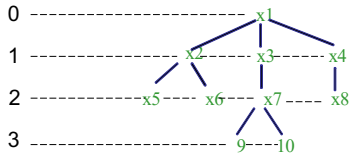
268

268

- la **taille** d'un arbre est définie par

$$taille(arbre - vide) = 0$$
$$taille(< o, B_1, B_2 >) = 1 + taille(B_1) + taille(B_2)$$

- la **profondeur** d'un nœud x est définie par
  prof(x)= 0 si x est la racine de B
  prof(x)=1+prof(y) si y est père de x

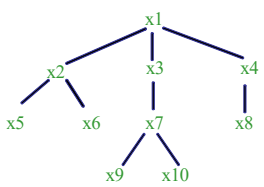


269

269

- La **hauteur** d'un arbre B est
  h(B)=max{prof(x) / x nœud de B} +1
- La hauteur d'un nœud x est le nombre de nœud sur le chemin qui le mène à la plus lointaine de ses feuilles

h(x8) = 1
h(x7) = 2
h(x3) = 3
h(*A*) = h(x1) = 4



270

270

- la **longueur de cheminement** d'un arbre B est

$$LC(B) = \sum_{x\ noeud\ de\ B} prof(x)$$

- la **longueur de cheminement externe** d'un arbre B est

$$LCE(B) = \sum_{f\ feuille\ de\ B} prof(f)$$

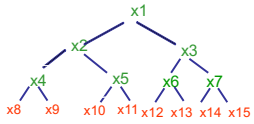
- la **longueur de cheminement interne** d'un arbre B est

$$LCI(B) = \sum_{x\ noeud\ interne\ de\ B} prof(x)$$

271

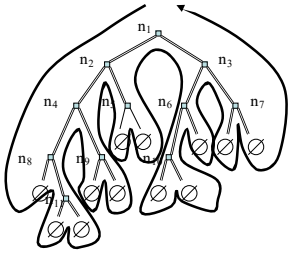
271

- Un arbre binaire est :
  - **dégénéré** s'il n'a que des nœuds simples ou feuilles
  - **complet** s'il contient 2^h nœuds au niveau h, $\forall h \geq 0$



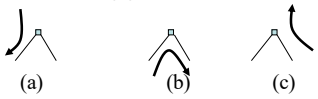
- **localement complet** s'il est non vide et s'il n'a pas de nœud simple
- un **peigne gauche** (resp. **droit**) s'il est localement complet et si tout fils droit (resp. gauche) est une feuille

Parcours d'un arbre (binaire)



Parcours en profondeur « à main gauche »
(rappels LIF3, LIF5)

- Chaque nœud de l'arbre est rencontré 3 fois : à la descente (a), en montée gauche (b) et enfin en montée droite (c)




```
procédure parcours-recursif(donnée-résultat t : AB, p : Noeud *)
début
  si (non testVide(t,p)) alors
    traitement1 ;
    parcours-recursif(t,p→gauche) ;
    traitement2 ;
    parcours-recursif(t,p→droit) ;
    traitement3 ;
  finsi
fin
```

On part à gauche, mais la pile des appels permettra de revenir ici quand on aura fini à gauche!

On part à droite, mais la pile des appels permettra de revenir ici quand on aura fini à droite!

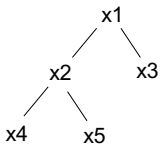
- 3 cas particuliers classiques (rappels) :
  - ordre **préfixe** (ou **préordre**) : celui dans lequel traitement1 est appliqué
  - ordre **infixe** (ou **symétrique**) : celui dans lequel traitement2 est appliqué
  - ordre **suffixe** (ou **postfixe**) : celui dans lequel traitement3 est appliqué


```
procédure parcours-itératif(donnée-résultat a : Arbre, p : Noeud * )
variables P : pile ; N : entier ; t : noeud *
début N← 1; initialise(P); t← p
  répéter
    si N = 1 alors
      tant que non testSsArbreVide(a,t) faire
        traitement1 ;empiler(P,(t,1))
        t← gauche(a,t)
      fintant que
    finsi
    si est-vide(P) = faux alors
      (t,N) ← sommet(P); dépiler(P)
      si N = 1 alors
        traitement2; empiler(P,(t,2))
        t← droit(a,t); N = 1
      sinon traitement3 finsi
    finsi
  jusqu'à est-vide(P) = vrai
fin
```

N vaut 1 à la descente

On utilise le pointeur de travail t pour aller à gauche toute (sans aucun dépilement)

Dès que l'on va à droite on se prépare à aller à nouveau à gauche toute…



- Etats successifs marquants de la pile P (haut de la pile représenté vers le bas) :

X1 1	X1 1	X1 1	X1 1	X1 1	X1 1	X1 1	etc.
X2 1	X2 1	X2 2	X2 2	X2 2	X2 2		
X4 1	X4 2		X5 1	X5 2			

278

278

Analyseur syntaxique

- But : analyser une expression arithmétique et construire l'arbre binaire associé
- On désignera par flot, le flot de caractères correspondant à l'expression
- Une première phase d'analyse lexicale permet la décomposition du flot en lexèmes :
  - opérateurs +, -, *, / (on dispose de lit_operateur)
  - parenthèses (,) (on dispose de lit_parenthèse)
  - constantes (on dispose de lit_constante)
- L'analyse syntaxique permet de reconnaître des combinaisons de lexèmes formant des entités syntaxiques (ici des expressions arithmétiques)

279

279

- Définitions
  - Une **expression** est composée d'une suite de termes séparés par des opérateurs **+** ou **-**
    Expression : Terme, {('+';'-'), Terme}
  - Un **terme** est composé d'une suite de facteurs séparés par des opérateurs ***** ou **/**
    Terme : Facteur, {('*';'/'), Facteur}
  - Un **facteur** correspond à une constante entière ou alors à une expression entre parenthèses
    Facteur : Constante ; '(', Expression, ')'

280

280

- Fonctions **internes** utilisées :
  - Lecture d'un facteur et création de la branche correspondante
    Nœud * lit_facteur(donnée-résultat f : flot)
  - Lecture d'un terme et création de la branche correspondante
    Nœud * lit_terme(donnée-résultat f : flot)
  - Lecture d'une expression et création de la branche correspondante
    Nœud * lit_expression(donnée-résultat f : flot)

281

281

- La procédure d'initialisation d'un AB à partir d'une expression arithmétique utilise la 3ème fonction interne, qui elle-même utilise les précédentes…

```
procédure initialise(résultat a : AB, donnée-résultat f: flot)
début
   a.racine ← lit_expression(f)
   a.autresinfos ← informations nécessaires
fin
```

282

282

- Version itérative :
  - On procède pour cela à une lecture de la gauche vers la droite de l'expression

```
Nœud * lit_expression(donnée-résultat f : flot)
variables
  b1, b2, b3 : pointeur sur Nœud     op : opérateur
début
   b1← lit_terme(f)
   tantque f contient ensuite + ou –  faire
      op ← lit_opérateur(f)
      b2 ← lit_terme(f)
      b3 ← nouveau nœud
      b3->info ← op, b3->gauche ← b1, b3->droite ←b2
      b1 ← b3
   fintantque
   retourne b1
fin
```

283

283

```
Nœud *  lit_terme(donnée-résultat f : flot)
  variables
    b1, b2, b3 : pointeur sur Nœud
    op : opérateur
  début
    b1 ← lit_facteur(f)
    tantque f contient ensuite * ou /  faire
        op ← lit_opérateur(f)
        b2 ← lit_facteur(f)
        b3 ← nouveau nœud
        b3->info ← op, b3->gauche ← b1, b3->droite ←b2
        b1 ← b3
    fintantque
    retourne b1
  fin
```

284

284

```
Nœud *  lit_facteur(donnée-résultat f : flot)
  variables
    b : pointeur sur Noeud, c : constante
  début
    si le prochain lexème dans le flot est ( alors
       lit_parenthèse_ouvrante(f)
       b ← lit_expression(f)
       lit_parenthèse_fermante(f)
    sinon
       c ← lit_constante(f)
       b1 ← nouveau nœud
       b1->info ← c,
       b1->gauche ← NULL, b1->droite ←NULL
    finsi
    retourne b1
  fin
```

285

285

Version récursive :

A votre avis quel est le sens le plus adapté pour lire une expression avec une approche récursive?

T1-T2+T3-T4

??? T1-(T2+(T3-T4)) ???

??? ((T1-T2)+T3)-T4 ???

286

286

Version récursive :

lecture de la droite vers la gauche

(ou lecture de la gauche vers la droite avec inversion des signes après chaque – (resp. /) rencontré)

287

287

Version récursive :

```
pointeur sur Nœud  lit_expression(donnée-résultat f : flot)
variables
  b1, b2, b3 : pointeur sur Nœud
  op : opérateur
début
  b1← lit_terme(f)
  si f contient ensuite + ou –  alors
      op ← lit_opérateur(f)
      b2 ← lit_expression(f)
      b3 ← nouveau noeud
      b3->info ← op, b3->gauche ← b2, b3->droite ←b1
      b1 ← b3
  finsi
  retourne b1
fin
```

288

288

Version récursive :

```
pointeur sur Nœud  lit_terme(donnée-résultat f : flot)
variables
  b1, b2, b3 : pointeur sur Nœud
  op : opérateur
début
  b1← lit_facteur(f)
  si f contient ensuite * ou /  alors
      op ← lit_opérateur(f)
      b2 ← lit_terme(f)
      b1 ← nouveau noeud
      b3->info ← op, b3->gauche ← b2, b3->droite ←b1
      b1 ← b3
  finsi
  retourne b1
fin
```

289

289

- Comment évaluer une expression arithmétique représentée sous forme d'un Arbre Binaire ?

290

290

```
entier fonction eval (A : AB, n : Nœud *)
début
  si n≠NULL alors
    cas n->genre =
      cste : retourne n->val
      autre : cas n->op =
                + : retourne eval(A,n->gauche)+eval(A,n->droite);
                - : retourne eval(A,n->gauche)-eval(A,n->droite);
                * : retourne eval(A,n->gauche)*eval(A,n->droite);
                / : retourne eval(A,n->gauche)/eval(A,n->droite);
              fin cas
    fincas
  finsi
fin
```

291

291

Arbre Binaire de Recherche

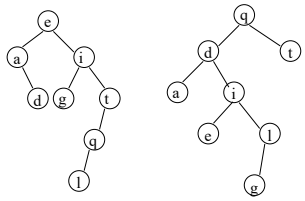
Rappels (LIF3, LIF5)

- Un Arbre Binaire de Recherche (ABR) est un Arbre Binaire dont les éléments considérés dans l'ordre infixé sont ordonnés dans l'ordre croissant
- Pour tout nœud n d'un ABR
  - Les éléments du sous-arbre gauche de n sont ≤ contenu(n)
  - Les éléments du sous-arbre droit de n sont > contenu(n)

292

292

- Attention, pour un même contenu la configuration d'un ABR n'est pas unique!



293

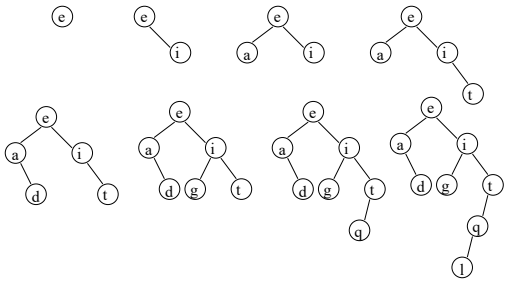
293

- Structure de donnée intéressante pour la recherche d'un élément
  - Seulement quand l'ABR est équilibré (performance en $O(lg_2 n)$, similaire à une dichotomie)
  - Si l'arbre est **dégénéré**, la recherche se fait en O(n) (similaire à une recherche dans une liste triée) ce qui est beaucoup moins intéressant!!
- Descente dans l'arbre à gauche OU à droite de chaque nœud rencontré
- Testez votre mémoire :
  - Dans la version récursive de la procédure de recherche d'un élément, combien y a-t-il d'appels récursifs?

294

294

- Insertion aux feuilles d'un élément (rappel)



295

295

```
procédure insérer(donnée e : Elément,
                  donnée-résultat A:ABR,
                                  n : Nœud *)
  début
    si n ≠ NULL alors
      si e ≤ n->info alors
        insérer(e,A,n->gauche)
      sinon
        insérer(e,A,n->droite)
      finsi
    sinon
      n←nouveau Nœud(e,NULL,NULL)
    finsi
  fin
```

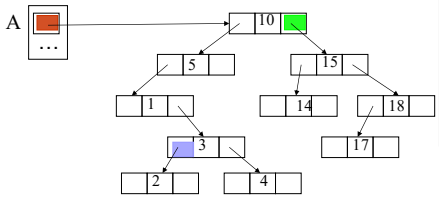
296

296

296

Suppression d'un élément e (rappel)

- Recherche de e dans l'ABR
  - Plus précisément rechercher l'emplacement p de type pointeur qui contient l'adresse du nœud contenant e (s'il existe)
  - **Attention, on sera amené à modifier le contenu de p et non pas d'une copie! (on veut l'accès à l'emplacement et pas seulement sa valeur…)**



Utilisation d'un pointeur de travail de type pointeur sur pointeur, pour effectuer la localisation

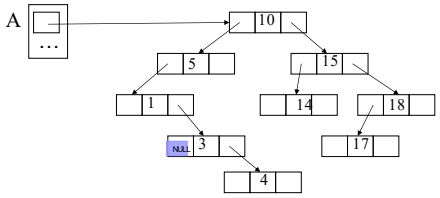
Recherche avant suppression de 10 Recherche avant suppression de 15

Recherche avant suppression de 2

297

297

- Dans tous les cas, libération de l'espace mémoire occupé par le nœud contenant e après réorganisation de l'arbre
- Cas d'une suppression en feuille
  - On met p à null



Etat de l'ABR du transparent précédent après suppression de 2

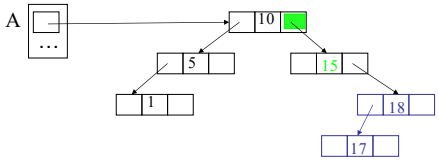
298

298

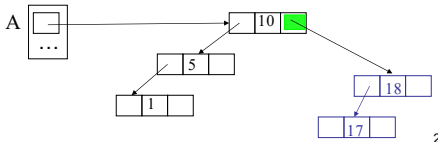
- Cas d'une suppression d'un nœud possédant 1 seul sous-arbre :
  - On transmet ce sous-arbre à p

Suppression de 15

Avant…



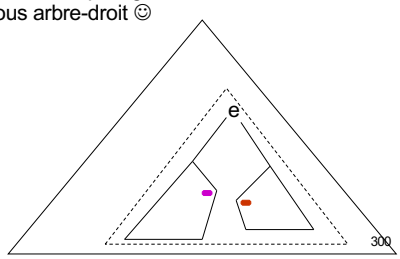
Après…



299

299

- Sinon …
  - On remplace e par le plus petit de ses descendants supérieurs, ou le plus grand de ses descendants inférieurs **après avoir supprimé ce remplaçant de son emplacement initial** (un des 2 cas précédents)
  - En effet, le plus petit descendant supérieur n'a pas de sous-arbre gauche, le plus grand descendant inférieur n'a pas de sous arbre-droit ☺

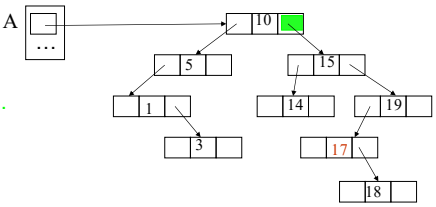


300

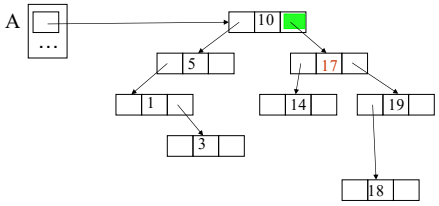
300

Suppression de 15

Avant …



Après …



301

301

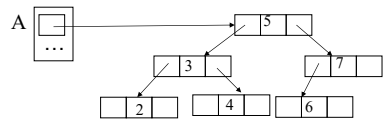
ABR équilibré

- Un ABR est en équilibre parfait si sa hauteur est minimale
  - Pour chaque nœud, le nombre d'éléments du sous arbre gauche et du sous arbre droit diffèrent de 1
  - Configuration obtenue en insérant l'élément médian m, puis l'élément médian des éléments <= m puis l'élément médian des éléments > m …
  - Dur à maintenir, une simple insertion peut entrainer toute une réorganisation de l'arbre!

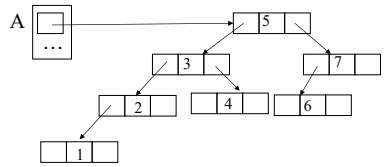
302

302

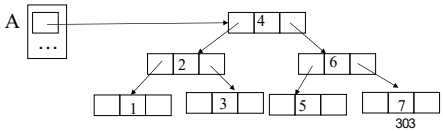
Arbre initial



Arbre initial dans lequel on insère 1



Retour à l'équilibre



303

303

- Pour éviter des réorganisations douloureuses en temps de calcul, on assouplit les conditions d'équilibrage,
  - pour limiter le coût de la réorganisation
  - tout en gardant des performances logarithmiques pour la recherche, l'insertion et la suppression!

304

304

AVL

- Du nom des auteurs de la méthode : Adelson-Velskij et Landis
- Garantir après chaque opération que
  - pour chaque nœud, les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1
- Cet équilibre peut-être maintenu à travers l'utilisation judicieuse de 4 opérations :
  - Rotation gauche
  - Rotation droite
  - Rotation droite-gauche (ou rotation double à gauche)
  - Rotation gauche-droite (ou rotation double à droite)

305

305