Structures de Données: arbres

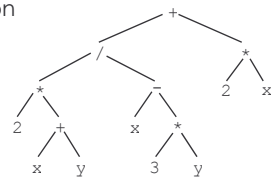
Licence IUP-MIAGE

Pascal PETIT

(sur la base d'un support de **Guillaume HUTZLER (2003-2004)**)

Les arbres

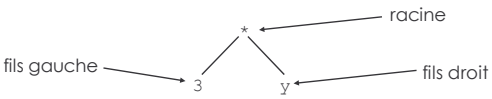
- SD très utilisée en informatique
- Exemples:
  - Arbres généalogiques
  - Arbres de classification
  - Arbres d'expression



  - Traduction de l'expression

```
(2 * (x + y)) / (x – 3 * y) + 2 * x
```


Les arbres binaires



Soit un arbre $A = <o, A_1, A_2>$

- *racine* de A, le nœud o
- *sous-arbre gauche* de A (ou o), l'arbre A_1
- *sous-arbre droit* de A (ou o), l'arbre A_2
- *sous-arbre* de A, un arbre C quelconque tq
  - soit $C = A$
  - soit C est sous-arbre de A_1
  - soit C est sous-arbre de A_2

TAD arbre binaire

```
type arbre
paramètre nœud
opérations
   ArbreVide  : → arbre
   <_, _, _>  : nœud x arbre x arbre → arbre
   racine     : arbre → nœud
   g          : arbre → arbre
   d          : arbre → arbre
préconditions
   racine(a)  : a ≠ ArbreVide
   g(a)       : a ≠ ArbreVide
   d(a)       : a ≠ ArbreVide
axiomes
   ∀ o : nœud, A1, A2 : arbre
   racine(<o, A1, A2>) = o
   g(<o, A1, A2>) = A1
   d(<o, A1, A2>) = A2
```


Terminologie avancée (1)

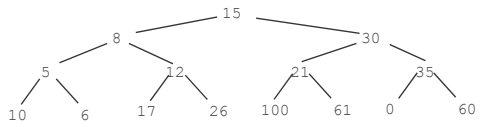
- ***fils gauche*** de n = racine de son sous-arbre gauche
- ***fils droit*** de n = racine de son sous-arbre droit
- ***fils*** de n = l'un quelconque des nœuds fils gauche et fils droit
- ***père*** de n = le nœud, s'il existe dont n est un fils
- ***ascendant*** de n = tout nœud qui est, soit père de n, soit ascendant du père de n
- ***descendant*** de n = tout nœud dont n est ascendant
  - = fils de n ou descendant d'un fils de n
  - = nœud quelconque du sous-arbre droit ou du sous-arbre gauche

Terminologie avancée (2)

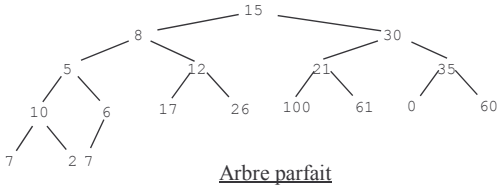
- ***nœud interne*** = nœud qui a deux fils
- ***nœud interne au sens large*** = nœud qui a un fils
- ***arbre complet*** = un arbre qui n'a pas de nœud interne au sens large
- ***arbre parfait*** = un arbre dont tous les niveaux sont remplis sauf éventuellement le dernier dont les feuilles sont le plus à gauche possible
- ***feuille*** = un nœud qui n'a pas de fils
- ***branche*** de l'arbre = tout chemin de la racine à l'une de ses feuilles
- ***bord gauche (droit)*** de l'arbre = le plus long chemin depuis la racine en ne suivant que les fils gauches (droits)

Structures arborescentes

Terminologie avancée (3)



Arbre complet



Arbre parfait

Structures arborescentes

Mesures sur les arbres (1)

- ***taille*** de l'arbre = le nombre de ses nœuds, noté ***taille(A)***
- ***nombre de feuilles***, noté ***nf(A)***
- ***hauteur d'un nœud*** = la longueur du chemin qui relie la racine à ce nœud = longueur de l'occurrence du nœud, noté ***h(x)***
- ***hauteur de l'arbre*** = la longueur de la plus longue branche, noté ***h(A)***

```
h(A) = max {h(x) | x nœud de A}
```

Structures arborescentes

Propriétés sur les arbres (1)

- **Lemme1.** taille(A) ≥ 2 nf(A) – 1
  - égalité pour un arbre complet
- **Lemme 2.** h(A) ≤ taille(A) – 1
  - égalité pour un arbre dégénéré filiforme
- **Lemme 3.** taille(A) ≤ $2^{h(A)+1}$-1
  - égalité pour un arbre complet dont toutes les feuilles ont la profondeur h(A)
- **Corollaire 4.**
  $\lfloor \log_2 taille(A) \rfloor \leq h(A) \leq taille(A) - 1$
  $\lceil \log_2 nf(A) \rceil \leq h(A)$
  - minorant atteint pour les arbres complets dont toutes les feuilles ont la profondeur h(A)
  - majorant pour les arbres dégénérés filiformes

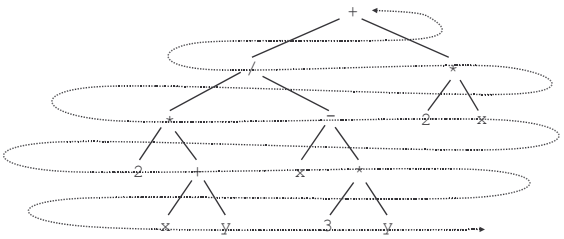
Structures arborescentes

Exploration

- pas aussi simple que dans le cas des listes
- pas d'ordre « naturel »
- Deux types de parcours
  - En largeur d'abord
  - En profondeur d'abord
    - trois types principaux de traitement
      - préfixé
      - infixé
      - postfixé

Structures arborescentes

Parcours en largeur d'abord (1)



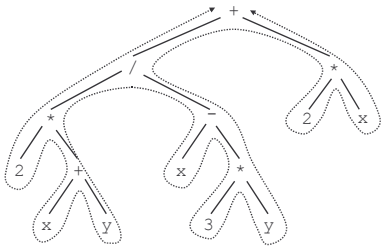
- Ordre d'évaluation des noeuds :
  - `+ / * * - 2 x 2 + x * x y 3 y`

Structures arborescentes

Parcours en largeur d'abord (2)

```
Void ParcourirEnLargeur(arbre a){
  file ce_niveau=fileVide(),
    niveau_inferieur=fileVide();
  ajouter(a, &ce_niveau);
  while (! estVide(ce_niveau)){
    niveau_inférieur = fileVide();
    while (! estVide(ce_niveau)){
      o=premier(ce_niveau);
      traiter(o);
      if (!estVide(g(o)))
         ajouter(g(o),&niveau_inférieur);
      if (!estVide(d(o)))
         ajouter(d(o),&niveau_inférieur);
    }
    ce_niveau = niveau_inférieur
  }
}
```

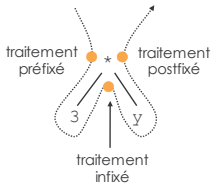
Parcours en profondeur d'abord



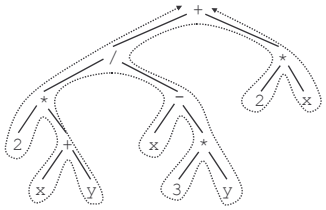
- Ordre d'évaluation des noeuds :
  - Ça dépend...

Parcours en profondeur : algo

```
void explorer (arbre A ){
  if (estVide(A))
   trait_arbre_vide();
  else{
   trait_préfixé(racine(A));
   explorer(g(A));
   trait_infixé(racine(A));
   explorer(d(A));
   trait_postfixé(racine(A));
  }
}
```

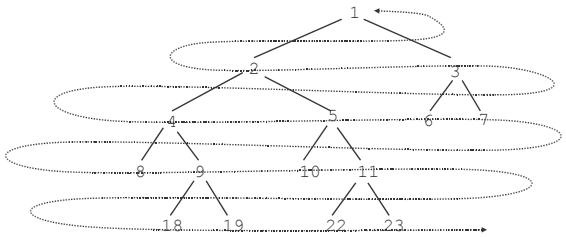


Parcours en profondeur : ex.



- Parcours infixé
  - `2 * x + y / x - 3 * y + 2 * x`
- Parcours préfixé :
  - `+ / * 2 + x y - x * 3 y * 2 x`
- Parcours postfixé
  - `2 x y + * x 3 y * - / 2 x * +`

Implantation contiguë par niveaux – 0



Implantation contiguë par niveaux – 0

- On numérote les noeuds dans l'ordre du parcours en largeur d'abord.
- pere(n)=n/2
- filsGauche(n)=2*x
- fildDroit(n)=2*n+1

Implantation contiguë par niveaux – 1

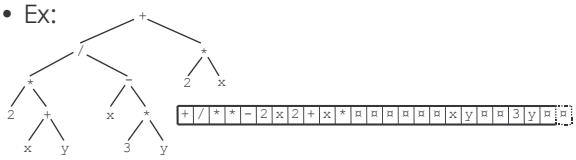
- Occurrence de x:
  - pour tout nœud x, mot de $\{0, 1\}^*$
- Code
  - bijection entre $\{0, 1\}^*$ et N^{+*}
  - code : $\{0, 1\}^* \rightarrow N^{+*}$
    - code(ε) = 1
    - code(μ0) = 2 * code(μ)
    - code(μ1) = 2 * code(μ) + 1
  - code(μ) = entier dont la représentation à base 2 est 1μ

Implantation contiguë par niveaux – 2

- a tout nœud x, on associe un numéro égal au code de son occurrence, noté également **code (x)**
- Pour tout nœud x :
  - code(père(x)) = code(x) div 2
  - code(filsgauche(x)) = 2 * code(x)
  - code(filsdroit(x)) = 2 * code(x) + 1
  - h(x) = $\lfloor \log_2 code(x) \rfloor$
- corollaires
  - pour tout nœud x de niveau p, code(x) ∈ $[2^p..2^{p+1}[$
  - pour tout nœud x d'un arbre A,code(x) < $2^{h(A)+1}$

Implantation contiguë par niveaux – 3

- Représentation : espace mémoire contigu
  - au moins $2^{h(A)+1} - 1$ emplacements
  - numérotés à partir de 1
  - on range x à l'indice code(x)
  - si l'emplacement ne correspond pas à un nœud, il est marqué inexistant
- Ex:



Implantation contiguë par niveaux – 4

- algos efficaces pour la recherche et l'exploration en longueur et en largeur
- ajout d'un nœud
  - changer la marque d'inexistence
- suppression d'un nœud
  - marquage de l'emplacement correspondant
  - suppression des sous-arbres gauche et droit
- déplacement d'un sous arbre
  - déplacement de l'ensemble des nœuds du sous-arbre

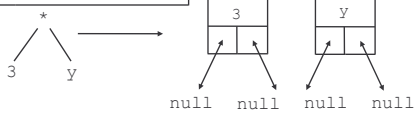
Implantation contiguë par niveaux – 5

- Problème majeur :
  - mauvaise occupation mémoire
  - occupation à 100% dans le cas des arbres binaires parfaits
  - Le déplacement de sous-arbres est coûteux
- Lemme
  - Un arbre binaire A est parfait ssi
    ∀x nœud de A, code(x) ≤ taille(A)
  - h(A) = $\lfloor \log_2 taille(A) \rfloor$

Implantation chaînée (1)

- La plus naturelle

```
Typedef struct tnoeud {
  int valeur;
  struct tnoeud * gauche;
  struct tnoeud * droit;
} noeud;
typedef noeud * arbre;
```



Implantation chaînée (2)

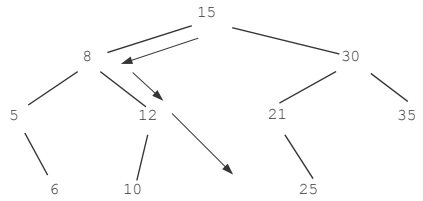
- algos efficaces pour la recherche et l'exploration
- Ajout/suppression de nœuds
  - modification d'une référence
- déplacement d'un sous arbre
  - Modification de 2 références

Définition: Arbres binaires de recherche

- arbre binaire construit sur des éléments qui possèdent une clé d'identification
  - pour tout nœud, sa clé est
    - supérieure à toutes celles de ses descendants de gauche
    - inférieure à toutes celles de ses descendants de droite
  - la liste correspondant au parcours infixe est la liste ordonnée par clé croissante
  - implantation sous forme chaînée
- Def équivalente: arbre binaire tel qu'un parcours infixé liste les éléments dans l'ordre croissant

exemple

- recherche/ajout de 13:



Recherche

- parcours dans l'arbre, depuis la racine jusqu'à l'élément recherché (ou une feuille si l'élément n'existe pas) en branchant à chaque nœud en fonction de la valeur de la clé
- Complexité au pire: h(a)

```
Arbre recherche(int e, arbre a){
  if (estVide(a))
    return arbreVide();
  else if (a->valeur == e)
    return a;
  else if (a->valeur > e)
    return recherche(e, a->droit);
  else
    return recherche(e, a->gauche);
}
```


Ajout

- recherche des valeurs récemment ajoutées
  - ajout à la racine
- recherche des valeurs anciennes
  - ajout aux feuilles

Ajout d'une feuille

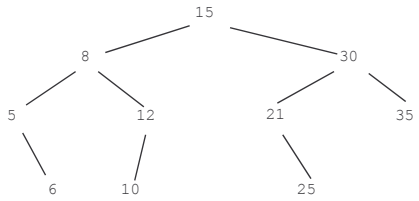
- recherche de la clé de l'élément que l'on essaye d'insérer
  - **si** élément existant **alors** rien à faire
  - **sinon** la recherche s'est arrêtée sur un arbre vide, qu'il suffira de remplacer par l'élément à insérer
- complexité au pire de h(a)

Ajout d'une feuille

```
Arbre ajouter(int e, arbre a){
  if (estVide(a))
    a= creeNoeud(e, arbreVide(), arbreVide());
  else if (a->valeur > e)
    a->droit=ajouter(e, a->droit);
  else
    a->gauche=ajouter(e, a->gauche);
  return a;
}
```

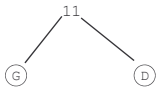
Ajout à la racine : exemple (1)

- Ajout de 11 à l'arbre suivant :



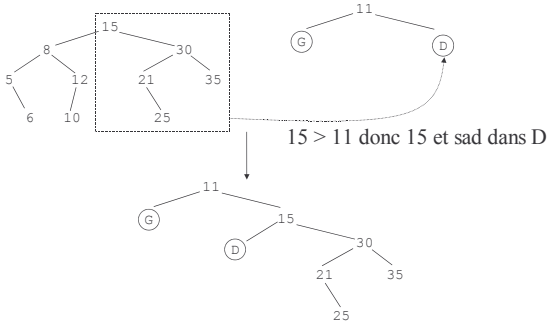
Ajout à la racine : exemple (2)

Nouvel arbre :

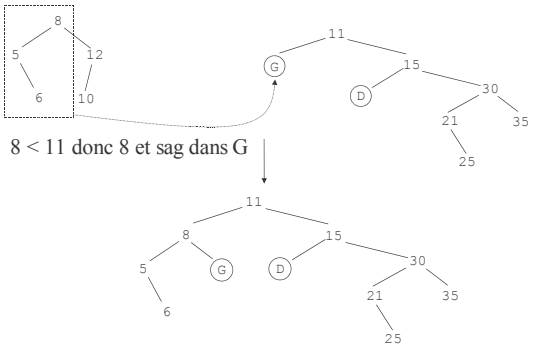


- Comment déterminer efficacement les éléments de l'arbre d'origine
  - qui sont supérieur à 11 : ils iront dans D
  - ceux qui sont inférieur à 11 : ils iront dans G
- Un algorithme inefficace consiste à insérer tous les éléments de l'arbre d'origine dans un arbre ne contenant que 11.

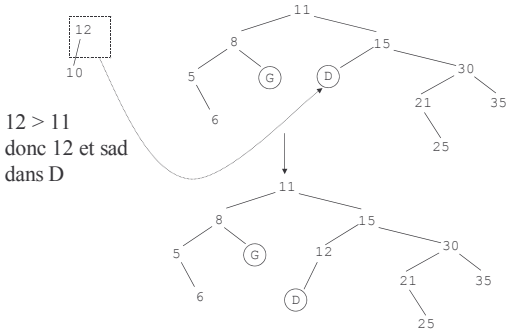
Ajout à la racine : exemple (3)



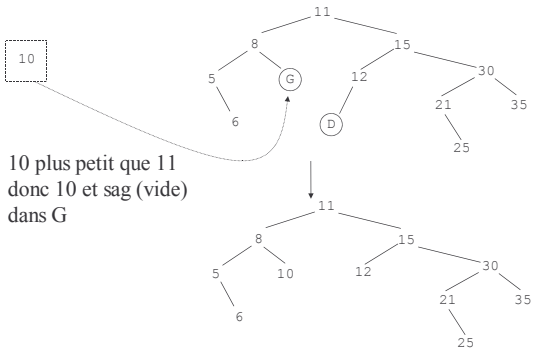
Ajout à la racine : exemple (4)



Ajout à la racine : exemple (5)



Ajout à la racine : exemple (6)

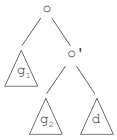


Généralisation (1)

- Soit un arbre `a = <o', g, d>`
- Ajouter le nœud `o` à `a`, c'est construire l'arbre `<o, a1, a2>` tel que :
  - `a1` contienne tous les nœuds dont la clé est inférieure à celle de `o`
  - `a2` contienne tous les nœuds dont la clé est supérieure à celle de `o`

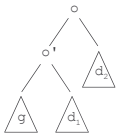
Généralisation (2)

- `si la_clé(o) < la_clé(o')`
  - `a1 = g1 et a2 = <o', g2, d>`
- `g1` = nœuds de g dont la clé est inférieure à la clé de o
- `g2` = nœuds de g dont la clé est supérieure à la clé de o



Généralisation (3)

- `si la_clé(o) > la_clé(o')`
  - `a1 = <o', g, d1> et a2 = d2`
- `d1` = nœuds de d dont la clé est inférieure à la clé de o
- `d2` = nœuds de d dont la clé est supérieure à la clé de o



Généralisation (4)

```
void coupure(int e, arbre a, arbre * g, arbre * d){
  if (estVide(a)){
    *g=arbreVide();
    *d=arbreVide();
  } else if (e < a->valeur) {
    *d=a;
    coupure(e, a->gauche, g, &((*d)->gauche));
  }else {
      *g=a;
      coupure(e, a->droit, &((*g)->droit),d);
    }
}
arbre ajouterRacine(int e, arbre a){
  arbre g=NULL,d=NULL;
  coupure(e,a,&g,&d);
  a=cree_noeud(e,g,d);
  return a;
}
```


Suppression

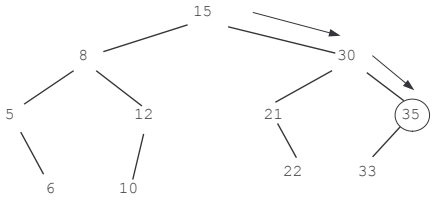
- Recherche du nœud
  - si feuille, suppression simple
  - si nœud interne au sens large, suppression du nœud et raccordement du sous-arbre
  - si nœud interne, suppression du nœud et remplacement soit par
    - le nœud du sous-arbre gauche dont la clé est la plus grande
    - le nœud du sous-arbre droit dont la clé est la plus petite
- Complexité
  - complexité au pire de h(a)

Suppression (2)

```
Arbre supprimerRacine(arbre a){
  int m;
  if (estVide(a->gauche)){
   atmp=a;
   a=a->droit;
   free(atmp);
  } else {
  m=max(a->gauche);
  a->gauche=supprimerMax(a->gauche);
  a->valeur=m;
 }
  return a;
}
```

Suppression(3): max

- Le plus grand élément d'un arbre binaire de rechercher sera à droite de tous les noeuds rencontrés



Suppression(4): max

```
int max(arbre a)
{
  assert( ! estVide(a));
  if (estVide(a->droit))
    return a->valeur;
  else return max(a->droit);
}
```


Suppression (5)

```
Arbre supprimer(int e, arbre a){
  if (!estVide(a)){
   if (a->valeur == e)
     return supprimeRacine(a);
   else if (e < a->valeur)
     a->gauche=supprimer(e, a->gauche);
   else
     a->droit=supprimer(e, a->droit);
  }
  return a;
}
```


Notes sur la complexité

- Les différentes opérations ont une complexité au pire de h(a)
  - $\lfloor \log_2 n \rfloor \leq h(a) \leq n-1$
  - Pour les arbres complets, complexité en $O(\log_2 n)$
  - Pour les arbres dégénérés filiformes, complexité en O (n)
- La complexité dépend de la forme de l'arbre, qui dépend des opérations d'ajout et de suppression
  - ajout d'éléments par clés croissantes → arbre dégénéré
  - Pour un arbre remplit aléatoirement, la profondeur moyenne est un $O(\log_2 n)$
  - but = équilibrer les arbres en hauteur

Notes sur la complexité(2)

- Pour minimiser la complexité, il faut travailler sur des arbres aussi peu hauts que possible.
  - Compromis entre la hauteur de l'arbre obtenu et la complexité des opérations d'ajout/suppression
  - On travaille sur des arbres dont la hauteur est proche de l'optimal sans être optimaux
- Exemples:
  - Arbres AVL: pour tout noeud, h(sag) et h(sad) diffèrent au plus d'un. h(arbre) = O(log n)
  - Arbres rouges-noir: min(h(f)) et max(h(f)) diffèrent d'un facteur 2 au plus.
- Généralisation des ABR: arbres 2-3-4, B-Arbres