Pimp A - info121A

Programmation IMPérative Avancée

Burkhart Wolff
Frédéric Vernier

Programme

- ~~*Tri*~~
- ~~*Récursivité*~~
- ~~*Dichotomie*~~
- *Arbre*
- *Graphe*
- *Examen blanc*

Recherche linéaire

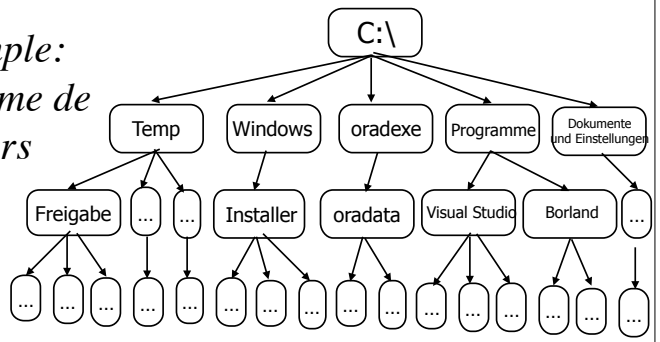
Diviser pour mieux régner

Traverse et Calculs dans Arbres

- *Arbres sont omniprésent !*
  - *Examples*
  - *Représentations d'arbres*
  - *Parcours et Calculs*
  - *etc.*

Exemple: Arbres d'Hierarchie

- *Exemple: Systeme de fichiers*

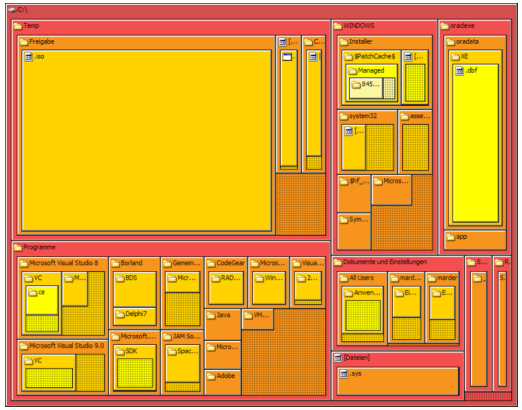


- *Chemins comme moyens de structurer* "name spaces" : C:\Temp\Freigabe\ ...

Exemple: Arbres d'Hierarchie

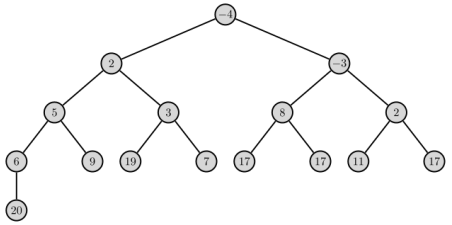
- *Exemple: Systeme de fichiers*

  (represen ration exo- tique …)



Example: Tas

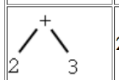
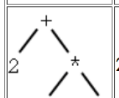
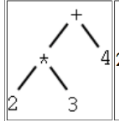
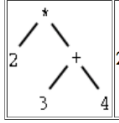
- *Tas (Heaps) père plus petits (plus grands) que ses fils …*



- *Structure importante si on veut trouver un élément de priorité max ou min VITE*
- *…peut être utilisé pour un tri (heapsort)*

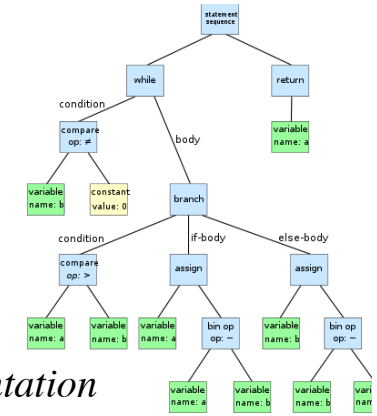
Example: Arbre Syntaxique

- *Expressions et différentes notations lineares et textuelles*

Tree	Infix notation	Postfix notation
	2 + 3	2 3 +
	2 + 3 * 4	2 3 4 * +
	2 * 3 + 4	2 3 * 4 +
	2 * (3 + 4)	2 3 4 + *

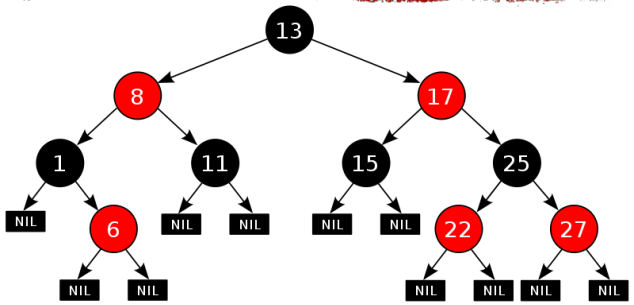
Example: Arbre Syntaxique

```
while (b ≠ 0)
  if (a > b)
        a := a − b
  else
        b := b − a
  return(a)
```



- *Programme et representation sous forme d'arbre syntaxique*

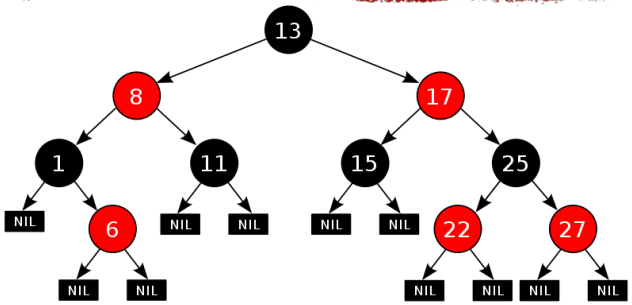
Example: Arbre Rouge-Noir



Proprietés:

1. Un nœud est soit rouge soit noir.
2. La racine est noire.
3. Le parent d'un nœud rouge est noir.
4. Le chemin de chaque feuille à la racine contient le même nombre de nœuds noirs.

Example: Arbre Rouge-Noir

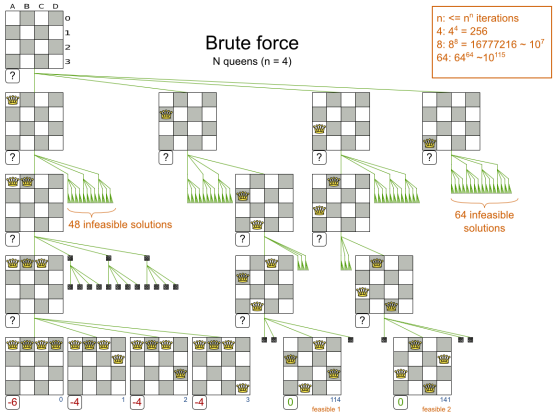


Proprietés

1. insertion et suppression en *O(ln n)*, donc trier O(n*(ln n))
2. recherche en O(ln n)

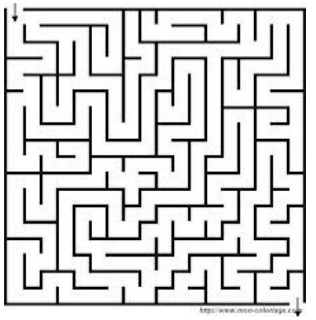
Example: Recherche en Jeux

- *n queens*
- *Recherche systéma–tique des solutions en jeux*
- *Arbre implicit*



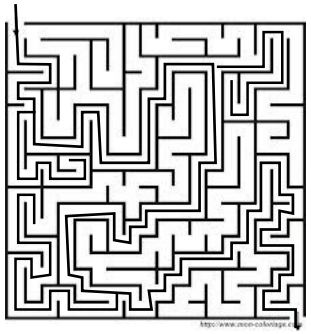
Example: Recherche Labyrinthe

- *La recherche d'une sortie d'un laby-rinthe …*



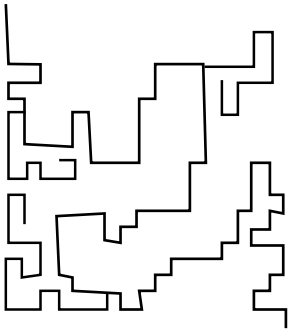
Example: Recherche Labyrinthe

- *La recherche d'une sortie d'un laby-rinthe …*



Example: Recherche Labyrinthe

- *La recherche d'une sortie d'un laby-rinthe représente un arbre implicitement*



Arbres, Arbres, Arbres …

- *Differentes formes d'arbres … [Wikipedia]*

v · d · m	Arbre enraciné	[masquer]
Arbre binaire	Arbre binaire de recherche · Arbre de fouille · Cartesian tree (en) · MVP Tree (en) · Top tree (en) · T-tree (en)	
Arbre équilibré	AA tree (en) · Arbre AVL · LLRB tree (en) · **Arbre bicolore** · Scapegoat tree (en) · Arbre splay · Treap	
Arbre B	B+ tree · B*-tree (en) · Bx-tree (en) · UB-tree (en) · 2-3 tree (en) · Arbre 2-3-4 · (a,b)-tree (en) · Dancing tree · Htree (en)	
Trie	Arbre des suffixes · Arbre radix · Arbre ternaire de recherche · X-fast trie (en) · Y-fast trie (en)	
Partition binaire de l'espace trees	Quadtree · Octree · Arbre kd · Implicit k-d tree (en) · vp-tree (en)	
Arbres non binaires	Exponential tree (en) · Fusion tree (en) · Interval tree (en) · PQ tree (en) · Range tree (en) · SPQR tree (en) · Van Emde Boas tree (en)	
Arbre de base de données spatiales	R-tree (en) · R+ tree (en) · R* tree (en) · X-tree (en) · M-tree (en) · Segment tree (en) · Hilbert R-tree (en) · Priority R-tree (en)	
Autres arbres	Arbre de Merkle · Arbre couvrant de poids minimal · Arbre syntaxique · Arbre syntaxique abstrait · Finger tree (en) · Order statistic tree (en) · Metric tree (en) · Cover tree (en) · BK-tree (en) · Doubly chained tree (en) · iDistance (en) · Link-cut tree (en) · Fenwick tree (en) · Tas · Tas binomial · Tas de Fibonacci	

Arbres : Definitions

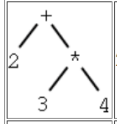
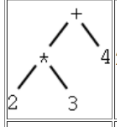
- *… sont des graphes particuliers (voir: théorie de graphes)*
- *… consistent des nœuds (lat: nodus) (autre terminologie: sommets)*
- *… et une relation entre des nœuds: père et fils. Un nœuds sans père: racine, sans fils: feuille.*
- *… le nombre maximale des fils dans un arbre M est appelé le degré de l'arbre M*
- *… arbres M avec degré 2 sont appelés binaires (3: ternaires, n-aires); le premier gauche(M), le deuxième droite(M).*

Arbre implicite:

- *Revision:*

  Réprèsentation dans un tableau

  arbre implicite dans le positionnement

Tree	Infix notation	Postfix notation
	2 + 3	2 3 +
	2 + 3 * 4	2 3 4 * +
	2 * 3 + 4	2 3 * 4 +

Code C++

```
#include <iostream>   ...

enum Kind   { Value, Operator };
enum OpKind { Mult, Add };


struct Entry {
    enum Kind kind;
    union {
        struct { int val; }   valueData;
        struct { OpKind op; } operatorData;
    } kindData;
};

typedef vector<Entry> Expr;

Expr expr;
```

Code C++

```
void mkValueEntry(int n) {
    Entry e;
    e.kind = Value;
    e.kindData.valueData.val = n;
    expr.push_back(e);
}

void mkOperatorEntry(OpKind opn) {
    Entry e;
    e.kind = Operator;
    e.kindData.operatorData.op = opn;
    expr.push_back(e);
}
```

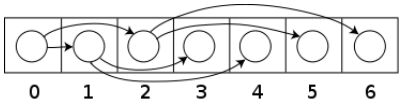
Code C++

```
void mk_SyntaxTree(){
/* Postcond: sets expr to expresseion "2 * 3 + 4" */
    mkValueEntry(2);
    mkValueEntry(3);
    mkOperatorEntry(Mult);
    mkValueEntry(4);
    mkOperatorEntry(Add);
}
int get_right(int x) {
    /* Pre: 2 <= x < expr.size() */
    return (x - 1);
}

int get_left(int x) {
    /* Pre: 2 <= x < expr.size() */
    ? ? ?
}
```

Arbre explicite

- *Structure de donné auxiliaire: un Tas <ref, X> (angl: Heap) avec positions.*



- *elements clé: operations*
  - *<ref> null*
  - *<ref> create (<X> x)*
  - *<X> get(<ref> ref)*

C++-code

```
#include <iostream>   ...

enum Kind   { Value, Operator };
enum OpKind { Mult, Add };

struct Node {
    enum Kind kind;
    union {
        struct { int val; }   valueData;
        struct { OpKind op;
                 int left, right;} operatorData;
    } kindData;
};

typedef vector<Node> Expr;

Expr expr;
```

C++-code

```
/* The Heap */

int null = -1;

int create_Node(Node node){
    expr.push_back(e);
    return(expr.size()-1)
}

Node get_Node(int n){
    return(expr[n]);
}
```

C++-code

```
// Constructors
int mkValueEntry(int n) {
    Entry e;
    e.kind = Value; e.kindData.valueData.val = n;
    return(create_Node(e));
}

void mkOperatorEntry(OpKind opn; int l,r) {
    Entry e;
    e.kind = Operator; e.kindData.operatorData.op = opn;
    e.kindData.operatorData.left=l;
    e.kindData.operatorData.right=r;
    return(create_Node(e));
}

// Selectors
int get_left(int x) {
    // Pre: 0 <= x < expr.size() && e.kind == Operator
    return (get_Node(x).kindData.operatorData.left);
} . . .
```

C++-code

```
// Recall: How to work on union types with switch'es.
// A printing function:

void print_Node(Node e){
    switch (e.kind) {
        case Value: cout<<(e.kindData.valueData.val)<<' ';break;
        case Operator:switch(e.kindData.operatorData.op){
             case Mult : cout<<"*["<< e.kindData.operatorData.left
                <<":" << e.kindData.operatorData.right<<"]";
                break;
             case Add : cout<<"+["<< e.kindData.operatorData.left
                <<":" << e.kindData.operatorData.right<<"]";
                break;
             };
             break;
    }
}
. . .
```

C++-code

```
int mk_SyntaxTree(){
/* Postcond: sets expr to expresseion "2 * 3 + 4" */
    int m1 = mkValueEntry(2);
    int m2 = mkValueEntry(3);
    int m3 = mkOperatorEntry(Mult,m1,m2);
    int m4 = mkValueEntry(4);
    int m5 = mkOperatorEntry(Add, m3,m4);
    return(m5);
}

// or just:

int mk_SyntaxTree(){
/* Postcond: sets expr to expresseion "2 * 3 + 4" */
    int m3 = mkOperatorEntry(Mult,mkValueEntry(2),
                                  mkValueEntry(3));
    return(mkOperatorEntry(Add, m3,mkValueEntry(4)));
}
```

Parcours d'arbres

- *Parcours en profondeur (depth-first)*
  - *Variante A: Parcours Préfixe.*

```
visiterPréfixe(Arbre A) :
   visiter(A)
   if nonVide(gauche(A))
      visiterPréfixe(gauche(A))
   if nonVide(droite(A))
      visiterPréfixe(droite(A))
```

Parcours d'arbres

- *Parcours en profondeur (depth-first)*
  - *Variante B: Parcours Infixe.*

```
visiterInfixe(Arbre A) :
   if nonVide(gauche(A))
      visiterInfixe(gauche(A))
   visiter(A)
   if nonVide(droite(A))
      visiterInfixe(droite(A))
```

Parcours d'arbres

- *Parcours en profondeur (depth-first)*
  - *Variante C: Parcours Postfixe.*

```
visiterPostfixe(Arbre A) :
   if nonVide(gauche(A))
      visiterPostfixe(gauche(A))
   if nonVide(droite(A))
      visiterPostfixe(droite(A))
   visiter(A)
```

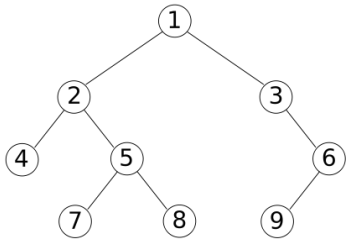
Parcours sur arbres

- *Parcours en profondeur (depth-first)*
  - *Variante D: Parcours Postfixe modifié pour l'evaluation …*

```
<C> evalPostfixe(Arbre A) :
  <C> res1, res2;
  if nonVide(gauche(A))
      res1 = evalPostfixe(gauche(A))
  if nonVide(droite(A))
      res2 = evalPostfixe(droite(A))
  return(calculer(A, res1, res2))
```

Parcours d'Arbres

- *Comparison des Parcours:*



  - *Rendu du parcours infixe : 4, 2, 7, 5, 8, 1, 3, 9, 6*
  - *Rendu du parcours postfixe : 4, 7, 8, 5, 2, 9, 6, 3, 1*
  - *Rendu du parcours préfixe : 1, 2, 4, 5, 7, 8, 3, 6, 9*

Parcours d'arbres

- *Parcours en largeur (breadth first)*
  - *Structure de donné auxiliaire: File <X> (angl: Queue)*
  - *First-in-first-out principle (FIFO)*
  - *elements clé: operations*
    - *File<X> FileVide*
    - *enfiler(File<X>, <X> x)*
    - *<X> defiler(File<X>) (inout)*

Parcours d'arbres

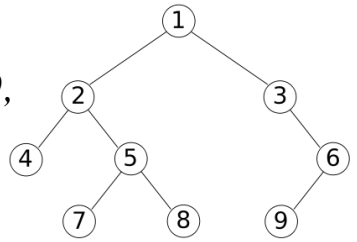
- *Parcours en largeur (breadth first)*

```
ParcoursLargeur(Arbre A){
   f = FileVide;
   enfiler(Racine(A), f);
   while (f != FileVide) {
       nœud = defiler(f);
       Visiter(nœud)    //On choisit de faire une opération
       if (nonVide(gauche(nœud))
           enfiler(gauche(nœud), f)
       if (nonVide(droite(nœud)))
           enfiler(droite(nœud), f)
   }
}
```

Parcours d'arbres

- *Rendu du parcours en largeur*

1, 2, 3, 4, 5, 6, 7, 8, 9,



Exo

- *Exo: Max filetree.*
- *Exo: postfix-tree.*
- *Exo: Print expr in infix-order (para)*
- *Exo: Adv: How to modify that (min para).*