DUT 2 Informatique – S3
M3103 – Algorithmique avancée

CM 04 et 05 – Algorithmique des arbres

Équipe pédagogique : Camille Kurtz, Jacques Alès-Bianchetti, Simon Fuhlhaber

prénom.nom@parisdescartes.fr

18 août 2016

Université Sorbonne Paris Cité

UNIVERSITÉ PARIS DESCARTES INSTITUT UNIVERSITAIRE DE TECHNOLOGIE

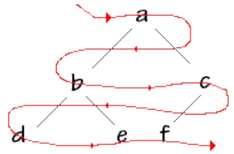
Plan

Plan

Introduction

Algorithmique des arbres

- Les arbres sont des **structures complexes** pour stocker et traiter des données
- Les algorithmes de traitement de ces structures s'expriment naturellement . . . de manière récursive
- Nous étudions certains de ces algorithmes permettant **la création, le traitement et le parcours** d'arbres
- Nous nous focaliserons ici sur l'étude des **arbres binaires** et du cas particulier des **arbres binaires de recherche**



Exemple du parcours d'un arbre binaire

$\Longrightarrow$ La manipulation et le traitement des arbres font appel à des **algorithmes fondamentaux en informatique**

Plan

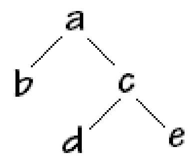
Retour sur les arbres binaires simples

Définitions sur les arbres binaires *simples*

- Les **arbres binaires** sont des arbres dont les nœuds .ont 2 fils au plus
- Un arbre binaire de taille n a une **hauteur moyenne** égale à $\log_2(n)$
- Pour tout arbre binaire de taille n et de hauteur h : . . .

$$h < n < 2^h - 1$$

- Tout arbre binaire de n nœuds possède $n + 1$ **chemins distincts** issus de sa racine
- Parmi ceux-là, n passent par des nœuds et 1 chemin est vide



Exemple d'un arbre binaire stockant des caractères

Rappel : Implantation des arbres binaires par chaînage

```
public class ArbreBin {
    private int valeur;
    private Arbre sousAbGauche=null;
    private Arbre sousAbDroit=null;

    // CONSTRUCTEURS
    public ArbreBin(int x) {
        this.valeur = x;
    }
    public ArbreBin(int x, ArbreBin g, ArbreBi
        this.valeur = x;
        this.sousAbGauche = g;
        this.sousAbDroit = d;
    }
    // ACCESSEURS
    public int getValeur() {
        return this.valeur;
    }
    public Arbre getSousArbreGauche() {
        return this.sousAbGauche;
    }
    public Arbre getSousArbreDroit() {
        return this.sousAbDroit;
    }
    // LE MAIN POUR TESTER
    public static void main(String[] arg) {
        ArbreBin b = new ArbreBin(2,new ArbreBin(1),new ArbreBin(4));
        ArbreBin c = new ArbreBin(10,new ArbreBin(8),new ArbreBin(12));
        ArbreBin racine = new ArbreBin(6,b,c);
        ...
     }
}
```

Explications

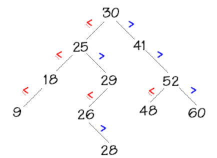
- L'accès à l'info de la racine de l'arbre s'effectue ainsi : racine.getValeur()
- L'accès au fils gauche de la racine : racine.getSousArbreGauche()
- L'accès au droit de la racine : racine.getSousArbreDroit()

Arbres binaires de recherche

Propriétés

Un **arbre binaire de recherche** (ABR) satisfait aux critères suivants :

- L'ensemble des étiquettes (dénommées clefs) est totalement ordonnées (en général des nombres)
- Les clefs de tous les nœuds du **sous-arbre gauche** d'un nœud X, sont inférieurs ou égaux à la clef de X
- Les clefs de tous les nœuds du **sous-arbre droit** d'un nœud X, sont supérieurs à la clef de X



Exemple

- Prenons par exemple le nœud (25) son sous-arbre droit est bien composé de nœuds dont les clefs sont supérieures à 25 : (29,26,28)
- Le sous-arbre gauche du nœud (25) est bien composé de nœuds dont les clefs sont inférieures à 25 : (18,9)

Arbres binaires de recherche : complexité

Complexité des opérations de base pour différentes structures de données

	Nombre d'opérations (en moyenne)		
Structure considérée	Ajout	Suppression	Recherche
Table	$\Theta(n)$	$\Theta(n)$	$\Theta(\log(n))$
Liste chaînée	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$

Motivations

- La **recherche en table**, en particulier la recherche dichotomique dans une table triée, nécessitent en moyenne . Theta(log(n)) comparaisons
- Avec **une liste chaînée** (plus gourmande en mémoire qu'un tableau) on aura en moyenne des temps de **suppression ou de recherche** au pire . . .

de l'ordre de Theta(n)L'ajout en fin de liste ou en début de liste demandant un temps constant $\Theta(1)$

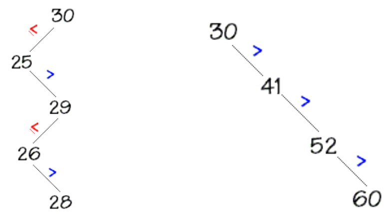
Remarque

Les arbres binaires de recherche . sont un bon compromis pour un temps équilibré entre ajout, suppression et recherche

Arbres binaires de recherche

Cas particulier : ABR dégénéré

On appelle **arbre binaire de recherche dégénéré** un ABR dont . . .



2 arbres binaires de recherche dégénérés

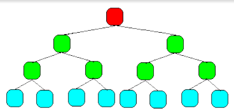
Exemple

- Dans les 2 cas nous avons affaire à une **liste chaînée** donc le nombre d'opérations pour la **suppression ou la recherche** est . . Theta(n)
- Il faudra utiliser une **catégorie spéciale d'arbres binaires (AVL)** qui restent équilibrés (leurs feuilles sont sur 2 niveaux au plus) pour assurer une recherche au pire en . . Theta(log(n))

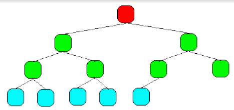
Arbres binaires de recherche

Cas particulier : arbre binaire parfait

Un arbre binaire parfait est un arbre binaire dont tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau. Dans ce cas l'arbre parfait est un **arbre binaire incomplet** et les feuilles du dernier niveau doivent être regroupées à partir de la gauche de l'arbre



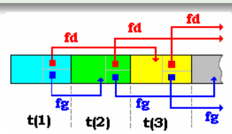
Un arbre binaire parfait complet



Un arbre binaire parfait incomplet

Implémentation facile dans un tableau

- Les nœuds de l'arbre sont dans les cellules du tableau, il n'y a **pas d'autres informations dans une cellule** du tableau
- **L'arborescence est simulée à travers un calcul d'indices** permettant de parcourir les cellules du tableau selon un certain « ordre » de numérotation correspondant en fait à un parcours hiérarchique de l'arbre



Complexité : construction d'un ABR

Idée sous-jacente

Pour construire un ABR, ...

Complexité au pire

- **Dans le pire des cas, l'arbre va se construire sous forme d'une liste** (arbre dégénéré). C'est le cas, par exemple, lorsque l'ensemble des valeurs est initialement trié
- Dans ce cas, **pour ajouter le p-ième élément, il faut effectuer** $(p-1)$ **comparaisons**
- Soit une complexité $T(n)$ donnée par la formule suivante :

  Somme pour p allant de 2 à n de p - 1 = 1/2 n² - 1/2 n

  ...

Ce qui correspond à une **complexité dans le pire des cas en** $\Theta(n^2)$

Complexité : construction d'un ABR

Complexité dans le meilleur des cas

- Le meilleur des cas correspond au cas où **l'arbre est complet** une fois construit : un arbre de hauteur h et dont toutes les branches de profondeur $(h-1)$ possèdent un arbre droit et un arbre gauche
- Dans ce cas **l'arbre est capable de stocker** le nombre d'éléments suivant :

Profondeur	Nombre d'éléments stockés
0	1
1	$1+2=3$
2	$1+2+4=7$
j	1+2+4 + ... + 2j

- Quand le j-ième niveau est rempli, il y a donc le nombre d'éléments suivant dans l'arbre :

Somme pour i allant de 0 à j de $2^i = 2^{i+1} - 1$

Introduction ABR Fonctions de base Parcours d'un arbre Tri par ABR AVL Application

Complexité : construction d'un ABR

Complexité dans le meilleur des cas (suite)

- Dans le meilleur des cas, pour insérer le p-ième élément, **il faut effectuer le nombre de comparaisons correspondant au nombre de niveaux remplis**
- Si le p-ième élément est inséré au niveau j, on a $p = 2^{j+1}-1 \Rightarrow p+1 = 2^{j+1} \Rightarrow j+1 = \log_2(p+1) \Rightarrow j =$ log2(p+1)+1
- **Ce résultat correspond ...**
- **Pour insérer les n éléments de l'ensemble à classer dans l'arbre**, il faut effectuer le nombre de comparaisons $T(n)$ suivant :

$$T(n) = \sum_{p=1}^{n} \left(\frac{\ln(p+1)}{\ln(2)} - 1 \right) \Rightarrow T(n) = \sum_{p=1}^{n} \left(\frac{\ln(p+1)}{\ln(2)} \right) - n$$

- Or, $\ln(a) + \ln(b) = \ln(a.b)$, d'où le résultat suivant :

$$T(n) = \frac{\ln\left(\prod_{p=1}^{n}(p+1)\right)}{\ln(2)} - n \Rightarrow T(n) = \ldots$$ (ln((n+1)! / ln(2)= - n

Complexité : construction d'un ABR

Complexité dans le meilleur des cas (suite)

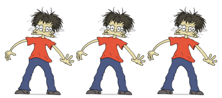
- Il s'agit du **calcul exact de la complexité pour construire un arbre binaire de recherche complet à n éléments** !
- Cependant, ce calcul est assez difficile à exploiter
- Nous allons montrer ici qu'**il s'agit d'un coût en $\Theta(n \ln(n))$**. Pour cela, il faut montrer l'égalité suivante : $\lim_{n \to +\infty} \frac{T(n)}{n \ln(n)} = \alpha$, avec $\alpha =$ constante
- On pose :

$$f(n) = \frac{\frac{\ln((n+1)!)}{\ln(2)} - n}{n \ln(n)} \Rightarrow f(n) = \frac{\ln((n+1)!)}{\ln(2) n \ln(n)} - \frac{1}{\ln(n)}$$

- Or on a $\lim_{n \to +\infty} \frac{1}{\ln(n)} = 0$. On obtient donc :

$$\lim_{n \to +\infty} f(n) = \ldots$$

lim (ln(n+1)) / ln(2n ln (n))



Complexité : construction d'un ABR

Complexité dans le meilleur des cas (suite)

- Pour calculer cette limite, **il faut en chercher un équivalent à l'infini**. Or, d'après la formule de Stirling, on a $n!$ qui est équivalent à l'infini à $\left(\frac{n}{e}\right)^n \sqrt{2}\sqrt{\pi n}$
- Soit $f(n)$ qui est équivalent à l'infini à :

$$\frac{\ln\left(\left(\frac{n}{e}\right)^n \sqrt{2}\sqrt{\pi n}\right)}{\ln(2) n \ln(n)} \quad \text{où } e \approx 2.72828 \text{ est le nombre d'Euler}$$

- Expression qui se simplifie ainsi (rappel : $\ln(e) = 1$) :

$$\frac{n \ln(n) - n + \frac{1}{2}\ln(2\pi) + \frac{1}{2}\ln(n)}{\ln(2) n \ln(n)} \Rightarrow \frac{1}{\ln(2)} - \frac{1}{\ln(n)} + \frac{1}{2}\frac{\ln(2\pi)}{\ln(2) n \ln(n)} + \frac{1}{2}\frac{1}{n}$$

- Or les trois derniers termes de cette expression tendent de manière évidente vers 0. D'où le résultat :

lim f(n) = 1 / ln(2)
n -> +infini

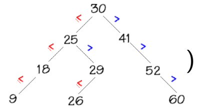
- On obtient donc : $\lim_{n \to +\infty} \frac{T(n)}{n \ln(n)} = \alpha$, d'où ... **CQFD**

La complexité de la construction d'un ABR complet . est en Theta(n ln (n))

Plan

1. Introduction
2. Arbres binaires de recherche (ABR) : complexité
3. **Fonctions de base sur la manipulation des ABR**
4. Parcours d'un arbre
5. Tri par ABR
6. Arbres binaires de recherche équilibrés (AVL)
7. Utilisation d'arbres binaires pour le traitement d'images

Recherche du maximum dans un ABR

Max()


```
/**
 * @param a un arbre
 * @return un entier representant le maximum dans l'arbre
 *          binaire de recherche
 */
public static int max(ArbreBin a) throws Exception{

    if (a == null) throw new Exception("Arbre vide !");

    //On cherche l'element le plus a droite
    if (a.getSousArbreDroit() != null)
        return .max(a.getFilsD());

    return .a.getValeur();
}
```

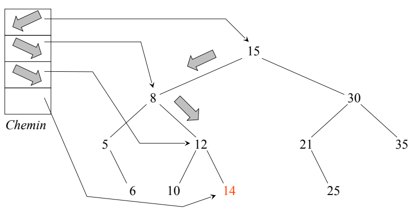
Ajout d'un élément dans un ABR

Question

Comment ajouter un élément dans un ABR ?

Exemple d'ajout

- Ajout de 14
- 14 est plus petit que 15
- 14 est plus grand que 8
- 14 est plus grand que 12 ⇐ repère d'un arbre vide
- **Remplacement de l'arbre vide par le nœud 14**



Ajout d'un élément dans un ABR

```
/**
* @param value la valeur a inserer dans l'arbre
*/
public void insertion(int valeur) {
    if (valeur == getValeur())
        return;  // la valeur est deja dans l'arbre

    if (valeur < getValeur()) {
        if (getSousArbreGauche() != null)
            .getFilsG().insertion(valeur);
        else
            this.sousAbGauche = ...valeur;
    }

    if (valeur > getValeur()) {
        if (getSousArbreDroit() != null)
            getFilsD().insertion(valeur);
        else
            this.sousAbDroit = .valeur:
    }
}
```

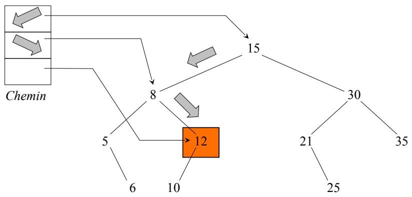
Recherche d'un élément dans un ABR

Question

Comment rechercher si un élément est dans un ABR ?

Exemple de recherche

- Recherche de 12
- 12 est plus petit que 15
- 12 est plus grand que 8
- **12 est trouvé !**



Recherche d'un élément dans un ABR

```java
/**
* @param "valeur" la valeur a recherche dans l'arbre
* @return un boolean indiquant si value a ete trouve ou non
*/
public boolean recherche(int valeur) {
    if (valeur == getValeur())
        return true;

    if ((valeur < getValeur()) && (getSousArbreGauche() != null))
        return (getSousArbreGauche().recherche(valeur));

    if ((valeur > getValeur()) && (getSousArbreDroit() != null))
        return (getSousArbreDroit().recherche(valeur));

    return false;
}
```

Complexité

Complexité

Un arbre binaire de recherche A de taille n (n nœuds) est une structure de données pour laquelle **les coûts de l'insertion et la recherche** sont en $\Theta(h(A))$:

- Dans le pire des cas (arbre dégénéré), le coût est en Theta(n) car il faut parcourir n étages
- En moyenne, . . . Theta(log(n)) car il faut parcourir $\log(n)$ étages

Exemple : $\log_2(12) = 3.58...$

Suppression d'un élément dans un ABR

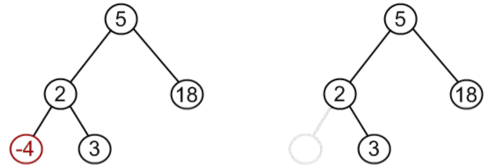
Question

Comment supprimer un élément dans un ABR ?

3 cas sont à considérer

Cas 1 **Le nœud à supprimer n'a aucun enfant** : . trivial

Exemple : On retire 4 de l'arbre



Suppression d'un élément dans un ABR

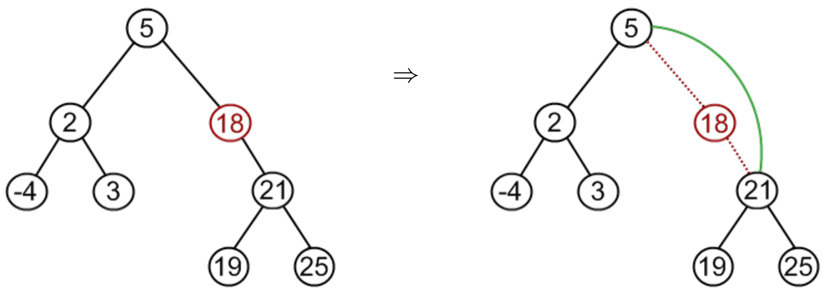
Question

Comment supprimer un élément dans un ABR ?

3 cas sont à considérer

Cas 2 **Le nœud à supprimer a un seul enfant** : . . .
retirer le noeud et rebrancher le fils droit au père

Exemple : On retire 18 de l'arbre



Suppression d'un élément dans un ABR

Question

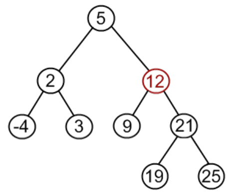
Comment supprimer un élément dans un ABR ?

3 cas sont à considérer

Cas 3 **Le nœud à supprimer a deux enfants** : Ce cas est le plus complexe :

1. trouver le min dans le sous arbre droit
2. remplacer le noeud à supprimer par le noeud
3. réappliquer la suppression sur le sous arbre droit pour éliminer la valeur dupliquée

Exemple : On retire 12 de l'arbre



Suppression d'un élément dans un ABR

Question

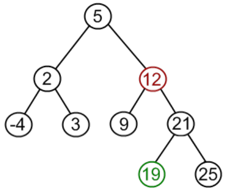
Comment supprimer un élément dans un ABR ?

3 cas sont à considérer

Cas 3 **Le nœud à supprimer a deux enfants** : Ce cas est le plus complexe :

1. trouver le min dans le sous arbre droit
2. remplacer le noeud à supprimer par le noeud
3. réappliquer la suppression sur le sous arbre droit pour éliminer la valeur dupliquée

Rechercher le min dans le sous-arbre droit. Ici le min est 19.



Introduction ABR **Fonctions de base** Parcours d'un arbre Tri par ABR AVL Application

Suppression d'un élément dans un ABR

Question

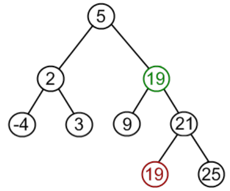
Comment supprimer un élément dans un ABR ?

3 cas sont à considérer

Cas 3 **Le nœud à supprimer a deux enfants** : Ce cas est le plus complexe :

1. trouver le min dans le sous arbre droit
2. remplacer le noeud à supprimer par le noeud
3. réappliquer la suppression sur le sous arbre droit pour éliminer la valeur dupliquée

Remplacer 12 par 19. Attention, 2 nœuds ont maintenant la même valeur !



Suppression d'un élément dans un ABR

Question

Comment supprimer un élément dans un ABR ?

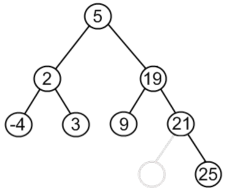
3 cas sont à considérer

Cas 3 **Le nœud à supprimer a deux enfants** : Ce cas est le plus complexe :

1. ...
2. ...
3. ...

ouais

Supprimer 19 du sous-arbre gauche.



Introduction ABR **Fonctions de base** Parcours d'un arbre Tri par ABR AVL Application

Suppression d'un élément dans un ABR

```java
/**
* @param value la valeur a supprimer dans l'arbre :
      pt d'entree dans la methode
*/
public static boolean supprimer(int valeur, ArbreBin root) {
    if (root == null)
      return false;
    else {
      if (root.getValeur() == valeur) {
        ArbreBin auxRoot = new ArbreBin(0);
        auxRoot.setSousArbreGauche(root);
        boolean result = root.supprimerRec(valeur, auxRoot);
        root = auxRoot.getSousArbreGauche();
        return result;
      } else {
        return root.supprimerRec(valeur, null);
      }
    }
}
```

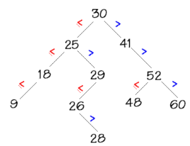
Suppression d'un élément dans un ABR

```
public boolean supprimerRec(int valeur, ArbreBin parent) {
    if (valeur < this.valeur) { // on part a gauche
      if (sousAbGauche != null)
        return sousAbGauche.supprimerRec(valeur, this);
      else return false;
    } else if (valeur > this.valeur) { // on part a droite
      if (sousAbDroit != null)
        return sousAbDroit.supprimerRec(valeur, this);
      else return false;
    } else {
      if (sousAbGauche != null && sousAbDroit != null) {
        //cas 3: 2 enfants
        this.valeur = sousAbDroit.minValeur();
        sousAbDroit.supprimerRec(this.valeur, this);
      } else if (parent.sousAbGauche == this) {
        //cas 1 et 2: >= 1 enfant
        parent.sousAbGauche = (sousAbGauche != null) ?
            sousAbGauche : sousAbDroit;
      } else if (parent.sousAbDroit == this) {
        parent.sousAbDroit = (sousAbGauche != null) ?
            sousAbGauche : sousAbDroit;
      }
      return true;
    }
```

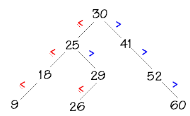

Suppression d'un élément dans un ABR

```
public int minValeur() {
    if (left == null)
      return value;
    else
      return left.minValeur();
}
```

Tester si deux ABR sont égaux



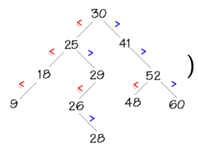
⇔ ?




```java
/**
* Teste si deux arbres sont egaux, meme valeurs et disposition
* @param a b les arbres a comparer
* @return un boolean indiquant si les arbres sont egaux
*/
public static boolean arbresEgaux(ArbreBin a, ArbreBin b) {
if ((a == null) && (b == null))
    return true;
if ((a == null) && (b != null))
    return false;
if ((a != null) && (b == null))
    return false;

// a et b != null, on peut acceder a leurs champs
if (a.getValeur() != b.getValeur())
    return false;
return arbresEgaux(a.getFilsG(), b.getFilsG()) && arbresEgaux(a.getFilsD(), b.getFilsD())

}
```


Introduction ABR Fonctions de base Parcours d'un arbre Tri par ABR AVL Application

Affichage d'un ABR

Affichage()

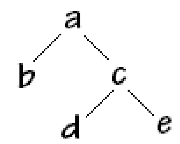

```java
public String toString() { // AFFICHAGE
    return toString("\t");
}

public String toString(String s) {
    if (sousAbGauche!=null) {
        if (sousAbDroit!=null)
            return(s+valeur+"\n"+sousAbGauche.toString(s+"\t")+
                                sousAbDroit.toString(s+"\t"));
        else
            return(s+valeur+"\n"+
                sousAbGauche.toString(s+"\t")+"\n");
    }else{
        if (droit!=null)
            return(s+valeur+"\n\n"+sousAbDroit.toString(s+"\t"));
        else
            return(s+valeur+"\n");
    }
}
```

Plan

Parcours d'un arbre binaire

Motivations

- Les arbres sont des graphes dont les nœuds **contiennent des informations**
- Différents algorithmes effectuant des opérations sur ces informations nécessitent de pouvoir . . .
- L'opération qui consiste à visiter tous les nœuds d'un arbre et d'y appliquer un traitement se dénomme . . .



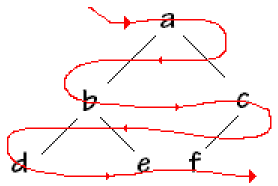
Différents algorithmes de parcours

1. Parcours en ... largeur
2. Parcours en ... profondeur
3. Algorithme de parcours en ... pré ordre, post ordre, ordre symétrique

Parcours en largeur (ou hiérarchique)

Stratégie

- La stratégie consiste à explorer chaque nœud d'un niveau donné . . . de gauche à droite , puis à passer au niveau suivant
- Cet algorithme nécessite l'utilisation d'un file du type FIFO (« First In First Out ») dans laquelle l'on stocke les nœuds



Parcours en largeur d'un arbre

Parcours en largeur (ou hiérarchique)

Stratégie

- La stratégie consiste à explorer chaque nœud d'un niveau donné . . . de gauche à droite , puis à passer au niveau suivant
- Cet algorithme nécessite l'utilisation d'un file du type FIFO (« First In First Out ») dans laquelle l'on stocke les nœuds

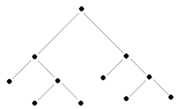
```
/**
* Affiche l'arbre selon un parcours en largeur */
public void parcoursLargeur() {
    File file = new File();
    file.ajouter(this);

    ArbreBin a,b;
    while (!file.estVide()) {
        a = (ArbreBin) file.retirer();
        System.out.println(a.getValeur()); //Traitement du noeud
        b = a.getSousArbreGauche();
        if (b != null) file.ajouter(b);
        b = a.getSousArbreDroit();
        if (b != null)  file.ajouter(b);
    }
}
```

Parcours en profondeur

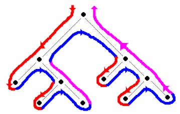
Stratégie

Un parcours est en **profondeur** lorsque, si l'arbre n'est pas vide, le parcours de l'un des deux sous-arbres est terminé avant que ne commence celui de l'autre

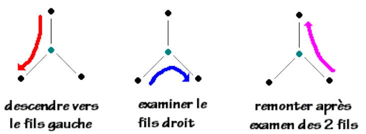
- La stratégie consiste à **descendre jusqu'aux feuilles** d'un nœud, puis lorsque toutes les feuilles du nœud ont été visitées, l'algorithme **remonte au nœud supérieur le plus bas** dont les feuilles n'ont pas encore été visitées
- En général ce parcours s'effectue en commençant par le fils gauche (« profondeur par la gauche »), puis en examinant le fils droit



Exemple d'arbre binaire



Parcours en profondeur par la gauche



Chaque nœud a été examiné selon les principes du parcours en profondeur

Parcours en profondeur

Remarque

Il existe **trois variantes de cet algorithme**, celles qui constituent des parcours ordonnés de l'arbre en fonction de l'application du traitement de l'information située aux nœuds. Chacun de ces 3 parcours définit un ordre implicite (préfixé, infixé, postfixé) sur le traitement des données contenues dans l'arbre.

- Parcours **préfixé**
- Parcours **infixé**
- Parcours **postfixé**

```java
/**
* Affiche l'arbre selon un parcours prefixe
*/
public void parcoursPrefixe() {
    System.out.println(getValeur()); //Traitement du noeud
    if (getSousArbreGauche() != null)
        getSousArbreGauche().parcoursPrefixe();
    if (getSousArbreDroit() != null)
        getSousArbreDroit().parcoursPrefixe();
}
```

Parcours en profondeur

Remarque

Il existe **trois variantes de cet algorithme**, celles qui constituent des parcours ordonnés de l'arbre en fonction de l'application du traitement de l'information située aux nœuds. Chacun de ces 3 parcours définit un ordre implicite (préfixé, infixé, postfixé) sur le traitement des données contenues dans l'arbre.

- Parcours **préfixé**
- Parcours **infixé**
- Parcours **postfixé**

```
/**
* Affiche l'arbre selon un parcours infixe
*/
public void parcoursInfixe() {
    if (getSousArbreGauche() != null)
        getSousArbreGauche().parcoursInfixe();
    System.out.println(getValeur()); //Traitement du noeud
    if (getSousArbreDroit() != null)
        getSousArbreDroit().parcoursInfixe();
}
```


Parcours en profondeur

Remarque

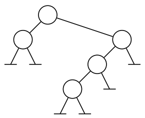
Il existe **trois variantes de cet algorithme**, celles qui constituent des parcours ordonnés de l'arbre en fonction de l'application du traitement de l'information située aux nœuds. Chacun de ces 3 parcours définit un ordre implicite (préfixé, infixé, postfixé) sur le traitement des données contenues dans l'arbre.

- Parcours **préfixé**
- Parcours **infixé**
- Parcours **postfixé**

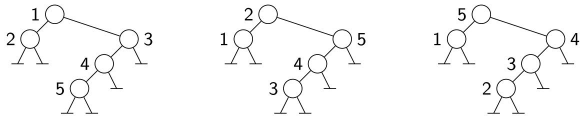
```
/**
* Affiche l'arbre selon un parcours postfixe
*/
public void parcoursPostfixe() {
    if (getSousArbreGauche() != null)
        getSousArbreGauche().parcoursPostfixe();
    if (getSousArbreDroit() != null)
        getSousArbreDroit().parcoursPostfixe();
    System.out.println(getValeur()); //Traitement du noeud
}
```

Parcours en profondeur

Pour l'arbre :



les ordres de traitement des nœuds sont, selon les parcours :



préfixe *infixe* *suffixe*

Complexité du parcours

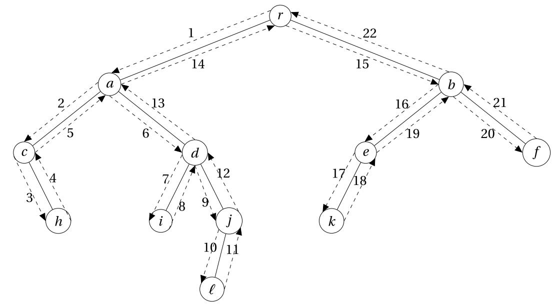
- Pour parcourir un arbre A, la complexité est proportionnelle à la taille de A, c'est-à-dire le nombre d'éléments que l'arbre contient
- **La complexité algorithmique du parcours est . . .** (où n est le nombre d'éléments dans l'arbre) dans tous les cas

Parcours en profondeur

Autre définition

On se balade autour de l'arbre en suivant les pointillés dans l'ordre des numéros. A partir de ce contour, on définit trois parcours des sommets de l'arbre :

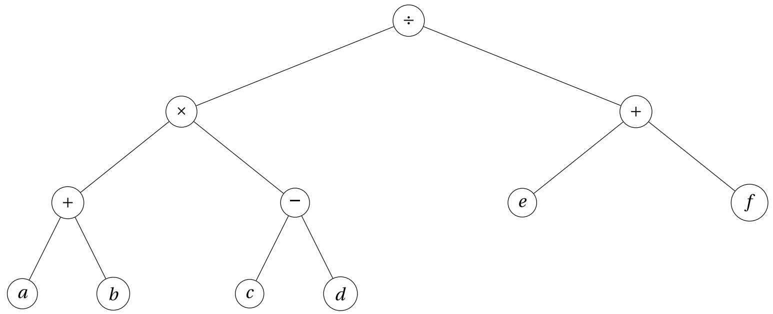
- **ordre préfixe** : on liste chaque sommet la première fois qu'on le rencontre dans la balade. Ce qui donne ici : r, a, c, h, d, i, j, l, b, e, k, f
- **ordre infixe** : on liste chaque sommet ayant un fils gauche la seconde fois qu'on le voit et chaque sommet sans fils gauche la première fois qu'on le voit. Ce qui donne ici : c, h, a, i, d, l, j, r, k, e, b, f
- **ordre postfixe** : on liste chaque sommet la dernière fois qu'on le rencontre. Ce qui donne ici : h, c, i, l, j, d, a, k, e, f, b, r



Parcours d'un arbre binaire

Exemple d'utilisation des parcours

- **Écrire les sommets de l'arbre pour les ordres préfixe, infixe, postfixe**
- Pour le parcours infixe, on ajoute une parenthèse ouvrante quand on entre dans un sous-arbre et on ajoute une fermante quand on quitte ce dernier



Modélisation d'une équation par un arbre binaire

Parcours d'un arbre binaire

Exemple d'utilisation des parcours

- **Écrire les sommets de l'arbre pour les ordres préfixe, infixe, postfixe**
- Pour le parcours infixe, on ajoute une parenthèse ouvrante quand on entre dans un sous-arbre et on ajoute une fermante quand on quitte ce dernier

Résultat

- **Préfixe** (notation polonaise) : $\div, \times, +, a, b, -, c, d, +, e, f$
- **Postfixe** (polonaise inverse) : $a, b, +, c, d, -, \times, e, f, +, \div$
- **Infixe** : $a, +, b, \times, c, -, d, \div, e, +, f$
  Avec un ajout de parenthèses (ouvrante en rencontrant le nœud racine du sous arbre pour la première fois et fermante lorsqu'on le rencontre pour la dernière fois) : $(a + b) \times (c - d) \div (e + f)$

L'infixe nécessite cette convention pour lever les ambiguïtés. La préfixe consiste à voir les opérateurs comme des fonctions de deux variables :

$$\div, \times, +, a, b, -, c, d, +, e, f = \ldots$$ /(* [+(a,b) - (c,d)])

Idem avec la postfixe mais avec la fonction écrite sur la droite

Plan

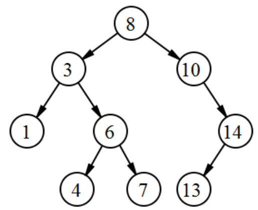
Tri par ABR

Principe du tri par arbre binaire de recherche (ABR)

Soit $T[n]$ un tableau à trier :

1. On fabrique un ABR contenant n données
2. On effectue le le parcous infixe

$T[n]$	8	1	4	7	13	14	10	6	3	Non trié



Exemple d'un ABR pour $n = 9$

Tri par ABR

Complexité en moyenne

- Si la liste est suffisamment aléatoire, la hauteur de l'ABR est $\log n$ et sa construction prend un temps $\Theta(n \log n)$
- **Complexité(Tri par ABR)**
  = Complexité(Création Arbre) + Complexité(Parcours)
  = ...
  Theta(n log(n) + Theta(n) = Theta (n log (n))
- **Pour la complexité spatiale**, il faut stocker l'arbre d'où une taille $\boldsymbol{\Theta(n)}$

Complexité au pire

Le cas le pire (toutes les valeurs de la liste décroissent) conduit à un arbre de hauteur n. Le temps est ...

Optimisations et variantes possibles

L'usage d'arbres AVL (arbres binaires de recherche automatiquement équilibrés) pallie cet inconvénient. L'algorithme est alors en $\Theta(n \log n)$ dans le pire des cas, au détriment d'une programmation délicate.

Tri par ABR

Remarques

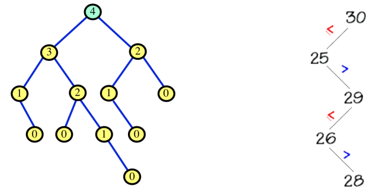
- **Le tri par ABR à** ... les mêmes caractéristiques que le tri rapide
- Le seul inconvénient de ce tri est qu'il nécessite la construction d'un ABR, qui **requiert un espace mémoire relativement important**
- **Le tri rapide**, lui, s'effectue en utilisant l'espace mémoire utilisé par le tableau à trier, il ne **nécessite donc pas d'espace mémoire supplémentaire** pour construire une nouvelle structure assez compliquée
- Le tri fusion utilise également un espace mémoire important, mais sa complexité en $\Theta(n \log(n))$ est garantie, quel que soit l'ensemble à trier

Plan

Arbres binaires de recherche

Pire des cas : ABR déséquilibré

Un arbre binaire de recherche est **déséquilibré quand les** hauteurs des branches de l'arbre sont fortement inégales



arbres binaires de recherche déséquilibré (gauche) et dégénéré (droite)

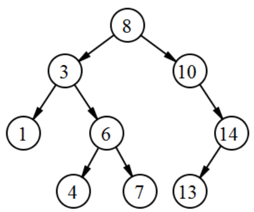
Complexité

- Dans le pire des cas le nombre d'opérations pour la **suppression ou la recherche** est en $\Theta(n)$
- On perd donc ici la complexité $\Theta(\log_2(n))$ qu'on a pour la recherche, l'insertion et la suppression d'un élément dans un ABR « bien » équilibré
- Il faudra utiliser une **catégorie spéciale d'arbres binaires (AVL)** . . .

Arbres binaires de recherche équilibrés (AVL)

Définition d'un AVL

- Pour résoudre ce problème d'équilibre, on utilise une **catégorie spéciale d'arbres binaires de recherche (AVL)** qui restent équilibrés
- Un arbre binaire est **un arbre équilibré AVL** (*Adelson-Velskii et Landis*) si, pour tout sommet, les hauteurs des sous-arbres gauche et droit différents d'au plus 1

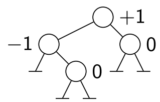


Exemple d'un ABR équilibré (AVL)

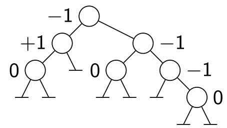
Équilibre d'un arbre

Définition

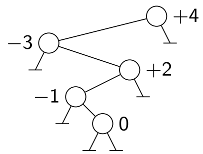
- **L'équilibre d'un arbre binaire** est un entier qui vaut 0 si l'arbre est vide et la différence des hauteurs des sous-arbres gauche et droit de l'arbre sinon
- Un **arbre binaire est équilibré** lorsque l'équilibre de **chacun de ses sous-arbres** non vides n'excèdent pas 1 en valeur absolue



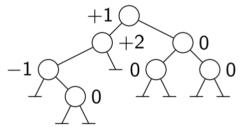
équilibré



équilibré



non équilibré



non équilibré

Tester l'équilibrage d'un ABR

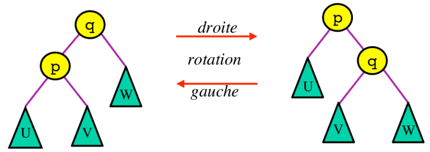
```
public int hauteur(){
 if (this.valeur == null){
  return 0;
 }else{
  int hSAG = sousArbGauche.hauteur();
  int hSAD = sousArbDroit.hauteur();
  return 1 + ((hSAD>hSAG)?hSAD:hSAG);
 }
}

public boolean estEquilibre(){
 if (this.valeur == null){
   return true;
 }else{
   return
         Math.abs(getFilsG().hauteur() - getFilsDroit().hauteur()) <= 1 ))
         &&
  }      getFilsG().estEquilibre() && getFilsD().estEquilibre();
}
```


Rotations et équilibrage

Définition

Les rotations gauche et droite transforment un arbre

- elles préservent l'ordre infixe
- elles se réalisent en temps constant

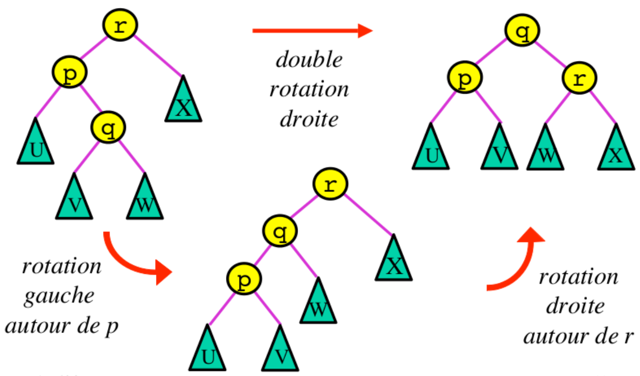



```
//De la gauche vers la droite
public void rotationDroite(){
  double tv = getValeur();
  setValeur(sousArbreGauche.getValeur());
  sousArbreGauche.setValeur(tv);
  ABR ta = this.sousArbreGauche;
  sousArbreGauche = this.sousArbreGauche.sousArbreGauche;
  ta.sousArbreGauche = ta.sousArbreDroit;
  ta.sousArbreDroit = sousArbreDroit;
  sousArbreDroit = ta;
}
```

Rotations

Proposition

- **Après une insertion** ou une suppression, il suffit de **deux rotations pour ré-équilibrer un arbre**
- Le maintient de l'équilibre est possible en temps constant

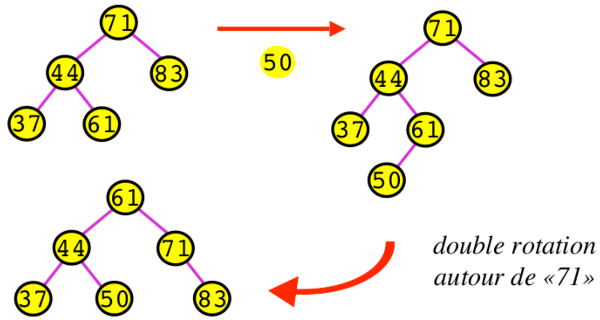


Introduction ABR Fonctions de base Parcours d'un arbre Tri par ABR AVL Application

Construire un AVL

Insertion dans un arbre équilibré

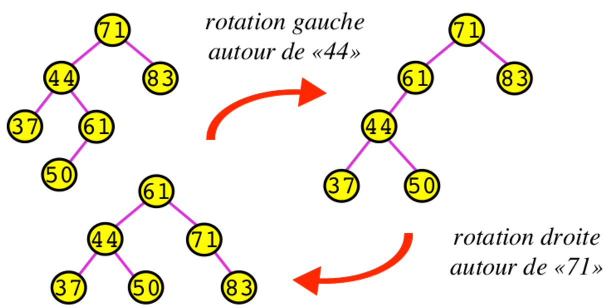
- **Pour obtenir un AVL à partir d'un ABR**, . . .
- `insertionEquilibre(valeur)` : insertion ordinaire dans un ABR, suivie de rééquilibrage



Construire un AVL

Insertion dans un arbre équilibré

- **Pour obtenir un AVL à partir d'un ABR**, . . .
- `insertionEquilibre(valeur)` : insertion ordinaire dans un ABR, suivie de rééquilibrage



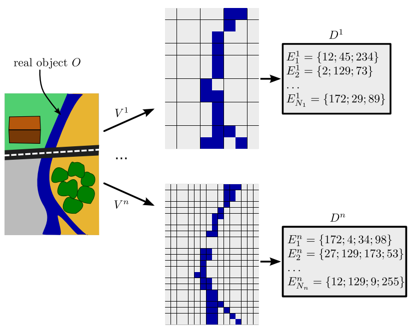
Introduction ABR Fonctions de base Parcours d'un arbre Tri par ABR AVL Application

Plan

Traitement d'images

Définition

- D'un point de vue informatique, **une image est une matrice de pixels** où chaque pixel p peut être représenté par une valeur v pour une image noir et blanc ou par un triplet (r, g, b) pour une image couleur
- Le **traitement d'images** est un ensemble d'opérations diverses permettant **de manipuler, d'améliorer, de restaurer le contenu** d'une image



2 images de la même scène avec différentes résolutions

Traitement d'images

Segmentation d'images

- La **segmentation d'images** est une opération de traitement d'images
- Elle consiste à la **partitionner une image en un ensemble de régions** connexes « primitives »
- Les frontières des régions obtenues correspondent **aux objets représentés dans l'image**
- La segmentation peut être utilisée pour la **reconnaissance des objets**

Segmentation d'une image de papillon

Traitement d'images

Segmentation d'images par arbres

La segmentation par arbres permet d'obtenir une **segmentation hiérarchique** où des objets à différentes tailles peuvent être extraits simultanément

Hiérarchie de segmentations basée régions

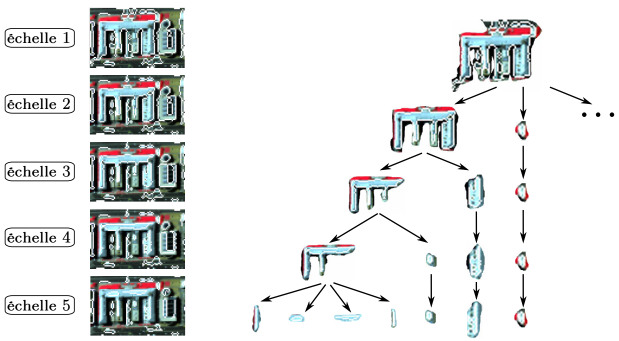
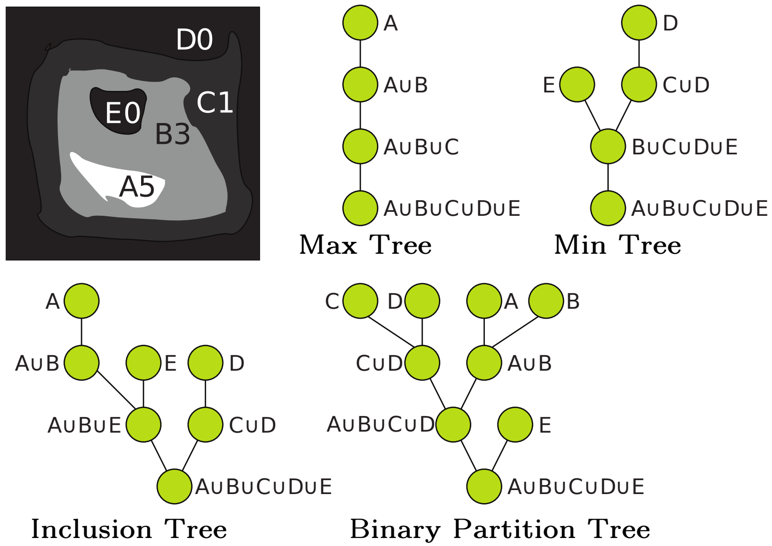


Image $\mathcal{I}$

Exemple de hiérarchie obtenue pour un toit de bâtiment

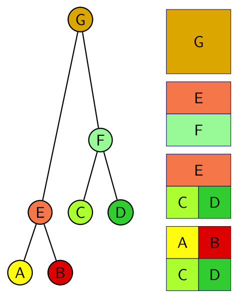
Différents types d'arbres



Arbres binaires de partition

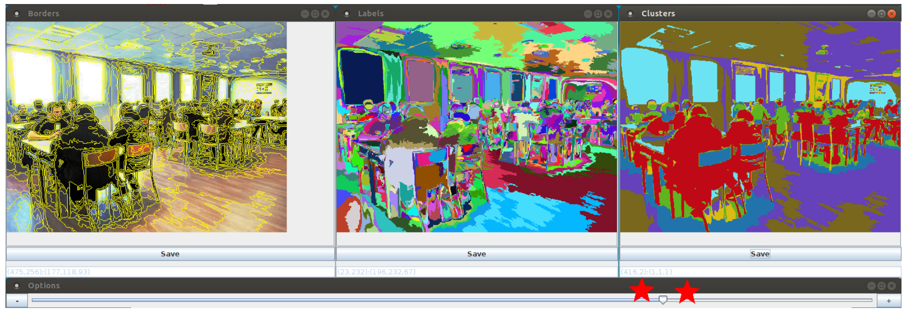
Définition d'un arbre binaire de partition

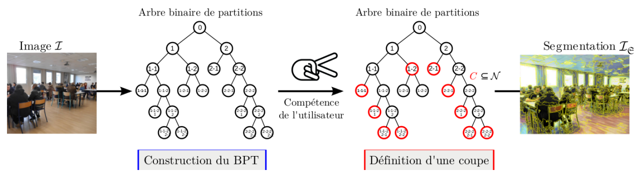
- C'est une **structure hiérarchique représentant une image**
- Les **feuilles de l'arbre** correspondent à une partition initiale (par exemple l'ensemble des pixels)
- Les **nœuds** représentent les régions formées par la **fusion de deux régions** filles qui **sont voisines et se ressemblent** (couleurs similaires)
- L'arbre est construit par le biais d'un **algorithme itératif de fusion de régions**



Arbres binaires de partition

Démonstration d'un logiciel de segmentation interactive par BPT





Références

Bibliographie

Des éléments de ce cours sont empruntés de [Cormen(2011), Passat(2007)]

T. Cormen.
Introduction à l'algorithmique.
Dunod, 2011.

N. Passat.
CM 4 – Arbres.
Module Structures de données et algorithmes fondamentaux, L3 mention Sciences du Vivant, Université Louis Pasteur, 2007.