

Arbres binaires

Alix Munier-Kordon et Maryse Pelletier

LIP6
Université P. et M. Curie
Paris

Algorithmique élémentaire

Plan du cours

- 1 Arbres binaires
- 2 Arbres binaires d'expressions

Arbres binaires : définitions (1)

Voici une définition **inductive** des arbres binaires.

Definition

Un *arbre binaire* T étiqueté sur un ensemble E est :

- soit l'arbre vide, noté \emptyset ,
- soit un triplet (x, G, D) où $x \in E$ et G, D sont des arbres binaires étiquetés sur E .

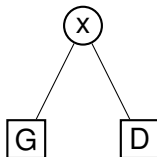
$$\left\{ \begin{array}{l} T = \emptyset \\ \text{ou} \\ T = (x, G, D) \end{array} \right.$$

Arbres binaires : définitions (2)

Definition

Si $T = (x, G, D)$ alors :

- x est la *racine* de T ,
- G est le *sous-arbre gauche* de T ,
- D est le *sous-arbre droit* de T .



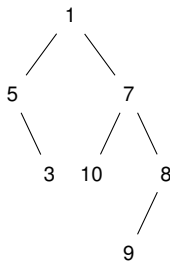
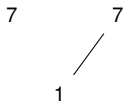
Arbres binaires : liens de parenté

Dans un arbre binaire $T = (x, G, D)$ on dit que :

- G et D sont les *fil*s (gauche et droit) de x ,
- x est le *père* de G et D ,
- G et D sont frères.

Exemples

$$E = \mathbb{N}$$



Représentation

- `AB(x, G, D)` renvoie un arbre de racine x , de sous-arbre gauche G et de sous-arbre droit D ,
- `ABvide()` renvoie l'arbre vide,
- `estABvide(T)` teste si l'arbre T est vide.

Pour un arbre T :

- `T.clef` désigne l'étiquette (ou clef) de T ,
- `T.gauche` désigne le sous-arbre gauche de T ,
- `T.droit` désigne le sous-arbre droit de T .

Complexité : en $\Theta(1)$ pour chaque primitive.

Définitions et preuves par induction

Les arbres binaires étant définis par induction, les définitions et preuves sur les arbres binaires peuvent être faites par **induction structurelle**.

- Définitions inductives de fonctions sur les arbres binaires :
 - cas de base : définition de la fonction pour l'arbre vide
 - la fonction étant définie pour les sous-arbres gauche et droit, on la définit pour l'arbre.
- Preuves inductives de propriétés sur les arbres binaires :
 - cas de base : preuve de la propriété pour l'arbre vide
 - on suppose la propriété vraie pour les sous-arbres gauche et droit et on montre qu'elle est vraie pour l'arbre.

Nœuds et feuilles

Definition

Les *nœuds* d'un arbre binaire sont sa racine, les nœuds de son sous-arbre gauche et les nœuds de son sous-arbre droit.

On note $\mathcal{N}(T)$ l'ensemble des nœuds de T .

Definition

Une *feuille* est un nœud dont les deux fils sont vides.

On note $\mathcal{F}(T)$ l'ensemble des feuilles de T .

Definition

Un *nœud interne* est un nœud qui n'est pas une feuille.

On note $\mathcal{I}(T)$ l'ensemble des nœuds internes de T .

Nœuds et feuilles

Définitions **inductives** des nœuds, feuilles, nœuds internes :

$$\mathcal{N}(T) = \begin{cases} \emptyset & \text{si } T = \emptyset \\ \{x\} \cup \mathcal{N}(G) \cup \mathcal{N}(D) & \text{si } T = (x, G, D) \end{cases}$$

$$\mathcal{F}(T) = \begin{cases} \emptyset & \text{si } T = \emptyset \\ \{x\} & \text{si } T = (x, \emptyset, \emptyset) \\ \mathcal{F}(G) \cup \mathcal{F}(D) & \text{si } T = (x, G, D) \end{cases}$$

$$\mathcal{I}(T) = \begin{cases} \emptyset & \text{si } T = \emptyset \text{ ou } T = (x, \emptyset, \emptyset) \\ \{x\} \cup \mathcal{I}(G) \cup \mathcal{I}(D) & \text{si } T = (x, G, D) \text{ avec } G \neq \emptyset \text{ ou } D \neq \emptyset \end{cases}$$

Taille d'un arbre binaire

La *taille* d'un arbre binaire est son nombre de nœuds.

Definition

La taille d'un arbre binaire est définie **inductivement** par :

$$n(T) = \begin{cases} 0 & \text{si } T = \emptyset \\ 1 + n(G) + n(D) & \text{si } T = (x, G, D) \end{cases}$$

```
def ABtaille(T):  
    if estABvide(T):  
        return 0  
    else:  
        return 1 + ABtaille(T.gauche) + ABtaille(T.droit)
```

Taille d'un arbre binaire

La *taille* d'un arbre binaire est son nombre de nœuds.

Definition

La taille d'un arbre binaire est définie **inductivement** par :

$$n(T) = \begin{cases} 0 & \text{si } T = \emptyset \\ 1 + n(G) + n(D) & \text{si } T = (x, G, D) \end{cases}$$

```
def ABtaille(T):  
    if estABvide(T):  
        return 0  
    else:  
        return 1 + ABtaille(T.gauche) + ABtaille(T.droit)
```

Hauteur d'un arbre binaire

La *hauteur* d'un arbre binaire est le plus grand nombre de nœuds que l'on peut rencontrer en suivant un chemin de la racine vers une feuille.

Definition

La *hauteur* d'un arbre binaire est définie **inductivement** par :

$$h(T) = \begin{cases} 0 & \text{si } T = \emptyset \\ 1 + \max(h(G), h(D)) & \text{si } T = (x, G, D) \end{cases}$$

```
def ABhauteur(T) :  
    if estABvide(T) :  
        return 0  
    else :  
        return 1 + max(ABhauteur(T.gauche), ABhauteur(T.droit))
```

Hauteur d'un arbre binaire

La *hauteur* d'un arbre binaire est le plus grand nombre de nœuds que l'on peut rencontrer en suivant un chemin de la racine vers une feuille.

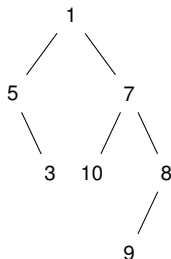
Definition

La *hauteur* d'un arbre binaire est définie **inductivement** par :

$$h(T) = \begin{cases} 0 & \text{si } T = \emptyset \\ 1 + \max(h(G), h(D)) & \text{si } T = (x, G, D) \end{cases}$$

```
def ABhauteur(T) :  
    if estABvide(T) :  
        return 0  
    else :  
        return 1 + max(ABhauteur(T.gauche), ABhauteur(T.droit))
```

Taille, hauteur : exemple



Taille : 7

Hauteur : 4

Complexité de $ABtaille$ et $ABhauteur$

Theorem

Pour un arbre T de taille n :

- *la complexité de la fonction $ABtaille(T)$ est en $\Theta(n)$*
- *la complexité de la fonction $ABhauteur(T)$ est en $\Theta(n)$.*

Preuve par induction structurelle.

Pour un arbre de taille n :

- on note $c(n)$ le nombre d'additions effectuées par $ABtaille$
- on note $d(n)$ le nombre d'additions effectuées par $ABhauteur$

On montre, par induction structurelle, que :

- $c(n) = 2n$
- $d(n) = n$

Complexité de `ABtaille` et `ABhauteur`

Theorem

Pour un arbre T de taille n :

- *la complexité de la fonction `ABtaille` (T) est en $\Theta(n)$*
- *la complexité de la fonction `ABhauteur` (T) est en $\Theta(n)$.*

Preuve par induction structurelle.

Pour un arbre de taille n :

- on note $c(n)$ le nombre d'additions effectuées par `ABtaille`
- on note $d(n)$ le nombre d'additions effectuées par `ABhauteur`

On montre, par induction structurelle, que :

- $c(n) = 2n$
- $d(n) = n$

Relations entre taille et hauteur

Theorem

Pour tout arbre de taille n et de hauteur h : $h \leq n \leq 2^h - 1$.

Corollary

Pour tout arbre de taille n et de hauteur h : $\log_2(n + 1) \leq h \leq n$.

Preuve du théorème par induction structurelle (en TD).

Le corollaire est une conséquence évidente du théorème.

Cas extrêmes. Pour une hauteur h fixée,

- arbre de taille minimum : arbre "longiligne"
- arbre de taille maximum : arbre "plein"

Relations entre taille et hauteur

Theorem

Pour tout arbre de taille n et de hauteur h : $h \leq n \leq 2^h - 1$.

Corollary

Pour tout arbre de taille n et de hauteur h : $\log_2(n + 1) \leq h \leq n$.

Preuve du théorème par induction structurelle (en TD).

Le corollaire est une conséquence évidente du théorème.

Cas extrêmes. Pour une hauteur h fixée,

- arbre de taille minimum : arbre "longiligne"
- arbre de taille maximum : arbre "plein"

Arbres binaires : égalité

L'égalité entre arbres binaires est définie par **induction structurelle**.

Definition

Deux *arbres binaires* T_1 et T_2 sont égaux si :

- $T_1 = \emptyset$ et $T_2 = \emptyset$
- ou bien $T_1 = (x_1, G_1, D_1)$, $T_2 = (x_2, G_2, D_2)$, avec $x_1 = x_2$, G_1 et G_2 égaux, D_1 et D_2 égaux.

Arbres binaires : égalité (suite)

Égalité entre arbres binaires :

```
def ABegal(T1, T2):  
    if estABvide(T1):  
        if estABvide(T2):  
            return True  
        return False  
    if estABvide(T2):  
        return False  
    return (T1.clef == T2.clef) and  
           ABegal(T1.gauche, T2.gauche) and  
           ABegal(T1.droit, T2.droit)
```

Complexité meilleur cas en $\Omega(1)$ et pire cas en $O(\min(n_1, n_2))$ (voir TD).

Parcours d'un arbre binaire

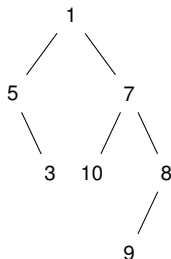
Trois parcours possibles :

- *parcours préfixe* : visiter la racine puis parcourir le sous-arbre gauche et enfin parcourir le sous-arbre droit
- *parcours infixe* : parcourir le sous-arbre gauche puis visiter la racine et enfin parcourir le sous-arbre droit
- *parcours suffixe* : parcourir le sous-arbre gauche puis parcourir le sous-arbre droit et enfin visiter la racine.

Le résultat d'un parcours est une liste.

Liste vide pour l'arbre vide.

Parcours : exemple



Parcours préfixe :
(1, 5, 3, 7, 10, 8, 9)

Parcours infixe :
(5, 3, 1, 10, 7, 9, 8)

Parcours suffixe :
(3, 5, 10, 9, 8, 7, 1)

Fonctions ABpref, ABinf, ABSuf

Parcours préfixe :

```
def ABpref(T):  
    if estABvide(T):  
        return []  
    else:  
        return [T.clef] + ABpref(T.gauche) + ABpref(T.droit)
```

Parcours infixe :

```
def ABinf(T):  
    if estABvide(T):  
        return []  
    else:  
        return ABinf(T.gauche) + [T.clef] + ABinf(T.droit)
```

Parcours suffixe :

```
def ABSuf(T):  
    if estABvide(T):  
        return []  
    else:  
        return ABSuf(T.gauche) + ABSuf(T.droit) + [T.clef]
```


Complexité de AB_{pref} , AB_{inf} , AB_{suf}

Theorem

Pour un arbre de taille n , la complexité de chacune des fonctions AB_{pref} , AB_{inf} et AB_{suf} est en $\Theta(n)$, si l'on représente les listes par des listes circulaires doublement chaînées.

Preuve par induction structurelle (voir TD).

Question : qu'en est-il si l'on représente les listes par des tableaux ou par des listes simplement chaînées ?

Tableaux : meilleur cas en $\Omega(n \log n)$ et pire cas en $O(n^2)$.

Listes simplement chaînées : meilleur cas en $\Omega(n)$ et pire cas en $O(n^2)$.

Complexité de AB_{pref} , AB_{inf} , AB_{suf}

Theorem

Pour un arbre de taille n , la complexité de chacune des fonctions AB_{pref} , AB_{inf} et AB_{suf} est en $\Theta(n)$, si l'on représente les listes par des listes circulaires doublement chaînées.

Preuve par induction structurelle (voir TD).

Question : qu'en est-il si l'on représente les listes par des tableaux ou par des listes simplement chaînées ?

Tableaux : meilleur cas en $\Omega(n \log n)$ et pire cas en $O(n^2)$.

Listes simplement chaînées : meilleur cas en $\Omega(n)$ et pire cas en $O(n^2)$.

Complexité de AB_{pref} , AB_{inf} , AB_{suf}

Theorem

Pour un arbre de taille n , la complexité de chacune des fonctions AB_{pref} , AB_{inf} et AB_{suf} est en $\Theta(n)$, si l'on représente les listes par des listes circulaires doublement chaînées.

Preuve par induction structurelle (voir TD).

Question : qu'en est-il si l'on représente les listes par des tableaux ou par des listes simplement chaînées ?

Tableaux : meilleur cas en $\Omega(n \log n)$ et pire cas en $O(n^2)$.

Listes simplement chaînées : meilleur cas en $\Omega(n)$ et pire cas en $O(n^2)$.

Complexité de ABpref (calculs)

```
def ABpref(T):  
    if estABvide(T):  
        return []  
    else:  
        return [T.clef] + ABpref(T.gauche) + ABpref(T.droit)
```

● Représentation par tableaux

Relation de récurrence : $c(n) = n + c(n_1) + c(n_2)$.

- Pire cas : $n_1 = n - 1$ à chaque appel récursif (arbre longiligne) d'où $O(n^2)$.
- Meilleur cas : $n_1 \approx n/2$ à chaque appel récursif (arbre plein) d'où $\Omega(n \log n)$.

● Représentation par listes chaînées

Relation de récurrence : $c(n) = n_1 + 1 + c(n_1) + c(n_2)$.

- Pire cas : $n_1 = n - 1$ à chaque appel récursif (arbre longiligne gauche) d'où $O(n^2)$.
- Meilleur cas : $n_1 = 0$ à chaque appel récursif (arbre longiligne droit) d'où $\Omega(n)$.

Exercice : recherche dans un arbre binaire

Recherche d'un élément x dans un arbre binaire T .

Principe. Comparer x à la racine de T . En cas de non-égalité, chercher x dans le sous-arbre gauche de T puis, éventuellement, dans le sous-arbre droit de T .

```
def ABcherche(x, T):  
    if estABvide(T):  
        return False  
    if x == T.clef:  
        return True  
    if ABcherche(x, T.gauche):  
        return True  
    return ABcherche(x, T.droit)
```

Exercice (voir TD)

- Prouver que `ABcherche(x, T)` se termine et retourne `True` si x est dans T et `False` sinon.
- Prouver que la complexité de `ABcherche(x, T)` est en $\Omega(1)$ dans le meilleur cas et en $O(n)$ dans le pire cas.

Exercice : recherche dans un arbre binaire

Recherche d'un élément x dans un arbre binaire T .

Principe. Comparer x à la racine de T . En cas de non-égalité, chercher x dans le sous-arbre gauche de T puis, éventuellement, dans le sous-arbre droit de T .

```
def ABcherche(x, T):  
    if estABvide(T):  
        return False  
    if x == T.clef:  
        return True  
    if ABcherche(x, T.gauche):  
        return True  
    return ABcherche(x, T.droit)
```

Exercice (voir TD)

- Prouver que `ABcherche(x, T)` se termine et retourne `True` si x est dans T et `False` sinon.
- Prouver que la complexité de `ABcherche(x, T)` est en $\Omega(1)$ dans le meilleur cas et en $O(n)$ dans le pire cas.

Arbres binaires d'expressions

Quatre opérations arithmétiques d'arité 2 : $+$, $-$, $*$, $/$.

Voici une définition **inductive** des arbres binaires d'expressions.

Definition

Un *arbre binaire d'expression* T est :

- soit un arbre réduit à une feuille dont l'étiquette est une valeur numérique,
- soit un triplet (x, G, D) où x est l'un des opérateurs $+$, $-$, $*$, $/$ et où G, D sont des arbres binaires d'expressions.

Remarque : pour l'induction structurelle sur les arbres d'expressions, le cas de base est le cas de l'arbre réduit à une feuille.

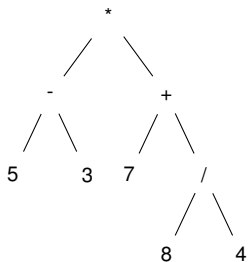
Arbres binaires d'expressions : primitives

Aux primitives sur les arbres binaires, on ajoute les primitives :

- `ABfeuille(x)` renvoie l'arbre réduit à une feuille d'étiquette x ,
- `estABfeuille(T)` teste si l'arbre T est réduit à une feuille.

Complexité : en $\Theta(1)$ pour chaque primitive.

Un exemple



Propriétés des arbres binaires d'expressions

Theorem

Dans un arbre binaire d'expression :

- *chaque nœud interne a exactement deux fils*
- *chaque nœud interne contient un opérateur (+, −, * ou /)*
- *chaque feuille contient une valeur numérique*
- *le nombre de valeurs numériques est égal au nombre d'opérateurs augmenté de 1.*

Preuve par induction structurelle.

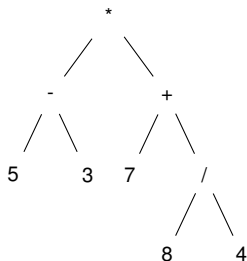
Arbres d'expressions : évaluation

Voici une définition **inductive** de l'évaluation d'un arbre d'expression.

```
def ABeval(T):  
    if estABfeuille(T):  
        return T.clef  
    if T.clef == "+":  
        return ABeval(T.gauche) + ABeval(T.droit)  
    if T.clef == "-":  
        return ABeval(T.gauche) - ABeval(T.droit)  
    if T.clef == "*":  
        return ABeval(T.gauche) * ABeval(T.droit)  
    return ABeval(T.gauche) / ABeval(T.droit)
```

Complexité de `ABeval(T)` : en $\Theta(n)$

Arbres d'expressions : parcours préfixe

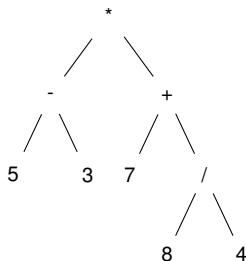


Parcours préfixe : $[* \ - \ 5 \ 3 \ + \ 7 \ / \ 8 \ 4]$

C'est une expression arithmétique préfixe (opérateur avant opérandes).

L'évaluation de l'arbre et celle de l'expression donnent le même résultat : 18.

Arbres d'expressions : parcours suffixe

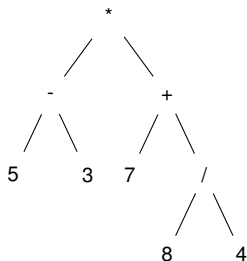


Parcours suffixe : [5 3 - 7 8 4 / + *]

C'est une expression arithmétique suffixe (opérateur après opérandes).

L'évaluation de l'arbre et celle de l'expression donnent le même résultat : 18.

Arbres d'expressions : parcours infixe



Parcours infixe : $[5 - 3 * 7 + 8 / 4]$

C'est une expression arithmétique infixe (opérateur entre opérandes).

Attention ! Sans parenthésage, l'évaluation de cette expression infixe est obtenue en appliquant les règles de priorité habituelles.
Le résultat est -14.

Expressions préfixes bien formées

Definition

Une liste L est une *expression préfixe bien formée* si elle est égale au parcours préfixe d'un arbre d'expression.

Theorem

Une liste L de longueur n est une *expression préfixe bien formée* ssi

- le nombre de valeurs numériques de L est égal au nombre d'opérateurs de L augmenté de 1
- pour tout $i < n - 1$, le nombre de valeurs numériques de $L[0..i]$ est inférieur ou égal au nombre d'opérateurs de $L[0..i]$.

Preuve en TD

Des expressions préfixes vers les arbres (1)

La fonction `operandesPref(L)` définie ci-dessous retourne le couple formé des deux opérandes de `L`, en supposant que `L` est une expression préfixe bien formée.

```
def operandesPref(L):  
    cpt_operateurs = 0  
    cpt_valeurs = 0  
    i = 1  
    while cpt_operateurs >= cpt_valeurs:  
        if estOperateur(L[i]):  
            cpt_operateurs = cpt_operateurs + 1  
        else:  
            cpt_valeurs = cpt_valeurs + 1  
            i = i + 1  
    return (L[1:i], L[i:])
```

La fonction `estOperateur(x)` teste si `x` est un opérateur.

```
>>> L0 = ["*", "-", 5, 3, "+", 7, "/", 8, 4]  
>>> operandesPref(L0)  
(['-', 5, 3], ['+', 7, '/', 8, 4])
```


Des expressions préfixes vers les arbres (2)

La fonction suivante associe un arbre à toute expression préfixe bien formée :

```
def prefVersAB(L) :  
    x = L[0]  
    if estNombre(x) :  
        return ABfeuille(x)  
    (L1, L2) = operandsPref(L)  
    return AB(x, prefVersAB(L1), prefVersAB(L2))
```

Theorem

Le parcours préfixe de l'arbre $\text{prefVersAB}(L)$ est égal à L .

Complexité de `prefVersAB`

- Représentation par tableaux

Complexité de `operandesPref(L)`

- Pire cas : $n_1 = n - 2$ d'où $O(n)$.
- Meilleur cas : $n_1 = 1$ d'où $\Omega(1)$.

Complexité de `prefVersAB(L)`

- Pire cas : $n_1 = n - 2$ à chaque appel récursif donc
 $c(n) = n + c(n_1) = n + c(n - 2)$ à chaque appel récursif d'où $O(n^2)$.
- Meilleur cas : $n_1 = 1$ à chaque appel récursif donc
 $c(n) = 1 + c(n_2) = 1 + c(n - 2)$ à chaque appel récursif d'où $\Omega(n)$.

- Représentation par listes chaînées

Complexité de `operandesPref(L)` en $\Theta(n)$.

Relation de récurrence : $c(n) = n + c(n_1) + c(n_2)$.

- Pire cas : $n_1 = n - 2$ à chaque appel récursif d'où $O(n^2)$.
- Meilleur cas : $n_1 \approx n/2$ à chaque appel récursif d'où $\Omega(n \log n)$.