

AlgoProg3

Laurent.Fuchs@univ-poitiers.fr

Chapitre 0

0.1 Présentation du cours

- Algorithmique sur les structures arborescentes
 - arbres binaires, arbres généraux, ABR, AVL, etc...
- Types abstraits de données
 - Comment spécifier des opérations sur un type sans ??
- Complexité des algorithmes
 - Comment évaluer l'efficacité d'un algorithme ?
- Implantation d'un type abstrait
 - Comment implanter en respectant la spécification

0.2 Organisation du cours

- 5 Cours, 10 TD et TP, intervenants
- CC :
 - 1 contrôle continue de 2h (coeff 2)
 - 1 projet (Coeff 2)
 - 1 examen de 3h (coeff 8)
 - ??? 2 de 3h

Table des Matières

Chapitre 1 : Les structures arborescentes	3
1.1 Arbres binaires	4
1.2 Type générique pour les arbres binaires	5
1.3 Mesures sur les arbres binaires	5
1.4 Cas particuliers	6
1.5 Propriétés	8
Propriétés des arbres binaires localement complets :	9
Dénombrement des arbres binaires	9
1.6 Cas particuliers Parcours d'un arbre binaire	10
Ordres induits par le parcours en profondeur :	10
2.1 Arbres généraux	11
2.2 Type générique pour les arbres généraux	13
2.3 Mesures sur les arbres généraux	14
2.4 Parcours d'un arbre général	14
Chapitre 2 : Les types abstraits de données	16
0. Introduction	16
1. Signature	17
2. Description des opérations sur une sorte	18
3. Réutilisation d'un type abstrait dans un type abstrait	19
4. Opérations définies partiellement et précondition.	19
4.1 Terme désignant une valeur d'un type	19
4.2 Opérations partiellement définies	20
4.3 Préconditions	20
5. Notion de constructeurs	21
6. Le type abstrait des arbres binaires	21
Chapitre 3 : Complexité des algorithmes	22
Mesure de la complexité en temps	22
Calcul de la complexité	22
Ordre de grandeur	25
Chap 4 : Implantation d'un type abstrait et dérécursification	27
Retour sur la notion de signature	27
Implantation d'un type abstrait	28
2.1 Implantation à l'aide d'un type somme	28
2.2 : Implantation à l'aide d'un type produit	29
2.3 : Implantation à l'aide des tableaux	32
Dérécursification	32
3.1 : Récursivité Simple	32
3.2 : Récursivité terminale	33
3.3 : Transformation en fonction récursive terminale	34
3.4 : Forme récursive terminal d'une boucle while	34
3.5 : Comment faire quand on ne sait pas mettre une fonction en forme récursive terminale ?	36
3.6 : Dérécursification d'une fonction récursive double	37
Chap 5 : Les arbres binaires de recherche	39
Définitions, Exemples	39
Recherche d'un élément dans un ABR	40
Ajout d'un élément	41
3.1 : Ajout aux feuilles	41
3.2 : Ajout à la racine	43

Chapitre 1 : Les structures arborescentes

Il s'agit de structures de données très importantes en informatique. Dès qu'une collection d'objets est organisée de façon hiérarchique, elle se représente par une structure arborescente.

- Compilation, structure syntaxique
- Base de données
- Systèmes de fichiers
- Algo de compression d'images
- Algo de synthèse d'images
- Dans les matériels réseau

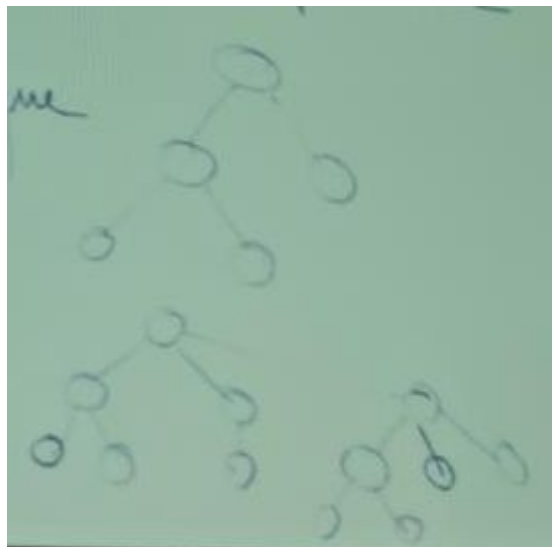


Schéma d'arbres

Une caractérisation des structures arborescentes est la récursivité. Ainsi bien les définitions que les algorithmes de manipulation s'écrivent naturellement de manière récursive.

1.1 Arbres binaires

Définition : Un arbre binaire est soit vide (noté symbole vide) soit il est de la forme $B : (v, B1, B2)$ où $B1$ et $B2$ sont des arbres binaires.

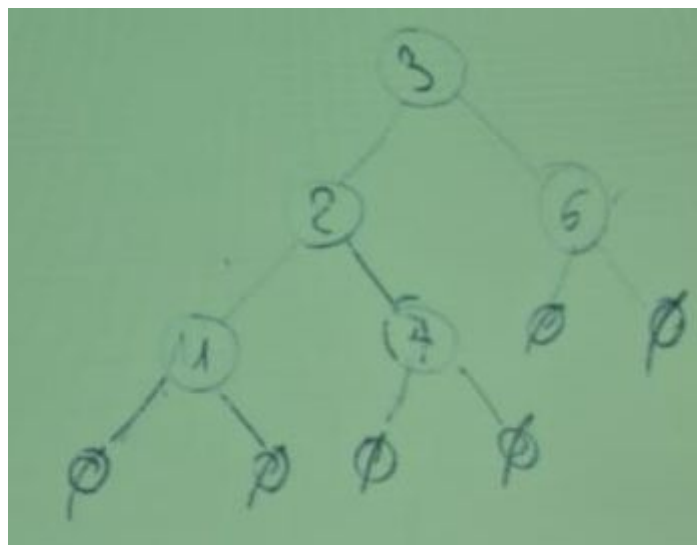
La structure $(v, B1, B2)$ est un noeud où v est la valeur contenue dans ce noeud.

La définition des arbres binaires est récursive. En termes d'ensembles, on peut l'écrire $B = \emptyset \cup (v, B, B)$ où B est l'ensemble des arbres binaires.

Vocabulaire : Soit un arbre binaire $B = (v, B1, B2)$

- $(v, B1, B2)$ est le noeud racine de l'arbre
- v est l'étiquette de la racine
- $B1$ est le fils gauche
- $B2$ est le fils droit
- Un noeud sans fils est une feuille (ou noeud externe)
- Un noeud avec au moins un fils est un noeud interne
- Une branche est une suite de noeuds consécutifs de la racine à une feuille

Exemple : $(3, (2, (11, \emptyset, \emptyset), (1, \emptyset, \emptyset)), (6, \emptyset, \emptyset))$



$$LC(B) = 22$$

$$P_{moy}(B) = 22/10 = 2,2$$

1.4 Cas particuliers

Un arbre est filiforme si chaque noeud n'a qu'un fils.

Un arbre binaire est complet s'il contient 1 noeud au niveau 0, 2 noeuds au niveau 1, 4 noeuds au niveau 2, ..., 2^l noeuds au niveau l .

La taille d'un arbre binaire complet **B** est $n = 1 + 2 + 4 + \dots + 2^{h(B)} = 2^{h(B)+1} - 1$



schéma filiforme



schéma binaire complexe

Un arbre binaire est parfait si tous ses niveaux sont remplis sauf éventuellement le dernier, auquel cas les feuilles sont le plus à gauche possible



schéma d'arbre binaire parfait

Un arbre binaire est localement complet si tous les noeuds qui ne sont pas des feuilles ont 2 fils. On peut donner une définition récursive des arbres binaires localement complets :

$$BC = (v, \emptyset, \emptyset) \cup (v, BC, BC)$$



Schéma arbres localement complet

Un arbre binaire est un peigne gauche s'il est localement complet et tout fils droit est une feuille.

On peut aussi donner une définition récursive des peignes gauches :

$$PG = (v, \emptyset, \emptyset) \cup (v, PG, (v, \emptyset, \emptyset))$$

Les peignes droits se définissent de façon semblable

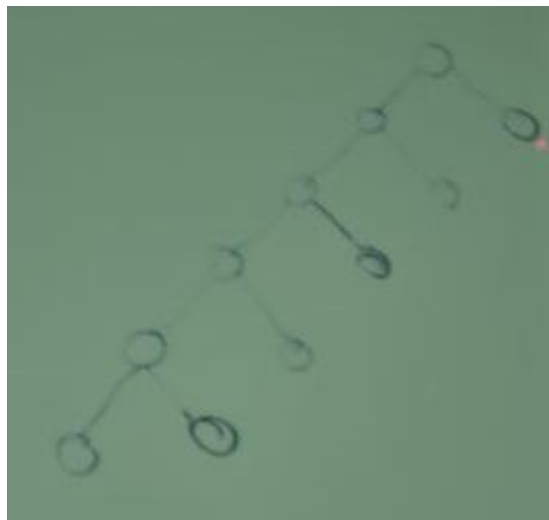


schéma peigne gauche

1.5 Propriétés

La hauteur et la longueur de cheminement sont des mesures importantes pour l'étude de la complexité des algorithmes qui manipulent des arbres. On voit dans cette section des propriétés sur ces mesures.

Encadrements de la hauteur et de la longueur de cheminement.

Propriété : Pour un arbre binaire de taille n et de hauteur h , on a :

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Où $\lfloor x \rfloor$ désigne la partie entière par défauts (pas un arrondi, mais une troncation), et $\log x$

le logarithme binaire $\log_2 x = \frac{\ln x}{\ln 2}$. On a $\log_2 x = a \Leftrightarrow x = 2^a$

Preuve :

Pour une hauteur h donnée, les arbres ayant le plus petit nombre de noeuds sont les arbres filiformes. Un arbre filiforme B de hauteur h est de taille $n = h + 1$. D'où $h \leq n - 1$.

D'autre part, l'arbre de hauteur h ayant le plus grand nombre de noeuds est l'arbre complet qui a pour taille $2^{(h+1)} - 1$.

Ainsi, pour un arbre binaire on a $n \leq 2^{h+1}$, c'est à dire $\log_2 n \leq h + 1$. D'où $\lfloor \log_2 n \rfloor \leq h$.
On a bien $\lfloor \log_2 n \rfloor \leq h \leq n - 1$ pour un arbre binaire.

Remarquez qu'un arbre binaire parfait de taille n a pour hauteur $\lfloor \log_2 n \rfloor$ car sa taille est entre 2^h noeuds et $2^{h+1} - 1$ noeuds

Propriété : Tout arbre binaire B non vide avec f feuilles est tel que $h(B) \geq \lceil \log_2 f \rceil$ où $\lceil x \rceil$ est la partie entière par excès (on arrondit au dessus, ex $2,1 \Rightarrow 3$).

Les preuves de cette propriété et de celles qui suivent seront traitées en exercices.

Propriété : Pour un arbre binaire B de taille n , on a :

$$\sum_{1 \leq k \leq n} \lfloor \log_2 k \rfloor \leq LC(B) \leq \frac{n(n-1)}{2}$$

Propriété : Un arbre binaire B avec f feuilles est tel que $PE(B) \geq \log_2 f$ où $PE(B)$ est la profondeur moyenne externe.

Propriétés des arbres binaires localement complets :

Propriété : Un arbre binaire localement complet avec n noeuds internes a $n + 1$ feuilles.

Preuve par récurrence sur le nombre de noeuds internes de l'arbre :

- La propriété est vraie pour l'arbre avec 0 noeuds internes et 1 feuilles.
- On suppose maintenant la propriété vraie pour les arbres avec moins de n noeuds. On considère un arbre $B = (v, B_1, B_2)$ avec n noeuds internes. On note n_1 le nombre de noeuds internes de B_1 et n_2 pour B_2 .
On a $n = n_1 + n_2 + 1$
Par l'hypothèse de récurrence B_1 a $n_1 + 1$ feuilles et B_2 a $n_2 + 1$ feuilles. Les feuilles de B sont des feuilles de B_1 ou des feuilles de B_2 .
Ainsi, le nombre de feuilles de B est $(n_1 + 1) + (n_2 + 1) = n + 1$. Ce que montre la propriété.

Propriété : Pour un arbre binaire localement complet B avec n noeuds internes on a :

$$LCE(B) = LCI(B) + 2n$$

Dénombrement des arbres binaires

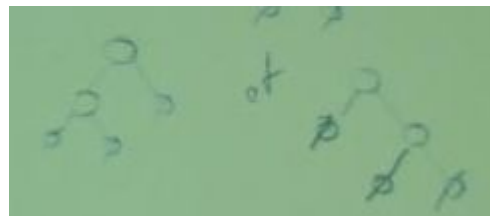
Propriété : Le nombre d'arbres binaires de taille n est $b_n = \frac{1}{n+1} \binom{2n}{n}$
où $\binom{n}{p} = C_p^n = \frac{n!}{p!(n-p)!}$ le Coefficient Binomial.

Exemples :

- Si $n = 0$, on a $b_0 = 1$ arbre, l'arbre vide

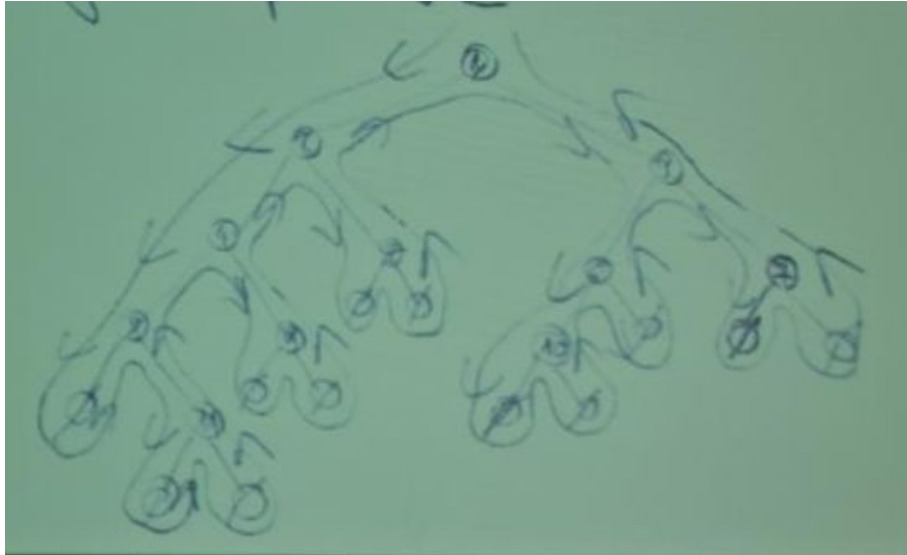
$$b_1 = \frac{1}{2} \binom{2}{1} = 1$$

- Si $n = 1$, on a
- Si $n = 2$, on a $b_2 = 2$
- Si $n = 3$, on a $b_3 = 5$



1.6 Cas particuliers Parcours d'un arbre binaire

Parcours en profondeur (à main gauche) : On part de la racine et on va toujours le plus à gauche possible



Algorithme récursif du parcours en profondeur (à main gauche)

```

let rec depth_path(a : 'a t_btree) : unit =
    if isEmpty(a)
    then ()
    else
        let r = root(a) in
        begin
            processing1(r);
            depth_path(lson(a));
            processing2(r);
            depth_path(rson(a));
            processing3(r);
        end
end
;;

```

Ordres induits par le parcours en profondeur :

→ **Ordre Préfixe** : les traitements 2 et 3 n'existent pas (les fonctions processing2 et processing3 ne font rien) l'ordre de parcours sur l'exemple est

1 2 4 8 11 9 5 3 6 10 7

→ **Ordre Infixe** : les traitements 1 et 3 n'existent pas. L'ordre du parcours sur l'exemple est

8 11 4 9 2 5 1 10 6 3 7

→ **Ordre Postfixe** : les traitements 1 et 2 n'existent pas. L'ordre du parcours sur l'exemple est :

11 8 9 4 5 2 10 6 4 3 1

2.1 Arbres généraux

Les arbres (planaires) généraux sont des arbres où le nombre de fils n'est pas limité à deux.

Définition : Un arbre est la donnée d'une racine et d'une liste finie éventuellement vide d'arbres $A = (v, A_1, \dots, A_p)$.

Une **forêt** est une liste finie éventuellement vide d'arbres.

Remarque : On obtient donc un arbre en ajoutant une racine à une forêt

Ces définitions s'expriment de façon mutuellement récursives en terme d'ensembles.

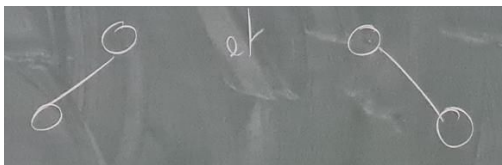
- \mathbb{A} désigne l'ensemble des arbres généraux
- \mathbb{F} désigne l'ensemble des forêts et on note \emptyset la forêt vide.

On a $\mathbb{A} = (v, \mathbb{F})$ et $\mathbb{F} = \emptyset \cup \mathbb{A} \cup (\mathbb{A}, \mathbb{A}) \cup (\mathbb{A}, \mathbb{A}, \mathbb{A}) \cup \dots$

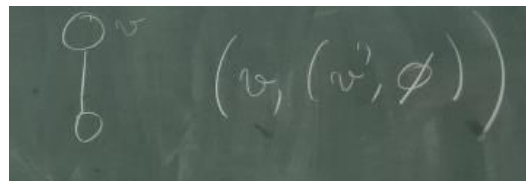
Exemple : Où l'on ne représente pas la forêt vide



L'arbre avec un seul noeud \circ est (v, vide) . Il n'y a pas de notion de gauche ou de droite. Contrairement aux arbres binaires, il n'y a qu'un seul arbre avec deux noeuds.

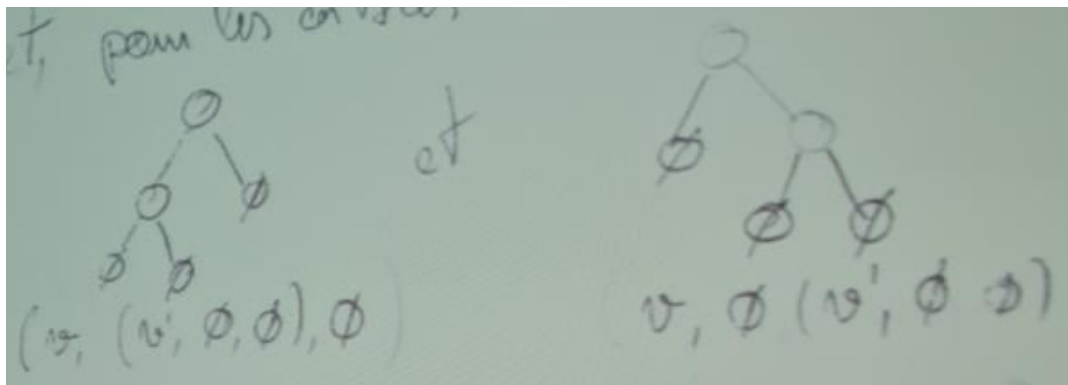


Arbres binaires avec 2 noeuds

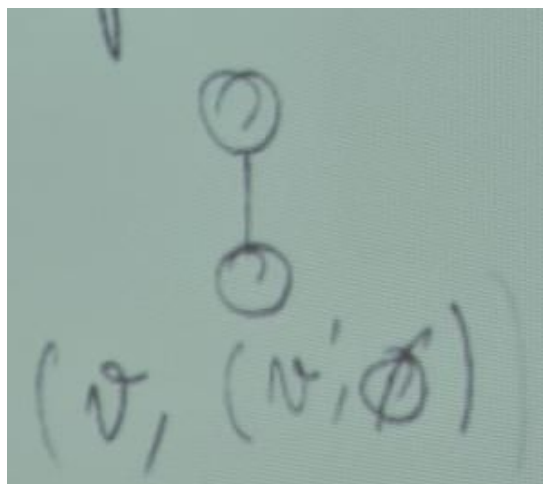


Arbre général avec deux noeuds

En effet, pour les arbres binaires on a :



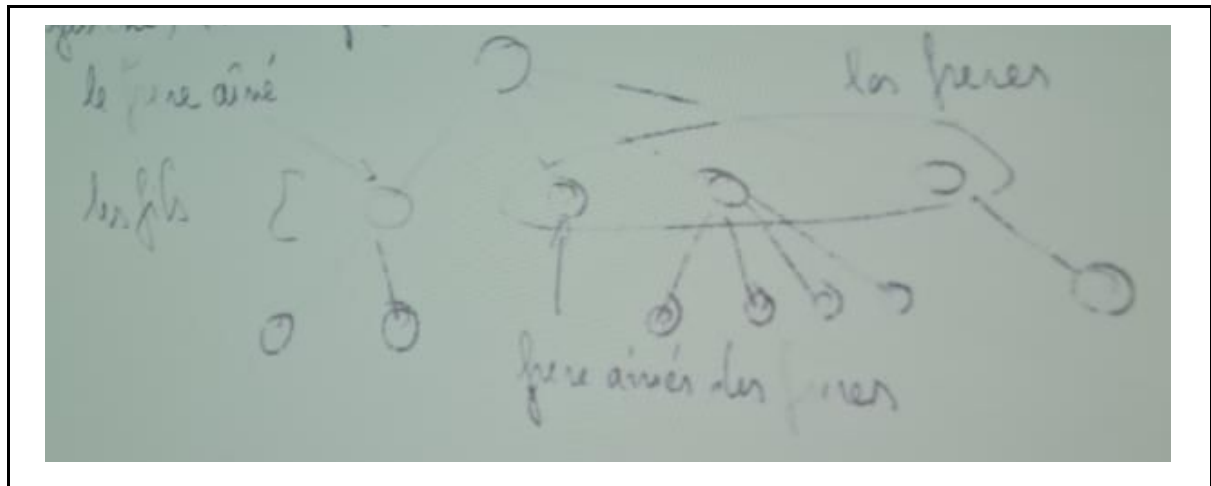
alors que pour les arbres généraux, on a seulement :



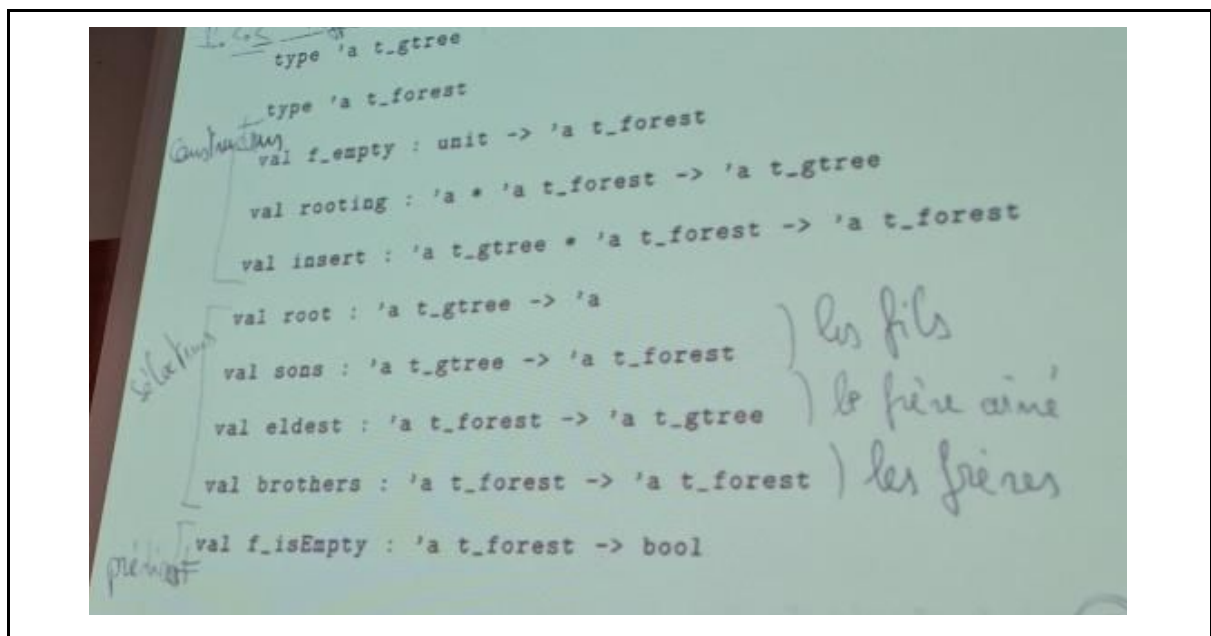
Remarquez aussi qu'un arbre n'est jamais vide !

→ l'arbre vide est une **forêt**

Comme il n'y a pas de notion de gauche et de droite pour les fils on parle du frère aîné (celui de gauche) et des frères.



2.2 Type générique pour les arbres généraux



2.3 Mesures sur les arbres généraux

On définit la taille et la hauteur d'un arbre général de façon similaire à la taille et la hauteur d'un arbre binaire.

La taille d'un arbre général est le nombre de ses noeuds

→ Pour calculer cette taille, on a besoin de la taille d'une forêt

On a $\text{tree_size}(a) = 1 + \text{forest_size}(\text{sons}(a))$

avec $\text{forest_size}(f_{\text{empty}}) = 0$

$\text{forest_size}(f) = \text{tree_size}(\text{eldest}(f)) + \text{forest_size}(\text{brothers}(f))$

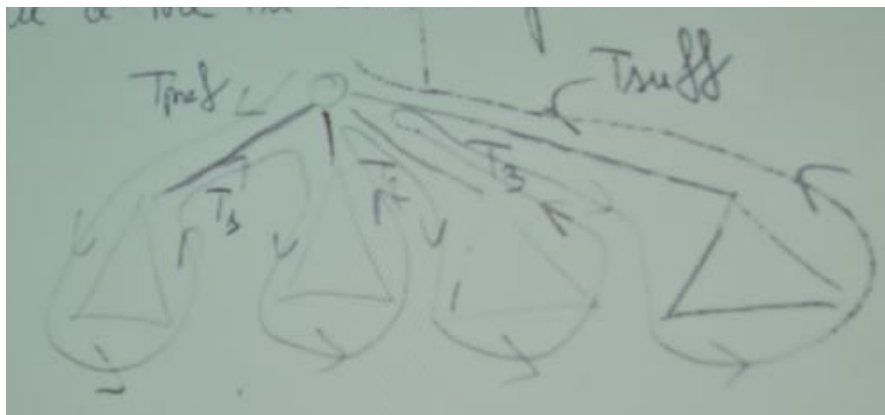
On remarque que ces fonctions sont mutuellement récurives.

La hauteur d'un arbre est un de plus que la hauteur maximum de ses fils.

Exercice : Écrire l'algorithme de calcul.

2.4 Parcours d'un arbre général

On effectue un parcours en profondeur pour chaque arbre de chaque forêt :



Pour chaque sous arbre Δ il ya un parcours en profondeur.

Les traitements se font entre chaque "remontée".

On arrête la descente en profondeur quand on arrive sur une feuille, c'est à dire sur une arbre dont la forêt est vide.

→ La fonction $\text{isLeaf}(a) = \text{f_isEmpty}(\text{sons}(a))$ teste si un arbre est une feuille.

```

(* Les fonctions tree_path et forest_path sont mutuellement récursives *)
let rec tree_path(a : 'a t_gtree) : unit =
  let r = root(a) in
  if isLeaf(a) then processleaf(r) (* Traitement d'une feuille *)
  else
    begin
      processpref(r); (* Traitement prefix *)
      forest_path(sons(a));
      processsuff(r); (* Traitement suffix *)
    end
and (* Le mot clé "and" permet de déclarer les deux fonctions en même temps *)
  forest_path(f : 'a t_forest) : unit =
    if f_isEmpty(f) then ()
    else
      begin
        tree_path(eldest(f));
        process(); (* Traitement entre deux fils *)
        forest_path(brothers(f));
      end
;;

```

Algorithme récursif de parcours en profondeur

Chapitre 2 : Les types abstraits de données

0. Introduction

Le concept de type abstrait est la justification théorique de la technique d'encapsulation qui consiste à cacher la représentation d'un type de données, c'est à dire sa structure de données, pour obliger à utiliser les fonctions de manipulation définies par le programmeur.

Les types abstraits sont la base théorique des notions de modules dans les langages des programmations de classes pour les langages de programmation orientée objets et d'API (Application Programming Interface) pour les bibliothèques logicielles.

De plus se baser sur le concept de type abstrait pour l'implantation d'une structure de données permet de délimiter les modifications lors d'un changements de structure de données et apporte la flexibilité nécessaire à la réutilisation d'une implantation d'une structure de données.

L'idée sous-jacente au concept de type abstrait est qu'il n'est pas nécessaire de donner la représentation interne d'un type (c'est à dire, la structure de donnée qui l'implante) pour définir le comportement des opérations de manipulation (c'est à dire leur spécification).

Exemple : Les booléens.

On sait que le type des booléens contient 2 valeurs : Vrai et Faux. On peut définir le comportement de la fonction de négation, `neg`, simplement en écrivant :

```
neg(Vrai) = Faux
neg(Faux) = Vrai
```

De même pour la fonction de conjonction, `et` (le "et" logique) en écrivant

```
et(Vrai, Vrai) = Vrai
et(Vrai, Faux) = Faux
et(Faux, Vrai) = Faux
et(Faux, Faux) = Faux
```

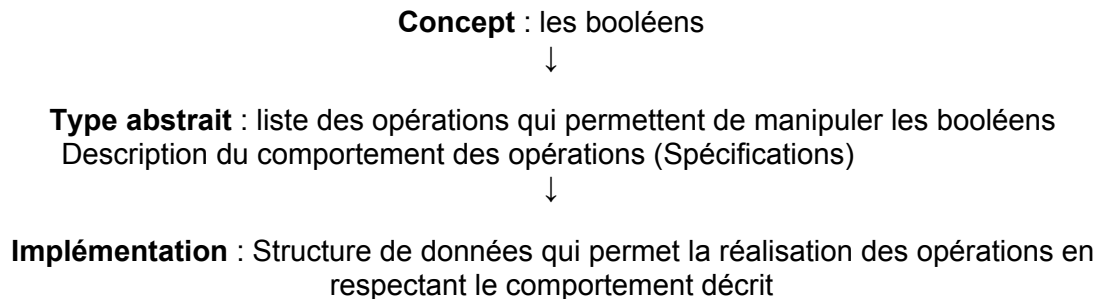
Telle qu'elle est écrite, la fonction `et` prend en argument un couple de booléens et renvoie un booléen. La notation `et(Faux, Vrai)` est la notation préfixée. On peut écrire la même chose en notation infixe `Faux et Vrai`. Enfin, dans beaucoup de langages la fonction `et` se note "&&" (le & se nomme "esperluette").

Remarquez : que peu importe la façon de noter la fonction "et", sa signification ne change pas.

Avec cet exemple, on constate que l'on a pas besoin de connaître l'implémentation des booléens dans un langage particulier pour donner la signification de leur fonctions de manipulation.

Dans la pratique, les implémentations des booléens en C et en OCaml sont très différentes, mais il s'agit toujours du même type (abstrait) des booléens.

On peut résumer la situation par le schéma suivant :



1. Signature

La signature d'un type abstrait donne la liste des noms des opérations et des noms des types de leurs arguments et valeurs de résultats.
Elle donne aussi le nom du type abstrait.

Définition : Le nom d'un type abstrait s'appelle une **sorte**

Dans le cadre des types abstraits, on parle de la sorte des booléens/entrées/tableaux...

Exemple : la signature des booléens

```
sorte Bool
opérations vrai  : → Bool
             faux : → Bool
             neg  : Bool → Bool
             et   : Bool * Bool → Bool
             ou   : Bool * Bool → Bool
```

Une opération qui ne prend pas d'arguments est une constante. Ici, les opérations "vrai" et "faux" sont des constantes.

La partie après les ":" est le type de l'opération. Par exemple , le type d'une opération contient une flèche qui sépare la sorte des arguments de la sorte des résultats.

La sorte `Bool * Bool` désigne les couples de booléens. l'opérateur "*" est un opérateur sur les sorte qui permet de construire des n-uplets (couples, triplets, etc...). Pour faciliter l'écriture du comportement des opérations (spécification) on peut préciser la place des arguments avec le caractère "souligné".

Par exemple, si l'on écrit `et : Bool * Bool → Bool` la syntaxe de l'opération "et" est "et(vrai, faux)". Si l'on écrit `_et_ : Bool * Bool → Bool`, la syntaxe de l'opération "et" est "vrai et faux".

Remarquez que l'on a défini que la syntaxe d'un type abstrait . si l'on rename la sorte Bool et les opérations, on peut écrire :

```

sorte A
opérations  x      :  $\rightarrow A$ 
               y      :  $\rightarrow A$ 
               z      :  $A \rightarrow A$ 
               u      :  $A * A \rightarrow A$ 
               v      :  $A * A \rightarrow A$ 

```

Qui est une signature équivalente à la précédente mais qui ne donne aucune indication sur sa signification.

2. Description des opérations sur une sorte

Il s'agit maintenant de donner une signification (une sémantique)aux opérations. Pour cela on utilise des égalités (plus généralement des formules logiques) que l'on appelle des axiomes . Par exemple les axiomes pour l'opération "neg" sont les égalités :

- $\text{neg}(\text{vrai}) = \text{faux}$
- $\text{neg}(\text{faux}) = \text{vrai}$

Ainsi la définition d'un type abstrait est la donnée d'une signature et d'un ensemble d'axiomes

```

sorte Bool
opérations  vrai   :  $\rightarrow \text{Bool}$ 
               faux   :  $\rightarrow \text{Bool}$ 
               neg    :  $\text{Bool} \rightarrow \text{Bool}$ 
               et     :  $\text{Bool} * \text{Bool} \rightarrow \text{Bool}$ 
               ou     :  $\text{Bool} * \text{Bool} \rightarrow \text{Bool}$ 

axiomes      neg(vrai) = faux
               neg(faux) = vrai
               et(vrai, vrai) = vrai
               et(vrai, faux) = faux
               et(faux, vrai) = faux
               et(faux, faux) = faux
               ou(a, b) = neg(et(neg(a), neg(b)))
                   avec  $a : \text{Bool}, b : \text{Bool}$ 

```

On introduit des variables de la sorte Bool pour écrire l'axiome qui définit l'opération "ou".

Exercice : Vérifier que l'opération "ou" est définie correctement par rapport à la table de vérité du "ou" logique.

3. Réutilisation d'un type abstrait dans un type abstrait

Il serait peu pratique de devoir réécrire la définition d'un type abstrait chaque fois que l'on veut l'utiliser dans un autre type abstrait.

Par exemple, pour la signature des tableaux on écrit :

```
sorte      Array
utilise    Nat, Élément
           Nat = Le type abstrait des entiers positifs
           Élément = Le type abstrait des éléments du tableau
opérations ième : Array * Nat → Élément
           change : Array * Nat * Élément → Array
           bornesup : Array → Nat
           borneinf : Array → Nat
```

L'introduction de la partie “**utilise**” indique que la signature de la sorte Array et l'union (disjointe) des signature des types abstrait Nat, Élément et Array.

Remarquez que ce mécanisme introduit la notion de surcharge d'une opération. En effet, comme la syntaxe d'une opération est définie par son nom et son type, l'opération :

$_+ _ : \text{Nat} * \text{Nat} \rightarrow \text{Nat}$ n'est pas la même que l'opération $_+ _ : \text{Int} * \text{Int} \rightarrow \text{Int}$ où Int est le type abstrait des entiers (positifs et négatifs).

Les sortes utilisées dans une syntaxe sont les sortes prédéfinies . Les nouvelles sortes introduites dans une signature sont les sorte définies. Dans l'exemple des tableaux , Nat et Élément sont des sortes prédéfinies et Array est la sorte définie.

On dit qu'une opération est interne si son résultat est d'un sorte définie.

Une opération est un observateur si au moins un de ses arguments est d'une sorte définie et son résultat est d'une sorte prédéfinie.

4. Opérations définies partiellement et précondition.

4.1 Terme désignant une valeur d'un type

Comme on ne connaît pas l'implantation d'un type abstrait, on ne peut pas manipuler directement ses valeurs. Celles-ci sont toujours désignées par un terme.

Ainsi la i ème valeur contenue dans un tableau ne peut être définie que par rapport à la fonction “change”. Pour un tableau $v : \text{Array}$, on sait que

```
ième (change(v, i, e), i) = e
si borneinf(v) ≤ i ≤ bornesup(v)
et que ième(change(v, i, e), j) = ième(v, j).
si borneinf(v) ≤ i ≤ bornesup(v)
et borneinf(v) ≤ j ≤ bornesup(v)
et i ≠ j.
```

Cela définit complètement le i ème élément d'un tableau à condition que celui-ci ait été initialisé.

4.2 Opérations partiellement définies

Cette dernière remarque indique que notre spécification n'est pas complète. Il manque l'opération qui permet de savoir si un élément d'un tableau a été initialisé. De plus, cela indique que l'opération "ième" n'est pas définie partout. C'est une opération partiellement définie.

4.3 Préconditions

Le domaine de définition d'une opération est l'ensemble des termes désignant une valeur pour lesquels l'opération est définie.

Pour définir le domaine de définition d'une opération, on utilise des préconditions qui sont des formules logiques indiquant les termes pour lesquels l'opération est définie.

Par exemple, la précondition pour l'opération "ième" est "ième(v,i) défini si et seulement si

borneinf(v) ≤ i ≤ bornesup(v)

et init(v, i) = vrai

où "init" est une opération (ou prédicat) qui indique si le ième élément d'un tableau a été initialisé.

On peut donner maintenant la spécification complète du type abstrait des tableaux.

sorte Array

utilise Nat, Bool, élément

Opérations

(opération qui définit l'intervalle des indices)(vect : Nat * Nat → Array

change : Array * Nat * Élément → Array

ième : Array * Nat → Élément

borneinf : 'Array → Nat

bornesup : 'Array → Nat

init : Array * Nat → Bool

avec: v : Array; i, j, k : Nat; e : Élément

préconditions

ième (v, i) défini si et seulement si

borneinf(v) ≤ i ≤ bornesup(v)

et init(v, i) = vrai

axiomes (ici un axiome conditionnel)

ième(change (v, i, e), j) =

si i ≠ j alors ième(v, i) sinon e

init(vect(i, j), k) = faux

init(change(v, i, e), j) =

si i ≠ j alors init(v, j)

sinon vrai

borneinf(vect(i, j)) = i

borneinf(change(v, i, e)) = borneinf(v)

```
bornesup(vect(i, j)) = j
bornesup(change(v, i, e)) = bornesup(v)
```

5. Notion de constructeurs

On peut remarquer que pour la spécification des tableaux toutes les opération (sauf “vect” et “change”) sont définies par rapport aux opération “vect” et “change”.

Les opérations “vect” et “change” sont des constructeurs. Les constructeurs définissent les termes qui désignent les valeurs d’un type abstrait.

Par exemple, le terme “vect(0,5)” désigne le tableau dont les indices sont entre 0 et 5 et pour lequel aucune valeur n’a été initialisée.

En effet, on peut vérifier que pour tout entier positif $k : \text{Nat}$ on a :

```
init(vect(0, 5), k) = faux
```

Le terme “change(vect(0,5), 2, ‘a’)” désigne le tableau pour lequel la case d’indice 2 a pour valeur le caractère ‘a’.

En utilisant les axiomes, on peut vérifier que la case d’indice 2 est bien initialisée.

```
init(change(vect(0, 5), 2, ‘a’), 2) = vrai
```

et que la valeur de la case d’indice 2 est le caractère ‘a’.

```
ième(change(vect(0, 5), 2, ‘a’), 2) = ‘a’
```

Note: init() correspond en fait à la fonction isnit().

6. Le type abstrait des arbres binaires

sorte t_btree

utilise Element, Bool

opérations

```
empty :→ t_btree
```

```
rooting: Element * t_btree * t_btree → t_btree
```

```
root: t_btree → Element
```

```
lson: t_btree → t_btree
```

```
rson: t_btree → t_btree
```

```
isEmpty: t_btree → Bool
```

avec $v : \text{Element}; a, l, r : \text{t_btree}$

préconditions (ssi = si et seulement si)

```
root(a) défini ssi isEmpty(a) = faux
```

```
lson(a) défini ssi isEmpty(a) = faux
```

```
rson(a) défini ssi isEmpty(a) = faux
```

axiomes

```
root(rooting(v, l, r)) = v
```

```
lson(rooting(v, l, r)) = l
```

```
rson(rooting(v, l, r)) = r
```

Ce qui correspond au type donné au chapitre 1.

<IMAGE SCANNES DE TURNUTE QUI A OUBLIÉ DE SCANNER LES FEUILLES>

Chapitre 3 : Complexité des algorithmes

L'objectif de l'analyse de la complexité d'un algorithme est de mesurer le temps d'exécution et la place mémoire utilisée afin de comparer différents algorithmes qui résolvent le même problème. Il est nécessaire de bien préciser ce que veut dire "mesurer le temps d'exécution" et "mesurer la place mémoire".

On cherche à mesurer la complexité d'un algorithme de façon indépendante de son implantation (càd de la machine utilisée, du langage de programmation, et du compilateur, etc ...). On ne peut donc pas se baser sur des quantités comme le nombre de cycles processeur pour réaliser une opération ou bien sur le nombre de mots mémoire nécessaire pour stocker une donnée.

1. Mesure de la complexité en temps

Pour mesurer la complexité en temps de l'algorithme, on va déterminer une ou plusieurs opérations fondamentales de l'algorithme. Le temps d'exécution de l'algorithme est proportionnel au nombre de ces opérations fondamentales.

Exemples :

- Recherche d'un élément dans une liste :
Comparaison entre l'élément et un élément de la liste.
- Trier les éléments d'une liste :
Comparaison entre deux éléments et déplacement d'un élément.
- Multiplier deux matrices :
Addition de deux nombres et multiplication de deux nombres.

Remarques :

- Ce nombre d'opérations fondamentales influence la précision de l'analyse. Par exemple, pour la multiplication de deux matrices on peut décider que l'addition et la multiplication de deux nombres ont le même temps d'exécution (ou pas).
- Lorsqu'on a choisi une opération fondamentale pour des algorithmes qui résolvent un problème. Il se peut que l'on invente un algorithme qui résout le même problème sans utiliser cette opération fondamentale. Il faut alors classer les algorithmes en différentes catégories et étudier chaque catégorie.

2. Calcul de la complexité

Il s'agit maintenant de compter le nombre d'opérations fondamentales réalisées par l'algorithme. Il n'y a pas de règles générales pour cela, néanmoins voici ce qui est communément admis :

- Lorsque les opérations sont dans une suite d'instructions, on ajoute les nombres d'opérations de chaque instruction.

- Pour les conditionnelles, on majore le nombre d'opérateurs de la façon suivante : $\text{nbOp}(\text{If } C \text{ Then } I_1 \text{ Else } I_2) \leq \text{nbOp}(C) + \max(\text{nbOp}(I_1), \text{nbOp}(I_2))$, où $\text{nbOp}()$ compte le nombre d'opérations.
- Pour les boucles, on se ramène au cas de la suite d'instruction en ajoutant autant de fois qu'il faut le nombre d'opérations. Cela peut être difficile à déterminer dans le cas d'une boucle while. Dans ce cas, on donne une majoration du nombre d'opérations.
- Pour les appels de fonction, on calcule le nombre d'opérations fondamentales réalisées par chaque fonctions. Dans le cas d'une fonction récursive, compter le nombre d'opérations demande de résoudre une équation de récurrence.

Exemple :

```
let rec fact(n : int) : int =
  if n <= 0
  then 1
  else n * (fact(n - 1))
;;
```

On choisit la multiplication de deux entier comme opération fondamentale.

On pose $M(n)$ le nombre d'opérations fondamentales réalisées par l'algorithme fact.

On a $M(0) = 0$ et $M(n) = M(n-1) + 1$ pour tout $n \geq 1$

(on ne considère pas les entiers $n < 0$).

On en déduit que $M(n) = n$.

Ainsi la complexité de l'algorithme fact est proportionnel à n . On dit aussi que son coût est proportionnel à n .

Dans l'exemple de l'algorithme fact, on a décidé que le coût de l'opération fondamentale (la multiplication de deux entiers) est égal à 1. On peut aussi attribuer un coût égal à une constante b à l'opération fondamentale.

Par exemple:

```
let rec print_list(l : int list) : unit =
  if l = []
  then ()
  else
    begin
      print_int(hd(l));
      print_list(tl(l));
    end
;;
```

Où hd donne le premier élément d'une liste et tl donne la liste dans son premier élément.

On choisit print_int comme opération fondamentale et on décide que son coût est la constante b .

On pose $P(n)$, où n est la longueur de la liste, comme étant le nombre d'opérations fondamentales. On a l'équation de récurrence $P(n) = P(n + 1) + b$. Cette équation se résout en $P(n) = P(0) + nb = nb$. Ainsi la complexité de l'algorithme `print_list` est proportionnel à $n * b$.

Pour les deux exemples, on peut remarquer que la complexité dépend de la taille des données. (grandeur de l'entier pour `fact`, longueur de la liste pour `print_init`).

D'une façon générale, la complexité en temps d'un algorithme dépend de la taille des données traitées par l'algorithme.

Considérons l'exemple suivant :

```
let rec seek_val(v, l : int * int list) : bool =
  if l = []
  then false
  else
    if v = hd(l)
    then true
    else seek_val(v, tl(l))
;;
```

Pour calculer sa complexité, on choisit comme opération fondamentale l'appel à la fonction `seek_val`.

Si la liste `l` est vide, ou si `v` est égal à la première valeur de la liste, il n'y a que l'appel initial à la fonction `seek_val`.

Si l'entier `v` ne figure pas dans la liste ou s'il est égal à la dernière valeur de la liste alors il y a n appels à la fonction `seek_val` où n est la longueur de la liste. Dans les autres cas, il y a j appels à la fonction `seek_val` où j est la position de la première valeur de la liste égale à l'entier `v`.

Aussi, dans le cas de la fonction `seek_val`, on constate que pour une même taille n de données, la complexité dépend des différentes données possibles. Il est donc utile d'introduire les notions de complexité dans le meilleur des cas, dans le pire des cas et en moyenne.

On note $C_A(d)$ la complexité de l'exécution de l'algorithme `A` sur la donnée `d` et on note $|d|$ la taille de la donnée `d`.

Définitions :

La complexité au meilleur des cas pour l'algorithme `A` sur les données de taille n est

$$CM_a(n) = \min_{\{d \text{ tel que } |d|=n\}} (C_a(d))$$

La complexité au pire des cas pour l'algorithme `A` sur les données de taille n est

$$CP_a(n) = \max_{\{d \text{ tel que } |d|=n\}} (C_a(d))$$

La complexité en moyenne pour l'algorithme `A` sur les données de taille n est

$$\gamma_a(n) = \sum_{\{d \text{ tel que } |d|=n\}} (p(d).C_a(d))$$

où $p(d)$ est la probabilité que l'on ait la donnée `d` en entrée de l'algorithme `A`.

Si les données ont toutes la même probabilités d'être en entrée de l'algorithme A alors on a)

$$\frac{1}{\{d \text{ tel que } |d| = n\}} \sum_{\{d \text{ tel que } |d|=n\}} C_a(d)$$

Par exemple on verra en TD que pour seek_val $\check{\theta}(n) = (n+1)/2$ si on sait que v est dans la liste l.

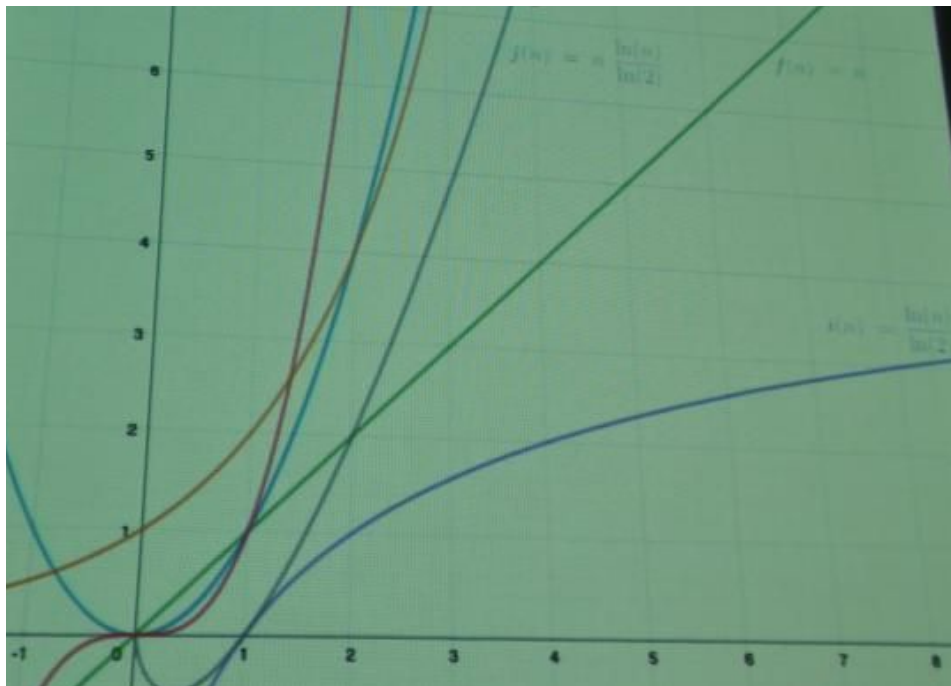
3. Ordre de grandeur

On a vu que la complexité d'un algorithme est une fonction de la taille des données traitées par cet algorithme.

Il est donc important d'avoir une estimation de la croissance de cette fonction lorsque la taille des données croît.

Cette estimation est l'ordre de grandeur asymptotique de la fonction.

Cet ordre de grandeur asymptotique est généralement comparé par rapport à l'ordre de grandeur asymptotique de fonctions usuelles 1, $\log_2 n$, $n \log_2 n$, n , n^2 , n^3 , 2^n , qui prennent une échelle de comparaison.



En vert : ordre de grandeur linéaire ($f(n) = n$).

En bleu foncé : $\log_2(n)$. Très en dessous de courbe linéaire

*En gris : $n * \log_2(n)$. Commence très petit, puis croît bien plus vite que vert et bleu*

En orange : 2^n . Croît encore plus vite

En bleu clair et violet : On croît encore encore plus vite

Pour comparer les ordres de grandeur asymptotiques, on utilise les notations de Landau. Soient f et g deux fonctions définies sur \mathbb{N} .

On note $f \in O(g)$ s'il existe $k > 0$, $n_0 \in \mathbb{N}$, tels que $|f(n)| \leq k * |g(n)|$ pour $n > n_0$

$f \in O(g)$ veut dire que l'ordre de grandeur asymptotique de f est plus petit que celui de g .
On dit aussi que f est dominée (asymptotiquement) par g .

Par exemple :

$2n \in O(n)$ mais $2n \notin O(1)$.
 $2n \in O(n^2)$

Ainsi, la complexité en temps de la fonction appartient à $O(n)$, de même pour la complexité en temps de `print_list`. L'ordre de grandeur asymptotique ne change pas si l'on fait n ou $n + 1$ opérations

La notation $f \in O(g)$ ne donne qu'un majorant de l'ordre de grandeur asymptotique de f . Cela n'est pas suffisant pour comparer les performances de différents algorithmes.

On introduit la notation Θ (grand Theta)

$f \in \Theta$ si $f \in O(g)$ et $g \in O(f)$

C'est à dire il existe $k_1 > 0$, et $k_2 > 0$ n_0 appartient \mathbb{N} tels que :

$k_1 |g(n)| \leq |f(n)| \leq k_2 |g(n)|$ pour $n > n_0$

On dit que f et g sont de même ordre de grandeur asymptotique.

Par exemple $2n \in \Theta(n)$, mais $2n \notin \Theta(n^2)$

L'ordre de grandeur asymptotique de la complexité d'un algo est important en pratique pour déterminer si un algo est utilisable ou non dans une situation donnée.

Le tableau A donne une estimation des temps d'exécution pour des algorithmes de grandeur différentes en fonction de la taille des données. Cela pour un ordinateur qui effectue 10^6 opérations par secondes

<tableau A de la feuille a scanner hein nunut ?>

Le tableau B donne une estimation de la taille maximale des données que l'on peut traiter pour un temps d'exécution fixée toujours pour un ordinateur qui effectue 10^6 opérations par seconde.

On note ∞ lorsque la valeur dépasse 10^{100}

Avec ces deux tableaux on peut déterminer facilement quels sont les algorithmes utilisables pour traiter des données de grandes tailles.

<tableau B de la feuille a scanner hein nunut ?>

Le tableau C montre comment la taille des données et le temps d'exécution varient en fonction l'un de l'autre. On voit que si l'on multiplie par 10 la vitesse de calcul de l'ordinateur, on ne modifie quasiment pas la taille maximale des données que l'on peut traiter avec un algorithme avec un ordre de grandeur asymptotique exponentiel.

Il est donc illusoire de compter sur les progrès technologiques lorsque l'on a pas un algorithme efficace !

<tableau C de la feuille a scanner hein nunut ?>

Chap 4 : Implantation d'un type abstrait et dérécursification

Avec la spécification de son comportement, un type abstrait ne donne pas d'indication sur la façon de le programmer dans un langage de programmation particulier.

Par ailleurs, les langages de programmation n'offrent pas tous les mêmes moyens syntaxiques pour implanter un type abstrait. Nous allons voir comment réaliser cette implémentation, et surtout comment vérifier que l'on a bien implémenté un type dont le comportement correspond bien au type abstrait qui a été spécifié.

1. Retour sur la notion de signature

La signature d'un type abstrait donne la liste des noms des opérations, des noms des type de leurs arguments et valeurs de résultats. Les langages de programmation traduisent par différents moyens la notion de signature.

Par exemple, en C, un fichier ".h" contient toutes les infos sur la signature d'un type abstrait. Si on lui associe un fichier ".c" contenant l'implémentation de ce type abstrait, on obtient une unité de compilation.

Cette unité de compilation peut être utilisée dans un programme via l'inclusion du fichier ".h" au moment de l'écriture du code et l'utilisation du fichier ".o" lors de la compilation (plus précisément au moment de la phrase d'édition des liens).

Concrètement, en C pour les arbres binaires le fichier btree.h s'écrit :

```
struct node;
typedef struct node * t_btree;
//On garde le type abstrait en cachant la structure cell. Un arbre est
un pointeur sur une structure cell
t_btree empty();
t_btree rooting(int v, t_btree ls, t_btree rs);
int root (t_btree t);
t_btree lson(t_btree t);
t_btree rson(t_btree t)
bool isEmpty(t_btree t);
//le type abstrait des booléens est définie dans une autre unités de
compilations.
```

En Java c'est la notion d'interface qui permet de traduire une signature d'un type abstrait.

Remarquez qu'une conséquence de ceci est que l'on peut avoir facilement différents implantations d'un même type abstrait. Une pour chaque classe qui implante l'interface qui correspond à ses signature.

On peut obtenir la même chose en C , il suffit d'inclure le fichier ".h" (par exemple btree.h) dans chaque fichier ".c" qui implante le type abstrait.

Il existe des mécanismes similaires en C++, en ADA, etc...

Généralement, tous les langages de programmation où une notion de module ou de classe existe permettent de traduire la signature d'un type abstrait.

En Ocaml, il existe deux façons de traduire une signature d'un type abstrait, soit par une interface d'une unités de compilation, soit par une signature d'une module.

En Ocaml, une unité de compilation est constituée de deux fichiers : un fichier ".mli" (équivalent à un ".h" en C), et un fichier ".ml" (équivalent à un ".c" en C).

La compilation de ces fichiers donnent respectivement un fichier ".cmi" et un fichier ".cmo" (vous en utiliserez en TP).

Par exemple, le fichier btree.mli contient:

```
type +'a t_btree
val empty : unit → 'a t_btree
val rooting : 'a * 'a t_btree * 'a t_btree → 'a t_btree
val root : 'a t_btree → 'a
val lson : 'a t_btree → 'a t_btree
val rson : 'a t_btree → 'a t_btree
val isEmpty : 'a t_btree → bool
```

2. Implantation d'un type abstrait

Il s'agit maintenant de donner une structure de données concrète pour un type abstrait. Dans cette partie, on prendra l'exemple des arbres binaires.

Dans les langages de programmations, on a généralement accès à trois outils pour définir une structure de données :

- les types sommes (énumérations, enum et union en C et C++)
- les types produits (n-uplet, enregistrement , struct en C, Class en Java et C ++)
- les tableaux (présent dans tous les langages de programmations impératifs)

2.1 Implantation à l'aide d'un type somme

Un type somme permet de mettre au même emplacement de la mémoire des données de différents types. Par exemple, on peut dire qu'à un emplacement mémoire il y a soit un entier, soit une chaîne de caractères. Pour savoir quelle est la donnée construite dans un type somme on a besoin d'un mécanisme de filtrage. Les types sommes existent dans quasiment tous les langages de programmations.

En fonction des moyens syntaxiques du langage et de leur compréhension par les programmeurs, ils sont plus ou moins utilisés selon les langages.

Les types somme sont très utilisés et faciles à utiliser dans les langages de programmation fonctionnelle comme Ocaml. Par exemple, le type t_btree s'écrit en Ocaml de la façon suivante :

```
type 'a t_btree = B_empty
                | B_node of 'a * 'a t_btree * 'a t_btree
(*La barre à la verticale se lit "ou"*)
```

Ainsi une valeur de type `t_btree` est soit la valeur `B_empty` qui est définie comme une constante arbitraire, soit un triplet composée d'une valeur de type `'a` et de deux valeurs de type `t_btree`.

On reconnaît que `B_empty` et `B_node` correspondent aux constructeurs `empty` et `rooting` du type abstrait des arbres binaires `b_tree`.

Ces deux constructeurs s'implémentent de la façon suivante :

```
let empty((): unit) : 'a t_btree = B_empty
let rooting(v, ls, rs : 'a * 'a t_btree * 'a t_btree) : 'a t_btree =
  B_node(v, ls, rs)
```

Les sélecteurs utilisent le mécanisme de filtrage pour déterminer quelle est la valeur contenue dans le type somme.

Par exemple, le sélecteur `root` s'écrit :

```
let root (t : 'a t_btree) : 'a =
  match t with
  | B_empty -> invalid_arg "root : Binary tree is empty"
  | B_node(v, ls, rs) -> v
```

La construction syntaxique “`match t with`” permet de filtrer le contenu de la valeur `t`.

Soit `t` contient `B_empty` et on lève une exception.

Soit `t` contient un triplet et on renvoie la première composante qui est la valeur à la racine de l'arbre.

2.2 : Implantation à l'aide d'un type produit

Un type produit est la traduction dans les langages de programmation de la notion mathématique de produit cartésien d'ensembles qui permet de construire des `n`-uplets. En C, un type produit est une structure.

Par exemple, le produit cartésien $N * N$ définit l'ensemble des couples d'entiers $(3, 7)$; $(9, 2)$, etc...

En C, cela se traduit par une structure :

```
struct couple
{
    int a;
    int b;
};

//Remarquez que les composantes du couple partant des valeurs
// ici a et b
```

L'implantation de types abstrait récursif comme les listes ou les arbres binaire avec un type produit conduit à l'utilisation des pointeurs pour chaîner les différentes valeurs du type produit.

Ainsi, en C, l'implantation du type `t_btree` s'écrit :

```
// Dans le ".h" on cache l'implantation de la structure node pour garder
l'obstruction du type
struct node;

typedef struct * tree;

// Dans le ".c" on donne l'implantation de la structure node.
// Le type tree est un pointeur sur une valeur du type produit qu'est la
// structure node. On choisit ainsi les sous arbres.
struct node
{
    int v;
    tree ls;
    tree rs;
};
```

Les constructeurs `empty` et `rooting` se basent sur le fait que le type `b_tree` est un porteur sur la structure `node`. Ils s'implantent de la façon suivante :

```
t_btree empty()
{
    return NULL;
}

t_btree rooting(int v, t_btree ls, t_btree rs)
{
    t_btree result;

    result = (t_btree)malloc(sizeof(struct node));
    //il faut allouer de la mémoire pour construire une nouvelle valeur
    //du type produit node
    result -> v = v;
    result -> ls = ls;
    result -> rs = rs;

    return result;
}
```

Les sélecteurs doivent tester si le pointeur de type `t_btree` donné en argument pointe ou non sur une structure `node`.

Par exemple, le sélecteur `root` s'écrit :

```

int root(t_btree t)
{
    if(isEmpty())
        exit(EXIT_FAILURE);
    else
        return (t -> v);
}
// il n'y a pas de mécanisme d'exception en C , le programme s'arrête si
// la précondition n'est pas respectée

```

Remarquez que votre implémentation des arbres binaire en C n'est pas polymorphe . On pouvait simuler cela en utilisant des macros ou des types sommes.

Un point important lorsqu'on utilise des pointeurs est la gestion de la mémoire allouée. Il est indispensable de la libérer correctement. Certains langages comme Java ou OCaml possèdent un mécanisme automatique de la mémoire allouée (garbage collector). Ce n'est pas le cas du C (ou C++ et d'autres). Il faut alors obligatoirement implanter une fonction de libération de la mémoire par le type abstrait implanté.

Pour notre implantation en C, on peut écrire une fonction récursive qui libère la mémoire en parcourant l'arbre à détruire.

Cette fonction s'implante de la façon suivante :

```

void free_btree(t_btree t)
{
    if(!isEmpty(t))
    {
        // On récupère les fils gauche et droit
        tree ls = lson(t);
        tree rs = rson(t);
        // On libère la mémoire qui correspond à la structure node
        // pointée par t
        free(t);
        // On appelle récursivement la fonction de libération de la
        // mémoire sur les fils gauche et droit
        free_btree(ls);
        free_btree(rs);
    }
}

```

2.3 : Implantation à l'aide des tableaux

Certains langages ne disposent ni de pointeurs, ni de types sommes, un moyen d'implanter un type abstrait comme les arbres binaires est alors de simuler les pointeurs en utilisant les indices des cases d'un tableau.

Dans certains cas, comme pour les arbres binaires complets, on obtient des implantations efficaces.

(Vous avez vu des exemples d'implantation des avec des tableaux en L2)

3. Dérécursification

Quand on spécifie un type abstrait ou une structure de données récursives comme les listes ou les arbres (binaires ou généraux) on écrit naturellement des fonctions récursives. Cependant, lors de l'implantation du type abstrait, il peut arriver que l'on ne souhaite implanter ces fonction de façon récursive. Car, par exemple, certains langages ne permettent pas la récursivité ou bien ils les gèrent mal. Il faut alors définir une version dérécursifier des fonctions.

3.1 : Récursivité Simple

Une fonction est simplement récursive si un appel de cette fonction n'engendre qu'un seul appel récursif

Exemples :

```
let rec fact(n : int) : int =  
  if n = 0  
  then 1  
  else n * fact(n - 1)  
;;  
  
let rec long(l : 'a list) : int =  
  if l = []  
  then 0  
  else 1 + long(List.tl(l))  
;;
```

Le cas général s'écrit :

```
let f(x : 'a) : 'b =  
  if c(x)  
  then g(x)  
  else h(f(s(x)), x)  
;;
```

où $c : 'a \rightarrow \text{bool}$ est la condition d'arrêt.

$g : 'a \rightarrow 'b$ permet le calcul de la valeur initiale

$h : 'a * 'a \rightarrow 'b$ effectue une étape du calcul

$s : 'a \rightarrow 'a$ calcule la valeur suivante à considérer dans le calcul de f

Pour fact on a :

$C(x) = (x = 0)$

$g(x) = 1$

$h(x, y) = x * y$

$s(x) = x - 1$

(Exercice : Donnez les fonction c,g,h,s pour long)

Si l'on suit les appels récursifs lors d'un calcul, on constate que ce n'est qu'une fois que tous les appels récursifs sont effectuées que l'on peut commencer le calcul.

Par exemple pour fact(4) on a :

$fact(4) \rightarrow fact(3) \rightarrow fact(2) \rightarrow fact(1) \rightarrow fact(0) \downarrow$
24 $\xleftarrow{\times 4}$ 6 $\xleftarrow{\times 3}$ 2 $\xleftarrow{\times 2}$ 1 $\xleftarrow{\times 1}$ 1 \leftarrow

Quand on regarde le cas général, cela vient du fait que l'appel récursif se situe dans les arguments de la fonction h.

Que se passe-t-il lorsqu'il n'y a pas de fonction h ?

3.2 : Récursivité terminale

Observons un cas où il n'y a pas de fonction h. Par exemple :

```
let rec pgcd(a, b : int * int) : int =  
  if b = 0  
  then a  
  else pgcd(b, a mod b)  
;;
```

$pgcd(18, 48) \rightarrow pgcd(48, 18) \rightarrow pgcd(18, 12) \rightarrow pgcd(12, 6) \rightarrow pgcd(6, 0) \downarrow$
6 \leftarrow 6 \leftarrow 6 \leftarrow 6 \leftarrow 6 \leftarrow

On a le résultat il n'y a plus de calcul à faire il suffit de dépiler les appels récursifs.

Dans le cas où il n'y a pas de fonction h, comme pour la fonction pgcd, on dit que l'on a une récursivité terminale. Une fonction est récursive terminale lorsque l'appel récursif est le dernier appel de fonction effectué dans ce cas, on obtient le résultat du calcul au dernier appel récursif.

La forme générale d'une fonction récursive terminale s'écrit :

```
let rec f(x : 'a) : 'b =  
  if C(x)  
  then g(x)  
  else f(s(x))  
;;
```

3.3 : Transformation en fonction récursive terminale

De nombreuses fonctions simplement récursives peuvent se transformer en une fonction récursive terminale en ajoutant dans leurs arguments un argument qui stocke le résultat du calcul.

Par exemple :

```
let rec factrt(n, v : int * int) : int =  
  if n = 0  
  then v  
  //lorsque l'on arrête les appels récursifs le résultat est dans v  
  else fact(n - 1, n * v)  
  //l'argument supplémentaire stocke le résultat de la multiplication  
  //à chaque étape  
;;
```

On fait de même pour la fonction long :

```
let rec longrt(l, v : 'a list * int) : int =  
  if l = []  
  then v  
  else longrt(List.tl(l), 1 + v)  
;;
```

L'intérêt des fonction récursive terminales est qu'elles peuvent s'écrire avec une boucle et l'on supprime ainsi les appel récursifs.

Pour bien comprendre cela, on va écrire une boucle "tant que" sous la forme d'une fonction récursive terminale.

3.4 : Forme récursive terminal d'une boucle while

Les éléments constitutifs d'une boucle while sont :

- Une condition de continuation (la négation de la condition d'arrêt)
- Un corps de boucle (qui sont les calculs effectués sur les arguments lors de l'appel récursif)
- Une valeur de résultat (qui est la valeur obtenue au dernier appel récursif)

Avec ces trois éléments, on peut écrire une fonction récursive terminale qui implante une boucle while de la façon suivante :

```
let rec whileloop(r, cont, next : 'a * ('a -> bool) * ('a -> 'a)) : 'a =  
  if not(cont(r))  
  then r  
  else whileloop(next(r), cont, next)  
;;
```

On peut maintenant écrire une version itérative des fonctions récursives terminales. Par exemple, pour la fonction `longrt` on obtient :

```
let longrt(l : 'a list) : int =
  let (k, res) : 'a list * int = whileloop(
    (l, 0),
    (function(l1, v) -> not(l1 = [])),
    (function(l1, v) -> (List.tl(l1), v + 1))
  ) in
  res
;;
```

On reconnaît ici les arguments de l'appel récursif

En utilisant les moyens syntaxiques du langage, cela se transforme en :

```
let longiter(l : 'a list) : int =
  let ll : 'a list ref = ref l
  and v : int ref = ref 0 in
  while not(!ll = []) do
    ll := List.tl(!ll);
    v := !v + 1;
  done;
  !v
;;
```

3.5 : Comment faire quand on ne sait pas mettre une fonction en forme récursive terminale ?

Quand on ne trouve de forme récursive terminale pour une fonction ou bien quand il on n'existe pas, on utilise une pile et deux boucles pour simuler les appels récursifs. Pour notre exemple du calcul de la longueur d'une liste et avec les moyens syntaxique du langage, cela donne (on suppose que vous disposez d'un module `ap3stack` qui implante les piles pour le cours d'AP3)

```
let longiterpllg(l : 'a list): int =
  let p : int stack ref = ref empty()
  and ll : 'a list ref = ref let
  and v : int ref = ref 0
  in
  (*On empile les données à utiliser tant que l'on n'a pas fini les appels
  récursifs*)
  while not (!ll=[])
  do
    p:= push(1, !p);
    ll:= List.tl(!ll);
  done;
  (*Quand les appels récursifs sont tous effectués, on dépile les données
  à utiliser et on effectue les calculs*)
  while not (isEmpty(!p)) do
    v := !v + top(!p);
    p := pop(!p);
  done;
  !v
;;
```

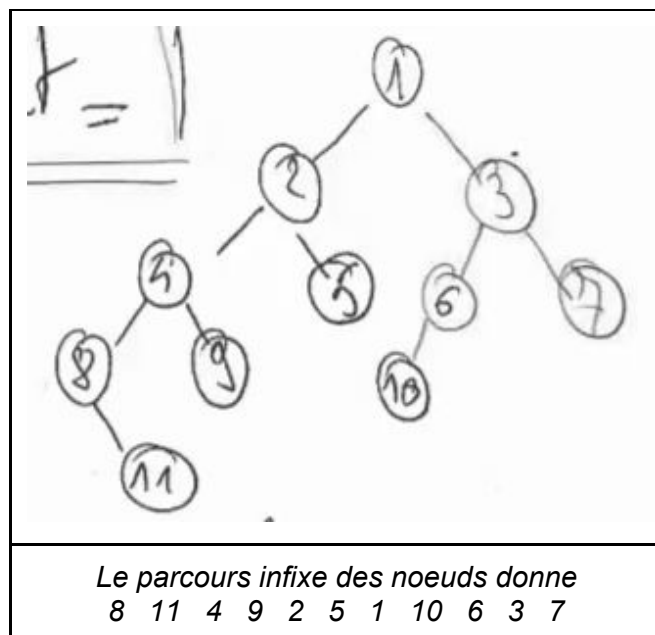
3.6 : Dérécursification d'une fonction récursive double

Lorsque l'on a une fonction où chaque appel récursif engendre deux appels récursifs, on dit que l'on a une récurrence double. C'est typiquement le cas pour les fonctions qui manipulent des arbres binaires. Ces fonctions de manipulation effectuent un parcours de l'arbre donné en argument pour réaliser leurs calculs.

Afin d'illustrer un principe général de dérécursification d'une fonction récursive double, nous allons voir comment dérécursifier le parcours en profondeur en ordre infixe d'un arbre binaire.

Tout d'abord, rappelons la forme générale de la fonction de parcours infixe.

```
let rec infix_path(t : 'a t_btree) : unit =  
  if isEmpty(t)  
  then ()  
  else  
    let r = root(t) in  
    begin  
      infix_path(lson(t));  
      processing(r);  
      infix_path(rson(t));  
    end  
;;
```



On voit que le premier appel récursif sert à descendre le long du sous-arbre gauche et que le traitement d'un noeud se fait lors de la remontée. Pour dérécursifier cela on va empiler les sous-arbres gauches dans une pile que l'on dépilera ensuite pour traiter la racine et effectuer une descente à gauche des sous-arbres droits de chaque arbre qui a été empilé.

En suivant cet algorithme, on commence donc par écrire une fonction `left_path` qui effectue la descente à gauche en empilant les sous-arbres gauches.

```
let left_path(t, p : 'a t_btree * ('a t_btree) stack) : ('a t_btree) stack =
  let a : 'a t_btree ref = ref t
  and p' : ('a t_btree) stack ref = ref p in
  while not(Btree.isEmpty(a))
  do
    p' := push(!a, !p');
    a := lson(!a);
  done;
  !p'
;;
```

On peut maintenant écrire la version itérative du parcours en profondeur en ordre infixe d'un arbre binaire

```
let infix_path(t : 'a t_btree) : unit =
  let p : ('a t_btree) stack ref = ref left_path(t, Stack.empty())
  and a : 'a t_btree ref = ref Btree.empty() in
  while not(Stack.isEmpty(!p))
  do
    a := top(!p);
    processing(root(!a));
    if not(Btree.isEmpty(rson(!a)))
    then p := left_path(rson(!a), depiler(!p));
    else p := depiler(!p);
  done;
  ()
;;
```

Ce schéma de dérécursification s'applique pour toute fonction doublement récursive. L'idée principale est d'empiler les données à utiliser d'abord lors du premier appel récursif et ensuite, sur la même pile, pour chaque second appel récursif.

A titre d'exercice, appliquez cela au calcul des nombres de Fibonacci.

Chap 5 : Les arbres binaires de recherche

lorsqu'on utilise des listes, les opérations de base d'ajout, de suppression et de recherche n'ont pas la même complexité.

Par exemple, l'ajout peut se faire en temps constant (on ajoute en tête de la liste), mais dans ce cas la suppression est en $\Theta(n)$ où n est la longueur de la liste de même pour la recherche d'un élément.

Si la liste est triée, on peut arriver à faire une recherche d'un élément en $\Theta(\log_2 n)$.

Pour certaines applications, il est impensable d'avoir des opérations de base d'ajout, de suppression et de recherche qui ont des complexités identiques et au moins logarithmique en moyenne.

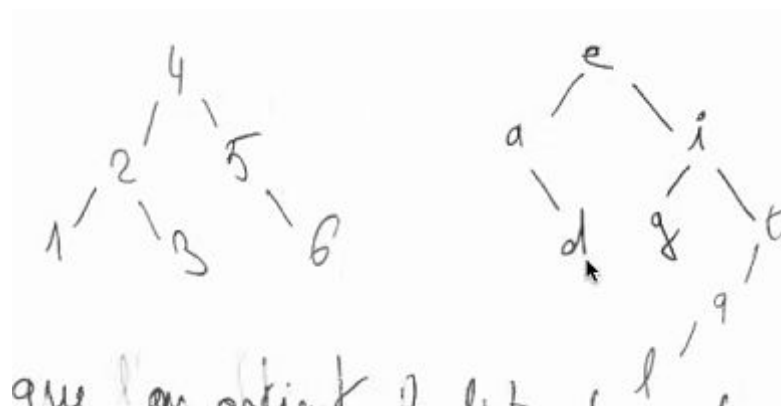
La structure de données qui permet cela est celle des arbres binaires de recherche (ABR)

1. Définitions, Exemples

Définition : Un arbre binaire de recherche (ABR) est un arbre binaire tel que la valeur de sa racine est supérieure à toutes les valeurs des noeuds du sous-arbre gauche et inférieure à toutes les valeurs des noeuds du sous-arbre droit.

Remarquez qu'il est nécessaire que les valeurs des noeuds puissent être ordonnées.

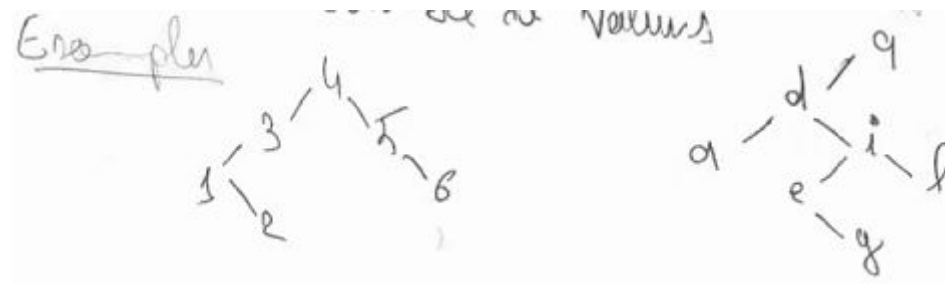
Exemples :



Remarquez que l'on obtient la liste des valeurs des noeuds tris en parcourant un ABR selon l'ordre infixe.

Remarquez aussi que l'on peut associer différents ABR à un même ensemble de valeurs :

Exemples :



On verra que pour un même ensemble de valeurs, les arbres “les mieux équilibrés” sont souvent les plus intéressants.

2. Recherche d'un élément dans un ABR

Pour rechercher un élément dans un ABR, on compare cet élément avec la racine, s'il lui est égal on a trouvé l'élément sinon si l'élément est plus petit que la racine on poursuit la recherche dans le sous arbre gauche sinon on poursuit la recherche dans le sous arbre droit.

Si l'on arrive à l'arbre vide l'élément n'est pas dans l'arbre.

Cet algorithme s'écrit facilement avec le type abstrait des arbres binaires.

`rechercher : Element * t_btree → Bool`

avec `e, v : Element; g, d : t_btree`

axiomes :

- `rechercher(e, empty) = faux`
- `rechercher(e, rooting(v, g, d)) = vrai` si `e = v`
- `rechercher(e, rooting(v, g, d)) = rechercher(e, g)` si `e < v`
- `rechercher(e, rooting(v, g, d)) = rechercher(e, d)` si `e > v`

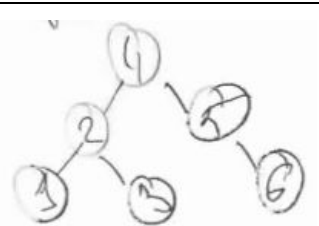
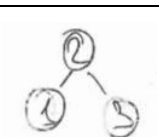

Cet algorithme termine car les appels récursifs se font sur les sous-arbres donc la hauteur de l'arbre diminue à chaque appel et finit soit par atteindre 0, soit par arriver à un racine dont la valeur est l'élément recherché.

Remarquez que dans le pire des cas (l'élément n'est pas dans l'arbre) on parcourt tous les sous-arbres. Dans ce cas la complexité en temps est en $\Theta(h)$ où h est la hauteur de l'arbre.

Pour les arbres parfaits cette recherche est alors en $\Theta(\log_2 n)$ où n est la taille de l'arbre.

Les arbres “équilibrés” vérifient aussi cette propriétés.

Exemple :

On cherche l'élément 3 dans :

Comme $3 < 4$ on cherche 3 dans l'arbre :

Comme $3 > 2$ on cherche 3 dans l'arbre :

Comme $3 = 3$ on a trouvé 3 dans l'arbre de départ

Remarquez que comme on élimine à chaque tour un sous arbre de la recherche, on évite à chaque fois des comparaisons avec environ la moitié des éléments si l'arbre n'est pas trop "déséquilibré". Ce qui explique la complexité dans le cas d'un arbre parfait. Dans le cas extrême d'un arbre filiforme la recherche a une complexité en $\Theta(n)$ dans le pire des cas, comme pour une liste.

3. Ajout d'un élément

Pour ajouter un élément, on doit d'abord trouver l'emplacement où l'ajouter puis réaliser l'ajout.

3.1 : Ajout aux feuilles

Pour ajouter un élément à une feuille d'un ABR, on le compare à la racine pour déterminer s'il faut l'ajouter dans le sous-arbre gauche ou dans le sous-arbre droit et recommence avec le sous-arbre choisi. On s'arrête quand on atteint l'arbre vide car cela correspond à l'emplacement où il faut ajouter l'élément.

Avec le type abstrait des arbres binaires cela donne :

$\text{aj_t_f} : \text{Element} * \text{t_btree} \rightarrow \text{t_btree}$

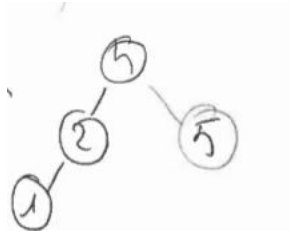
avec $e, v : \text{Element}$; $g, d : \text{t_btree}$

axiomes :

- $\text{aj_t_f}(e, \text{empty}) = \text{rooting}(e, \text{empty}, \text{empty})$
- $\text{aj_t_f}(e, \text{rooting}(v, g, d)) = \text{rooting}(v, \text{aj_t_f}(e, g), d)$ si $e \leq v$

- $\text{ajt_f}(e, \text{rooting}(v, g, d)) = \text{rooting}(v, g, \text{ajt_f}(e, d))$ si $e > v$

exemple : Ajoutons 3 à l'ABR



Cet arbre s'écrit :

$\text{rooting}(4, \text{rooting}(2, \text{rooting}(1, \text{empty}, \text{empty}), \text{empty}), \text{rooting}(5, \text{empty}, \text{empty}))$

Comme $3 \leq 4$ on ajoute dans le sous-arbre gauche, on utilise le deuxième axiome,
 $\text{rooting}(2, \text{rooting}(1, \text{empty}, \text{empty}), \text{empty})$

Comme $3 > 2$ on ajoute dans le sous-arbre droit, on utilise le troisième axiome, empty

Comme l'arbre est vide, on utilise le premier axiome :

$\text{ajt_f}(3, \text{empty}) = \text{rooting}(3, \text{empty}, \text{empty})$

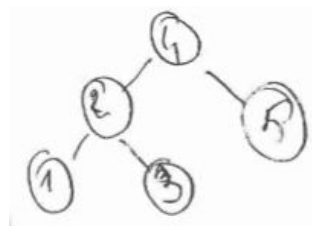
On "remonte" les appels récurifs, on obtient d'abord :

$\text{rooting}(2, \text{rooting}(1, \text{empty}, \text{empty}), \text{rooting}(3, \text{empty}, \text{empty}))$

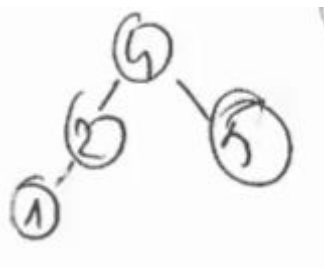
puis :

$\text{rooting}(4, \text{rooting}(2, \text{rooting}(1, \text{empty}, \text{empty}), \text{rooting}(3, \text{empty}, \text{empty})), \text{rooting}(5, \text{empty}, \text{empty}))$

Ce qui se dessine :



Exercice : Écrivez les termes successifs obtenus lors de l'ajout de 6 à l'ABR :

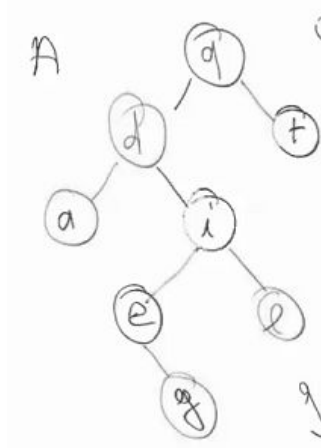


3.2 : Ajout à la racine

Pour ajouter un élément e à la racine d'un ABR, il faut couper l'ABR en deux ABR G et D tel que G contienne tous les éléments inférieurs à e et D contienne tous les éléments supérieurs à e .

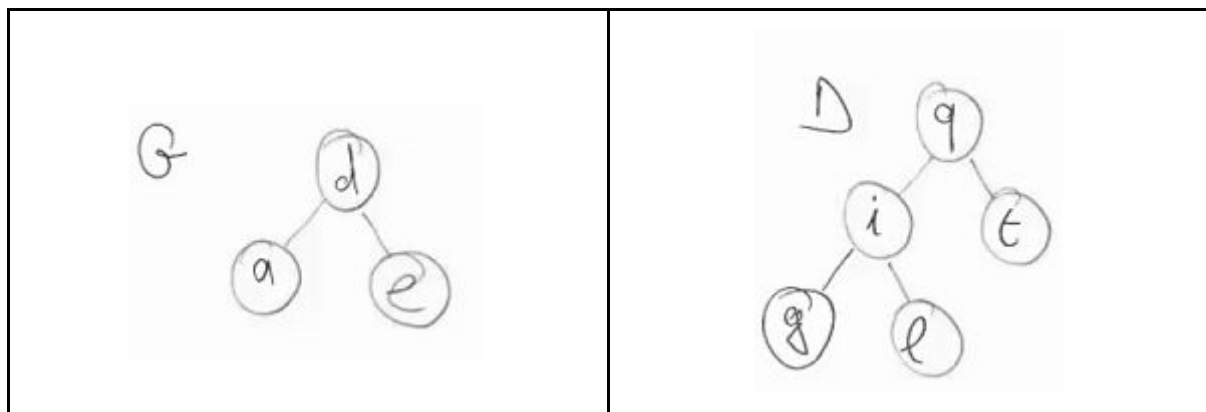
Exemple :

On l'arbre A :



On veut insérer le caractère f à la racine de A .

Il n'est pas nécessaire de parcourir tous les noeuds de A pour former G et D . Les noeuds qui constituent le chemin de recherche de l'élément déterminant la coupure.



Quand on écrit cela avec le type abstrait des arbres binaires cela donne :

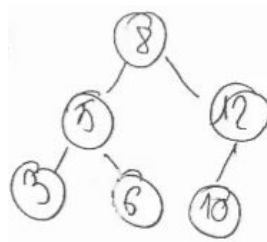
$\text{ajt_r} : \text{Element} * \text{t_btree} \rightarrow \text{t_btree}$

avec $e, v : \text{Element}$; $g, d : \text{t_btree}$

axiomes :

1. $\text{ajt_r}(e, \text{empty}) = \text{rooting}(e, \text{empty}, \text{empty})$
2. $\text{root}(\text{ajt_r}(e, \text{rooting}(v, g, d))) = e$
3. $\text{lson}(\text{ajt_r}(e, \text{rooting}(v, g, d))) = \text{lson}(\text{ajt_r}(e, g))$ si $e \leq v$
4. $\text{rson}(\text{ajt_r}(e, \text{rooting}(v, g, d))) = \text{rooting}(v, \text{rson}(\text{ajt_r}(e, g)), d)$ si $e \leq v$
5. $\text{lson}(\text{ajt_r}(e, \text{rooting}(v, g, d))) = \text{rooting}(v, g, \text{lson}(\text{ajt_r}(e, d)))$ si $e \geq v$
6. $\text{rson}(\text{ajt_r}(e, \text{rooting}(v, g, d))) = \text{rson}(\text{ajt_r}(e, d))$ si $e \geq v$

Exemple :



On insère 7 à la racine :

