STRUCTURE DES ORDINATEURS ET DES SYSTEMES

Johann PAVIS (selon source : M. de SAINTE CATHERINE)

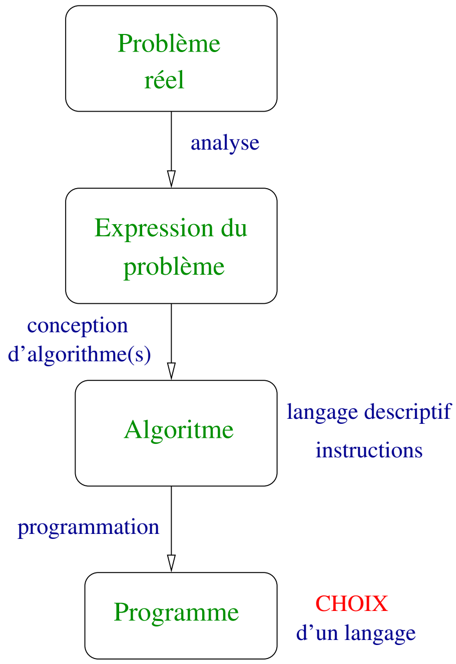
Table des matières

Chapitre 1

INTRODUCTION

1.1 Problèmes

1.1.1 Chaîne de résolution de problèmes



1.1.2 Exemples de problème

1.1.2.1 Résolution d'une équation du 1^{er} degré

⋆ Entrée : a et b réels
⋆ Sortie : x réel tel que $a \cdot x + b = 0$

Remarque : Solution calculable pour $a \neq 0$

⋆ Algorithme :
structure de contrôle → le test
données → a, b, x
type → réel

1.1.2.2 Tri

⋆ Entrée : séquence de n nombres $(a_1, a_2, \ldots, a_n)$
⋆ Sortie : permutation de la séquence pour obtenir $(a'_1, a'_2, \ldots, a'_n)$ telle que

$$\forall (i,j) \in [1,n]\ ,\ i \leq j \Rightarrow a'_i \leq a'_j$$

⋆ Algorithme :
structure de contrôle → boucle
structure de données → séquence : tableau

Remarque : Il existe de très nombreux algorithmes de tri

1.1.2.3 Carte à colorier

On dispose d'une carte que l'on veut colorier de manière à ce que deux zones voisines soient de couleurs différentes.
La carte est représentée par un ensemble de 5 régions et un ensemble de relations binaires de voisinage.
Dans notre cas, on utilisera ceci :

$\mathcal{R} = \{1,2,3,4,5\}$ et $\mathcal{V} = \{(1,2),(1,4),(2,3),(2,4),(2,5),(3,5),(4,5)\}$

On utilise un ensemble de trois couleurs : $\mathcal{C} = \{c_1, c_2, c_3\}$
On cherche la fonction $f : \mathcal{R} \rightarrow \mathcal{C}$ qui répond à notre problème.

⋆ Entrée : couleur et carte
⋆ Sortie : $f : \mathcal{R} \rightarrow \mathcal{C}$ qui vérifie $\forall v = (r_1, r_2) \in \mathcal{V}, f(r_1) \neq f(r_2)$
⋆ Algorithme :
structure de données → graphe (représentation de relations)
type → abstrait (description de données associées à une description des opérations sur celles-ci)

Remarques :

- Ce problème possède plusieurs solutions (6, pour être exact) et est dit **indéterministe**.
- Un algorithme peut énumérer toutes les solutions (programme logique), donner la première trouvée, ou la meilleure selon un (ou plusieurs) critères.
- Une *instance du problème* est un problème dont les *entrées sont fixées.*
- Un algorithme **correct** est un algorithme qui **termine** pour toutes les entrées et dont les sorties doivent respecter les conditions souhaitées.

1.2 Algorithmes

1.2.1 Tri par insertion

```
TRI_INSERTION (tableau A) → tableau
  pour j ← 2 à A.taille
  faire cle ← A[j]
        i ← j−1
        tant que i>0 et A[i]>clé
        faire A[i+1] ← A[i]
              i ← i−1
        A[i+1] ← cle
  retourner A
```

Remarque :

- Les mots-clé sont soulignés
- L'indentation indique les différentes structures de contrôle
- On a une fonction avec entrées et paramètres, sortie et résultat
- On ne fait pas de déclaration, à part pour les fonctions

1.2.2 Langage de description d'algorithme

Penser à bien respecter l'**indentation**. Mettre **une seule** instruction simple par ligne, sans point virgule.

VARIABLES :

Dans l'exemple précédent : i, j, A et *clé*

Une variable possède un *nom*, une *valeur* (qui peut éventuellement changer) et un *type* (*i.e.* le type de sa valeur)

CONSTANTES :

Dans l'exemple précédent : 0, 1, 2

Une constante possède un *type*

TYPES :

Dans l'exemple précédent : `tableau, entier`

Il y a des types de base, soit un ensemble de valeurs : réels, caractères, ...

Il existe également des types plus structurés : séquence, tableau (à n dimensions)

⋆ Instructions simples
- *Affection* : modifie la valeur d'une variable. Elle se symbolise par $\leftarrow$
- *Lire, écrire*

⋆ Structures de contrôle

On utilise une structure séquentielle, exécutée dans un **ordre strict**. On indente les structures de façon verticale.

⋆ Structure conditionnelle

`si p alors` s_1 (`sinon` s_2)

Dans le cas où la condition p est remplie, on exécute l'instruction s_1. Dans le cas contraire, on passe à l'instruction qui suit la boucle (ou bien on exécute s_2).

⋆ Boucles

`tant que p` `faire s`	`répéter s` `jusqu'à p`	`pour i` $\leftarrow$ v_1 `à` v_2 `faire s`

⋆ Fonctions

Une fonction se caractérise par un *nom*, des *paramètres* et des *types de sortie*

⋆ Instruction d'échappement
- *Retourner* : indique le ou les résultat(s) de l'instruction (`return`)
- *Continuer* : à l'intérieur d'une boucle, elle permet de se brancher sur le test (`continue`)
- *Sortir* : sortir d'une boucle

1.3 Récursivité

1.3.1 Conception

- ⋆ Diviser le problème en plusieurs sous-problèmes similaires (de taille inférieure)
- ⋆ Résoudre les sous-problèmes
- ⋆ Combiner les solutions des sou-problèmes pour résoudre le problème initial

Les sous-problèmes ne pouvant plus être divisés sont appelés **plancher** de la récursivité.

Définition 1 *Une fonction est dite récursive si elle fait appel à elle-même.*

1.3.2 Exemples

Calcul de $n!$ avec $n \in \mathbb{N}$

1.3.2.1 Version itérative

```
FACTORIELLE_ITE (n) → entier
   r ← 1
   pour i ← n à 1
   faire r ← r * i
   retourner r
```

1.3.2.2 Version récursive

On sait que : $n! = n \cdot (n-1)!$ et $0! = 1$ (plancher)

```
FACTORIELLE_REC (n) → entier
   si n=0
      alors retourner 1
      sinon retourner n * FACTORIELLE_REC(n−1)
```

Appels	Résultats	Environnements
FATORIELLE_REC(3)	3*FACTORIELLE_REC(2)	n=3
FATORIELLE_REC(2)	2*FACTORIELLE_REC(1)	n=2
FATORIELLE_REC(1)	1*FACTORIELLE_REC(0)	n=1
FATORIELLE_REC(0)	1	n=0

RESULTAT FINAL : 3*2*1*1=6

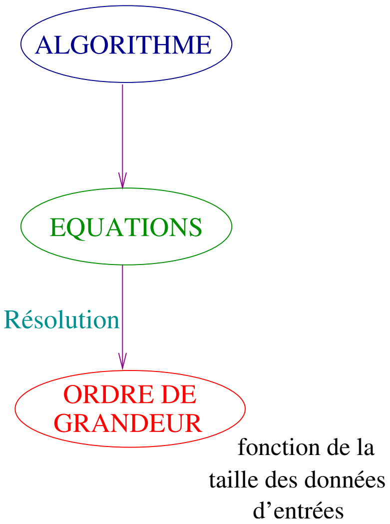
Remarque

Toutes les valeurs associées à n doivent être conservées et retourner au retour des appels récursifs.

Si on utilisait qu'une seule variable n, la dernière valeur écraserait toutes les valeurs précédemment mémorisées et le calcul serait erroné. On utilise à cet effet une **pile**.

Chapitre 2

ANALYSE D'ALGORITHMES



2.1 Modèle de machine

- unité de calculs $rightarrow$ expressions arithmétiques et logiques
- mémoire $\rightarrow$ stockage des données
  On suppose :
- l'accès à un élément de mémoire se fait en un temps fixe
- l'unité de calcul ne peut effectuer qu'une seule opération à la fois
- le coût du transfert d'informations est négligeable devant le coût des opérations

Pour un algorithme $\mathcal{A}$ implémenté dans un programme $\mathcal{P}$ qui s'exécute sur une machine $\mathcal{M}$ à partir de la sonnée d :

- le temps de calcul s'obtient en comptant le **nombre d'opérations effectuées** par $\mathcal{P}$ en utilisant un nombre de cycles horloges des opérations de $\mathcal{M}$
- l'espace mémoire est obtenu en comptant le nombre de données d en entrées et le nombre d'informations supplémentaires pour manipuler ces données, le tout multiplié par le nombre de mots mémoires nécessaires pour les stocker dans la mémoire de $\mathcal{M}$

2.2 Mesure de la complexité

2.2.1 Mesure du coût d'un algorithme

On va associer un coût à chacune des opérations de l'algorithme (*coût* $\leftarrow$ *temps de calcul / espace mémoire*).

On utilise pour cela le LDA :

- une séquence $\rightarrow$ on effectue la somme des coûts des structures en séquence
- structure conditionnelle $\rightarrow$ on exhibe un majorant valable dans les deux cas

2.2.2 Exemple du tri par insertion

Soit t_j le nombre de fois que le test de la boucle `tant que` est effectué pour la valeur j.

Soit $C(n)$ le coût :

$$C_{min}(n) = C_1 \cdot n + (C_2 + C_3 + C_7) \cdot (n-1) + C_4 \cdot \sum_{j=2}^{n} t_j + (C_5 + C_6) \cdot \sum_{j=2}^{n} (t_j - 1)$$

(les C_i sont des constantes fonctions des instructions numéro i de l'algorithme)

Dans le meilleur des cas, on a : $\forall j \in [2, n] t_j = 1$.

Dans ce cas, on a :

$$C(n) = (C_1 + C_2 + C_3 + C_4 + C_7) \cdot (n-1) + C_1$$

$$C(n) = n \cdot (C_1 + C_2 + C_3 + C_4 + C_7) - (C_2 + C_3 + C_4 + C_7)$$

Le pire des cas correspond à un tableau trié à l'envers.

2.2.3 Cas des fonctions récursives

On a : $C(0) = C_1 + C_2$ et $C(n) = C_1 + C_3 + C(n-1)$
Par substitutions successives, on obtient :

$$C(n) = C_1 + C_3 + (C_1 + C_3 + C(n-2))$$

$$C(n) = n \cdot (C_1 + C_3) + C(0)$$

$$C(n) = (n+1) \cdot C_1 + n \cdot C_3 + C_2$$

Complexité en espace
$\mathcal{M}(n) = m + C_m(n)$
On a par ailleurs : $C_1 = C_2 = 0$ et $C_3 = 1$
D'où : $C_m(n) = n$
Les n mots mémoires sont ceux qui servent à stocker les valeurs successives de n.

Remarque Factorielle itérative :
→ temps de calcul linéaire
→ constant en espace mémoire

2.2.4 Configuration des données et complexité

<u>**Définition**</u> **2** *Soit mathcalA un algorithme utilisant un ensemble $\mathcal{D}_n$ de données de taille n. On note $C^{\mathcal{A}}(d)$ la complexité de l'algorithme $\mathcal{A}$ agissant sur la donnée d.*

La complexité dans le meilleur des cas est :

$$C^{\mathcal{A}}_{min}(n) = min\{C^{\mathcal{A}}(d)/d \in \mathcal{D}_n\}$$

(identique pour le maximum)

La complexité en moyenne est :

$$C^{\mathcal{A}}_{moy}(n) = \sum_{d \in \mathcal{D}_n} C^{\mathcal{A}}(d)$$

Remarques :

- On a : $C^{\mathcal{A}}_{min}(n) \leq C^{\mathcal{A}}_{moy}(n) \leq C^{\mathcal{A}}_{max}(n)$
- Si les données sont équiprobables, alors :

$$C^{\mathcal{A}}_{moy}(n) = \frac{1}{Card(\mathcal{D}_n)} \cdot \sum_{d \in \mathcal{D}_n} C^{\mathcal{A}}(d)$$

- En pratique, on s'intéresse surtout à la complexité dans le pire des cas

Exemple Recherche itérative dans un tableau v d'une valeur d

C_1 `i ← 1`
C_2 `tant que` (i$\leq n$ `et` v[i]$\neq d$)
C_3 `faire` `i ← i+1`

On compte le nombre de tours de tests dans la boucle (avec $C_1 = C_3 = 0$ et $C_2 = 1$) :

$$C_{max}(n) = n \text{ et } C_{min}(n) = 1$$

On note q la probabilité que d se trouve dans le tableau v. Si d est bien dans v, toutes les places sont équiprobables. On note $\mathcal{D}_n^i$ l'ensemble des données de taille n où d est à la i^{ème} place et $\mathcal{D}_n^0$ l'ensemble des données de taille n où d est absent.

On a alors :

$$p(\mathcal{D}_n^1) = \frac{q}{n} \left(\text{en effet : } \sum_{i=1}^{n} p(\mathcal{D}_\backslash) = n \cdot \frac{q}{n} = q\right)$$

$$p(\mathcal{D}_n^0) = 1 - q(= 1 - p(\mathcal{D}_n))$$
$$C(\mathcal{D}_n^i) = i(i \in [1, n]) \text{ et } C(\mathcal{D}_n^0) = n$$

Ainsi :

$$C_{moy}(n) = \sum_{i=1}^{n} \frac{q \cdot i}{n} + (1 - q) \cdot n = \frac{q}{2} \cdot (n + 1) + (1 - q) \cdot n$$

En supposant que d est dans v, on a $q = 1$:

$$C_{moy}(n) = \frac{n + 1}{2}$$

Pour $q = \frac{1}{2}$, on obtient :

$$C_{moy}(n) = \frac{3 \cdot n + 1}{4}$$

2.3 Comparaison d'algorithmes

2.3.1 Grandeur des fonctions

Ordre de grandeur asymptotique par rapport à une échelle de comparaison constituée des fonctions usuelles qui sont des **fonctions de référence** pour la complexité des algorithmes.

2.3.2 Notation asymptotique

Notation $\mathcal{O}$

Pour une fonction g donnée, on note $\mathcal{O}(g(n))$ l'ensemble de fonctions suivant :

$$\{f \in \mathcal{F}(\mathbb{R},\mathbb{R}) / \exists (c_1, c_2) \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n > n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

On dit que g est une **borne approchée asymptotique** de f. Elles sont même grandeur asymptotique

Notation o

Elle définit une base asymptotique supérieure.

Pour une fonction g donnée, on note $o(g(n))$ l'ensemble de fonctions suivant :

$$\{f \in \mathcal{F}(\mathbb{R},\mathbb{R}) / \exists c > 0 \text{ et } n_0 \in \mathbb{N}, \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n)\}$$

On note $f(n) = o(g(n))$ et on dit que f est **asymptotiquement dominée** par g.

Remarques

1. $o(g(n)) \subset \mathcal{O}(g(n))$: $f \in o(g(n)) \Rightarrow f \in \mathcal{O}(g(n))$
2. $2n = o(n)$, donc $2n = \mathcal{O}(n^2)$ (on a également : $2n = \mathcal{O}(n^2)$)
3. Pour des pas d'ordres différents, il est parfois utile d'avoir une estimation des constantes :
   si $f(n) = 2n$ et $g(n) = n^2$, on sait que f est dominée par g pour $n > 2$
   si $f(n) = 2n \cdot \log_2 n$ et $g(n) = n \cdot \sqrt{n}$, f domine g pour $n > 256$

2.3.3 Exemple : cas récursif

On divise un problème de taille n en a sous-problèmes de taille $\frac{n}{b}$. On suppose qu'un problème de taille 1 à une complexité unitaire. On suppose que l'assemblage des solutions est de complexité n.

$$C(1) = 1 \text{ et } C(n) = a \cdot C\left(\frac{n}{b}\right) + n$$

On suppose que $b^k = n$

On obtient alors :

$$C(n) = a^i \cdot C\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j \cdot \frac{n}{b^j}$$

Donc, pour $i = k$:

$$C(n) = a^k + \sum_{j=0}^{k-1} a^j = T_1 + \sum_{j=0}^{k-1} a^j$$

On a :

$$\begin{aligned} T_2 = \sum_{j=0}^{k-1} a^j b^{k-1} &= b^k \cdot \sum_{j=0}^{k-1} \left(\frac{a}{b}\right)^j \\ &= b^k \cdot \frac{\left(\frac{a}{b}\right)^k - 1}{\frac{a}{b} - 1} \\ &= b \cdot \frac{a^k - b^k}{a - b} \end{aligned}$$

Ainsi, on a donc les résultat suivants :

<u>si $a > b$</u> :

$\mathcal{O}\left(a^k\right) = T_2(n)$ et $k = \log_b n$

$T_1(n) = \mathcal{O}\left(a^{\log_b n}\right) = n^{\log_b a}$

Complexité : $\mathcal{O}\left(n^{\log_b a}\right)$

<u>si $a < b$</u> :

$\mathcal{O}\left(b^k\right) = \mathcal{O}(n) = T_2(n)$

$T_1(n) = \mathcal{O}\left(a^k\right) = n^{\log_b a}$

Complexité : $\mathcal{O}(n)$

<u>si $a = b$</u> :

$$\sum_{j=0}^{k-1} a^j = k \cdot a^k = \log_b n \times n^{\log_b b}$$

$T_1(n) = n \cdot \log_b(n) = n$

Complexité : $\mathcal{O}(n + n \cdot \log_b n)$

2.3.4 Impact pratique de la notion d'ordre de grandeur de la complexité des algorithmes

On suppose que l'on dispose de 7 algorithmes de complexités dans le pire des cas de, respectivement : $1, \log_2 n, n, n \cdot \log_2 n, n^2, n^3, 2^n$

On suppose que la machine utilisée pour ces algorithmes effectue 10^6 opérations à la seconde.

TABLEAU A

(temps de calcul en fonction de la taille des données)

$\frac{Complexit\acute{e}}{Taille}$	1	$\log_2 n$	n	$n \cdot \log_2 n$	n^2	n^3	2^n
10^2	$-1\mu s$	$6,6\mu s$	$0,1ms$	$0,66ms$	$10ms$	$1s$	$4.10^7 \frac{milliards}{d'ann\acute{e}es}$
10^3	$-1\mu s$	$9,9\mu s$	$1ms$	$9,9ms$	$1s$	$16,6min$	∞
10^4	$-1\mu s$	$13,3\mu s$	$10ms$	$0,13s$	$1,5min$	$11,5j$	∞
10^5	$-1\mu s$	$16,6\mu s$	$0,1s$	$1,64s$	$2,7h$	$31,7a$	∞
10^6	$-1\mu s$	$19,9\mu s$	$1s$	$19,9s$	$11,5j$	$31,7.10^3\ a$	∞

TABLEAU B

(taille maximale du problème en fonction d'un temps donné)

$\frac{Complexit\acute{e}}{Tpscalcul}$	1	$\log_2 n$	n	$n \cdot \log_2 n$	n^2	n^3	2^n
$1s$	∞	∞	10^6	$6,3.10^4$	10^3	10^2	19
$1min$	∞	∞	6.10^7	$2,8.10^6$	7.10^3	4.10^2	25
$1h$	∞	∞	36.10^8	$1,3.10^8$	6.10^4	15.10^2	31
$1jour$	∞	∞	$8,6.10^9$	$2,7.10^9$	$2,9.10^5$	44.10^2	36

TABLEAU C

(augmentation de la complexité en fonction de la taille des données)

Complexité	1	$\log_2 n$	n	$n \cdot \log_2 n$	n^2	n^3	2^n
Évolution du temps quand la taille est multipliée par 10	t	$t + 3,32$	$10 \cdot t$	$\equiv 10^+ \cdot t$	$100 \cdot t$	$1000 \cdot t$	$t \uparrow 10$
Évolution de la taille quand le temps est multiplié par 10	∞	$n \uparrow 10$	$10 \cdot n$	$\equiv 10^- \cdot n$	$3,16 \cdot n$	$2,15 \cdot n$	$n + 3,32$

Les algorithmes utilisables sont ceux en complexité constante, logarithmique, linéaire et, à la limite, quadratique.

Chapitre 3

STRUCTURE DE DONNÉES

3.1 Notion de type abstrait

Représentation interne : issue de choix qui sont plus au moins stables. Elle doit être cachée.

Opérations associées au type : utilisation indépendante de la représentation interne.

3.1.1 Types abstraits de base

Ce sont des types qui correspondent aux types primitifs : `entier, réel, tableau, ...`

3.1.2 Types abstraits construits

Ils sont construits au moyen de types abstraits de base ou d'autres types abstraits construits.

3.2 Définition d'un type abstrait

Spécification fonctionnelle : signature (signature fonctionnelle) et axiomes (propriétés des opérations)

3.2.1 Signature

- ⋆ Ensemble de noms d'ensembles de valeurs : types ou sortes
- ⋆ Ensemble de noms d'opérateur avec leur signature (domaine de départ et d'arrivée)

Exemple

<u>sorte</u> : *booléen*
<u>opérateur</u> : *vrai :* $\rightarrow$ *booléen*
faux : $\rightarrow$ *booléen*
ou : booléen $\times$ *booléen* $\rightarrow$ *booléen*
et : booléen $\times$ *booléen* $\rightarrow$ *booléen*
non : booléen $\rightarrow$ *booléen*

Exemple Séquence d'éléments réels numérotés par des entiers

<u>sorte</u> : *séquence réelle*
<u>utilise</u> : *réel, entier*
<u>opérateur</u> : *séquence_réelle : entier* $\rightarrow$ *séquence_réelle*
i-ème : séquence_réelle $\times$ *réel* $\rightarrow$ *séquence_réelle*
changer_i-ème : séquence_réelle $\times$ *entier* $\times$ *réel* $\rightarrow$ *séquence_réelle*
taille : séquence_réelle $\rightarrow$ *entier*

Remarque La signature, en ce qui concerne *changer_i-ème*, n'indique pas si la séquence réelle résultat est la même séquence que le paramètre dont on aurait modifié le i^{ème} élément, ou bien d'une autre séquence différant seulement par le i^{ème} élément (*i.e.* construction d'une nouvelle séquence).

<u>Définition</u> 3

*Sorte définie : sorte de nom nouveau (*séquence_réelle*, par exemple)*
Sorte prédéfinie : sorte utilisée
Opérations internes : opération ayant pour résultat une sorte définie
*Accesseur ou observateur : opérateur ayant au moins un argument de sorte prédéfinie (*i-ème, taille*)*
Constructeur : opérateur n'ayant pas d'argument de sorte définie et ayant un résultat de sorte définie

3.2.2 Propriétés d'un type abstrait

Elles servent à donner une sémantique aux opérations.

Définition axiomatique 4 *(ensemble de formules)*
Soient i, s et r respectivement une séquence réelle, un entier et un réel

- i-ème *(*séquence_réelle*(n),i) :* ***non défini***
- *Si* $1 \leq i \leq$taille*(s)*
  Alors i-ème(changer_i-ème(s,i,r),i)=r
- *Si* $1 \leq i \leq$taille*(s) et si* $1 \leq j \leq$taille*(s) et si* $i \neq j$
  Alors i-ème(changer_i-ème(s,j,r),i)=i-ème*(s,i)*
- taille*(*séquence_réelle*(n))=n*
- taille*(*changer_i-ème*(s,i,r))=*taille(s)

Remarques :

- consistance : les axiomes ne pas contradictoires
- complétude : les axiomes sont suffisants pour décrire l'ensemble des propriétés du type abstrait
- complétude suffisante : on peut déduire des valeurs pour les accesseurs
- initialisation $\neq$ construction : il est possible d'écrire plusieurs constructeurs pour un même type abstrait.

3.3 Types abstraits de base

Leurs opérations sont unitaires

3.3.1 Types primitifs

entier, réel, caractère, ...
+, *, /, -, log, sin, ...

On les utilise de façon usuelle comme dans un langage de programmation (utiliser des parenthèses pour les indiquer les priorités)

Constructeurs : entier(n), réel(r), mais aussi : i $\leftarrow$ entier(0) (qu'on écrit plus simplement : i $\leftarrow$ 0)

3.3.2 Type tableau

C'est une séquence d'éléments de même type, permettant un **accès direct** à chaque élément, en fonction de son rang.

Chaque accès peut se faire en lecture ou en écriture.

On introduit le type séquence dite générique : le type séquence admet lui-même un paramètre représentant le type des éléments du tableau.

sorte : *séquence*
paramètre : *élément*
utilise : *entier*
opérateur : *seq : entier* $\times$ *élément* $\rightarrow$ *séquence*
i-ème : séquence $\times$ *entier* $\rightarrow$ *élément*
changer_i-ème : séquence $\times$ *entier* $\times$ *élément* $\rightarrow$ *séquence*
taille : séquence $\rightarrow$ *entier*

Pour simplifier les notations, on pourra écrire :

- `t=tab[réel](n,V)` (au lieu de : `t=séq[réel](n,V)`)
- `t[i]` (au lieu de : `i-ème(i)`)
- `t[i]` $\rightarrow$ `e` (au lieu de : `changer_i-ème(t,i,e)`)
- `t.taille` (au lieu de : `taille(t)`)

3.3.3 Type structure

Le type structure permet d'assembler des valeurs de types différents. Une structure set composée de champs ayant un **nom** et un **type**.

sorte : *structure*
paramètre : $E_1, E_2, \ldots, E_n$ *(nombre de pures variables)*
opérateur : *structure :* $id^n \times E_1 \times \ldots \times E_n \rightarrow$ *structure*
e_1 *: structure* $\rightarrow E_1$
$\vdots$
e_n *: structure* $\rightarrow E_n$
*changer_*e_1 *: structure* $\times E_1 \rightarrow$ *structure*
$\vdots$
*changer_*e_n *: structure* $\times E_n \rightarrow$ *structure*

Notations :

- `s=struct[`$E_1, E_2, \ldots, E_n$`]`$(id_1 = V_1, id_2 = V_2, \ldots, id_n = V_n)$
- `s.id`$_i$
- `s.id`$_i$ $\leftarrow$ `V`

3.3.4 Exemple : **matrice**

Signature

sorte : *matrice*

paramètre : *élément*

utilise : *entier*

opérateur : *matrice : entier×entier×élément → matrice*
nb_lignes : matrice → entier
nb_colonne : matrice → entier
i_j-ème : matrice×entier×entier → élément
changer : matrice×entier×entier×élément → matrice

Axiomes m matrice, $(i, j) \in [1, n] \times [1, c] \subset \mathbb{N}^2$ et v élément

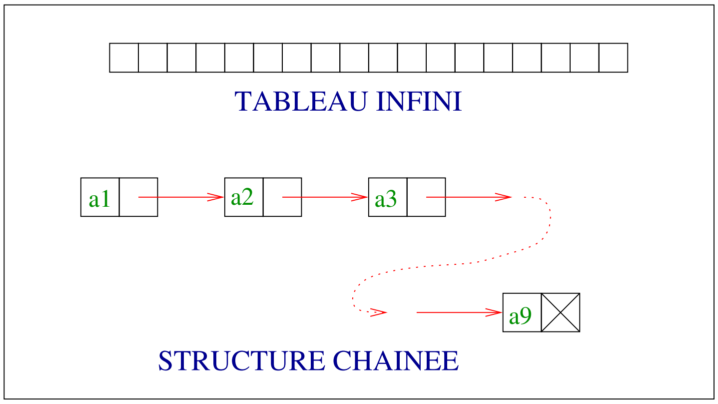
$nb\_ligne(matrice(n, c, v)) = n$
$nb\_colonne(matrice(n, c, v)) = c$
$nb\_ligne(changer(m, i, j, v)) = nb\_ligne(m)$
$nb\_colonne(changer(m, i, j, v)) = nb\_colonne(m)$
$i\_j\text{-}ème(matrice(n, c, v)) = v$
$i\_j\text{-}ème(changer(m, i, j, w), i, j) = w$
$i\_j\text{-}ème(changer(m, i', j', w), i, j) = v$ (où : $(i, j) \neq (i', j')$)

Implémentation au moyen du type tableau (des matrices réelles)

```
matrice[réel](entier n , entier c , réel v) : matrice
    retourner tab[tableau](n , tab[réel](c,v))

nb_colonne(matrice m) : entier
    retourner m.taille

nb_colonne(matrice m) : entier
    retourner m[1].taille

i_j-ème(matrice m , entier i , entier j) : matrice
    retourner m[i][j]

changer(matrice m , entier i , entier j , réel v)
    m[i][j] ← v
    retourner m
```

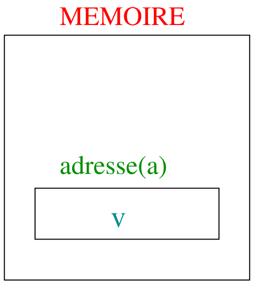
3.4 Structures de données dynamiques

On peut avoir à traiter un ensemble de données de taille inconnue au moment de l'écriture de l'algorithme.
On utilise alors une **structure dynamique contigue** (ou *tableau infini*), ou bien une **structure chaînée** (simple ou double) :



3.4.1 Clarification de la notion d'adresse

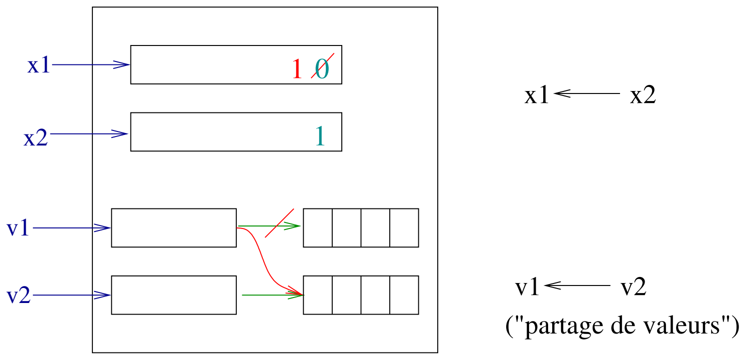
- Identificateurs associés à des valeurs et à un type
- Cette association utilise la notion d'adresse



- Références :
  - <u>*Variable de type primitif*</u>
  - <u>*Variable de type non primitif*</u> : structure, tableau, types construits à partir des précédents
    A une variable de ce type est associée une référence contenant l'adresse de la valeur de la variable

∘ *Affectation* :

- type primitif → lors d'une affectation de variable v_1 par une variable v_2 de type primitif, la valeur contenue dans v_1 est modifiée et reçoit la valeur de v_2
- type non primitif → lors de l'affectation d'une variable v_1 par une variable v_2 de type non primitif, l'adresse contenue dans v_1 est modifiée et reçoit l'adresse contenue dans v_2



Remarque : pour ne pas partager les valeurs, il faut construire un nouvel objet, clone de la valeur de v_2 et réaliser l'affectation avec ce clone : $v_1 \leftarrow tab[entier](v_2.taille, k)$

- Lors du passage en paramètre d'une variable d'un type non primitif, c'est sa référence qui est stockée dans la pile
- On notera `NULL` la valeur de référence ne référençant pas de valeur

3.4.2 Structure chaînée

sorte : *noeud*
paramètre : *élément*
utilise : *booléen* et *noeud*
opérateur : *noeud : élément*$\times$*noeud* → *noeud*
contenu : noeud rightarrow élément
suivant : noeud → *noeud*
changer_suivant : noeud$\times$*noeud* → *noeud*
changer_contenu : noeud$\times$*élément* → *noeud*
suivant ? : noeud → *booléen*

Axiomes : e élément, (s, n) deux noeuds
contenu(noeud(e , n)) = e

$$\textit{contenu(changer\_suivant(n , s)) = contenu(s)}$$
$$\textit{contenu(changer\_contenu(n , e)) = e}$$
$$\textit{suivant} \ldots$$
$$\vdots$$
$$\textit{suivant ?(n)} = \begin{cases} \textit{FAUX, si suivant(n)=NULL} \\ \textit{VRAI, sinon} \end{cases}$$

Implémentation
→ Utiliser une **structure**

3.4.3 Tableau infini

C'est une suite d'éléments, stockés de façon contigue en mémoire, permettant un accès direct à chaque élément par son rang et de taille infinie.

On utilise, pour l'implémentation, un tableau de taille fixe, qui **s'allonge lors d'un débordement**.

Remarque Le coût d'un accès n'est plus unitaire dans le pire des cas.

Signature :

sorte : *tableau_infini*
paramètre : *élément*
utilise : *entier*
opérateur : *tableau_infini : élément → tableau_infini*
changer_i-ème : tableau_infinitimesentier× élément → tableau_infini
i-ème : tableau_infini× entier → élément

Axiomes (v élément, t tableau_infini et (i, j) entiers)

$$\textit{i-ème(changer\_i-ème(t , i , v) , j)} = \begin{cases} \textit{v , si i=j} \\ \textit{i-ème(t , j), sinon} \end{cases}$$

$$\textit{i-ème(tableau\_infini(v) , j) = v}$$

Implémentation (avec une structure)
- tableau de taille fixe
- valeur d'initialisation

3.5 La pile

C'est une structure permettant de stocker des éléments par ordre d'arrivée et les faire sortir deans l'ordre inverse (LIFO : *Last In First Out*).

Signature :

sorte : *pile*

paramètre : *élément*

utilise : *booléen*

opération : *pile : → pile*

empiler : pile× élément → pile

dépiler : pile → pile

sommet : pile → élément

vide ? : pile → booléen

Axiomes (p pile, e élément)

- Les opérations *dépiler* et *sommet* ne sont définies que pour des piles non vides
- *sommet(empiler(p , e)) = e*
- *vide ?(pile()) = VRAI* et *vide ?(empiler(p , e)) = FAUX*
- *dépiler(empiler(p , e)) = p*

Implémentation :

Toutes les opérations de la pile peuvent s'implémenter avec un coût unitaire.

On peut implémenter l'ensemble des éléments d'une pile par un tableau ou par une structure chaînée.

Exemple pile : structure à deux champs

→ tableau des éléments

→ indice de sommet de pile

Pour la pile vide : indice nul, taille du tableau fixée : MAX

```
pile[e]()
    retourner structure[tableau,entier]
            (pile=tableau[élément](MAX,v),sommet=0)
```

```
empiler(pile p , élément e) : pile
      si p.sommet=MAX
          alors ERREUR
      p.sommet ← p.sommet + 1
      p.pile[p.sommet] ← e
      retourner p

dépiler(pile p) : pile
      si vide ?(p)
          alors ERREUR
      p.sommet ← p.sommet - 1
      retourner p

sommet(pile p) : élément
      retourner p.pile[p.sommet]

vide?(pile p) : booléen
      retourner (p.sommet = 0)
```

3.6 La file

Ajouts en queue de file. Suppression en tête de file.
Les opérations de la file peuvent être implémentées avec un coût unitaire.

Signature :

sorte : *file*
paramètre : *élément*
utilise : *booléen*
opérateur : *file : → file*
ajouter : file× élément → file
retirer : file → file
premier : file → élément
vide ? : file → booléen

Axiomes :

- *vide ?(file()) = VRAI*
- *vide ?(ajouter(f , e)) = FAUX*

- $vide\ ?(f) = \begin{cases} \text{VRAI, si premier(ajouter}(f\ ,\ e)) = e \\ \text{FAUX, si premier(ajouter}(f\ ,\ e)) = \text{premier}(f) \end{cases}$
- $vide\ ?(f) = \begin{cases} \text{VRAI, si vide ?(retirer}(f\ ,\ e)) = \text{VRAI} \\ \text{FAUX, sinon} \end{cases}$

Exemple Utilisation de la pile pour l'évaluation des expressions post-fixées

Entrée : tableau e contenant des entiers ou des opérateurs

On dispose d'une fonction permettant de differencier entier et opérateur :

```
eval(tablaeu e) : entier
    p ← pile[entier]()
    pour i de 1 à e.taille
        si e[i] est un entier
            alors p ← empiler(p , e[i])
            sinon a1 ← sommet(p)
                  p ← depiler(p)
                  a2 ← sommet(p)
                  p ← depiler(p)
                  p ← empiler(e[i] appliqué à a2 et a1)
    retourner sommet(p)
```

Implémentation d'une file (par le biais d'une structure)

On peut utiliser un **tableau circulaire**, avec un *pointeur de tête* et un *pointeur de queue.*

On utilise alors les prélicats suivants :

```
file vide ? → tête = queue
file pleine ? → tête = queue + 1
```

3.7 Les listes

Définitions :

- suite d'éléments
- possibilité d'insérer et de supprimer partout dans la liste
- toutes les opérations ne sont pas unitaires (pour optimiser, il faut tenir compte de l'utilisation qui sera faite de la liste)
- il existe plusieurs types de liste

3.7.1 Listes récursives

Listes LISP : $\mathfrak{L}$ est une liste si $\begin{cases} \mathfrak{L} \text{ est vide} \\ \mathfrak{L} \text{ privée de son } 1^{er} \text{ élément est une liste} \end{cases}$.

On appelle le 1^{er} élément de la liste `car(`$\mathfrak{L}$`)` et la liste privée de son 1^{er} élément `cdr(`$\mathfrak{L}$`)`.

Signature :

sortie : *liste_LISP*

paramètre : *élément*

utilise : *booléen*

opérateur : *liste_LISP : → liste_LISP*

car : liste_LISP → élément

cdr : liste_LISP → liste_LISP

cons : élément×liste_LISP → liste_LISP

null ? : liste_LISP → booléen

Au moeyn de cette signature, on peut écrire des fonctions d'ajout et de suppression (n'importe où) dans la liste.

Exemple

```
i-ème : liste_LISP×entier → élément
i-ème(liste_LISP l , entier i)
    si null ?(l)
        alors ERREUR
        sinon si i=1
            alors retourner car(l)
            sinon retourner i-ème(cdr(l) , i-1)
```

3.7.2 Liste avec insertion et suppression

Signature :

sortie : *liste*

paramètre : *élément*

utilse : *entier* et *booléen*

opérateur : *liste :* $\rightarrow$ *liste*

i-ème : liste$\times$*entier* $\rightarrow$ *élément*

insérer : liste$\times$*entier*$\times$*élément* $\rightarrow$ *liste*

supprimer : liste$\times$*entier* $\rightarrow$ *liste*

liste_vide : liste $\rightarrow$ *booléen*

Implémentation :

tableau de taille n : *i-ème* est unitaire

insérer et *supprimer* sont dans le pire des cas en $\mathcal{O}(n)$

Pour la structure chaînée, toutes les opérations sont en $\mathcal{O}(n)$

Utilisation :

Supprimer le $\text{k}^{ème}$ élément ayant une valeur v. Ceci s'effectue en $\mathcal{O}(n^2)$ dans le pire des cas.

Deuxième signature possible

On définit un curseur, pour éviter les parcours multiples de la liste. DEs fonctions sont associées au curseur pour le positionner :

- *début(l)* $\rightarrow$ qui place le curseur au début de la liste si elle n'est pas vide
- *fin(l)*
- *avant(l)* et *arrière(l)* $\rightarrow$ font se déplacer le curseur d'un élément (dans le sens "positif" ou "négatif")
- *aller(l , i)* $\rightarrow$ positionne le curseur sur le $\text{i}^{ème}$ élément de la liste
- *position(l)* $\rightarrow$ qui donne la position courante du curseur

On définit alors également les prédicats : *trop_à_gauche(l)* et *trop_à_droite(l)* qui valent VRAI lorsque le curseur dépasse la première / dernière position.

Fonctions d'insertion et de suppression

- *supprimer(l)* $\rightarrow$ supprime l'élément de ranf égal au curseur
- *insérer_à_droite(l , v)* $\rightarrow$ insére v à droite de l'élement de rang égal au curseur
- *idem* pour la fonction *insérer_à_gauche(l , v)*

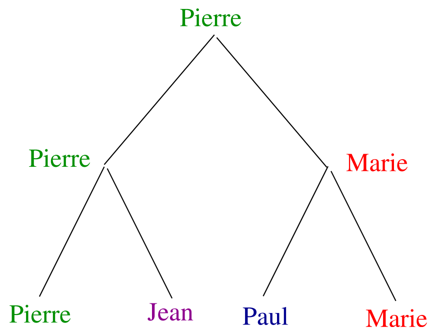
$\Rightarrow$ Il faut définir pour chaque opération l'action sur le curseur.

Chapitre 4

STRUCTURE DE DONNEES ARBORESCENTE

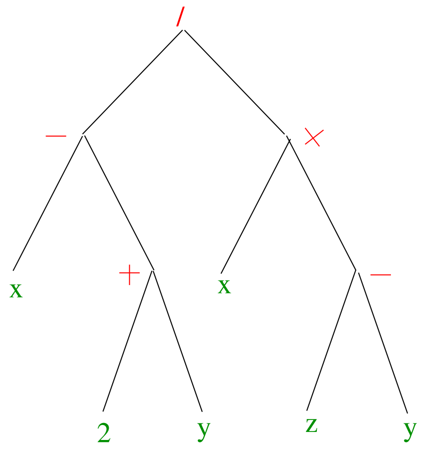
4.1 Arbres binaires

Exemple Tournoi d'échec



Expression arithmétique binaire :

$$\frac{x - 2y}{x + (y - z)}$$



4.1.1 Définitions

Un arbre bianire est :
→ vide (noté $\emptyset$)
→ $B =< r, B_1, B_2 >$ où B_1 et B_2 sont deux **arbres binaires disjoints** et r un noeud appelé **racine**.

Remarque Les sous arbres sont ordonnés. On repère les noeuds des arbres en leur donnant des noms symboliques différents

4.1.2 Signature

<u>sorte</u> : *arbre_binaire*
<u>utilise</u> : *noeud_binaire, booléen*
<u>opérateur</u> : *vide :* $\rightarrow$ *arbre_binaire*
$< \_, \_, \_ >$: *noeud_binaire*$\times$*arbre_binaire*$\times$*arbre_binaire* $\rightarrow$ *arbre_binaire*
racine : arbre_binaire $\rightarrow$ *noeud_binaire*
gauche : arbre_binaire $\rightarrow$ *arbre_binaire*
droit : arbre_binaire $\rightarrow$ *arbre_binaire*

Axiomes :

- *racine*($< r, B_1, B_2 >$)=*r*
- *gauche*($< r, B_1, B_2 >$)=B_1
- *droit*($< r, B_1, B_2 >$)=B_2

4.1.3 Terminologie

- ⋆ Soit $< r, B_1, B_2 >$ un arbre binaire. La racine de B_1 est l'**enfant (ou fils) gauche** de r. De même, celle de B_2 est le **fils droit** de r.
- ⋆ Soit $\mathcal{B} =< r, B_1, B_2 >$ un arbre binaire. r est **parent** des racines de B_1 et B_2, qui sont frères au sein de l'arbre $\mathcal{B}$.
- ⋆ Un noeud m_i est descendant de m_j si **son parent est** m_j ou bien si son parent est **un descendant de** m_j.
- ⋆ L'ascendance est la relation inverse de la descendance.
- ⋆ Les noeuds d'un arbre binaire ont **au plus** 2 fils. Un *noeud interne* a un ou deux fils, alors qu'un *noeud externe* (ou **feuille**) n'a pas d'enfant.
- ⋆ Un **chemin (ou branche)** est une suite de noeuds consécutifs allant de la **racine à une feuille**.

4.1.4 Caractéristiques d'un arbre binaire

La hauteur (ou profondeur) d'un noeud est donnée par la fonction suivante :

$$h : \left\{ \begin{array}{l} \{arbres\_binaires\} \rightarrow \mathbb{N} \\ x \mapsto \left\{ \begin{array}{l} 0, \text{ si } x \text{ est racine} \\ 1 + h(y), \text{ si } y \text{ est parent de } x \end{array} \right. \end{array} \right.$$

La profondeur d'un arbre binaire $\mathcal{A}$ est :

$$max \{h(x)/x \in \{noeuds\}\}$$

4.1.5 Bornes sur la profondeur d'un arbre binaire

Proposition 5 *Pour tout arbre bianire de taille n et de profondeur p et possèdant f feuilles, on a :*

$$\lfloor \log_2 n \rfloor \leq p \leq n - 1 \;\; et \; p \geq \lceil \log_2 f \rceil$$

Preuve :

Soit p une hauteur fixée.

- l'arbre binaire ayant le plus petit nombre de sommets a des internes unaires et une feuille : $p \leq n - 1$

– l'arbre binaire ayant le plus de sommets a des noeuds internes binaires. On a alors :

$$\left(\sum_{i=0}^{p} 2^i = 2^{p+1} - 1\right) \text{ sommets}$$

Donc : $n \leq 2^{p+1} - 1$, d'où : $\log_2 n < p + 1$
Soit : $\lfloor \log_2 n \rfloor \leq p$

– On a : $f \leq 2^p$. Donc : $p \geq \lceil \log_2 f \rceil$

4.1.6 Arbres binaires complets

Définition 6 *Un arbre binaire complet a des noeuds internes binaires (*i.e. *ayant tous deux fils exactement)*

Proposition 7 *Un arbre binaire complet possèdant n noeuds internes possède $n + 1$ feuilles.*

Preuve :

- $n = 0 \rightarrow$ l'arbre est réduit à une feuille
- Soit $n \in \mathbb{N}$. Supposons vraie la propriété pour les arbres à n noeuds internes au plus.
  Soit $\mathcal{B} =< r, B_1, B_2 >$ un arbre binaire complet à $n+1$ noeuds internes.
  B_1 et B_2 sont des arbres complets à n_1 et n_2 noeuds internes, avec : $n + 1 = n_1 + n_2 + 1$
  La récurrence peut s'appliquer à B_1 et B_2. Ils possèdent ainsi respectivement $n_1 + 1$ et $n_2 + 1$ feuilles.
  Donc $\mathcal{B}$ possède $n_1 + n_2 + 2$ feuilles (CQFD)

Proposition 8 *Il existe une bijection entre les arbres binaires de taille n et les arbres binaires complets de taille $2.n + 1$*

Preuve :

On retire toutes les feuilles d'un arbre binaire (ou on ajoute des feuilles de façon à ce que tous les noeuds internes de l'arbre binaire est deux enfants).

4.1.7 Parcours d'arbres

Un parcours est l'examen systématique, dans un certain ordre, des noeuds afin d'y effectuer un traitement.

4.1.7.1 Parcours en profondeur

⋆ Gauche : Ce parcours consiste à partir de la racine et à choisir toujours l'enfant gauche en premier.

Remarques :

- Chaque noeud de l'arbre ayant deux enfants est rencontré *trois fois* lors de ce parcours (1 fois lors du parcours de l'arbre gauche, 1 fois lors du parcours de l'arbre droit, 1 fois en descendant)
- Le parcours s'applique aux arbres binaires en général. Pour la définir par un algorithme, on transforme l'arbre en un arbre binaire complet en ajoutant des noeuds vides.

Algorithme :

```
parcours_profondeur(arbre_binaire a)
      si a ≠ ∅
      alors parcours_profondeur(gauche(a))
            parcours_profondeur(droit(a))
```

Trois ordres à ce parcours :

- préfixe : traitement des noeuds lors de la $1^{ère}$ rencontre
- infixe : traitement lors du 2^{nd} passage
- postfixe : traitement lors du $3^{ème}$ passage

(on utilise une **pile** pour mémoriser les résultats)

Odre préfixe :

```
parcours_préfixe(arbre_binaire a)
   si a ≠ ∅
   alors print(racine(a))
         parcours_préfixe(gauche(a))
         parcours_préfixe(droit(a))
```

<u>**Ordre infixe**</u> :

```
parcours_infixe(arbre_binaire a)
   si a ≠ ∅
   alors parcours_infixe(gauche(a))
         print(racine(a))
         parcours_infixe(droit(a))
```

Ce parcours donne l'écriture "usuelle" des expressions arithmétiques

<u>**Ordre postfixe**</u> :

```
parcours_postfixe(arbre_binaire a)
   si a ≠ ∅
   alors parcours_postfixe(gauche(a))
         parcours_postfixe(droit(a))
         print(racine(a))
```

Remarque Les algorithmes ont été écrits de façon récursive. Pour le faire de manière itérative, il est nécessaire de mémoriser les noeuds situés entre la racine et le noeud courant, à l'aide d'une pile.

```
parcours_infixe_ite(arbre_binaire r)
   p ← pile()
   n ← r
   répéter
      si n neq ∅
         alors p ← empiler(p , n)
               n ← gauche(n)
         sinon n ← sommet(p)
               p ← depiler(p)
               print(racine(n))
               n ← droit(n)
  jusqu'à vide(p) et n=r
```

⋆ <u>Droit</u> : On définit les mêmes notions, mais le parcours s'effectue d'abord sur le sous arbre droit (*i.e.* le fils droit).

4.1.7.2 Parcours en largeur

Il consiste à explorer les enfants d'un noeud avant ses petits enfants.

Algorithme :

```
parcours_largeur(arbre x , file f)
  si x ≠ ∅
      alors print(racine(x))
            f ← ajouter(f , gauche(x))
            f ← ajouter(f , droit(x))
            x ← premier(f)
            f ← retirer(f)
            parcours_largeur(x , f)
  sinon si non(vide(f))
        alors x ← premier(f)
              f ← retirer(f)
              parcours_largeur(x , f)
```

4.2 Arbre binaire de recherche

4.2.1 Définition

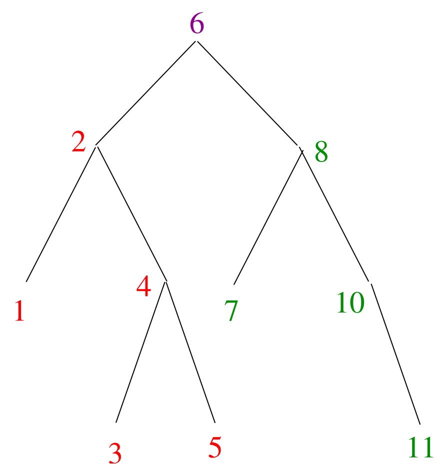
Un arbre binaire de recherche est un arbre binaire dans lequel on stocke des éléments (clés) admettant un ordre total. Les clés sont organisées selon la propriété suivante :

Soit x la racine de $\mathcal{B}$.

Si y est un noeud du sous arbre gauche de $\mathcal{B}$, alors la clé de x est plus grande que celle de y.

Si y est un noeud du sous arbre droit de $\mathcal{B}$, alors la clé de x est plus petite que celle de y.

4.2.2 Exemple



4.2.3 Implémentation d'un arbre binaire de recherche

Définition 9 *On utilisera une structure à 4 champs :*
clé, enfant gauche, enfant droit *et* parent

Type concret : *structure*

Type abstrait :

sorte : *noeud_binaire*

paramètre : *élément*

utilise : *noeud_binaire* et *booléen*

opérateur : *noeud_binaire : élément×noeud_binaire*^{3} → *noeud_binaire*

clé : noeud_binaire → *élément*

droit : noeud_binaire → *noeud_binaire*

gauche : noeud_binaire → *noeud_binaire*

parent : noeud_binaire → *noeud_binaire*

changer_cle : noeud_binaire×élément → *noeud_binaire*

changer_gauche : noeud_binaire×noeud_binaire → *noeud_binaire*

changer_droit : noeud_binaire×noeud_binaire → *noeud_binaire*

changer_gauche : noeud_binaire×noeud_binaire → *noeud_binaire*

Implémentation : *structure[élément,noeud_binaire,noeud_binaire,noeud_binaire]*
(clé,gauche,droit,parent)

Notation :

- *n=noeud_binaire(x,n_1,n_2,p)*
- *print(n.clé)* ou *print(clé(n))*
- *n.clé* ← x ou n ← *changer_clé(n,x)*

Implémentation des arbres binaires de recherche

```
vide()
    b ← structure[noeud_binaire](racine==NULL)
    retourner b

<r,b1,b2 >
    n ← noeud_binaire(r.clé,b1.racine,b2.racine,NULL)
    b1.parent ← n
    b2.parent ← n
    b ← structure[noeud_binaire](racine=r)
    retourner b

racine(arbre_binaire x)
    retourner x.racine

gauche(arbre_binaire x)
    b ← structure[noeud_binaire](racine=x.racine.gauche)
    retourner b

droit(arbre_binaire x)
    b ← structure[noeud_binaire](racine=x.racine.droit)
    retourner b
```

Seconde implémentation

Il est parfois commode et pratique de confondre les types *arbre_binaire* et *noeud_binaire*, lorsque cela est possible.

```
vide()
    retourner NULL

<r,b1,b2 >
    n ← noeud_binaire(r,b1,b2,NULL)
    b1.parent ← n
    b2.parent ← n
retourner n
```

```
racine(x)
    retourner x.clé

gauche(x)
    retourner x.gauche

droit(x)
    retourner x.droit
```

4.2.4 Parcours infixe à gauche

```
parcours_infixe(x)
    parcours_infixe(x.gauche)      % (écrire des clés plus petites)
    print(x)
    parcours_infixe(x.droit)      % (écrire des clés plus petites)
```

Ceci a pour effet d'afficher les clés dans l'ordre croissant.

4.2.5 Requêtes dans un arbre binaire de recherche

- rechercher une clé
- successeur et prédécesseur d'une clé
- maximum et minimum de l'arbre

RECHERCHER : paramètre : *noeud, clé*

retourner : *référence du noeud contenant la clé ou NULL*

```
arbre_rechercher(x , k)
    si x.clé=k ou x=NULL
        alors retourner x
    si k<x.clé
        alors retourner arbre_rechercher(x.gauche , k)
        sinon retourner arbre_rechercher(X.droit , k)
```

Remarque Cette fonction passe d'un paramètre x à un l'un de ses enfants ou bien s'arrête.
Elle construit donc un chemin issu du noeud initial (la racine) jusqu'au noeud recherché, ou bien jusqu'à une feuille. Sa complexité en donc bornée par la taille de l'arbre.

Version itérative

```
arbre\_rechercher_ite(x , k)
   tant que x≠NULL et x.clé≠k
   faire si x.clé>k
            alors x ← x.gauche
            sinon x ← x.droit
   retourner x
```

Remarque La complexité en temps est la même. La complexité en espace varie selon l'implémentation et la récursivité du langage.

En général, on utilise une pile ⇒ $(taille(x) + taille(k)) * p$, avec p la profondeur de l'arbre.

MINIMUM :

paramètre : *noeud, racine de l'arbre dont on recherche le minimum*

```
arbre_minimum(x)
   tant que x.gauche≠NULL
   faire x ← x.gauche
   retourner x
```

Remarque Si x n'a pas d'enfant gauche, la racine est le minimum.

La fonction maximum est la symétrique de la fonction minimum

SUCCESSEUR ET PRÉDÉCESSEUR :

paramètre : *noeud x*

retourne : *noeud y dont la clé est le minimum de l'ensemble des clés qui sont plus grandes que celle de x*

2 cas peuvent se présenter :

- ⋆ si x a un sous-arbre droit, son successeur est le minimum de ce sous-arbre
- ⋆ sinon, il faut remonter à partir de x jusqu'à emprunter un lien gauche. Le successeur de x est le parent admettant ce lien gauche

```
arbre_successeur(x)
    si x.droit≠NULL
        alors retourner arbre_minimum(x.droit)
    y ← x.parent
    tant que y≠NULL et x=y.droit
    faire x ← y
          y ← x.parent
    retourner y
```

Remarque Complexité en $\mathcal{O}(p)$. Le successeur d'un noeud à 2 enfants a au plus 1 enfant.

4.2.6 Insertion et suppression

```
insérer(T , z)
   x ← T.racine
   y ← NULL
   tant que x≠NULL
   faire y ← x
         si z.clé<x.clé
               alors x ← x.gauche
               sinon x ← x.droit
         z.parent ← y
   si y=NULL      % insertion à la racine
         alors x ← z
         sinon si y.clé>z.clé
                   alors y.droit ← z
                   sinon y.gauche ← z
```