Algorithmique et Structures de Données

Myriam Desainte-Catherine

ENSEIRB – LaBRI – Université Bordeaux I

Cours destiné à la première année de la filière informatique et de la filière télécommunication de l'ENSEIRB

BIBLIOGRAPHIE

Jean Roman *cours manuscrit*

Thomas Cormen, **Charles Leiserson**,
Ronald Rivest *Introduction à l'algorithmique*

Première version 2001/2002

Table des matières

Chapitre 1

Introduction

- Étude des problèmes calculatoires
- Algorithme = outil de résolution d'un problème.
- Algorithme de mise en oeuvre : introduction d'une notation, langage de description algorithmique et de description des données.
- Un problème a généralement plusieurs solutions algorithmiques.
- Analyse d'algorithmes (complexité) : temps de calcul, taille des données, croissance en fonction de la taille des données.

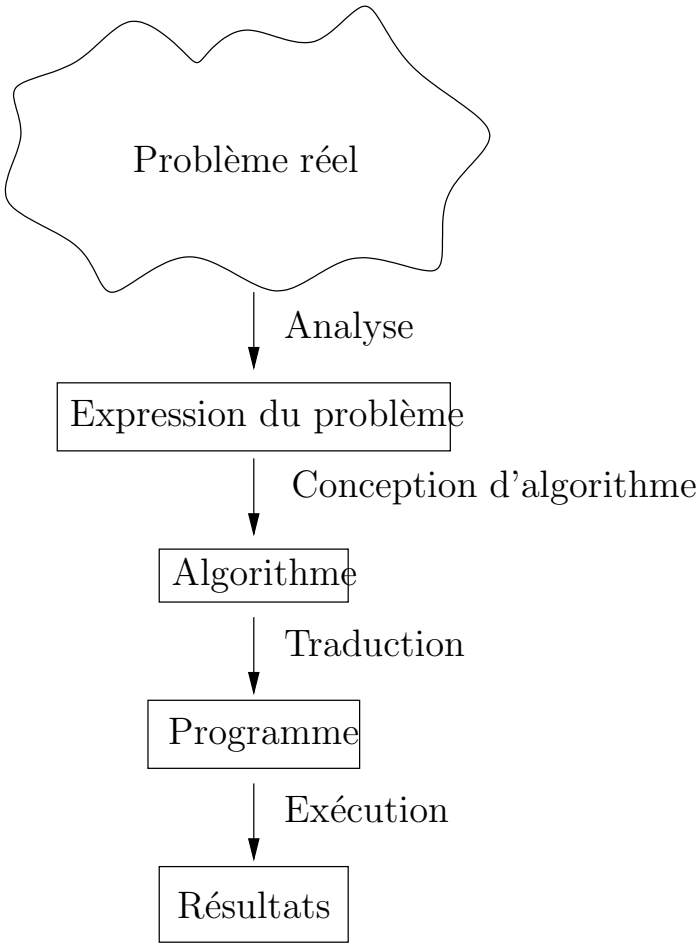
Des problèmes, des solutions algorithmiques, comparaisons selon temps calcul, espace mémoire, facilité écriture, ...

1.1 Problèmes

1.1.1 Chaîne de résolution de problèmes

Du problème à sa solution, il y a une chaîne de traitements dans laquelle l'algorithmique prend place.

Un problème admet des **entrées** et des **sorties**.



ASD : La phase d'analyse correspond à la recherche d'une expression mathématique d'un problème réel. Cette expression se présente généralement sous la

forme de **relations entre les entrées et les sorties**. (Elle présuppose une connaissance parfaite du problème réel).

La phase de conception d'algorithme consiste à construire une procédure de calcul permettant d'établir les relations désirées entre les données d'**entrée** et les données de **sorties**. Cette procédure s'exprime généralement au moyen d'un ensemble d'**instructions** à exécuter dans un **ordre précis**. En effet, il s'agit de décrire un calcul s'effectuant possiblement en plusieurs étapes. L'algorithme apparaît comme un **outil de résolution** d'un problème.

Programmation : La traduction de l'algorithme présuppose le choix d'un langage de programmation. L'exéxution du programme nécessite la donnée des entrées.

L'**ASD** fournit des concepts généraux sur les algorithmes et les structures de données qui sont utiles pour tous les langages de programmation.

La **programmation** consiste à prendre en considération la modularisation des programmes selon les structures syntaxiques disponibles. Elle gère aussi des situations plus concrètes comme la gestion de la mémoire, les entrées sorties, la gestion des erreurs. Les **langages de programmation** : machine et évolués. Classes ou paradigmes de langages de programmation impliquant différentes méthodologies de programmation.

Le **génie logiciel** consiste à prendre en compte la totalité de la chaîne qui est en fait un cycle, compte tenu des erreurs possibles à chaque étape et de la maintenance du logiciel résultant.

1.1.2 Exemples de problèmes

Expressions plus ou moins immédiates selon les problèmes. L'expression des **relations** définissant le problème nécessite l'expression des données d'**entrée** et de **sortie**. Pour l'expression du problème, on utilise des définitions mathématiques. Mais pour concevoir un algorithme, on utilisera des **Structures de contrôle** permettant d'enchaîner les étapes du calcul, des **Structures de données**, qui sont une description de données implémentées dans la mémoire d'un ordinateur, et de **Types abstraits**, qui sont des constructions abstraites permettant d'associer des données et des opérations utiles pour l'expression de l'algorithme.

1. RÉSOLUTION D'UNE ÉQUATION DU PREMIER DEGRÉ :

   Entrée : a et b réels

   Sortie : x réel tel que $ax + b = 0$

   Solution calculable si $a \neq 0$. Méthode de calcul connue $x = \frac{-b}{a}$.

   Structure de données : variable et constante.

   Structure de contrôle : test

2. TRI

   Entrée : Une séquence de n nombres $< a_1, a_2, \ldots, a_n >$.

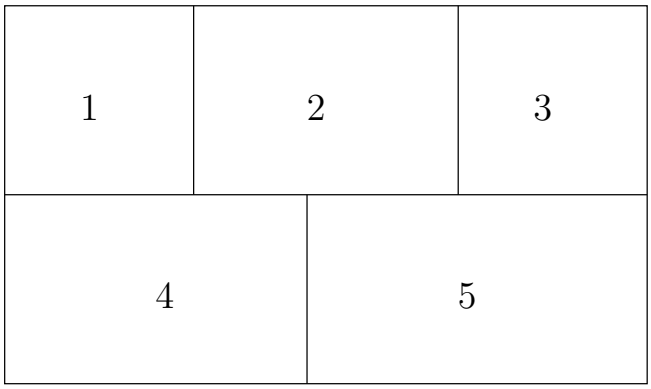
   Sortie : Une permutation $< a'_1, a'_2, \ldots, a'_n >$ de la séquence telle que $a'_1 \leq a'_2 \ldots \leq a'_n$.

   Solutions algorithmiques diverses à comparer selon temps de calcul, taille données, cas de configurations des données (séquence presque triée, ...).

   Structure de données: notion de séquence de données $\longrightarrow$ tableaux

Structure de contrôle : boucle *Exemple.* Donner une séquence d'entrée et de sortie.

3. CARTE À COLORIER : il s'agit de colorier une carte avec un certain nombre de couleurs. De façon à ce qu'elle soit lisible, deux régions voisines ne doivent pas admettre la même couleur.



Soit la carte c représentée sur la figure. Elle admet l'ensemble de régions $R = \{1, 2, 3, 4, 5\}$. La relation de voisinage entre deux régions est donnée par l'ensemble des couples : $V = \{(1,2),\ (1,4),\ (2,3),\ (2,4),\ (2,5),\ (3,5),\ (4,5)\}$. Considérons l'ensemble de trois couleurs $C = \{c_1, c_2, c_3\}$. Le résultat recherché est une fonction de R dans C satisfaisant les critères souhaités. Le problème général peut s'exprimer comme suit :

Entrée : $c = (R, V)$, C

Sortie : $i : R \longrightarrow C$, telle que $\forall v = (r_1, r_2) \in V, i(r_1) \neq i(r_2)$.

Type abstrait graphe pour représenter une relation. Implémentation au moyen de *structure de données* telles que tableaux à 2 dimensions, ou de structures chaînées.

1.1.3 Remarques

Parfois, un problème admet plusieurs solutions. Un problème est dit **déterministe** si il admet une seule solution. Sinon, il est dit indéterministe. Dans le cadre de ce cours, nous considèrerons surtout des problèmes déterministes. Dans les cas indéterministes, il est possible de concevoir des solutions algorithmiques énumérant toutes les solutions, ou bien produisant la solution la meilleure selon un critère. (Voir programmation logique).

On appelle **Instance d'un problème** un problème muni de toutes ses entrées nécessaires pour calculer la solution.

Il est malheureusement possible d'écrire des algorithmes incorrects (produisant des résultats ne vérifiant pas les relations requises) ou ne se terminant pas. Un algorithme est **correct** si pour chaque séquence d'entrée il se termine et fournit la sortie correcte.

1.2 Algorithmes

Les algorithmes seront décrits au moyen d'un **pseudo-code** proche des langages C, Pascal ou Algol. Parfois le langage naturel pourra être utilisé. Pour l'introduire, nous présentons d'abord l'exemple du tri par insertion.

1.2.1 Tri par insertion

- Main droite : cartes non triées,
- Main gauche : cartes triées.
- Initialisation : main gauche vide.

– Insertion de la première carte de la main droite dans la main gauche à sa place.

TRI-INSERTION(tableau A) $\longrightarrow$ tableau
pour $j \longleftarrow 2$ **à** $A.taille$
    faire $cle \longleftarrow A[j]$
        $i \longleftarrow j - 1$
        tant que $i > 0$ **et** $A[i] > cle$
            faire $A[i+1] \longleftarrow A[i]$
                $i \longleftarrow i - 1$
        $A[i+1] \longleftarrow cle$
        retourner A

Il s'agit d'une fonction qui s'appelle TRI-INSERTION, prenant comme entrée ou **paramètre** le tableau A et ayant en sortie ou en **résultat** un tableau. On accède au i-ème élément du tableau A par la notation $A[i]$. Le nombre d'éléments du tableau A est obtenu par $A.taille$.

– Deux boucles imbriquées. La première est la plus externe, la seconde, la plus interne.
– Main droite : A de $j + 1$ à *longueur* de A
– Main gauche : A de 1 à j
– La première carte à transférer est la $j$ième. La première valeur de j est donc deux. (Le transfert de la première carte est réalisé en affectant j à 2).

1.2.2 Éléments de base du langage de description algorithmique

La présentation des algorithmes est donnée par une **indentation** s'appliquant aux instructions d'un même bloc séquenciel. Le découpage en lignes est tel qu'une instruction simple apparaît seule sur sa ligne.

Dans cette présentation, nous notons en minuscules et en gras les mots clef du langage, et en majuscules, les autres formes.

1.2.3 Variables

Une **variable** possède

- un **nom** (identificateur) mnémonique fixé par l'utilisateur et permettant de manipuler l'objet,
- une **valeur** courante (compatible avec le type) qui peut varier au cours du déroulement de l'algorithme.
- un **type**, associé à la valeur de la variable.

Les variables de l'exemple sont : j, cle, A, i.

1.2.4 Constantes

Une **constante** est un objet similaire à une variable, mais ayant une valeur fixe. Une constante a un type. Les constantes de l'exemple sont 2, 1, 0.

1.2.5 Types

Le **type** d'une variable (ou d'une constante) peut être **simple** (scalaire entier, réel, ...) ou **structuré** (tableau à une ou plusieurs dimensions, structures, ...). Les types de l'exemple sont : entier (j, i, cle, 1, 2, 0), tableau d'entiers à une dimension (A).

Remarque. Pas de déclarations de variables. Seules, les fonctions sont signées. (cf types abstraits).

1.2.6 Instructions simples

- L'affectation : l'instruction d'**affectation** modifie la valeur d'une variable. Elle est notée ⟵. Les affectations de l'exemples permettent de modifier les indices pour parcourir et décaler le tableau, et modifier la valeur de cle. Affectation multiple : x,y ←f(z).
- lire, écrire, ...

1.2.7 Structures de contrôle

Une **structure de contrôle** est un ensemble d'instructions qui s'exécutent **séquentiellement**. Ces instructions sont soit des instructions simples, soit des structures de contrôle.

Donner la syntaxe en langage C des structures de contrôle.

– Structure séquentielle :

s_1

s_2

...

s_n

Exemple : lignes 2 et 3, 5 et 6.

– Structures conditionnelles ou sélectives (ou tests):

si P
alors S_1
sinon S_2

si P
alors S_1

Décrire le fonctionnement.

– Structures itératives ou répétitives (ou boucles):

tant que P
faire S

répéter S
jusqu'à ce que $\not P$

pour I **à** P
faire S

Dans chaque cas, la structure de contrôle S est appelée le **corps de la boucle**.

Exemple : lignes 1 et 4.

Décrire le fonctionnement.

- Procédures et Fonctions:

  NOM(LISTE DE PARAMETRES TYPES) $\longrightarrow$ *type resultats*

  S

  Une fonction comporte pour tous ces chemins d'exécution une occurrence de l'instruction **retourner** .

  Les paramètres sont passés par valeur, c'est-à-dire que la procédure reçoit la valeur de ses paramètres. Nous préciserons plus loin, lors de la présentation des structures de données, les valeurs associés à chaque type.

1.2.8 Instructions d'échappement

Les instructions d'**échappements** permettent la sortie anticipée de structures de boucles.

- **retourner** : cette instruction s'utilise à l'intérieur d'une fonction pour renvoyer un résultat. Elle est donc suivie de l'expression du résultat. Dans l'exemple, le tri se fait sur place dans le tableau reçu en paramètre, par des affectation à ses éléments. Il n'y a pas de résultat à retourner.
- **continuer** : cette instruction s'utilise à l'intérieur d'une structure répétitive (ou boucle) pour se brancher sur le test de continuation.
- **sortir**: cette instruction s'utilise à l'intérieur d'une structure de contrôle pour en sortir. (Branchement sur l'instruction suivant la structure de contrôle).

Durée : 1H30

EXEMPLE. A = [5, 2, 4, 6, 1, 3] à faire tourner.

1.3 Récursivité

1.3.1 Conception récursive

Il existe diverses façons de concevoir des algorithmes. L'exemple du tri par insertion procède de façon *incrémentale* : la taille de la partie du tableau triée augmente effectivement à chaque itération. Une autre approche très utile est la récursivité. Il s'agit de résoudre un problème de taille n en le divisant en plusieurs problèmes *similaires* mais de tailles inférieures. Dans cette approche, la fonction résolvant le problème s'appelle elle-même pour résoudre les problèmes de tailles inférieures. La procédure de conception générale est la suivante.

- Diviser le problème en plusieurs sous-problèmes similaires.
- Résoudre les sous-problèmes.
- Combiner les solutions des sous-problèmes pour obtenir la solution du problème initial.

Quand on parvient à un sous-problème ne pouvant pas être divisé, il faut alors le résoudre directement sans appel récursif. On parle du **plancher** de la récursivité.

DÉFINITION. Une procédure (ou une fonction) est dite récursive si elle fait appel à elle-même.

1.3.2 Exemples

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

1. Version itérative

   FACTORIELLE-ITERATIVE(n)
   r ←1
   pour i ←n **à** 1 **par pas de** -1
   r ←r * i
   retourner r

2. Version récursive

   FACTORIELLE-RECURSIVE(n)
   si n = 0
   alors
   % Plancher de la récursivité
   retourner 1
   sinon
   % Combinaison du résultat du sous-problème
   % pour obtenir le résultat du problème initial
   retourner n * factorielle-récursive(n-1)

1.3.3 Fonctionnement

Prenons par exemple n=3 et regardons quelle est la suite d'appels obtenue.

Appels	Résultat	Environnement
factorielle-récursive(3)	n * factorielle-récursive(n-1)	n = 3
factorielle-récursive(2)	n * factorielle-récursive(n-1)	n = 2
factorielle-récursive(1)	n * factorielle-récursive(n-1)	n = 1
factorielle-récursive(0)	1	

Observons l'expression obtenue et l'ordre dans lequel elle est calculée.

- 3 * factorielle-récursive(2)
- 3 * (2 * factorielle-récursive(1))
- 3 * (2 * (1 * factorielle-récursive(0)))
- 3 * (2 * (1 * 1))
- 3 * (2 * 1)
- 3 * 2
- 6

Le calcul nécessite 4 appels récursifs à la fonction. On constate que lors des trois premiers appels, l'expression calculant le résultat ne peut être évaluée qu'au retour de l'appel récursif. D'où la nécessité de conserver toutes les valeurs prises par n au sein des différents appels afin d'être en mesure de calculer les multiplications au retour des appels récursifs. La struture de donnée utilisée pour conserver ces valeurs est une pile (qui sera étudiée plus loin dans ce cours). Ceci est un phénomène général de la récursivité : l'espace mémoire croît avec le nombre d'appels imbriqués.

Comparons ce mécanisme avec celui mis en oeuvre pour exécuter la fonction itérative FACTORIELLE-ITERATIVE(n).

- r = 1, i = 3, n=3

- r = 3, i = 2, n=3
- r = 6, i = 1, n=3
- r = 6, i = 0, n=3

Dans cet algorithme, le calcul du résultat s'effectue progressivement à chaque tour de boucle. Il est inutile de conserver les valeurs précédentes. Seules les variables r et i sont donc nécessaires pour effectuer le calcul du résultat.

Chapitre 2

Analyse des algorithmes

Analyser un algorithme revient à prévoir les ressources nécessaires à son exécution. Les ressources pertinentes peuvent être diverses : la place mémoire, la largeur de bande d'une communication, mais le plus souvent, on s'intéresse au temps de calcul.

La complexité d'un algorithme est le temps et l'espace mémoire nécessaires à son exécution. L'analyse de la complexité permet de comparer les algorithmes indépendamment des langages de programmation et des machines sur lesquelles ils doivent s'exécuter. On pourra dire qu'un algorithme est meilleur qu'un autre pour des données de grande taille, ou bien qu'il est optimal pour résoudre un type de problème.

2.1 Modèle de machine

Pour analyser la complexité des algorithmes, on doit expliciter le modèle de machine utilisé pour l'exécution.

Un ordinateur est une machine disposant

1. d'une unité de calcul pour exécuter les opérations arithmétiques et logiques.
2. d'une mémoire permettant le stockage des instructions et des données.

et on suppose les propriétés suivantes

- on a accès à un élément dans la mémoire en un temps fixe,
- l'unité de calcul ne peut traiter qu'une opération à la fois,
- le coût de transfert des informations est négligeable devant le coût des opérations.

Il faut alors quantifier les grandeurs physiques que sont le temps d'exécution et la place mémoire.

Pour un algorithme donné $\mathcal{A}$ implémenté par un programme $\mathcal{P}$ et s'exécutant sur une machine $\mathcal{M}$ à partir de la donnée d, on exprime

- **la complexité en temps** en comptant le nombre d'opérations effectuées par le programme $\mathcal{P}$, et en utilisant le nombre de cycles horloge nécessités par chaque opération sur la machine $\mathcal{M}$.
- **la complexité en espace** en comptant le nombre de données d du programme, le nombre d'informations supplémentaires pour manipuler ces données, et en utilisant le nombre de mots mémoire nécessaires pour stocker chacune d'elles dans la mémoire de $\mathcal{M}$

2.2 Mesure de la complexité

Le temps d'exécution d'un algorithme dépend de la **taille de son entrée**. Pour de nombreux problèmes, comme le tri, la mesure est le *nombre d'éléments constituant l'entrée*, par exemple la longueur du tableau pour le tri. Pour chaque algorithme analysé, la mesure utilisée pour la taille de l'entrée doit être précisée.

Parfois on souhaite analyser un algorithme selon certaines opérations jugées fondamentales. Par exemple, pour le tri par insertion, on compte les comparaisons entre éléments et décalages d'éléments. Pour une multiplication de deux matrices, on compte les opérations entre les nombres (additions et multiplications). On peut alors chercher des algorithmes optimaux selon le nombre de ces opérations.

2.2.1 Mesure du coût d'un algorithme

On va donc associer un coût à chacune des opérations de l'algorithme, ce coût pouvant représenter son temps d'exécution, l'espace mémoire nécessaire supplémentaire ou son nombre d'opérations fondamentales.

Pour calculer le coût d'un algorithme, on utilise sa description en langage algorithmique.

- Dans une structure de contrôle séquentielle, on ajoute les coûts des diverses instructions.
- Dans une structure de contrôle conditionnelle, on exhibe un majorant valable pour toutes les branches possibles de la structure.

EXEMPLE. Soit l'instruction

si C

alors I_1

sinon I_2

Si on note $c(x)$ le coût de l'entité x, on aura

$$c(\textbf{si } C \textbf{ alors } I_1 \textbf{ sinon } I_2) \leq c(C) + max\{c(I_1), c(I_2)\}$$

– dans une structure de contrôle itérative, on aura

$$\sum_i c(b_i)$$

où i compte les itérations, et où $c(b_i)$ désigne le coût de l'exécution numéro i du corps de la boucle. Si le nombre de tours n'est pas connu explicitement, on doit alors majorer cette grandeur.

2.2.2 Exemple du tri par insertion

Considérons toutes les opérations de l'algorithme et associons-leur un coût correspondant à leur temps d'exécution. Chaque ligne i prendra alors le temps c_i où c_i est une constante. Notons n la longueur du tableau. Pour chaque $j = 2, \ldots n$, on appelle t_j le nombre de fois que le test de la boucle **tant que** à la ligne 4 est exécuté pour cette valeur de j.

Coût	Nombre fois	TRI-INSERTION(A)
c_1	n	1 **pour** $j \longleftarrow 2$ **à** $longueur[A]$
c_2	$n-1$	2 **faire** $cle \longleftarrow A[j]$
c_3	$n-1$	3 $i \longleftarrow j-1$

c_4	$\sum_{j=2}^{n} t_j$	4	**tant que** $i > 0$ **et** $A[i] > cle$
c_5	$\sum_{j=2}^{n}(t_j - 1)$	5	**faire** $A[i+1] \longleftarrow A[i]$
c_6	$\sum_{j=2}^{n}(t_j - 1)$	6	$i \longleftarrow i - 1$
c_7	$n - 1$	7	$A[i+1] \longleftarrow cle$

On obtient la formule suivante pour la complexité de l'algorithme.

$$\mathcal{C}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1)$$

$$+c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$$

Cette formule peut être exploitée en la spécialisant selon certains cas dépendant de la nature de l'entrée. Par exemple, si le tableau est déjà trié, on effectue le test de la ligne 5 qu'une seule fois par élément, donc $t_j = 1$ pour $j = 2, \ldots, n$ et le temps d'exécution devient une fonction linéaire de n.

$$\mathcal{C}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n + (c_2 + c_3 + c_4 + c_7)$$

Généralement, la complexité est etudiée dans le meilleur cas, le pire des cas, ou en moyenne. De plus, ce sont les ordres de grandeur des formules (quand la taille des données est très grande) qui nous renseignent le mieux.

Si le tableau est trié en ordre inverse, il s'agit du cas le pire. À ce moment-là, $t_j = j$ pour $j = 2, \ldots, n$. En utilisant les identités suivantes

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1, \quad \sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2} \tag{2.1}$$

on obtient la fonction quadratique de n suivante (les coûts sont considérés comme des constantes).

$$\mathcal{C}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right)$$

$$+c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + c_7\right)n$$

$$-(c_2 + c_3 + c_4 + c_7)$$

2.2.3 Fonctions et procédures récursives

Dans le cas de fonctions récursives, on est confrontés à la résolution d'équations récurrentes. Le coût $\mathcal{C}(n)$ d'un algorithme récursif pour une donnée de taille n s'écrit selon la méthode récursive en fonction de $f(k)$ pour $k < n$.

Exemple 1

$n!$

```
    FACTORIELLE(n)
c1  si n = 0
    alors
c2      retourner 1
    sinon
c3      retourner n * factorielle(n-1)
```

Calculons le coût de l'algorithme en fonction de n.

$$\begin{cases} \mathcal{C}(0) = c_1 + c_2 \\ \mathcal{C}(n) = c_1 + f(n-1) + c_3, \ si \ n \geq 1 \end{cases}$$

On a immédiatement

$$\mathcal{C}(n) = (n+1)c_1 + c_2 + nc_3$$

Complexité en temps. Considérons que la complexité en temps que nous noterons $T(n)$ ne dépend que du nombre de multiplications effectuées. Dans ce cas, on pose

$$c_1 = 0$$
$$c_2 = 0$$
$$c_3 = 1$$

Ce qui donne

$$T(n) = n$$

L'algorithme est bien linéaire en temps d'exécution. Un calcul similaire sur l'algorihme itératif de la factorielle donnerait exactement le même résultat.

Complexité en espace. En ce qui concerne la complexité en espace mémoire, que nous noterons $M(n)$, elle est donnée par la formule

$$M(n) = m + \mathcal{C}(n)$$

où m représente l'espace mémoire nécessaire pour stocker les données d'entrée, et les coûts de la fonction f sont associés à la place mémoire supplémentaire nécessaire pour l'exécution de chaque instruction. Nous avons donc les coûts

suivants, en nombre d'unités mémoire pour stocker un entier.

$$c_1 = 0$$
$$c_2 = 0$$
$$c_3 = 1$$
$$m = 1$$

La complexité en espace de la fonction est alors donnée par la formule

$$M(n) = n + 1$$

Remarque. Nous avons supposé dans ces calculs que chaque entier est stocké dans un espace mémoire de même taille. Alors, les opérations sur ces entiers (stockage, lecture et multiplication) sont d'un coût constant. Cette supposition est très réductrice en pratique puisque dans un espace mémoire donné, on ne peut coder qu'un nombre fini d'entiers. Une borne supérieure peut alors être exhibée pour la complexité. Si l'on considère, par contre, que la taille des entiers dépend de leur valeur (par exemple, $log_2 n$ bits pour la valeur n), les opérations associées aux entiers ne sont plus d'un coût constant mais dépendent de la valeur de l'entier. Ces coûts entrent en jeu dans le calcul de la complexité qui n'est alors plus du tout la même.

Effectuons, à titre de comparaison le même calcul sur l'algorihme itératif (avec les mêmes suppositions). La boucle **pour** comporte plusieurs opérations qu'il est nécessaire de dissocier. Nous associons le coût c_2 à l'opération d'initialisation de la boucle (qui ne s'effectue qu'une seule fois), le coût c_3 au test de la boucle (qui s'effectue $n + 1$ fois) et c_4 à l'opération d'incrémentation (qui s'effectue n fois).

FACTORIELLE-ITERATIVE(n)

c_1 r ←1

c_2 **pour** i ←n **à** 1 **par pas de** -1% Initialisation

c_3 % Test

c_4 % Incrémentation

c_5 r ←r * i

c_6 **retourner** r

Dans le cas de cet algorithme nous avons

$$M(n) = m + c_1 + c_2 + (n+1)c_3 + nc_4 + nc_5 + c_6$$

Afin de calculer la complexité en espace mémoire, on peut utiliser les valeurs de coûts suivantes.

c_1=1 Place de r

c_2=1 Place de i

c_3=0

c_4=0

c_5=0

c_6=0

m=1 Place de n

Alors, on obtient

$$M(n) = 1 + 1 + 1 = 3$$

Ce qui confirme que la fonction factorielle récursive a une complexité en espace mémoire linéaire et égale à la valeur de son entrée, alors que la fonction itérative a une complexité constante.

Exemple 2

Soit un problème de taille n que l'on subdivise en a sous-problèmes de tailles $\frac{n}{b}$. Pour simplifier, supposons qu'un sous-problème de taille 1 a une complexité unitaire, et que la combinaison d'assemblage des solutions des sous-problèmes pour résoudre le problème de taille n est $d(n)$.

On peut alors écrire :

$$C(1) = 1$$
$$C(n) = aC(\tfrac{n}{b}) + d(n)$$

En substituant de proche en proche, on aboutit à la relation :

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

On suppose qu'il existe k tel que $\frac{n}{b^k} = 1$ (et donc $k = log_b n$). k représente le nombre d'itérations nécessaire pour atteindre le cas unitaire à partir de n. Remplaçons i par k et $C(1)$ par 1. On obtient :

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

car $a^k = a^{log_b n} = n^{log_b a}$.

Durée : 1H30

2.2.4 Configuration des données et complexité

EXEMPLE. Recherche séquentielle d'un élément dans un tableau.

On se donne un vecteur v à n éléments et on veut savoir si un élément d donné appartient à v ou pas.

RECHERCHE-ELEMENT-TABLEAU(v)

$i \leftarrow 1$

tant que $(i \leq n)$ **et** $(v[i] \neq d)$

faire $i \leftarrow i + 1$

si $i > n$

alors $i \leftarrow 0$

retourner i

Cet algorithme effectue

- i comparaisons si i est l'indice dans v de la première occurrence de d dans v,
- n comparaisons si d n'est pas dans v.

La complexité dépend de la taille n des données, dépend aussi des différents configurations de données possibles : **pour n fixé, elle varie de 1 à n.**

DÉFINITIONS Soit $\mathcal{A}$ un algorithme utilisant un ensemble $\mathcal{D}_n$ de donnees de taille n. On note $\mathcal{C}_A(d)$ la complexité de l'algorithme $\mathcal{A}$ agissant sur la donnée d.

- La **complexité dans le meilleur des cas** est

$$Min_A(n) = min\{\mathcal{C}_A(d), d \in \mathcal{D}_n\}$$

- La complexité **dans le pire des cas** est

$$Max_A(n) = max\{\mathcal{C}_A(d), d \in \mathcal{D}_n\}$$

- La complexité **en moyenne** est

$$Moy_A(n) = \sum_{d \in \mathcal{D}_n} p(d)\mathcal{C}_A(d)$$

où $p(d)$ est la probabilité d'avoir la donnée d en entrée de l'algorithme.

REMARQUES

- On a la relation

$$Min_A(n) \leq Moy_A(n) \leq Max_A(n)$$

- Si toutes les données sont équiprobables, on aura

$$Moy_A(n) = \frac{1}{|\mathcal{D}_n|} \sum_{d \in \mathcal{D}_n} \mathcal{C}_A(d)$$

  ($|\mathcal{D}_n|$ représente le nombre de données de taille n).

- Sinon on partitionne $\mathcal{D}_n$ en sous-ensembles $\mathcal{D}_n^i$, dont les données impliquent la même complexité pour $\mathcal{A}$ tels que

$$\forall d \in \mathcal{D}_n^i \, , \mathcal{C}_A(d) = constante = \mathcal{C}_A(\mathcal{D}_n^i)$$

  et on aura une probabilité $p(\mathcal{D}_n^i)$ pour chaque sous-ensemble.

$$Moy_A(n) = \sum_{\mathcal{D}_n^i \subset \mathcal{D}_n} p(\mathcal{D}_n^i) \mathcal{C}_A(\mathcal{D}_n^i)$$

- La complexité dans le pire des cas est certainement la plus employée. En effet, elle représente une borne supérieure pour une entrée quelconque. Sa connaissance nous assure que nous ne nous trouverons janais dans un cas plus défavorable. De plus, pour les algorithmes de recherche dans une base de données, le pire des cas correspond à l'absence de l'élément recherché, ce qui se produit fréquemment. Enfin, il n'est pas rare que la complexité en moyenne soit de l'ordre de la complexité dans le pire des cas. Nous en verrons des exemples.

EXEMPLES

1. Produit de deux matrices de tailles n

   PRODUIT-MATRICES(A,B)
   pour i $\leftarrow$1 **à** n **par pas de** 1
   faire pour j $\leftarrow$1 **à** n **par pas de** 1
   faire $C[i,j] \leftarrow 0$
   pour k $\leftarrow$1 **à** n **par pas de** 1
   faire $C[i,i] = C[i,j] + A[i,k] \times B[k,j]$

   La complexité de cet algorithme ne dépend que de la taille des données. Si l'on considère uniquement le nombre de mulktiplications et d'additions, on obtient

   $$\mathcal{C}(n) = 2n^3$$

   En effet, la boucle la plus interne est effectuée n^3 fois et elle comporte deux opérations fondamentales.

2. Recherche séquentielle d'une valeur d dans un tableau v.

   – On a

   $$Max(n) = n$$

   et

   $$Min(n) = 1$$

   – Pour la complexité en moyenne, on doit définir des lois probabilistes sur v et sur d.

   – soit q la probabilité pour que d soit dans v. On suppose de plus que si d est dans v, toutes les places sont équiprobables.

- on note $\mathcal{D}_n^i$, $1 \leq i \leq n$, l'ensemble des données de taille n où d est en $i$ème place.
- on note $\mathcal{D}_n^0$ l'ensemble des données de taille n où d est absent.

– On a alors

$$p(\mathcal{D}_n^i) = \frac{q}{n}, i = 1, n \ (\textstyle\sum_i p(\mathcal{D}_n^i) = q)$$
$$p(\mathcal{D}_n^0) = 1 - q$$

– D'autre part,

$$\mathcal{C}_A(\mathcal{D}_n^i) = i, \ i = 1, n$$
$$\mathcal{C}_A(\mathcal{D}_n^0) = n$$

Donc

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

– Si on sait que d est dans v $(q = 1)$

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

– Si d a une chance sur deux d'être dans v $(q = 1/2)$

$$Moy_A(n) = \frac{1}{4}(3n+1)$$

On constate que dans le cas de cet algorithme, la complexité dans le cas le pire est linéaire avec un coefficient égal à un. Les complexités en moyenne sont linéaires avec des coefficients variant selon les probabilités, mais étant toujours inférieurs à 1. Le coefficient du cas $q = 1/2$ s'approchant déjà bien de 1. Les ordres de grandeur de ces complexités sont équivalents. (voir la section suivante).

2.3 Comparaison d'algorithmes

On a déterminé la complexité d'un algorithme comme une **fonction de la taille des données**. Il faut maintenant étudier **la croissance** de cette fonction quand la taille des données croît. Ceci est primordial pour comparer deux algorithmes quand on veut les appliquer à des problèmes **réels** qui sont quasiment toujours de grandes tailles.

2.3.1 Grandeur des fonctions

On analyse la complexité d'un algorithme $\mathcal{A}$ (souvent au sens de Max_A) en déterminant **l'ordre de grandeur asymptotique** de la fonction par rapport à une échelle de comparaison constituée de fonctions usuelles qui sont des *fonctions de référence* pour la complexité des algorithmes.

EXEMPLES. Voir la figure 2 de la feuille, pour une échelle de comparaison *classique* de fonctions qui vont en croissant strictement.

Pour comparer les ordres de grandeur asymptotiques des fonctions, on utilise les notations mathématiques O et θ.

2.3.2 Notation asymptotique

Notation θ

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

$\theta(g(n)) = \{f(n)$: il existe des constantes strictement positives c_1 et c_2 et un n_0 telles que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ pour tout $n > n_0\}$

Bien que $\theta(g(n))$ soit un ensemble, on note $f(n) = \theta(g(n))$ pour indiquer que $f(n)$ est un membre de $\theta(g(n))$.

Faire un dessin. Remarquer que f et g sont positifs.

On dit alors que $g(n)$ est une **borne approchée asymptotique** de $f(n)$ ou que f et g ont **même ordre de grandeur asymptotique**.

EXEMPLE. Montrons $\frac{1}{2}n^2 - 3n = \theta(n^2)$. Il suffit de trouver les constantes c_1 et c_2 ainsi que n_0 telles que

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

pour tout $n \geq n_0$. Divisons par n^2

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Pour respecter l'inégalité de gauche, il faut prendre une valeur de n au moins égale à 7. La constante c_1 peut alors prendre la valeur $\frac{1}{14}$. A droite, on peut prendre $c_2 = \frac{1}{2}$ à partir de $n_0 \geq 1$. En résumé, $n_0 = 7$, $c_1 = \frac{1}{14}$ et $c_2 = \frac{1}{2}$.

Remarque. On peut montrer que pour tout polynôme

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

où les a_i sont des constantes et $a_d > 0$, on a

$$p(n) = \theta(n^d)$$

Une constante est alors en $\theta(n^0)$, c'est-à-dire en $\theta(1)$.

Notation O

Elle définit une **borne supérieure asymptotique**. Pour une fonction $g(n)$ donnée, on note $O(g(n))$ l'ensemble de fonctions :

$$O(g(n)) = \{f(n) : \text{il existe des constantes strictement positives } c \text{ et } n_0 \text{ telles que } 0 \leq f(n) \leq cg(n) \text{ pour tout } n > n_0\}$$

Bien que $\theta(g(n))$ soit un ensemble, on note $f(n) = O(g(n))$ pour indiquer que $f(n)$ est un membre de $O(g(n))$. On dit aussi que f **est dominée asymptotiquement par** g.

REMARQUES

1. $\theta(g(n)) \subset O(g(n))$ et donc $f(n) = \theta(g(n)) \Rightarrow f(n) = O(g(n))$.
2. $2n = \theta(n)$ et donc $2n = O(n)$. Mais on a aussi $2n = O(n^2)$. On voit ici l'intérêt de θ par rapport à O.
3. Les définitions supposent l'existence des constantes finies c, d, n_0 (seuil) mais rien n'est dit sur leur valeur.

   Pour des fonctions dont les ordres de grandeur sont différents, il est souhaitable d'avoir une estimation de ces constantes pour mieux cerner la **portée pratique** de la comparaison entre deux fonctions de complexité.

   Si $f(n) = 2n,\ g(n) = n^2$ alors $f(n) < g(n)$ pour $n > 2$.
   Si $f(n) = 10000n,\ g(n) = n^2$ alors $f(n) < g(n)$ pour $n > 10^4$.
   Si $f(n) = 2n\log_2(n),\ g(n) = n\sqrt{n}$, alors $f(n) < g(n)$ pour $n > 256$.

4. Si les fonctions ont même ordre de grandeur, il faut pour les comparer, évaluer finement les constantes et souvent évaluer des termes d'ordre inférieur dans les comportements asymptotiques. (Difficile).

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

> Il est toujours d'actualité de rechercher des algorithmes efficaces même si les progrès technologiques accroissent les performances du matériel.

On dispose pour résoudre un problème donné de 7 algorithmes dont les complexités dans le cas le pire ont pour ordre de grandeur 1 (fonction constante qui ne dépend pas de la taille des données), $log_2 n$, n, $nlog_2 n$, n^2, n^3, 2^n.

On suppose que l'ordinateur peut faire 10^6 opérations par seconde.

- Tableau A : donne une estimation du temps d'exécution pour chaque algorithme et pour différentes tailles.
- Tableau B : estimation de la taille maximale du problème que l'on peut résoudre en un temps donné.
- Tableau C : augmentation de la complexité en fonction de l'augmentation de la taille des données.

Il apparaît clairement que certains algorithmes sont peu utilisables ou même inutilisables pour résoudre des problèmes sur un ordinateur. Ceux qui sont utilisables sont ceux qui s'exécutent en temps

1. Constant.

2. Logarithmique : recherche dichotomique, nombre de comparaisons dans le pire des cas.
3. Linéaire : recherche séquentielle d'un élément dans une liste non triée, nombre de comparaisons dans le pire des cas.
4. $nlog(n)$: tri par fusions successives, nombre de comparaisons dans le pire des cas.
5. Polynômial ie $\theta(n^k)$, $k > 0$
   - $k = 2$ nombre de déplacements dans le pire des cas pour les 2 tris précédents.
   - $k = 3$ nombre d'opérations pour multiplier 2 matrices.

Au delà de $k = 3$ (et encore), et pour les algorithmes **exponentiels** ie en $\theta(2^n)$, on ne parle plus d'algorithmes utilisables (jeu d'échecs) : on envisage alors des méthodes approchées appelées **heuristiques** qui sont des *méthodes produisant la plupart du temps un résultat acceptable non nécessairement optimal, dans un temps raisonnable.*

Durée : 1H30

Chapitre 3

Structures de données

3.1 Notion de type abstrait

Un algorithme utilise des **structures de données** pour représenter et manipuler les données du problème qu'il solutionne. Jusqu'à présent, nous avons utilisé des données numériques ainsi que des tableaux. Dans ce chapitre, nous allons voir comment spécifier de nouveaux types qui pourront être utilisés dans un algorithme.

Un algorithme doit être aussi indépendant que possible d'une implémentation particulière. Les données doivent être considérées de manière abstraite, détachées au maximum de leur représentation interne. Toutefois, il ne faut évidemment pas perdre de vue qu'elles sont destinées à être implémentées et donc s'assurer de cette possibilité.

Un **type abstrait** est une spécification de données décrivant l'ensemble des opérations associées à ces données et l'ensemble des propriétés de ces opérations. Cette spécification ne précise pas la représentation interne. La

conception de l'algorithme se fait en utilisant les opérations des types abstraits. De ce fait, les algorithmes sont indépendants de la représentation interne des données. Nous pouvons distinguer deux sortes de types abstraits :

- Les types abstraits de base qui correspondent aux types disponibles dans la plupart des langages de programmation. Ces types abstraits sont utilisés pour construire d'autres types abstraits.
- Les types abstraits construits au moyens de types abstraits de base ou bien d'autres types abstraits construits prédéfinis.

Concernant les types abstraits de base, il est possible de les définir de la même façon que les types abstraits construits, c'est-à-dire, sans préciser leur représentation interne.

Exemple. On peut définir le type réel par les opérations et leurs propriétés, sans connaître leur représentation interne (mantisse, exposant).

3.2 Définition d'un type abstrait

La définition d'un type de donnée consiste à en donner une **spécification**. Un type abstrait est la donnée

- de sa **signature** décrivant la syntaxe du type avec le nom des opérations valides ainsi que le type de leurs arguments,
- des propriétés des opérations du type qui seront données sous forme d'un ensemble d'**axiomes** (formules logiques). On introduit alors une **sémantique** (une signification) aux objets de la signature.

3.2.1 Signature

La signature d'un type abstrait est la donnée

- d'un ensemble de **noms** pour un certain nombre d'ensembles de valeurs; ces noms sont appelés des **sortes** (cette notion correspond à la notion de types pour des langages de programmation).
- d'un ensemble de **noms d'opérations** et de leur **signature**. La signature d'une opération est la précision de son domaine de définition ainsi que l'ensemble auquel appartient le résultat.

EXEMPLE 1. Signature du type abstrait booléen

Sorte booléen

Opérations

vrai : $\rightarrow$ booléen

faux : $\rightarrow$ booléen

$\neg$ - : booléen $\rightarrow$ booléen

- $\wedge$ -: booléen $\times$ booléen $\rightarrow$ booléen

- $\vee$ -: booléen $\times$ booléen $\rightarrow$ booléen

Les fonctions *vrai* et $faux$ sont sans arguments. Ce sont des constantes. On donne en même temps que le nom de la fonction la place des arguments quand la notation n'est pas fonctionnelle.

EXEMPLE 2. On considère le type abstrait **séquence d'éléments réels numérotés par des entiers**.

Il serait peu pratique de donner un détail la signature complète des sortes entier et réel.

On s'autorise alors la réutilisation de types abstraits prédéfinis (types abstraits de base), ou définis auparavant.

Sorte séquence-réelle

Utilise entier, réel

Opérations

séquence-réelle	: *entier* $\rightarrow$ *séquence-réelle*
ième	: *séquence-réelle* $\times$ *entier* $\rightarrow$ *réel*
changer-ième	: *séquence-réelle* $\times$ *entier* $\times$ *réel* $\rightarrow$ *séquence-réelle*
taille	: *séquence-réelle* $\rightarrow$ *entier*

La signature est l'union des signatures. Cet exemple introduit la notion de hiérarchie entre les types. On introduit une classification parmi les sortes et les opérations d'un type abstrait

- sorte définie : sorte de nom nouveau (exemple séquence).
- sorte prédéfinie : sorte de nom utilisé (exemple entier et réel).
- opération interne : opération donnant un résultat d'une sorte définie (exemple changer-ième).
- observateur : opération ayant au moins un argument de sorte définie et donnant un résultat de sorte prédéfinie (exemple ième, taille).

3.2.2 Propriétés d'un type abstrait

On donne une sémantique aux sortes et aux opérations de la signature au moyen d'axiomes agissant sur les variables. On obtient une définition algébrique ou axiomatique du type abstrait.

Soient s, i, r des variables de sortes séquence-réelle, entier et réel. On considère les axiomes suivants :

- $1 \leq i \leq taille(s) \Rightarrow ième(changer\text{-}ième(s, i, r), i) = r$
- $1 \leq i \leq taille(s)$ et $1 \leq j \leq taille(s)$ et $i \neq j \Rightarrow ième(changer\text{-}ième(s, i, r), j) = ième(s, j)$
- $taille(changer\text{-}ième(s, i, r)) = taille(s)$
- $taille(séquence\text{-}réelle(l)) = l$

Les deux premiers axiomes expriment que la fonction *changer-ième* ne change que le ième élément de la séquence.

Remarques

- **Consistance**: les axiomes ne sont pas contradictoires.
- **Complétude**: les axiomes sont suffisants pour décrire l'ensemble des propriétés du type abstrait. En fait, seule la **complétude suffisante** est utilisée pour les types abstraits :

  > *On doit pouvoir déduire une valeur pour tous les observateurs, sur tout objet d'une sorte définie appartenant au domaine de définition de chaque observateur.*

3.3 Types abstraits de base

Dans cette section, nous allons rappeler ou bien définir au moyen du formalisme des types abstraits, les types de base que l'on trouve communément dans les langages de programmation. Ces définitions fourniront une plateforme utile pour l'implémentation.

3.3.1 Types primitifs

Les types primitifs correspondent à des quantités numériques codées sur des cases mémoire de taille fixe. On y trouve les entiers, réels, caractères, etc. Les opérations admises par les nombres sont les opérations arithmétiques habituelles (+,*,/,-) ainsi que des fonctions mathématiques (log, sin, cos, etc.). Nous ne proposerons pas de définition abstraite pour ces types, bien que cela soit tout à fait possible, mais nous les utiliserons pour spécifier des types abstraits non primitifs.

3.3.2 Type tableau

Un tableau est une séquence finie d'éléments de même type permettant un accés direct à chaque élément en fonction de son indice. Cet accés permet la lecture de la valeur de l'élément ou bien sa modification.

Nous pouvons utiliser une définition analogue à celle du type abstrait séquence-réelle pour définir un tableau. Toutefois, pour être totalement conforme au type tableau généralement fourni par les langages de programmation, la définition doit être générique, c'est-à-dire permettre de varier les types d'éléments. Nous introduisons donc un nouveau type *séquence* dit générique,

admettant le type de ses éléments en paramètre.

Sorte séquence

Paramètre élément

Utilise entier

Opérations

seq	: $entier \times element \rightarrow séquence$
ième	: $séquence \times entier \rightarrow élément$
changer-ième	: $séquence \times entier \times élément \rightarrow séquence$
taille	: $séquence \rightarrow entier$

Si l'on conserve les mêmes axiomes que ceux du type *séquence-réelle*, le type abstrait *séquence* correspond à peu près au type tableau du langage C, à un décalage près de la numérotation des indices (on commence à 1 au lieu de 0 en C).

Pour simplifier la notation, puisque le type tableau est largement utilisé, on pourra écrire :

- $t \longrightarrow$ *tableau*[réel](n, e) pour construire un nouveau tableau t de n éléments de type réel initialisés à e.
- $t[i]$ pour *ième(t,i)*
- $t[i] \leftarrow e$ pour *changer-ième(t,i,e)*
- *t.taille* pour *taille(t)*

On remarquera la notation *tableau*[réel] permettant de spécifier le type correspondant au paramètre *élément* au moment de la construction d'un ta-

bleau.

3.3.3 Type structure

Le type structure permet d'assembler des valeurs de types différents. On dit qu'une structure est composée de champs qui ont chacun un nom et un type. La valeur de chaque champ peut être lue ou modifiée. Soient E_1, E_2, ..., E_n n types abstraits, le type structure peut se définir de la façon suivante.

Sorte structure

Paramètres E_1, E_2, ..., E_n

Opérations

structure	$: Id^n \times E_1 \times E_2 ... \times E_n \rightarrow structure$
$e1$	$: structure \rightarrow E_1$
$e2$	$: structure \rightarrow E_2$
...	
en	$: structure \rightarrow E_n$
changer-e1	$: E_1 \times structure \rightarrow structure$
changer-e2	$: E_2 \times structure \rightarrow structure$
...	
changer-en	$: E_n \times structure \rightarrow structure$

La sorte *structure* admet en paramètres les types de ses champs. La fonction de construction admet les noms des champs en paramètres (Id est l'ensemble des identificateurs possibles) et les valeurs d'initialisation. Les opérations sont limitées aux accés en lecture ou en écriture aux champs.

De même que pour les tableaux, on pourra utiliser une notation plus simple que la notation fonctionnelle et plus proche de celle des langages de programmation.

- s = structure[E_1, E_2, ..., E_n]($id_1 = v_1, id_2 = v_2, \ldots, id_n = v_n$) pour construire une structure s composée des champs de noms id_1, id_2, ..., id_n admettant respectivement pour types E_1, E_2, ..., E_n et intialisés respectivement à v_1, v_2, ..., v_n.
- $s.id_i$ pour $e_i(s)$, $1 \leq i \leq n$
- $s.id_i = e$ pour *changer-ei*(e, s), $1 \leq i \leq n$

Exercice : écrire les axiomes du type structure.

Durée : 1H30

3.3.4 Exemple : matrice rectangulaire

Signature

Sorte matrice

Paramètre élément

Utilise entier

Opérations

$matrice$	$: entier \times entier \times element \rightarrow matrice$
$nblignes$	$: matrice \rightarrow entier$
$nbcolonnes$	$: matrice \rightarrow entier$
$changer$	$: matrice \times entier \times entier \times élément \rightarrow matrice$
$ijième$	$: matrice \times entier \times entier \rightarrow élément$

Axiomes

- $nblignes(matrice(n, c, v)) = n$.
- $nbcolonnes(matrice(n, c, v)) = c$.
- $nblignes(changer(m, i, j, e)) = nblignes(m)$.
- $nbcolonnes(changer(m, i, j, e)) = nbcolonnes(m)$.
- $ijème(changer(m, i, j, e), i, j) = e$, si $1 \leq i \leq nblignes(m)$ et $1 \leq j \leq nbcolonnes(m)$.
- $ijème(changer(m, i, j, e), k, l) = ijème(m, k, l)$, si $1 \leq i \leq nblignes(m)$, $1 \leq j \leq nbcolonnes(m)$, $1 \leq k \leq nblignes(m)$ et $1 \leq l \leq nbcolonnes(m)$, et $k \neq i$ $l \neq j$.
- $ijème(matrice(n, c, v)) = v$.

Implémentation au moyen de tableau

MATRICE[e](n,c,v)

retourner tableau[tableau](n, tableau[e](c,v))

NBLIGNES(m)

retourner m.taille

NBCOLONNES(m)

retourner m[1].taille

CHANGER(m,i,j,e)

m[i][j] = e

IJIEME(m,i,j)

retourner m[i][j]

3.4 Structures de données dynamiques

Un algorithme est souvent amené à traiter des ensembles de données. Or, il est parfois impossible de prédire le nombre d'éléments survenant au moment de l'éxecution. Les structures de données dynamiques permettent de représenter des ensembles ayant un nombre quelconque de données, inconnu au moment de l'écriture de l'algorithme. Le terme *dynamique* fait référence à l'éxecution, alors que le terme *statique* fait référence à l'écriture de l'algorithme ou du programme.

Nous allons voir dans cette section deux sortes de structures dynamiques :

– Les structures contigues : tableaux infinis

– Les structures chaînées : chaînes simples ou doubles de noeuds.

Nous nous bornons dans cette section à l'étude des structures linéaires.

Avant d'introduire les types abstraits de ces structures, il est nécessaire de clarifier la notion d'adresse et la façon dont nous allons l'utiliser.

3.4.1 Clarifications sur la notion d'adresse et sur l'affectation

Références (présentation formelle)

Un algorithme admet un ensemble d'identificateurs auxquels sont associés des types et des valeurs. Pour associer une valeur à un identificateur, le langage C utilise la notion d'adresse en mémoire et propose un type de base pointeur. D'autres langages (lisp, java) utilisent les références. C'est le modèle sur lequel nous allons nous baser.

Soit Id l'ensemble des identificateurs d'un algorithme A, soit S l'ensemble des types de ces identificateurs. Soit M l'ensemble des cases mémoire stockant les valeurs associées aux identificateurs à un instant donné t_0. On peut définir M comme suit

$$M = \{(i, d), i \in N, \exists T \in S | d \in T\}$$

où chaque couple (i, d) représente une case mémoire numérotée i et contenant la valeur d. Considérons une fonction de valuation à un l'instant t_0, associant à chaque identificateur une case mémoire :

$$v : Id \rightarrow M$$

Nous allons préciser l'affectation au moyen de cette définition. Soient

id_1 et id_2 deux identificateurs de même type. Soit v la valuation courante, telle que $v(id_1) = (i_1, d_1)$ et $v(id_2) = (i_2, d_2)$ et soit M l'ensemble des cases mémoire. Considérons l'affectation suivante

$$id_1 \leftarrow id_2$$

Dans le cas où le type de id_1 et id_2 est primitif (entier, etc.), le nouvel ensemble de cases mémoire M' est défini comme suit :

$$M' = \{c \in M, c \neq (i_1, d_1)\} \cup \{(i_1, d_2)\}$$

et la nouvelle valuation

$$v'(id) = \begin{cases} v(id), \forall id \in Id, id \neq id_1 \\ (i_1, d_2), id = id_1 \end{cases}$$

Dans le cas où le type de id_1 et id_2 n'est pas primitif, l'ensemble des cases mémoire est inchangé et la valuation v est remplacée par v' telle que

$$v'(id) = \begin{cases} v(id), \forall id \in Id, id \neq id_1 \\ v(id_2), id = id_1 \end{cases}$$

Les deux identificateurs id_1 et id_2 référencent alors la même valeur.

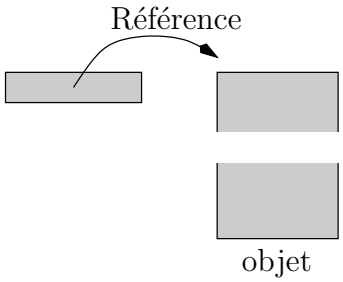
FAIRE DES DESSINS

Références (présentation intuitive)

Variable (Rappel). Une variable permet de ranger une valeur en mémoire. C'est une zone mémoire modifiable, identifiée par un nom (identificateur) et caractérisée par un type.

Variable de type primitif. Elle contient une quantité numérique (codée en base deux) dont l'interprétation dépend de son type.

Variable de type non primitif. Une variable dont le type n'est pas primitif ne contient pas directement la valeur qui lui est associée, mais l'adresse de cette valeur. On dit qu'elle est une référence.



Affectation.

Type primitif: lors de l'affectation d'une variable v_1 par une variable v_2, de types primitifs, la valeur contenue dans la variable v_1 est modifiée et reçoit la valeur contenue dans la variable v_2. Lors de l'affectation d'une variable par une valeur, son contenu reçoit aussi la valeur affectée.

Type non primitif: lors de l'affectation d'une variable v_1 par une variable v_2, de types non primitifs, l'adresse contenue dans la variable v_1 est modifiée et reçoit l'adresse contenue dans la variable v_2. Les deux variables v_1 et v_2 référencent alors la même valeur. Lors de l'affectation d'une variable par une valeur, son contenu reçoit l'adresse de la valeur.

Remarques.

- En ce qui concerne le passage de variables en paramètres ou bien en retour de fonction, le mécanisme est le même.
- On notera $NULL$ la valeur de référence ne référençant pas de valeur.

3.4.2 Tableaux infinis

Il est parfois impossible de prédire une borne supérieure pour le nombre d'éléments d'un tableau. (En particulier quand ce nombre dépend d'une intéraction extérieure). Nous proposons un type abstrait correspondant à un tableau qui s'étend en fonction des besoins de l'algorithme qui l'utilise.

> Il est à noter que la complexité des opérations de ce type n'est plus en $O(1)$, comme pour les types abstraits présentés précédemment.

Signature

Sorte tableau-infini

Paramètre élément

Utilise entier

Opérations

tableau-infini	: *element* $\rightarrow$ *tableau-infini*
changer-ième	: *tableau-infini* $\times$ *entier* $\times$ *élément* $\rightarrow$ *tableau-infini*
ième	: *tableau-infini* $\times$ *élement* $\rightarrow$ *entier*

Axiomes

- Pour $1 \leq i$ et $1 \leq j$

$$\textit{ième}(\textit{changer-ième}(t, i, e), j) = \begin{cases} e, \text{ si } i = j \\ \textit{ième}(t, j) \text{ si } i \neq j \end{cases}$$

- *ième(tableau-infini(v)) = v*

Remarque. Nous ne nous intéressons ici qu'aux dépassements provoqués par des indices trop grands.

Implémentation avec des tableaux fixes

La construction initiale d'un tableau infini s'implémente par la construction d'un tableau t de taille fixe UNITE. Ensuite, à chaque fois qu'un accès en lecture ou bien en écriture se produit à un indice i dépassant la taille du tableau, un nouveau tableau t' de taille suffisante est construit. Après avoir recopié t dans t', on peut accéder au ième élément.

Remarque. Un tableau de taille fixe peut changer d'adresse après un allongement (donc après un accés par les fonctions *changerième* et *ième*). Notre tableau infini pourrait changer d'adresse après la fonction *changerième*, puisqu'elle renvoie le tableau en résultat. Mais la fonction *ième*, quant à elle, renvoie un résultat entier. Il est donc nécessaire de fournir une implémentation qui ne change pas l'adresse du tableau infini. Pour cela, on peut utiliser une structure ayant pour champ un tableau fixe.

Allongement. L'algorithme d'extension du tableau que nous proposons ici est le suivant.

$$t.taille + k \times UNITE$$
$$\text{avec } t.taille \times (k-1)UNITE < i \leq t.taille + k \times UNITE$$

Cet algorithme est simple et robuste, mais il est inefficace. Si UNITÉ est la taille d'un élément du tableau, lors de l'ajout du i-ème élément, il faut recopier $i - 1$ éléments en plus de l'affectation du nouvel élément. Par

conséquent, l'ordre de grandeur de la complexité dans le pire des cas de l'affectation d'un élement n est en $\theta(n)$, puisqu'elle consiste à affecter n éléments dans le tableau. L'ordre de grandeur de l'initialisation d'un tableau en partant du premier élément jusqu'au dernier est en $\theta(n^2)$. Si UNITÉ est égal à la taille de k mots mémoire, cette complexité est simplement divisée par k. On verra en TD des algorithmes plus efficaces.

TABLEAU-INFINI[e]()

retourner structure[tableau[e](UNITE)](tab)

CHANGERIEME(t,i,e)

taille ←t.tableau.taille

% Si l'indice depasse la taille

si taille < i

% Construction d'un tableau suffisamment grand

% taille * (k-1) UNITE < i ≤ taille + k*UNITE

alors t' ←tableau[e](taille + k * UNITE)

    % Copie du tableau initial dans le nouveau tableau

    pour j ←1 a taille

    faire t'[j] ←t.tableau[j]

    % Affectation du nouveau tableau

    t.tableau ←t'

% Affectation du ieme element

t.tableau[i] ←e

retourner t

IEME(t,i)

taille ←t.tableau.taille

si taille < i

alors t' ←tableau[e](taille + k * UNITE)

pour j ←1 a taille

faire t'[j] ←t.tableau[j]

t.tableau ←t'

retourner t.tableau[i]

Remarques.

- On pourra discuter de la possibilité d'initialiser un tableau infini au moment de sa construction.
- En pratique, il faudrait que le rallongement se produise rarement afin que la complexité en moyenne soit proche de $O(1)$.
- Le type abstrait tableau infini ne doit être utilisé que dans le cas où les éléments doivent être contigus. Il faut remarquer que l'accés direct n'est plus en 0(1) dans tous les cas. Il est parfois préférable, en ce qui concerne la complexité d'utiliser des structures chaînées (voir section suivante) qui ne nécessitent pas la recopie.

Durée : 1H30 ??

3.4.3 Structures chaînées

Une structure chaînée est un type structure ayant au moins un champ de type structure. Dans le cas où une structure S a des champs de type S, on a une définition récursive. On dit qu'une structure est simplement chaînée si elle comporte un champ de type S (habituellement nommé *suivant*) et qu'elle est doublement chaînée si elle comporte deux champs de type S (habituellement nommés *suivant* et *précédant*.

FAIRE UN DESSIN

Une telle structure est utilisée comme "brique de base" pour chaîner des éléments entre eux. Elle est communément appelée *noeud*.

Signature du noeud simple

Un noeud simple est une structure particulière.

Sorte noeud

Paramètre élément

Utilise noeud et booléen

Opérations

noeud	: *élément* $\times$ *noeud* $\rightarrow$ *noeud*
contenu	: *noeud* $\rightarrow$ *élément*
changer-contenu	: *noeud* $\times$ *élément* $\rightarrow$ *noeud*
suivant	: *noeud* $\rightarrow$ *noeud*
changer-suivant	: *noeud* $\times$ *noeud* $\rightarrow$ *noeud*
suivant?	: *noeud* $\rightarrow$ *booléen*

Axiomes du noeud

- suivant(noeud(e,s)) = s
- contenu(noeud(e,s)) = e
- suivant(changer-suivant(n,s)) = s
- contenu(changer-contenu(n,e)) = e
- suivant(changer-contenu(n,e)) = suivant(n)
- contenu(changer-suivant(n,s)) = contenu(n)
- suivant?(n) = faux si suivant(n) = NULL
- suivant?(n) = vrai si suivant(n) $\neq$ NULL

Implémentation du noeud

Le noeud s'implémente de façon triviale au moyen d'une structure.

NOEUD[E](e,n)

s ←structure[E,noeud](contenu=e, suivant=n)

s.contenu ←e

s.suivant ←n

retourner s

CONTENU(n)

retourner n.contenu

CHANGER-CONTENU(n,e)

n.contenu ←e

retourner n

SUIVANT(n)

retourner n.suivant

CHANGER-SUIVANT(n1,n2)

n1.suivant ←n2

retourner n1

SUIVANT?(n)

retourner n.suivant = NULL

3.5 La pile

La pile est une structure de donnée très simple et très utile en informatique. Elle permet de stocker des éléments par ordre d'arrivée et de les faire sortir dans l'ordre inverse : le dernier arrivé est le premier sorti. (Penser à une pile de crêpes). Beaucoup d'auteurs utilisent le sigle anglosaxon $LIFO$ (signifiant *Last In First Out*) pour y qualifier cette structure.

3.5.1 Signature du type abstrait pile

Les opérations de base sur la pile sont :

– Tester si une pile est vide.

– Accéder à la valeur au sommet de la pile.

– Empiler un élément : ajouter un élément au sommet de la pile.

– Dépiler un élément : enlever l'élément qui se trouve au sommet de la pile.

La signature du type abstrait *pile* est donc la suivante.

Sorte pile

Paramètre élément

Utilise booléen

Opérations

pile	$:\rightarrow pile$
empiler	$: pile \times élément \rightarrow pile$
dépiler	$: pile \rightarrow pile$
sommet	$: pile \rightarrow élément$
vide?	$: pile \rightarrow booléen$

Les opérations *empiler*, *dépiler* et *pile* sont des opérations internes, alors que les opérations *sommet* et *vide?* sont des observateurs de pile.

3.5.2 Axiomes du type abstrait pile

– Les opérations *sommet* et *dépiler* ne sont définies que pour des piles non vides.

– sommet(empiler(p,e)) = e

– dépiler(empiler(p,e)) = p

- vide?(pile()) = vrai
- vide?(empiler(p,e)) = faux

3.5.3 Implémentation d'une pile avec une structure et un tableau

Nous allons représenter une pile par un couple composé d'un tableau stockant les éléments de la pile et un entier stockant l'indice dans le tableau du sommet de la pile. Un tel couple peut donc être représenté par une structure à deux champs

- pile : tableau de MAX éléments. Discuter du problème de définition de MAX et du problème de son dépassement. Un tableau infini serait ici préférable.
- sommet : un entier pointant un élément dans le tableau.

Selon cette représentation, pile vide est repérée par le fait que son sommet est nul.

PILE[e]()

p =structure[tableau[e](MAX),entier](pile,sommet)

p.sommet = 0

retourner p

EMPILER(p,e)

% La specification formelle du type n'est pas respectee ici

si p = MAX

alors erreur

sinon p.sommet = p.sommet + 1

p.pile[p.sommet] = e

retourner p

DEPILER(p)

si vide?(p)

alors erreur

sinon p.sommet = p.sommet - 1

retourner p

SOMMET(p)

si vide?(p)

alors erreur

sinon retourner p.pile[p.sommet]

VIDE?(p)

retourner p.sommet=0

3.5.4 Exemple : évaluation d'une expression postfixée

On représente un expression postfixée par un tableau d'entiers e dont les éléments sont soit des entiers soit des opérateurs binaires. On suppose que

l'on dispose d'une fonction permettant de différencier les deux cas.

EVAL(e)
p ←pile[entier](e.taille)
pour i ←1 a e.taille
faire 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)
        r ←e[i] applique a a1 et a2
        empiler r
retourner sommet(p)

Exemple. 6 5 4 + 2 * -

Durée : 1H30

3.6 La file

Dans le cas d'une file, on fait des ajouts à une extrémité et les accés et les suppressions par l'autre extrémité. On a une structure FIFO "First In First Out" pour "premier entré premier sorti".

3.6.1 Signature du type abstrait file

Sorte file

Paramètre élément

Utilise booléen

Opérations

$file$	$:\rightarrow file$
$ajouter$	$: file \times élément \rightarrow file$
$retirer$	$: file \rightarrow file$
$premier$	$: file \rightarrow élément$
$vide?$	$: file \rightarrow booléen$

Les opérations *ajouter*, *retirer* et *file* sont des opérations internes, alors que les opérations *premier* et *vide?* sont des observateurs de file.

3.6.2 Axiomes du type abstrait file

- Les opérations *premier* et *retirer* ne sont définies que pour des files non vides.
- vide?(file()) = vrai
- vide?(ajouter(f,e)) = faux
- vide?(f) = vrai ⇒ premier(ajouter(f,e)) = e
- vide?(f) = faux ⇒ premier(ajouter(f,e)) = premier(f)
- vide?(f) = vrai ⇒ vide?(retirer(ajouter(f,e))) = vrai
- vide?(f) = faux ⇒ retirer(ajouter(f,e)) = ajouter(retirer(f),e)

3.6.3 Implémentation d'une file par un tableau circulaire

3.7 Les listes

La liste n'est pas un type abstrait aussi simple et bien défini que la pile ou la file. Il est impossible de proposer une définition précise. Le terme de liste est souvent associé à un type dans lequel les éléments sont rangés successivement et dans lequel il est possible d'insérer et de supprimer n'importe où. Mais une telle propriété ne suffit pas à définir un type car plusieurs signatures peuvent y correspondre.

Une référence standard, car elle est reconnue de fait, est la structure de listes du langage *lisp*. Elle correspond à la structure des données principale du langage ainsi qu'à la structure des programmes. Pourtant, ces listes n'ont pas une signature offrant la possibilité d'insérer et de supprimer n'importe quel élément. Bien au contraire, elle fournissent une signature minimale et optimale en terme de complexité permettant d'implémenter toute sorte d'opérations sur les ensembles dont la taille est inconnue.

Certains auteurs utilisent le terme de listes chaînées. Ils font alors référence à une implémentation de liste au moyen de structures chaînées et non à un type abstrait.

Dans cette section, nous présentons d'abord la référence *lisp* (qui correspond en fait exactement à la pile). Puis, nous présentons une signature assez classique, que l'on trouve souvent dans la littérature, fournissant des fonctions de suppression et d'insertion n'importe où dans la liste. Nous mettons ensuite en évidence les défauts de cette signature et proposons une meilleure

signature (en terme de complexité) utilisant un curseur.

3.7.1 La liste récursive

Définition

Une liste récursive l est

- soit la liste vide
- soit $l = \{e, l'\}$ où e est le premier élément de l, aussi appelé tête ou car(l) et l' est le reste de la liste, obtenu quand on enlève e. Alors, l' est aussi une liste récursive.

3.7.2 Signature

Appelons *listelisp* la sorte liste récursive. Nous utilisons ici les noms des accesseurs lisp.

Sorte listelisp

Paramètre élément

Utilise booléen

Opérations

$listelisp$	$:\rightarrow listelisp$
car	$: listelisp \rightarrow élément$
cdr	$: listelisp \rightarrow listelisp$
$cons$	$: listelisp \times élément \rightarrow listelisp$
$listelisp - vide?$	$: listelisp \rightarrow booléen$

Remarque. Le type abstrait liste récursive est le même, à un renommage des noms de fonctions près, que le type pile.

3.7.3 Axiomes

Voir la pile.

3.7.4 Implémentation

Une liste récursive peut aussi s'implémenter au moyen d'une structure chaînée ou d'un tableau.

LISTELISP[E]() **retourner** NULL

CONS(l,e)
retourner noeud[E](e,l)

CAR(l)
si vide?(p)
  alors erreur
  sinon retourner l.contenu

CDR(l)
si vide?(p)
  alors erreur
  sinon retourner l.suivant

LISTELISP-VIDE(l)

retourner l=NULL

3.7.5 Listes avec insertion et suppression

Première signature

Nous proposons une signature assez répandue dans les manuels d'ASD. La liste est une suite d'éléments ayant chacun un rang. On peut insérer un élement en fournissant le rang qui sera le sien après insertion. On peut supprimer un élement en indiquant son rang. De plus une fonction *ieme* permet de connaître la valeur de l'élément d'un certain rang.

Sorte liste

Paramètre élément

Utilise booléen

Opérations

$liste$	$:\rightarrow liste$
$ieme$	$: liste \times entier \rightarrow élément$
$inserer$	$: liste \times entier \times element \rightarrow liste$
$supprimer$	$: liste \times entier \rightarrow liste$
$liste - vide?$	$: liste \rightarrow booléen$

Implémentation au moyen d'un tableau

L'opération *ieme* est en $O(1)$ seulement si le tableau est de taille fixe. Mais nous considérons des ensembles dont le nombre d'éléments est inconnu, donc le tableau infini est nécessaire, et la complexité dans le pire des cas devient $O(n)$, avec n étant le nombre d'éléments de la liste. De même, les opérations *inserer* et *supprimer* nécessitent de décaler des éléments et sont donc en $O(n)$ dans le pire des cas.

Implémentation avec des structures chaînées

Les trois opérations *ieme*, *inserer* et *supprimer* sont en $O(n)$ dans le pire des cas.

Dans le cas de cette implémentation, considérons le scénario suivant.

Supposons que l'on souhaite supprimer le premier élément d'une liste l ayant la valeur v et considérons l'algorithme à mettre en oeuvre avec cette signature.

- Rechercher le k-ème élément de l ayant pour valeur v. Il faut faire une boucle parcourant l depuis le début et comparer le résultat de la fonction $ieme(l, i)$ à la valeur v. Renvoyer le rang i de l'élément correspondant ou $n + 1$ si la valeur n'est pas trouvée. Notons $rechercher(l, k, v)$ cette fonction (voir ci-dessous).
- Insérer un élément au rang i en utilisant directement la fonction *inserer*. Dans cette implémentation, un parcours depuis le début de la liste est à nouveau nécessaire : $inserer(l, rechercher(l, k, v), v)$.

RECHERCHER(l,k,v)

% Recherche de la kieme occurrence de v dans l

i ←1

j ←0

tant que $i \leq n$ **et** $ieme(l,i) \neq v$ **et** $j < k$

faire si ieme(l,i) = v

alors j ←j+1

i ←i+1

retourner i

On constate que l'opération d'insertion d'un élément selon sa valeur est en $O(2n)$, ce qui est inacceptable pour une opération de base.

Afin de solutionner ce problème, on pourrait envisager dans un premier temps les possibilités suivantes :

- Renvoyer un noeud et non une valeur pour l'élément trouvé par la fonction de recherche. Cette solution est extrèmement mauvaise car elle affiche dans la signature de la sorte des contraintes d'implémentation.
- Ajouter autant de fonctions que de services nécessaires identifiés. Cette solution est aussi mauvaise car le nombre de fonctions risque d'être important si l'on prend en compte tous les cas possibles (y compris les cas d'erreur) :
  - *inserer-avant-par-valeur*
  - *inserer-apres-par-valeur*
  - *inserer-avant-par-rang*
  - *inserer-apres-par-rang*

- *supprimer-avant-par-valeur*
- ...
- *inserer-a-la-fin-si-absent*
- ...

Deuxième signature

Cette signature associe à une liste l un curseur explicite qui va éviter les parcours multiples et redondants dans le cas d'une implémentation avec des structures chaînées. Des fonctions vont être associées au curseur pour le positionner. Elle admettent toutes une liste en argument et renvoient une liste en retour.

- *debut*(l) positionne le curseur sur le premier élément de la liste si celle-ci est non vide.
- $fin(l)$ positionne le curseur sur le dernier élément de la liste si celle-ci est non vide.
- *avant*(l) et *arriere*(l) font avancer ou reculer le curseur.
- *aller* positionne le curseur sur une position spécifique.
- *position* donne le rang du curseur.

Les cas de débordement seront identifiés par les fonctions suivantes.

- *trop-a-gauche*(l) est une fonction booléenne indiquant si le curseur a débordé à gauche de la première position.
- *trop-a-droite*(l) est une fonction booléenne indiquant si le curseur a débordé à droite de la dernière position.

Ainsi les fonctions d'insertion et de suppression admettent un paramètre de moins car elles opèrent sur le rang courant du curseur. On peut introduire les fonctions suivantes avec les conventions suivantes :

- La fonction *supprimer*(l) suprime l'élement de rang égal à celui du curseur et positionne le curseur sur son voisin de gauche.
- La fonction *inserer-a-droite*(l, v) insère un élement de valeur v à droite du curseur, sans modifier ce dernier.
- La fonction *inserer-a-gauche*(l, v) insère un élement de valeur v à gauche du curseur, sans bouger ce dernier. Son rang se trouve alors augmenté d'une position.
- La fonction *change-valeur*(l, v) change la valeur de l'élément sous le curseur. La valeur de cet élément est donnée par la fonction *valeur*.

Propriétés. Soit l une liste et n son nombre d'éléments.

- $0 \leq$ *position*$(l) \leq n + 1$
- *est-vide*(l) si et seulement si *trop-a-gauche*(l) et *trop-a-droite*(l)
- *trop-a-gauche*(l) si et seulement si *est-vide*(l) ou *position*$(l) = n - 1$.
- *trop-a-droite*(l) si et seulement si *est-vide*(l) ou *position*$(l) = n + 1$.

La recherche suivie d'une insertion peut alors s'écrire comme suit :

---

l = recherche(l,v,k)

si non *trop-a-droite*(l)

alors inserer-droite(l,v)

L'insertion d'une valeur v en position i s'écrit :

```
inserer(aller(l,i),v)
```

Remarques

- La complexité dépend de l'implémentation de cette sorte. Il faut remarquer qu'en plus du curseur, il faut conserver en interne des références vers les noeuds précédant et suivant (le curseur) afin d'assurer l'efficacité de l'insertion et de la suppression.
- On doit pouvoir aussi l'implémenter au moyen de deux files ayant pour tête le curseur et un voisin. À ce moment-là, les opérations *avant* et *arriere* sont en $O(1)$ mais pas les autres.
- Il conviendrait de comparer l'implémentation en structures chaînées de cette liste avec celle utilisant un tableau infini. Ainsi que les deux sortes listes avec la même implémentation sous forme de tableau.

Durée : 1H30

Chapitre 4

Structures de données arborescentes

Un arbre est un ensemble d'élements appelés *noeuds* (ou sommets) organisés de façon *hiérarchique* à partir d'un noeud distingué appelé *racine*. Un arbre est défini de façon récursive et induit par sa structure des algorithmes récursifs.

Exemple. La hiérarchie Unix pour les fichiers, avec la notion de chemin (PATH) et de répertoire (directory)

4.1 Les arbres binaires

On peut prendre comme exemple le déroulement d'un tournoi d'échec (père = vainqueur des deux fils) ou le codage d'une expression arithmétique binaire permettant de reconstituer le calcul.

$$(x - 2y)/(x(z - y))$$

DESSIN de l'arbre de syntaxe abstraite.

4.1.1 Définition

Un arbre binaire est soit vide (noté $\emptyset$), soit de la forme

$$B =< r, B_1, B_2 >$$

où B_1 et B_2 sont des arbres binaires disjoints et r un noeud appelé *racine*.

Remarques

- Il n'y a pas de symétrie gauche-droite, c'est-à-dire $< r, B_1, B_2 > \neq < r, B_2, B_1 >$ si $B_1 \neq B_2$.
- On repère les noeuds d'un arbre en leur donnant des noms symboliques tous différents.

4.1.2 Signature du type abstrait arbre binaire

Sorte ArbreBinaire

Utilise NoeudBinaire, booléen

Opérations

$\emptyset$	$:\rightarrow ArbreBinaire$
$<,,>$	$: NoeudBinaire \times ArbreBinaire \times ArbreBinaire \rightarrow ArbreBinaire$
$racine$	$: ArbreBinaire \rightarrow NoeudBinaire$
$gauche$	$: ArbreBinaire \rightarrow ArbreBinaire$
$droit$	$: ArbreBinaire \rightarrow ArbreBinaire$

Axiomes

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

Ces opérations ne sont définies que si leur argument est non vide.

4.1.3 Terminologie sur les arbres

- Soit $B =< r, B_1, B_2 >$ un arbre binaire. Le symbole r est la racine de B, B_1 (resp. $B_@$) est le sous-arbre gauche (resp. droit) de B. On dit que C est un *sous-arbre* de B si $C = B_1$ ou bien $C = B_2$.
- L'enfant gauche (droit) d'un *noeud* est la racine de son sous-arbre gauche (droit) et il y a un lien gauche (droit) entre le noeud et son enfant gauche (droit). Si n_i a pour enfant gauche (droit) n_j, n_i est le *parent* de n_j; Deux noeuds ayant même parents sont frères (siblings).
  Exemple : n_0 est la racine, n_1 est l'enfant gauche de n_0, n_3 est enfant droit de n_2, n_5 et n_6 sont frères de parent n_4.
- n_i est un ascendant de n_j ssi n_i est le parent de n_j ou un ascendant du

parent de n_j; n_j est un descendant de n_i.

Exemple : n_1 est un ascendant de n_3, n_5 est un descendant de n_0.

- Les noeuds d'un arbre binaire ont au plus deux enfants.
  - Un noeud interne binaire a deux enfants.
  - Un noeud interne unaire a un enfant (gauche ou droit).
  - Un noeud externe, ou feuuille, n'a pas d'enfants.

  Exemple : n_0 et n_4 sont des noeuds binaires, n_1 est unaire à gauche, n_2 est unaire à droite, n_3, n_5 et n_7 sont des feuilles.
- On appelle *branche* de l'arbre tout chemin (suite consécutive de noeuds) allant de la racine à une feuille; il y a autant de branches que de feuilles. La branche gauche (droite) est le chemin issu de la racine en ne suivant que des liens gauches (droits).

  Exemple : (n_0, n_4, n_5) est une branche, (n_0, n_1, n_2) est la branche gauche, (n_0, n_4, n_6) est la branche droite.

4.1.4 Caractéristiques d'un arbre binaire

- La *profondeur* (ou niveau) d'un noeud x est donnée par

$$p(x) = 0, \text{ si n est la racine}$$
$$p(x) = 1 + h(y) \text{ si y est parent de x.}$$

  Exemple : $h(n_0) = 0, h(n_1) = h(n_4) = 1, h(n_2) = h(n_5) = h(n_6) = 2, h(n_3) = h(n_7) = 3$.
- La profondeur (ou hauteur) d'un arbre binaire B est définie par

$$h(B) = max\{p(x),\ x \textit{ noeud de } B\}$$

– La longueur de cheminement de l'arbre B est

$$LC(B) = \sum_{x \ noeud} h(x) = \sum_{x \ feuille} h(x) + \sum_{x \ interne} h(x)$$

$$LC(B) = LCE(B) + LCI(B)$$

où LCE représente la longueur de cheminement externe de B et LCI représente la longueur de cheminement interne de B.

Exemple : $LCE(B) = 8$, $LCI(B) = 6$, $LC(B) = 14$.

– La profondeur moyenne externe de l'arbre B ayant f feuilles est $PE(B) = \frac{1}{f}LCE(B)$ (profondeur moyenne d'une feuille).

Exemple : PE(B) = 8/3 = 2.66

4.1.5 Un exemple de codage des arbres binaires

On peut coder un arbre binaire par un ensemble de mots, chacun d'entre eux étant le codage du chemin allant de la racine à un noeud donné. Ces mots sont obtenus par concaténation de 0 ou de 1 (alphabet = $\{0, 1\}$) :

– code(racine) = ϵ, le mot vide

– code(enfant gauche de x) = code(x)0

– code(enfant droit de x) = code(x)1

Exemple : $\{\epsilon, 0, 00, 001, 1, 10, 11, 110\}$

4.1.6 Bornes sur la profondeur d'un arbre binaire

Proposition. Pour tout arbre binaire de taille (nombre total de noeuds) n, de profondeur p, et ayant f feuilles, on a

$$\lfloor \log_2(n) \rfloor \leq p \leq n - 1$$

et

$$p \geq \lceil \log_2(f) \rceil$$

Preuve : Pour p profondeur fixée

- l'arbre binaire ayant le plus petit nombre de sommets est celui qui n'est formé que de noeuds unaires et d'une feuille. On a dans ce cas $p+1$ sommets. Donc

  $$n \geq p+1$$

  c'est-à-dire

  $$p \leq n-1$$

- l'arbre binaire ayant le plus de sommets est celui où tous les noeuds de hauteurs 0, 1, ..., p-1 sont binaires. On a alors

  $$1 + 2 + 2^2 + \ldots + 2^{p-1} + 2^p = 2^{p+1} - 1$$

  noeuds en tout. On aura donc

  $$n \leq 2^{p+1} - 1$$

  c'est-à-dire

  $$n < 2^{p+1} \Rightarrow \log_2(n) < p+1$$

  Donc

  $$\lfloor \log_2(n) \rfloor \leq p$$

On a montré la première double inégalité.

Pour montrer la deuxième, il suffit de voir que le nombre de feuilles f est toujours tel que

$$f \leq 2^p$$

(2^p est le nombre de feuilles d'un arbre binaire parfait, ce qui correspond au cas précédent avec $n = 2^{p+1} - 1$ et donc $f = (n+1)/2$). On aura alors $\log_2(f) \leq p$ c'est-à-dire $p \geq \lceil \log_2(f) \rceil$.

4.1.7 Arbres binaires complets

Un arbre binaire est **complet** si tous les noeuds ont soit aucun enfant soit deux enfants.

Exemples.

Proposition 1. Un arbre binaire complet ayant n noeuds internes a $(n+1)$ feuilles.

Preuve. On fait une récurrence sur n.

- La propriété est vraie pour un arbre réduit à une feuille ($n = 0$).
- On suppose la propriété vraie pour tous les arbres binaires complets ayant moins de n noeuds internes. Soit

$$B =< \emptyset, B_1, B_2 >$$

un arbre binaire complet ayant n noeuds internes. Les arbres B_1 et B_2 sont deux arbres binaires complets et on a

$$n = n_1 = n_2 + 1$$

où n_1 (n_2) est le nombre de noeuds internes de B_1 (B_2). Par hypothèse de récurrence, le nombre de feuilles de B_1 (B_2) est $n_1 + 1$ ($n_2 + 1$).

Or, une feuille de B est une feuille de B_1 ou de B_2, donc le nombre de feuilles de B est $n_1 + 1 + n_2 + 1 = n + 1$

Durée : 1H30

Proposition 1. Il existe une bijection entre l'ensemble $\mathcal{B}_n$ des arbres binaires de taille n, et l'ensemble $\mathcal{BC}_{2n+1}$ des arbres binaires complets de taille $2n + 1$.

Preuve. On enlève toutes les feuilles d'un arbre binaire complet et on obtient un arbre binaire. La bijection inverse consiste à remettre des feuilles partout de sorte que chaque noeud ait deux enfants.

Faire un dessin.

Propriété. Le nombre b_n d'arbres binaires de taille n (et donc bc_{n+1} le nombre d'arbres bianires complets de taille $2n + 1$ est égal à $\frac{1}{n+1}C_{2n}^n$.

4.1.8 Parcours d'arbres binaires

L'opération de parcours sur les arbres binaires consiste à examiner systématiquement, dans un certain ordre, les noeuds de l'arbre pour y effectuer un traitement donné.

Parcours en profondeur

Le parcours en profondeur à gauche consiste à partir de la racine et à choisir toujours l'enfant le plus à gauche d'abord. De façon imagée, l'arbre

peut être vu comme un mur que l'on suit avec sa main gauche en partant à gauche depuis la racine.

Exemple. Faire un dessin en faisant figurer les noeuds vides et en considérant l'arbre complet.

```
PARCOURS(x)
si x ≠NULL
   alors PARCOURS(gauche(x))
         PARCOURS(droit(x))
```

On constate que chaque noeud est rencontré trois fois : une fois en descendant, puis une deuxième fois en remontant depuis l'enfant gauche, et une troisième fois en remontant depuis l'enfant droit.

Ce parcours contient trois ordres importants d'exploration des arbres selon la place du traitement de chaque noeud par rapport aux appels récursifs.

- **Ordre préfixe** ou préordre : le traitement du noeud s'effectue avant le premier appel récursif. Supposons que ce traitement soit un simple affichage du noeud.

  ```
  PARCOURS-PREFIXE(x)
  si x ≠NULL
     alors PRINT(x)
           PARCOURS(gauche(x))
  ```

PARCOURS(droit(x))

Exemple : dessin + affichage.

– **Ordre infixe** ou symétrique : le traitement du noeud s'effectue entre les deux appels récursifs.

PARCOURS-INFIXE(x)
si x ≠NULL
alors PARCOURS(gauche(x))
PRINT(x)
PARCOURS(droit(x))

Exemple : dessin + affichage.

– **Ordre postfixe** ou suffixe : le traitement du noeud s'effectue après les deux appels récursifs.

PARCOURS-POSTFIXE(x)
si x ≠NULL
alors PARCOURS(gauche(x))
PARCOURS(droit(x))
PRINT(x)

Exemple : dessin + affichage.

Remarques

- Dans le premier chapitre de ce cours, nous avons dit qu'une fonction récursive nécessitait la mémorisation de ses données et de ses paramètres pour pouvoir effectuer des opérations au retour de l'appel récursif. La structure de donnée utilisée est une pile. Faire tourner la fonction PARCOURS-INFIXE sur un exemple et donner l'évolution de la pile.
- En conséquence, afin d'écrire une fonction itérative représentant le même traitment qu'une fonction récursive, il faut gérer explicitement une pile.

Exercice. Écrire ces algorithmes de façon itérative, en gérant soi-même une pile de noeuds.

Parcours en largeur

Le parcours en largeur consiste à explorer tous les enfants d'un noeud avant d'explorer les enfants de ses enfants.

Exemple. Dessiner un arbre binaire et montrer la nécessité d'utiliser une file pour traiter les noeuds dans l'ordre de leur arrivée.

La fonction suivante doit être appelée avec la racine de l'arbre et une file vide en arguments.

PARCOURS-LARGEUR(x,f)

si x $\neq$NULL

alors PRINT(x)

f ←ajouter(file, gauche(x))

f ←ajouter(file, droit(x))

x ←premier(file)

f ←retirer(file)

parcours-largeur(x,f)

% Il peut rester des noeuds dans la file

sinon si non vide?(f)

x ←premier(file)

f ←retirer(file)

parcours-largeur(x,f)

Durée : 1H30

4.1.9 Arbres binaires de recherche

Un arbre binaire de recherche permet de traiter des ensembles dynamiques de données admettant un ordre en fournissant beaucoup d'opérations (rechercher, minimum, maximum, prédécesseur, successeur, insérer, supprimer).

La complexité en temps des opérations de base d'un arbre de recherche est proportionnelle à la hauteur de l'arbre. Si l'arbre binaire est complet et a n noeuds, cette complexité est donc en $\theta(\log_2 n)$. Si par contre l'arbre est une chaîne ayant une seule feuille, cette complexité est en $\theta(n)$.

Définition

Un arbre binaire de recherche est un arbre binaire dans lequel on stocke des éléments, appelés des clés, à chaque noeud. Les clés admettent un type muni d'une relation d'ordre. Les clés sont organisées dans l'arbre selon la propriété suivante :

> *Soit x un noeud de B, si y est un noeud du sous-arbre gauche de B, alors la clé de x est plus grande que la clé de y. Si y est un noeud du sous-arbre droit de B, alors la clé de x est plus petite que la clé de y.*

Exemple. FAIRE UN DESSIN

Implémentation d'arbre binaire de recherche

Un arbre binaire peut se représenter au moyen d'un tableau ou bien de structures chaînées. Dans les deux cas, il faut définir au préalable le type abstrait des noeuds. C'est une structure à quatre champs : la clé, l'enfant gauche, l'enfant droit et le parent.

Sorte NoeudBinaire

Paramètre élément

Utilise NoeudBinaire et booléen

Opérations

$NoeudBinaire$	: $élément \times NoeudBinaire^3 \rightarrow NoeudBinaire$
cle	: $NoeudBinaire \rightarrow élément$
$gauche$	: $NoeudBinaire \rightarrow NoeudBinaire$
$droit$	: $NoeudBinaire \rightarrow NoeudBinaire$
$parent$	: $NoeudBinaire \rightarrow NoeudBinaire$
$changer\text{-}cle$	: $NoeudBinaire \times élément \rightarrow NoeudBinaire$
$changer\text{-}gauche$	: $NoeudBinaire \times NoeudBinaire \rightarrow NoeudBinaire$
$changer\text{-}droit$	: $NoeudBinaire \times NoeudBinaire \rightarrow NoeudBinaire$
$changer\text{-}parent$	: $NoeudBinaire \times NoeudBinaire \rightarrow NoeudBinaire$

Un tel type abstrait s'implémente au moyen d'une structure à quatre champs : la clé, l'enfant gauche, l'enfant droit et le parent :

structure[élément, NoeudBinaire, NoeudBinaire, NoeudBinaire]

(cle, gauche, droit, parent)

Un arbre binaire s'implémente alors au moyen d'un ensemble de noeuds chaînés entre eux par les liens $enfants$. Ces noeuds peuvent être disposés de façon contiguë dans un tableau (faire un dessin), ou bien de façon non contiguë dans la mémoire. En réalité, ce choix a une incidence seulement au moment de la construction d'un arbre binaire (allocation de noeuds), et lors de la suppression de noeuds d'un arbre binaire (libération de noeuds). Cet aspect (gestion de la mémoire avec allocation et libération de structures) sera étudié ultèrieurement.

Choisissons ici d'implémenter notre arbre binaire au moyen de structures chaînées (ensemble de noeuds disposés de façon non contiguë dans la mémoire). Alors, un arbre est implémenté par une référence à sa racine que

l'on peut encapsuler dans une structure :

structure[NoeudBinaire](racine)

$\emptyset()$

b = structure[NoeudBinaire](racine)

b.racine = NULL

retourner b

<,>(r,b1,b2)

n = NoeudBinaire(r,b1.racine,b2.racine, NULL)

b1.racine.parent ←n

b2.racine.parent ←n

b = structure[NoeudBinaire](racine)

b.racine = n

retourner b

RACINE(x)

retourner x.racine

GAUCHE(x)

retourner x.racine.gauche

DROIT(x)

retourner x.racine.droit

Parcours en profondeur d'un arbre binaire de recherche

L'organisation des clés d'un arbre binaire de recherche est telle que lorsque l'on effectue un parcours en profondeur en ordre infixe de l'arbre, on obtient les clés en ordre croissant.

Requêtes dans un arbre binaire de recherche

Plusieurs sortes de requêtes sont envisageables dans un arbre binaire de recherche : rechercher (une clé), minimum, maximum, successeur et prédecesseur. Ces opérations sont toutes de complexité en $O(h)$, où h est la profondeur de l'arbre.

– La fonction ARBRE-RECHERCHER doit être appelée avec la racine de l'arbre et la clef recherchée. Elle retourne une référence vers un noeud contenant la clef ou bien NULL.

ARBRE-RECHERCHER(x,k)
si x = NULL ou k = x.cle
  alors retourner x
si k < x.cle
  alors retourner arbre-rechercher(x.gauche,k)
  sinon retourner arbre-rechercher(x.droit,k)

Les noeuds rencontrés par la fonction forment une branche allant de la racine vers une feuille de l'arbre. Sa longueur est donc au plus la profondeur de l'arbre.

Cette fonction peut s'écrire de façon itérative. Pour certains langages, la version itérative est plus efficace que la version récursive.

ARBRE-RECHERCHER-ITERATIVE(x,k)

tant que x $\neq$NULL **et** k $\neq$x.cle

faire si k $<$ x.cle

alors x $\leftarrow$x.gauche

sinon x $\leftarrow$x.droit

retourner x

– On peut trouver la clef minimum d'un arbre binaire de recherche en prenant l'enfant gauche de chaque noeud. En effet, si le noeud x ne possède pas d'enfant gauche, la clef minimum du sous-arbre enraciné en x est $x.cle$, puisque toutes les clefs à droite sont plus grandes. Si par contre, x a un sous-arbre gauche, la clef minimum de l'arbre enraciné en x est la clef minimum de son sous-arbre gauche.

ARBRE-MINIMUM(x)

tant que x.gauche $\neq$NULL

faire x $\leftarrow$x.gauche

retourner x

La fonction maximum est symétrique. Ces deux fonctions s'exécutent en $O(h)$ puisqu'elles suivent chacune une branche depuis la racine jusqu'à une feuille.

– La structure d'un arbre binaire de recherche permet de déterminer le successeur d'un noeud sans même effectuer de comparaisons entre les clefs.

  Il faut considérer deux cas. En effet, si le sous-arbre droit du noeud n'est pas vide, le successeur de x est le minimum du sous-arbre droit. Sinon, il faut remonter à partir du noeud x jusqu'à emprunter un lien gauche. Le successeur est le premier noeud obtenu sur ce chemin par un lien gauche (en remontant vars la racine).

Durée : 1H30

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 ←y.parent
  retourner y

Insertion et suppression

p245 Cormen

Remarques

La hauteur d'un arbre binaire de recherche varie alors que les éléments sont insérés ou supprimés. La hauteur moyenne d'un arbre crée par des insertions et suppression n'est pas connue en général. Par contre, si l'arbre est crée par des insertions, où les n! permutations sont équiprobables, on peut montrer que sa hauteur moyenne est $O(\log n)$.

Durée : 1H30

4.1.10 Arbres rouge et noir

Les arbres rouge et noir sont un des nombreux schémas d'arbres binaires de recherche dits "équilibré", garantissant que les opérations s'éxécutent en $O(\log n)$.

Propriétés des arbres rouge et noir

1. Chaque noeud est soit rouge soit noir.
2. Chaque feuille est noire.
3. Si un noeud est rouge alors ses deux fils sont noirs.
4. Les branches allant de la racine à une feuille possèdent le même nombre de noeuds noirs.

Définition On appelle le **hauteur noire** d'un noeud x, le nombre de noeuds présent dans la branche descendant de ce noeud vers une feuille. On la notera $h_n(x)$.

Lemme Un arbre rouge et noir comportant n noeuds internes a une hauteur au plus égale à $2\log(n+1)$.

Preuve

- un arbre enraciné en x quelconque contient au moins

$$2^{h_n(x)} - 1$$

  noeuds internes (par récurrence).
- Soit h la hauteur de l'arbre. Au moins $h/2$ noeuds sont noirs, donc

$$h_n(r) \geq n/2$$

  Donc selon le point précédant

$$2^{h/2} - 1 \leq n$$

Implémentation Afin de simplifier les algorithmes, on va représenter différemment les feuilles d'un arbre rouge et noir. Au lieu de représenter les feuilles par une simple référence $NULL$, nous allons utiliser la technique des sentinelles et considérer que les feuilles sont représentées par un noeud externe d'une référence donnée.

∅()
b = structure[NoeudBinaire,NoeudBinaire](racine,feuille)
b.feuille = NoeudBinaire(NULL,NULL,NULL,NULL)
b.racine = feuille
retourner b

<,>(r,b1,b2)

n = NoeudBinaire(r,b1.racine,b2.racine, NULL)

b1.racine.parent ←n

b2.racine.parent ←n

b = structure[NoeudBinaire](racine)

b.racine = n

b.feuille = b1.feuille

retourner b

RACINE(x)

retourner x.racine

GAUCHE(x)

retourner x.racine.gauche

DROIT(x)

retourner x.racine.droit

Rotations

Les opérations de rotations permettent de modifier les couleurs des noeuds et de changer leurs chaînage. Elles sont utiles pour transformer l'arbre en cas de violation de ses propriétés aprés une insertion ou une suppression.

Rotations gauche et droite. Donner l'exemple et l'algorithme p261 du

Cormen.

Insertion

Exemple et code p263 du Cormen.

Durée : 1H30

Suppression

Exemple et code p267 du Cormen.

Durée : 1H30

4.1.11 Les tas

Définition

Arbre binaire presque complet. Implémentation possible dans un tableau.

- Longeur : nombre d'éléments du tableau.
- Taille : nombre d'élements du tas.

Donner parent, gauche et droit Cormen p139.

Propriété de tas.

$$A[pere(i)] \geq Ai$$

Conservation de la structure de tas

- Procédure entasser(A,i) : i peut violer la propriété de tas : on va le faire descendre.
- Calcul complexité : $T(n) \geq T(2n/3) + \Theta(1)$. $O(\lg n)$.

Construction d'un tas

- Construire-tas(A) : un tableau quelconque est transformé en tas. On appelle la procédure entasser en partant de la fin du tableau.
- Borne large : $O(\lg n)$ pour entasser et $O(n)$ pour le nombre d'appels.

$$O(n \lg n)$$

- Borne plus précise : en tenant compte des profondeurs différentes des sous-tas à traiter. **remarque :** le nombre de noeuds dont le sous-arbre est de profondeur p dans un tas à n sommet est égal au nombre de noeuds de profondeur $\lg n - p + 1$. Ce nombre est égal à

$$2^{\lg n - p + 1} = \frac{2^{\lg n}}{2^{p+1}} = \frac{n}{2^{p+1}}$$

pour les arbres binaires complets, et donc borné par cette valeur pour les tas. Le temps requis par ENTASSER pour un tas de profondeur p étant $O(p)$, on peut écrire pour le temps de CONSTRUIRE-TAS :

$$\sum_{p=0}^{\lg n} \frac{n}{2^{p+1}} O(p) = O(n \sum_{p=0}^{\lg n} \frac{p}{2^p})$$

Or $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$. Donc en substituant $x = 1/2$, on a $\sum_{p=0}^{\infty} p/2^p = \frac{1/2}{(1-1/2)^2} = 2$. Donc

$$O(n \sum_{p=0}^{\lg n} \frac{p}{2^p}) = O(n \sum_{p=0}^{\infty} \frac{p}{2^p}) = O(n)$$

Code et exemple. Cormen p143-144.

Tri par tas

- On se sert de CONSTRUIRE-TAS pour construire le tas sur le tableau d'éntrée. Ensuite, on échange l'élément se trouvant à la racine avec celui se trouvant en fin de tableau. Puis, on exclut cet élément et on corrige éventuellement la racine avec ENTASSER(A,1).
- Complexité : $O(n \lg n)$.

Files de priorité

SorteFile de priorité

Paramètre élément

Utilise booléen

Opérations

$file - priorite$	$:\rightarrow file - priorite$
$inserer$	$: file - priorite \times élément \rightarrow file - priorite$
$maximum$	$: file - priorite \rightarrow élément$
$extraire - max$	$: file - priorite \rightarrow file - priorite$

Applications. Plannification des tâches sur un ordinateur à ressources partagées. Simulateur événementiel.

Implémentation. Tas.

Code et exemple. Cormen p 147 et 148.

- EXTRAIRE-MAX : ressemble à la procédure TRIER-TAS. On extrait la racine. Complexité $O(\lg n)$.
- INSERER : On met l'élément supplémentaire à la fin et on le fait remonter. Complexité $O(\lg n)$.

Durée : 1H30

Table des figures