Structures arborescentes

Denis Lapoire

22 novembre 2006

Table des matières

Chapitre 1

Introduction

Le cours précédent d'Initiation Algorithmique a présenté les deux notions de problèmes et d'algorithme. Il a insisté sur différentes familles d'algorithmes (algorithme glouton, algorithme divide and conquer, programmation dynamique). Il a montré comment un même problème pouvait être résolu par des algorithmes différents appartenant à ces différentes familles. La qualité première attendue de ces algorithmes était leur correction.

Nous nous intéresserons ici plus particulièrement à une seconde qualité à savoir leur moindre coût en terme d'utilisation d'espace mémoire et de temps.

À cette fin, nous verrons comment représenter une *ensemble* dans une zone mémoire de la machine sur laquelle est exécutée un programme et ce en définissant un objet intermédiaire : le *type abstrait*.

1.1 De l'abstrait au concret et inversement

Les problèmes algorithmiques rencontrées lors de ce se semestre admettent pour entrée principalement un unique objet plus complexe que de simples éléments : l'objet *ensemble*.

Le modèle de calcul, c'est à dire la représentation conceptuelle de la machine sur laquelle sont exécutés les programmes, fournit une mémoire dont les accès en lecture et en écriture se font en temps constant pourvu que l'on ait l'adresse des octets à lire ou modifier. Ainsi, d'un point de vue théorique, la zone mémoire peut etre considérée comme un immense *tableau*.

Les algorithmes que nous souhaitons écrire sont des algorithmes possédant la plupart des propriétés ci-dessous :

- leur indépendance vis à vis et de la machine et du langage de programmation dans lequel seront écrits le programme.
- la simplicité d'écriture, qui permet leut compréhension et la preuve de leur correction.
- leur faibles complexités en temps et en espace.

En conséquence, les algorithmes ne seront pas écrits en code assembleur mais dans un langage de haut niveau. Ainsi, dans un algorithme ne sera pas décrit les instructions de bas niveau accédant à telle ou telle adresse de la zone mémoire. Cette tache est dévolue à des fonctions auxiliaires utilisées dans l'algorithme qui seront regroupées dans des ensembles logiques cohérents que nous nommons "types abstraits".

Ainsi, entre l'objet mathématique très abstrait qu'est l'ensemble et l'objet très concret le tableau qui est la zone mémoire, nous concevrons et manipulerons :

1. de nouveaux objets mathématiques plus structurés que l'ensemble, comme par exemple la *séquence*, la *relation*.
2. des types abstraits qui réalisent cette interface entre un univers mathématique logique et un univers informatique calculatoire. Ainsi, chaque type abstrait sera un ensemble d'opérations défini logiquement et pouvant etre implémentées à l'aide de programmes exécutables. Ces types abstraits auront pour noms l'*ensemble*, le *tableau*, la *structure*, la *pile*, la *file*, la *liste*, l'*arbre binaire*, le *tas*, etc...

1.2 Brefs rappels mathématiques

La connaissance et la maîtrise des notions suivantes est indispensable. Se référer à tout ouvrage de mathématique.

1.2.1 Ensemble

Un *ensemble* est un objet mathématique qui possèdent d'autres objets appelés ses *éléments*; l'appartenance de l'élément a à l'ensemble A est noté $a \in A$.

L'*ensemble vide*, noté $\emptyset$, est l'unique ensemble ne contenant aucun élément.

Un *singleton* est un ensemble contenant un unique élément. Une *paire* est un ensemble contenant exactement deux éléments.

Notions et notations importantes

Intervalle, cardinalité, partie, opérations union, intersection, différence, complément.

1.2.2 Relation binaire

Le *produit cartésien* de deux ensembles A et B est l'ensemble noté $A \times B$ ayant pour éléments tous les couples de la forme (a, b) avec $a \in A$ et $b \in B$.

Une *relation binaire* r définie sur A et B est une partie du produit cartésien $A \times B$. L'appartenance $(a, b) \in r$ est souvent noté $a\ r\ b$.

Notions et notations importantes

Réflexivité, associativité, symétrie, relation d'ordre (partiel ou total), relation bien fondé, relation d'équivalence, classes d'équivalence, les ensemble $\mathbb{N}$ et $\mathbb{N}^*$, intervalle d'entiers, relation d'arité k.

1.2.3 Fonction

Etant donnés deux ensembles A et B, une *fonction* f de A vers B est une relation binaire $r \subseteq A \times B$ telle que pour tout élément $a \in A$ il existe un et un seul élément $b \in B$ tel que $(a, b) \in f$. L'appartenance $(a, b) \in f$ est noté $f(a) = b$.

Notions et notations importantes :
Injectivité, surjectivité, bijectivité, morphisme, isomorphisme, composition, restriction, inverse, involution, permutation.

1.2.4 Séquence

Une *séquence* à valeurs dans un ensemble A est une fonction de la forme $s : \mathbb{N}^* \to A$ ou de la forme $s : [1, i] \to A$ avec $i \in \mathbb{N}$. Dans ce dernier cas, la séquence s peut etre notée $(s(1), \ldots, s(i))$, l'entier i éventuellement nul, est la *longueur* de s.

La séquence vide définie à partir de $[1, 0] = \emptyset$ est notée $()$ et est de longueur 0.

Notions et notations importantes

Sous-séquence extraite, concaténation, mot.

1.3 Un type abstrait : l'ensemble

Le type abstrait qui sera le fil conducteur de ce cours est le type ensemble.

Sa signature est simplement :

```
estVide       : ensemble            → booléen
ensembleVide  :                     → ensemble
appartient    : élément × ensemble  → booléen
ajouter       : élément × ensemble  → ensemble
enlever       : élément × ensemble  → ensemble
choisir       : ensemble            → élément
```

En supposant que l'on mette à votre disposition chacun des types cités, comment implémenter de façon optimale à l'aide de l'un de ces types ce type ensemble. Il n y a pas de réponse à une telle question. Tout dépend :

1. d'informations supplémentaires sur la réalité des ensembles manipulés. Quelles sont leurs cardinalités ? Quelle est la cardinalité de leur univers ? Quelles sont les relations et opérations opérant sur les éléments de ces ensembles ?
2. d'informations sur les opérations les manipulant ? Quelles sont les primitives jamais ou rarement utilisées ? souvent utilisées ?
3. et plus généralement de l'algorithme considéré.

Afin de faciliter l'écriture algorithmique, nous séparons :

1. l'écriture d'un algorithme utilisant par exemple le type `ensemble`.
2. de l'implémentation du type ensemble et la compréhension de l'exécution machine de ses primitives.

Cette séparation est nécessaire dans une première étape et permet en distinguant les deux problèmes de pouvoir les résoudre plus facilement car séparément. Dans la dernière étape conclusive, le choix de l'implémentation devra être justifié en considérant à la fois l'algorithme et l'implémentation et ainsi permettre d'évaluer le choix de cet algorithme et de cette implémentation en tentant de répondre à la question :

- est-ce que pour cet algorithme, cette implémentation permet d'avoir une complexité en temps (et ou en espace) optimale ?
- est-ce pour le problème considéré, cet algorithme et ce cette implémentation ont une une complexité en temps (et ou en espace) optimale ?

Deux familles d'implémentations

Cette section balaie très rapidement les liens entre différentes stratégies de représentation machine d'un ensemble et leurs liens avec les notions de séquence ou d'arbre.

L'implémentation d'un type abstrait sera réalisé à l'aides d'autres types abstraits. Cependant l'analyse des coûts en temps et en espace nécessite de comprendre la façon dont l'objet sera représenté in fine sur la machine. Rappelons que la mémoire est formé d'un nombre N de blocs de même taille, identifiés par des *adresses* ; l'accès au contenu de chacun de ces blocs à partir de son adresse se faisant en temps constant.

1.3.1 À taille constante

Une première classe d'implémentations machines sont celles qui utilisent un nombre blocs constant pour mémoriser un ensemble de taille variable (pourvu qu'il ne dépasse pas une certaine cardinalité).

Tableaux de booléens

Une façon de représenter l'ensemble du dernier tirage au Loto $\{3, 7, 13, 19, 23, 25, 28\}$ est d'utiliser le tableau de booléens à indices $1, \ldots, 49$ suivant :

```
indices: 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
valeurs: 0  0  1  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0  1  0

indices: 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
valeurs: 0  0  1  0  1  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0

indices: 41 42 43 44 45 46 47 48 49
valeurs:  0  0  0  0  0  0  0  0  0
```

L'avantage d'une telle représentation est de pouvoir exécuter les différentes primitives `appartient`, `enlever`, `ajouter` en temps constant. Constante qui dépend de l'accès à un bloc mémoire ; l'opération dure une fraction de seconde.

Les exécutions de `estVide`, `ensVide`, `choisir` sont ici aussi en temps constant ; mais une constante éventuellement importante, proportionnelle au nombre d'indices. En ajoutant un nouveau champ indiquant le nombre d'éléments de l'ensemble, nous obtenons une complexité en temps de `estVide` "petitement" constant.

Dans l'exemple ci-dessus la solution est envisageable car l'univers de tous les éléments possibles d'un ensemble est de cardinalité "proche" (49 dans l'exemple) de celle de l'ensemble (7 dans l'exemple).

Tables de hachage

Supposons que l'on s'intéresse à l'ensemble des patronymes français (que nous supposerons au nombre de $100000 = 10^5$ tirages).

Sachant qu'un caractère latin est codé à l'aide de 5 bits et qu'un patronyme possède au plus 20 caractères, on pourrait utiliser un tableau de booléens à ensemble d'indices $2^{5 \cdots 20} \simeq 10^{30}$. Le coût et la perte de mémoire est alors prohibitif (seul $10^{-23}\%$ de l'espace est réellement utilisé !).

Une idée consiste à transformer selon une fonction de hachage $h : \mathcal{A}^* \to [0, 10^6]$ chacun des noms en une valeur de taille inférieure par exemple ayant 6 digits (le nombre d'indices du tableau est ainsi ramené de 10^{30} à 10^6.

L'ensemble $\{$`PICCOLET D'HERMILLON`, `PICHAT`, $\ldots\}$ est ainsi représenté par le tableau

```
indices : ... h(PICCOLET D'HERMILLON)  ... h(PICHAT) ...
valeurs :  ... PICCOLET D'HERMILLON   ... PICHAT   ...
```

Observons qu'avec une telle représentation, toutes les "bonnes" propriétés du tableau à valeurs booléennes sont conservées. Les opérations sont à coût constant.

Une première difficulté apparaît : l'existence de *collisions* à savoir des tirages $t \neq u$ ayant même hachés ($h(t) = h(u)$).

1.3.2 À taille proportionnelle à la cardinalité de l'ensemble représenté

Différents choix s'offrent alors.

1.3.3 Représentations par une "séquence"

Une façon naturelle de représenter tout ensemble par une séquence : c'est à dire par exemple représenter l'ensemble $\{3, 7, 13, 19, 23, 25, 28\}$ par la séquence $(13, 23, 3, 7, 28, 25, 19)$. Observons ici les différents choix qui s'offrent selon que :

1. on autorise ou non la duplication des éléments. A t-on intérêt à représenter $\{3, 7, 13, 19, 23, 25, 28\}$ par la séquence $(3, 7, 13, 19, 13, 19, 28, 3, 3, 23, 25, 28)$?
2. on impose ou non à ce que les éléments apparaissent dans la séquence en respectant l'éventuel ordre total de l'ensemble. A t-on intérêt à représenter $\{3, 7, 13, 19, 23, 25, 28\}$ par la séquence $(3, 7, 13, 19, 23, 25, 28)$?

Les types abstraits permettant d'implémenter de telles séquences ne manquent pas :

1. le type Tableau
2. le type Pile.
3. le type File.
4. le type Liste.

Représentation contiguë

Une première technique pour représenter la liste $(3, 7, 13, 19, 23, 25, 28)$ est d'utiliser 7 blocs mémoire contigus dont les adresses définissent intrinsèquement l'ordre des éléments qu'elles contiennent.

La liste $(3, 7, 13, 19, 23, 25, 28)$ peut ainsi avoir pour représentation machine :

`adresses`	`12123`	`12124`	`12125`	`12126`	`12127`	`12128`	`12129`
`contenus`	`3`	`7`	`13`	`19`	`23`	`25`	`28`

Cette représentation est la représentation naturelle du type Tableau ; cette proximité entre ce type et cette implémentation ne doit cependant pas être considérée comme absolue : en conclusion de ce chapitre, nous verrons pourquoi représenter des tableaux sous la forme de listes chaînées !

L'un des inconvénients d'une telle représentation est la gestion de cet espace mémoire. Si au cours de l'exécution de votre algorithme, vous souhaitiez augmenter l'ensemble $\{3, 7, 13, 19, 23, 25, 28\}$ d'un seul élément et si les blocs se trouvant aux adresses `12122` et `12130` n'étaient pas libre, il serait alors nécessaire de trouver un minimum de 8 blocs contigus libre et de recopier tous les éléments.

Représentation par chaînage

Une technique pour implémenter l'un de ces types est le chaînage c'est à d'accoler à l'élément de la séquence l'adresse de l'élément qui lui succède. Ainsi la séquence $(13, 23, 3, 7, 28, 25, 19)$ aura par exemple pour représentation-machine l'ensemble des blocs d'adresses respectives :

```
adresses    12123   12231     13213    34548    90808    98773    98799
```

ayant respectivement pour contenu :

```
contenus 28:90808 7:12123  23:98799 13:13213 25:98773  19:------ 3:12231
```

L'avantage d'une telle méthode est de pouvoir en temps contant :

1. supprimer en temps constant un élément de la séquence, pourvu que l'on ait son adresse.
2. ajouter en temps constant un élément qui n'est pas déjà présent dans la séquence.
3. tester si une séquence est vide.
4. créer une liste vide.

L'un des inconvénients est de se mettre en situation de parcourir jusqu'à la séquence toute entière pour :

1. pour décider si un élément s'y trouve.
2. calculer l'adresse d'un élément de cette séquence.

1.3.4 Représentation par un arbre

Décider en temps linéaire l'appartenance d'un élément à un ensemble étant un handicap, une alternative convaincante à la structure uni-dimensionnelle qu'est la séquence est l'objet bidimensionnel qu'est l'arbre.

Différentes sortes d'arbre existent. Ceux étudiés dans ce cours seront tous "enracinés", c'est à dire possédant un sommet singulier appelé la "racine" de l'arbre. Citons-en quelques-uns :

- les arbres binaires de recherche.
- les arbres rouges et noirs, une variante des précédents.
- les tas.

L'idée commune à ces arbres est d'utiliser les propriétés relatives des éléments (en supposant qu'ils sont munis d'un ordre total : ce qui peut toujours être supposé) pour les placer intelligemment dans une structure dans laquelle la recherche d'un élément ne nécessite pas de parcourir tous les éléments comme cela se passe dans une séquence chaînée.

Arbre binaire

L'exemple le plus simple est l'arbre binaire de recherche dans lequel tout noeud de l'arbre possède (généralement) un noeud fils gauche et un noeud fils droit et tels que les trois éléments de ces trois noeuds p, g, d vérifient respectivement $g \leq p < d$.

L'ensemble $\{3, 7, 13, 19, 23, 25, 28\}$ peut ainsi être représenté par l'arbre suivant :

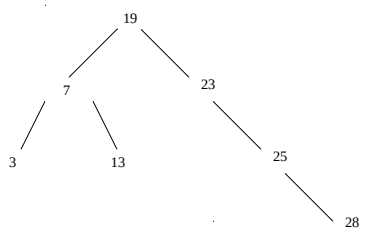


FIG. 1.1 – Un arbre binaire de recherche

L'ensemble des opérations

```
appartient      élément, ensemble -> booléen
ajouter         élément, ensemble -> ensemble
enlever         élément, ensemble -> ensemble
```

admettent des algorithmes de complexité en temps proportionnel à la hauteur de l'arbre (dans l'exemple précédent 3). Ces algorithmes seront étudiés dans le chapitre 6.

Des techniques relativement simples peuvent garantir que une hauteur d'arbre raisonnable voire même optimale à savoir $log_2(n)$ où n est la cardinalité de l'ensemble. L'une de ces techniques est justement le fait des arbres rouges et noirs.

Tas : un exemple remarquable de limiter les primitives au strict minimum

Parfois le type ensemble utilisé est dégradé c'est à dire défini à partir du type ensemble présenté plus haut en :

- supprimant une ou plusieurs opérations.
- en restreignant l'utilisation de une ou plusieurs opérations.

Par exemple, l'opération `enlever` peut consister uniquement à supprimer l'élément de plus grande valeur. Cet appauvrissement du type `ensemble` facilite fortement son implémentation. Une solution souvent retenue est l'utilisation d'arbres binaires appelés "tas" dans lequel l'élément d'un noeud est supérieur aux éléments des fils. Ainsi, l'élément minimal se trouve à la racine. Calculer l'élément maximal de l'ensemble consiste alors simplement à considérer l'élément se trouvant à la racine de l'arbre ! Nous verrons comment supprimer cet élément ou ajouter tout nouvel élément en temps logarithmique.

Forêts

D'autres variantes du type `ensemble` existent. Par exemple une qui permet de manipuler une partition d'un ensemble donné en n'autorisant que les unions de ces parties. Les arbres sont une nouvelle fois utilisés et permettent (pour certains algorithmes) de réaliser l'union de deux parties à partir de deux de leurs éléments en temps (quasi) constant !

Chapitre 2

Types abstraits

Intuitivement un type abstrait est un ensemble d'opérations pour lesquelles on définit une syntaxe et une sémantique. Plus formellement un type abstrait est la donnée :

- de sa *signature* décrivant la syntaxe du type, les noms des types utilisés pour sa définition) ainsi que le nom des opérations et le type de leurs arguments.
- d'un ensemble d'*axiomes* définissant les propriétés des opérations.

La définition d'un tel ensemble d'axiomes est un exercice parfois délicat car cet ensemble doit être à la fois :

- *consistant* c'est à dire les axiomes ne doivent pas être contradictoires.
- *complet*, c'est à dire les axiomes sont suffisants pour décrire l'ensemble des propriétés du type abstrait.

Exemple 1
Ainsi, le type primitif `booléen` peut ainsi être défini :

Nom `Booléen`
Opérations

`vrai`	:		→ `booléen`
`faux`	:		→ `booléen`
¬	: `booléen`		→ `booléen`
∧	: `booléen`	× `booléen`	→ `booléen`
∨	: `booléen`	× `booléen`	→ `booléen`

Les axiomes sont :

```
¬(vrai())=faux()
¬(faux())=vrai()
∧(vrai(),vrai())=vrai()
∧(vrai(),faux())=faux()
etc ...
```

Lors de la définition d'un type abstrait, il est recommandé pour des raisons de simplicité de limiter le nombre d'opérations. Dans le cas présent, on démontre

aisément que toute opération booléenne peut s'écrire sous forme normale disjonctive, c'est à sous la forme d'une disjonction (opérateur "OU") de conjonctions (opérateur "ET") de littéraux positifs (variable simple) ou négatifs (variable précédée de l'opérateur $\neg$). Par exemple, l'opération $\Rightarrow$ se définit par $A \Rightarrow B := \neg(A) \vee B$; l'opération $\Leftrightarrow$ se définit par $A \Leftrightarrow B := ((A) \wedge B) \vee (\neg(A) \wedge \neg(B))$.

Remarque 1 Le plus célèbre des théorèmes en informatique est le théorème dit d'incomplétude de Gödel qui indique qu'aucun ensemble fini d'axiomes n'est suffisant pour prouver l'ensemble des propriétés du type `entier`. Ainsi, formellement il est impossible de définir un type abstrait `entier` qui soit universel. Cependant, rassurez-vous. Le type `entier` ou tout autre type que vous serez amenez à définir doit l'être dans le contexte de quelques algorithmes à écrire dont la correction ou la terminaison nécessite un nombre fini de propriétés sur ces entiers, et donc, d'un nombre fini d'axiomes.

2.1 Un exemple de type abstrait : le type ensemble

Définir un type abstrait est réalisé en fonction du contexte, c'est à dire en fonction du problème à résoudre voire de l'algorithme le résolvant. Définir un type abstrait consiste à choisir un ensemble d'opérations. Réaliser un tel choix nécessite un arbitrage entre différents critères :

1. faire des choix conformes avec des définitions courantes.
2. simplicité de la définition du type : que ce soit en ce qui concerne le nombre d'opérations, ou la définition des axiomes.
3. simplicité de la définition de l'algorithme.
4. optimisation de l'algorithme en ce qui concerne sa complexité en temps ou en espace.

2.1.1 Quelles opérations choisir ?

Considérons le type ensemble. Pour des raisons de simplicité, l'ensemble de ces opérations doit être minimal tout en étant suffisant pour permettre l'écriture de l'algorithme. Une définition universelle semblerait être :

Nom `ensemble`
Utilise `élément , booléen`
Opérations

```
estVide      : ensemble            → booléen
ensembleVide :                     → ensemble
appartient   : élément × ensemble → booléen
ajouter      : élément × ensemble → ensemble
```

```
enlever      : élément × ensemble → ensemble
choisir      : ensemble           → élément
```

Considérons un algorithme qui nécessite de calculer régulièrement la cardinalité d'un ensemble. Cette opération peut être défini à l'aide des opérations précédentes :

```
fonction cardinalité(E:ensemble):entier

i ← 0 ;

tantque ¬(estVide(E))
    i ← i+1 ;
    E ← enlever(choisir(E),E) ;

retourner i
```

Cependant, la complexité en temps de l'exécution de cette fonction (ici linéaire) peut s'avérer excessive alors que cette opération peut s'exécuter en temps constant par la simple présence dans la structure implémentant le type ensemble d'un entier indiquant sa cardinalité. Aussi, dans le cas où l'algorithme considéré requiert de calculer souvent la taille de l'ensemble ou, plus encore, si sa complexité en temps dépend de l'exécution de `cardinalité` en temps constant, il est préférable d'ajouter l'opération `cardinalité : ensemble -> entier` dans la définition même du type.

2.1.2 Quels axiomes choisir ?

Que signifient les opérations suivantes ?

```
ajouter      élément×ensemble → ensemble
enlever      élément×ensemble → ensemble
```

Quatre significations s'offrent à nous selon que l'on accepte ou non l'exécution de l'opération `ajouter` sur un élément contenu dans l'ensemble et selon que l'on accepte ou non l'exécution de l'opération `enlever` sur un élément n'appartenant pas à l'ensemble.

Exemple 2 Supposons que nous souhaitions écrire un algorithme réalisant l'union de deux ensembles. Si l'opération `ajouter` peut s'exécuter sur un élément déjà présent dans l'ensemble, l'algorithme s'écrit simplement :

```
fonction union1(A,B:ensemble):ensemble

  tantque ¬(estVide(B))
       b ← choisir(B) ;
       B ← enlever(b,B) ;
       A ← ajouter(b,A) ;
```

```
  retourner A ;
```

Si l'opération `ajouter` ne peut pas s'exécuter sur un élément déjà présent dans l'ensemble, l'algorithme s'écrit simplement :

```
fonction union2(A,B:ensemble):ensemble

  tantque ¬(estVide(B))
       b ← choisir(B) ;
       B ← enlever(b,B) ;
       si ¬(appartient(a,A)) alors
           A ← ajouter(b,A) ;

  retourner A ;
```

Si l'on considère les deux critères que sont la simplicité de l'algorithme et sa complexité en temps la préférence va à la première solution : le code est plus simple, la complexité en temps est au moins aussi bonne que la seconde.

Exemple 3 Supposons que l'on souhaite ne retenir d'un ensemble que les entiers pairs. Une solution à ce problème est :

```
fonction impairs(A:ensemble):ensemble

  B ← ensVide() ;

  tantque ¬(estVide(A)
       a ← choisir(A) ;
       A ← enlever(a,A) ;
       si estPair(a) alors
           B ← ajouter(a,B) ;

  retourner B ;
```

Ici, nous savons que tout élément candidat à être ajouter dans l'ensemble ne peut pas y appartenir. Nous avons donc tout intérêt à définir l'opération `ajouter` en interdisant l'ajout d'un élément déjà présent, et ce pour au moins deux raisons :

1. en restreignant la puissance opératoire de `ajouter`, son implémentation peut être moins coûteuse en temps. Si l'ensemble est représentée par une liste sans répétition, l'ajout d'un élément non présent est fait en temps constant : on l'insère en première position. Par contre, l'ajout d'un élément éventuellement déjà présent nécessite de vérifier sa présence avant de l'y insérer.
2. en restreignant la puissance opératoire de `ajouter`, on détecte d'éventuels erreurs dans l'écriture d'un exécutable. Vous pourrez demander lors de

l'exécution de votre programme de vous signaler toute tentative d'ajouter un élément déjà présent et utiliserez ce signal pour détecter une erreur soit dans le programme soit dans l'algorithme lui-même.

2.1.3 Écriture formelle des axiomes

Définir le sens des opérations est réalisé à l'aide d'axiomes, c'est à dire de formules logiques. Contrairement à une définition mathématique où l'on restreint la portée d'une fonction en restreignant les domaines des arguments, lors d'une définition axiomatique on restreint la portée d'une fonction en définissant le sens que pour des valeurs appartenant à des domaines précis.

Exemple 4 Considérons l'opération `enlever` que l'on autorise à opérer sur un élément b n'appartenant pas éventuellement à un ensemble A.

L'opération ensembliste mathématique sous-jacente est la fonction qui à tout ensemble A et à tout élément a associe l'ensemble à $A \cup \{a\}$.

Sa définition axiomatique est :

```
∀a : element ∀b : element ∀A : ensemble
appartient(a,ajouter(b,A)) = (a = b) ∨ appartient(a,A)
```

Exemple 5 Considérons l'opération `ajouter` que l'on autorise à opérer sur un élément b appartenant nécessairement à un ensemble A.

L'opération ensembliste mathématique sous-jacente est la fonction qui à tout ensemble A et à tout élément $a \notin A$ associe l'ensemble noté $A \backslash a$ égal à $A - \{a\}$. Le mathématicien considèrera que l'expression $\{a\} \backslash a$ est incorrecte.

La définition axiomatique de `enlever` est :

```
∀a : element ∀b : element ∀A : ensemble
      ¬(appartient(b,A)) ⇒
      ( appartient(a,ajouter(b,A)) = (a = b) ∨ appartient(a,A) )
```

L'axiomatique garantit des propriétés dans la mesure où les spécifications des fonctions ont été respectées : ici, on ne peut ajouter un élément à un ensemble qu'à la condition qu'il n'y appartienne pas. Si cette condition n'est pas respectée, l'algorithme peut s'exécuter mais sa correction n'est pas garantie.

Par exemple, l'informaticien admettra éventuellement que l'algorithme suivant puisse s'exécuter :

```
fonction test(): booléen
  A ← ensembleVide() ;
  A ← ajouter(1,A) ;
  A ← ajouter(1,A) ;
  A ← enlever(1,A) ;
  retourner appartient(1,A) ;
```

Cependant, il ne garantira pas que le booléen évaluant l'expression `appartient`$(1, A)$ soit faux !

Ceci n'est pas une coquetterie obscure. Démontrons qu'une implémentation correcte, naturelle et efficace du type `ensemble` a pour conséquence que `test`() retourne `vrai`.

Implémentons le type `ensemble` à l'aide d'une liste d'éléments sans répétition et ce sans se soucier de l'ordre des éléments. Une implémentation efficace de `ajouter`(a, A) est d'insérer a au premier rang de la liste (représentant) A. Une implémentation efficace de `enlever`$(1, A)$ est de supprimer la première occurrence de 1 dans la liste A : puisque aucune répétition n'est admise dans la liste, il n'est pas utile de supprimer toutes les occurrences de 1 mais seulement la première. Ainsi, lors de l'exécution de `test` l'ensemble A est successivement représenté par :

()
(1)
(1, 1)
(1)

Le booléen retourné par l'appel de `test`() est ainsi `vrai`.

2.2 Quelques types abstraits "séquence"

Il existe plusieurs façons de manipuler une séquence. Nous en présenterons 5 principales qui forment autant de types abstraits :

- le tableau
- la structure
- la pile
- la file
- la liste

Cette liste n'est pas exhaustive. Vous serez parfois amené à en définir de nouvelles inspirés ou non de celles-ci. De même, vous serez parfois amené à enrichir l'une d'entre elles en ajoutant de nouvelles primitives.

2.2.1 Le Tableau

Le premier type abstrait est le type tableau. Les opérations "tableau" permettent en fait de manipuler des séquences :

- dont les élément sont de même type.
- dont la taille est fixée à la création ; on enrichit parfois le type tableau d'une ou plusieurs opérations permettant de modifier sa taille. Mais leurs utilisations doivent être rares. Si la longueur de la séquence doit souvent varier, il est souvent préférable de considérer une liste.

Signature

La signature du type tableau est la suivante :

Nom `Tableau`
Utilise `élément, entier`
Opérations

```
tableau : entier × élément              → tableau
taille  : tableau                       → entier
lire    : tableau × entier              → élément
changer : tableau × entier × élément    → tableau
```

La syntaxe étant peu pratique, nous en définirons une plus proche de celle utilisée dans différents langages de programmation. Ainsi,

- $r \leftarrow t[4]$ permet d'associer à une variable r la valeur de l'élément se trouvant à l'indice 4 du tableau t. Cette écriture se substitue donc à l'instruction $r \leftarrow \texttt{lire}(t, 4)$.
- $t[4] \leftarrow a$ permet de modifier la valeur de l'élément se trouvant à l'indice 4 du tableau t en lui attribuant la valeur de la variable a. Cette écriture se substitue donc à l'instruction $t \leftarrow \texttt{changer}(t, 4, a)$.

Axiomatique

Voici quelques axiomes définissant dans un langage logique l'ensemble des primitives :

$\forall$`i` $\in \mathbb{N}$ $\forall$`e : élément` `taille(tableau[i](e))=i`
$\forall$`i` $\in \mathbb{N}$ $\forall$`e : élément` $\forall$`j` $\in [1,$`i`$]$ `lire(tableau[i](e),j)=e`
$\forall$`t : tableau` $\forall$`e : élément` $\forall$`j` $\in [1,$`taille(t)`$]$ $\forall$`k` $\in [1,$`taille(t)`$] \setminus j$
`lire(changer(t,j,e),k)=lire(t,j,e)`
$\forall$`t : tableau` $\forall$`e : élément` $\forall$`j` $\in [1,$`taille(t)`$]$
`lire(changer(t,j,e),j)=e`

Exemples

La suite d'instructions suivantes :

```
t ← tableau[5](0) ;
t[4] ← 10 ;
t[3] ← 2·t[4] ;
```

a pour conséquence de créer un tableau d'entiers de taille 5 indicé de 1 à 5 dont la valeur est successivement :

```
      1   2   3   4   5
t = (0,  0,  0,  0,  0)
t = (0,  0,  0, 10,  0)
t = (0,  0, 20, 10,  0)
```

Remarque

Afin d'obtenir une définition simple cohérente du tableau avec celle de la liste, le premier indice d'un tableau `t` est l'entier 1. Ainsi le $i$ème élément (son rang) est l'élément d'indice i.

Ce choix diffère de celui fait en langage `C` : le premier élément d'un tableau a pour indice 0. La raison est de pouvoir manipuler adresse et indice aisément : en `C`, l'adresse de `t[i]` n'est autre que la somme de `t+i`.

Si vous le souhaitez, il est possible d'enrichir le type tableau en définissant une opération `tableau` qui permet de choisir comme premier indice d'un tableau n'importe quel entier positif ou nul, voire n'importe quel entier relatif, voire n'importe quel caractère (si vous décides de prendre pour indices des caractères latins : voir exercice de TD).

Remarque

Le terme tableau a souvent un autre sens : celui issu de langages de programmation tels que le langage C. En effet une implémentation très courante d'un tableau est d'utiliser un tableau, c'est à dire une zone mémoire où les éléments seront à des adresses correspondant à leurs rangs dans la séquence. Ainsi, en codant un entier sur 4 octets, la séquence-tableau $(23, 12, 34, 56, 3)$ sera codée sur $5 \cdot 4$ octets : si le premier élément 23 et à l'adresse 23002101, les éléments 12, 34, 56 et 3 seront aux adresses respectives 23002105, 23002109, 23002113 et 23002117.

La conséquence pratique de ce choix est pouvoir accéder à un élément partir de son rang (ou indice) en temps constant. Ainsi, quand on évoque un tableau on pense souvent à des opérations de lecture et d'écriture en temps constant.

2.2.2 La structure

Le second type abstrait "séquence" est le type structure. Les opérations "structurent" permettent en fait de manipuler des séquences :

- dont les élément sont de type différents.
- dont la taille est fixée à la création.

On dit qu'une structure est composée de champs qui ont chacun un nom et un type.

Signature

Soient n types abstraits $E_1, \ldots, E_n$ ainsi qu'un type "identifiant" Id, une définition du type structure peut être :

Nom `structure`
Utilise $Id, E_1, \ldots, E_n$

Opérations

`structure`	: $Id \times \ldots \times Id \times E_1 \ldots \times E_n$	$\rightarrow$ `structure`
`lire`$_1$	: `structure`	$\rightarrow E_1$
...		
`lire`$_n$	: `structure`	$\rightarrow E_n$
`changer`$_1$	: `structure`$\times E_1$	$\rightarrow$ `structure`
...		
`changer`$_n$	: `structure`$\times E_1$	$\rightarrow$ `structure`

La syntaxe étant peu pratique nous en définirons une autre. Ainsi :
- pour tout entier $i \in [1, n]$, $a \leftarrow s.id_i$ se substitue à $a \leftarrow$ `lire`$_i(s)$
- pour tout entier $i \in [1, n]$, $s.id_i \leftarrow a$ se substitue à $s \leftarrow$ `changer`$_i(s, a)$.

Axiomatique

Les axiomes du type structure sont laissés en exercice.

Exemples

La suite d'instructions suivantes :

```
t ← structure[abs,ord](1.0 , 2.0 ) ;
t.abs ← 3.0 ;
t.ord ← t.ord + t.abs ;
```

a pour conséquence de créer une structure `t` ayant deux champs d'identifiants `abs` et `ord` de types réels ; la structure est successivement égale à :

```
t = ( 1.0 , 2.0 )
t = ( 3.0 , 2.0 )
t = ( 3.0 , 5.0 )
```

2.2.3 La Pile

Une pile est une séquence dont toutes les opérations se font à une extrémité appelé la *tête*.

Signature

La signature du type pile est la suivante :

Nom `pile`
Utilise `élément`
Opérations

`estVide`	: `pile`	$\rightarrow$ `booléen`
`pileVide`	:	$\rightarrow$ `pile`
`tete`	: `pile`	$\rightarrow$ `élément`

```
empiler  : pile × élément     → pile
dépiler  : pile               → pile
```

Axiomatique

Voici quelques axiomes définissant dans un langage logique l'ensemble des primitives :

```
estVide(pileVide()) = vrai()
∀p : pile ∀e : élément      estVide(empiler(p,e)) = faux
∀p : pile ∀e : élément      tete(empiler(p,e)) = e
∀p : pile ∀e : élément      p = dépiler(empiler(p,e))
```

Exemples

La suite d'instructions suivantes :

```
p ← empiler(pileVide(),10) ;
a ← tete(p) ;
p ← empiler(p,20) ;
b ← tete(p) ;
p ← empiler(p,30) ;
c ← tete(p) ;
p ← dépiler(p) ;
d ← tete(p) ;
```

a pour conséquence de créer une pile `p` et un entier `a` de valeurs successivement égales à :

```
p = (10)          a = 10
p = (20,10)       b = 20
p = (30,20,10)    c = 30
p = (20,10)       d = 20
```

Remarques

En fonction des problèmes à résoudre ou des algorithmes à écrire, on peut ajouter ou remplacer des primitives par de nouvelles. À titre d'exemple, on peut supposer l'existence d'une opération fournissant la longueur de de la séquence :

```
taille  : pile → entier
```

ou une autre qui retire la tête de la pile :

```
extraireTête : pile → élément × pile
```

dont la définition algorithme pourrait simplement être :

```
fonction extraireTête(p:pile) : élément × pile

    retourner(tête(p),dépiler(p))
```

2.2.4 La File

Une file est une séquence dont toutes les opérations se font aux deux extrémités :
- la suppression et la lecture à la première extrémité.
- l'ajout à la dernière extrémité.

Signature

La signature du type file est la suivante :

Nom `file`
Utilise `élément, booléen`
Opérations

```
estVide    : file              → booléen
fileVide   :                   → file
1erElément : file              → élément
défiler    : file              → file
enfiler    : file× élément     → file
```

Axiomatique

Voici quelques axiomes définissant dans un langage logique l'ensemble des primitives :

```
estVide(fileVide()) = vrai()
∀e : élément           1erElément(enfiler(fileVide(),e))= e
∀e : élément           estVide(défiler(enfiler(fileVide(),e))) = vrai()
∀f : file ∀e : élément
          non(estVide(f)) ⇒ défiler(enfiler(f,e))=enfiler(défiler(f),e)
∀f : file ∀e : élément
          estVide(enfiler(f,e))=faux()
```

Exemples

La suite d'instructions suivantes :

```
f ← enfiler(fileVide(),10) ;
a ← 1erElément(f) ;
f ← enfiler(f,20) ;
b ← 1erElément(f) ;
f ← enfiler(f,30) ;
c ← 1erElément(f) ;
f ← défiler(f) ;
d ← 1erElément(f) ;
```

a pour conséquence de créer une file `f` et des entiers de valeurs successivement égales à :

```
f = (10)          a = 10
f = (10,20)       b = 10
f = (10,20,30)    c = 10
f = (20,30)       d = 20
```

2.2.5 La Liste

Une liste est une séquence munie d'opérations qui permettent d'accéder à chacun des éléments à partir de leurs positions (ici un rang) et qui modifient la séquence à partir d'insertion ou de suppression.

Signature

La signature du type `liste` est la suivante :

Nom `liste`
Utilise `élément, entier, booléen`
Opérations

```
estListeVide: liste                        → booléen
listeVide   :                              → liste
ièmeElmt    : liste × entier               → élément
insérer     : élément × entier × liste     → liste
supprimer   : entier × liste               → liste
longueur    : liste                        → entier
```

Axiomatique

Voici quelques axiomes définissant dans un langage logique l'ensemble des primitives :

```
estListeVide(listeVide()) = vrai()
∀l : liste, ∀e : élément, ∀i ∈ ℕ
  1≤i≤longueur(l)+1 ⇒ taille(insérer(e,i,l)))= taille(l)+1
∀l : liste, ∀i ∈ ℕ
  1≤i≤longueur(l) ⇒ taille(supprimer(i,l)))= taille(l)-1
∀l : liste, ∀e : élément, ∀(i, j) ∈ ℕ²
 1≤j<i≤longueur(l)+1 ⇒ IèmeElmt(insérer(e,i,l),j) = IèmeElmt(l,j)
 1≤i=j≤longueur(l)+1 ⇒ IèmeElmt(insérer(e,i,l),j) = e
 1≤i<j≤longueur(l)+1 ⇒ IèmeElmt(insérer(e,i,l),j) = IèmeElmt(l,j-1)
∀l : liste, ∀(i, j) ∈ ℕ²
 1≤j<i≤longueur(l)   ⇒ IèmeElmt(supprimer(i,l),j) = IèmeElmt(l,j)
 1≤i≤j≤longueur(l)-1 ⇒ IèmeElmt(supprimer(i,l),j) = IèmeElmt(l,j+1)
```

Exemples

La suite d'instructions suivantes :

```
l ← insérer(10,1,listeVide()) ;
l ← insérer(20,2,l) ;
l ← insérer(30,3,l) ;
l ← insérer(40,2,l) ;
a ← IèmeElmt(l,1) ;
b ← IèmeElmt(l,2) ;
l ← supprimer(3,l) ;
```

a pour conséquence de créer une file `l` et des entiers de valeurs successivement égales à :

```
l = (10)
l = (10,20)
l = (10,20,30)
l = (10,40,20,30)
a = 10
b = 40
l = (10,40,30)
```

2.3 Conclusion

Nous avons présenté dans ce cours des types comme des ensembles d'opérations définies logiquement. Nous verrons dans le prochain chapitre comment les implémenter à l'aide de deux types le type `tableau` et le type `structure` qui sont fourni par la plupart des langages de programmation.

Cette approche logique ne suffit pas, si nous souhaitons étudier la complexité en temps des algorithmes, il nous faut définir un modèle de calcul réaliste qui garantisse un faible coût en temps des primitives considérées. Ce sera l'objet d'un prochain chapitre.

Chapitre 3

Implémentations de types

Nous verrons ici quelques exemples d'implémentations d'objets.

3.1 Types primitifs et effets de bord

3.1.1 Types primitifs

Les types primitifs correspondent à des objets pouvant être codées sur des blocs mémoire de taille fixe. Ils permettent de représenter les quantités numériques que sont les booléens, les entiers, les rationnels, les réels. Conséquence de la taille fixe de leurs représentations en mémoire, toutes les opérations les concernant ($\vee$, $\neq$, $\wedge$, $+$, $\cdot$, , log, *sin*, etc...) sont supposés être de complexité en temps constant.

Nous supposerons admises leur définitions.

Insistons sur un point : tout type primitif est représenté à l'aide d'un nombre fixe d'octets. Ainsi, définir un type entier à l'aide d'un type primitif suppose que l'on se restreint à un nombre fini d'entiers (sur un octet l'ensemble d'entiers représentables est $[0, 2^8 - 1]$, sur 4 octets il s'agit de $[0, 2^{4 \cdot 8} - 1]$). En conséquence de quoi, si vous souhaitez manipuler des entiers n de taille non arbitrairement fixé, il est nécessaire de les représenter à l'aide d'un type non primitif, par exemple une liste de longueur $\simeq \log_2(n)$. Le coût des opérations ne pourra pas être considéré comme constant mais dépendra alors de la longueur des listes.

Seront considérés comme non primitifs les types `tableau`, `structure` et tous les types construits à partir de ceux-ci notamment tous les types permettant de représenter des ensembles.

3.1.2 Effets de bord

Nous avons présenté des types permettant de manipuler des ensembles d'objets qu'ils soient ordonnés ou non (ensembles, séquences, et bientôt arbres). Ces ensembles nécessitent trivialement un espace mémoire de taille non constante (contrairement aux booléens, "petits" entiers et "petits" réels).

Nous avons défini pour ces objets de grande taille des opérations permettant de les modifier (par exemple `dépiler` pour la pile, `ajouter` pour l'ensemble, `insérer` pour une liste).

Une alternative s'offre à nous :

Opérations de complexité en temps faible voire constant

Considérons l'exemple de l'opération `dépiler` du type `pile` et l'instruction :

```
p← dépiler(q) ;
```

Il est possible de fournir une implémentation en temps constant (voire section prochaine). Or en temsp constant, il est impossible de recopier l'espace mémoire utilisé par la pile `q`, éventuellement très grand ; en d'autres termes, la pile `p` utilise pour totalité ou presque totalité la zone mémoire de `q` et ce quelle que soit l'implémentation d'une pile et de la fonction `dépiler`. En conséquence de quoi, une modification ultérieure de `q` entrainera sûrement une modification de `p`.

Il est par exemple possible que après exécution de `p<-dépiler(p)` les piles `p` et `q` soient égales, ou sinon aient les mêmes éléments.

Ceci remet en cause la définition axiomatique définissant les piles. Un choix à peine contraignant pour garantir les axiomes est de prendre pour membre droit d'une instruction uniquement la variable prise en argument et d'imposer ainsi comme seules instructions possibles :

```
q← dépiler(q) ;
```

Nous verrons que sous cette hypothèse `dépiler(q)` peut s'exécuter en temps constant. Ainsi, l'algorithme suivant qui cherche la présence d'un élément `e` dans une pile `p`

```
fonction appartient(e:élément ; p : pile) : booléen

    tantque non(estVide(p))
        si tete(p)=e
            retourner vrai() ;
        p = dépiler(p)

    retourner faux()
```

a une complexité en temps $\Theta(n)$ (avec $n = taille(p)$) et en espace $\Theta(1)$. Nous définirons dans le prochain chapitre un modèle de calcul complet et verrons dans cet exemple que la pile `p` après exécution de `appartient(e,p)` n'est pas modifiée.

Si vous souhaitez créer une nouvelle pile `p` obtenue en dépilant une pile `q` il faut alors réaliser une copie préalable de `q` qui crée une pile ayant les mêmes éléments mais utilisant une zone mémoire disjointe de celle utilisée par `q`. La complexité en temps de cette opération est bien-sûr linéaire : c'est à dire proportionnelle au nombre d'octets nécessaire à la représentation de `q`. La suite d'instructions devient alors :

```
p← dépiler(copie(q)) ;
```

Sous ces conditions, puisque les zones mémoires associées à `p` et `q` ne partagent aucun octet, aucune modification de `p` n'entraîne de modification de `q` et inversement.

Opérations de complexité en temps linéaire

Une autre solution consisterait lorsque l'on définit l'opération `dépiler` de systématiquement réaliser une copie de l'ensemble (ici une pile) passé en argument. Cela aurait l'avantage de pouvoir exécuter l'instruction

```
p← dépiler(copie(q)) ;
```

sans craindre qu'une modification ultérieure de `p` ne modifie `q` (et inversement). L'intérêt est logique : les axiomes sont garantis. Mais l'utilité de par exemple `dépiler` est réduit.

Ainsi, l'algorithme suivant qui cherche la présence d'un élément `e` dans une pile `p`

```
fonction appartient(e:élément ; p : pile) : booléen

    tantque non(estVide(p))
        si tete(p)=e
            retourner vrai() ;
        p ← dépiler(p)

    retourner faux()
```

a une complexité en temps $\Theta(n^2)$ (avec $n = taille(p)$) et en espace $\Theta(n^2)$.

3.2 Tableau infini

En introduction de ce cours, nous avons vu comment l'objet "séquence" pouvait être représenté en mémoire selon une zone mémoire contiguë qui permet l'accès à une élément selon son rang en temps constant ou selon un chaînage avec un accès en temps non constant.

Contrairement au principe qui souhaite ne pas mélanger la définition du type abstrait de son implémentation, les usages courants en algorithmique font que tout type abstrait "tableau" désigne un type séquence implémenté selon une zone contiguë.

Un exemple fameux de type tableau est `tableauInfini` dont la signature est :

Nom `tableauInfini`
Utilise `entier, élément`
Opérations
`tableauInfini : élément` → `tableauInfini`

```
ième           : tableauInfini × entier           → élément
changer-ième   : tableauInfini × entier × élément → tableauInfini
```

Définir un tableau infini est fait simplement à l'aide d'une structure contenant deux champs :

1. un premier champ de nom `tab` de type `tableau`.
2. un second champ de nom `val` contenant la valeur de type `élément` apparaissant une infinité de fois.

La figure 3.1 fournit une définition du constructeur `tableauInfini`.

```
fonction tableauInfini(e: élément) : structure

  t ← tableau[e](100) ;
  retourner structure(tab,val)(t,e) ;
```

FIG. 3.1 – Définition de `tableauInfini`

La définition de `ième` ne pose aucune difficulté :

```
fonction ième(t:tableauInfini , i: entier): élément

  si i ≤ taille(t.tab) alors
      retourner t.tab[i]
  sinon
      retourner t.val
```

Voici une implémentation de `changerIème` :

```
fonction changerIème(t:tableauInfini,i:entier,e:élément):tableauInfini

  t ← extensionÉventuelle(t,i) ;

  t.tab[i] ← e ;

  retourner t
```

où `extensionÉventuelle` est ainsi défini :

```
fonction extensionÉventuelle(t:tableauInfini,i:entier):tableauInfini

  si i ≤ taille(t.tab)
      retourner t
  sinon
      ntaille ← stratégieExtension(taille(t.tab),i) ;

      u ← tableau(ntaille)(t.val) ;
```

```
pour i ← 1 à taille(t.tab)
     u[i] ← t.tab[i]

t.tab ← u ;

retourner t
```

Dés que l'on souhaite modifier la valeur d'un élément se trouvant au delà du tableau fini `tab`, on étend ce tableau à une taille au moins égale à cet indice. Trois stratégies définies par la fonction `stratégieExtension` se présentent alors :

1. Soit on fait du juste mesure. On étend le tableau exactement de ce que l'on a besoin. Plus formellement nous avons :

   ```
   fonction stratégieExtension(n:entier,i:entier):entier ;
       retourner i
   ```

2. On peut affiner cette méthode en l'étendant à l'indice désiré mais à l'indice immédiatement supérieur multiple d'une constante fixée, par exemple égale à 100. Plus formellement nous avons :

   ```
   fonction stratégieExtension(n:entier,i:entier):entier ;
      retourner ⌈i/100⌉ · 100
   ```

3. Soit on anticipe des besoins futurs de façon à limiter le nombre de telles extensions. Une stratégie efficace consiste à systématiquement doubler la taille du tableau. Plus formellement nous avons :

   ```
   fonction stratégieExtension(n:entier,i:entier):entier ;
      nt ← n ;

      faire
          nt ← nt · 2 ;
      jusqu'à nt ≥ i

      retourner nt ;
   ```

Exemple 6 Ainsi l'algorithme :

```
t ← tableauInfini(0) ;
pour i de 1 à n
  t ← changer-ième(t,i,i·2) ;
```

réalisera

1. n extensions selon la première stratégie. La complexité en temps cumulée est $\Theta(n^2)$. Ceci est toujours extrêmement coûteux. Cette stratégie doit être abandonnée.

2. $\frac{n}{100}$ extensions selon la seconde (dans le cas où la constante est 100). La complexité en temps cumulée est $\Theta(\frac{n^2}{100}^2)$. Cette stratégie est théoriquement identique à la première, mais peut être employée dans certains algorithmes.
3. $\log_2(n)$ extensions selon la troisième. La complexité en temps cumulée est $\Theta(2 \cdot n)$ car égal à $\Theta(1 + 2 + 2^2 + \ldots + 2^{\log_2(n)})$. Cette stratégie est pour cette raison préférable aux deux précédentes.

3.3 Représentation d'une pile par une zone mémoire contiguë

Une utilisation immédiate d'un tableau infini est l'implémentation d'une pile.

Nom `pile`
Utilise `élément`
Opérations

`pileVide` :		$\rightarrow$ `pile`
`estVide` :	`pile`	$\rightarrow$ `booléen`
`empiler` :	`pile` $\times$ `élément`	$\rightarrow$ `pile`
`depiler` :	`pile`	$\rightarrow$ `pile`
`tete` :	`pile`	$\rightarrow$ `élément`

Et ce sous la forme d'une structure ayant deux champs, ce tableau ainsi que l'indice de l'élément au sommet de la pile, l'indice étant égal à 0 si la pile est vide.

```
fonction pileVide():pileRéel
  t← tableauInfini(0.0)
  retourner structure(tab,ind)(t,0)
```

```
fonction empiler(p:pileRéel, r:réel):pileRéel
  p.ind ← p.ind + 1 ;
  p.tab ← changerIème(p.tab,p.ind,r) ;
  retourner p
```

Exercice 1 Écrire les autres primitives.

Remarque 2 Conformément au modèle présenté lors du dernier chapitre, les piles sont des types non primitifs. Ainsi, l'algorithme

```
fonction test():booléen
  p ← pileVide() ;
  q ← p ;
  q ← empiler(q, 1.5) ;
  retourner (estVide(q)) ;
```

retourne `faux`.

En effet, les objets de types non primitifs `p` et `q` partagent toujours la même référence donc désignent toujours le même objet égal à la fin à la pile contenant l'unique réel 1.5 :
l'affectation `q` $\leftarrow$ `p`; qui fait en sorte que les deux objets de types non primitifs `p` et `q` partagent la même référence, donc désignent le même objet, l'instruction suivante `q` $\leftarrow$ `empiler`(`q`, 1.5) ne modifie pas la référence de de `q` qui reste donc égale à celle de `p`.

Remarque 3 Si l'on souhaite à partir d'une pile `p` définir une pile `q` ayant les mêmes éléments mais dont aucune modification sur `p` n'entraîne de modifications sur `q`, il nous faut nécessairement réaliser une "copie" de la première et ce dans une zone mémoire disjointe de la seconde.

Exercice 2 Écrire une telle fonction `copie`.

3.4 Représentation d'une liste itérative à l'aide d'un chaînage

Une définition du type abstrait `liste` est la suivante :

Nom `liste`
Utilise `élément, entier`
Opérations

`estListeVide`	`: liste`	$\rightarrow$ `booléen`
`listeVide`	`:`	$\rightarrow$ `liste`
`ièmeElmt`	`: liste` $\times$ `entier`	$\rightarrow$ `élément`
`insérer`	`: élément` $\times$ `entier` $\times$ `liste`	$\rightarrow$ `liste`
`supprimer`	`: entier` $\times$ `liste`	$\rightarrow$ `liste`
`longueur`	`: liste`	$\rightarrow$ `entier`

3.4.1 Encapsulation du type élément dans un noeud

Une première idée qui sera réutilisée dans le cas d'objets plus complexes, tels les arbres, est de définir un nouveau type abstrait, le type `noeud`, qui encapsule le type élément. Cette définition complexifie la définition du type `liste` en nécessitant de définir le nouveau type abstrait `noeud`. Ce coût initial est largement compensée par la simplicité des algorithmes implémentant les différentes primitives comme nous le verrons plus loin. Voici la signature d'un tel type :

Nom `noeud`
Utilise `élément, booléen`
Opérations

`constructNoeud`	`: élément` $\times$ `noeud`	$\rightarrow$ `noeud`

```
?suivant       : noeud              → booléen
suivant        : noeud              → noeud
contenu        : noeud              → élément
changerCont    : noeud × élément    → noeud
changerSuiv    : noeud × noeud      → noeud
```

L'implémentation d'un tel objet est réalisé à l'aide d'une structure contenant deux champs :

1. un champ de nom `cont` de type `élément`.
2. un champ de nom `suiv` désignant le noeud suivant.

Voici l'écriture de quelques primitives :

```
fonction constructNoeud(e:élément,n:noeud):noeud

    retourner structure(cont,suiv)(e,n)

fonction changerSuiv(n:noeud,p:noeud):noeud

    n.suiv ← p ;
    retourner n
```

3.4.2 Sentinelle avant

Si l'on souhaite insérer un élément e en $i$ème position, deux cas apparaissent selon que i est égal à 1 ou non :
- si $i = 1$, le nouveau premier noeud est celui contenant cet élément e.
- si $i \neq 1$, le premier noeud est inchangé.

Cette singularité $i = 1$ complexifie cet algorithme. Il est facile de se douter que d'autres algorithmes le seront aussi.

Une seconde idée permet de simplifier l'écriture de nombreux algorithmes. Elle consiste à utiliser une "sentinelle avant" : l'idée étant de représenter en mémoire la liste $(3, 5)$ par une liste composée de trois éléments $(\#, 3, 5)$, la valeur du premier élément n'ayant aucune importance quant à la valeur de la liste représenté : cet élément s'appelle la *sentinelle avant*.

Ainsi, une liste sera représentée par un type structure contenant un unique champ de nom `sentinelleAvant` et désignant cette sentinelle avant.

Pour des raisons similaires, il est souhaitable de rajouter une sentinelle arrière. Supposons donc enrichi le type noeud de façon à pouvoir produire des noeuds sentinelles et à pouvoir tester si un noeud est une sentinelle. En d'autres termes, supposons les opérations :

```
sentinelle  :                → noeud
?sentinelle : noeud          → booléen
```

Ainsi, la fonction `listeVide` peut se définir ainsi :

```
fonction listeVide():liste
    n ← sentinelle() ;
    n ← changerSuivant(n,sentinelle()) ;
    retourner structure(sentinelleAvant)(n) ;
```

Une fonction très utile de nom `ièmeNoeud` permet à partir d'une liste et d'un entier i de retourner son ième noeud (le noeud sentinelle si $i = 0$). La fonction `ièmeNoeud` peut ainsi être définie :

```
fonction ièmeNoeud(l:liste;i:entier):noeud

   n ← l.sentinelleAvant ;

   pour i ← 1 à i faire
     n ← suivant(n) ;

   retourner n
```

Il en découle les définitions des primitives `ième` et `insérer` (Figure 3.2 et 3.3).

```
fonction ième(l:liste;i:entier):élément

   retourner contenu(ièmeNoeud(l,i)) ;
```

FIG. 3.2 – Fonction `ième`

```
fonction insérer(l:liste,i:entier,e:élément):liste

   prec ← ièmeNoeud(l,i-1) ;
   n ← constructNoeud[e,suivant(prec)](cont,suiv);
   prec ← changerSuiv(prec,n) ;

   retourner l
```

FIG. 3.3 – Fonction `insérer`

Exercice 3 Écrire la fonction `supprimer`.

Exercice 4 Indiquer comment singulariser un noeud comme noeud sentinelle. En d'autres termes, écrire les primitives `sentinelle` et `?sentinelle`.

3.4.3 Autre attribut de la liste : sa longueur

Si l'algorithme le requiert, pour simplifier ou optimiser l'algorithme nous pouvons définir de nouveaux attributs au type liste.

Nous pouvons par exemple implémenter le type liste de façon à ce que le calcul de la longueur se fasse en temps constant.

Pour cela :

1. nous enrichissons le type abstrait de la fonction : `longueur liste -> entier`.
2. nous ajoutons dans la structure implémentant la liste un champ `longueur` initialisé à 0 et modifier lors de toute modification de la liste.

La fonction `listeVide` peut se définir ainsi :

```
fonction listeVide():liste
    n ← sentinelle() ;
    n ← changerSuivant(a,sentinelle()) ;

   retourner structure(sentinelleAvant,longueur)(n,0) ;
```

Exercice 5 Écrire les autres primitives du type `liste`.

3.4.4 Un autre attribut de la liste : le curseur

L'exécution de la fonction `test` de la Figure 3.4 double la valeur de chaque élément entier d'une liste l ; il a une complexité en temps en $\theta(longueur(l)^2)$, puisque l'exécution de `iemeNoeud(l,i)` est égal à $\theta(i)$.

```
fonction test(l:liste) : liste

  n ← taille(l) ;

  pour i ← 1 à n
     p ← ièmeNoeud(l,i) ;
     p ← changerCont(p,2·contenu(p))

  retourner l
```

FIG. 3.4 – Parcours d'une liste

Certes, on aurait pu réécrire l'algorithme en parcourant les noeuds de proche en proche et obtenir un algorithme linéaire. Mais nous allons montrer qu'une meilleure implémentation permet d'obtenir une complexité en temps linéaire sans modifier l'algorithme.

Cette idée consiste à ajouter à l'implémentation de la liste un noeud curseur (nom du champ `curseur`) ainsi que son rang (nom du champ `rangCurseur`). Ce curseur est positionné sur le $i$ème noeud à chaque appel de la fonction `ièmeNoeud`. Ainsi, si le curseur est le noeud de rang i (le i-ième noeud) d'une liste l, la complexité en temps de l'exécution de `ièmeNoeud` sur l'entrée l et $i+1$ (voire i, $i+2$) est constant et non plus $\theta(i+1)$.

L'écriture de `ièmeNoeud` est alors :

```
fonction ièmeNoeud(l:liste;i:entier):noeud

   si l.rangCurseur ≤ i
        n ← n.curseur ;
        j ← n.rangCurseur ;
   sinon
        n ← l.sentinelleAvant ;
        j ← 0 ;

   tantque j < i faire
       n ← suivant(n) ;
       j ← j + 1 ;

   l.curseur ← n ;
   l.rangCurseur ← i ;

   retourner n
```

En conséquence de quoi, il est facile d'observer que la fonction `test` de la Figure 3.4 a une complexité en temps égale non pas à $\theta(longueur(l)^2)$ mais $\theta(longueur(l))$.

Exercice 6 Écrire les autres primitives du type `liste` selon que le type `noeud` présente un simple ou un double chaînage.

3.4.5 Implémentation du type liste

Implémenter le type `liste` se fait en utilisant un type structure composé des champs :

- `sentinelleAvant`
- `curseur`
- `rangCurseur`
- `longueur`

Les différentes fonctions se définissent ainsi :

```
fonction estVide(l : liste) : booléen

    retourner l.longueur = 0

fonction longueur(l : liste) : entier

    retourner l.longueur
```

```
fonction listeVide() : liste
    n ← sentinelle() ;
    n ← changerSuivant(a,sentinelle()) ;

    retourner structure(sentinelleAvant,curseur,indiceCurseur, longueur)
                        (n,             n,      0,            0);


fonction ièmeElmt(l : liste ; i : entier) : élément

    retourner contenu(ièmeNoeud(l,i))



fonction supprimer(l : liste ; i : entier) : liste

    n ← ièmeNoeud(l,i-1) ;
    n ← changerSuivant(n,suivant(n)) ;

    retourner l


fonction insérer(e : élément ; i : entier ; l : liste ) : liste

    n ← ièmeNoeud(l,i-1) ;
    ajout ← constructNoeud(e,suivant(n)) ;
    n ← changerSuiv(n,ajout) ;

    retourner l
```

3.4.6 Implémentation d'un nouveau type liste

Il est possible de repenser totalement la définition du type abstrait, c'est à dire du nombre et du sens de chacune des opérations en fonction des concepts introduits. Ceci permet d'écrire des algorithmes en utilisant des routines de bas niveau manipulant ici par exemple des curseurs et permettant d'évaluer très précisément la complexité de ces algorithmes.

Voici, un exemple de nouvelle signature :

Nom `nouvelleListe`
Utilise `noeud, entier, booléen`
Opérations

```
estVide     : nouvelleListe  → booléen
listeVide   :                → nouvelleListe
```

```
curseur     : nouvelleListe  → noeud
rangCurseur : nouvelleListe  → entier
debut       : nouvelleListe  → nouvelleListe
fin         : nouvelleListe  → nouvelleListe
avant       : nouvelleListe  → nouvelleListe
tropADroite : nouvelleListe  → booléen
```

où la signification des opérations est ainsi "vulgairement" défini :
- `début` (`fin`) positionne le curseur sur le premier (resp. dernier) noeud.
- `avant` avance le curseur.
- `tropADroite` indique si le curseur a débordé de la dernière position.

Exercice 7 Définir un type `noeud` permettant de d'implémenter en temps constant une opération `arrière:liste -> liste` qui permet de reculer le curseur.

Remarque

La liste d'opérations définissant le type `nouvelleListe` est fourni à titre d'exemple. On peut la modifier en ajoutant par exemple des opérations du type `liste` comme `ièmeElmt`, de nouvelles opérations comme `ièmeNoeud`, `avancerIèmeRang` etc... Il est inutile de vouloir définir une type liste universel car le nombre d'opérations serait trop élevé. Le choix de ces opérations est dicté par le problème à résoudre et l'algorithme à écrire.

3.5 Conclusion

Nous avons présenté quelques principales améliorations possibles au type abstrait ou à son implémentation. Une amélioration parfois utile est de pouvoir parcourir la liste dans les deux sens.

Cela se fait simplement en enrichissant le type noeud en permettant l'accès au noeud précédent par ajout des fonctions :

```
?prec       : noeud              → booléen
preced      : noeud              → noeud
changerPrec : noeud × noeud      → noeud
```

et d'ajouter dans la structure une nouveau champ `prec`.

Cette liste d'amélioration n'est pas exhaustive. Vu leur nombre, il est fastidieux d'intégrer toutes les fonctionnalités dans un même type. Le choix des fonctions accessibles et de leur implémentation doit être réalisé en fonction du problème à résoudre et de l'algorithme à écrire.

Chapitre 4

Arbres binaires

Une façon très courante de représenter un ensemble est d'utiliser non une séquence mais une arborescence, c'est à dire une structure bidimensionnelle accessible à partir d'un premier élément appelé la racine et qui permet d'accéder de façon récursive de père en fils à chacun des éléments d'une unique façon.

Cette structure récursive permet d'implémenter les ensembles en obtenant des primitives de complexité très faible (en temps logarithmique).

Afin de simplifier, nous nous intéresserons à une sous classe des arborescences, les arbres binaires.

4.1 Définition

Une définition simple d'un arbre binaire est sa définition récursive :

Définition 1 (Arbre binaire) Un *arbre binaire* T est :

soit l'arbre vide, noté $\emptyset$.

soit un triplet (r, g, d) où :
- r est un noeud, appelé la *racine* de T, noté $rac(T)$.
- g est un arbre binaire, appelé le *sous-arbre gauche* de T,noté $ga(T)$.
- d est un arbre binaire, appelé le *sous-arbre droit* de T, noté $dr(T)$.

Définition 2 (Père, frère,...) La racine d'un arbre T est le *père* de la racine du sous-arbre gauche (resp. droit) si celle-ci existe. Deux noeuds ayant même père sont déclarés *frères*. L'*ascendant* d'un noeud est son père ou un ascendant de celui-ci. La *distance* d'une racine à un noeud est le nombre d'ascendants de ce dernier. Une *feuille* est un noeud sans fils.

Définition 3 (Hauteur) La *hauteur* d'un arbre est la plus grande distance d'un noeud à la racine : un arbre ayant pour unique sommet sa racine est de hauteur nulle.

Une premier type abstrait découlant de cette définition est le suivant :

Nom `Arbre`
Utilise `élément, entier, booléen`
Opérations

`estVide`	`: Arbre`	`→ booléen`
`arbreVide`	`:`	`→ Arbre`
`arbreGauche`	`: Arbre`	`→ Arbre`
`arbreDroit`	`: Arbre`	`→ Arbre`
`racine`	`: Arbre`	`→ élément`
`consArbre`	`: élément×Arbre×Arbre`	`→ Arbre`

Ce premier type abstrait a l'avantage de sa simplicité, il pourra donc être utilisé dans certaines premières définitions d'algorithme. Mais, si l'on souhaite écrire des algorithmes avancées et évaluer leur complexité en temps et en espace, il souffre des mêmes limites rencontrées par le type abstrait `liste` à l'opposé de `nouvelleListe`.

4.2 Arbre binaire parfait

Définition 4 Un arbre binaire de hauteur est *parfait* si il est complètement rempli sur tous les niveaux, sauf parfois, le plus bas qui est rempli en partant de la gauche jusqu'à un certain point.

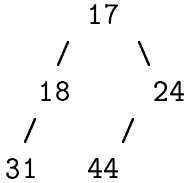
Supposons que nous ayons à implémenter des arbres binaires parfaits. Une solution serait de considérer n'importe quelle implémentation d'arbre binaire : qui peut le plus peut le moins. Ceci peut être une erreur car cette implémentation ne bénéficierait des singularités d'un arbre parfait.

Les seules modifications autorisées d'un arbre binaire parfait sont ou de modifier le contenu d'un noeud, ou de supprimer le dernier noeud (le plus à droite sur le dernier niveau) ou d'ajouter un noeud à la "suite" du dernier le plus à droite sur le dernier niveau (ou si celui-ci est rempli) d'en ajouter un le plus à gauche sur le nouveau dernier niveau.

Une implémentation consiste à utiliser un tableau dont la taille est égale à la taille n de l'arbre (c.a.d don nombre de noeuds) ayant pour ensemble d'indices $[1, n]$ de telle sorte que :

1. le noeud racine est l'indice 1.
2. le noeud fils gauche d'un noeud i est $2 \cdot i$.
3. le noeud fils droit d'un noeud i est $2 \cdot i + 1$.

Exemple 7 Ainsi l'arbre binaire parfait suivant :



a pour représentation le tableau suivant :

```
  1   2    3    4    5
-----------------------
|17| 18 | 24 | 31 | 44 |
-----------------------
```

On observe que le noeud d'élément 24 a pour fils gauche le noeud d'élément 45 ; leurs indices respectifs dans le tableau sont 3 et 6 et vérifient : $3 \cot 2 = 6$.

4.3 Implémentation chaînée d'un arbre binaire

Pour les mêmes raisons qui nous ont amené à définir une liste comme une séquence de noeuds mutuellement chaînés, nous définissons un arbre comme un ensemble de noeuds mutuellement chaînés.

Les définitions que nous présentons ici ne sont pas universelles, elles peuvent évoluer selon le problème à résoudre ou l'algorithme à écrire. À l'image du type `nouvelleListe`, la définition du type `arbre` repose sur la définition du type abstrait `noeud`.

Nom `arbre`
Utilise `noeud, entier, booléen`
Opérations

`estVide`	`: arbre`	$\rightarrow$ `booléen`
`arbreVide`	`:`	$\rightarrow$ `arbre`
`racine`	`: arbre`	$\rightarrow$ `noeud`
`prendreRacine`	`: noeud`	$\rightarrow$ `arbre`

Une définition du type `noeud` peut être la suivante :

Nom `noeud`
Utilise `élément, booléen`
Opérations

`contenu`	`: noeud`	$\rightarrow$ `élément`
`?filsGauche`	`: noeud`	$\rightarrow$ `booléen`
`?filsDroit`	`: noeud`	$\rightarrow$ `booléen`
`filsGauche`	`: noeud`	$\rightarrow$ `noeud`
`filsDroit`	`: noeud`	$\rightarrow$ `noeud`
`estSentinelle`	`: noeud`	$\rightarrow$ `booléen`
`constructSentinelle`	`:`	$\rightarrow$ `noeud`
`constructNoeud`	`: élément` $\times$ `noeud` $\times$ `noeud`	$\rightarrow$ `noeud`
`changerFilsGauche`	`: noeud` $\times$ `noeud`	$\rightarrow$ `noeud`
`changerFilsDroit`	`: noeud` $\times$ `noeud`	$\rightarrow$ `noeud`
`changerContenu`	`: noeud` $\times$ `élément`	$\rightarrow$ `noeud`

Afin de rendre plus lisible les algorithmes en définissant de nouvelles opérations obtenues en dégradant l'opération `constructNoeud`. Voici de nouvelles opérations possibles :

Utilise `élément, booléen`
Opérations(suite)

`changerFilsGauche`	`: noeud × noeud`	`→ noeud`
`changerFilsDroit`	`: noeud × noeud`	`→ noeud`
`changerContenu`	`: noeud × élément`	`→ noeud`

Le type défini ici ne permet en fait que de parcourir l'arbre de haut en bas (de la racine vers les feuilles). On peut souhaiter pouvoir accéder à partir d'un noeud à son père. Il nous faut alors rajouter les opérations suivantes :

Utilise `booléen`
Opérations(suite)

`?père`	`: noeud`	`→ booléen`
`père`	`: noeud`	`→ noeud`
`changerPère`	`: noeud × noeud`	`→ noeud`

et modifier bien sûr en conséquence le constructeur noeud `constructNoeud` dont la nouvelle signature doit être :

`constructNoeud : élément × noeud × noeud × noeud → noeud`

4.3.1 Définitions des primitives

Un arbre peut ainsi être défini à l'aide d'une structure composée d'un unique champ :

- de nom `racine` de type `noeud`.

Ce qui fournit la primitive suivante :

```
fonction racine(t : arbre) : noeud

    retourner t.racine ;
```

Dans le cas singulier de l'arbre vide (qui ne contient ni noeud, ni racine), une représentation simple est de le représenter à l'aide d'un noeud qui est une sentinelle :

```
fonction arbreVide() : arbre

    n ← constructSentinelle() ;

    retourner structure(racine)(n) ;

fonction estVide(t : arbre) : booléen
```

```
    retourner estSentinelle(t.racine) ;

fonction prendreRacine(n : noeud) : arbre

    retourner structure(racine)(n)
```

Un noeud peut être défini à l'aide d'une structure composée de cinq champs :
- un champ de nom `cont` de type `élément`
- trois champs de noms `gauche`, `droit`, `père` de type noeud.
- un champ de nom `sent` de type `booléen` indiquant si c'est une sentinelle.

Il en découle les définitions suivantes :

```
fonction constructNoeud(e : élément ; g,d,p : noeud ) : noeud

    retourner structure(cont,gauche,droit,père,sent)(e,g,d,p,faux())
fonction constructSentinelle() : noeud

    retourner structure(cont,gauche,droit,père,sent)(0,0,0,0,vrai())
fonction estSentinelle(n : noeud) : booléen

    retourner n.sent
fonction ?filsGauche(n : noeud) : booléen

    retourner estSentinelle(filsGauche) ;
fonction changerFilsDroit(n,droit : noeud) : noeud

    n.droit ← droit

    retourner n
```

Chapitre 5

Parcours d'arbres

Un parcours d'arbres est un algorithme qui permet de visiter chacun des noeuds de cet arbre. Il existe plusieurs façons de les parcourir. Nous en présentons ici deux. Ces techniques concernent en fait tous les arbres. En vue de simplifier cet exposé, nous les présenterons sur des arbres binaires.

La première est familière à une définition récursive de l'arbre. Il s'agit du parcours en "profondeur" qui consiste à définir le parcours d'un arbre comme le parcours de son sous-arbre gauche augmenté du parcours de son sous-arbre droit. Ce parcours défini ici récursivement admet une écriture algorithmique équivalente itérative utilisant une Pile.

La seconde consiste à parcourir les noeuds de l'arbre en traitant prioritairement les noeuds les plus proches de la racine. Ce parcours est appelé parcours en largeur. Nous verrons comment écrire itérativement un tel parcours à l'aide d'une file.

5.1 Parcours en profondeur

L'algorithme de parcours en profondeur est défini en utilisant une fonction

```
procédure parcoursArbreProfondeur(T:arbre)

    si non(estVide(T))
         parcoursProfondeur(racine(T));
```

auxiliaire parcourant récursivement les noeuds `parcoursProfondeur` (Figure 5.1).

L'algorithmique est générique et utilise trois fonctions auxiliaires `traiter1`, `traiter2`, `traiter3`. Afin de simplifier l'exposé, nous pouvons supposer que ces opérations ne réalisent pas de modifications de la structure de l'arbre et ne réalisent qu'un calcul local au noeud traité (éventuellement elles ne font rien).

```
procédure parcoursProfondeur(n:noeud)

    si estSentinelle(n) alors

        traiter4(n)

    sinon

        traiter1(n) ;

        parcoursProfondeur(filsGauche(n))

        traiter2(n) ;

        parcoursProfondeur(filsDroit(n))

        traiter3(n)
```

FIG. 5.1 – Parcours en profondeur récursif

Exemple 8 Sur l'exemple de l'arbre binaire de la Figure 5.2 (les sentinelles sont indifféremment nommées s) l'exécution de l'algorithme entraîne l'exécution des

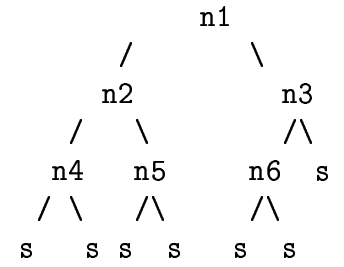


FIG. 5.2 – Un arbre binaire

instructions suivantes :

```
traiter1(n1) ; traiter1(n2) ; traiter1(n4) ; traiter4(s)  ;
traiter2(n4) ; traiter4(s)  ;
traiter3(n4) ;
traiter2(n2) ;
traiter1(n5) ; traiter4(s)  ;
traiter2(n5) ; traiter4(s)  ;
traiter3(n5) ; traiter3(n2) ;
traiter2(n1) ;
traiter1(n3) ; traiter1(n6) ; traiter4(s)  ;
```

```
traiter2(n6) ; traiter4(s)  ;
traiter3(n6) ;
traiter2(n3) ; traiter4(s)  ;
traiter3(n3) ; traiter3(n1) ;
```

5.1.1 Parcours préfixe, infixe ou suffixe

Il arrive parfois que le problème à résoudre nécessite un unique traitement sur chacun des noeuds (2 des trois traitements `traiter1`, `traiter2`, `traiter3` consistent à ne rien faire). Nous parlons alors de parcours :

préfixe si `traiter2` et `traiter3` sont des instructions vides.

infixe si `traiter1` et `traiter3` sont des instructions vides.

suffixe si `traiter1` et `traiter2` sont des instructions vides.

Exemple 9 Sur l'exemple de l'arbre précèdent, le parcours préfixe consistant à n'exécuter que les instructions `traiter1` provoque l'exécution de :

```
traiter1(n1) ; traiter1(n2) ; traiter1(n4) ;
traiter1(n5) ;
traiter1(n3) ; traiter1(n6) ;
```

Exemple 10 Sur l'exemple de l'arbre précèdent, le parcours infixe consistant à n'exécuter que les instructions `traiter1` provoque l'exécution de :

```
traiter2(n4) ;
traiter2(n2) ;
traiter2(n5) ;
traiter2(n1) ;
traiter2(n6) ;
traiter2(n3) ;
```

Exemple 11 Sur l'exemple de l'arbre précèdent, le parcours postfixe consistant à n'exécuter que les instructions `traiter1` provoque l'exécution de :

```
traiter3(n4) ;
traiter3(n5) ; traiter3(n2) ;
traiter3(n6) ;
traiter3(n3) ; traiter3(n1) ;
```

Exemple 12 Si nous souhaitons placer dans une file les contenus des différents noeuds en utilisant un parcours préfixe, l'algorithme de parcours s'écrit ainsi :

```
fonction parcoursArbre1(T:arbre):file

      retourner parcours1(racine(T),fileVide());
```

```
fonction parcours1(n:noeud,F:file):file

    si non(estSentinelle(n)) alors

        F ← enfiler(contenu(n),F) ;

        F  ← parcours1(filsGauche(n),F)

        F  ← parcours1(filsDroit(n),F)

    retourner F
```

FIG. 5.3 – Mise en séquence préfixe des noeuds

où `parcours1` est défini par la Figure 5.3.

L'exécution d'un tel algorithme sur l'arbre binaire de l'exemple 5.2 retourne la file `(n1,n2,n4,n5,n3,n6)`.

5.1.2 Écriture itérative d'un parcours en profondeur

L'exécution d'un algorithme récursif pouvant être coûteux en espace mémoire, il est souvent préférable de fournir une version itérative.

Une version itérative équivalente à l'algorithme `parcoursArbre` est celle décrite sur la Figure 5.4.

5.2 Parcours en largeur

Un second parcours d'un arbre consiste à traiter la racine, puis ses fils, ses petits fils et ainsi de suite. Ainsi, lors de ce parcours, sont traités les noeuds à distance 0 de la racine (la racine), puis ceux à distance 1, puis ceux à distance 2 et ainsi de suite.

La définition d'un tel parcours nécessite une file et peut être celle décrite par la Figure 5.5.

Exemple 13 Sur l'exemple de l'arbre binaire de la Figure 5.2, l'exécution de l'algorithme entraîne l'exécution des instructions suivantes :

```
traiter1(n1) ; traiter2(n1) ;
traiter1(n2) ; traiter2(n2) ;
traiter1(n3) ; traiter2(n3) ;
traiter1(n4) ; traiter2(n4) ;
traiter1(n5) ; traiter2(n5) ;
traiter1(n6) ; traiter2(n6) ;
```

```
procédure parcoursArbreProfondeurItératif(T:arbre)

     si non(estVide(T)) alors

         P ← empiler((racine(T),0),pileVide()) ;

         tantque non(estVide(P)) faire

             (n,i) ← tete(P) ;
             P ← depiler(P) ;

             si estSentinelle(n) alors

                 traiter4(n)

             sinon si i=0 alors

                 P ← empiler((n,3),P) ;

                 P ← empiler((filsDroit(n),0),P) ;

                 P ← empiler((n,2),P) ;

                 P ← empiler((filsGauche(n),0),P) ;

                 P ← empiler((n,1),P) ;

             sinon si i=1 alors

                 traiter1(n) ;

             sinon si i=2 alors

                 traiter2(n)

             sinon

                 traiter3(n)
```

FIG. 5.4 – Parcours en profondeur itératif

```
procédure parcoursLargeur(T:arbre)

     si non(estVide(T))
          F ← enfiler(racine(n),fileVide()) ;

     tantque non(estVide(F)) faire

          n ← premier(F) ;
          F ← defiler(F) ;

          traiter1(n) ;

          si non(estSentinelle(n)) alors

               F ← enfiler(filsGauche(n),F) ;
               F ← enfiler(filsDroit(n),F) ;

          traiter2(n) ;
```

FIG. 5.5 – Parcours en largeur

Chapitre 6

Arbres binaires de recherche

Les arbres binaires de recherche permettent d'implémenter de façon très efficace les multiensembles en permettant d'accéder à un élément en un temps qui dépend non de la cardinalité de l'ensemble mais de la hauteur de l'arbre qui le représente.

6.1 Définitions

Définition 5 Un *arbre binaire de recherche* (abrégé en A.B.R.) est un arbre binaire dans lequel la clé de tout noeud n

1. est strictement supérieure à la clé de tout noeud du sous-arbre gauche de n.
2. est strictement inférieure à la clé de tout noeud du sous-arbre droit de n.

Exemple 14 Un exemple d'arbre binaire de recherche est celui décrit par la Figure 6.1.

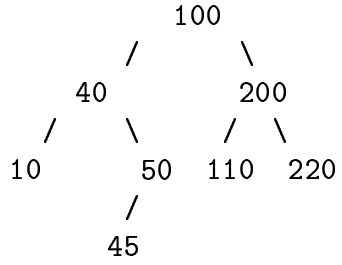


FIG. 6.1 – Un arbre binaire de recherche

Aussi, nous supposerons que le type noeud est enrichi d'une fonction qui lui fournit la clé :

```
clé : noeud                → entier
```

6.2 Le problème Recherche

Résoudre le problème de recherche :

```
Recherche
Entrée : un arbre T, une clé k
Sortie : un couple (b,n) formé
             du booléen b indiquant si T contient un noeud de clé k
             et si oui d'un tel éventuel noeud n
```

est résolu aisément à l'aide de la fonction `rechercheArbre` utilisant une fonction récursive `recherche` définies par la Figure 6.2.

```
fonction rechercheArbre(T:arbre,k:entier):booléen × noeud

    retourner recherche(racine(T),k) ;

fonction recherche(n:noeud,k:entier):booléen × noeud

    si estSentinelle(n)
        retourner (faux,_) ;

    sinon si k=clé(n)
        retourner (vrai,n) ;

    sinon si k < clé(n)
        retourner recherche(filsGauche(n),k);

    sinon
        retourner recherche(filsDroit(n),k);
```

FIG. 6.2 – Fonction `recherche`

Exercice 8 Prouver que `recherche` est correct, c'est à dire résout le problème `Recherche`.

Exercice 9 Prouver que la complexité en temps dans le pire des cas de `recherche` est $\Theta(h)$ où h est la hauteur de l'arbre T.

Exercice 10 Écrire de façon itérative l'algorithme `recherche`.

6.3 Insertion d'un nouveau noeud

Pour insérer un nouveau noeud `x` dans un arbre `T`, on parcourt l'arbre de façon similaire à l'algorithme `recherche` jusqu'à trouver un noeud `y` sans fils gauche

et de clé supérieure (resp. ou sans fils droit et de clé inférieure). À cet instant on place `x` fils gauche (resp. droit) de `y`.

Afin de simplifier l'écriture, on supposera que le noeud `n` a pour père, pour fils gauche et pour fils droit 3 noeuds sentinelles. L'algorithme `insérer` est décrit sur la Figure 6.3.

```
fonction insérer(T:arbre, n:noeud) :arbre

    si estVide(T)

        retourner changerRacine(T,n) ;

    sinon
        x ← racine(T) ;

        faire
            y ← x ;
            si clé(n) < clé(x)
                x ← filsGauche(x) ;
                àGauche ← vrai() ;
            sinon
                x ← filsDroit(x) ;
                àGauche ← faux() ;

        jusqu'à estSentinelle(x)

        si àGauche
            y ← changerFilsGauche(y,n) ;
        sinon
            y ← changerFilsDroit(y,n) ;

    retourner(T)
```

FIG. 6.3 – Insertion d'un noeud dans un A.B.R.

Exercice 11 Prouver la correction de `insérer`.

Exercice 12 Prouver que la complexité en temps dans le pire des cas de `insérer` est $\Theta(h)$ où h est la hauteur de l'arbre T.

6.4 Le problème du successeur

Définition 6 Dans un A.B.R., un noeud o est *successeur* d'un noeud n si aucun noeud de l'arbre ne possède une clé strictement comprise entre `clé(n)` et `clé(o)`.

Exemple 15 Dans le cas de l'arbre de l'exemple 6.1, le noeud de clé `50` a pour père le noeud de clé `100`.

Fait 4 Un noeux x a pour successeur y si et seulement si :
- x a un fils droit et y est le noeud de clé minimale dans le sous-arbre droit de x.
- x n'a pas de fils droit et y est le plus proche ancêtre de x dont le fils gauche est ancêtre de x.

preuve :
La preuve est laissée au lecteur.

La fonction suivante requiert une routine auxiliaire qui calcule le noeud de clé minimale parmi les descendants d'un noeud `n`, lui y compris. Cette routine est définie par la Figure 6.4.

```
fonction minimumDescendants(n:noeud): noeud

    tantque ?FilsGauche(n) faire
        n ← filsGauche(n) ;

    retourner(n)
```

FIG. 6.4 – Calcul du minmum parmi les descendants

Conséquence directe du Fait 4, la définition et la correction de l'algorithme qui calcule le successeur d'un noeud (Figure 6.5).

```
fonction successeur(T:arbre ; n : noeud) : noeud

    si ?FilsDroit(n)
        retourner minimumDescendants(filsDroit(n)) ;

    sinon
        p ← père(n) ;

        tantque non(estSentinelle(p)) et n ≠ filsGauche(p)
            n ← p ;
            p ← père(p) ;

        retourner(p)
```

FIG. 6.5 – Calcul du successeur

Exercice 13 Prouver que la complexité dans le pire des cas de `successeur` est en espace de $\Theta(1)$ et en temps de $\Theta(h)$ où h est la hauteur de l'arbre T.

6.5 Suppression d'un noeud dans un arbre binaire

Le problème qu'on souhaite résoudre ici est la suppression d'un noeud `n` dans un arbre binaire de recherche de façon naturellement à conserver un arbre binaire de recherche. Deux premiers cas simples se présentent alors :

1. `n` n'a aucun fils.
   La situation est évidente : on supprime le noeud `n`.
2. `n` a possède un seul fils.
   La situation est tout aussi évidente : on déclare l'unique fils de `n` fils du père de `n`. On réalise l'opération dit de *détachement*.

Écrivons dès à présent cette fonction `détacher` (Figure 6.7) qui prend pour argument un arbre binaire T et un noeud `x` (qui n'est pas une sentinelle) possédant au plus un fils et qui supprime dans l'arbre binaire le noeud `x` en plaçant son éventuel unique fils comme fils gauche (resp. droit) du père de `x` si `x` est fils gauche (resp. droit) de son père. Observons que cette fonction dans le cas où `x` n'a aucun fils, supprime simplement le noeud `x`.

Exemple 16 Un exemple de détachement est celui décrit par la Figure 6.6. Le noeud `50` est supprimé, son ancien père (le noeud `40`) prend pour nouveau fils droit l'ancien fils gauche du noeud `50` à savoir le noeud `45`.

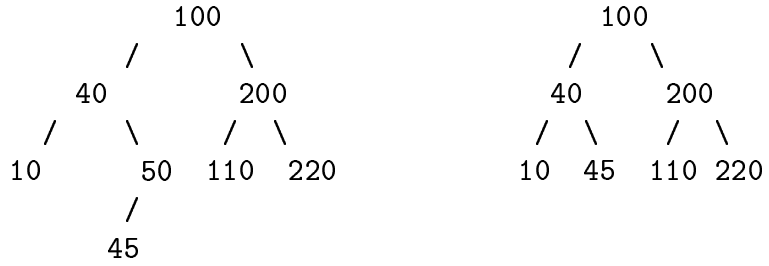


FIG. 6.6 – Détachement du noeud 50

En ce qui concerne le problème de suppression, un troisième cas se distingue :

(3.) `n` possède deux fils.
   On considère son noeud successeur `y`. Celui-ci est nécessairement un descendant de `n` (`n` possède un fils droit) et donc ne peut pas posséder de fils gauche. Il suffit alors que le père de `y` prenne pour nouveau fils le fils droit de `y` (opération de détachement de `y`) et que `y` prenne la place de `n` pour obtenir un arbre binaire de recherche.

```
fonction détacher(T : arbre; x : noeud) : arbre

    si ?FilsDroit(x)
        f ← filsDroit(x) ;
    sinon
        f ← filsGauche(x) ;

    si estRacine(T,x) alors
        T ← changerRacine(T,f) ;
    sinon
        p ← père(x) ;

        si filsGauche(p)=x
            p ← changerFilsGauche(p,f);

        sinon                    % on a filsDroit(p)=x
            p ← changerFilsDroit(p,f);

    retourner T ;
```

FIG. 6.7 – Algorithme de détachement d'un noeud

Exemple 17 Un exemple de suppression d'un noeud (le noeud 100) ayant deux fils est celui décrit par la Figure 6.8. Sur cet exemple, nous voyons que le noeud 100 est remplacé par son successeur (le noeud 110) et que celui est détaché (car n'ayant pas de fils gauche), son ancien père (le noeud 200 prend pour nouveau fils gauche l'ancien fils droit de 110 à savoir le noeud 130).

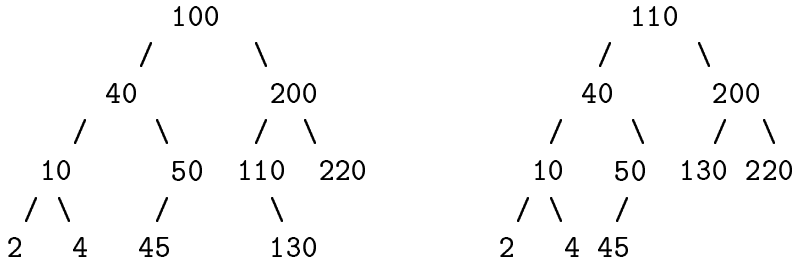


FIG. 6.8 – Suppression du noeud 100

L'algorithme de suppression est décrite dans la Figure 6.9.

```
fonction supprimer(T:arbre ; x : noeud): arbre

    si non(?FilsGauche(x)) OU non(?FilsDroit(x))

        retourner détacher(T,x) ;

     sinon

        s ← successeur(T,x) ;
        x ← changerContenu(x,contenu(s)) ;
        retourner détacher(T,s) ;
```

FIG. 6.9 – Algorithme de suppression d'un noeud

Remarque

Dans l'algorithme `détacher`, nous avons autoriser la comparaison de deux noeuds en évaluant l'expression `filsGauche(p)=x`. Or dans le type abstrait `noeud`, cette possibilité n'était pas offerte. Deux solutions s'offrent à nous :

1. on enrichit le type abstrait d'une opération permettant de tester l'égalité entre deux noeuds :

   `égalité : noeud × noeud` $\rightarrow$ `booléen`

2. on ramène l'égalité de deux noeuds en comparant leurs clefs (sachant que les clefs caractérisent les noeuds). En clair on remplace l'expression `filsGauche(p)=x` par `clé(filsGauche(p))=clé(x)`.

6.6 Conclusion

Nous avons vu que la complexité en temps des fonctions `insérer`, `supprimer` et `recherche` sont au plus égale à la hauteur de l'arbre.

Cette majoration est une bonne nouvelle, car non seulement il existe des arbres de grande taille n et de hauteur $\log_2(n)$ (par exemple des arbres binaires parfaits) mais de plus tous le sont ou presque : on démontre sans trop de difficulté qu'un arbre binaire de recherche construit aléatoirement de taille n est de taille $3 \cdot \log_2(n)$

Ainsi, "sans malchance", la complexité en temps est $\Theta(\log_2(n))$.

Ce propos malheureusement n'empêche pas des situations d'infortunes. En effet, si l'arbre de recherche est dégénérée c'est à dire de hauteur proche de sa taille (par exemple des arbres dans lequel chaque noeud a très souvent au plus un unique fils), la complexité est $\Theta(n)$.

Il existe de nombreuses techniques qui permettent de maintenir un arbre équilibré c'est à dire vérifiant $hauteur(T) = \log(taille(T))$. Les familles d'arbres

de recherches équilibrés sont d'arité :

- 2. Un exemple sont les arbres rouges et noirs.
- 3 : chaque noeud possède au plus 3 fils. Un exemple sont les 2,3-arbres.

Chapitre 7

Les arbres rouges et noirs

Nous présentons dans ce chapitre un ensemble de techniques permettant d'implémenter le type ensemble :

```
estVide       : ensemble            → booléen
ensembleVide  :                     → ensemble
choisir       : ensemble            → élément
appartient    : élément × ensemble  → booléen
ajouter       : élément × ensemble  → ensemble
enlever       : élément × ensemble  → ensemble
```

selon une complexité en espace $\Theta(1)$ et, en notant n la cardinalité de l'ensemble manipulé, selon une complexité en temps dans le pire des cas égale à :

```
estVide       : Θ(1)
ensembleVide  : Θ(1)
choisir       : Θ(1)
appartient    : Θ(log(n))
ajouter       : Θ(log(n))
enlever       : Θ(log(n))
```

Nous avons vu dans le dernier chapitre que la représentation d'un ensemble E à l'aide d'un arbre binaire de recherche T garantissait une complexité en temps égale à $\Theta(1)$ pour les trois premières fonctions et égale à $\Theta(hauteur(T))$ pour les trois dernières.

Observant que tout arbre binaire a une hauteur au moins égale à $\log(n)$ (plus exactement à $\lfloor \log(n) \rfloor$), pour obtenir une complexité en $\Theta(\log(n))$ il suffit de manipuler des arbres binaires de recherche T vérifiant

$$hauteur(T) = \Theta(\log(taille(T)))$$

Plusieurs familles de tels arbres existent, celle que nous présentons ici sont les arbres rouges et noirs.

7.1 Definition

Un *arbre rouge et noir* est un arbre binaire de recherche tel que :

1. chaque noeud est colorié ou en rouge ou en noir.
2. chaque sentinelle est colorié en noir.
3. si un noeud est rouge, ses deux fils sont noirs.
4. chaque chemin de la racine à une feuille contient le même nombre de noeuds noirs.

Fait 5 Tout arbre rouge et noir T vérifie :

$$hauteur(T) = \Theta(\log(taille(T)))$$

En fait nous avons : $\lfloor \log(taille(T)) \rfloor \leq hauteur(T) \leq 2 \cdot \log(taille(T)))$

preuve :
Pour tout arbre rouge et noir T, notons $noir(T)$ l'arbre (non binaire) obtenu à partir de T en coloriant en noir sa racine (voir Exercice 14) et en rattachant chaque fils d'un noeud rouge à son grand père nécessairement noir. Il vient :

$$taille(noir(T)) \leq taille(T)$$

Tout fils d'un sommet rouge étant noir, il vient :

$$hauteur(T) \leq 2 \cdot (hauteur(noir(T)))$$

Clairement, $noir(T)$ est un arbre dont tout chemin de la racine à l'un des feuilles est de longueur constante égale à $hauteur(noir(T))$ et dont chaque noeud possède au moins deux fils, il en découle :

$$hauteur(noir(T)) \leq \log(taille(noir(T)))$$

Ce qui suffit à conclure.

Exercice 14 Démontrer qu'en coloriant en noir la racine d'un arbre rouge et noir, on conserve un arbre rouge et noir.

7.2 Une opération locale : la rotation

Voici deux opérations qui transforment tout arbre binaire de recherche en un arbre binaire de recherche :

Définition 7 Soit `T` un arbre rouge et noir. Soit `y` un noeud dont le fils gauche `x` n'est pas une sentinelle. Nous notons `rotationDroite(T,y)` l'arbre obtenu à partir de `T` en prenant :

1. pour nouveau père de `x` l'ancien père de `y`,
2. pour nouveau fils gauche de `y` l'ancien fils droit de `x`,
3. pour nouveau fils droit de `x` le noeud `y`.

Soit `x` un noeud de `T` dont le fils droit `y` n'est pas une sentinelle. Nous notons `rotationGauche(T,x)` l'arbre obtenu à partir de `T` en prenant :

1. pour nouveau père de `y` l'ancien père de `x`,
2. pour nouveau fils droit de `x` l'ancien fils gauche de `y`,
3. pour nouveau fils gauche de `y` le noeud `x`.

Exercice 15 Écrire un algorithme de complexité en temps $\Theta(1)$ réalisant la rotation gauche.

Exercice 16 En reprenant les notations de la définition précédente, démontrer que l'arbre `T` est égal aux deux arbres :

```
rotationGauche(rotationDroite(T,y),x)
rotationDroite(rotationGauche(T,x),y)
```

Exercice 17 Démontrer que tout arbre binaire de recherche peut être transformé en une "chaîne droite" en au plus $n-1$ rotations. En déduire que tout arbre binaire de recherche peut être transformé en tout autre arbre binaire de recherche possédant les mêmes noeuds en au plus $2 \cdot (n-1)$ rotations.

7.3 Insérer un noeud

Le problème d'insertion :

```
problème RN-insérer
Entrée : un arbre rouge et noir T, un noeud x
Sortie : un arbre rouge et noir U dont l'ensemble des noeuds est
         celui de T augmenté de x
```

admet comme solution l'algorithme de la Figure 7.1. Afin de simplifier l'écriture nous utiliserons une procédure `colorationRouge` qui permet de colorier en rouge tout sommet (ainsi la notation `colorationRouge(n)` remplace `n<-colorationRouge(n)`. De plus nous utilisons de nouvelles primitives `frère`, `estFilsGauche` à signification évidente dont l'écriture est laissé en exercice.

Exercice 18 Écrire les algorithmes `frère`, `estFilsGauche`, `oncle`.

7.3.1 Correction sommaire

Pour prouver la correction de cet algorithme, il suffit de démontrer de l'invariance de la propriété suivante : l'arbre `T` est un arbre rouge et noir exception faite que le noeud `x` et son père peuvent être rouges.

En conséquence, à la sortie de boucle `tantque`, il est assuré que ou x est racine ou que son père est noir. Ainsi, l'arbre `T` retourné est rouge et noir.

7.3.2 Complexité

Il est assez immédiat d'observer qu'à chaque passage de boucle, la distance de `x` à la racine diminue strictement. La hauteur étant de l'arbre étant initialement $\log(n)$, la complexité en temps est $\Theta(\log(n))$.

7.4 Suppression

Le problème de suppression d'un noeud :

```
problème RN-supprimer
Entrée : un arbre rouge et noir T, un noeud y appartenant à T
Sortie : un arbre rouge et noir U dont l'ensemble des noeuds est
         celui de T privé de y
```

admet comme solution l'algorithme défini par les Figures 7.2 et 7.3.

7.4.1 Correction très sommaire

Conséquence du fait que la modification de l'arbre n'est réalisée qu'au travers de rotations gauches et droites, l'arbre conserve l'ensemble de noeuds initial et demeure un arbre binaire de recherche tout au long de l'algorithme.

Reste à démontrer que l'arbre retourné est un arbre rouge et noir. Cette preuve est présentée au tableau noir de l'amphithéâtre par utilisation de craies rouges.

7.4.2 Complexité

Infortunément à chaque passage de boucle, le sommet x ne se rapproche pas de la racine, ainsi dans le cas où son frère est rouge, une conséquence de la rotation est que, dans le nouveau noeud x peut rester à une même distance de la racine que l'ancien noeud x. Cependant, dans ce cas on est assuré que le nouveau noeud x a un frère noir, ce qui permet par le prochain passage de boucle de se rapprocher strictement de la racine.

Ainsi, après deux passages de boucles consécutifs, la distance de x à la racine décroît strictement. Toutes les instructions de l'algorithme `correction` étant de complexité en temps $\Theta(1)$, on déduite que `correction` est de complexité en temps $\Theta(\log(n))$.

On étend naturellement ce résultat à `détacher` et `RN-supprimer`.

Exercice 19 Est-ce que pour tout noeud `x` et pour tout arbre rouge et noir `T`, l'arbre `T` est égal à `RN-supprimer(RN-insérer(T,x),x)`.

```
fonction RN-insérer(T : RN-arbre ; x : noeud) : RN-arbre

    colorationRouge(x) ;

    colorationNoir(racine(T)) ;

    T ← insérer(T,x) ;

    tantque x ≠ racine(T) et couleur(père(x)) = rouge faire

        si estFilsGauche(père(x)) alors

            si couleur(oncle(x)) = rouge alors
                colorationNoir(père(x)) ;
                colorationNoir(oncle(x)) ;
                colorationRouge(père(père(x))) ;
                x ← père(père(x)) ;

            sinon
                si estFilsDroit(x) alors
                    T ← rotationGauche(T,père(x)) ;
                    x ← filsGauche(x) ;

                colorationNoir(père(x)) ;
                colorationRouge(père(père(x))) ;
                T ← rotationDroite(T,père(père(x))) ;

        sinon % même instruction en échangeant gauche et droite
    retourner(T) ;
```

FIG. 7.1 – Insertion dans un arbre rouge et noir

```
fonction RN-supprimer(T: RN-arbre ; y : noeud) : RN-arbre

    si ?FilsGauche(y) ET ?FilsDroit(y) alors
        y ← changerContenu(y,successeur(T,y)) ;
        y ← successeur(T,y) ;

    si ?FilsGauche(y) alors
        x ← filsGauche(y) ;
    sinon
        x ← filsDroit(y) ;

    T ← détacher(T,y) ;

    si couleur(y) = rouge alors
        retourner T
    sinon
        retourner correction(T,x) ;
```

FIG. 7.2 – Suppression dans un arbre rouge et noir

```
fonction correction(T: RN-arbre ; x : noeud) : RN-arbre

    tantque x ≠ racine(T) ET couleur(x) = noir faire
        si estFilsGauche(x) alors

            w ← frère(x) ;

            si couleur(w)=rouge alors
                colorationNoir(w) ;
                colorationRouge(père(x)) ;
                T ← rotationGauche(T,père(x)) ;
                w ← frère(x) ;

                                    % on a couleur(w)=noir

            si couleur(filsGauche(w))=noir ET couleur(filsDroit(w))=noir alors
                colorationRouge(w) ;
                colorationNoir(père(x)) ;
                x ← père(x) ;

            sinon

                si couleur(filsGauche(w)) = rouge alors
                    colorationNoir(filsGauche(w)) ;
                    colorationRouge(w) ;
                    T ← rotationDroite(T,w) ;
                    w ← filsDroit(père(x)) ;

                                    % on a couleur(filsGauche(w))=noir
                                    %   et couleur(filsDroit(w))=rouge

                si  couleur(père(x)) = noir alors
                    colorationNoir(filsDroit(w)) ;

                T ← rotationGauche(T,père(x)) ;

                x ← racine(T) ;

    retourner T
```

FIG. 7.3 – Suppression dans un arbre rouge et noir

Chapitre 8

Le type partition

Un objet commun en mathématique est la notion de partition d'un ensemble E. Nous montrons dans ce chapitre commet implémenter efficacement des partitions à l'aide d'arbres et obtenir des instructions de complexité en temps constante !

8.1 Préalables mathématiques

Une partition d'un ensemble E est un ensemble de parties de E non vides deux à deux disjointes (d'intersection vide) dont l'union forme E.

Rappelons que la notion de partition d'un ensemble est identique à la notion de relation d'équivalence ; une relation d'équivalence est une relation réflexive, symétrique et transitive :

1. toute partition P d'un ensemble E admet pour relation d'équivalence la relation $\sim$ qui relient deux objets appartenant à une même partie de E :

$$a \sim b :\Leftrightarrow \exists A \in P \{a, b\} \subseteq A$$

2. inversement, toute relation d'équivalence sur un ensemble E induit comme partition celle composée des classes d'équivalence :

$$P := \{[a]_\sim \mid a \in E\}$$

La *partition discrète* d'un ensemble E est la partition composée des singletons $\{\{a\} \mid a \in E\}$.

Nous appellerons union l'opération qui associe à une partition P d'un ensemble E et à deux éléments non équivalents a et b la partition obtenue à partir de P en réalisant l'union des classes de a et b.

8.2 Un type abstrait partition

Suite aux définitions précédentes, nous définissons le type abstrait suivant :

```
Type Partition
Utilise : booléen, ensemble
Opérations :
partitionDiscrete   : ensemble                          → partition
equivalent          : partition × élément × élément     → booléen
union               : partition × élément × élément     → partition
```

8.3 Quelques premières implémentations

8.3.1 À l'aide de séquences et de listes chaînées

Une première implémentation consiste à définir chacune des classes à l'aide d'une séquence.

Exemple 18 La partition $\{\{1,4,6\},\{2,5\},\{3\}\}$ se représente par exemple à l'aide de l'ensemble $\{(1,6,4),(2,5),(3)\}$.

Décider si deux éléments a et b sont équivalents nécessitent alors de tester l'appartenance de a à la séquence contenant b.

Si la séquence est représentée à l'aide d'une liste chaînée, ce test est de complexité en temps linéaire.

8.3.2 À l'aide d'étoiles (arbres de hauteur 1)

Pour réaliser le test d'équivalence en temps constant, il est nécessaire d'associer pour chaque élément en temps constant un objet qui caractérise la classe à laquelle il appartient.

Une solution est d'associer à chaque élément un élément singulier de sa classe d'équivalence, que nous appellerons son *représentant*. L'objet ainsi défini peut se définir de façon équivalente comme :

1. un objet mathématique : une fonction f de E dans E vérifiant $f \circ f = f$.
2. un objet informatique : un tableau à indices dans E à valeurs dans E.
3. une objet graphique : une étoile.

Exemple 19 Ainsi, la partition $\{\{1,4,6\},\{2,5\},\{3\}\}$ se représente par exemple à l'aide du tableau `T` :

```
1 2 3 4 5 6
4 2 3 4 5 4
```

Ainsi tester l'équivalence de deux éléments a et b se fait en temps constant : il suffit de tester l'égalité `T[a]=T[b]`.

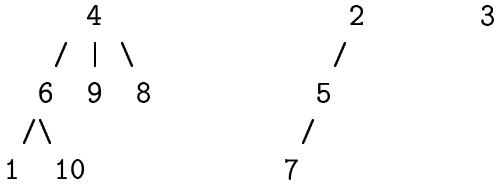
Malheureusement, si l'on souhaite réaliser l'union de deux éléments, il est nécessaire pour l'une des deux classes dont on souhaite faire l'union de modifier l'objet caractéristique de cette classe. Cette modification doit concerner chacun

des éléments de cette classe. Sa complexité en temps est proportionnelle à la cardinalité de cette classe et est donc linéaire.

8.4 Implémentation à l'aide d'arbres

Une idée consiste à considérer un objet plus complexe que l'étoile à savoir un arbre : ainsi toute classe est représentée à l'aide d'une arbre dont la racine est le représentant.

Exemple 20 Ainsi, la partition $\{\{1,4,6,8,9,10\},\{2,5,7\},\{3\}\}$ se représente par exemple à l'aide des trois arbres :



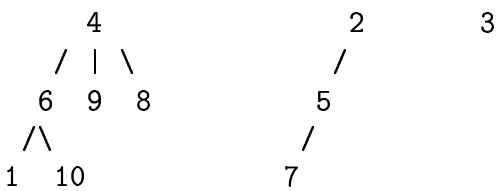
Calculer le représentant d'un élément nécessite alors de calculer la racine de l'arbre auquel il appartient.

Décider si deux éléments sont équivalents revient à calculer puis à comparer leurs représentants.

Réaliser l'union de deux éléments revient à calculer leurs représentants puis à placer l'un des deux représentants fils de l'autre représentant.

Ainsi, les opérations sur ces arbres ne consistent qu'à "remonter" dans l'arbre afin de calculer la racine ascendante d'un élément courant. Une représentation optimale d'un arbre est de le représenter à l'aide d'un simple tableau indiquant pour chaque élément son père (pour des raisons de simplicité, on considèrera le père de la racine égale à la racine elle-même).

Exemple 21 Ainsi, l'ensemble des trois arbres représentant la partition $\{\{1,4,6,8,9,10\},\{2,5,7\},\{3\}\}$



est représenté à l'aide du tableau :

1	2	3	4	5	6	7	8	9	10
6	2	3	4	2	4	5	4	4	6

8.4.1 Une première implémentation

Pour des raisons de simplification, nous supposerons que l'univers E forme un intervalle d'entiers de la forme $[1, i]$ avec $i \geq 0$. Ainsi, la signature de `partitionDiscrete` devient :

`partitionDiscrete : entier` $\rightarrow$ `partition`

En implémentant une partition comme une structure contenant pour seul champ un champ de nom pere de type tableau, nous obtenons :

```
fonction partitionDiscrete(n : entier) : partition

    tab ← tableau(n)(1) ;

    pour i de 1 à n faire
        tab[i] ← i

    p ← structure(pere)(tab) ;

    retourner p
```

Conformément à la section précédente, nous enrichissons le type abstrait de l'opération :

`représentant : partition` $\times$ `élément` $\rightarrow$ `élément`

qui a pour implémentation :

```
fonction représentant(p:partition ; a : élément):élément

    si p.pere[a] ≠ a alors
        retourner représentant(p, p.pere[a])
    sinon
        retourner p.pere[a]

fonction equivalent(p:partition ; a : élément ; b : élément): booléen

    retourner representant(p,a) = representant(p,b)

fonction union(p:partition ; a : élément ; b : élément): partition

    a ← representant(p , a) ;
    b ← representant(p , b) ;

    p.pere[a] ← b ;

    retourner p ;
```

La complexité en temps de `union` ou `equivalent` dépend de la complexité de `représentant` qui est exactement la *profondeur* dans l'arbre de l'élément a, à savoir la distance de celui-ci à la racine qui peut être égale à la taille de l'arbre.

Exercice 20 Dessiner la partition retourner par l'algorithme suivant. Démontrer que le programme suivant a une complexité en temps égale à $\Theta(n^2)$.

```
fonction test(n:entier) : partition

    p ← partitionDiscrete(n) ;

    pour i de 2 a n faire
        p ← union(p,1,i) ;

    retourner p
```

Exercice 21 Dessiner la partition retourner par l'algorithme suivant. Démontrer que le programme suivant a une complexité en temps égale à $\Theta(n)$.

```
fonction test(n:entier) : partition

    p ← partitionDiscrete(n) ;

    pour i de 2 a n faire
        p ← union(p,i-1,i) ;

    retourner p
```

8.4.2 Une seconde implémentation

Une première idée : transformation dynamique de l'arbre

Afin de diminuer ces complexités en temps, une première idée consiste chaque fois que l'on calcule le représentant r d'un élément a de profiter de ce calcul pour modifier l'arbre de façon à ce que le prochain calcul soit plus rapide. Dans notre cas, il suffit d'écraser l'arbre en faisant de a et de tous ses ascendants des fils de r.

Exemple 22 Ainsi, le calcul du représentant de l'élément `11` dans l'arbre de gauche modifiera l'arbre à gauche de la Figure 8.1 en l'arbre dessinée à droite :

L'algorithme permettant de calculer le représentant d'un élément est définie sur la Figure 8.2. Il nécessite une fonction auxiliaire permettant de calculer l'ensemble des noeuds ascendants d'un noeud donné (algorithme de la Figure 8.3).

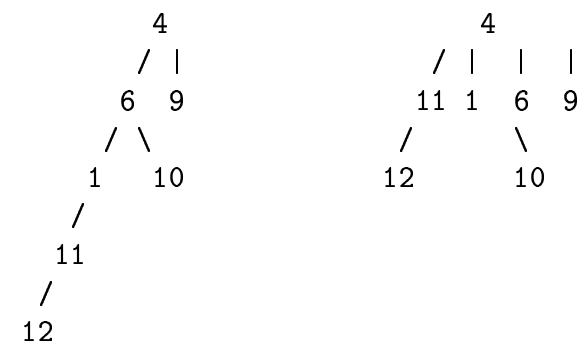


FIG. 8.1 – Un arbre écrasé dynamiquement

```
fonction représentant2(p:partition ; a : élément):élément × partition


    ascendants ← calculAscendants(p , a ) ;

    (r,ascendants) ← extraire(ascendants) ;

    tantque est(nonVide(ascendants)) faire
        (b,ascendants) ← extraire(ascendants) ;
        p.pere[b] ← r ;

    retourner (r,p) ;
```

FIG. 8.2 – Calcul du représentant

Une seconde idée : minimisation des hauteurs

Afin de d'obtenir des arbres de hauteur minimale, on enrichit la partition d'une information sur chacun des éléments appelée son *poids*, mesurant en fait l'importance du noeud racine. Ce poids n'est modifié que lorsque une racine admet pour nouveau fils un élément de même poids.

Les algorithmes deviennent :

```
fonction partitionDiscrete2(n : entier) : partition

    tab ← tableau(n)(1) ;
    poids ← tableau(n)(0) ;

    pour i de 1 à n faire
        tab[i] ← i

    p ← structure(pere,poids)(tab,poids) ;
```

```
fonction calculAscendants(p:partition ; a : élément) : pile

    ascendants ← empiler(a,pileVide()) ;

    tantque a ≠ p.pere[a] faire
        a ← p.pere[a]
        ascendants ← empiler(a,ascendants) ;

    retourner ascendants ;
```

FIG. 8.3 – Calcul des ascendants

```
    retourner p
fonction union2(p:partition ; a : élément ; b : élément): partition

    (a,p) ← representant2(p , a) ;
    (b,p) ← representant2(p , b) ;

    si p.poids[a] > p.poids[b] alors
        p.pere[b] ← a ;
    sinon
        p.pere[a] ← b ;

        si p.poids[a]=p.poids[b] alors
           p.poids[b] ← p.poids[b]+1 ;

    retourner p ;
```

8.4.3 Calcul des complexités

Complexité logarithmique

Il est aisé d'observer qu'un élément a fils d'un d'un élément b a un poids strictement inférieur. Ainsi, la hauteur d'un arbre n'excède pas le poids de sa racine.

On peut démontrer que tout arbre dont la racine est de poids p possède au moins 2^p éléments.

Conséquence de ces deux observations, la complexité en temps dans le pire des cas de `union2`, `représentant2` ainsi que `equiv` est $\Theta(\log(n))$.

Exercice 22 Démontrer tout arbre dont la racine est de poids p possède au moins 2^p éléments.

Complexité amortie (quasi) constante

En réalité, nous avons un résultat de complexité bien plus intéressant. Pour cela nous ne considérons pas la complexité dans le pire des cas de chaque exécution d'une des fonctions, mais nous considérons un programme globalement. Considérons un programme lors duquel est créée une partition de n éléments et lors duquel sont exécutés m instructions `union2`, `représentant2` ou `equiv`.

Il est alors possible de prouver (une preuve est disponible dans le chapitre 22 du Cormen) que le coût total de ces m instructions est $\Theta(m \cdot \log^*(n))$, ce qui ramène le coût (amorti) de chaque instruction à un coût (quasi) constant (voir la définition de $\log^*(n)$ ci-dessous).

Présentation succincte de $\log^*(n)$

Une définition précise de $\log^*(n)$ est disponible dans le Cormen. Disons simplement que $\log^*(n)$ est une fonction $\mathbb{N} \to \mathbb{N}$ croissante qui associe à tout entier de la forme $2^{(2^{(2^{\cdots^2})})}$ son nombre de niveaux. Nous avons : $\log^*(2) = 1$, $\log^*(2^2) = 2$, $\log^*(2^4) = 3$, $\log^*(2^{16}) = 4$, $\log^*(2^{65536}) = 5$, etc.

En conséquence de quoi, un entier x pour lequel $\log^*(x) \geq 5$ est au moins égal à 2^{65536} et est donc bien supérieur au nombre d'atomes présents dans l'univers observable (évalué à $10^{80} \simeq 2^{265}$).

Ainsi, d'un point de vue pratique, $\log^*(n)$ doit être considéré comme constant : nous ne manipulerons jamais une partition de cardinalité supérieure à 2^{65536}.

8.5 Utilisation de partition

Le premier ascendant commun de deux noeuds dans un arbre est l'ascendant commun de plus grand profondeur. On considère ici qu'un noeud est ascendant de lui-même.

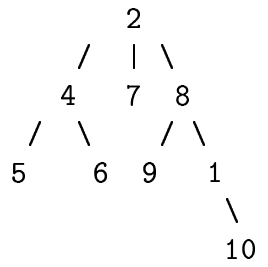
Si nous souhaitons calculer le premier ascendant commun à deux noeuds, il est nécessaire de remonter (éventuellement totalement) les deux chemins allant de ces deux noeuds en direction de la racine. La complexité est alors au moins la différence de la profondeur de cet ascendant avec celle du noeud le plus profond. Dans le pire des cas, cette complexité peut être la hauteur de l'arbre et donc la taille de l'arbre.

Le problème auquel nous nous nous intéressons est de calculer dans un arbre pour un ensemble de couples C le premier ascendant commun de chacun des couples. Si nous utilisons une méthode naive qui consisterait à pour chacun des couples à calculer le premier ascendant commun nous obtiendrons un algorithme en $\Theta(n \times m)$.

L'algorithme présenté ici est de complexité en temps optimale car linéaire (égal à $\Theta(n + m)$ où n est le nombre de sommets de l'arbre et m le nombre de

couples de C).

Exemple 23 Ainsi, si `T` est l'arbre dessiné ci-dessous :



et si `C` est l'ensemble $((4,7),(5,1),(10,9),(8,10))$, nous souhaitons calculer l'ensemble des triplets :

$$((4,7,2),(5,1,2),(4,9,2),(10,9,8),(8,10,8))$$

L'algorithme consiste en un parcours en profondeur de l'arbre selon un traitement postfixe. L'ensemble des noeuds traités sont coloriés en noir. Traiter un noeud u consiste à traiter ses fils, puis à calculer l'ascendant de u avec chacun des noeuds noirs souhaités. Afin de simplifier les notations, les variables `couleur`, `ascendant`, `partition`, `solution` sont considérées comme globales. L'algorithme est le suivant et est défini récursivement à l'aide de la fonction `parcours` (Figure 8.4).

```
fonction premierAscendant(T:arbre ; C : ensemble de couples):ensemble

    n ← taille(T) ;
    partition ← partitionDiscrete(n) ;

    ascendant ← tableau(n)(0) ;

    pour i de 1 à n faire
         couleur ← tableau(n)(blanc) ;

    solution ← ensembleVide() ;

    premierAscendantRec(racine(T)) ;

    retourner solution
```

La définition de `premierAscendantRec` colorie des sommets en gris ; cette coloration n'est pas utilise à l'exécution de l'algorithme (le seul test réalisé est de tester si un sommet est noir). Cet ajout permet une meilleure compréhension de l'algorithme : lors de l'étude la correction de l'algorithme, nous verrons que ces sommets gris forment une chemin de la racine au sommet couramment parcouru.

```
procédure premierAscendantRec(u : noeud)

    couleur[u] ← gris ;

    pour chaque fils  v de u dans T faire
        premierAscendantRec(v) ;
        partition ← union(partition,u,v) ;
        ascendant[representant[u]] ← u


    pour chaque noeud w tel que {u,w}∈C et couleur[w]=noir faire
        solution ← ajouter(solution,(u,w,ascendant[representant(w)]);
    couleur[u] ← noir ;
```

FIG. 8.4 – Calcul du premier ascendant

Correction

La correction de cet algorithme est la conséquence de l'invariance des propriétés suivantes :

1. si un noeud est gris, son père est gris.
2. si un noeud est noir son père est ou gris ou noir.
3. un noeud est noir si tous ses fils le sont.
4. un noeud admet au plus un fils gris.
5. le noeuds gris forment un chemin dont l'une des extrémités est la racine et dont la seconde est le noeud `u` parcouru.
6. pour tout noeud noir `w`, le noeud `ascendant[representant(w)]` est un ascendant de `w` qui est gris mais dont le fils qui est ascendant de `w` est noir.

Conséquence de ces propriétés, tout premier ascendant commun d'un noeud noir `w` et d'un noeud gris `u` est un noeud gris. Puisque le noeud `u` considéré dans l'algorithme est le prochain noeud colorié en noir, tout sommet gris est ascendant de `u`. Ainsi, le noeud gris `ascendant[representant(w)]` est un ascendant commun à `u` et `w` et est le premier puisque son unique fils qui est ascendant de `w` est noir et ne peut donc ainsi être un ascendant de `u`.

Complexité

Notons n le nombre de noeuds de l'arbre et m la cardinalité de l'ensemble des couples C.

Si l'on suppose que la complexité globale de toutes les exécutions de de l'instruction `pour chaque noeud w tel que {u,w}....` est égale à $\Theta(m)$ (voir exercice), toutes les instructions liées aux appels de fonction `union`, `representant`,

`couleur`, `ascendant` sont de complexité en temps constant. Le nombre d'appels récursifs de la fonction `parcours` est $\Theta(n)$. La complexité en temps est donc $\Theta(n+m)$. L'algorithme est donc linéaire.

Exercice 23 En choisissant une structure appropriée pour représenter l'ensemble C, faites en sorte que la complexité globale de toutes les exécutions de de l'instruction `pour chaque noeud w tel que {u,w}...` soit $\Theta(m)$.