Cours d'algorithmique en L1-MIEE

Serge Haddad
Professeur de l'Université Paris-Dauphine,
Place du Maréchal de Lattre de Tassigny
Paris 75775 cedex 16, France
adresse électronique : haddad@lamsade.dauphine.fr
page personnelle : www.dauphine.fr/∼haddad/

25 mai 2007

Table des matières

1 Introduction **3**
1.1 Problèmes et instances 3
1.1.1 Exemples 3
1.1.2 Taille d'une instance 6
1.2 Algorithmes 7
1.3 Complexité d'un algorithme 8
1.3.1 Temps d'exécution des instructions 8
1.3.2 Définitions 9
1.3.3 Illustration 9

2 Gestion dynamique des données : listes et piles **13**
2.1 Gestion d'annuaires 13
2.1.1 Principe d'un annuaire 13
2.1.2 Gestion à l'aide d'un tableau indicé par *id* 14
2.1.3 Gestion à l'aide d'un tableau séquentiel 15
2.1.4 Gestion à l'aide d'une liste 16
2.2 Le tri-fusion avec des listes 18
2.3 Gestion de piles 20
2.3.1 Principe d'une pile 20
2.3.2 Gestion d'une pile par un tableau 22
2.3.3 Gestion d'une pile par une liste 22
2.3.4 La pile d'exécution d'un programme 23

3 Gestion dynamique des données : tables de hachage **26**
3.1 Gestion d'un annuaire par table de hachage 26
3.1.1 Principe d'une table de hachage 26
3.1.2 Considérations de complexité 28
3.1.3 Etude de la complexité sous hypothèses probabilistes . . . 29
3.1.4 Choix de la fonction de hachage 31
3.2 Rappels d'arithmétique 31
3.3 Gestion d'un dictionnaire par table de hachage 33
3.4 Gestion d'un annuaire par « double » hachage 36
3.5 Gestion d'un dictionnaire par « double » hachage 37

4 Gestion dynamique des données : arbres binaires **40**
4.1 Gestion d'un annuaire par un arbre binaire 40
4.1.1 Principe d'un arbre binaire 40
4.1.2 Affichage trié d'un arbre binaire 43

4.1.3 Suppression dans un arbre binaire 43
4.1.4 Complexité des opérations 45
4.2 Le tri par tournoi 51
4.2.1 Représentation d'un arbre par un tableau 51
4.2.2 Représentation d'un tournoi par un « arbre-tableau » . . 51
4.2.3 Recherche des éléments par ordre décroissant 52
4.2.4 Complexité du tri tournoi 55
4.3 Arbre d'évaluation d'une expression 55
4.3.1 Principe d'un arbre d'évaluation 55
4.3.2 Evaluation à partir d'un arbre 55
4.3.3 Construction d'un arbre d'évaluation 57

Chapitre 1

Introduction

Objectifs. Introduire la notion de problème, d'instance de problème et de taille d'instance. Présenter un formalisme pour exprimer les algorithmes. Définir le temps d'exécution d'un algorithme puis sa complexité.

1.1 Problèmes et instances

1.1.1 Exemples

Planarité d'un graphe

Une instance du problème. Imaginons que dans le cadre de l'aménagement du territoire, on doive choisir le lieu d'implantation de trois usines et le lieu des sources de distribution du gaz, d'électricité et d'eau.

Le problème consiste à établir s'il est possible que le cablage et la tuyauterie allant des sources aux entreprises soit fait au même niveau du sous-sol et dans l'affirmative de produire un plan comprenant les usines, les sources et les liens.

La figure 1.1 décrit un plan qui conduit à un croisement et ne répond pas aux exigences énoncées.

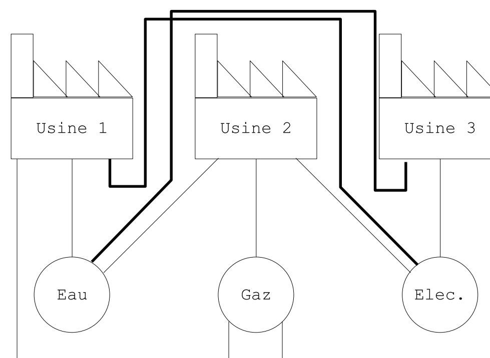

Fig. 1.1: Un plan de cablage

	Orange	Banane	Pomme		Apport requis
Vitamine A	30 mg	40 mg	120 mg		15 mg
Vitamine B	14 mg	14 mg	10 mg		10 mg
Vitamine C	250 mg	8 mg	44 mg		75 mg
Prix au kg	4 €	2 €	4.5 €		

TAB. 1.1: Informations nécessaires au régime

Le problème. On se donne n lieux à placer sur le plan et m couples de lieux qui doivent être liés. Le problème à résoudre consiste à déterminer, s'il est possible de placer les lieux et les liens sur le plan de telle sorte qu'il n'y ait aucun croisement et dans l'affirmative de produire un tel plan.

Programmation linéaire

Une instance du problème. Supposons qu'une personne doive suivre un régime nutritif en fruits qui lui garantit un apport quotidien suffisant en vitamines A, B et C.

Cette personne ne mange que trois fruits : des oranges, des bananes et des pommes. Pour chaque fruit, elle connaît son prix et la quantité de vitamines par kilo comme indiqué dans le tableau 1.1.

Le problème à résoudre consiste à déterminer une quantité (éventuellement fractionnaire) de chaque fruit de telle sorte que l'apport requis en vitamines soit atteint et que le prix du régime soit *minimal*.

De manière plus formelle, il s'agit de trouver parmi les triplet (xo, xb, xp) qui vérifient :

$$30 \cdot xo + 40 \cdot xb + 120 \cdot xp \geq 15$$

$$14 \cdot xo + 14 \cdot xb + 10 \cdot xp \geq 10$$

$$250 \cdot xo + 8 \cdot xb + 44 \cdot xp \geq 75$$

celui qui minimise $4 \cdot xo + 2 \cdot xb + 4.5 \cdot xp$.

Le problème. On se donne n éléments composés $\{comp_1, \ldots, comp_n\}$ et m éléments de base $\{base_1, \ldots, base_m\}$. La composition d'un élément $comp_i$ est donnée par $\{a_{j,i}\}_{1 \leq j \leq m}$ et son coût est donné par c_i. La quantité d'élément de base $base_j$ à se procurer est b_j. Le problème s'énonce ainsi. Trouver parmi les tuples $(x_1, \ldots, x_n)$ qui vérifient :

$$\forall 1 \leq j \leq m, \sum_{i=1}^{n} a_{j,i} \cdot x_i \geq b_j$$

celui qui minimise $\sum_{i=1}^{n} c_i \cdot x_i$.

Recherche de seuils

Une instance du problème. Supposons que le département de ressources humaines d'une entreprise maintienne un fichier des employés (déclaré à la CNIL). Ce fichier contient entre autres le salaire de chaque employé. Le département

souhaite établir quel est le seuil correspondant au 10% des employés les mieux payés. Si cette entreprise a 500 employés, cela revient à trouver quel est le 50ème plus gros salaire.

Le problème. Soit T un tableau de valeurs numériques (avec répétitions éventuelles) et k un entier inférieur ou égal à la taille de T. Le problème consiste à déterminer la $k$ième plus grande valeur du tableau et un indice du tableau contenant cette valeur.

Déduction automatique

Une instance du problème. Supposons connues les affirmations suivantes :

- *Tous les chats sont blancs ou noirs.*
- *Platon est un chat.*
- *Platon n'est pas noir.*
- *Aristote est blanc.*

On souhaiterait savoir si chacune des deux phrases suivantes est une conséquence des affirmations précédentes.

- *Platon est blanc.*
- *Aristote est un chat.*

Le problème. Etant donnée une logique (*e.g.*, logique propositionnelle ou logique du premier ordre), un ensemble de formules $\{\varphi_i\}_{1\leq i\leq n}$ de cette logique et une formule φ, le problème consiste à déterminer si φ se déduit (à l'aide des axiomes et des règles de déduction de cette logique) de $\{\varphi_i\}_{1\leq i\leq n}$.

Terminaison d'un programme

Une instance du problème. Supposons que nous ayons écrit un programme correspondant à l'algorithme 1 et qu'avant de l'exécuter, nous souhaitions savoir si son exécution se terminera.

Algorithme 1 : Se termine-t-il ?

```
x ← 1
while x ≠ 0 do
    if x%3 = 0 then
        x ← x + 5
    else
        x ← x − 2
    end
end
```

Le problème. Etant donné le texte d'un programme écrit par exemple en JAVA, le problème consiste à déterminer si ce programme se termine. Plusieurs variantes sont possibles. Ainsi si le programme comprend une entrée, le problème pourrait consister à déterminer si le programme se termine pour toutes les entrées possibles.

1.1.2 Taille d'une instance

L'algorithmique consiste à résoudre des problèmes de manière efficace. Il est donc nécessaire de définir une mesure de cette efficacité. Du point de vue de l'utilisateur, un algorithme est efficace si :

1. il met peu de temps à s'exécuter ;
2. il occupe peu de place en mémoire principale.

Cependant ces mesures dépendent de la taille de l'instance du problème à traiter. Il convient donc de définir la taille d'une instance. Plusieurs définitions sont possibles dans la mesure où une même instance peut s'énoncer de différentes manières. En toute rigueur, l'efficacité d'un algorithme devrait prendre en compte non pas l'instance mais sa représentation fournie en entrée de l'algorithme. Cependant la plupart des représentations *raisonnables* d'une instance conduisent à des tailles similaires. Plutôt que de formaliser cette notion, nous la préciserons pour chaque problème traité.

Si on prend comme unité de mesure le bit, nous commençons par faire quelques hypothèses.

- Le nombre de bits nécessaires pour représenter un caractère est constant (en réalité, il dépend de la taille de l'alphabet mais celle-ci ne change pas de manière significative). On le notera B_c.
- Le nombre de bits nécessaires pour représenter un entier est constant. Cette hypothèse n'est valable que si, d'une part on connaît *a priori* une borne supérieure de la taille d'un entier intervenant dans une instance et si, d'autre part les opérations effectuées sur les entiers par l'algorithme ne conduisent pas à un dépassement de cette borne[1]. On le notera B_e.
- Dans la plupart des systèmes d'information, les informations stockées dans des fichiers sont accessibles *via* des *identifiants* qui peuvent être des valeurs numériques (*e.g.* le n° de sécurité sociale) ou des chaînes de caractères (*e.g.* la concaténation du nom et du prénom). On supposera aussi que le nombre de bits nécessaires pour représenter un identifiant est constant et on le notera B_i.

Illustrons maintenant la taille d'une instance à l'aide des exemples précédents.

Planarité d'un graphe. Il suffit de représenter les liaisons, *i.e.* des paires d'identifiants. On obtient donc $2m \cdot B_i$. Cette représentation appelle deux remarques.

D'un point de vue technique, le programme implémentant l'algorithme doit savoir où se termine la représentation : soit par la valeur m précédant la liste des paires soit par un identifiant spécial (différent des identifiants possibles) qui suit la liste. Dans les deux cas, on a ajouté un nombre constant de bits.

D'un point de vue conceptuel, un lieu qui n'apparaît dans aucune liaison, est absent de la représentation. Vis à vis du problème traité, cela n'est pas significatif car si le plan a pu être établi, il suffit d'ajouter ces lieux hors de l'espace occupé par le plan.

[1] pour certains problèmes, comme ceux liés à la cryptographie, cette hypothèse doit être levée.

Programmation linéaire. On précise d'abord m et n, puis les éléments $a_{i,j}$ (ligne par ligne ou colonne par colonne), les élements b_j et finalement les éléments c_i. Ces nombres ne sont pas nécessairement des entiers. En faisant l'hypothèse que ce sont des rationnels on peut les représenter sous forme de fractions d'entier, *i.e.* par deux entiers[2]. On obtient comme taille du problème $(2 + 2m + 2n + 2m \cdot n) \cdot B_e$.

Recherche de seuils. On précise d'abord n la taille du tableau et k le seuil puis les cellules du tableau, ce qui nous donne (en supposant que les salaires soient des entiers) $(2 + n) \cdot B_e$.

Déduction automatique. On indique le nombre d'hypothèses puis les hypothèses suivi de la conclusion. *A priori*, les formules peuvent être de taille quelconque. En notant $|\varphi|$ le nombre de caractères d'une formule ϕ, la taille du problème est alors $B_e + (\sum_{i=1}^{n} |\varphi_i| + |\varphi|) \cdot B_c$.

Terminaison d'un programme. La taille de l'instance est $n \cdot B_c$ où n est le nombre de caractères du texte du programme.

Remarque importante. Nous nous intéressons au comportement des algorithmes sur des instances de grande taille. De manière encore plus précise, nous souhaitons estimer la nature de la variation du temps d'exécution de l'algorithme lorsque la taille de l'instance augmente. Aussi il est raisonnable de :

– conserver le terme prédominant de la taille d'une instance ;
– dans ce terme, remplacer les constantes multiplicatives par 1.

Ainsi pour le cas de la programmation linéaire, le terme prédominant est $2m \cdot n \cdot B_e$. Par conséquent, en oubliant les constantes, on prendra pour la taille d'une instance $m \cdot n$.

1.2 Algorithmes

Sauf mention du contraire, on considèrera que l'algorithme est défini par une fonction dont les entrées sont indiquées par le mot-clef **Input** et les sorties (il peut y en avoir plusieurs) sont indiqués par le mot-clef **Output**.

Les déclarations des variables locales de la fonction sont indiquées par le mot-clef **Data** dans le corps de la fonction après les entrées et les sorties et avant le bloc d'instructions de la fonction.

L'opérateur d'affectation est noté $\leftarrow$ à ne pas confondre avec le test d'égalité $=$ utilisé pour construire des expressions booléennes. On n'utilisera le « ; » que pour concaténer des instructions placées sur la même ligne. Les opérateurs de comparaison sont notés $=, \neq, <, >, \leq, \geq$, les opérateurs arithmétiques sont notés $+, *$ et les opérateurs logiques sont notés **and**, **or**, **not**.

Lorsque nous introduirons un opérateur spécifique à un nouveau type de donnée, nous en préciserons la signification. Pour l'instant, nous travaillerons avec des tableaux dont la dimension sera précisée lors de leur définition. Pour accéder à un élément du tableau, on utilise la notation $Tableau[indice]$.

Les constructeurs de programmes (avec leur interprétation usuelle) sont :

[2]Ceci ne préjuge pas de la représentation utilisée par le programme.

- **if** *condition* **then**
  instructions
  else if *condition* **then** (optionnel)
  ...
  else (optionnel)
  instructions
  end
- **while** *condition* **do**
  instructions
  end
- **repeat**
  instructions
  until *condition*
- **for** *indice* ← *valeur initiale* **to** *valeur finale* **do**
  instructions
  end

Lorsqu'un bloc correspondant à un constructeur tient sur une ligne, nous omettons le mot-clef **end**.

Pour renvoyer le résultat d'une fonction, on utilise le mot-clef **return** *résultats*. A l'inverse, l'appel d'une fonction se note `NomFonction`(*paramètres*).

L'algorithme 2, que nous étudierons plus loin dans ce chapitre, illustre ces conventions d'écriture.

Algorithme 2 : Tri d'un tableau

`Trie`(T, deb, fin) : tableau d'entiers

Input : T un tableau, $deb \leq fin$ deux indices du tableau
Output : un tableau contenant les valeurs de $T[deb]$ à $T[fin]$ triées

Data : T' un tableau de $fin + 1 - deb$ entiers
Data : T_1 un tableau de $\lfloor (fin + 1 - deb)/2 \rfloor$ entiers
Data : T_2 un tableau de $\lceil (fin + 1 - deb)/2 \rceil$ entiers

if $deb = fin$ **then**
  $T'[1] \leftarrow T[deb]$
else
  $T_1 \leftarrow$ `Trie`$(T, deb, \lfloor (deb + fin - 1)/2 \rfloor)$
  $T_2 \leftarrow$ `Trie`$(T, \lfloor (deb + fin - 1)/2 \rfloor + 1, fin)$
  $T' \leftarrow$ `Fusion`$(T_1, \lfloor (fin + 1 - deb)/2 \rfloor, T_2, \lceil (fin + 1 - deb)/2 \rceil)$
end
return T'

1.3 Complexité d'un algorithme

1.3.1 Temps d'exécution des instructions

On considèrera qu'une instruction d'affectation est exécutée en une unité de temps à condition que l'expression à évaluer ne comporte pas d'appel de fonctions et que la variable à affecter soit d'un type élémentaire. Dans le cas d'un appel de fonction, il faut ajouter le temps d'exécution des appels de fonction.

Dans le cas d'un type complexe, il faut tenir compte du type. Ainsi si on affecte un tableau de taille t, le temps d'exécution sera t.

On considère aussi que l'évaluation d'un test intervenant dans un constructeur s'effectue aussi en une unité de temps avec les mêmes restrictions que pour l'affectation.

Enfin le temps d'exécution du renvoi du résultat d'une fonction est aussi une unité de temps.

1.3.2 Définitions

Dans un premier temps, nous nous limitons à la complexité temporelle de l'algorithme. Nous recherchons une garantie de performances ; aussi nous nous intéressons au pire cas d'exécution d'un programme.

Soit $\mathcal{I}$ une instance du problème traité par l'algorithme $\mathcal{A}$, notons $size(\mathcal{I})$ la taille de $\mathcal{I}$ et $Time(\mathcal{A}, \mathcal{I})$ le temps d'exécution de $\mathcal{A}$ lorsqu'il s'applique à $\mathcal{I}$.

Etant donnée une taille maximale n de problème, le pire cas d'exécution de $\mathcal{A}$ sur les instances de taille inférieure ou égale à n, noté $Time(\mathcal{A})(n)$ est défini par $Time(\mathcal{A})(n) \equiv \max(Time(\mathcal{A}, \mathcal{I}) \mid size(\mathcal{I}) \leq n)$.

Autrement dit, $Time(\mathcal{A})$ est une fonction croissante de $\mathbb{N}$ dans $\mathbb{N}$. Ce qui nous intéresse, c'est l'efficacité de $\mathcal{A}$ sur les problèmes de grande taille, c'est à dire le comportement asymptotique de $Time(\mathcal{A})(n)$ lorsque n tend vers ∞. A cette fin, nous introduisons des notations asymptotiques.

Définition 1 *Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Alors :*
- $O(g) = \{f \mid \exists c \in \mathbb{N}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 \in N, f(n) \leq c \cdot g(n)\}$
  On note par abus de langage $f = O(g)$ ssi $f(n) \in O(g)$;
- $\Omega(g) = \{f \mid \exists c \in \mathbb{N}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 \in N, f(n) \geq c \cdot g(n)\}$
  On note par abus de langage $f = \Omega(g)$ ssi $f \in \Omega(g)$;
- *On note $f = \Theta(g)$ ssi $f(n) = O(g)$ et $f(n) = \Omega(g)$.*

La constante c de la définition précédente tient compte du fait que si on exécute un programme sur une machine dont le processeur est c fois « plus rapide », alors le temps d'exécution sera divisé par c.

On dira que la complexité d'un algorithme est en $O(g)$ (resp. $\Omega(g)$, $\Theta(g)$) si $Time(\mathcal{A}) = O(g)$ (resp. $Time(\mathcal{A}) = \Omega(g)$, $Time(\mathcal{A}) = \Theta(g)$). Nous établirons que la complexité de la plupart des algorithmes que nous verrons en cours, est dans $O(n^k)$ ou dans $O(n^k \cdot \log^{k'}(n))$.

Notations. Lorsque l'algorithme $\mathcal{A}$ sera déterminé sans ambiguïté, nous désignerons plus simplement la fonction $Time(\mathcal{A})$ par $Time$.

1.3.3 Illustration

Fusion de tableaux triés Un tableau T de valeurs numériques (ou plus généralement de valeurs prises dans un domaine totalement ordonné) est dit *trié* ssi $\forall i < j, T[i] \leq T[j]$.

La fusion de deux tableaux T_1 et T_2 triés consiste à produire un tableau T trié composé des valeurs des deux tableaux en tenant compte des répétitions. L'algorithme 3 résout ce problème. Indiquons son principe :

Algorithme 3 : Fusion de deux tableaux triés.

```
Fusion(T_1, n_1, T_2, n_2) : tableau d'entiers
Input : ∀i ∈ {1,2} T_i un tableau trié, n_i sa dimension
Output : La fusion triée des deux tableaux
Data : T tableau de dimension n_1 + n_2
Data : i, i_1, i_2 entiers
i_1 ← 1 ; i_2 ← 1
for i ← 1 to n_1 + n_2 do
    if i_1 > n_1 then
        T[i] ← T_2[i_2]
        i_2 ← i_2 + 1
    else if i_2 > n_2 then
        T[i] ← T_1[i_1]
        i_1 ← i_1 + 1
    else if T_2[i_2] ≤ T_1[i_1] then
        T[i] ← T_2[i_2]
        i_2 ← i_2 + 1
    else
        T[i] ← T_1[i_1]
        i_1 ← i_1 + 1
    end
end
return T
```

– Il maintient un indice i_k par tableau T_k pointant sur la prochaine valeur à insérer dans T ou au delà du plus grand indice si toutes les valeurs ont été insérées. L'indice de la boucle i sert d'indice d'insertion dans le tableau T puisqu'à chaque tour de boucle, on insère un élément.
– Si les deux indices pointent encore dans leur tableau respectif, on insère la valeur pointée la plus petite et on incrémente l'indice correspondant. Sinon, on insère la seule valeur disponible en incrémentant également l'indice.

Calculons sa complexité. La taille de la représentation de deux tableaux de taille n_1 et n_2 est $(n_1 + n_2) \cdot B_e$. Comme nous en avons déjà discuté, les calculs de complexité se font à une constante près. Aussi on considérera que la taille de l'entrée est $n \equiv n_1 + n_2$. L'algorithme exécute deux instructions élémentaires puis n tours de boucle puis renvoie la valeur. Au cours d'un tour, il exécute au moins 4 instructions (en incluant le test d'entrée dans le tour) et au plus 6 instructions. Donc $3 + 4n \leq Time(n) \leq 3 + 6n$. Ce qui signifie que $Time(n) = \Theta(n)$.

Notons que le résultat est aussi vrai l'algorithme 4 (une légère variante de l'algorithme précédent) mais qu'il est un peu plus difficile à prouver.

Tri de tableaux par fusion. Le tri d'un tableau T consiste à produire un tableau T' avec les mêmes valeurs (et les mêmes répétitions) que T mais triées par ordre croissant. Pour ce faire, l'algorithme 2 procède ainsi :

1. Il partage le tableau en deux sous-tableaux (de dimension approximative-

Algorithme 4 : Autre fusion de deux tableaux triés.

`Fusion`(T_1, n_1, T_2, n_2) : tableau d'entiers

Input : $\forall i \in \{1, 2\}$ T_i un tableau trié, n_i sa dimension
Output : La fusion triée des deux tableaux

Data : T tableau de dimension $n_1 + n_2$
Data : i, i_1, i_2 entiers

```
i1 ← 1 ; i2 ← 1 ; i ← 1
while i1 ≤ n1 and i2 ≤ n2 do
    if T2[i2] ≤ T1[i1] then
        T[i] ← T2[i2]
        i2 ← i2 + 1
    else
        T[i] ← T1[i1]
        i1 ← i1 + 1
    end
    i ← i + 1
end
while i1 ≤ n1 do
    T[i] ← T1[i1]
    i1 ← i1 + 1
    i ← i + 1
end
while i2 ≤ n2 do
    T[i] ← T2[i2]
    i2 ← i2 + 1
    i ← i + 1
end
return T
```

ment égales) qu'il trie (en s'appelant récursivement).

2. Puis il fusionne les deux tableaux à l'aide de l'algorithme 3.

La correction de cet algorithme est évidente. Intéressons-nous à sa complexité et notons n la taille du tableau à trier qu'on condidère comme la taille de l'entrée. Notons d'abord que $Time(1) = 2$ et que :

$$\forall n > 1, Time(n) \leq 1 + n + c \cdot n + Time(\lfloor n/2 \rfloor) + Time(\lceil n/2 \rceil)$$

avec c la constante de l'algorithme précédent.

Posons $c' \equiv c + 2$ et montrons d'abord par récurrence que :
$\forall k \geq 1, Time(2^k) \leq c' \cdot k \cdot 2^k \cdot Time(1)$.
Nous laissons le soin au lecteur de le vérifier pour $k = 1$.

Appliquons l'hypothèse de récurrence :

$$Time(2^{k+1}) \leq c' \cdot 2^{k+1} + 2 \cdot Time(2^k)$$

$$\leq (c' \cdot 2^{k+1} + 2c' \cdot k \cdot 2^k) \cdot Time(1) = (c' \cdot (k+1) \cdot 2^{k+1}) \cdot Time(1)$$

Soit maintenant n quelconque et l'unique k tel que $n \leq 2^k < 2n$.

$$Time(n) \leq Time(2^k) \leq (c' \cdot k \cdot 2^k) \cdot Time(1)$$

$$\leq (c' \cdot \log_2(2n) \cdot 2n) \cdot Time(1) = (2c' \cdot (\log_2(n) + 1) \cdot n) \cdot Time(1)$$

D'où $Time(n) = O(n \log_2(n))$. En réalité, on a plus précisément $Time(n) = \Theta(n \log_2(n))$ *(faites-en la preuve).*

Pour terminer, remarquons que si la récursivité facilite la conception d'algorithmes efficaces, elle introduit aussi une possibilité de non terminaison dans les algorithmes, comme les constructeurs **while** et **repeat**.

Chapitre 2

Gestion dynamique des données : listes et piles

Objectifs. Présenter des structures *simples* dédiées à la gestion de données dont la « durée de vie » est éventuellement inférieure à celle de l'exécution d'un algorithme.

2.1 Gestion d'annuaires

2.1.1 Principe d'un annuaire

Contenu d'un annuaire

Un annuaire est un ensemble d'enregistrements où chaque enregistrement fournit des informations relatives à une entité d'un catégorie donnée.
Par exemple :
- Un annuaire des produits vendus par une société ; les informations relatives à chaque produit pourraient être son code, son libellé, son prix, une brève description, etc.
- Un annuaire des employés d'une société ; les informations relatives à chaque employé pourraient être son adresse électronique, son nom, son prénom, le département auquel il est rattaché, etc.
- Un annuaire des patients de la Sécurité Sociale ; les informations relatives à chaque patient pourraient être son n° de SS, son nom, son prénom, son adresse, etc.

Première observation. Notons d'abord qu'on stocke le même type d'information pour chaque enregistrement d'un même annuaire. Ceci nous amène à introduire dans notre langage algorithmique un constructeur de type de données que nous appelerons **struct**. Sa syntaxe est la suivante :

struct *nom de la structure d'informations* {
nom du premier champ : *type de données du premier champ*
...
nom du dernier champ : *type de données du dernier champ*
}

Par exemple une structure (minimale) des informations relatives aux patients s'écrirait :

struct *patient* {
  $n^\circ SS$: un entier sur 15 chiffres
  nom : une chaîne de 50 caractères
  prenom : une chaîne de 50 caractères
}

Et si on suppose que *unpatient* est une variable de type *patient*, alors on accède aux différents champs par la syntaxe suivante : $unpatient.n^\circ SS$, *unpatient.nom* et *unpatient.prenom*.

Deuxième observation. Parmi les champs d'un enregistrement, il y a un champ particulier qui permet d'identifier l'enregistrement. Autrement deux enregistrements différents ne peuvent avoir la même valeur pour ce champ. Dans les exemples précédents, il s'agit du code produit, de l'adresse électronique et du n° de SS. Dans la suite, nous l'appellerons *id* (pour identifiant).

Opérations sur un annuaire

Les opérations usuelles supportées par un annuaire sont les suivantes.

- *la recherche* des informations relatives à un enregistrement désigné par son identifiant. Cette opération doit prévoir que l'enregistrement peut ne pas exister.
- *l'ajout* d'un enregistrement désigné par son identifiant. Cette opération doit interdire l'ajout si un enregistrement de même identifiant est déjà présent.
- *la suppression* d'un enregistrement désigné par son identifiant. La suppression n'a lieu que si l'enregistrement existe.
- *la modification* d'un enregistrement désigné par son identifiant avec de nouvelles informations. La modification n'a lieu que si l'enregistrement existe.

Du point de vue algorithmique, les champs autres que le champ *id* sont traités de manière identique. Aussi nous supposerons qu'il n'y a qu'un second champ appelé $info$. D'autre part, la modification est en fait une infime variante de la recherche, aussi nous ne l'étudierons pas.

Les paragraphes qui suivent présentent différentes solutions de gestion d'un annuaire. Nous aborderons d'autres solutions plus élaborées dans les chapitres suivants.

2.1.2 Gestion à l'aide d'un tableau indicé par *id*

Prenons le cas de l'annuaire de la Sécurité Sociale, l'identifiant est un entier. Cet identifiant pourrait être l'indice du tableau d'infos. Cependant, il faut distinguer le cas où l'enregistrement existe et celui où il n'existe pas. Pour ce faire, nous supposons que le champ $info$ prend la valeur spéciale `NULL` lorsque l'enregistrement n'existe pas. L'algorithme 5 présente une telle gestion (*nmax* représente la valeur du plus grand identifiant possible).

A priori, cette gestion semble tout à fait satisfaisante puisque toutes les opérations se font en temps constant. Cependant cette simplicité masque un inconvénient rédhibitoire : la place occupée.

Algorithme 5 : Première gestion d'un annuaire

Data : T un tableau de $nmax$ informations toute initialisées à `NULL`

`Chercher`($unid$) : information
Input : $unid$ l'identifiant d'un enregistrement
Output : les informations relatives à cet enregistrement ou `NULL` s'il est absent de l'annuaire

return $T[unid]$

`Ajouter`($unid, uneinfo$) : booléen
Input : $unid$ l'identifiant d'un enregistrement à ajouter, $uneinfo$ ses informations
Output : un booléen indiquant si l'ajout s'est déroulé correctement

if $T[unid] \neq$ `NULL` **then return false**
else $T[unid] \leftarrow uneinfo$; **return true**

`Supprimer`($unid$) : booléen
Input : $unid$ l'identifiant d'un enregistrement à supprimer
Output : un booléen indiquant si la suppression s'est déroulée correctement

if $T[unid] =$ `NULL` **then return false**
else $T[unid] \leftarrow$ `NULL` ; **return true**

En effet, le tableau est indicé par l'ensemble des identifiants possibles. Par exemple dans le cas du n° de SS, cela conduit à un tableau de 10^{15} entrées (qui ne pourra être stocké sur un disque dur que dans des dizaines d'années). Or on estime le nombre de numéros attribués à 10^8. Ce qui signifie que, même en multipliant par 10 cette attribution pour couvrir dix générations, on aurait un ratio d'occupation du tableau de 10^{-6} ce qui est inacceptable.

2.1.3 Gestion à l'aide d'un tableau séquentiel

Une deuxième idée possible consiste à prévoir un tableau dont la taille majore de manière réaliste le nombre maximal attendu d'enregistrements.

Les enregistrements sont stockés par indice croissant au fur et à mesure de leur ajout. Un compteur *sommet* permet de savoir où s'arrêtent les enregistrements et d'éviter le débordement de tableau.

La suppression est un peu plus délicate car il s'agit de déplacer vers le bas les éléments situés au dessus de l'enregistrement supprimé.

La recherche se fait « séquentiellement » en parcourant le tableau du premier indice au sommet. Ce parcours est interrompu si on a rencontré l'enregistrement.

L'algorithme 6 décrit formellement cette gestion. Celle-ci a trois inconvénients :

- Le concepteur définit lui-même la taille maximum alors que d'une machine à l'autre, celle-ci pourrait être différente.
- Lorsque le tableau n'est pas plein la place est quand même allouée alors qu'elle pourrait être utilisée par un autre programme.

– Le temps de chaque opération est de l'ordre de *sommet*, *i.e.* du nombre d'enregistrements de l'annuaire.

2.1.4 Gestion à l'aide d'une liste

Espace dynamique

Cette troisième gestion vise à remédier aux deux premiers inconvénients de la gestion précédente. Pour ce faire, nous devons introduire le concept *d'espace dynamique*. Cet espace est géré par le système et permet à un programme de s'allouer de l'espace pour y stocker des *objets* qu'il accède au moyen de *références*.

Plus précisément, un objet est créé par la fonction **new**(*type*) qui contiendra une valeur du type indiqué. Pour lire ou modifier son contenu, il faut passer par l'intermédiaire d'une variable de type référence qui contiendra la référence de l'objet fournie par le retour de la fonction **new**.

Ainsi les deux instructions :
$refentier \leftarrow$ **new**(*entier*)
$*refentier \leftarrow 1$
où $refentier$ est une variable de type référence, créent un objet référencé par $refentier$ et l'initialise à 1.

Notez que pour accéder à l'objet, il faut placer l'astérisque devant la variable référence.

Lorsqu'un objet n'est plus nécessaire, alors on désalloue la place qu'il occupe à l'aide de la fonction **delete**($variabledetyperéférence$). Dans l'exemple, cela donne :
delete($refentier$)

On affecte à une variable (ou un champ) référence qui ne pointe pas sur un objet, la valeur `NULL` afin de faciliter les tests comme nous le verrons par la suite.

Les objets dynamiques sont à la base des listes. Une liste est une suite d'objets structurés et dont un des champs pointe sur l'élément suivant de la liste.

Dans le cas de l'annuaire, le type de données à définir est :
struct *enregistrement* {
id : identifiant
$info$: information
suiv : référence
}

A priori pour accéder à un champ, disons *id*, d'un enregistrement référencé par une variable *penr*, il faut employer la syntaxe $(*penr).id$. Comme cet accès est extrêmement fréquent dans le cas d'objet dynamique, nous le noterons $penr \rightarrow id$.

La troisième gestion procède ainsi. Une variable *liste* référence le premier enregistrement de l'annuaire dans la liste. S'il n'en y a pas (la liste est vide) alors cette variable vaut `NULL`.

Pour rechercher un enregistrement on compare l'identifiant avec les identifiants de la liste en suivant le champ *suiv* de chaque enregistrement jusqu'à ce que l'identifiant soit recontré ou qu'il n'y ait plus d'éléments à examiner.

Les opérations d'ajout et de suppression soulèvent deux difficultés supplémentaires.

Algorithme 6 : Deuxième gestion d'un annuaire

```
Type : struct enregistrement { id : identifiant ; info : information }
Data : T un tableau de nmax enregistrements
Data : sommet un indice pointant sur le dernier enregistrement du
       tableau, initialisé à 0

Chercher(unid) : indice dans le tableau
Input : unid l'identifiant d'un enregistrement
Output : l'indice du tableau contenant cet enregistrement ou 0 s'il est
         absent de l'annuaire
Data : i, un indice du tableau

for i ← 1 to sommet do
|  if T[i].id = unid then return i
end
return 0

Ajouter(unid, uneinfo) : booléen
Input : unid l'identifiant d'un enregistrement à ajouter, uneinfo ses
        informations
Output : un booléen indiquant si l'ajout s'est déroulé correctement
Data : i, un indice du tableau

if sommet = nmax then return false
i ← Chercher(unid)
if i ≠ 0 then return false
else
|  sommet ← sommet + 1
|  T[sommet].id ← unid
|  T[sommet].info ← uneinfo
|  return true
end

Supprimer(unid) : booléen
Input : unid l'identifiant d'un enregistrement à supprimer
Output : un booléen indiquant si la suppression s'est déroulée
         correctement
Data : i, j, deux indices du tableau

i ← Chercher(unid)
if i = 0 then return false
else
|  for j ← i + 1 to sommet do
|  |  T[j − 1].id ← T[j].id
|  |  T[j − 1].info ← T[j].info
|  end
|  sommet ← sommet − 1
|  return true
end
```

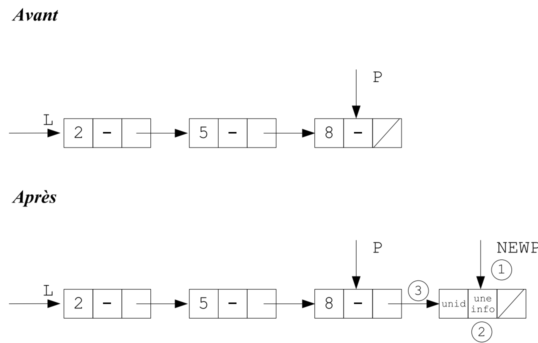

FIG. 2.1: Ajout d'un enregistrement à une liste

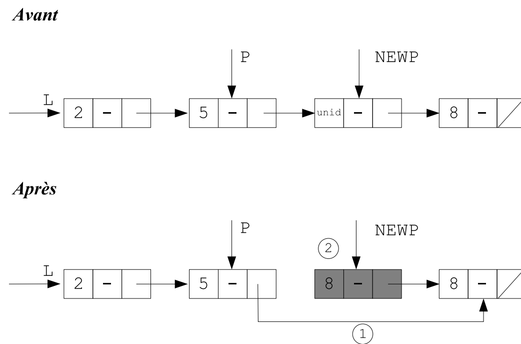

FIG. 2.2: Suppression d'un enregistrement d'une liste

D'une part, il faut maintenir deux variables références (*p* et *newp*) pointant le plus souvent sur un enregistrement et son suivant de telle sorte que lorsque la deuxième référence vaut `NULL`, la première pointe sur le dernier élément de la liste. Ainsi l'ajout se fait en créant un nouvel objet et en l'attachant en fin de liste (voir la figure 2.1).

D'autre part, si la suppression s'applique au premier élément de la liste alors la variable *liste* doit référencer le suivant du premier élément ce qui implique un traitement particulier. La figure 2.2 traite le cas général.

Ces opérations sont décrites par l'algorithme 7.

2.2 Le tri-fusion avec des listes

Utiliser les listes dans un tri-fusion permet d'économiser de manière significative l'espace utilisé. Aucun espace supplémentaire n'est nécessaire pour trier une liste car il s'agit alors d'une manipulation de listes.

Nous développons tout d'abord une fusion de deux listes. La liste qui contient le plus petit élément « accueille » les éléments de la première liste en les examinant tour à tour. L'insertion d'un élément de la première liste se fait par une

Algorithme 7 : Troisième gestion d'un annuaire

```
Type : struct enregistrement
        { id : identifiant ; info : information ; suiv : référence }
Data : liste une référence pointant sur le premier enregistrement de
        l'annuaire, initialisé à NULL
Chercher(unid) : référence d'un enregistrement
Input : unid l'identifiant d'un enregistrement
Output : une référence de cet enregistrement ou NULL en cas d'absence
Data : p une variable référence

p ← liste
while p ≠ NULL do
 |  if p→id = unid then return p
 |  p ← p→suiv
end
return NULL

Ajouter(unid, uneinfo) : booléen
Input : (unid, uneinfo) un enregistrement à ajouter
Output : un booléen indiquant si l'ajout s'est déroulé correctement
Data : p, newp deux variables références
p ← NULL ; newp ← liste
while newp ≠ NULL do
 |  p ← newp
 |  if p→id = unid then return false
 |  newp ← p→suiv
end
newp ← new(enregistrement)
newp→id ← unid ; newp→info ← uneinfo ; newp→suiv ← NULL
if p=NULL then liste ← newp
else p→suiv ← newp
return true

Supprimer(unid) : booléen
Input : unid l'identifiant d'un enregistrement à supprimer
Output : un booléen indiquant si la suppression s'est bien déroulée
Data : p, newp deux variables références
if liste = NULL then return false
if liste→id = unid then
 |  p ← liste→suiv ; delete(liste) ; liste ← p ; return true
end
p ← liste ; newp ← liste→suiv
while newp ≠ NULL do
 |  if newp→id = unid then
 |   |  p→suiv = newp→suiv ; delete(newp) ; return true
 |  end
 |  p ← newp ; newp ← p→suiv
end
return false
```

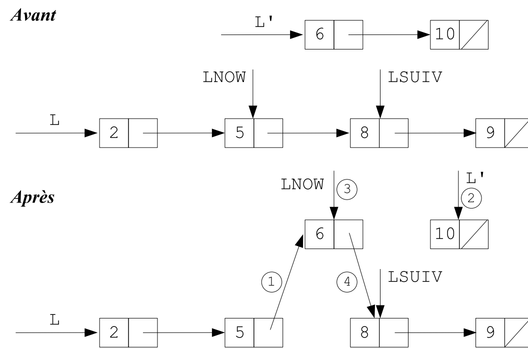

Fig. 2.3: Fusion de deux listes triées

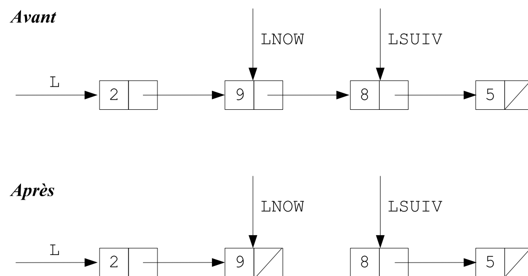

Fig. 2.4: Scission d'une liste

manipulation de variables références relativement simple (voir la figure 2.3). L'algorithme 8 réalise cette fusion.

Le tri effectué par l'algorithme 9 procède de la manière suivante. Il calcule la longueur de la liste. Si cette longueur est inférieure ou égale à 1, il renvoie la liste (*i.e.* il n'y a rien à trier). Dans le cas contraire, il se positionne sur le « milieu » de la liste et procède à la scission de la liste en deux sous-listes (voir la figure 2.4). Chaque sous-liste est alors triée par un appel récursif. Puis l'algorithme fusionne les deux sous-listes triées à l'aide de l'algorithme 8.

2.3 Gestion de piles

2.3.1 Principe d'une pile

Afin d'économiser l'espace utilisé par un algorithme, nous avons vu que la gestion dynamique par création et suppression d'objets permet d'allouer l'espace nécessaire à une information uniquement lorsque cette information est utile à l'algorithme. Nous allons maintenant étudier un cas particulier mais très fréquent de données dynamiques. Supposons que lors d'un algorithme un ensemble d'informations dynamiques vérifie la propriété suivante :

Algorithme 8 : Fusion de deux listes triées.

Type : **struct** *enregistrement*{ *val* : entier ; *suiv* : référence }

`Fusion`(L_1, L_2) : liste d'enregistrements

Input : $\forall i \in \{1, 2\}$ L_i une liste d'enregistements triés par valeur
Output : La fusion triée des deux listes

Data : L, L' des variables référence

if $L_1 =$ `NULL` **then return** L_2
if $L_2 =$ `NULL` **then return** L_1
if $L_1 \rightarrow val \leq L_2 \rightarrow val$ **then** $L \leftarrow L_1$; $L' \leftarrow L_2$
else $L \leftarrow L_2$; $L' \leftarrow L_1$
$LNOW \leftarrow L$
repeat
  repeat
    $LSUIV \leftarrow LNOW \rightarrow suiv$
    if $LSUIV =$ `NULL` **then**
      $LNOW \rightarrow suiv \leftarrow L'$; **return** L
    end
    if $LSUIV \rightarrow val < L' \rightarrow val$ **then** $LNOW \leftarrow LSUIV$
  until $LNOW \neq LSUIV$
  $LNOW \rightarrow suiv \leftarrow L'$
  $L' \leftarrow L' \rightarrow suiv$
  $LNOW \leftarrow LNOW \rightarrow suiv$
  $LNOW \rightarrow suiv \leftarrow LSUIV$
until $L' =$ `NULL`
return L

Algorithme 9 : Tri d'une liste

`Trie`(L) : liste d'enregistrements

Input : L une liste d'enregistrements
Output : la liste triée

Data : $LNOW, LSUIV, L_1, L_2$ variables référence
Data : i, n entiers

$n \leftarrow 0$; $LNOW \leftarrow L$
while $LNOW \neq$ `NULL` **do** $n \leftarrow n + 1$; $LNOW \leftarrow LNOW \rightarrow suiv$
if $n \leq 1$ **then return** L
$LNOW \leftarrow L$
for $i \leftarrow 1$ **to** $\lfloor n/2 \rfloor - 1$ **do** $LNOW \leftarrow LNOW \rightarrow suiv$
$LSUIV \leftarrow LNOW \rightarrow suiv$
$LNOW \rightarrow suiv \leftarrow$ `NULL`
$L_1 \leftarrow$ `Trie`(L)
$L_2 \leftarrow$ `Trie`$(LSUIV)$
$L \leftarrow$ `Fusion`(L_1, L_2)
return L

Lorsqu'une information est produite,
elle deviendra inutile avant toutes celles déjà produites.

Dans ce cas, la gestion des ces données par une *pile* est tout à fait indiquée.

2.3.2 Gestion d'une pile par un tableau

Une pile peut être vue comme un tableau muni d'un indice appelé le sommet de pile. Chaque fois qu'une information est produite on incrémente le sommet de pile et on insère l'information dans cette cellule, *i.e* on *empile* l'information. A l'inverse, lorsqu'une information devient inutile (nécessairement celle se trouvant au sommet de la pile) on décrémente l'indice, *i.e* on *dépile* l'information.

L'algorithme 10 gère une pile à l'aide d'un tableau d'entiers, T de taille n. Les deux premières fonctions s'occupent de la « création » et de la « suppression » d'informations tandis que la dernière accède à l'information. Remarquez que ces fonctions gèrent les cas de débordement (par le haut ou par le bas) du tableau et l'accès à une information invalide.

Algorithme 10 : Gestion d'une pile par un tableau

Data : T un tableau de n entiers
Data : sp l'indice du sommet de la pile initialement 0

`Empiler`(val) : booléen

Input : val un entier à insérer dans la pile
Output : un booléen indiquant si l'opération s'est déroulée correctement

if $sp = n$ **then return false**
else $sp \leftarrow sp + 1$; $T[sp] \leftarrow val$; **return true**

`Depiler`() : booléen

Output : un booléen indiquant si l'opération s'est déroulée correctement

if $sp = 0$ **then return false**
else $sp \leftarrow sp - 1$; **return true**

`Lire`(dep) : entier, booléen

Input : dep un déplacement (positif) par rapport au sommet de pile
Output : l'entier correspondant à l'indice $sp - dep$
Output : un booléen indiquant si l'opération s'est déroulée correctement

if $dep \geq sp$ **then return** $0,$ **false**
else return $T[sp - dep],$ **true**

2.3.3 Gestion d'une pile par une liste

La gestion d'une pile par une liste est particulièrement appropriée puisqu'il s'agit de deux structures dynamiques. L'algorithme 11 gère une pile à l'aide d'une liste.

Empiler un entier consiste à créer un enregistrement et l'insérer en tête de la liste tandis que dépiler consiste à supprimer l'enregistrement en tête de la liste.

Le seul inconvénient de cette gestion est la lecture dans la pile. Ici le paramètre de la lecture est un déplacement à partir du sommet de la pile avec un déplacement nul dans le cas où on veut lire le sommet de la pile. Ainsi la complexité de cette opération est proportionnelle au déplacement dans la pile contrairement à la lecture dans le cas d'une gestion par tableau qui se fait en temps constant. Cependant dans la pratique (voir la section suivante), les lectures se font généralement à des emplacements proches du sommet.

Algorithme 11 : Gestion d'une pile par une liste

Type : **struct** *enregistrement* { v : entier ; *suiv* : référence }
Data : *pile* une liste d'entiers initalisée à `NULL`

`Empiler`(*val*)

Input : *val* un entier à insérer dans la pile
Data : *suite* une référence

$suite \leftarrow pile$
$pile \leftarrow$ **new**(*enregistrement*)
$pile \rightarrow v \leftarrow val$; $pile \rightarrow suiv \leftarrow suite$
return

`Depiler`() : booléen

Output : un booléen indiquant si l'opération s'est déroulée correctement

if $pile =$ `NULL` **then return false**
else $suite \leftarrow pile \rightarrow suiv$; **delete**(*pile*) ; $pile \leftarrow suite$; **return true**

`Lire`(*dep*) : entier, booléen

Input : *dep* un déplacement dans la pile
Output : l'entier correspondant au déplacement *dep*
Output : un booléen indiquant si l'opération s'est déroulée correctement
Data : p une variable référence

$p \leftarrow pile$
while $p \neq$ `NULL` **do**
    if $dep = 0$ **then return** $p \rightarrow v$, **true**
    $p \leftarrow p \rightarrow suiv$
    $dep \leftarrow dep - 1$
end
return 0, **false**

2.3.4 La pile d'exécution d'un programme

La gestion de données à l'aide d'une pile est un mécanisme essentiel pour l'exécution d'un programme par le système d'exploitation. Plus précisément, lorsque le système lance l'exécution d'un programme, il lui alloue un espace mémoire qu'il gère comme une pile. Cette pile est appelée pile d'exécution. Elle sert principalement pour les appels et les retours de fonctions.

A n'importe quel instant de l'exécution d'un programme, la pile d'exécution contient, empilés les uns sur les autres, les contextes de chaque appel de fonction

non terminé. Trois registres de la machine interviennent dans la gestion de cette pile : le *compteur ordinal* (CO) qui contient l'adresse de la prochaine instruction à exécuter par le programme, le *sommet de pile* (SP) qui contient l'adresse du sommet de pile et le *contexte de pile* (CP) qui contient l'adresse dans pile du début des informations relatives à l'exécution de l'appel courant de fonction. Lors de l'appel d'une fonction, le système d'exploitation effectue les opérations suivantes :

1. Il empile (avec mise à jour automatique du registre SP comme ce sera le cas de toutes les opérations) l'adresse de l'instruction qui suit l'appel de fonction afin de poursuivre son exécution dans la fonction appelante, une fois l'appel terminé.
2. Il empile la valeur actuelle du CP afin d'adresser les paramètres et les variables locales de la fonction appelante ;
3. Il empile l'adresse des variables qui recevront les résultats de la fonction, une fois l'appel terminé. Il met à jour le registre CP avec la valeur courante du registre SP.
4. Il calcule l'adresse relative des paramètres et des variables locales de la fonction appelée et ces informations et les stocke sous une forme qui peut varier d'un système à un autre (aussi nous ne les détaillons pas). Ces informations combinées avec la valeur du registre CP permettent d'adresser les paramètres et les variables locales au cours de l'exécution de la fonction.
5. Il évalue les paramètres d'appel et les empile (on suppose que, par défaut, le passage de paramètres se fait par valeur).
6. Il empile les variables locales avec une valeur initiale (éventuellement par défaut).

Lors du retour de la fonction, le système procède ainsi :

1. Il recopie les résultats dans les variables correspondant aux adresses qui se trouvent sous le registre CP.
2. Il dépile les variables locales, les paramètres et les informations d'adressage.
3. Il met à jour CP avec le sommet de pile et le dépile.
4. Il dépile l'adresse des variables qui ont reçu les résultats.
5. Il met à jour CO avec le sommet de pile et le dépile.

Remarque. Ainsi une fonction peut s'appeler récursivement sans risque de confusion entre les variables relatives à chaque appel puisqu'elles sont localisées à différents endroits de la pile et que le registre CP et les informations d'adressage permettent de déterminer l'adresse des variables appropriées.

Nous employons assez souvent la récursivité car elle facilite la conception d'algorithmes efficaces et que la complexité de tels algorithmes se calcule relativement facilement à l'aide de formules de récurrence.

Algorithme 12 : Tri d'un tableau

`Trie`(T, deb, fin) : tableau d'entiers

Input : T un tableau, $deb \leq fin$ deux indices du tableau
Output : un tableau contenant les valeurs de $T[deb]$ à $T[fin]$ triées

Data : T' un tableau de $fin + 1 - deb$ entiers
Data : T_1 un tableau de $\lfloor (fin + 1 - deb)/2 \rfloor$ entiers
Data : T_2 un tableau de $\lceil (fin + 1 - deb)/2 \rceil$ entiers

1 **if** $deb = fin$ **then**
2 | $T'[1] \leftarrow T[deb]$
3 **else**
4 | $T_1 \leftarrow$ `Trie`$(T, deb, \lfloor (deb + fin - 1)/2 \rfloor)$
5 | $T_2 \leftarrow$ `Trie`$(T, \lfloor (deb + fin - 1)/2 \rfloor + 1, fin)$
6 | $T' \leftarrow$ `Fusion`$(T_1, \lfloor (fin + 1 - deb)/2 \rfloor, T_2, \lceil (fin + 1 - deb)/2 \rceil)$
7 **end**
8 **return** T'

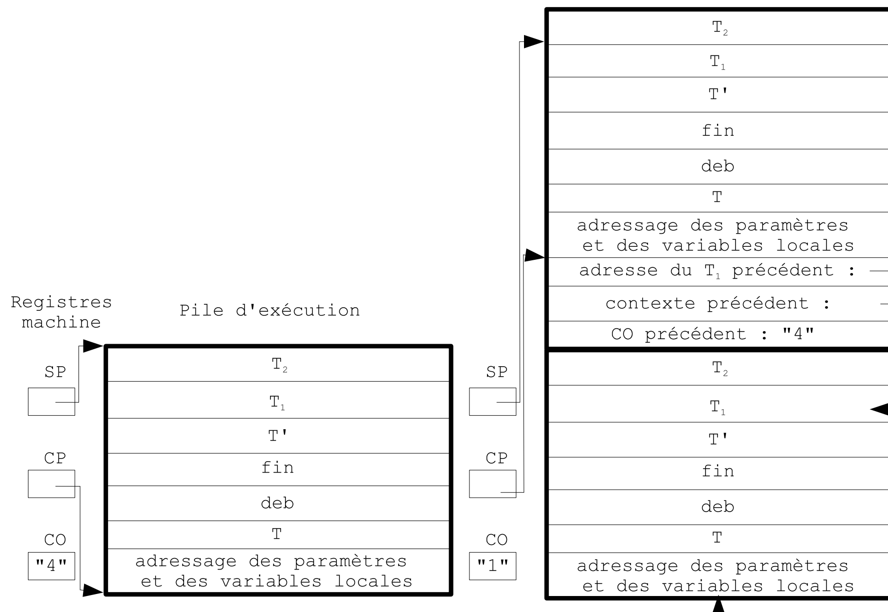

FIG. 2.5: La pile d'exécution avant et après l'appel à `Trie`

Chapitre 3

Gestion dynamique des données : tables de hachage

Objectifs. Présenter des structures combinant aspects statiques et dynamiques dédiées à la gestion de données afin de diminuer les temps d'accès tout en conservant l'avantage de la dynamicité.

3.1 Gestion d'un annuaire par table de hachage

3.1.1 Principe d'une table de hachage

Appelons $\mathcal{ID}$, l'espace des identifiants. La gestion d'un annuaire par table de hachage est basée sur :

– La donnée d'un entier m inférieur à la cardinalité de $\mathcal{ID}$ et généralement très petit devant celle-ci.
– Une fonction surjective h de $\mathcal{ID}$ vers $\{1, \ldots, m\}$. Afin que la gestion soit efficace, cette fonction doit vérifier certaines propriétés présentées plus loin.

Pour l'instant, nous souhaitons que la fonction soit calculable efficacement (et de préférence en temps constant). Voici deux exemples de fonctions de hachage.

– Si les identifiants sont des chaînes de caractères dont le premier est une lettre (un nom ou une adresse email) alors h pourrait être l'indice de cette lettre dans l'alphabet. Ici $m = 26$.
– Si les identifiants sont des valeurs numériques alors on pourrait choisir $h(id) = (id \mod m) + 1$.

L'algorithme 13 décrit la gestion d'annuaire par une table de hachage. La table de hachage T est une table de m références, chacune pointant sur une liste d'enregistrements. Lorsqu'on insére un enregistrement (dont l'identifiant n'existe pas déjà), on calcule l'indice de la liste dans laquelle il doit être inséré à l'aide de la fonction de hachage appliquée à l'identifiant. Lors de la recherche (ou de la suppression) on détermine l'indice de la liste dans la table de la même façon. La gestion de chaque liste se fait comme au chapitre 2.

Algorithme 13 : Gestion d'un annuaire par table de hachage

```
Type : struct enregistrement
        { id : identifiant ; info : information ; suiv : référence }
Data : T un tableau de m références initialisées à NULL
Chercher(unid) : référence d'un enregistrement
Input : unid l'identifiant d'un enregistrement
Output : une référence de cet enregistrement ou NULL s'il est absent de
         l'annuaire
Data : p une variable référence

p ← T[h(unid)]
while p ≠ NULL do
|  if p→id = unid then return p
|  p ← p→suiv
end
return NULL

Ajouter(unid, uneinfo) : booléen
Input : unid l'identifiant d'un enregistrement à ajouter, uneinfo ses
        informations
Output : un booléen indiquant si l'ajout s'est déroulé correctement
Data : p une variable référence, i un indice de tableau
if Chercher(unid) ≠ NULL then return false
i ← h(unid) ; p ← T[i]
T[i] ← new(enregistrement)
T[i]→id ← unid ; T[i]→info ← uneinfo
T[i]→suiv ← p
return true

Supprimer(unid) : booléen
Input : unid l'identifiant d'un enregistrement à supprimer
Output : un booléen indiquant si la suppression s'est déroulée
         correctement
Data : p, newp deux variables références, i un indice de tableau
i ← h(unid)
if T[i] = NULL then return false
if T[i]→id = unid then
|  p ← T[i]→suiv ; delete(T[i]) ; T[i] ← p
|  return true
end
p ← T[i] ; newp ← T[i)]→suiv
while newp ≠ NULL do
|  if newp→id = unid then
|  |  p→suiv = newp→suiv
|  |  delete(newp)
|  |  return true
|  end
|  p ← newp ; newp ← p→suiv
end
return false
```

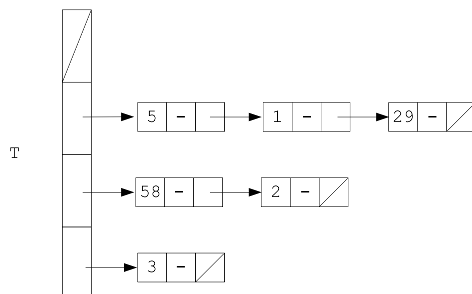

FIG. 3.1: Une table de hachage

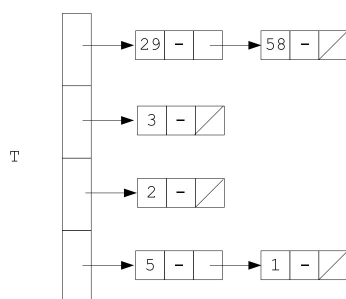

FIG. 3.2: Une autre table de hachage

La figure 3.1 présente une table de hachage dans le cas d'identifiants numériques, avec $m = 4$ et $h(x) = (x \mod 4) + 1$. La figure 3.2 présente une table de hachage avec les mêmes identifiants, $m = 4$ et $h(x) = ((3x \mod 59) \mod 4) + 1$.

3.1.2 Considérations de complexité

Complexité spatiale. Contrairement à la gestion de listes qui ne nécessite qu'une variable référence dans le cas d'un annuaire vide, la table de hachage contient m références. La place allouée statiquement dépend donc du choix de m. Généralement, m est choisi approximativement égal au nombre maximal d'enregistrements attendus. On pourrait donc en conclure hativement que cette gestion n'est pas dynamique puisque la place statiquement allouée est proportionnelle au nombre maximal d'enregistrements. Une analyse plus précise prend en compte le ratio entre la taille occupée par une référence et la taille occupée par un enregistrement. Ce ratio est faible : une référence occupe généralement 4 octets tandis qu'un enregistrement occupe le plus fréquemment entre 500 et 1000 octets. Par conséquent, la table de hachage représente moins de 1% de la taille maximale ce qui est tout à fait acceptable.

Complexité temporelle dans le pire cas. Interessons-nous maintenant à la complexité des opérations sur l'annuaire. Puisque la recherche de l'indice de la liste se fait en temps constant, le facteur déterminant est le temps de

parcours de la liste, proportionnel à sa longueur. Dans le cas le plus défavorable, tous les enregistements sont insérés dans la même liste. Si n est le nombre d'enregistrements alors la complexité des opérations est en $\Theta(n)$.

3.1.3 Etude de la complexité sous hypothèses probabilistes

Cette section requiert les connaissances de base en probabilités discrètes.

Complexité temporelle dans le cas moyen pour une liste. Il ne semble pas y avoir d'avantages à la gestion par table de hachage vis à vis de la gestion par liste. En réalité, la différence apparaît si on considère une complexité *en moyenne*. Celle-ci suppose des hypothèses probabilistes. Nous allons l'illustrer avec la recherche d'un enregistrement par identifiant dans une liste. Si l'identifiant n'est pas présent dans la liste alors la complexité (en moyenne) de l'opération est encore en $\Theta(n)$. Le cas où l'enregistrement est présent nécessite l'introduction d'une telle hypothèse :

Tous les enregistrements de l'annuaire
ont la même probabilité d'être consultés.

Autrement dit, en notant p_i la probabilité que l'élement consulté soit le $i^{ème}$ enregistrement, on suppose que $p_i = 1/n$.

Dans ce cas, en utilisant les probabilités conditionnelles et en se rappelant que le temps d'accès au $i^{ème}$ enregistrement est de l'ordre de $c \cdot i$ pour un certain c, on obtient :

$$\texttt{Tmoyen}(n) = \sum_{i=1}^{n} c \cdot i \cdot (1/n) = c \cdot (n+1)/2$$

Par conséquent, la complexité en moyenne (notée Tmoyen(n)) est toujours en $\Theta(n)$.

Complexité temporelle dans le cas moyen pour une table de hachage. Afin de procéder à cette analyse, il nous faut faire une hypothèse sur le comportement de la fonction de hachage h lors de l'insertion de nouveaux éléments :

Lors de l'insertion d'un nouvel enregistrement,
tous les indices ont la même probabilité d'être choisis.

Appelons N_i la variable aléatoire qui donne la longueur de la liste d'indice i. D'après l'hypothèse précédente, $\forall i, j, \mathbf{E}(N_i) = \mathbf{E}(N_j)$ où $\mathbf{E}$ représente l'espérance. On a aussi $\sum_{i=1}^{m} N_i = n$ et par conséquent $\sum_{i=1}^{m} \mathbf{E}(N_i) = n$. Il vient donc $\forall i, \mathbf{E}(N_i) = n/m$. Puisque m est choisi comme le nombre maximal d'enregistrements $n \leq m$ et $\forall i, \mathbf{E}(N_i) \leq 1$. Autrement dit, $\texttt{Tmoyen}(n)$ est en $\Theta(1)$, c'est à dire indépendant de n !

Garantie probabiliste de complexité. Comme l'écart entre le cas moyen et le pire des cas est considérable, on souhaite approfondir l'approche probabiliste, pour obtenir un résultat tel que : “Avec une probabilité tendant vers 1 quand n tend vers l'infini, la taille de la plus grande liste est inférieure ou égale à $f(n)$.” où f reste à préciser.

Pour cette analyse, nous renforçons notre hypothèse probabiliste.

Lors de l'insertion d'un nouvel enregistrement,
tous les indices ont la même probabilité d'être choisis
indépendamment des insertions précédentes.

Pour simplifier l'analyse on pose $n = m$ ce qui est le cas le plus défavorable d'après nos hypothèses.

Définition 2 (Variables aléatoires relatives aux listes)
- *La v.a.* X_i *est l'indice de la liste où est inséré le* $i^{\text{ème}}$ *enregistrement.*
- *La v.a.* L_k *est la longueur de la liste d'indice* k*.*
- *La v.a.* $L = max(L_k)$ *est la plus grande longueur d'une liste de la table.*

Le lemme suivant est introduit afin de faciliter l'expression des bornes.

Lemme 1 *Soit* l *un entier non nul, alors :*

$$\frac{1}{l!} < 1/2 \times \left(\frac{e}{l}\right)^l$$

Preuve
Utilisons le développement en série de l'exponentielle : $e^l = \sum_{i=0}^{\infty} \frac{l^i}{i!} > \frac{l^l}{l!} + \frac{l^{l-1}}{(l-1)!} = 2\frac{l^l}{l!}$. D'où le résultat en divisant par $2l^l$.

c.q.f.d. ◊◊◊

Nous commençons par borner la probabilité de l'événement $\{L_k \geq l\}$ où l est un entier arbitraire.

Proposition 1 *Soit* l *un entier au plus égal à* n*, alors :*
$$\mathbf{Pr}(\{L_k \geq l\}) < 1/2 \times \left(\frac{e}{l}\right)^l$$

Preuve
Pour que L_k soit supérieur ou égal à l, il faut et il suffit qu'il existe l indices $i_1, \ldots, i_l$ tels que $\forall j \leq l, X_{i_j} = k$. Donc $\{L_k \geq l\} = \bigcup_{i_1,\ldots,i_l} \{\bigwedge_{j \leq l} X_{i_j} = k\}$.
Par conséquent :
$\mathbf{Pr}(\{L_k \geq l\}) = \mathbf{Pr}(\bigcup_{i_1,\ldots,i_l} \{\bigwedge_{j \leq l} X_{i_j} = k\})$
$\leq \sum_{i_1,\ldots,i_l} \mathbf{Pr}(\{\bigwedge_{j \leq l} X_{i_j} = k\})$
$= \sum_{i_1,\ldots,i_l} \mathbf{Pr}(\{X_{i_1} = k\}) \times \ldots \times \mathbf{Pr}(\{X_{i_l} = k\}) = \sum_{i_1,\ldots,i_l} (1/n)^l$
les précédentes égalités d'après nos hypothèses probabilistes
$= C_n^l (1/n)^l = \frac{\prod_{i=0}^{l-1} n-i}{n^l} \times \frac{1}{l!} < \frac{1}{l!} < 1/2 \times (e/l)^l$
en utilisant le lemme 1

c.q.f.d. ◊◊◊

Nous en déduisons une garantie probabiliste sur la longueur de la plus grande liste.

Proposition 2 *Soit* $n \geq 100$*, alors :*
$$\mathbf{Pr}(\{L \geq 3\log(n)/\log(\log(n))\}) < 1/2n$$

Preuve
$\mathbf{Pr}(\{L \geq l\}) = \mathbf{Pr}(\bigcup_k \{L_k \geq l\}) \leq \sum_k \mathbf{Pr}(\{L_k \geq l\}) < (n/2) \cdot (e/l)^l$
en utilisant la proposition précédente

Choisissons $l = 3\log(n)/\log(\log(n))$.

$\mathbf{Pr}(\{L \geq l\}) < (n/2)(\frac{e\log(\log(n))}{3\log(n)})^{3\log(n)/\log(\log(n))}$
$\leq (n/2)(\frac{\log(\log(n))}{\log(n)})^{3\log(n)/\log(\log(n))}$
puisque $e < 3$
$= (1/2)e^{\log(n)}(e^{\log(\log(\log(n)))-\log(\log(n))})^{3\log(n)/\log(\log(n))}$
$= (1/2)e^{\log(n)}(e^{3\log(n)\log(\log(\log(n)))/\log(\log(n))-3\log(n)})$
$= (1/2)e^{-2\log(n)+3\log(n)\log(\log(\log(n)))/\log(\log(n))}$

Remarquons que $\log(\log(\log(n)))/\log(\log(n))$ est une fonction décroissante et que $\log(\log(100)) \geq 1.5$ et $\log(\log(\log(100))) \leq 0.5$.

Donc dès que $n \geq 100$,
$(1/2)e^{-2\log(n)+(3\log(n)\log(\log(\log(n))))/\log(\log(n))} \leq (1/2)e^{-2\log(n)+\log(n)} = 1/2n$.

c.q.f.d. ◊◊◊

3.1.4 Choix de la fonction de hachage

Lors de notre analyse probabiliste, nous avons fait des hypothèses probabilistes sur le comportement de la fonction de hachage. Ces hypothèses correspondent à une fonction de hachage « idéale ».

Dans la pratique, on a recours à une analyse statistique sur un échantillon initial de l'annuaire pour déterminer la meilleure fonction de hachage. Dans certains cas, il est facile de déterminer qu'une fonction de hachage ne convient pas. Ainsi choisir la première lettre du nom est un mauvais choix car il est bien connu que la distribution des premières lettres n'est pas équiprobable (comparez par exemple dans votre groupe de TD le nombre d'étudiants dont le nom commence par 'X' et par 'D'). De la même façon, si un identifiant numérique se termine très souvent par un chiffre pair, il n'est pas judicieux d'effectuer un modulo par rapport à un nombre pair. Afin d'éviter ce genre de problème, on choisit très souvent un modulo par rapport à un nombre premier.

3.2 Rappels d'arithmétique

Les fonctions de hachage sont très souvent construites à partir de l'opérateur *modulo*. Afin de faciliter les développements des sections suivantes, nous rappelons maintenant quelques notions d'arithmétique.

Définition 3 *$\mathbb{Z}/n\mathbb{Z}$ (pour n entier non nul) est l'ensemble des entiers compris entre 0 et $n-1$ muni des opérations usuelles que nous indiçons par n pour éviter les ambiguïtés :*

$$a +_n b \equiv (a + b) \mod n$$

$$a -_n b \equiv (a - b) \mod n$$

$$a \times_n b \equiv (a \times b) \mod n$$

Les tables de la figure 3.3 illustrent l'addition et la multiplication dans $\mathbb{Z}/5\mathbb{Z}$. Soit $a \in \mathbb{Z}$, on note $(a)_n \equiv a \mod n$.

Proposition 3 $(\mathbb{Z}/n\mathbb{Z}, +_n, \times_n)$ *est un* anneau commutatif*, c'est à dire :*
- $\forall a, b \in \mathbb{Z}/n\mathbb{Z}, a +_n b = b +_n a$
  (commutativité de l'addition)
- $\forall a, b, c \in \mathbb{Z}/n\mathbb{Z}, (a +_n b) +_n c = a +_n (b +_n c)$
  (associativité de l'addition)
- $\forall a \in \mathbb{Z}/n\mathbb{Z}, a +_n 0 = a$
  (existence d'un élément neutre)
- $\forall a \in \mathbb{Z}/n\mathbb{Z}, a +_n (-a)_n = 0$
  (existence d'un opposé)
- $\forall a, b \in \mathbb{Z}/n\mathbb{Z}, a \times_n b = b \times_n a$
  (commutativité de la multiplication)
- $\forall a, b, c \in \mathbb{Z}/n\mathbb{Z}, (a \times_n b) \times_n c = a \times_n (b \times_n c)$
  (associativité de la multiplication)
- $\forall a, b, c \in \mathbb{Z}/n\mathbb{Z}, (a +_n b) \times_n c = (a \times_n c) +_n (b \times_n c)$
  (distributivité de la multiplication par rapport à l'addition)

Preuve
Nous faisons uniquement la preuve de la dernière assertion car les autres preuves sont similaires. Calculons $((a + b) \times c) \mod n$ de deux façons différentes.

$((a + b) \times c) \mod n = (((a + b)_n + k \times n) \times c) \mod n$
pour un certain k
$= ((a + b)_n \times c + k \times n \times c) \mod n$
distributivité dans $\mathbb{Z}$
$= ((a + b)_n \times c) \mod n$
définition du modulo
$= (a +_n b) \times_n c$

$((a + b) \times c) \mod n = (a \times c + b \times c) \mod n$
distributivité dans $\mathbb{Z}$
$= ((a \times c)_n + k \times n + (b \times c)_n + k' \times n) \mod n$
pour un certain k *et un certain* k'
$= ((a \times c)_n + (b \times c)_n) \mod n$
définition du modulo
$= a \times_n c +_n b \times_n c$

c.q.f.d. $\Diamond\Diamond\Diamond$

Etendre la division est plus délicat et n'est possible que si n est premier. Nous nous appuyons pour cela sur l'algorithme d'Euclide (énoncé ici sous forme de proposition).

Proposition 4 *Soit* a *et* b *deux nombres entiers strictement positifs et* d *leur p.g.c.d. (plus grand commun diviseur), alors* $\exists u, v \in \mathbb{Z}, au + bv = d$.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

×	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

FIG. 3.3: Tables d'addition et de multiplication dans $\mathbb{Z}/5\mathbb{Z}$

Preuve
Nous effectuons une preuve par induction sur $a + b$.

Si $a + b = 2$ alors $a = b = d = 1$ et on prend $u = 0$ et $v = 1$.

Supposons $a \geq b$. Nous effectuons la divison entière de a par b, *i.e.*, $a = bq + r$ avec $0 \leq r < a$. Si $r = 0$ alors b divise a, par conséquent $d = b$ et on prend $u = 0$ et $v = 1$.

Sinon d est aussi le p.g.c.d. de b et de r *(démontrez-le)*. Puisque $r < a$, $r + b < a + b$. En appliquant l'hypothèse d'induction, il existe u', v' entiers relatifs tels que $ru' + bv' = d$, soit en réécrivant r en fonction de a et b, $(a - bq)u' + bv' = d$ ou encore $au' + b(v' - qu') = d$. On choisit donc $u = u'$ et $v = v' - qu'$.

c.q.f.d. ◊◊◊

Proposition 5 *Soit $a \in \mathbb{Z}/p\mathbb{Z}$ avec $a \neq 0$ et p un nombre premier, alors :* $\exists! x, a \times_p x = 1$. *On note a^{-1} cet élément x.*

Preuve
a et p sont premiers entre eux. Donc il existe u et v tels que $au + pv = 1$. En transcrivant cette égalité *modulo* p, on obtient : $(au + pv)_p = (1)_p$, soit $(a)_p \times_p (u)_p + (p)_p \times_p (v)_p = 1$ et finalement $a \times_p (u)_p = 1$.

Montrons que $(u)_p$ est l'unique nombre $x \in \mathbb{Z}/p\mathbb{Z}$ qui vérifie $a \times_p x = 1$. Multiplions cette equation par $(u)_p$. On obtient : $(u)_p \times_p (a \times_p x) = (u)_p$. Par associativité et commutativité, ceci nous donne : $x = (u)_p$.

c.q.f.d. ◊◊◊

Par exemple dans $\mathbb{Z}/5\mathbb{Z}$, $2^{-1} = 3$ et $4^{-1} = 4$ (voir la figure 3.3).

3.3 Gestion d'un dictionnaire par table de hachage

Un dictionnaire est un annuaire dont la fréquence de modification est très faible devant la fréquence des consultations. Par exemple, les correcteurs orthographiques des traitements de texte ont recours à un dictionnaire.

Dans la mesure où les données du dictionnaire sont pratiquement inchangées durant une longue période, il est intéressant de choisir la fonction de hachage

parmi un ensemble de fonctions de telle sorte que la taille de la plus longue liste obtenue par la fonction retenue soit minimale.

Nous allons illustrer ce principe à l'aide d'un dictionnaire dont les identifiants sont numériques. Plus précisément, nous faisons l'hypothèse que le dictionnaire est constitué de n enregistrements dont les identifiants sont compris entre 0 et $p-1$ avec p premier (cette condition étant facile à satisfaire). De plus, nous désirons stocker ce dictionnaire dans une table de hachage à m entrées avec $m \leq p$ (généralement $m \ll p$).

Etant donnés deux identifiants $id \neq id'$, ont dit que id et id' provoquent une collision (par rapport à une fonction de hachage) s'ils ont même image par la fonction de hachage. Notons qu'une fonction de hachage provoque au plus $\frac{n(n-1)}{2}$ collisions et ceci dans le cas où les n enregistrements sont rangés dans la même liste. A l'inverse si $n \leq m$, il se peut qu'une fonction de hachage ne provoque aucune collision.

Le fonction de hachage que nous recherchons se trouve parmi l'ensemble $H = \{h_{a,b} \mid 0 < a < p \wedge 0 \leq b < p\}$ où :

$$h_{a,b}(x) = (((ax + b) \mod p) \mod m) + 1$$

Nous observons que $\mathbf{Card}(H) = p(p-1)$. Dans un premier temps, nous cherchons à majorer *nbcoll*, le nombre total des collisions produites toutes les fonctions de H :

$$nbcoll = \sum_{id < id'} \mathbf{Card}(\{(a,b) \mid h_{a,b}(id) = h_{a,b}(id')\})$$

Proposition 6 *Soient id et id' deux identifiants différents et $h_{a,b} \in H$, alors : $a \times_p id +_p b \neq a \times_p id' +_p b$.*

Preuve
Par l'absurde, supposons : $a \times_p id +_p b = a \times_p id' +_p b$. On soustrait b. D'où : $a \times_p id = a \times_p id'$. Puis en multipliant par a^{-1} :
$id = id'$. Ce qui est contraire à l'hypothèse.

c.q.f.d. $\diamond\diamond\diamond$

Proposition 7 *Soient id et id' deux identifiants différents, alors :*

$$\forall 0 \leq u \neq v < p, \exists! h_{a,b} \in H, a \times_p id +_p b = u \wedge a \times_p id' +_p b = v$$

Preuve
Après soustraction d'une équation par l'autre puis multiplication par $(id -_n id')^{-1}$ on obtient :

$$a = (u -_p v) \times_p (id -_p id')^{-1} \wedge b = u -_p a \times_p id = v -_p a \times_p id'$$

c.q.f.d. $\diamond\diamond\diamond$

Nous sommes maintenant en mesure de majorer le nombre total de collisions.

Proposition 8 *Le nombre total de collisions vérifie l'inégalité suivante :*

$$nbcoll \leq (n(n-1)/2) \cdot \frac{p(p-1)}{m} \tag{3.1}$$

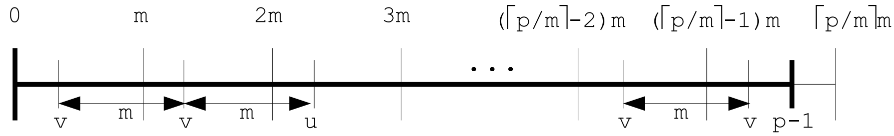

FIG. 3.4: Les couples (u, v) sources de collision

Preuve
Pour que $h_{a,b}(id) = h_{a,b}(id')$ il faut et il suffit que $a \times_p id +_b = u$, $a \times_p id' +_p b = v$ avec $(u - v)_m = 0$. D'après la proposition 6, $u \neq v$.

Il y a au plus $p(\lceil \frac{p}{m} \rceil - 1) \leq p(\frac{p+m-1}{m} - 1) \leq \frac{p(p-1)}{m}$ couples (u, v) qui vérifient ces relations (voir la figure 3.4).

D'après la proposition 7, chaque couple (u, v) est « atteint » par exactement une fonction de hachage. Par conséquent,

$$nbcoll = \sum_{id<id'} \mathbf{Card}(\{(a, b) \mid h_{a,b}(id) = h_{a,b}(id')\}) \leq (n(n-1)/2) \cdot \frac{p(p-1)}{m}$$

c.q.f.d. ◊◊◊

Définition 4 *$H' \subseteq H$ est le sous-ensemble de fonctions qui produisent au moins une collision.*

Proposition 9 *Lorsque $m = n^2$,* $\mathbf{Card}(H') < (1/2)\mathbf{Card}(H)$

Preuve
En remplaçant m par n^2 dans l'équation 3.1, on obtient $nbcoll < p(p-1)/2$. Sachant qu'une fonction de H' produit au moins une collision, $\mathbf{Card}(H') \leq nbcoll < p(p-1)/2$. Le résultat découle alors du fait que $\mathbf{Card}(H) = p(p-1)$.

c.q.f.d. ◊◊◊

Par conséquent, plus de la moitié des fonctions ne produisent aucune collision ! Pour trouver l'une d'entre elles, il suffit d'appliquer successivement chaque fonction de hachage jusqu'à ce que l'une d'entre elles ne produise pas de collision. Cette recherche est présentée dans l'algorithme 14.

Comme plus de la moitié d'entre elles vérifient cette propriété, une telle fonction sera trouvée rapidement. Nous allons effectuer une analyse probabiliste simple pour majorer le nombre moyen d'itérations de la boucle externe de l'algorithme 14. Appelons *Nbtest* la v.a. associée à ce nombre d'itérations. Nous faisons ici l'hypothèse que la fonction `Extraire` choisit de manière équiprobable n'importe laquelle des fonctions non encore examinées.

Rappelons ici un résultat élémentaire de probabilités discrètes.

Proposition 10 *Soit X une v.a. à valeurs dans $\{1, \ldots, n\}$, alors :*
$$\mathbf{E}(X) = \sum_{i=1}^{n} \mathbf{Pr}(X \geq i)$$

Preuve
$\mathbf{E}(X) = \sum_{i=1}^{n} i \cdot \mathbf{Pr}(X = i) = \sum_{i=1}^{n} \sum_{j=1}^{i} \mathbf{Pr}(X = i)$
$= \sum_{j=1}^{n} \sum_{i=j}^{n} \mathbf{Pr}(X = i) = \sum_{j=1}^{n} \mathbf{Pr}(X \geq j)$
L'inversion des sommes correspond à l'égalité suivante :
$\{(i, j) \mid 1 \leq i \leq n, 1 \leq j \leq i\} = \{(i, j) \mid 1 \leq j \leq n, j \leq i \leq n\}$

c.q.f.d. $\Diamond\Diamond\Diamond$

Par conséquent la probabilité d'un échec lors d'une itération est toujours inférieure à $1/2$ (cette probabilité décroit après chaque échec). Donc, la probabilité d'avoir au moins i échecs est inférieure à $(1/2)^i$.

Ce qui nous conduit à :
$$\mathbf{E}(Nbtest) = \sum_{i=1}^{\mathbf{Card}(H)} \mathbf{Pr}(Nbtest \geq i) < \sum_{i=0}^{\infty} (1/2)^i = 2$$

Algorithme 14 : Recherche d'une fonction de hachage optimale

```
Type : struct enregistrement { id : identifiant ; info : information }
Data : H un ensemble de fonctions de hachage
Rechercher(T) : fonction
Input : T un tableau séquentiel de n enregistrements
Output : une fonction de H qui ne provoque pas de collision entre les
         enregistrements de T
Data : Tbool un tableau de m booléens, trouve un booléen
Data : h ∈ H une fonction de hachage, i, j indices de tableaux
repeat
    h ← Extraire(H)
    for i ← 1 to n do Tbool[i] ← false
    i ← 1 ; trouve ← true
    repeat
        j ← h(T[i].id)
        if Tbool[j] = true then
            trouve ← false
        else
            Tbool[j] ← true ; i ← i + 1
        end
    until i > n or not trouve
    if trouve then return h
until H = ∅
```

Cependant la table de hachage a n^2 entrées. Nous allons obtenir un meilleur résultat avec un « double » hachage.

3.4 Gestion d'un annuaire par « double » hachage

Le double hachage opère de la façon suivante.

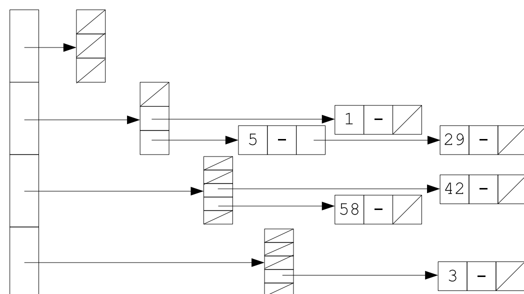

FIG. 3.5: Double hachage

- Chaque entrée de la table de hachage *primaire* est une référence vers une table de hachage *secondaire*.
- Chaque table de hachage *secondaire* possède sa fonction de hachage propre. On note h_i la fonction de hachage associée à l'entrée i de la table primaire et m_i la taille de la table de l'entrée i.
- Lorsqu'un nouvel enregistrement d'identifiant id est ajouté à la table on calcule $i = h(id)$ puis $j = h_i(id)$ afin de déterminer dans quelle entrée de la table de hachage référencée depuis l'entrée i, il doit être inséré. L'insertion se fait comme pour une table de hachage simple.
- La recherche et la suppression suivent un schéma similaire.

La figure 3.5 présente une table de hachage dans le cas d'identifiants numériques, avec :
- $m = 4$ et $h(x) = (x \mod 4) + 1$,
- $\forall 1 \leq i \leq 2, m_i = 3 \wedge h_i(x) = (x \mod 3) + 1$ et
- $\forall 3 \leq i \leq 4, m_i = 5 \wedge h_i(x) = (x \mod 5) + 1$.

L'algorithme 15 développe ces idées. Il suppose que les tables secondaires ont déjà été créées (car ce sont des objets dynamiques).

3.5 Gestion d'un dictionnaire par « double » hachage

Nous rappelons brièvement les hypothèses faites à propos du dictionnaire et l'inéquation 3.1. Le dictionnaire est constitué de n enregistrements dont les identifiants sont compris entre 0 et $p - 1$ avec p premier. Le dictionnaire est stocké dans une table de hachage à m entrées avec $m \leq p$.

Soit $H = \{h_{a,b} \mid 0 < a < p \wedge 0 \leq b < p\}$ une ensemble de fonctions de hachage avec $h_{a,b}(x) = (((ax + b) \mod p) \mod m) + 1$. Alors le nombre total de collisions *nbcoll* par l'ensemble des fonctions de H vérifie l'inéquation suivante :

$$nbcoll \leq (n(n-1)/2) \cdot \frac{p(p-1)}{m}$$

Définition 5 *$H'' \subseteq H$ est le sous-ensemble des fonctions de hachage qui produisent au moins n collisions.*

Algorithme 15 : Gestion d'un annuaire par « double » hachage

Type : **struct** *enregistrement*
  { id : identifiant ; $info$: information ; $suiv$: référence }
Data : T un tableau de m références telles que $T[i]$ pointe sur un tableau de m_i références initialisées à `NULL`

`Chercher`($unid$) : référence d'un enregistrement
Input : $unid$ l'identifiant d'un enregistrement
Output : une référence de cet enregistrement ou `NULL` en cas d'absence
Data : p une variable référence
Data : i, j deux entiers

$i \leftarrow h(unid)$; $j \leftarrow h_i(unid)$; $p \leftarrow (*T[i])[j]$
while $p \neq$ `NULL` **do**
  if $p \rightarrow id = unid$ **then return** p
  $p \leftarrow p \rightarrow suiv$
end
return `NULL`

`Ajouter`($unid, uneinfo$) : booléen
Input : $(unid, uneinfo)$ un enregistrement à ajouter
Output : un booléen indiquant le statut de l'opération
Data : p une variable référence
Data : i, j deux entiers
if `Chercher`($unid$) $\neq$ `NULL` **then return false**
$i \leftarrow h(unid)$; $j \leftarrow h_i(unid)$; $p \leftarrow (*T[i])[j]$
$(*T[i])[j] \leftarrow$ **new**($enregistrement$)
$(*T[i])[j] \rightarrow id \leftarrow unid$; $(*T[i])[j] \rightarrow info \leftarrow uneinfo$
$(*T[i])[j] \rightarrow suiv \leftarrow p$
return true

`Supprimer`($unid$) : booléen
Input : $unid$ l'identifiant d'un enregistrement à supprimer
Output : un booléen indiquant le statut de l'opération
Data : $p, newp$ deux variables références
Data : i, j deux entiers
$i \leftarrow h(unid)$; $j \leftarrow h_i(unid)$
if $(*T[i])[j] =$ `NULL` **then return false**
if $(*T[i])[j] \rightarrow id = unid$ **then**
  $p \leftarrow (*T[i])[j] \rightarrow suiv$; **delete**($(*T[i])[j]$) ; $(*T[i])[j] \leftarrow p$
  return true
end
$p \leftarrow *T[i])[j]$; $newp \leftarrow p \rightarrow suiv$
while $newp \neq$ `NULL` **do**
  if $newp \rightarrow id = unid$ **then**
    $p \rightarrow suiv = newp \rightarrow suiv$
    delete($newp$)
    return true
  end
  $p \leftarrow newp$; $newp \leftarrow p \rightarrow suiv$
end
return false

Proposition 11 *Lorsque* $m = n$, $\mathbf{Card}(H'') < (1/2)\mathbf{Card}(H)$

Preuve
Puisque chaque fonction de H'' produit au moins n collisions :

$$n\mathbf{Card}(H'') \leq \sum_{id<id'} \mathbf{Card}(\{(a,b) \mid h_{a,b}(id) = h_{a,b}(id') \wedge h_{a,b} \in H''\})$$

$$\leq \sum_{id<id'} \mathbf{Card}(\{(a,b) \mid h_{a,b}(id) = h_{a,b}(id') \wedge h_{a,b} \in H\}) = nbcoll$$

Lorsque $m = n$, l'inégalité 3.1 devient $nbcoll \leq (n-1)p(p-1)/2$. D'où :

$$n\mathbf{Card}(H'') \leq \frac{(n-1)p(p-1)}{2}$$

Par conséquent :

$$\mathbf{Card}(H'') < \frac{p(p-1)}{2} = (1/2)\mathbf{Card}(H)$$

c.q.f.d. $\Diamond\Diamond\Diamond$

Par conséquent, plus de la moitié des fonctions de hachage produisent moins de n collisions. Il est donc rapide de trouver une telle fonction h avec un algorithme similaire à l'algorithme 14. Choisissons l'une de ces fonctions h comme fonction de hachage de la table primaire.

Notons n_i le nombre d'identifiants id du dictionnaire tels que $h(id) = i$. De manière évidente, le nombre de collisions produites par la fonction de hachage est $\sum_i n_i(n_i - 1)/2$ ($< n$ par construction).

Nous choisissons une table secondaire pour l'entrée i de la table primaire de taille $m_i = \max(n_i^2, 1)$ et une fonction de hachage h_i qui ne produit pas de collision pour les n_i enregistrements à insérer dans la table. Cette fonction est rapide à trouver d'après les résultats de la section 3.3. Nous obtenons ainsi une table de double hachage sans collisions.

Calculons maintenant la taille des entrées des différentes tables de hachage. La table primaire occupe n entrées. Les tables secondaires occupent :

$$\sum_{i=1}^{n} \max(n_i^2, 1) \leq \sum_{i=1}^{n} (n_i^2 + 1)$$

$$= n + \sum_{i=1}^{n} n_i + \sum_{i=1}^{n} n_i(n_i - 1)$$

$$= 2n + \sum_{i=1}^{n} n_i(n_i - 1)$$

$$< 2n + 2n = 4n$$

Il y a donc au plus $5n$ entrées à comparer avec n^2 entrées dans le cas d'une unique table de hachage sans collision.

Chapitre 4

Gestion dynamique des données : arbres binaires

Objectifs. Présenter des structures dynamiques qui tirent profit de l'ordre sur les identifiants, pour améliorer les performances des accès.

4.1 Gestion d'un annuaire par un arbre binaire

4.1.1 Principe d'un arbre binaire

Comme nous l'avons vu en TD, la recherche d'un enregistrement dans un tableau de n enregistrements où les enregistrements sont triés par identifiant s'effectue en $O(\log(n))$. Le principe des arbres binaires de recherche est de maintenir une structure dynamique « triée » accessible par son « milieu » d'où il est possible d'accèder soit à la « moitié » inférieure des enregistrements, soit à la « moitié » supérieure des enregistrements. De la même façon, ces « moitiés » sont aussi accessibles par leur « milieu ».

Ceci nous conduit naturellement à la structure suivante :
struct *enregistrement* {
id : identifiant
$info$: information
$suivG, suivD$: référence
}

L'enregistrement contient l'identifiant et les informations de l'enregistrement ainsi que deux références *suivG* et *suivD* sur des enregistrements. Le point clef est que tous les enregistrements accessibles par *suivG* (resp. *suivD*) ont un identifiant plus petit (resp. plus grand) que l'identifiant de l'enregistrement.

La figure 4.1 présente un arbre binaire de recherche. L'axe horizontal sur lequel nous avons reporté à la verticale les identifiants montre l'intuition qui se cache derrière cette structure : les identifiants sont rangés « de gauche à droite » par ordre croissant.

Expliquons comment s'effectue la recherche d'un identifiant dans un arbre binaire de recherche. Celle-ci se déroule de manière similaire à la recherche dans

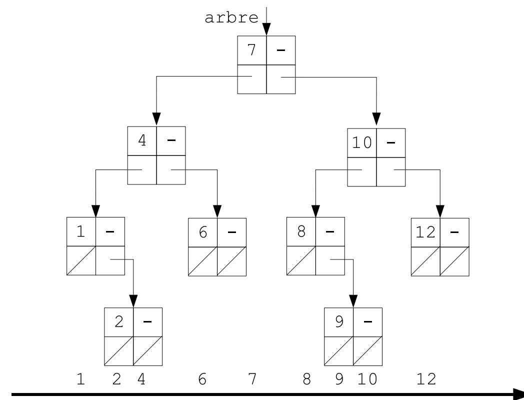

Fig. 4.1: Un arbre binaire de recherche

un tableau trié. Si l'arbre est vide alors la recherche est infructueuse. Sinon, on compare l'identifiant de l'enregistrement à l'identifiant de la racine. Si les deux identifiants sont égaux alors on renvoie une référence sur l'enregistrement de la racine. Si l'identifiant recherché est plus petit (resp. plus grand) on se déplace en suivant la référence vers le *sous-arbre* gauche (droit) et on procède ainsi itérativement.

Le cas de l'ajout est traité de manière analogue. Il s'agit de savoir où l'enregistrement doit être inséré. L'algorithme utilise deux variables p et new, p est le *père* de $newp$, excepté initialement lorsque $newp$ référence la racine (qui n'a pas de père). Dans ce cas, p est égal à `NULL`. On effectue le même parcours de l'arbre que pour une recherche et on n'insère l'enregistrement que si la recherche a été infructueuse. Dans ce cas, $newp$ est égal à `NULL` et p référence le futur père du nouvel enregistrement.

Il convient d'examiner trois cas. p est `NULL` ; par conséquent, l'arbre est vide et l'enregistrement est inséré à la *racine* de l'arbre. L'identifiant de p est plus grand (resp. plus petit) que l'identifiant de l'enregistrement à insérer ; par conséquent, l'enregistrement est inséré comme *fils gauche* (resp. *fils droit*) de l'enregistrement référencé par p. Puisque la recherche a été infructueuse, on est assuré que le sous-arbre gauche (resp. droit) de l'enregistrement référencé par p est vide. La recherche et l'ajout sont décrits par l'algorithme 16.

La structure décrite est appelée arbre car en retournant la figure représentant cette structure, elle présente une analogie avec un arbre. La racine se trouve au niveau le plus bas et le tronc se subdivise (éventuellement) pour former des sous-arbres. Les enregistrements qui n'ont ni sous-arbre gauche, ni sous arbre droit sont appelés des *feuilles*. Les enregistrement sont aussi appelés *noeuds* car ils sont à l'origine des embranchements.

Si la figure n'est pas retournée alors elle ressemble à un arbre généalogique (sexiste) où seuls les hommes et les fils (au plus deux) sont présentés. Chaque enregistrement sauf la racine a un père et chaque enregistrement a au plus deux fils. Notons qu'être fils gauche ou droit n'est pas équivalent.

Algorithme 16 : Gestion d'un annuaire par un arbre

Type : **struct** *enregistrement*
  { id : identifiant ; $info$: information ; $suivG, suivD$: référence }
Data : $arbre$ une référence pointant sur la racine de l'annuaire, initialisée à `NULL`

`Chercher`($unid$) : référence d'un enregistrement
Input : $unid$ l'identifiant d'un enregistrement
Output : une référence de cet enregistrement ou `NULL` s'il est absent de l'annuaire
Data : p une variable référence

```
p ← arbre
while p ≠ NULL do
    if p→id = unid then return p
    else if p→id > unid then p ← p→suivG
    else p ← p→suivD
end
return NULL
```

`Ajouter`($unid, uneinfo$) : booléen
Input : $unid$ l'identifiant d'un enregistrement à ajouter, $uneinfo$ ses informations
Output : un booléen indiquant si l'ajout s'est déroulé correctement
Data : $p, newp$ deux variables références

```
newp ← arbre ; p ← NULL
while newp ≠ NULL do
    p ← newp
    if p→id = unid then return false
    else if p→id > unid then newp ← p→suivG
    else newp ← p→suivD
end
newp ← new(enregistrement)
newp→id ← unid ; newp→info ← uneinfo
newp→suivG ← NULL ; newp→suivD ← NULL
if p = NULL then arbre ← newp
else if p→id > unid then p→suivG ← newp
else p→suivD ← newp
return true
```

4.1.2 Affichage trié d'un arbre binaire

Comme nous l'avons déjà indiqué, un arbre est implicitement trié. Nous le vérifions avec l'algorithme 17 qui affiche les enregistrements dans l'ordre croissant des identifiants. L'algorithme procède ainsi : il affiche par un appel récursif les enregistrements du sous-arbre gauche, puis la racine et enfin, par un deuxième appel récursif, les enregistrements du sous-arbre droit.

Analysons sa complexité. Pour chaque enregistrement, l'algorithme exécute 5 instructions auquel on ajoute (éventuellement) les 2 instructions lors d'un appel infructueux pour un sous arbre vide, soit dans le pire des cas 9 instructions par enregistrement (et dans le meilleur des cas 5 instructions). Par conséquent, si $n > 0$ est le nombre d'enregistrements alors l'algorithme exécute au plus $9n$ instructions (et au moins $5n$ instructions). Sa complexité est en donc en $\Theta(n)$ comme l'affichage (trié) d'un tableau trié.

Cet algorithme présente néanmoins deux inconvénients. D'une part, l'espace occupé par l'algorithme est proportionnel aux nombre d'appels simultanés *i.e.* à la hauteur de l'arbre plus une unité (voir la définition 6). D'autre part, le mécanisme de retour d'appel de fonction lors d'une « remontée » par $suivD$ conduit à une visite inutile d'un noeud. La structure de données et l'algorithme associés, proposés au TD6, remédient à ces deux inconvénients.

Algorithme 17 : Affichage d'un arbre

```
Afficher(p)
Input : p une référence d'arbre
if p ≠ NULL then
    Afficher (p→suivG)
    print p→id, p→info
    Afficher (p→suivD)
end
return
```

4.1.3 Suppression dans un arbre binaire

La suppression d'un enregistrement d'un arbre binaire est nettement plus difficile que dans le cas des structures précédentes. Afin de mettre en oeuvre cette suppression, nous décomposons le problème en quatre cas, du plus simple au plus complexe.

Cas n° 1 (voir la figure 4.2) L'enregistrement (désigné par *oldp* dans l'algorithme 18) a un sous-arbre gauche vide. Il suffit alors de rattacher le sous-arbre droit au père de *oldp*, appelé p dans l'algorithme 18, en lieu et place de celui-ci.

Cas n° 2 (voir la figure 4.3) Ce deuxième cas est très similaire au précédent. L'enregistrement (désigné par *oldp* dans l'algorithme 18) a un sous-arbre droit vide. Il suffit alors de rattacher le sous-arbre gauche au père de *oldp*, appelé p dans l'algorithme 18, en lieu et place de celui-ci.

Les deux cas restants nécessitent l'introduction d'une fonction auxiliaire. Cette fonction, appelée `PluspetitD` dans l'algorithme 18, prend en entrée un enregistrement dont le sous-arbre droit est non vide et renvoie le père du plus

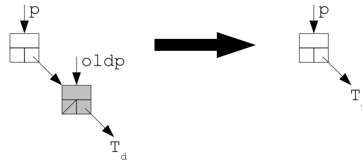

FIG. 4.2: Premier cas de suppression

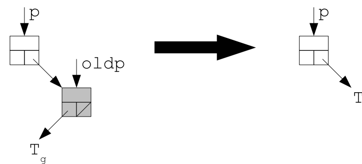

FIG. 4.3: Deuxième cas de suppression

petit enregistrement de ce sous-arbre. Expliquons son fonctionnement. L'enregistrement le plus petit d'un arbre se trouve en suivant les fils gauches jusqu'à ce que l'un d'eux soit `NULL`. C'est ce que fait cette fonction en conservant dans *p* le père du plus petit enregistrement désigné par *newp*.

Si on n'entre pas dans les deux cas précédents, l'enregistrement à supprimer a ses deux sous-arbres non vides. On appelle alors la fonction `PluspetitD` avec pour entrée l'enregistrement à supprimer.

Cas n° 3 (voir la figure 4.4) Si le résultat de l'appel à la fonction (désigné par *pere* dans l'algorithme 18) est l'enregistrement à supprimer (*oldp*), cela signifie que son fils droit a un sous-arbre gauche vide. Dans ce cas il suffit de rattacher le sous-arbre gauche de *oldp* à son fils droit (appelé *newp*) puis de rattacher *newp* à *p* le père de l'enregistrement à supprimer, en lieu et place de celui-ci.

Cas n° 4 (voir la figure 4.5) Le dernier cas nécessite deux opérations conjointes. *newp*, étant le petit élément du sous-arbre droit de *oldp*, va être détaché de son père (*pere*) pour être attaché à *p* en lieu et place de *oldp* en « héritant » des sous-arbres de *oldp*. Il ne faut cependant pas oublier que *newp* a peut-être un

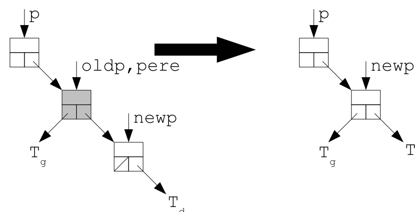

FIG. 4.4: Troisième cas de suppression

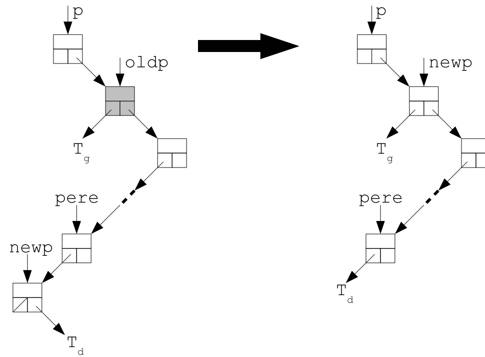

FIG. 4.5: Quatrième cas de suppression

sous-arbre droit non vide (mais un sous-arbre gauche vide). Aussi ce sous-arbre droit sera rattaché à son père en lieu et place de *newp*.

Quelque soit le cas étudié précédemment, au moment du rattachement de *newp* à *p*, il y a trois possibilités :

- *p* est `NULL`, ce qui signifie que l'enregistrement supprimé était la racine. *newp* devient la nouvelle racine.
- *p* référence un enregistrement plus petit que celui référencé par *newp*. *newp* est rattaché à « droite » de *p* (cas représenté dans les figures).
- *p* référence un enregistrement plus grand que celui référencé par *newp*. *newp* est rattaché à « gauche » de *p*.

A titre d'exemple, l'arbre résultant de la suppression de l'enregistrement d'identifiant 7 dans l'arbre de la figure 4.1 est présenté dans la figure 4.6.

4.1.4 Complexité des opérations

Afin de mesurer plus précisément la complexité des opérations sur les arbres binaires de recherche, nous introduisons une quantité entière relative à un arbre et à ses noeuds, la *hauteur*.

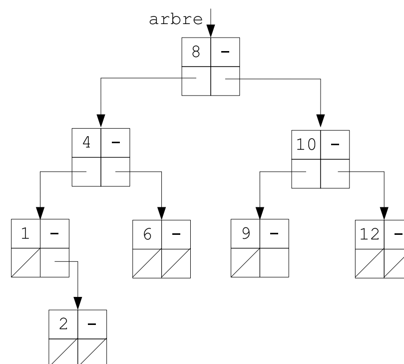

FIG. 4.6: Une suppression dans un arbre (cas n°4)

Algorithme 18 : Suppression dans un arbre

```
PluspetitD(uneref) : référence
Input : uneref une référence d'enregistrement qui a un suivant droit
Output : le « père » du plus petit enregistrement à droite de uneref
Data : p, newp deux variables références
p ← uneref ; newp ← p→suivD
while newp→suivG ≠ NULL do  p ← newp ; newp ← p→suivG
return p

Supprimer(unid) : booléen
Input : unid l'identifiant d'un enregistrement à supprimer
Output : un booléen indiquant le statut de l'opération
Data : oldp, p, newp, pere variables références
newp ← arbre ; p ← NULL
while newp ≠ NULL do
    if newp→id = unid then
        oldp ← newp
        if oldp→suivG = NULL then [1]
            newp ← oldp→suivD
        else if oldp→suivD = NULL then [2]
            newp ← oldp→suivG
        else
            pere ← PluspetitD(oldp)
            if pere = oldp then [3]
                newp ← pere→suivD
                newp→suivG ← oldp→suivG
            else [4]
                newp ← pere→suivG
                pere→suivG ← newp→suivD
                newp→suivG ← oldp→suivG
                newp→suivD ← oldp→suivD

        end
        if p = NULL then arbre ← newp
        else if p→id > unid then p→suivG ← newp
        else p→suivD ← newp
        delete(oldp) ; return true
    end
    p ← newp
    if p→id > unid then newp ← p→suivG
    else newp ← p→suivD
end
return false
```

Définition 6 *La hauteur d'un noeud dans un arbre est définie inductivement par :*

- *La hauteur de la racine est 1.*
- *La hauteur d'un autre noeud est égal à la hauteur de ce noeud dans le sous-arbre de la racine où il est présent incrémentée de 1.*

La hauteur d'un arbre (notée h) est définie par :

- *La hauteur d'un arbre vide est 0.*
- *La hauteur d'un arbre non vide est le maximum de la hauteur de ses noeuds (*i.e.*, le maximum de la hauteur de ses deux sous-arbres incrémenté de 1).*

Analyse au pire des cas

Analysons la recherche d'un arbre dans le cas d'une recherche infructueuse (le pire cas). Elle exécute au plus 4 opérations relatives à un enregistrement avant de continuer la recherche dans un sous-arbre. Autrement dit sa complexité est inférieure ou égale à $4h + 2$ (en comptant la première et la dernière instruction) soit une complexité en $O(h)$. La complexité de l'ajout est similaire à la recherche au plus $5h + 10$ soit aussi une complexité en $O(h)$.

La suppression qui est conceptuellement plus difficile est aussi en $O(h)$ car on « descend » jusqu'à l'élément à supprimer puis (au pire des cas) le long de son sous-arbre droit.

Supposons que l'arbre comporte n enregistrements, la plus grande valeur possible pour h est alors n. En effet, en imaginant une insertion d'enregistrements par ordre croissant, l'arbre est analogue à une liste chaînée par le champ $suivD$.

Analyse en moyenne

Pour apprécier l'intérêt des arbres, il faut procéder à une analyse probabiliste de la hauteur des noeuds vues comme des variables aléatoires. L'hypothèse probabiliste que nous faisons est la suivante.

L'arbre est obtenu par ajout de n enregistrements où l'enregistrement à ajouter est choisi de manière équiprobable parmi les enregistrements restants.

Dans la suite nous notons les identifiants des n enregistrements par $id_1 < \ldots < id_n$. Préalablement à nos résultats, nous caractérisons le fait que deux enregistrements seront comparés lors de la construction de l'arbre.

Lemme 2 *Soient $1 \leq i < j \leq n$, les enregistrements d'identifiants id_i et id_j seront comparés si et seulement si l'un des deux identifiants est le premier à être inséré dans l'annuaire parmi l'ensemble des identifiants compris entre id_i et id_j,* i.e. $ID(i,j) = \{id_i, id_{i+1}, \ldots, id_{j-1}, id_j\}$.

Preuve

Tant qu'aucun enregistrement d'identifiant appartenant à $ID(i,j)$ n'est inséré dans l'arbre, la recherche de tels identifiants conduit au même chemin puisque les tests conduisent aux mêmes résultats. Quand le premier enregistrement d'identifiant de $ID(i,j)$ est inséré, la branche qui y conduit sera ensuite parcourue par tous les autres enregistrements de $ID(i,j)$.

Par conséquent si cet enregistrement a pour identifiant id_i (resp. id_j), alors l'enregistrement d'identifiant id_j (resp. id_j) sera comparé à lui lors de son insertion.

Si à l'inverse cet enregistrement est différent de id_i et de id_j, alors id_i et id_j seront insérés respectivement dans son sous-arbre gauche et droit et ne seront pas comparés.

c.q.f.d. $\Diamond\Diamond\Diamond$

Nous introduisons maintenant quelques variables aléatoires nécessaires à notre raisonnement.

Définition 7 *Soient* $1 \leq i < j \leq n$,

- $X_{i,j}$ *désigne la variable aléatoire qui vaut 1 si* id_i *et* id_j *ont été comparés et* 0 *sinon (par conséquent* $\mathbf{E}(X_{i,j}) = \mathbf{Pr}(X_{i,j} = 1)$*).*
- $h(i)$ *désigne la variable aléatoire correspondant à la hauteur de l'enregistrement d'identifiant* id_i
- $h_m = \frac{1}{n}\sum_{i=1}^{n} h(i)$ *désigne la variable aléatoire correspondant à la hauteur moyenne d'un enregistrement dans un arbre aléatoire.*
- $h_{max} = \max(\{h(i) \mid 1 \leq i \leq n\})$ *désigne la variable aléatoire correspondant à la hauteur d'un arbre aléatoire.*

h_m s'exprime en fonction des $X_{i,j}$ ainsi que l'établit le lemmme suivant.

Lemme 3 $h_m = 1 + \frac{1}{n}\sum_{i<j} X_{i,j}$

Preuve
La hauteur d'un noeud est égale à 1 plus le nombre de comparaisons avec les autres noeuds lors de son insertion. Par conséquent la somme des hauteurs des noeuds est égale à $n + \sum_{i<j} X_{i,j}$. Par conséquent, la hauteur moyenne d'un noeud $h_m = 1 + \frac{1}{n}\sum_{i<j} X_{i,j}$.

c.q.f.d. $\Diamond\Diamond\Diamond$

Avant d'entamer nos raisonnements probabilistes, nous effectuons un bref rappel d'analyse.

Définition 8 *Un fonction* f *de domaine inclus dans* $\mathbb{R}$ *et à valeurs dans* $\mathbb{R}$ *est convexe si :*
$\forall x, y, \forall 0 \leq p_x, p_y$ *t.q.* $p_x + p_y = 1, f(p_x x + p_y y) \leq p_x f(x) + p_y f(y)$

La notion duale de la convexité est la concavité avec l'inégalité inverse.

Lemme 4 *Soit* f *une fonction de domaine inclus dans* $\mathbb{R}$ *et à valeurs dans* $\mathbb{R}$

- *Si* f *est dérivable de dérivée croissante (respectivement décroissante), alors* f *est convexe (respectivement concave).*
- *Si* f *est convexe alors :*
  $\forall x_1, \ldots, x_n, \forall 0 \leq p_1, \ldots, p_n$ *t.q.* $\sum p_i = 1, f(\sum p_i x_i) \leq \sum p_i f(x_i)$.
- *Si* f *est convexe (resp. concave) et* X *une v.a. (ici prenant un nombre fini de valeurs) alors* $f(\mathbf{E}(X)) \leq \mathbf{E}(f(X))$ *(resp.* $f(\mathbf{E}(X)) \geq \mathbf{E}(f(X))$*).*

Preuve
Supposons $x < y$.
$(f(p_x x + p_y y) - f(x))/(p_x x + p_y y - x) = f'(a)$ pour un $a \in [x, p_x x + p_y y]$.
$(f(y) - f(p_x x + p_y y))/(y - p_x x - p_y y) = f'(b)$ pour un $b \in [p_x x + p_y y, y]$.

Puisque f' est croissante :
$(f(p_x x + p_y y) - f(x))/(p_x x + p_y y - x) \leq (f(y) - f(p_x x + p_y y))/(y - p_x x - p_y y)$
Soit $(f(p_x x+p_y y)-f(x))(y-p_x x-p_y y) \leq (f(y)-f(p_x x+p_y y))(p_x x+p_y y-x)$.
Après simplifications :
$f(p_x x+p_y y)y-f(x)(y-p_x x-(1-p_x)y) \leq f(y)((1-p_y)x+p_y y-x)+f(p_x x+p_y y)x$
$f(p_x x + p_y y)(y - x) \leq f(x)(p_x(y - x)) + f(y)(p_y(y - x))$
D'où le résultat en divisant par $y - x$.

L'inégalité se démontre par récurrence. Le cas $n = 2$ est la définition de la convexité. Notons $s = \sum_{i=1}^{n-1} p_i$.
$f(\sum_{i=1}^{n} p_i x_i) = f(s \sum_{i=1}^{n-1} (p_i/s)x_i + p_n x_n)$
$\leq sf(\sum_{i=1}^{n-1} (p_i/s)x_i) + p_n f(x_n)$ *(convexité de f)*
$\leq s \sum_{i=1}^{n-1} (p_i/s)f(x_i) + p_n f(x_n)$ *(hypothèse de récurrence)*
$= \sum_{i=1}^{n} p_i f(x_i)$
La dernière inégalité est une conséquence directe de la précédente en remplaçant les espérances par leur définition.

c.q.f.d. $\Diamond\Diamond\Diamond$

Proposition 12 $2(1 + \frac{1}{n}) \log(n + 1) - 3 \leq \mathbf{E}(h_m) \leq 2(1 + \frac{1}{n})(1 + \log(n)) - 3$

Preuve
D'après le lemme 2, la probabilité que les enregistrements d'identifiants soient comparés est égale à $\frac{2}{j-i+1}$, soit $\mathbf{E}(X_{i,j}) = \frac{2}{j-i+1}$.
Donc :
$\mathbf{E}(h_m) = 1 + \frac{1}{n} \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k}$ *(linéarité de l'espérance)*
$= 1 + \frac{1}{n} \sum_{k=2}^{n} \sum_{i=1}^{n+1-k} \frac{2}{k}$ *(inversion des sommes)*
$= 1 + \frac{1}{n} \sum_{k=2}^{n} \frac{2(n+1-k)}{k}$
$= 1 - \frac{2(n-1)}{n} + \frac{2(n+1)}{n} \sum_{k=2}^{n} \frac{1}{k}$
$= 1 - \frac{2(n-1)}{n} + \frac{2}{n} \sum_{k=2}^{n} \frac{1}{k} + 2 \sum_{k=2}^{n} \frac{1}{k}$
$= -1 + \frac{2}{n} \sum_{k=1}^{n} \frac{1}{k} + 2 \sum_{k=2}^{n} \frac{1}{k}$
$= -3 + 2(1 + \frac{1}{n}) \sum_{k=1}^{n} \frac{1}{k}$

Par conséquent,
$-3 + 2(1 + \frac{1}{n}) \int_1^{n+1} \frac{1}{x}dx \leq \mathbf{E}(h_m) \leq -3 + 2(1 + \frac{1}{n})(1 + \int_1^n \frac{1}{x}dx)$
$2(1 + \frac{1}{n}) \log(n + 1) - 3 \leq \mathbf{E}(h_m) \leq 2(1 + \frac{1}{n})(1 + \log(n)) - 3$.

c.q.f.d. $\Diamond\Diamond\Diamond$

En supposant que les enregistrements sont recherchés avec la même probabilité, $\mathbf{E}(h_m)$ nous fournit le nombre de noeuds visités lors d'une recherche fructueuse. Nous en conclusons que la complexité d'une recherche fructueuse dans un arbre est en $\Theta(\log(n))$.

Les cas de l'ajout et de la recherche infructueuse sont identiques. Supposons que l'on recherche (ou ajoute) un noeud d'identifiant id placé entre id_i et id_{i+1}. La somme des probabilités d'une comparaison avec tous les noeuds nous fournit le nombre moyen de noeuds visités par cette recherche (noté nb). En distinguant selon les identifiants plus petits et plus grands, on obtient :
$nb = \sum_{k=1}^{i} \frac{1}{k} + \sum_{k=1}^{n-i} \frac{1}{k}$
$\leq 1 + \int_1^i \frac{1}{x}dx + 1 + \int_1^{n-i} \frac{1}{x}dx$

$= 2 + \log(i) + \log(n-i) \leq 2 + 2\log(\frac{n}{2})$ *(concavité du* $\log$*)*
$\leq 2\log(n) + 1$

Par conséquent, en moyenne la recherche infructeuse et l'ajout se font aussi en $O(\log(n))$.

Dans ce qui suit, on note h_n la v.a. h_{max} d'un arbre obtenu par ajout aléatoire de n enregistrements et $f_n = 2^{h_n}$ la *hauteur exponentielle* de l'arbre. Nous allons raisonner sur f_n et ceci parce que l'opérateur max apparait dans nos équations de récurrence. Au chapitre précédent, nous avons utilisé une majoration simple $\max(x,y) \leq x+y$. Dans le cas défavorable où $x = y$, on remplace x par $2x$. En passant à l'exponentielle, on obtient : $2^{\max(x,y)} = \max(2^x, 2^y) \leq 2^x + 2^y$. D'où $\max(x,y) \leq \log_2(2^x + 2^y)$. Cette fois-ci dans le cas défavorable où $x = y$, on remplace x par $x + 1$!

Proposition 13 $\forall n, \mathbf{E}(f_n) \leq n^3 + 1$

Preuve
Le premier enregistrement inséré est choisi de manière équiprobable parmi les n enregistrements. S'il s'agit du i^{eme} enregistrement (par ordre croissant), alors son sous-arbre gauche aura $i-1$ enregistrements et une hauteur dont la distribution est celle de h_{i-1} et son sous-arbre droit aura $n-i$ enregistrements et une hauteur dont la distribution est celle de h_{n-i}. Notons $f_{n,i}$ la v.a. représentant la hauteur exponentielle sachant que l'on le premier enregistrement inséré est celui d'identifiant id_i. Alors :
$f_{n,i} = 2^{1+\max(h_{i-1},h_{n-i})} = 2 \cdot 2^{\max(h_{i-1},h_{n-i})} \leq 2(2^{h_{i-1}} + 2^{h_{n-i}}) = 2(f_{i-1} + f_{n-i})$
Par conséquent :
$\mathbf{E}(f_n) = \frac{1}{n}\sum_{i=1}^{n}\mathbf{E}(f_{n,i}) \leq \frac{1}{n}\sum_{i=1}^{n} 2(\mathbf{E}(f_{i-1}) + \mathbf{E}(f_{n-i})) = \frac{4}{n}\sum_{i=0}^{n-1}\mathbf{E}(f_i)$

Vérifions l'inéquation de la proposition par récurrence.
Elle est satisfaite pour $n = 0$ puisque $f_0 = 1$.
Elle est satisfaite pour $n = 1$ puisque $f_1 = 2$.
Soit $n \geq 2$. Supposons-la satisfaite jusqu'à $n-1$. Alors :
$\mathbf{E}(f_n) \leq \frac{4}{n}\sum_{i=0}^{n-1}\mathbf{E}(f_i) \leq \frac{4}{n}\sum_{i=0}^{n-1}(i^3+1) \leq \frac{4}{n}(\int_0^{n-1} x^3 dx + (n-1)^3 + n)$
$= \frac{4}{n}(\frac{(n-1)^4}{4} + (n-1)^3 + n)$
$= \frac{4}{n}(\frac{1}{4}n^4 - n^3 + \frac{3}{2}n^2 - n + \frac{1}{4} + n^3 - 3n^2 + 3n - 1 + n)$
$= \frac{4}{n}(\frac{1}{4}n^4 - \frac{3}{2}n^2 + 3n - \frac{3}{4})$
$\leq \frac{4}{n}(\frac{1}{4}n^4 - \frac{3}{4})$ *(dès que* $n \geq 2$*)*
$\leq n^3 < n^3 + 1$

c.q.f.d. $\diamond\diamond\diamond$

Nous sommes maintenant en mesure de borner la hauteur moyenne d'un arbre.

Proposition 14 $\forall n \geq 1, \mathbf{E}(h_n) \leq 3\log_2(n) + 1$

Preuve
$n^3 + 1 \geq \mathbf{E}(f_n) = \mathbf{E}(2^{h_n}) \geq 2^{\mathbf{E}(h_n)}$ *(convexité de* 2^x *et lemme 4)*
D'où :
$\mathbf{E}(h_n) \leq \log_2(n^3+1) = \log_2(n^3(1+\frac{1}{n^3})) = \log_2(n^3) + \log_2(1+\frac{1}{n^3}) \leq 3\log_2(n)+1$

c.q.f.d. $\diamond\diamond\diamond$

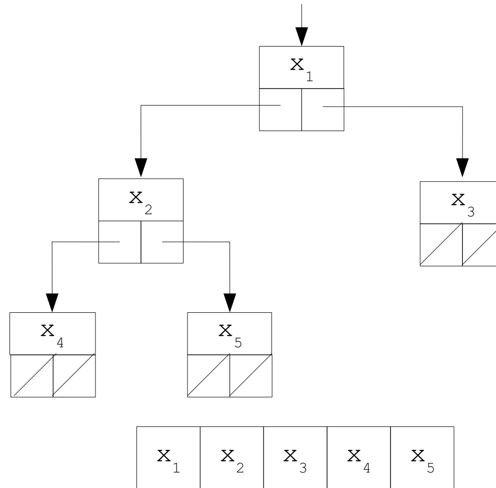

FIG. 4.7: L'arbre implicite correspondant à un tableau de taille 5

En conclusion, les opérations sur les arbres sont beaucoup plus rapides que les opérations sur les listes en $O(n)$. Par contre les opérations sur un arbre sont moins efficaces que celles sur une table de hachage ($O(1)$ en moyenne et même $O(\log(n)/\log(\log(n)))$ avec une probabilité d'au moins $1 - 1/2n$). Rappelons cependant que les tables de hachage nécessitent une allocation statique préalable ce qui n'est pas le cas des arbres.

4.2 Le tri par tournoi

4.2.1 Représentation d'un arbre par un tableau

Un tableau T de dimension n peut être vu comme un arbre de la manière suivante.

- $T[1]$ est la racine.
- $T[i]$ a pour fils gauche $T[2 * i]$ si $2i \leq n$ et pour fils droit $T[2 * i + 1]$ si $2i + 1 \leq n$.

Remarquons que le père de $T[i]$ (pour $i > 1$) est $T[\lfloor i/2 \rfloor]$. La figure 4.7 illustre ce point de vue pour un tableau de taille 5.

Calculons la hauteur de l'arbre ainsi défini. De manière évidente, $T[n]$ est de hauteur maximale. On démontre par récurrence que les noeuds compris entre $T[2^{i-1}]$ et $T[2^i - 1]$ ont une hauteur égale à $i + 1$. Par conséquent, soit l'unique h tel que $2^{h-1} \leq n < 2^h$, alors la hauteur de $T[n]$ est égale à h. En passant au logarithme, on obtient : $h - 1 \leq \log_2(n) < h$. Ce qui se traduit par $h = 1 + \lfloor \log_2(n) \rfloor$. On dira que h est la hauteur de T

4.2.2 Représentation d'un tournoi par un « arbre-tableau »

Nous allons obtenir le plus grand élément du tableau T en organisant un tournoi entre les cellules de T. Pour cela, on utilise un tableau *Tour* de taille

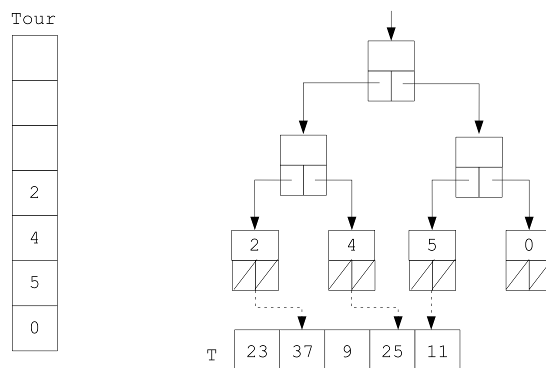

FIG. 4.8: Après la première phase du tournoi

$2^{\lceil \log_2(n) \rceil} - 1$ où $n \geq 2$ est la taille de T. Ce tableau contient des indices de T correspondant au vainqueur d'un match. On remplit ce tableau par indice décroissant. Notons $m = \lceil \log_2(n) \rceil$.

La première phase correspond au premier tour du tournoi.

- Un indice i compris entre 2^{m-1} à $2^m - 1$, correspond *a priori* au vainqueur du match entre $T[2*i - 2^{m-1} + 1]$ et $T[2*i - 2^{m-1} + 2]$.
- Aussi si $2*i - 2^{m-1} + 1 > n$ alors $Tour[i] = 0$ pour indiquer qu'il n'y a pas de vainqueur faute de combattants.
- Si $2*i - 2^{m-1} + 1 = n$ (ce qui implique que n est impair) alors $Tour[i] = n$ puisque l'indice n de T n'a pas d'adversaire.
- Enfin lorsque $2*i - 2^{m-1} + 2 \leq n$, si $T[2*i - 2^{m-1} + 1] \geq T[2*i - 2^{m-1} + 2]$ alors $Tour[i] = 2*i - 2^{m-1} + 1$ sinon $Tour[i] = 2*i - 2^{m-1} + 2$.

La figure 4.8 illustre les « résultats » du tournoi à l'issue du premier tour.

La deuxième phase correspond aux tours suivants du tournoi.

- Un indice i compris entre 1 à $2^{m-1} - 1$, correspond au vainqueur du match entre $T[Tour[2*i]]$ et $T[Tour[2*i+1]]$.
- Si $Tour[2*i+1] = 0$ alors $Tour[i] = Tour[2*i]$ puisque, s'il est non nul, l'indice $Tour[2*i]$ n'a pas d'adversaire.
- Autrement si $T[2*i - 2^h + 1] \geq T[2*i - 2^h + 2]$ alors $Tour[i] = Tour[2*i]$ sinon $Tour[i] = Tour[2*i+1]$.

La figure 4.9 illustre tous les « résultats » du tournoi.

4.2.3 Recherche des éléments par ordre décroissant

A la fin de la deuxième phase, $Tour[1]$ est l'indice de la plus grande cellule de T. Il s'agit maintenant de trouver le deuxième (puis la troisième, etc) plus grande cellule. Nous avons les résultats du tournoi. Aussi l'idée est d'imaginer que $Tour[1]$ a déclaré forfait. Dans ce cas on doit uniquement refaire tous les matchs auxquels $Tour[1]$ a participé pour trouver le nouveau vainqueur. On procède ainsi itérativement.

Les figures 4.10 et 4.11 illustrent les résultats du tournoi réactualisés après le premier et le deuxième « forfaits ».

L'algorithme 19 est le résultat de cette analyse.

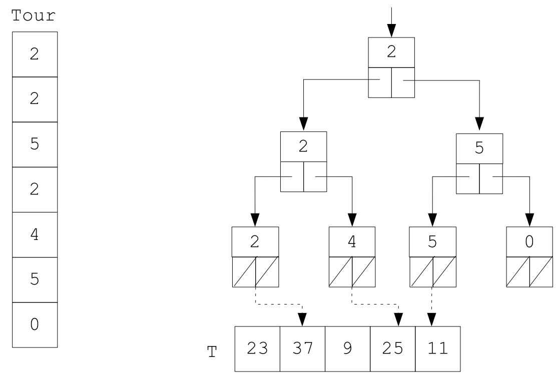

Fig. 4.9: Après la deuxième phase du tournoi

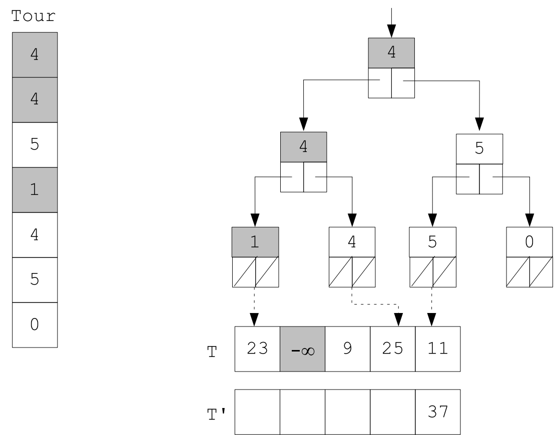

Fig. 4.10: Le tournoi après un forfait

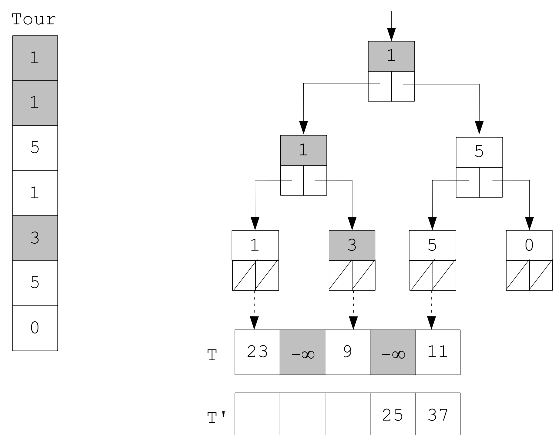

Fig. 4.11: Le tournoi après deux forfaits

Algorithme 19 : Le tri tournoi

```
Trier(T) : tableau de nombres
Input : T un tableau de n ≥ 2 nombres
Output : le tableau trié
Data : T' un tableau de n nombres
Data : Tour un tableau de 2^⌈log2(n)⌉ − 1 nombres
Data : i, j, k, deb des entiers
deb ← 2^(⌈log2(n)⌉−1)
for i ← 2 ∗ deb − 1 downto deb do
|   j ← 2 ∗ i − 2 ∗ deb + 1
|   if j > n then Tour[i] ← 0
|   else if j = n then Tour[i] ← n
|   else if T[j] ≥ T[j + 1] then Tour[i] ← j
|   else Tour[i] ← j + 1
end
for i ← deb − 1 downto 1 do
|   if Tour[2 ∗ i + 1] = 0 then Tour[i] ← Tour[2 ∗ i]
|   else if T[Tour[2 ∗ i]] ≥ T[Tour[2 ∗ i + 1]] then Tour[i] ← Tour[2 ∗ i]
|   else Tour[i] ← Tour[2 ∗ i + 1]
end
for i ← n downto 2 do
|   T'[i] ← T[Tour[1]] ; T[Tour[1]] ← −∞
|   j ← deb + ⌊(Tour[1] − 1)/2⌋
|   k ← 4 ∗ ⌊(Tour[1] + 1)/2⌋ − 1 − Tour[1]
|   if k > n then Tour[j] ← 0
|   else if T[k] = −∞ then Tour[j] ← 0
|   else Tour[j] ← k
|   while j > 1 do
|   |   j ← ⌊j/2⌋
|   |   if Tour[2 ∗ j + 1] = 0 then Tour[j] ← Tour[2 ∗ j]
|   |   else if Tour[2 ∗ j] = 0 then Tour[j] ← Tour[2 ∗ j + 1]
|   |   else if T[Tour[2 ∗ j]] ≥ T[Tour[2 ∗ j + 1]] then
|   |   |   Tour[j] ← Tour[2 ∗ j]
|   |   else Tour[j] ← Tour[2 ∗ j + 1]
|   end
end
T'[1] ← T[Tour[1]]
return T'
```

4.2.4 Complexité du tri tournoi

En reprenant les explications qui ont conduit à l'algorithme du tri tournoi, il apparaît clairement que la complexité est proportionnelle au nombre de matchs auxquels a donné lieu le tournoi.

Pour simplifier les formules, on note $n' = 2^{\lceil \log_2(n) \rceil}$. Remarquons que $n \leq n' < 2n$. La taille du tableau *Tour* est égale à $n' - 1$. Le tournoi qui désigne le vainqueur consiste à effectuer $n' - 1$ matchs. Puis on réactualise le tournoi lors des $n - 1$ forfaits. Chaque réactualisation consiste exactement en $\log_2(n')$ matchs à rejouer. Par conséquent nb, le nombre de matchs effectués, est égal à :

$$nb = n' - 1 + (n - 1)\log_2(n')$$

En utilisant l'encadrement de n', on obtient :

$$(n-1)(\log_2(n) + 1) \leq nb < (n-1)(\log_2(n) + 1) + 2n - 1$$

Par conséquent, $nb = \Theta(n \log(n))$ d'où une complexité équivalente à celle du tri fusion.

4.3 Arbre d'évaluation d'une expression

4.3.1 Principe d'un arbre d'évaluation

Les arbres sont utilisés dans d'autres contextes que ceux des annuaires. Par exemple, une expression arithmétique peut être représentée sous forme d'un arbre (voir la figure 4.12). Afin d'interpréter une expression sans ambiguïté, il faut se donner des règles d'évaluation des opérateurs :

- Il y a des règles de priorité dans l'évaluation des opérateurs. Dans notre cas, nous nous limitons aux deux opérateurs $+$ et $*$ et suivant la règle communément admise nous supposons que le $*$ est prioritaire devant le $+$
- En cas d'égalité de priorité l'opérateur le plus à gauche est évalué d'abord. Dans notre cas, cette règle ne s'applique que pour des opérateurs identiques.
- Une expression entre parenthèses est évaluée indépendamment de son contexte.

Par exemple, dans l'expression $x_2{*}x_5{+}x_1{*}x_3{*}x_1$, le premier $*$ doit être évalué avant le $+$. Autrement dit cette expression est équivalente à $(x_2{*}x_5){+}x_1{*}x_3{*}x_1$. Hors des parenthèses, le $*$ est prioritaire devant le $+$ qui précède et le $*$ qui suit ce qui nous donne $(x_2{*}x_5){+}(x_1{*}x_3){*}x_1$. Enfin le dernier $*$ est prioritaire devant le $+$ ce qui conduit à l'expression parenthésée finale $(x_2 * x_5) + ((x_1 * x_3) * x_1)$.

Dans cette expression, un seul opérateur est externe aux parenthèses. Il constitue la racine de l'arbre d'évaluation. Son sous-arbre gauche (respectivement droit) est obtenu par traduction de l'expression parenthésée qui se situe à sa gauche (respectivement droit).

4.3.2 Evaluation à partir d'un arbre

Lorsqu'on dispose d'un arbre pour une expression et que la valeur des variables est connue (dans notre cas, la valeur de x_i est donnée par $Var[i]$), l'évaluation de l'expression s'effectue de façon naturelle grace à la récursivité. Si

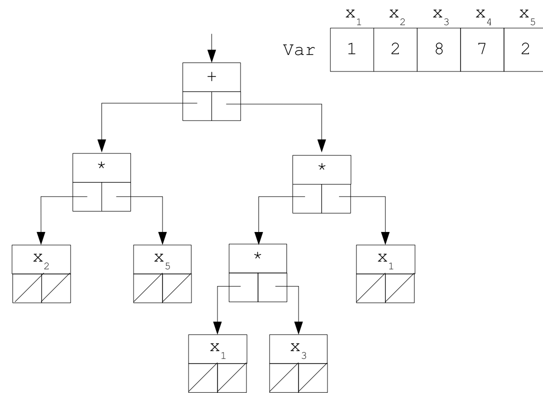

FIG. 4.12: Un arbre correspondant à l'expression $x_2 * x_5 + x_1 * x_3 * x_1$

l'expression est réduite à une variable, on renvoie sa valeur. Sinon on évalue le sous-arbre arbre gauche, le sous-arbre droit et on applique l'opérateur « racine » sur les valeurs trouvées.

Afin de simplifier la gestion des types, -1 représente le $+$, 0 représente le $*$ et $i > 0$ représente la variable x_i. L'algorithme 20 applique le principe d'évaluation décrit précédemment.

Algorithme 20 : Evaluation d'un arbre représentant une expression

---

Type : **struct** *enregistrement* { *op* : entier ; *suivG*, *suivD* : référence }
Constant : *fois* $= 0$; *plus* $= -1$
Data : Var un tableau contenant la valeur des variables

`Evaluer`(p)
Input : p une référence d'arbre d'expression
Data : vg, vd des entiers résultats de l'évaluation de sous-arbres
Data : i un indice
$i \leftarrow p \rightarrow op$
if $i > 0$ **then return** $Var[i]$
$vg \leftarrow$ `Evaluer`($p \rightarrow suivG$)
$vd \leftarrow$ `Evaluer`($p \rightarrow suivD$)
if $i = fois$ **then return** $vg * vd$
return $vg + vd$

Ce parcours d'arbre s'appelle un parcours *postfixe* car on parcourt les sous-arbres avant de d'examiner le noeud courant. Dans le cas de l'affichage d'un arbre binaire de recherche, on a affaire à un parcours *infixe* car on parcourt le sous-arbre gauche, on traite le noeud courant puis on parcourt le sous-arbre droit.

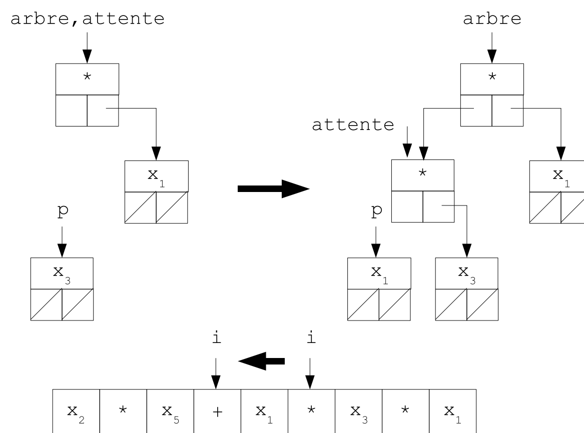

FIG. 4.13: Analyse d'un $*$

4.3.3 Construction d'un arbre d'évaluation

Nous nous tournons maintenant vers un problème plus difficile : comment écrire l'algorithme qui transforme une expression arithmétique en un arbre ? D'après nos explications, il s'agit de trouver l'opérateur à évaluer après tous les autres. Il s'agit donc soit du $+$ le plus à droite soit du $*$ le plus à droite si l'expression ne contient pas de $+$.

Le principe de l'algorithme consiste donc à parcourir l'expression de droite à gauche et à construire ce qui sera soit le sous-arbre droit du $+$ externe soit en cas d'absence de $+$ ce qui sera l'arbre de l'expression. Dans le sous-arbre courant (dénoté par la variable *arbre*), seul le $*$ le plus à gauche (dénoté par la variable *attente*) ne connait pas encore son sous-arbre gauche. Il s'agit soit de la variable qui est à sa gauche (dénotée par p) si l'opérateur à gauche de la variable est un $+$ ou si on est au début de l'expression. Dans le cas contraire, il s'agit de l'opérateur qui se trouve à sa gauche (un $*$) qui aura pour sous-arbre droit p et qui deviendra le $*$ en attente de sous-arbre gauche. Lorsqu'on rencontre le $+$ externe, son sous-arbre gauche est obtenu par un appel récursif limité à la sous-expression qui se trouve à gauche de ce $+$. L'algorithme 21 est le résultat de ces considérations.

Nous illustrons le déroulement de l'algorithme sur l'expression $x_2 * x_5 + x_1 * x_3 * x_1$. La figure 4.13 représente l'état de l'exécution après la lecture du suffixe $x_3 * x_1$. Un arbre incomplet a été bâti car à ce stade, on ne sait encore où rattacher le terme x_3. A la lecture du deuxième $*$, x_3 y est rattaché et cet arbre incomplet constitue le sous-arbre droit du premier $*$. De même, le deuxième x_1 est en attente de rattachement.

La figure 4.14 représente la lecture du $+$. On sait alors que x_1 doit être rattaché au $*$ en attente et que l'arbre courant constitue le sous-arbre droit du $+$. Pour obtenir le sous-arbre gauche, on appelle récursivement l'algorithme de construction appliqué au préfixe $x_2 * x_5$.

La plupart des expressions arithmétiques comportent des parenthèses. Il s'agit donc de les prendre en compte. Examinons comment adapter l'algo-

Algorithme 21 : Construction d'un arbre représentant une expression

```
Type : struct enregistrement { op : entier ; suivG, suivD : référence }
Constant : fois = 0 ; plus = −1
Data : Exp un tableau contenant l'expression à transformer en arbre

Construire(n) : référence
Input : n la taille du sous-tableau à examiner
Output : une référence sur l'arbre de l'expression
Data : arbre, p, q, attente des références
Data : i un indice
p ← new(enregistrement) ; p→op ← Exp[n]
p→suivG ← NULL ; p→suivD ← NULL
if n = 1 then return p
i ← n − 1 ; arbre ← NULL
repeat
    q ← new(enregistrement) ; q→op ← Exp[i]
    if arbre = NULL then
        arbre ← q ; arbre→suivD ← p
        if q→op = plus then
            arbre→suivG ← Construire(i − 1) ; return arbre
        end
        attente ← q
    end
    if q→op = plus then
        attente→suivG ← p ; q→suivD ← arbre ; arbre ← q
        arbre→suivG ← Construire(i − 1) ; return arbre
    end
    q→suivD ← p ; attente→suivG ← q ; attente ← q
    p ← new(enregistrement) ; p→op ← Exp[i − i]
    p→suivG ← NULL ; p→suivD ← NULL
    i ← i − 2
until i=0
attente→suivG ← p ; return arbre
```

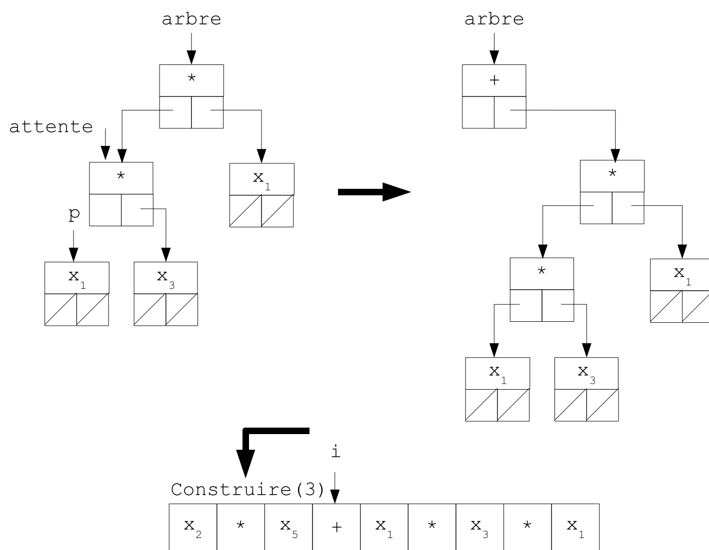

FIG. 4.14: Analyse d'un +

rithme précédent. Lorsque l'algorithme rencontre une parenthèse fermante, il sait qu'elle termine une sous-expression destinée à être évaluée indépendamment du contexte. Aussi il s'appelle récursivement pour construire le sous-arbre correspondant à cette expression et ensuite le traiter comme la « variable qu'il attendait ». D'autre part maintenant, un appel à la fonction se termine lorsqu'on arrive au début de l'expression ou qu'on rencontre une parenthèse ouvrante. Enfin un appel ne peut plus se contenter de renvoyer une référence vers le sous-arbre construit, il doit aussi indiquer à l'appelant où continuer l'analyse de l'expression. Ces considérations conduisent à l'algorithme 22.

Nous illustrons le traitement d'une parenthèse fermante à l'aide de l'expression $x_2 * (x_5 + x_1) * x_3 * x_1$. La figure 4.15 représente l'état avant la lecture de la parenthèse fermante. L'algorithme s'appelle récursivement en « sautant » la parenthèse fermante.

Nous illustrons le traitement de la parenthèse ouvrante correspondant à la parenthèse fermante rencontrée précédemment. La figure 4.15 représente l'état de l'algorithme avant la lecture de x_5. Il y a trois appels en cours : l'appel initial, puis l'appel lors de la rencontre de la parenthèse fermante et enfin l'appel suite à la rencontre du + pour construire son sous-arbre gauche. A la lecture de x_5, l'algorithme construit un noeud. Puisque le symbole suivant est une parenthèse ouvrante, il se termine en renvoyant l'arbre réduit à ce noeud et l'indice de lecture pointant sur le symbole qui précède la parenthèse ouvrante. Le deuxième appel conplète son arbre avec le sous-arbre renvoyé et se termine en renvoyant son arbre et en transmettant l'indice de lecture. Le premier appel affecte ce sous-arbre à p, référence du terme en attente d'attachement et met à jour son indice de lecture avec l'indice renvoyé.

Algorithme 22 : Construction d'un arbre pour une expression avec parenthèses

Type : **struct** *enregistrement* { *op* : entier ; *suivG*, *suivD* : référence }
Constant : $fois = 0$; $plus = -1$; $parouvr = -2$; $parferm = -3$
Data : *Exp* un tableau contenant l'expression à transformer en arbre

`Construire`(n) : référence, entier
Input : n la taille du sous-tableau à examiner
Output : une référence sur l'arbre de l'expression
Output : l'indice de la prochaine cellule à examiner
Data : $arbre, p, q, attente$ des références
Data : i un indice

```
if Exp[n] = parouvr then p, i ← Construire(n − 1)
else
    p ← new(enregistrement) ; p→op ← Exp[n]
    p→suivG ← NULL ; p→suivD ← NULL
    i ← n − 1
end
if i = 0 then return p, 0
if Exp[i] = parouvr then return p, i − 1
arbre ← NULL
repeat
    q ← new(enregistrement) ; q→op ← Exp[i]
    if arbre = NULL then
        arbre ← q ; arbre→suivD ← p
        if q→op = plus then
            arbre→suivG, i ← Construire(i − 1) ; return arbre, i
        end
        attente ← q
    end
    if q→op = plus then
        attente→suivG ← p ; q→suivD ← arbre ; arbre ← q
        arbre→suivG, i ← Construire(i − 1) ; return arbre, i
    end
    q→suivD ← p ; attente→suivG ← q ; attente ← q
    i ← i − 1
    if Exp[i] = parferm then p, i ← Construire(i − 1)
    else
        p ← new(enregistrement) ; p→op ← Exp[i]
        p→suivG ← NULL ; p→suivD ← NULL
        i ← i − 1
    end
until i = 0 or Exp[i] = parouvr
attente→suivG ← p
if i = 0 then return arbre, 0
return arbre, i − 1
```

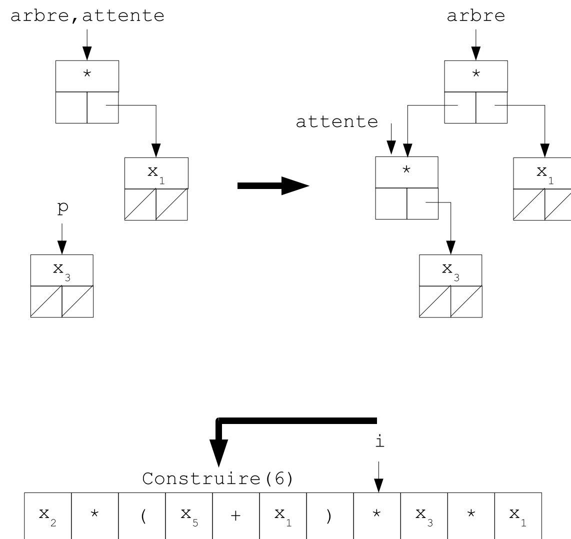

FIG. 4.15: Analyse d'une parenthèse fermante

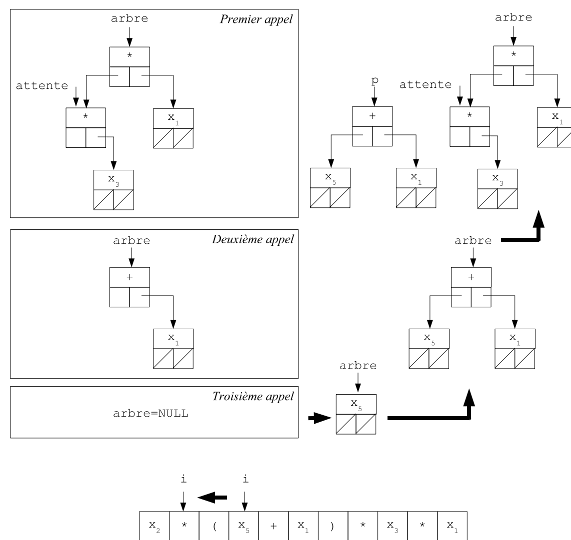

FIG. 4.16: Analyse d'une parenthèse ouvrante