Algorithmique 1

Licence d'informatique (2ème année)

Université d'Aix-Marseille

F. Denis, S. Grandcolas, Y. Vaxès

2012 - 2013

Chapitre 1

Introduction.

Ce cours constitue une introduction à l'algorithmique. Il est destiné aux étudiants de deuxième année de la licence d'informatique de l'université d'Aix-Marseille, supposés avoir déjà des bases solides en programmation.

1.1 Contenu du cours

Le premier chapitre est une introduction à l'analyse des algorithmes et en particulier, aux notions fondamentales de preuves et de complexité des algorithmes. Le second chapitre est consacré aux structures de données linéaires (tableaux, listes, piles, files et tables de hachage). Le troisième chapitre est dédié aux principaux algorithmes de tris et à l'étude de leur complexité. Le quatrième chapitre est consacré aux arbres binaires de recherche, aux tas et aux arbres lexicographiques. Le cinquième chapitre introduit la notion de programmation dynamique qui s'applique dans certains cas aux méthodes de type *diviser pour régner*. Le sixième et dernier chapitre est consacré aux graphes qui interviennent dans la représentation de nombreux problèmes, et pour lesquels un grand nombre d'algorithmes ont été développés, notamment pour le calcul de plus courts chemins. Nous présenterons quelques applications pratiques des graphes, entre autre dans le domaine du routage.

Ce cours est largement inspiré de "Introduction à l'algorithmique" de T. Cormen, C. Leiserson et R. Rivest (éditions DUNOD) qui est la référence pour les cours algorithmique 1 et algorithmique 2 de la licence d'informatique.

Ce support de cours correspond aux cours magistraux, qui sont complétés par des travaux dirigés et des travaux pratiques. Les algorithmes sont donnés en pseudo-code qui permet de s'abstraire de détails d'implémentation (voir

section suivante) ou en C, langage utilisé dans les travaux pratiques.

1.2 Description des algorithmes en pseudo-code

Un *algorithme* [1] est la description d'un *calcul* effectué sur des *données discrètes*. On distingue les données fournies en *entrée* de l'algorithme et celles qu'il calcule en *sortie*.

Un algorithme est décrit par un *pseudo-code* suffisamment précis pour pouvoir être implémenté dans tout langage de programmation, et en C en particulier.

Conventions retenues pour le pseudo-code :

1. utilisation d'*indentations* pour identifier les blocs d'instructions ;
2. les tests usuels ($=, <, >, \leq, \geq$) sont disponibles ainsi que les connecteurs logiques de base (`ET, OU, NON`) [2] ;
3. les structures de contrôles conditionnelles usuelles (`si ...alors ...` et `si ...alors ...sinon ...` sont disponibles ;
4. les structures itératives usuelles `tant que, pour, répéter ...jusqu'à`) sont disponibles ;
5. l'`affectation` de variable est notée $\leftarrow$;
6. il est possible de définir et d'utiliser des `fonctions` et `procédures` ; les passages de paramètres se font `par valeur` ; les appels récursifs sont possibles.

1.3 Premiers exemples d'algorithmes

L'algorithme 1 décrit une méthode simple de tri, le *tri par insertion*, qui consiste à parcourir un tableau du second indice jusqu'au dernier et à insérer de façon ordonnée chaque élément courant parmi les éléments précédents. L'*insertion ordonnée* d'un élément dans un tableau trié est réalisée par l'algorithme 2. La division de l'algorithme de tri en deux rend sa compréhension, et donc sa vérification, plus simple. Remarquez que dans la condition d'arrêt de la boucle de l'algorithme 2, $T[i]$ n'est pas évalué si $i = 0$ puisque dans ce cas, la conjonction est fausse.

1. d'après le nom du mathématicien perse Al Khawarizmi (783-850).
2. on supposera que comme en C, l'évaluation de `p ET q` (resp. `p OU q`) commence par `p` et s'arrête si `p` est évalué à `FAUX` (resp. à `VRAI`).

Algorithme 1: TriInsertion

entrée : $T[1, n]$ est un tableau d'entiers, $n \geq 1$.
résultat : les éléments de T sont ordonnés par ordre croissant.

```
début
  pour j = 2 à n faire
    InsertionOrdonnée(T[j],T[1, j-1])
  finpour
fin
```

Algorithme 2: InsertionOrdonnée

entrée : x est un entier ; $T[1, n]$ est un tableau d'entiers trié par ordre croissant.
sortie : $T[1, n+1]$ contient les éléments de $T \cup \{x\}$ ordonnés par ordre croissant.

```
début
  i ← n
  tant que i > 0 ET T[i] > x faire
    T[i + 1] ← T[i]
    i ← i − 1
  fintq
  T[i + 1] ← x
fin
```

L'algorithme 3 décrit la recherche d'un élément x dans un tableau trié $T[m, n]$: il commence par comparer x à l'élément qui se trouve au centre du tableau puis, selon les cas, arrête la recherche ou la continue sur l'une des deux moitié de tableaux. On parle de recherche *dichotomique*. Cet algorithme suit la méthode *divide and conquer*, qui consiste à diviser le problème initial en sous problèmes de tailles inférieures, à résoudre ces sous-problèmes puis à combiner leurs solutions pour trouver la solution du problème initial. C'est un algorithme *récursif* puisqu'il comporte des appels à lui-même.

```
Fonction rechercheDichotomique(T, m, n, x)
  entrée : T[m, n] est un tableau d'entiers trié par ordre croissant,
           1 ≤ m ≤ n ; x est un entier.
  sortie : 0 si x ∉ T et i si T[i] est la première occurrence de x dans
           T[m, n].
  début
    si m = n alors
      si T[m] = x alors
        retourner m
      sinon
        retourner 0
      finsi
    sinon
      k ← ⌊(m+n)/2⌋ [a]
      si T[k] < x alors
        retourner rechercheDichotomique(T, k + 1, n, x)
      sinon
        retourner rechercheDichotomique(T, m, k, x)
      finsi
    finsi
  fin
```

a. Pour tout réel x, $\lceil x \rceil$ (resp. $\lfloor x \rfloor$) désigne la *partie entière supérieure* (resp. la *partie entière inférieure*) de x, c'est-à-dire le plus petit entier supérieur ou égal à x (resp. le plus grand entier inférieur ou égal à x).

Chapitre 2

Analyse des algorithmes

Analyser un algorithme consiste à calculer les ressources qui seront nécessaires à son exécution. Mais avant de l'analyser, il faut d'abord s'assurer qu'il est *correct*, c'est-à-dire qu'il calcule bien ce pourquoi il a été écrit.

2.1 Preuves de la correction d'un algorithme

Comment s'assurer qu'un algorithme calcule bien ce qu'il est censé calculer ? Comment s'assurer qu'un algorithme termine quelle que soit les données qui lui seront fournies en entrée ? Les algorithmes sont suffisamment formalisés pour qu'on puisse *prouver* qu'ils possèdent les propriétés attendues de *correction* ou de *terminaison*. Les preuves sont facilitées lorsque les algorithmes sont bien écrits, et en particulier s'ils sont décomposés en de nombreux sous algorithmes courts et bien spécifiés.

Ce sont bien évidemment les structures itératives et les appels récursifs qui nécessitent le plus de travail.

2.1.1 Preuve d'une structure itérative

Considérons le schéma d'algorithme suivant :

Algorithme 3: Structure itérative générique

```
résultat : R
début
    Initialisation
    tant que Condition faire
        Instructions
    fintq
fin
```

Un *invariant de boucle* est une propriété ou une formule logique,
- qui est vérifiée après la phase d'initialisation,
- qui reste vraie après l'exécution d'une itération,
- et qui, conjointement à la condition d'arrêt, permet de montrer que le résultat attendu est bien celui qui est calculé.

Exemple 1 Pour l'algorithme 2, qui insère un élément x dans un tableau trié $T[1, n]$, considérons la propriété I suivante :

> ≪ *la juxtaposition des tableaux* $T[1, i]$ *et* $T[i+2, n+1]$ [1] *est égale au tableau initial et* x *est plus petit que tous les éléments du deuxième tableau* [2] ≫.

1. La propriété est vérifiée après l'initialisation puisque $T[1, i]$ est égal au tableau initial ($i = n$), et que le second tableau $T[i+2, n+1]$ est vide.
2. Si la propriété I est vraie au début d'une itération, elle est vraie à la fin de cette itération puisque le dernier élément $T[i]$ du premier tableau passe en tête du second (l'ordre n'est pas modifié), que l'on a $x < T[i]$ et que l'indice est modifié en conséquence.

   Si la condition reste vraie, alors la propriété est donc vérifiée au début de l'itération suivante.
3. Si la condition est fausse, alors
   - soit $i = 0$, le premier tableau est vide, x est donc plus petit que tous les éléments du tableau initial, reportés dans le tableau $T[2, n+1]$ et il suffit d'écrire $T[1] = T[i+1] = x$ pour obtenir le résultat attendu ;
   - soit $i > 0$ et $x \geq T[i]$: x est donc $\geq$ que tous les éléments du tableau $T[1, i]$ et $<$ que tous les éléments du tableau $T[i + 2, n]$. Il suffit d'écrire $T[i + 1] = x$ pour obtenir le résultat attendu.

Pour prouver la correction d'une boucle Pour, on peut remarquer que

```
Pour i=1 à n
    Instructions
```

1. si $i + 2 > n + 1$, $T[i + 2, n + 1]$ désigne un tableau vide.
2. propriété vraie si le deuxième tableau est vide.

équivaut à

```
i=1
Tant que i<= n
   Instructions
   i=i+1
```

et utiliser la technique précédente.

Exemple 2 Pour prouver la correction de l'algorithme 1 de tri par insertion, on considère l'invariant de boucle suivant :

≪ $T[1, j-1]$ *est trié* ≫.

1. Après l'initialisation, la propriété est vraie puisque $j = 2$ et que le tableau $T[1, j-1]$ ne contient qu'un seul élément.
2. Supposons que $T[1, j-1]$ soit trié au début d'une itération. Après appel de l'algorithme insertionOrdonnee, le tableau $T[1, j]$ est trié. Après l'incrémentation de j, le tableau $T[1, j-1]$ est trié.
3. Si la condition devient fausse, c'est que $j = n + 1$. Le tableau $T[1, n]$ est donc trié.

2.1.2 Preuve d'un algorithme récursif

On prouve la correction d'un algorithme récursif au moyen d'un raisonnement par récurrence.

Exemple 3 Pour prouver la correction de l'algorithme 3 (recherche dichotomique), on effectue une récurrence sur le nombre $N = n-m+1$ d'éléments du tableau.

- si $N = 1$, alors $m = n$ et on vérifie que l'algorithme est correct dans ce cas ;
- soit $N \geq 1$. Supposons que le programme soit correct pour tout tableau de longueur $\leq N$ et considérons un tableau $T[m, n]$ de $N+1$ éléments : $n - m + 1 = N + 1$. En particulier, $m \neq n$ puisque $N \geq 1$. Soit $k = \lfloor \frac{m+n}{2} \rfloor$. On a $m \leq k < n$. Donc, les longueurs des deux tableaux passés en argument des appels récursifs vérifient : $k - m + 1 \leq n - m \leq N$ et $n - k \leq n - m \leq N$. Par hypothèse de récurrence, quelque soit le résultat du test $T[k] < x$, l'algorithme donnera une réponse correcte.

2.1.3 Exercices

Prouvez la correction des algorithmes suivants :

1. Recherche du plus grand élément d'un tableau (version itérative) :

```
Fonction MaxEltIter(T)
  entrée : T[1,n] est un tableau d'entiers.
  sortie : le plus grand élément de T[1,n]
  début
  | Max ← T[1]
  | pour i = 2 à n faire
  | | si T[i] > Max alors
  | | | Max ← T[i]
  | | finsi
  | finpour
  | retourner Max
  fin
```

2. Recherche du plus grand élément d'un tableau (version récursive) :

```
Algorithme 4: MaxEltRec
  entrée : T[1,n] est un tableau d'entiers.
  sortie : le plus grand élément de T[1,n]
  début
  | si n = 1 alors
  | | retourner T[1]
  | sinon
  | | retourner Max(T[n], MaxEltRec(T[1, n − 1]))
  | finsi
  fin
```

3. Recherche conjointe du plus petit élément et du plus grand élément d'un tableau.

Algorithme 5: MinMaxElt

entrée : $T[1, n]$ est un tableau d'entiers.
sortie : le plus petit et le plus grand élément de $T[1, n]$
début
  $Min \leftarrow T[1], Max \leftarrow T[1]$
  pour $j = 1$ *à* $\lfloor \frac{n-1}{2} \rfloor$ **faire**
    si $T[2 * j] < T[2 * j + 1]$ **alors**
      si $T[2 * j] < Min$ **alors**
        $Min \leftarrow T[2 * j]$
      finsi
      si $T[2 * j + 1] > Max$ **alors**
        $Max \leftarrow T[2 * j + 1]$
      finsi
    sinon
      si $T[2 * j + 1] < Min$ **alors**
        $Min \leftarrow T[2 * j + 1]$
      finsi
      si $T[2 * j] > Max$ **alors**
        $Max \leftarrow T[2 * j]$
      finsi
    finsi
  finpour
  si n *est pair* **alors**
    si $T[n] < Min$ **alors**
      $Min \leftarrow T[n]$
    finsi
    si $T[n] > Max$ **alors**
      $Max \leftarrow T[n]$
    finsi
  finsi
  retourner Min, Max
fin

2.2 Complexité des algorithmes

En plus d'être correct, c'est-à-dire d'effectuer le calcul attendu, on demande à un algorithme d'être efficace, l'efficacité étant mesurée par le temps que prend l'exécution de l'algorithme ou plus généralement, par les quantités de ressources nécessaires à son exécution (espace mémoire, bande passante,

...).

Le temps d'exécution d'un algorithme dépend évidemment de la taille des données fournies en entrée, ne serait-ce que parce qu'il faut les lire avant de les traiter. La taille d'une donnée est mesurée par le nombre de bits nécessaires à sa représentation en machine. Cette mesure ne dépend pas du matériel sur lequel l'algorithme sera programmé.

Mais comment mesurer le temps d'exécution d'un algorithme puisque les temps de calcul de deux implémentations du **même** algorithme sur deux machines différentes peuvent être très différents ? En fait, quelque soit la machine sur laquelle un algorithme est implémenté, le temps de calcul du programme correspondant est égal au nombre d'opérations élémentaires effectuées par l'algorithme, **à un facteur multiplicatif près**, et ce, quelle que soit la taille des données d'entrée de l'algorithme. Par opérations élémentaires on entend évaluations d'expressions, affectations, et plus généralement traitements en temps constant ou indépendant de l'algorithme (entrées-sorties,...). Le détail de l'exécution de ces opérations au niveau du processeur par l'intermédiaire d'instructions machine donnerait des résultats identiques à un facteur constant près.

Autrement dit, pour toute machine $\mathcal{M}$, il existe des constantes m et M telles que pour tout algorithme $\mathcal{A}$ et pour toute donnée d fournie en entrée de l'algorithme, si l'on note $T(d)$ le nombre d'opérations élémentaires effectuées par l'algorithme avec la donnée d en entrée et $T_{\mathcal{M}}(d)$ le temps de calcul du programme implémentant $\mathcal{A}$ avec la donnée d en entrée,

$$mT(d) \leq T_{\mathcal{M}}(d) \leq MT(d).$$

Soit, en utilisant les notations de Landau (voir section 8.1),

$$T = \Theta(T_{\mathcal{M}}).$$

On peut donc définir la *complexité en temps* d'un algorithme comme le nombre d'opérations élémentaires $T(n)$ effectuées lors de son exécution sur des données de taille n. Il s'agit donc d'une fonction. On distingue

- la complexité *dans le pire des cas* (*worst case complexity*) : $T_{pire}(n) =$ le nombre d'opérations maximal calculé sur toutes les données de taille n ;
- la complexité *dans le meilleur des cas* (*best case complexity*) : $T_{min}(n) =$ le nombre d'opérations minimal calculé sur toutes les données de taille n ;
- la complexité *en moyenne* (*average case complexity*) : $T_{moy}(n) =$ le nombre d'opérations moyen calculé sur toutes les données de taille n, en supposant qu'elles sont toutes équiprobables.

Analyser un algorithme consiste à évaluer la ou les fonctions de complexité qui lui sont associées. En pratique, on cherche à évaluer le *comportement asymptotique* de ces fonctions relativement à la taille des entrées (voir section 8.1). En considérant le temps d'exécution, dans le pire des cas ou en moyenne, on parle ainsi d'algorithmes

- en *temps constant* : $T(n) = \Theta(1)$,
- *logarithmiques* : $T(n) = \Theta(\log n)$,
- *linéaires* : $T(n) = \Theta(n)$,
- *quadratiques* : $T(n) = \Theta(n^2)$,
- *polynomiaux* : $T(n) = \Theta(n^k)$ pour $k \in \mathbb{N}$,
- *exponentiels* : $T(n) = \Theta(k^n)$ pour $k > 0$.

Si un traitement élémentaire prends un millionième de seconde, le tableau suivant décrit les temps d'exécutions approximatifs de quelques classes de problèmes en fonction de leurs tailles

Taille n	$log_2\, n$	n	$n\, log_2\, n$	n^2	2^n
10	$0.003\, ms$	$0.01\, ms$	$0.03\, ms$	$0.1\, ms$	$1\, ms$
100	$0.006\, ms$	$0.1\, ms$	$0.6\, ms$	$10\, ms$	$10^{14}\, siecles$
1000	$0.01\, ms$	$1\, ms$	$10\, ms$	$1\, s$	
10^4	$0.013\, ms$	$10\, ms$	$0.1\, s$	$100\, s$	
10^5	$0.016\, ms$	$100\, ms$	$1.6\, s$	$3\, heures$	
10^6	$0.02\, ms$	$1\, s$	$20\, s$	$10\, jours$	

D'un autre point de vue, le tableau ci-dessous donne la taille des problèmes que l'on peut traiter en une seconde en fonction de la rapidité de la machine utilisée (nTs est le nombre de traitements par secondes).

nTs	2^n	n^2	$n\, log_2\, n$	n	$log_2\, n$
10^6	20	1000	63000	10^6	10^{300000}
10^7	23	3162	600000	10^7	$10^{3000000}$
10^9	30	31000	4.10^7	10^9	
10^{12}	40	10^6	3.10^{10}		

En pratique, un algorithme est constitué d'instructions agencées à l'aide de structures de contrôle que sont les *séquence*, les *embranchements* et les *boucles* (nous verrons plus tard comment traiter le cas particulier des fonctions récursives, mais de toutes façons toute fonction récursive s'écrit aussi itérativement). Le coût d'une séquence de traitements est la somme des

coûts de chaque traitement. Le coût d'une structure itérative est la somme des coûts des traitements effectués lors des passages successifs dans la boucle. Par exemple il arrive fréquemment que l'on effectue n fois une boucle avec des coûts dégressifs ($\Theta(n)$, $\Theta(n-1)$,...,$\Theta(1)$). Dans ce cas

$$T(n) \; = \; n + (n-1) + (n-2) + \ldots + 1 \; = \; \sum_{i=1}^{n} i \; = \; \frac{n \times (n+1)}{2}$$

Le coût d'un embranchement (`si test alors traitement1 sinon traitement2`) est le coût du plus couteux des deux traitements.

2.2.1 Exemples d'analyse d'algorithmes

L'algorithme d'insertion d'un élément dans un tableau trié Dans le pire des cas, celui où l'entier à insérer est plus petit que tous les éléments du tableau, l'algorithme 2 requiert

- $2N + 2$ affectations,
- $2N$ comparaisons d'entiers,
- N soustractions et 1 addition,
- N évaluations d'une expression booléenne

pour un tableau de N éléments.

En supposant que les entiers ont une taille constante k, une donnée en entrée de taille n permet de représenter n/k entiers, soit un tableau de $N = \lfloor n/k - 1 \rfloor$ entiers. On remarque que $N = \Theta(n)$. En supposant de plus que chaque instruction élémentaire s'exécute en un temps constant, on en déduit que $T_{pire}(n) = \Theta(n)$. L'algorithme d'insertion ordonnée est linéaire dans le pire des cas : à des constantes additive et multiplicative près, le temps d'exécution est égal au nombre d'éléments du tableau.

On voit facilement que $T_{min}(n) = \Theta(1)$: en effet, dans le meilleur des cas, l'élement à insérer est plus grand que tous les éléments du tableau et la boucle n'est pas exécutée.

Si l'on suppose que l'élément à insérer et un élément pris au hasard dans le tableau suivent la même loi de probabilité, son insertion nécessitera en moyenne N comparaisons. On aura donc, $T_{moy}(n) = \Theta(n)$, soit une complexité moyenne du même ordre que la complexité du pire des cas.

L'algorithme de tri par insertion Dans le pire des cas, celui où le tableau en entrée est déjà trié, mais par ordre décroissant, l'algorithme nécessite

- $N-1$ comparaisons et affectations pour la condition de la boucle pour
- $N-1$ appels à l'algorithme d'insertion ordonnée, pour des tailles de tableaux variant de 1 à $N-1$.

A des constantes additive et multiplicative près, le temps d'exécution dans le pire des cas de l'algorithme de tri par insertion est donc égal à

$$(N-1)+1+2+\ldots+(N-1) = N-1+\frac{N(N-1)}{2} = \Theta(N^2)$$

soit en appliquant la remarque sur l'égalité à des constantes additive et multiplicative près de la taille des données et du nombre d'éléments du tableau,

$$T_{pire}(n) = \Theta(n^2).$$

On voit facilement que $T_{min}(n) = \Theta(n)$, cas où le tableau est déjà ordonné par ordre croissant.

Si tous les éléments du tableau en entrée sont tirés selon la même loi de probabilités, on peut montrer que $T_{moy}(n) = \Theta(n^2)$. Autrement dit, s'il peut arriver que le temps d'exécution soit linéaire, il y a beaucoup plus de chance qu'il soit quadratique en la taille des données en entrée.

L'algorithme de recherche dichotomique Si le tableau contient un seul élément, l'algorithme exécutera 2 comparaisons (coût total : c_1)

Si le tableau contient $N > 1$ éléments, l'algorithme exécutera

- 3 opérations arithmétiques, une affectation et une comparaison entre 2 entiers (coût total : c_2), et
- un appel récursif sur un tableau possédant $\lceil N/2 \rceil$ ou $\lfloor N/2 \rfloor$ éléments, soit $\lceil N/2 \rceil$ dans le pire des cas.

On en déduit que

$$T_{pire}(N) = T_{pire}(\lceil N/2 \rceil) + c_2.$$

Conformément à la remarque faite ci-dessus sur l'équivalence entre la taille d'un tableau d'entiers et sa longueur, on supposera que N est égal à la taille des données.

Cette équation est simple à résoudre lorsque N est égal à une puissance de 2 :

$$T_{pire}(2^k) = T_{pire}(2^{k-1}) + c_2 = T_{pire}(2^{k-2}) + 2c_2 = \ldots = c_1 + kc_2.$$

Dans le cas général, soit

$$N = a_k 2^k + \ldots + a_1 2^1 + a_0$$

l'écriture de N en base 2 : $a_k = 1$ et $0 \leq a_i \leq 1$ pour $0 \leq i < k$. On a

$$2^k \leq N < 2^{k+1} \text{ et donc } 2^{k-1} \leq \lceil N/2 \rceil \leq 2^k.$$

On en déduit que

$$c_1 + kc_2 \leq T_{pire}(N) \leq c_1 + (k+1)c_2.$$

En remarquant que $k = \Theta(\log_2 N)$, on en déduit que $T_{pire}(N) = \Theta(\log_2 N)$. La recherche dichotomique d'un élément dans un tableau trié se fait donc en temps logarithmique.

L'analyse d'algorithmes conduit souvent à des équations récursives de la forme

$$T(n) = a \times T(n/b) + f(n).$$

Le théorème 1 de la section 8.2 donne la solution de ces équations dans le cas général. Dans l'exemple précédent, on se trouve dans le cas (2) du théorème, avec $a = 1$ et $b = 2$: on retrouve que $T_{pire}(N) = \Theta(\log_2 N)$.

Le tri par fusion

Algorithme 6: TriFusion

entrée : $T[1, n]$ est un tableau d'entiers, $n \geq 1$.
résultat : les éléments de T sont ordonnés par ordre croissant.
début
  si $n > 1$ **alors**
    $T_1 = triFusion(T[1, \lfloor n/2 \rfloor])$
    $T_2 = triFusion(T[\lfloor n/2 \rfloor + 1, n])$
    $T = Fusion(T_1, T_2)$
  finsi
fin

La récurrence est

$$T(n) = 1 + n + 2 \times T(n/2) + n$$

D'autre part $T(0) = T(1) = 1$.

Solution méthode générale (voir le théorème 1 de la section 8.2). On est dans le cas 2 puisque $f(n) = 2 \times n + 1$ et puisque $a = b = 2$ on a $\log_b a = 1$ et $f(n) = \Theta(n)$. Donc $T(n) = \Theta(n \times \lg n)$.

Solution en développant la récurrence.

$$T(n) = 2n + 1 + 2 \times T(n/2) = 2n + 1 + 2n + 2 + 4 \times T(n/4)$$
$$= 2n + 1 + 2n + 2 + 2n + 4 + 8 \times T(n/8) = \ldots$$

On en déduit que puisque $T(1) = 1$, on a une suite de $\lceil log_2 n \rceil$ termes puisqu'on divise n par 2 à chaque fois, et donc

$$T(n) = \Sigma_{i=0}^{log_2 n}(2n + 2^i) = 2n \times log_2 n + \Sigma_{i=0}^{log_2 n} 2^i$$

que l'on sait calculer [3]. Donc

$$T(n) = 2nlog_2 n + 2^{log_2 n+1} - 1 = 2nlog_2 n + 2n = 2n(log_2 n + 1) = O(nlog_2 n)$$

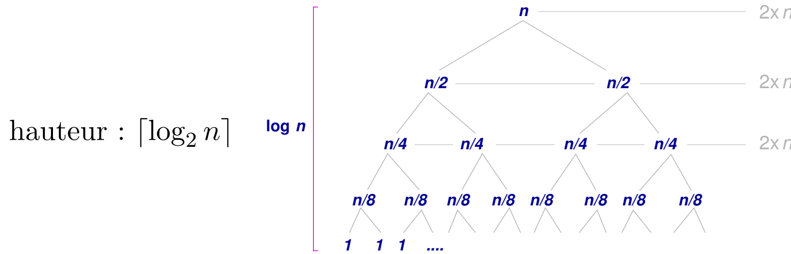


FIGURE 2.1 – Arbre des appels récursifs pour l'algorithme de tri par fusion.

Solution intuitive. On remarque que l'exécution de la procédure produit le parcours d'un arbre virtuel de profondeur $\lceil log_2 n \rceil$, et qu'à chaque niveau de l'arbre on effectue $\Theta(n)$ traitements (2^p noeuds à chaque niveau, et des suites de longueurs $n/(2^p)$ éléments en chaque noeud). On pressent donc intuitivement que le coût est quelque chose de la forme $nlog_2 n$. Mais attention il peut y avoir des termes de classe inférieure. On va donc partir sur une forme complète de $T(n)$ avec des constantes

$$T(n) = a \times n \times log_2 n + b \times n + c$$

Nous essaierons ensuite de fixer les constantes, d'une part en développant l'équation récursive en substituant à $T(n/2)$ la formule ci-dessus, mais aussi en utilisant les cas pour lesquels on connait le résultat, c'est-à-dire quand n est très petit.

3. Quelques formules de sommation : $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} = O(n^2)$; $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1} = O(x^n)$ si $x \neq 1$.

Donc $T(n/2) = a \times n/2 \times log_2 n - a \times n/2 + b \times n/2 + c$. En reprenant la récurrence on obtient

$$T(n) = (2c + 1) + (b + 2 - a) \times n + a \times n \times log_2 n$$

On en déduit (1) $2c + 1 = c$ donc $c = -1$, (2) $b + 2 - 1 = b$ donc $a = 2$. Puisque $T(1)$ vaut 1, on en déduit que $2 \times n \times log_2 1 + b - 1 = 1$ et donc $b = 2$. Donc

$$T(n) = 2 \times n \times log_2 n + 2 \times n - 1 = O(n \times log_2 n)$$

2.2.2 Exercices

Exercice 1 Quelle est la classe de complexité de la fonction $f(n)$ suivante (démontrez-le)

$$f(n) = 2 \times n^3 + 4 \times n^2 + 2^3$$

Exercice 2 Quelle est la complexité de la fonction `bidon` suivante :

> **fonction** bidon(n)
> **début**
>   **pour** $i := 1$ **à** $n * n$ **faire**
> $u := i$,
>   **tant que** $(u > 1)$ **faire**
> $u := u/2$,
>   **fin faire**
>   **fin faire**
> **fin fonction**

Exercice 3

1. Écrivez un algorithme qui prend en entrée un tableau d'entiers et calcule les valeurs minimale et maximale qu'il contient. Évaluez la complexité de votre algorithme (mesurée en nombre de comparaisons effectuées).
2. Évaluez la complexité de l'algorithme 7 (même mesure).

Exercice 4 Calculez la complexité de la fonction `ProdMat(MAT A, MAT B, MAT C, int n)` qui calcule le produit des matrices carrées `A` et `B` de `n` lignes et `n` colonnes.

Exercice 5 Calculer la complexité de la fonction $exp(x, n)$ qui calcule x^n à partir de la formule suivante :

$x^n = (x^2)^{n/2}$ si n est pair,
$x^n = x \times x^{n-1}$ si n est impair

En utilisant ce principe, quel est le coût de l'élévation d'une matrice à la puissance n.

Exercice 6 : Calcul de l'élément majoritaire d'un tableau.

Etant donné un ensemble E de n éléments, on veut savoir s'il existe un élément majoritaire et quel est-il (un élément majoritaire apparaît plus d'une fois sur deux, i.e. si k est son nombre d'apparitions, $2 * k > n$).

Méthode naïve Donnez une méthode naive pour ce calcul. Quelle est sa complexité ?

Méthode 2 : diviser pour régner. On considère maintenant l'approche *diviser pour régner* suivante : pour calculer l'élément majoritaire dans l'ensemble E s'il existe, on réparti les n éléments de E dans deux ensembles de mêmes tailles E_1 et E_2, et on calcule (récursivement) dans chacune de ces parties l'élément majoritaire. Pour que e soit majoritaire dans E il suffit que e soit majoritaire dans E_1 et dans E_2, ou que e soit majoritaire dans E_1 et non dans E_2 (ou inversement), mais qu'il apparaisse suffisamment dans E_2.

Ecrivez une procédure de calcul correspondant à cette idée. Quelle est sa complexité ?

Une troisième méthode sera proposée au chapitre suivant.

Chapitre 3

Structures linéaires

Les données que doivent traiter les programmes informatiques sont souvent liées par des relations qui leur confèrent une certaine structure : dans certains jeux de cartes, on peut être obligé de déposer des cartes au sommet d'un tas et n'avoir accès qu'à la dernière carte déposée : on parle de structure de *pile* ; les processus envoyés à une imprimante doivent être gérés de manière que le premier processus envoyé soit le premier traité : on parle de *file d'attente* ; les positions sur un échiquier peuvent être représentés par les noeuds d'un *arbre* dont les successeurs sont toutes les positions atteignables en un coup ; les noeuds d'un réseau de communication peuvent être représentés par les sommets d'un*graphe*, ...

On a répertorié un certain nombre de *structures de données* qui reviennent dans la très grande majorité des traitements informatique. Les structures de données en informatique jouent un peu le rôle des structures abstraites en mathématiques, groupes, anneaux, corps, espaces vectoriels, ... : elles sont suffisamment générales pour qu'il y ait avantage à les étudier indépendamment de toute application spécifique. On peut par exemple étudier la question du tri d'un tableau ou de la recherche d'un chemin dans un graphe sur des structures abstraites : les algorithmes qui résolvent ces problèmes dans le cas général pourront alors être utilisés dans n'importe quelle situation spécifique.

Les langages de programmation proposent généralement des types simples, quelques types plus complexes (tableaux, listes, etc) et des contructeurs de types permettant d'implémenter toutes les structures de données.

On parle de *structure linéaire* lorsque les données sont représentées séquentiellement : chaque élément, sauf le dernier, possède un successeur. On étudiera en particulier les *tableaux* à une dimension, les *listes*, les *piles*, les

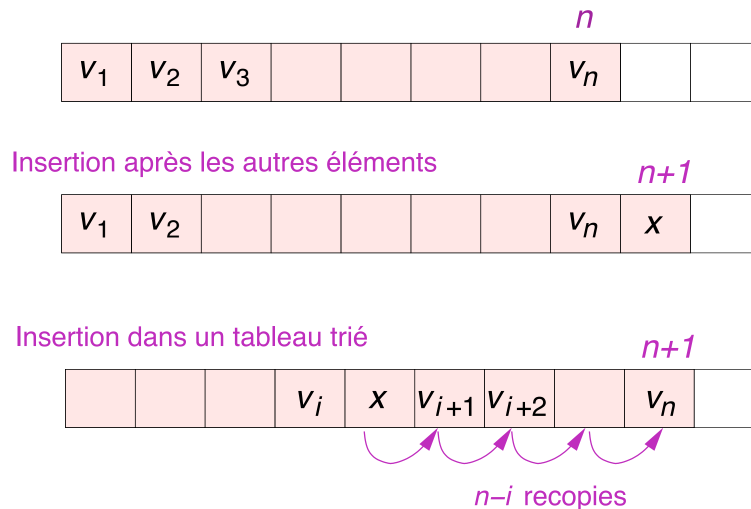


FIGURE 3.1 – Insertion d'un élément en fin de tableau ou à un indice donné.

files et les *tables de hachage*. Les *matrices* carrées, les *arbres* et les *graphes* sont des structures non linéaires.

3.1 Tableaux

Un *tableau* (*array*) T est un ensemble d'éléments accessibles par un indice $i \in [1 \ldots n]$. On suppose que le calcul du nombre d'éléments d'un tableau, `longueur`$(T) = n$, l'accès au i-ème élément $T[i]$ et l'ajout d'un élément en fin de tableau peuvent être réalisés en temps constant $(O(1))$. En revanche, l'insertion d'un nouvel élément à un indice i donné nécessite de déplacer tous les éléments suivants (temps linéaire dans le pire des cas). De même la recherche d'un élément dans un tableau non trié se fait en temps linéaire dans le pire des cas. Nous avons vu précédemment que la recherche d'un élément dans un tableau trié pouvait être effectuée en temps logarithmique.

Implantation des tableaux en C

```
/* Définition */
#define NBMAX 1000 /* nombre maximum d'éléments */
typedef int ELEMENT; /* les éléments du tableau */
typedef ELEMENT TABLEAU[NBMAX];
```

```
/* Déclaration */
TABLEAU t;
```

ou avec allocation dynamique de la mémoire,

```
/* Définition */
typedef ELEMENT* TABLEAU;

/* Déclaration */
TABLEAU t;
int nb_elts=50; /* nombre d'éléments du tableau */

/* Allocation */
t=malloc(nb_elts*sizeof(ELEMENT));
```

3.2 Listes chaînées

Une *liste chaînée* (*linked list*) est une structure linéaire, composée de *maillons*, chaque maillon comprenant un champ *valeur* qui permet de stocker un élément et un champ *suivant* qui pointe vers le maillon suivant ou est égal à `NIL`, s'il s'agit du dernier maillon de la liste. Une *liste chainée* `L` est un objet qui est égal à `NIL` si la liste est vide ou qui pointe vers un maillon, le premier de la liste, si elle est non vide.

Implantation des listes chaînées en C

```
/* Définition */

typedef int ELEMENT; /* par exemple */

typedef struct maillon *LISTE;

typedef struct maillon{
           ELEMENT val;
           LISTE suiv;
}MAILLON;


/* Déclaration */
```

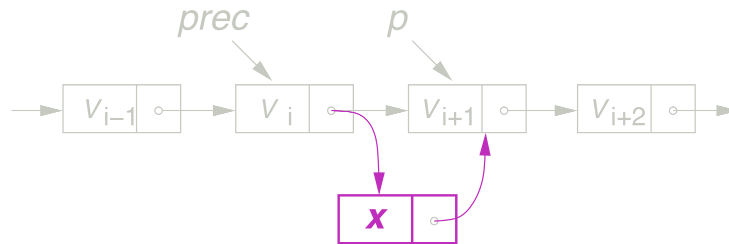


FIGURE 3.2 – Insertion d'un élément dans une liste à une place donnée.

```
LISTE l;

/* Insertion d'un élément en tête d'une liste */

LISTE CreerMaillon(ELEMENT elt, LISTE liste_initiale){
   LISTE l;
   l=malloc(sizeof(MAILLON));
   l->val=elt;
   l->suiv=liste_initiale;
   return l;
}
```

On voit que l'insertion d'un élément en tête de liste se fait en temps constant.

Le traitement des listes chaînées requiert de pouvoir :

- insérer un élément en queue de liste,
- rechercher si un élément est présent dans la liste,
- supprimer la première occurrence d'un élément,
- insérer un élément dans une liste ordonnée,
- trier une liste,
- vider une liste de ses éléments.

Exercice 1

1. Quelle est la complexité dans le pire des cas des algorithmes précédents ?
2. Programmez-les en C.

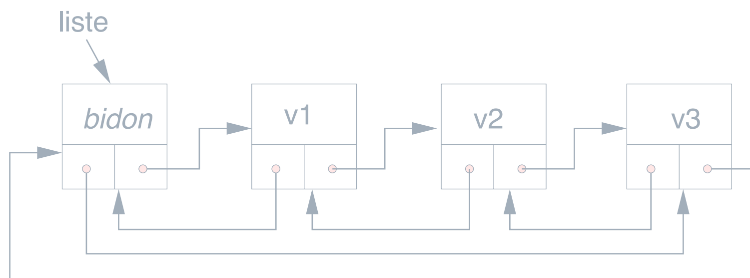


FIGURE 3.3 – Liste doublement chaînée circulaire avec maillon vide.

Améliorations de la structure de liste chaînée Il n'est pas possible d'accéder directement au maillon précédent le maillon courant d'une liste chaînée. Cela complique notamment l'écriture de l'algorithme de suppression d'un maillon. Pour remédier à ce défaut, on considère des listes *doublement chaînées* dans lesquels les maillons comprennent également un champ `précédent` qui pointe vers le maillon précédent ou est égal à NIL, s'il s'agit du premier maillon de la liste.

Par ailleurs, les algorithmes de traitement des listes chaînées sont alourdis par le fait qu'il faut traiter différemment le premier maillon, qui n'a pas de maillon précédent, des maillons suivants. Pour remédier à cela, on introduit un *maillon vide*, dont le champ *val* prend une valeur quelconque, et qui constituera le premier maillon de la liste chaînée.

Mais le même problème se pose alors pour le dernier maillon, qui n'admet pas de maillon suivant. On suppose alors que le champ *suivant* du dernier maillon pointe circulairement sur le maillon vide et que le champ *précédent* du maillon vide pointe vers le dernier maillon.

On obtient ainsi la notion de *liste doublement chaînée circulaire avec maillon vide* (*circular, doubly linked list, with a sentinel*).

Le tableau 3.5 compare la complexité de quelques opérations élémentaires sur des tableaux et des listes chaînées.

Exercice 2 Écrire les algorithmes de traitement des listes doublement chaînées circulaires avec maillon vide.

Exercice 3 On représente un polynôme à coefficients entiers par une liste chainée dont les maillons possèdent un champ `coefficient` et un champ `exposant` ; on suppose de plus que les listes sont ordonnées par valeurs croissantes d'exposant. Implantez cette structure de données en C et programmez les fonctions suivantes :

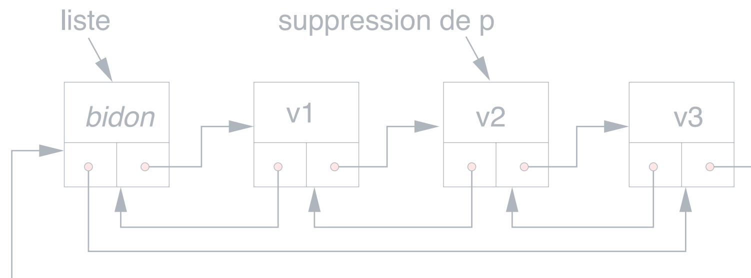


FIGURE 3.4 – Suppression d'un maillon dans une liste doublement chaînée circulaire avec maillon vide.

Opération	Tableau	Tableau trié	Liste	Liste ordonnée	Liste doublement chaînée	Liste ordonnée doublement chaînée
accès i-ème élément	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
insertion à une place donnée	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
recherche élément	$\Theta(n)$	$\Theta(\log_2 n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
recherche succ.	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
recherche pred.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
recherche maximum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
recherche minimum	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

FIGURE 3.5 – Comparatif tableaux / listes chaînées.

- `int degre(POLYNOME p)`
- `void affiche(POLYNOME p)`
- `POLYNOME somme(POLYNOME p1, POLYNOME p2)`
- `POLYNOME multiplication_scalaire(POLYNOME p, int a)`
- `POLYNOME multiplication(POLYNOME p1, POLYNOME p2)`
- `POLYNOME derivee(POLYNOME p)`
- `float eval(POLYNOME p, float x)`

3.3 Piles et files

Les *piles* (*stack*) et les *files* (*queues*) sont des structures linéaires qui diffèrent par la manière dont les éléments sont insérés et supprimés : dans une pile, le dernier élément entré est le premier sorti (*Last In First Out : LIFO*) ; dans une file, le premier élément entré est le premier sorti (*First In First Out : FIFO*).

Trois fonctions (ou opérations) suffisent à faire fonctionner une pile :
- la fonction booléenne *pileVide ?(P)* (*StackEmpty(P)*) qui retourne VRAI si la pile P est vide et FAUX sinon,
- la fonction *empiler(P,x)* (*Push(P,x)*) qui insère un élément x dans la pile P,
- la fonction *dépiler(P)* (*Pop(P)*) qui retourne le dernier élément inséré ou un message d'erreur si la pile est vide.

De même, trois fonctions (ou opérations) suffisent à faire fonctionner une file :
- la fonction booléenne *fileVide ?(F)* (*QueueEmpty(F)*) qui retourne VRAI si la file F est vide et FAUX sinon,
- la fonction *enfiler(F,x)* (*EnQueue(F,x)*) qui insère un élément x dans la file F,
- la fonction *défiler(F)* (*DeQueue(F)*) qui retourne le premier élément inséré ou un message d'erreur si la file est vide.

Toute implémentation d'une pile ou d'une file nécessite l'implémentation des fonctions correspondantes.

Implémentation d'une pile par un tableau. Une pile peut être représentée par les premiers éléments d'un tableau de taille fixe et par un entier indiquant l'indice du dernier élément inséré. La dimension du tableau étant fixée, il est nécessaire d'implanter aussi une fonction booléenne *PilePleine ?* qui

indiquera si la pile est pleine ou non. En C, cela peut se programmer de la façon suivante :

```
/* Déclarations */
#define TAILLE_MAX 1000 // Nombre maximal d'éléments dans la pile

typedef int ELEMENT; /* pour une pile d'entiers */

typedef struct {
  ELEMENT elts[TAILLE_MAX];
  int nbElts;} // nombre d'éléments de la pile
  PILE;

PILE p;

/* Initialisation */
p.nbElts=0;

/* retourne 1 si la pile p est vide, 0 sinon */
char pileVide(PILE p){
  return (p.nbElts==0);
}

/* retourne 1 si la pile p est pleine, 0 sinon */
char pilePleine(PILE p){
  return (p.nbElts==TAILLE_MAX);
}

/* empile l'élément x sur la pile p, si celle-ci n'est pas pleine */
void empiler(PILE *p, ELEMENT x){
   assert(!pilePleine(*p));
   p->elts[p->nbElts++]=x;
}

/* retourne l'élément x au sommet de la pile p, si celle-ci n'est pas vide */
ELEMENT depiler(PILE *p){
   assert(!pileVide(*p));
   return p->elts[--p->nbElts];
}
```

Implémentation d'une file par un tableau. Une file peut être représentée par les éléments consécutifs d'un tableau (circulaire) de taille fixe et par deux entiers indiquant l'indice du premier élément inséré et le nombre total d'éléments insérés. La dimension du tableau étant fixée, il est nécessaire d'implanter une fonction booléenne *FilePleine ?* qui indiquera si la file est pleine ou non. En C, cela peut se programmer de la façon suivante :

```
/* Déclarations */
#define TAILLE_MAX 1000 // Nombre maximal d'éléments dans la file

typedef int ELEMENT; /* pour une pile d'entiers */

typedef struct {
  ELEMENT elts[TAILLE_MAX];
  int indPremierElt; // indice du premier élément
  int nbElts; // nombre d'éléments
} FiLE;

FILE f;

/* Initialisation */
f.nbElts=f.indPremierElt=0;

/* retourne 1 si la file f est vide, 0 sinon */
char fileVide(FiLE f){
  return (f.nbElts==0);
}

/* retourne 1 si la file f est pleine, 0 sinon */
char filePleine(FiLE f){
  return (f.nbElts==TAILLE_MAX);
}

/* enfile l'élément x dans la file f, si celle-ci n'est pas pleine */
void enfiler(FiLE *f, ELEMENT x){
   assert(!filePleine(*f));
   f->elts[(f->indPremierElt+f->nbElts++) % TAILLE_MAX]=x;
}

/* retourne le premier élément de la file f, si elle n'est pas vide */
```

```
ELEMENT defiler(FiLE *f){
   assert(!fileVide(*f));
   ELEMENT x=f->elts[f->indPremierElt];
   f->nbElts--;
   f->indPremierElt=(f->indPremierElt+1)%TAILLE_MAX;
   return x;
}
```

Exercice 4 La notation *post-fixée* des expressions arithmétiques, appelée encore *notation polonaise inversée*, consiste à écrire les opérandes avant les opérateurs. L'expression $((3+(5\times4)\times2)$ s'écrira par exemple $3\ 5\ 4\ \times\ +\ 2\ \times$. Aucune parenthèse n'est nécessaire et l'évaluation d'une telle expression se fait simplement au moyen de l'algorithme suivant :

- les termes sont parcourus de gauche à droite ;
- si le terme courant est un nombre, il est empilé ;
- si le terme courant est un opérateur, les deux derniers nombres empilés sont dépilés, l'opérateur leur est appliqué et le résultat est empilé ;
- lorsque l'expression est totalement parcourue, la pile ne contient plus qu'un seul élément : le résultat de l'évaluation.

Ecrivez un programme qui évalue une expression à partir de sa notation post-fixée (on supposera, pour simplifier, que les opérandes sont compris entre 0 et 9).

Exercice 5 Quelles structures de listes chaînées conviennent-elles le mieux pour implémenter un pile ? une liste ? Implémentez ces structures en C, ainsi que les fonctions correspondantes, au moyen de listes chaînées.

Exercice 6 Calcul de l'élément majoritaire (3ème méthode : les deux premières ont été décrites dans des exercices de la section précédente).

Supposons que l'on ait à déterminer s'il existe une couleur majoritaire parmi n balles colorées. On dispose d'une étagère pour poser les balles les unes à coté des autres et d'une corbeille pouvant contenir autant de balles que nécessaire. La méthode se déroule en deux phases :

phase 1 : Prendre les balles les unes après les autres en les mettant soit sur l'étagère, si la dernière balle posée sur l'étagère est de couleur différente, soit dans la corbeille dans le cas contraire. Dans le premier cas on posera ensuite sur l'étagère une balle prise au hasard dans la corbeille (s'il y en a).

phase 2 : Soit C la couleur de la dernière balle posée sur l'étagère. On jette les balles (pas dans la corbeille, dans la poubelle !) de l'étagère les unes

après les autres en commençant par les dernières de la façon suivante : (1) si la balle à jeter est de la couleur C et si elle a une voisine sur l'étagère, les jeter toutes les deux, (2) si la dernière balle sur l'étagère n'est pas de couleur C, la jeter et jeter aussi une balle de la corbeille si elle n'est pas vide, dans le cas contraire on peut s'arrêter, il n'y a pas d'élément majoritaire. Une fois le processus terminé, on regarde le contenu de la corbeille après y avoir mis éventuellement la denière boule restant sur l'étagère : si la corbeille est vide c'est qu'il n'y a pas de couleur majoritaire, sinon c'est que la couleur majoritaire est C.

1. Montrez qu'à tout moment de la phase 1, les balles de la corbeille, s'il y en a, sont toutes de la couleur de la dernière balle posée sur l'étagère. En déduire qu'alors s'il y a une couleur dominante c'est cette couleur la et que l'algorithme est donc correct.
2. Ecrivez l'algorithme en utilisant des piles. Quelle est la complexité dans le pire des cas (en nombre de comparaisons de couleurs) ?

3.4 Tables de hachage

Les *dictionnaires* ou *tableaux associatifs* sont des structures de données composées d'éléments de la forme *(clé,valeur)* où chaque clé détermine au plus une valeur.

On peut par exemple considérer le dictionnaire composé du numéro d'un étudiant inscrit à l'université d'Aix-Marseille et de son dossier, le dictionnaire composé d'un numéro de sécurité sociale et de l'assuré correspondant ou du dictionnaire composé des mots du français et de leur définition.

Les opérations de base associées à un dictionnaire sont : l'*insertion* d'un nouvel élément, la *recherche* d'un élément et la *suppression* d'un élément. Lorsque les clés sont des entiers appartenant à l'intervalle $[0 \dots n-1]$ (on peut toujours se ramener à ce cas) et lorsque n n'est pas trop grand, les dictionnaires peuvent être implémentés par des tableaux T à adressage direct dans lesquels les clés ne correspondant à aucune valeur sont associées conventionnellement à une valeur NIL : en effet, insérer un élément (c, x), rechercher si élément x de clé c est présent ou supprimer l'élément (c, x) se fait en temps constant. Mais le cas le plus fréquent est celui où le nombre de valeurs renseignées est très petit par rapport à n : voir les exemples cités précédemment.

Si l'on représente les n élements par une liste chaînée, chaque opération de base s'effectue en temps linéaire (par rapport à n) : on souhaiterait pouvoir les réaliser en temps constant.

Les *tables de hachage* (*hash tables*) sont des structures de données qui généralisent les tableaux au cas où l'ensemble U des clés possibles est gigantesque par rapport au nombre de valeurs à stocker. L'idée de base consiste à introduire une *fonction de hachage*

$$h : U \mapsto [0 \ldots m-1]$$

telle que m soit de l'ordre du nombre de valeurs à stocker et telle qu'*en pratique*, le nombre de *collisions*, c'est-à-dire le nombre de cas où deux éléments distincts du dictionnaire (c, x) et (c', x') vérifient $h(c) = h(c')$ soit le plus petit possible. Si l'on pouvait assurer qu'il n'y a pas de collisions, on pourrait implémenter le dictionnaire par un tableau T, chaque élément (c, x) étant représenté par l'élément $T(h(c))$. Mais il est en général impossible d'éviter toute collision.

Deux questions doivent être donc résolues :

1. comment gérer les collisions ?
2. comment définir une fonction de hachage minimisant le nombre de collisions ?

3.4.1 Gestion des collisions par chaînage externe

On appelle $h(c)$ la *position* de l'élément (c, x). Une collision correspond donc à deux éléments du dictionnaire possédant la même position. On choisit de représenter les éléments ayant la même position par une liste (simplement ou doublement) chaînée : $T[i]$ pointe alors vers les éléments dont la position est i. Insérer, supprimer ou rechercher un élément (c, x) est réalisé par les opérations correspondantes dans la liste chaînée $T(h(c))$.

Le pire des cas est celui où tous les éléments du dictionnaire occupent la même position ! Dans ce cas, on est ramené au cas du stockage du dictionnaire dans une liste chaînée et les opérations de base s'effectuent en temps linéaire. Mais si les clés se répartissent à peu près équitablement entre les différentes positions et si le nombre de position est de l'ordre du nombre d'élément à stocker, le nombre d'éléments de chaque liste chaînée est borné par une constante et les opérations de base peuvent donc être réalisées en temps constant. Voir ci-dessous les algorithmes d'insertion et de recherche d'un élément.

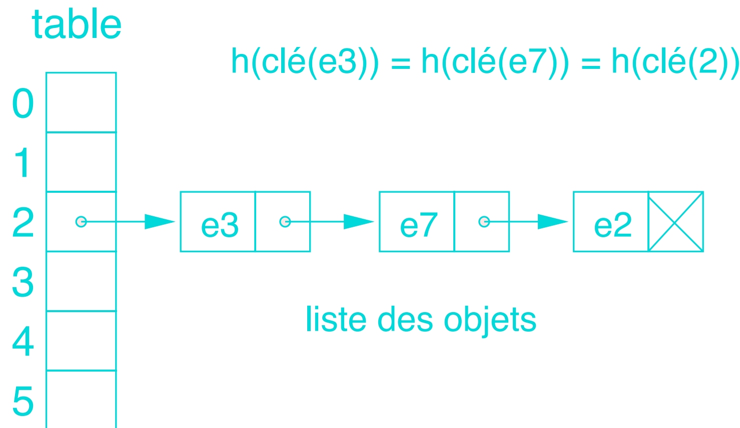


FIGURE 3.6 – Gestion des collisions par chaînage externe.

Algorithme 7: insertionTableHachage

---

entrée : Table de Hachage T, élément e de clé c
résultat : e est inséré dans T
début
 | Insérer e en tête de la liste chainée $T(h(c))$.
fin

Fonction rechercheTableHachage(T,c)

---

entrée : Table de Hachage T, clé c
sortie : un pointeur vers l'élément de clé c ou NIL si l'élément est absent
début
 | $p = T[h(c)]$
 | **tant que** $p \neq NIL\ ET\ p$ *ne pointe pas vers l'élément de clé* c **faire**
 | | $p \leftarrow p.suiv$
 | **fintq**
 | **retourner** p
fin

3.4.2 Gestion des collisions par adressage ouvert

La gestion des collisions par adressage ouvert consiste à utiliser la table de hachage elle-même pour stocker les éléments. Pour cela, on utilise une

fonction de hachage modifiée

$$h : U \times [0 \dots m-1] \mapsto [0 \dots m-1]$$

telle que pour toute clé c, $h(c,0), \dots, h(c,m-1)$ réalise une permutation de $[0 \dots m-1]$.

Pour insérer un nouvel un élément e de clé c, on cherche le premier indice i tel que $T[h(c,i)]$ soit libre et on insére e dans $T[h(c,i)]$: toutes les cases du tableau peuvent donc potentiellement recevoir tous les éléments.

Pour savoir si un élément de clé c est présent dans T, on parcourt les éléments de la liste $T[h(c,0)], \dots, T[h(c,m-1)]$ jusqu'à le trouver ou tomber sur une case libre ou avoir parcouru toutes les cases du tableau.

En revanche, la suppression d'un élément est plus complexe car il ne suffit pas de vider la case qui le contient. En effet, supposons

- que l'élément de clé c soit stocké dans la case d'indice $h(c,i)$,
- que l'élément de clé c' ait été stocké postérieurement à l'élément de clé c dans la case d'indice $h(c',i')$,
- qu'il existe un indice $j < i'$ tel que $h(c',j) = h(c,i)$.

Si on libère simplement la case $T(h(c,i))$, l'élément de clé c' ne sera plus trouvé. Une solution consiste à distinguer les cases *libres* des cases *supprimées*, dans lesquelles de nouvelles valeurs pourront être insérées mais qui ne devront pas être considérées comme libres lors d'une recherche d'élément. D'après l'ouvrage de référence, la gestion des collisions par chaînage externe est plus souvent utilisée lorsqu'il doit y avoir des suppressions.

3.4.3 Fonctions de hachage

Une bonne fonction de hachage doit satisfaire la condition suivante : les clés des éléments du dictionnaire doivent se répartir (à peu près) uniformément entre les différentes positions.

Des informations sur la distribution des clés peuvent permettre de définir une fonction de hachage optimale. C'est par exemple le cas lorsqu'on sait que les clés sont des nombres réels indépendamment et uniformément distribués dans l'intervalle $[0,1[$, on peut utiliser la fonction $h(c) = \lfloor mc \rfloor$.

En l'absence d'information, on cherche à définir des fonctions de hachage dépendant le moins possible des données. On décrit ci-dessous deux méthodes courantes lorsque les clés sont des entiers, cas auquel on peut toujours se ramener.

Méthode par division On définit

$$h(c) = c \mod m.$$

La position de la clé c est son reste dans la division par m. C'est un méthode simple qui peut donner des résultats peu satisfaisants pour certaines valeurs de m. Par exemple, si $m = 2^p$ est une puissance de 2, la position d'une clé ne dépend que de ses p derniers bits ; si ceux-ci ne sont pas uniformément répartis, la fonction de hachage correspondante ne sera pas uniforme non plus.

En pratique, on recommande de choisir pour m un nombre premier pas trop proche d'une puissance de 2.

Méthode par multiplication On définit

$$h(c) = \lfloor m\{cA\} \rfloor$$

où A est un nombre réel tel que $0 < A < 1$ et où $\{x\}$ désigne la *partie fractionnaire* de x, c'est-à-dire $x - \lfloor x \rfloor$.

Cette méthode est plus robuste que la précédente au choix des valeurs de A et m. On choisit souvent m égal à une puissance de 2 et Donald Knuth [1] suggère de prendre $A = (\sqrt{5} - 1)/2$ (cité dans l'ouvrage de référence).

On peut toujours trouver des exemples qui mettent en défaut n'importe quelle fonction de hachage, c'est-à-dire tels que presque toutes les clés se retrouvent assignés une position unique. On peut remédier à ce problème en introduisant des fonctions de hachage randomisées, mais ces techniques dépassent le niveau de ce cours.

Les techniques précédentes ne concernent que les méthodes de hachage par chaînage externe. Si l'on souhaite gérer les collisions par adressage ouvert, on doit définir des fonctions de hachage à deux arguments. Les méthodes le plus couramment utilisées sont :

le sondage linéaire (linear probing) : à partir d'une fonction de hachage simple $h : U \mapsto [0 \ldots m-1]$, on définit la fonction $h' : U \times [0 \ldots m-1] \mapsto [0 \ldots m-1]$ par

$$h'(c, i) = h(c) + i \mod m.$$

On vérifie aisément que pour c fixée, $h'(c, i)$ parcourt toutes les valeurs de $[0 \ldots m-1]$ lors que i décrit $[0 \ldots m-1]$.

L'inconvénient principal de cette fonction est qu'elle a tendance à créer de longues suites consécutives de positions occupées, ce qui a pour effet de ralentir le temps moyen de recherche d'un élément. Un moyen de remédier à cela est de considérer des incréments qui ne soient pas constants.

1. Donald Knuth est l'un des principaux pionniers en algorithmique et son livre *The art of computer programming* reste une référence majeure.

le sondage quadratique (quadratic probing) : à partir d'une fonction de hachage simple $h : U \mapsto [0 \ldots m-1]$, on définit la fonction $h' : U \times [0 \ldots m-1] \mapsto [0 \ldots m-1]$ par

$$h'(c,i) = h(c) + ai + bi^2 \mod m$$

où a et b sont des constantes auxiliaires. Voir dans un exercice ci-dessous un exemple de choix de ces constantes lorsque m est une puissance de 2.

double hachage (double probing) : à partir de deux fonctions de hachage simples $h_1, h_2 : U \mapsto [0 \ldots m-1]$, on définit la fonction $h' : U \times [0 \ldots m-1] \mapsto [0 \ldots m-1]$ par

$$h'(c,i) = h_1(c) + ih_2(c) \mod m.$$

Pour que $h'(c,i)$ réalise une permutation de $[0 \ldots m-1]$ pour toute position $h(c)$, il est nécessaire que $h_2(c)$ soit premier avec m. C'est toujours le cas si m est une puissance de 2 et si h_2 ne prend que des valeurs impaires, ou si m est premier et si h_2 ne prend que des valeurs $< m$. On peut prendre par exemple

$$h_1(c) = c \mod m \text{ et } h_2(c) = 1 + (c \mod m-1)$$

où m est premier.

La méthode de double hachage est considérée comme l'une des meilleures.

Exercice 7 On considère l'ensemble des mots construits sur l'alphabet latin (26 lettres) et l'on définit la fonction de hachage suivante :

$$h(c_0 c_1 \ldots c_k) = (c_0 + 26c_1 + \ldots + 26^k c_k) \mod m$$

où les lettres c_i sont identifiées à leur numéro $\in \{0 \ldots 25\}$ et où m est un entier bien choisi.

1. Que se passe t-il si l'on prend $m = 26$?
2. Montrez que si $m = 25$, tous les anagrammes ont la même position.
3. Proposez une méthode algorithmique pour calculer pratiquement la position d'un mot quelconque et programmez-la.

Exercice 8 Sondage quadratique. Montrez que si $m = 2^p$, et si $a = b = 1/2$, $h(c) + ai + bi^2 \mod m$ réalise une permutation de $[0 \ldots m-1]$ quelle que soit la valeur de $h(c)$.

Exercice 9 Etudiez les valeurs prise par la fonction de double hachage proposée ci-dessus pour $m = 11$.

Chapitre 4

Algorithmes de tris

Le tri d'un ensemble d'objets consiste à les ordonner en fonction de clés et d'une relation d'ordre définie sur cette clé. Le tri est une opération classique et très fréquente. De nombreux algorithmes et méthodes utilisent des tris. Par exemple pour l'algorithme de Kruskal qui calcule un arbre couvrant de poids minimum dans un graphe, une approche classique consiste, dans un premier temps, à trier les arêtes du graphe en fonction de leurs poids. Autre exemple, pour le problème des éléphants, trouver la plus longue séquence d'éléphants pris dans un ensemble donné, telle que les poids des éléphants dans la séquence soient croissants et que leurs Q.I. soient décroissants, une approche classique consiste à considérer une première suite contenant tous les éléphants ordonnés par poids croissants, une deuxième suite avec les éléphants ordonnés par Q.I. décroissants, puis à calculer la plus longue sous-séquence commune à ces deux suites. Trier un ensemble d'objets est aussi un problème simple, facile à décrire, et qui se prête à l'utilisation de méthodes diverses et variées. Ceci explique l'intérêt qui lui est porté et le fait qu'il est souvent présenté comme exemple pour les calculs de complexité.

Dans le cas général on s'intéresse à des tris *en place*, c'est-à-dire des tris qui n'utilisent pas d'espace mémoire supplémentaire pour stocker les objets, et *par comparaison*, c'est-à-dire que le tri s'effectue en comparant les objets entres eux. Un tri qui n'est pas par comparaison nécessite que les clés soient peu nombreuses et connues à l'avance, et peuvent être indexées facilement. Un tri est *stable* s'il préserve l'ordre d'apparition des objets en cas d'égalité des clés. Cette propriété est utile par exemple lorsqu'on trie successivement sur plusieurs clés différentes. Si l'on veut ordonner les étudiants par rapport à leur nom puis à leur moyenne générale, on veut que les étudiants qui ont la même moyenne apparaissent dans l'ordre lexicographique de leurs noms.

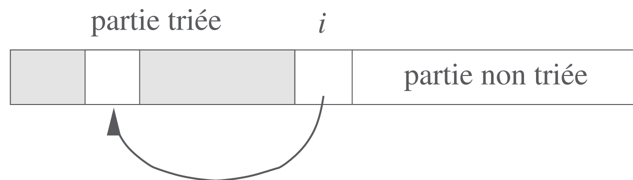
Dans ce cours nous distinguerons les tris en $O(n^2)$ (tri à bulle, tri par insertion, tri par sélection), les tris en $O(n \times log\, n)$ (tris par fusion, tri par tas et tri rapide, bien que ce dernier n'ait pas cette complexité dans le pire des cas) et les autres (tris spéciaux instables ou pas toujours applicables). Il convient aussi de distinguer le coût théorique et l'efficacité en pratique : certains tris de même complexité ont des performances très différentes dans la pratique. Le tri le plus utilisé et globalement le plus rapide est le tri rapide (un bon nom–*quicksort*) ; nous l'étudierons en TD.

En général les objets à trier sont stockés dans des tableaux indexés, mais ce n'est pas toujours le cas. Lorsque les objets sont stockés dans des listes chainées, on peut soit les recopier dans un tableau temporaire, soit utiliser un tri adapté comme le tri par fusion.

4.1 Tri par sélection, tri par insertion.

Le **tri par sélection** consiste simplement à *sélectionner* l'élément le plus petit de la suite à trier, à l'enlever, et à répéter itérativement le processus tant qu'il reste des éléments dans la suite. Au fur et à mesure les éléments enlevés sont stockés dans une pile. Lorsque la suite à trier est stockée dans un tableau on s'arrange pour représenter la pile dans le même tableau que la suite : la pile est représentée au début du tableau, et chaque fois qu'un élément est enlevé de la suite il est remplacé par le premier élément qui apparaît à la suite de la pile, et prends sa place. Lorsque le processus s'arrête la pile contient tous les éléments de la suite triés dans l'ordre croissant.

Le **tri par insertion** consiste à *insérer* les éléments de la suite les uns après les autres dans une suite triée initialement vide. Lorsque la suite est stockée dans un tableau la suite triée en construction est stockée au début du tableau. Lorsque la suite est représentée par une liste chainée on insère les maillons les uns après les autres dans une nouvelle liste initialement vide.



procédure **TriParInsertion**(E)
(**InOut :** E la suite $(e_1, \ldots, e_n)$)
début
    pour $i := 2$ **jusqu'à** n **faire**
        $j := i$, $v := e_i$,
        tant que $((j > 1)$ et $(v < e_{j-1}))$ **faire**
            $e_j := e_{j-1}$,
            $j := j - 1$,
        fin faire
        $e_j := v$,
    fin faire
fin procédure

Le tri effectue $n - 1$ insertions. A la $i$ème itération, dans le pire des cas, l'algorithme effectue $i - 1$ recopies. Le coût du tri est donc $\Sigma_{i=2}^{n}(i - 1) = O(n^2)$. Remarquons que dans le meilleur des cas le tri par insertion requiert seulement $O(n)$ traitements. C'est le cas lorsque l'élément à insérer reste à sa place, donc quand la suite est déja triée (lorsque la suite est stockée dans une liste chainée c'est la cas lorsque la liste est triée à l'envers puisqu'on insère en tête de liste).

4.2 Tri par fusion et tri rapide

4.2.1 Tri par fusion

Le tri par fusion (*merge sort* en anglais) implémente une approche de type diviser pour régner très simple : la suite à trier est tout d'abord scindée en deux suites de longueurs égales à un élément près. Ces deux suite sont ensuite triées séparément avant d'être fusionnées. L'efficacité du tri par fusion vient de l'efficacité de la fusion : le principe consiste à parcourir simultanément les deux suites triées dans l'ordre croissant de leur éléments, en extrayant chaque fois l'élément le plus petit.

Le tri par fusion est bien adapté aux listes chainées : pour scinder la liste il suffit de la parcourir en liant les éléments de rangs pairs d'un coté et les éléments de rangs impairs de l'autre. La fusion de deux listes chainées se fait facilement. Inversement, si la suite à trier est stockée dans un tableau il est nécessaire de faire appel à un tableau annexe lors de la fusion, sous peine d'avoir une complexité en $O(n^2)$.

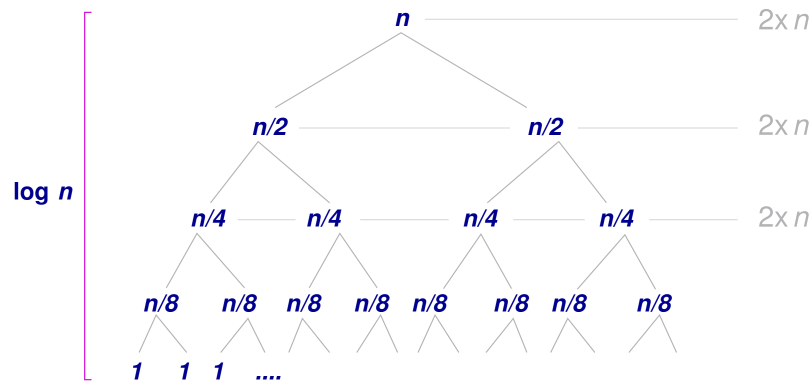
Fonction TriParFusion(S)

```
Data : S : la suite à trier
begin
    if |S| ≥ 1 then
        scinder la suite S en deux suites S1 et S2 de longueurs égales;
        S1 := TriParFusion (S1);
        S2 := TriParFusion (S2);
        S := fusion (S1, S2);
    return S;
```

Dans le cas général, on peut évaluer à $O(n)$ le coût de la scission de la suite S et à $O(n)$ le coût de la fusion des suites S_1 et S_2. L'équation récursive du tri par fusion est donc

$$T(n) = 1 + O(n) + 2 \times T(n/2) + O(n)$$

On en déduit que le tri par fusion est en $O(n \log n)$. On le vérifie en cumulant les nombres de comparaisons effectuées à chaque niveau de l'arbre qui représente l'exécution de la fonction (voir figure ci-dessous) : chaque noeud correspond à un appel de la fonction, ses fils correspondent aux deux appels récursifs, et son étiquette indique la longueur de la suite. La hauteur de l'arbre est donc $\log_2 n$ et à chaque niveau le cumul des traitements locaux (scission et fusion) est $O(n)$, d'où on déduit un coût total de $O(n) \times \log_2 n = O(n \log n)$.



La fusion peut-être effectuée en conservant l'ordre des éléments de même valeur (le tri par fusion est stable), mais elle nécessite l'utilisation de tableaux auxiliaires (au moins dans ses implémentations les plus courantes).

Exercice 1 Écrivez l'algorithme de fusion.

4.2.2 Tri rapide

Le tri rapide est une méthode de type *diviser pour régner*. L'idée de base est la suivante : pour trier la suite $S = (s_g, \ldots, s_d)$ on la partitionne en deux sous suites non vides $S' = (s_g, \ldots, s_q)$ et $S'' = (s_q, \ldots, s_d)$ telles que les éléments les plus petits sont dans la première et les éléments les plus grands dans la seconde. En appliquant récursivement le tri sur les suites S' et S'' la suite S est triée.

Il existe plusieurs méthodes pour partitionner une suite. Le principe général consiste à utiliser un élément particulier de la suite, le *pivot*, comme valeur de partage. Il faut faire très attention à ne pas produire une suite vide et l'autre contenant tous les éléments de la suite initiale, auquel cas le tri risque de boucler indéfiniment. Une façon simple de contourner ce problème est d'isoler les éléments de même valeur que le pivot. Ces éléments sont simplement placés entre les deux sous suites après la partition, c'est leur place définitive.

Nous allons écrire plusieurs versions de la partition, en utilisant des tableaux pour stocker les éléments de la suite à trier.

Question 1. Ecrivez une fonction `partition` en $O(n)$ où n est le nombre d'éléments de la suite en vous inspirant de la méthode dite *du drapeau* : la partie du tableau où sont stockés les éléments de la suite est segmentée en quatre zones, contenant respectivement des éléments de valeur inférieure au pivot, des élément de même valeur que le pivot, des éléments de valeur supérieure au pivot et enfin les éléments restant (qui n'ont pas encore été traités). Le partitionnement consiste simplement à prendre un élément de la dernière zone et à l'ajouter dans l'une des trois premières zones suivant sa valeur, puis à répéter cette opération tant qu'il reste des éléments non traités.

Question 2. Ecrivez une fonction récursive de tri rapide qui utilise la fonction de partition précédente.

Question 3 . En terme de complexité, quel est pire des cas pour le tri rapide, et comment éviter ce cas avec une bonne probabilité.

Question 4. Il existe une façon plus efficace pour partitionner la suite : le principe général consiste à parcourir la suite simultanément en partant de

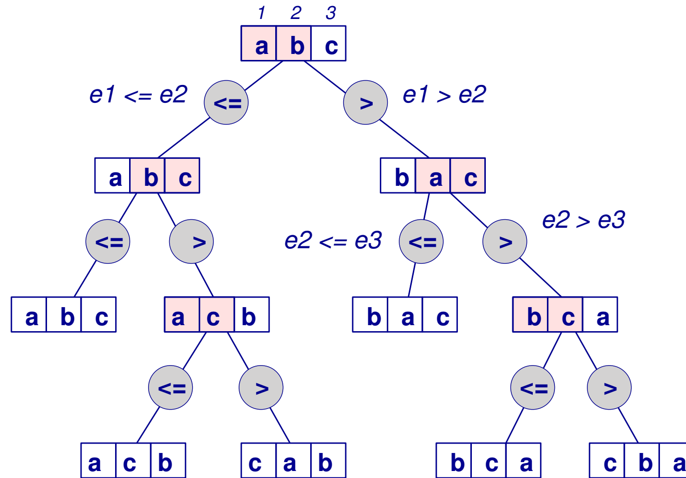
la gauche (et en allant vers la droite) et en partant de la droite (et en allant vers la gauche), afin de trouver à gauche une valeur supérieure à la valeur du pivot, et à droite une valeur inférieure à la valeur du pivot. Il suffit ensuite d'échanger ces deux valeurs et de continuer le processus tant que les deux indices ne se sont pas croisés. Cette partition produit deux sous suites, l'une contenant des valeurs plus petites ou égales à la valeur du pivot et l'autre contenant des valeurs plus grandes ou égales à la valeur du pivot. Ecrivez une fonction de partition qui suit ce schéma et justifiez la, c'est à dire démontrez qu'elle se termine et que le résultat est le bon. En particulier il vous faut démontrer qu'aucune des deux suites produites ne peut être vide.

Ecrivez une nouvelle version du tri rapide qui intégre cette partition.

Question 5. Ecrivez une fonction qui, étant donné un entier k et une suite S, renvoie la valeur du $k$ème plus petit élément du tableau, c'est à dire de l'élement de rang k dans la suite triée. Cette fonction fera appel à la fonction `partition`. Dans quels cas cette méthode est-elle meilleure que si on avait tout d'abord trié la suite.

4.3 Complexité optimale d'un algorithme de tri par comparaison

L'*arbre de décision* d'un tri par comparaison représente le comportement du tri dans toutes les configurations possibles. Les configurations correspondent à toutes les permutations des objets à trier, en supposant qu'ils soient tous comparables et de clés différentes. S'il y a n objets à trier, il y a donc $n!$ configurations possibles. On retrouve toutes ces configurations sur les feuilles de l'arbre, puisque deux permutations initiales distinctes ne peuvent pas produire le même comportement du tri : en effet, dans ce cas le tri n'aurait pas fait son travail sur un des deux ordonnancements. Chaque noeud de l'arbre correspond à une comparaison entre deux éléments et a deux fils, correspondants aux deux ordres possibles entre ces deux éléments.



Sur la figure ci-dessus nous avons représenté l'arbre de décision du tri par insertion sur une suite de trois éléments. A la racine de l'arbre le tri compare tout d'abord les deux premiers éléments de la suite a et b, afin d'insérer l'élément b dans la suite triée constituée uniquement de l'élément a. Suivant leur ordre les deux éléments sont permutés (fils droit) ou laissés en place (fils gauche). Au niveau suivant l'algorithme compare les éléments de rang 2 et de rang 3 afin d'insérer l'élement de rang 3 dans la suite triée constituée des 2 premiers éléments, et ainsi de suite. Remarquons que les branches de l'arbre de décision du tri par insertion n'ont pas toutes la même longueur du fait que dans certains cas l'insertion d'un élément est moins couteuse, en particulier quand l'élement est déja à sa position.

Puisque le nombre de permutations de n éléments est $n!$, l'arbre de décision d'un tri a donc $n!$ feuilles. La hauteur d'un arbre binaire de $n!$ feuilles est, dans le meilleur des cas, c'est-à-dire si l'arbre est parfaitement équilibré (le nombre de noeuds est multiplié par deux chaque fois qu'on descend d'un niveau)

$$h = \log(n!)$$

or, d'après la formule de Stirling,

$$\log(n!) \geq \log\left(\frac{n}{e}\right)^n \simeq n \times \log n - n \times \log e$$

et donc la hauteur minimale de l'arbre est de l'ordre de $n \times \log n$.

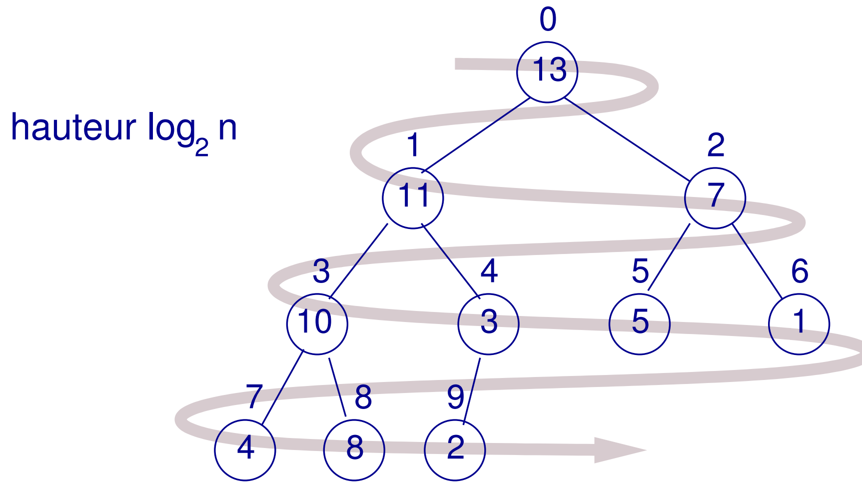
On en conclut qu'aucun tri par comparaison ne peut avoir une complexité meilleure que $O(n \times \log n)$.

4.4 Tri par tas.

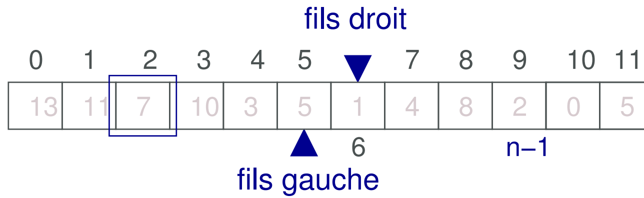
Un *tas* (*heap* en anglais) ou *file de priorité* (*priority queue*) est un arbre binaire étiqueté presque complètement équilibré : tous ses niveaux sont remplis sauf peut-être le dernier, qui est rempli à partir de la gauche jusqu'à un certain noeud. On définit ensuite une propriété sur les tas : chaque noeud est dans une relation d'ordre fixée avec ses fils. En général on considère des tas dans lesquels chaque noeud a une valeur plus petite que celles de ses fils. Pour le tri par tas on utilise des *arbres maximiers* (*max-heap*), c'est-à-dire des tas dans lesquels chaque noeud porte une valeur plus grande que celles de ses fils. La valeur la plus grande du tas se trouve donc à la racine. Les opérations et les techniques présentées dans ce chapitre pour des arbres maximiers s'appliquent de la même façon à des tas basés sur l'ordre inverse. Les opérations essentielles sur les tas sont :

- la construction du tas,
- l'extraction du maximum,
- l'ajout d'une nouvelle valeur,
- la modification de la valeur d'un noeud.

Habituellement les tas sont des arbres binaires représentés dans des tableaux. Les valeurs du tas sont stockées dans les premières cases du tableau. Si le tas est composé de n éléments, ces derniers apparaissent donc aux indices $0, 1, \ldots, n-1$. La racine du tas figure dans la case d'indice 0. La disposition des éléments du tas dans le tableau correspond à un parcours de l'arbre par niveau, en partant de la racine et de gauche à droite.



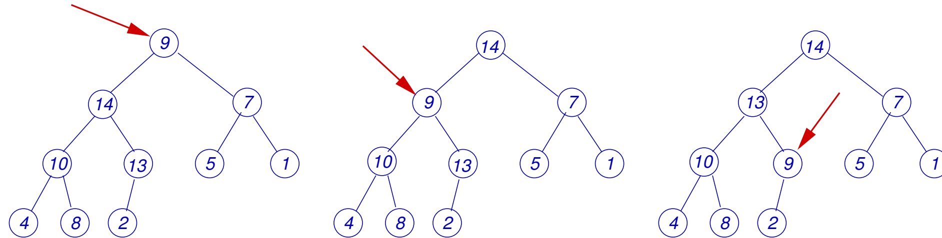
Le fils gauche du noeud qui figure à l'indice i, s'il existe, se trouve à l'indice $\text{FilsG}(i) = 2 \times i + 1$, et son fils droit, s'il existe, se trouve à l'indice $\text{FilsD}(i) = 2 \times i + 2$. Inversement, le père du noeud d'indice i non nul se trouve à l'indice $\lfloor \frac{i-1}{2} \rfloor$.



Cette disposition entraîne que l'arbre est forcément équilibré (i.e. toutes ses branches ont la même longueur à un élément près), et les plus longues branches sont à gauche. La hauteur d'un tas, i.e. son nombre de niveaux, contenant n éléments est donc $\lfloor \log_2 n \rfloor + 1$ (le nombre de fois que l'on peut diviser n par 2 avant d'obtenir 1). Si le dernier noeud du tas se trouve à la position $n - 1$, son père se trouve à la position $\lfloor \frac{n}{2} - 1 \rfloor$. C'est le dernier noeud qui a au moins un fils. Les feuilles de l'arbre se trouvent donc entre la position $\frac{n}{2}$ et la position $n - 1$ dans le tableau puisqu'il n'y a pas de feuilles avant le dernier père.

L'opération *Entasser* est une opération de base sur les tas. Elle est utilisée notamment pour la construction des tas ou encore pour l'extraction de la

valeur maximale. Elle consiste à reconstruire le tas lorsque seule la racine viole (éventuellement) la propriété de supériorité entre un noeud et ses fils, en faisant descendre dans l'arbre l'élément fautif par des échanges successifs.



Fonction Entasser(i,T,n)

entrée : T est un tas ; le fils gauche et le fils droit du noeud d'indice i vérifient la propriété *Max-heap* ; ce n'est pas forcément le cas du noeud d'indice i.
sortie : La propriété *max-heap* est vérifiée par le noeud d'indice i.
début
- $iMax \leftarrow i$
- **si** (FilsG(i) $< n$) *ET* (T[FilsG(i)] $> T[iMax]$) **alors**
  - $iMax :=$ FilsG(i)
- **si** (FilsD(i) $< n$) *ET* (T[FilsD(i)] $> T[iMax]$) **alors**
  - $iMax :=$ FilsD(i)
- **si** ($iMax \neq i$) **alors**
  - Echanger $T[i]$ et $T[iMax]$
  - Entasser($iMax$,T,n)

La fonction procède de la façon suivante : si l'on n'a pas atteint une feuille ou que la valeur du noeud courant d'indice i viole la propriété de supériorité des pères sur leurs fils, on échange cette valeur avec la plus grande des valeurs des fils. De cette façon le noeud d'indice i aura une valeur plus grande que les valeurs de ses fils. On réitère l'opération tant que la propriété de tas n'est pas rétablie.

La fonction **Entasser** a un coût en $O(\log_2 n)$ puisque, dans le pire des cas, il faudra parcourir une branche entièrement.

Extraction de la valeur maximale. La valeur maximale d'un tas qui vérifie la propriété *Max-heap* est à la racine de l'arbre. Pour un tas de taille

n stocké dans le tableau T c'est la valeur $T[0]$ si n est non nul. L'extraction de la valeur maximale consiste à recopier en $T[0]$ la dernière valeur du tas $T[n-1]$, à décrémenter la taille du tas, puis à appliquer l'opération *Entasser* à la racine du tas, afin que la nouvelle valeur de la racine prenne sa place.

```
fonction ExtraireLeMax(T, n)
    max := T[0],
    T[0] := T[n − 1],
    Entasser(0, T, n − 1),
    renvoyer ⟨max, T, n − 1⟩,
fin fonction
```

Insérer une nouvelle valeur dans un tas consiste à ajouter la valeur à la fin du tas, en dernière position dans le tableau, puis à la faire remonter dans le tas en l'échangeant avec son père tant qu'elle ne se trouve pas à la racine et que la valeur de son père lui est inférieure. L'opération d'insertion n'est pas utile pour le tri par tas.

Principe général du tri par tas. Supposons que l'on ait à trier une suite de n valeurs stockée dans un tableau T. On commence par construire un tas dans T avec les valeurs de la suite. Ensuite, tant que le tas n'est pas vide on répète l'extraction de la valeur maximale du tas. Chaque fois, la valeur extraite est stockée dans le tableau immédiatement après les valeurs du tas. Lorsque le processus se termine on a donc la suite des valeurs triée dans le tableau T.

Construction du tas.

Les valeurs de la suite à trier sont stockées dans le tableau T. La procédure consiste à parcourir les noeuds qui ont des fils et à leur appliquer l'opération Entasser, en commençant par les noeuds qui sont à la plus grande profondeur dans l'arbre. Il suffit donc de parcourir les noeuds dans l'ordre décroissant des indices et en partant du dernier noeud qui a des fils, le noeud d'indice $\lfloor (n/2) \rfloor - 1$. L'ordre dans lequel les noeuds sont traités garantit que les sous-arbres sont des tas.

```
fonction ConstruireUnTas(T, n)
    pour i := n/2 − 1 jusqu'à 0 faire
        Entasser(i, T, n),
fin fonction
```

Complexité de la construction. De façon évident la complexité est au plus $O(n \log n)$. En effet la hauteur du tas est $\log n$ et on effectue $n/2$ fois l'opération Entasser. En fait le coût de la construction est $O(n)$. La fonction effectue un grand nombre d'entassements sur des arbres de petites hauteur (pour les noeuds les plus profonds), et très peu d'entassements sur la hauteur du tas. Considérons pour simplifier un tas complet, c'est-à-dire dans lequel toutes les branches ont la même longueur. On dénombre $\frac{n}{2}$ noeuds à la hauteur 0 (les feuilles), $\frac{n}{2^2}$ noeuds à la hauteur 1, $\frac{n}{2^3}$ noeuds à la hauteur 2,..., On a donc $\frac{n}{2^{h+1}}$ noeuds à la hauteur h, pour chacun desquels on effectuera au plus h échanges dans l'opération Entasser. Le coût de la construction du tas est donc borné par

$$\sum_{h=1}^{\log n} \lceil \frac{n}{2^{h+1}} \rceil \times h \leq n \times \sum_{h=1}^{\log n} \frac{h}{2^h}$$

Puisque

$$\sum_{i=0}^{\infty} i \times x^i = \frac{x}{(1-x)^2}$$

on a

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

et donc le nombre d'échanges est borné par

$$n \times \sum_{h=0}^{\log n} \frac{h}{2^h} \leq 2 \times n = O(n)$$

Tri par tas.

On construit un tas. On utilise le fait que le premier élément du tas est le plus grand : autant de fois qu'il y a d'éléments, on extrait le premier du tas et on reconstruit le tas avec les éléments restants. Enlever le premier élément consiste simplement à l'échanger avec le dernier du tas et à décrémenter la taille du tas. On rétablit la propriété de tas en appliquant l'opération *Entasser* sur ce premier élément.

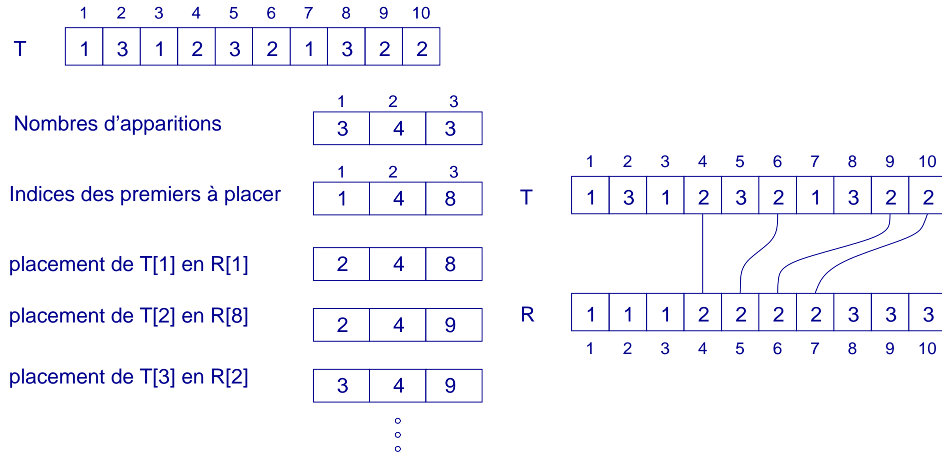
fonction **TriParTas**(T, n)
  ConstruireUnTas(T, n),
  pour $i := n - 1$ **jusqu'à** 1 **faire**
    Echanger$(T, 0, i)$,
    Entasser$(0, T, i)$,
fin fonction

La construction du tas coûte $O(n)$. On effectue ensuite n fois un échange et l'opération Entasser sur un tas de hauteur au plus log$n$. La complexité du tri par tas est donc $O(n \log n)$.

4.5 Tri par dénombrement, tri par base.

4.5.1 Tri par dénombrement

Le tri par dénombrement (*counting sort*) est un tri sans comparaisons qui est stable, c'est-à-dire qu'il respecte l'ordre d'apparition des éléments dont les clés sont égales. Un tri sans comparaison suppose que l'on sait indexer les élément en fonction de leur clé. Par exemple, si les clés des éléments à trier sont des valeurs entières comprises entre 0 et 2, on pourra parcourir les éléments et les répartir en fonction de leur clé sans les comparer, juste en utilisant les clés comme index. Le tri par dénombrement utilise cette propriété pour tout d'abord recenser les éléments pour chaque valeur possible des clés. Ce comptage préliminaire permet de connaître, pour chaque clé c, la position finale du premier élément de clé c qui apparaît dans la suite à trier. Sur l'exemple ci-dessous on a recensé dans le tableau T, 3 éléments avec la clé 0, 4 éléments avec la clé 1 et 3 éléments avec la clé 2. On en déduit que le premier élément avec la clé 0 devra être placé à la position 0, le premier élément avec la clé 1 devra être placé à la position 3, et le premier élément avec la clé 2 devra être placé à la position 7. Il suffit ensuite de parcourir une deuxième fois les éléments à trier et de les placer au fur et à mesure dans un tableau annexe (le tableau R de la figure), en n'oubliant pas, chaque fois qu'un élément de clé c est placé, d'incrémenter la position de l'objet suivant de clé c. De cette façon les éléments qui ont la même clés apparaissent nécessairement dans l'ordre de leur apparition dans le tableau initial.



fonction **TriParDenombrements**(T, n)
{**In :** T un tableau de n éléments}
{**Out :** R le tableau trié des éléments de T}
début

pour $i := 1$ **à** k **faire**	*initialisations*
$\quad Nb[i] := 0,$	
pour $i := 1$ **à** n **faire**	*calcul des nombres d'apparitions*
$\quad Nb[T[i]] := Nb[T[i]] + 1,$	
$Nb[k] := n - Nb[k] + 1,$	*calcul des indices du premier*
pour $i := k - 1$ **à** 1 **faire**	*élément de chaque catégorie*
$\quad Nb[i] := Nb[i + 1] - Nb[i],$	
pour $i := 1$ **à** n **faire**	*recopie des élément originaux*
$\quad R[Nb[T[i]]] := T[i],$	*du tableau T dans R*
$\quad Nb[T[i]] := Nb[T[i]] + 1,$	
renvoyer R	

fin procédure

La suite des objets à trier est parcourue deux fois, et la table Nb contenant le nombre d'occurrences de chaque clé est parcourue une fois pour l'initialiser. La complexité finale est donc $O(n+k)$ si les clés des éléments à trier sont comprises entre 0 et k. Le tri par dénombrement est dit *linéaire* (modulo le fait que k doit être comparable à n).

4.5.2 Tri par base.

Le tri par dénombrement est difficilement applicable lorsque les valeurs que peuvent prendre les clés sont très nombreuses. Le principe du tri par base (*radix sort*) consiste, dans ce type de cas, à fractionner les clés, et à effectuer un tri par dénombrement successivement sur chacun des fragments des clés. Si on considère les fragments dans le bon ordre (i.e. en commençant par les fragments de poids le plus faible), après la dernière passe, l'ordre des éléments respecte l'ordre lexicographique des fragments, et donc la suite est triée.

Considérons l'exemple suivant dans lequel les clés sont des nombres entiers à au plus trois chiffres. Le fractionnement consiste simplement à prendre chacun des chiffres de l'écriture décimale des clés. La colonne de gauche contient la suite des valeurs à trier, la colonne suivante contient ces mêmes valeurs après les avoir trié par rapport au chiffre des unités,... Dans la dernière colonne les valeurs sont effectivement triées. Du fait que le tri par dénombrement est stable, si des valeurs ont le même chiffre des centaines, alors elles apparaitront dans l'ordre croissant de leurs chiffres des dizaines, et si certaines ont le même chiffre des dizaines alors elles apparaitront dans l'ordre croissant des chiffres des unités.

536	592	427	167
893	462	536	197
427	893	853	427
167	853	462	462
853	536	167	536
592	427	592	592
197	167	893	853
462	197	197	893

Supposons que l'on ait n valeurs dont les clés sont fractionnées en c fragments avec k valeurs possibles pour chaque fragment. Le coût du tri par base est alors $O(c \times n + c \times k)$ puisque l'on va effectuer c tris par dénombrement sur n éléments avec des clés qui auront k valeurs possibles.

Si $k = O(n)$ on peut dire que le tri par base est linéaire. Dans la pratique, sur des entiers codés sur 4 octets que l'on fragmente en 4, le tri par base est moins rapide que le tri rapide (Quick Sort).

4.6 Tri shell.

C'est une variante du tri par insertion. Le principe du tri shell est de trier séparément des sous-suites de la table formées par des éléments pris de h en h dans la table (on nommera cette opération h-*ordonner*).

Définition. La suite $E = (e_1, \ldots, e_n)$ est h-*ordonnée* si pour tout indice $i \leq n - h$, $e_i \leq e_{i+h}$.

Si h vaut 1 alors une suite h-ordonnée est triée.

Pour trier, Donald Shell le créateur de ce tri, propose de h-ordonner la suite pour une série décroissante de valeurs de h. L'objectif est d'avoir une série de valeurs qui permette de confronter tous les éléments entres eux le plus souvent et le plus tôt possible. Dans la procédure ci-dessous la suite des valeurs de h est : $\ldots, 1093,\ 364, 121,\ 40,\ 13,\ 4,\ 1$.

La complexité de ce tri est $O(n^2)$. Avec la suite de valeurs de h précédente on atteint $O(n^{3/2})$ (admis). Cependant dans la pratique ce tri est très performant et facile à implémenter (conjectures $O(n \times (\log n)^2)$ ou $n^{1,25}$).

```
procédure TriShell(E)
(InOut : E la suite (e_1, ..., e_n))
début
    h := 1,
    tant que h ≤ n/9 faire
        h := 3 × h + 1,
    fin faire
    tant que h > 0 faire
        {Tri par insertion pour h-ordonner E}
        pour i := h + 1 jusqu'à n faire
            j := i, v := e_i,
            tant que ((j > h) et (v < e_{j−h})) faire
                e_j := e_{j−h}, j := j − h,
            fin faire
            e_j := v,
        fin faire
        h := h/3,
    fin faire
fin procédure
```

On notera que le tri utilisé pour h-ordonner la suite est un tri par insertion. La dernière fois que ce tri est appliqué (avec h qui vaut 1) on exécute donc simplement un tri par insertion. Les passes précédentes ont permis de

mettre en place les éléments de façon à ce que cette ultime exécution du tri par insertion soit très peu couteuse.

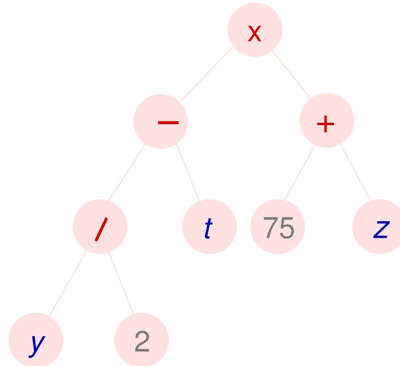
Chapitre 5

Arbres binaires de recherche, arbres lexicographiques

Les arbres et les structures arborescentes sont très utilisés en informatique. D'une part les informations sont souvent hierarchisées, et se présentent donc naturellement sous une forme arborescente, et d'autre part de nombreuses structures de données parmi les plus efficaces sont représentées par des arbres (les tas, les arbres binaires de recherches, les B-arbres, les forêts, . . .).

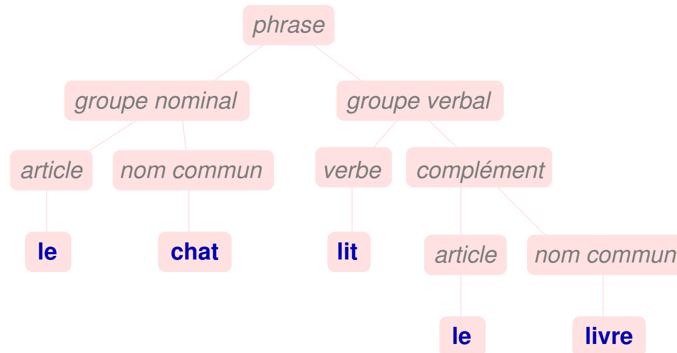
5.1 Quelques exemples

Expressions arithmétiques. On représente souvent des expressions arithmétiques avec des arbres étiquetés par des opérateurs, des constantes et des variables. La structure de l'arbre élimine les ambiguités qui peuvent apparaître en fonction des priorités des opérateurs et qui sont supprimées en utilisant des parenthèses.

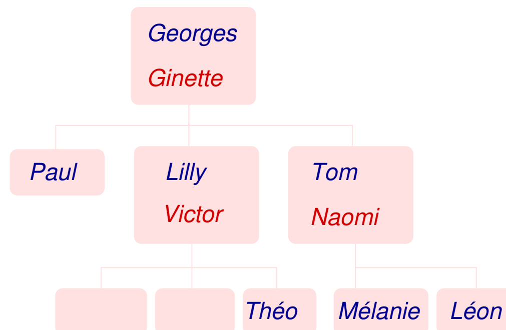


$$(y/2 - t) \times (75 + z)$$

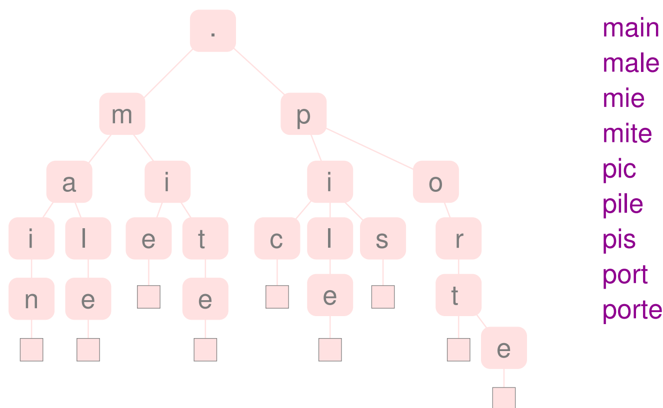
Arbres syntaxiques. Un arbre syntaxique représente l'analyse d'une phrase à partir d'un ensemble de règles qui constitue la grammaire : une phrase est composée d'un groupe nominal suivi d'un groupe verbal, un groupe nominal peut-être constitué d'un article et d'un nom commun,. . .



Arbres généalogiques. Un arbre généalogique (descendant dans le cas présent) représente la descendance d'une personne ou d'un couple. Les noeuds de l'arbre sont étiquetés par les membres de la famille et leurs conjoints. L'arborescence est construite à partir des liens de parenté (les enfants du couple).



Arbre lexicographique. Un arbre lexicographique, ou arbre en parties communes, ou dictionnaire, représente un ensemble de mots. Les préfixes communs à plusieurs mots apparaissent une seule fois dans l'arbre, ce qui se traduit par un gain d'espace mémoire. De plus la recherche d'un mot est assez efficace, puisqu'il suffit de parcourir une branche de l'arbre en partant de la racine, en cherchant à chaque niveau parmi les fils du noeud courant la lettre du mot de rang correspondant.



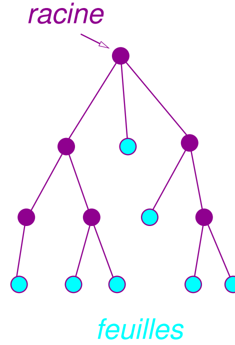
main
male
mie
mite
pic
pile
pis
port
porte

5.2 Définitions et terminologie

Définition. Un arbre est un ensemble organisé de noeuds dans lequel chaque noeud a un *père* et un seul, sauf un noeud que l'on appelle la *racine*.

Si le noeud p est le père du noeud f, nous dirons que f est un *fils* de p, et si le noeud p n'a pas de fils nous dirons que c'est une *feuille*. Chaque noeud porte une *étiquette* ou *valeur* ou *clé*. On a l'habitude, lorsqu'on dessine un

arbre, de le représenter avec la tête en bas, c'est-à-dire que la racine est tout en haut, et les noeuds fils sont représentés en-dessous du noeud père.



Un noeud est défini par son *étiquette* et ses *sous-arbres*. On peut donc représenter un arbre par un n-uplet $\langle e, a_1, \ldots, a_k \rangle$ dans lesquel e est l'étiquette portée par le noeud, et $a_1, \ldots, a_k$ sont ses sous-arbres. Par exemple l'arbre correspondant à l'expression arithmétique $(y/2-t)\times(75+z)$ sera représenté par

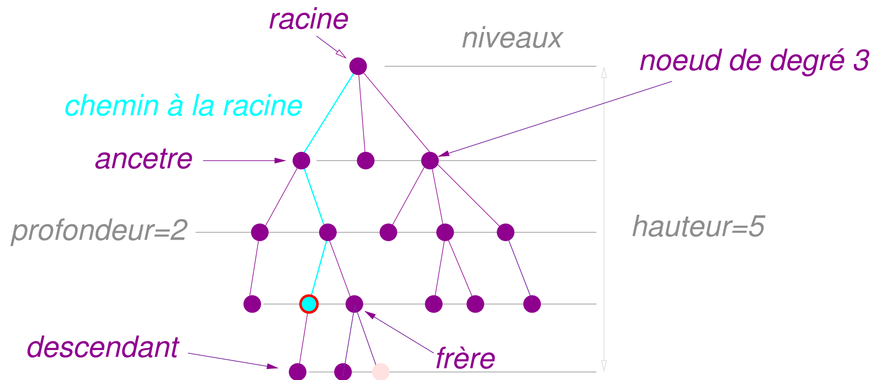
$$\langle \times, \langle -, \langle /, \langle y \rangle, \langle 2 \rangle \rangle, \langle t \rangle \rangle, \langle +, \langle 75 \rangle, \langle z \rangle \rangle \rangle$$

On distingue les *arbres binaires* des arbres généraux. Leur particularité est que les fils sont singularisés : chaque noeud a un fils gauche et un fils droit. L'un comme l'autre peut être un arbre vide, que l'on notera $\langle\rangle$. L'écriture d'un arbre s'en trouve modifiée, puisqu'un noeud a toujours deux fils. En reprenant l'expression arithmétique précédente, l'arbre binaire qui la représente s'écrit

$$\langle \times, \langle -, \langle /, \langle y, \langle\rangle, \langle\rangle \rangle, \langle 2, \langle\rangle, \langle\rangle \rangle \rangle, \langle t, \langle\rangle, \langle\rangle \rangle \rangle, \langle +, \langle 75, \langle\rangle, \langle\rangle \rangle, \langle z, \langle\rangle, \langle\rangle \rangle \rangle \rangle$$

On utilise pour les arbres une terminologie inspirée des liens de parenté :

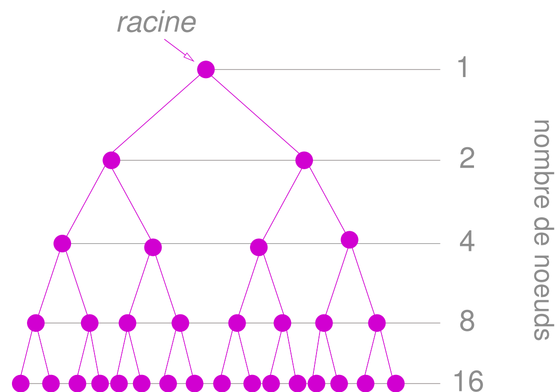
- les *descendants* d'un noeud p sont les noeuds qui apparaissent dans ses sous-arbres,
- un *ancêtre* d'un noeud p est soit son père, soit un ancêtre de son père,
- le chemin qui relie un noeud à la racine est constitué de tous ses ancêtres (c'est-à-dire de son père et des noeuds du chemin qui relie son père à la racine),
- un *frère* d'un noeud p est un fils du père de p, et qui n'est pas p.



Les noeuds d'un arbre se répartissent par *niveaux* : le premier niveau (par convention ce sera le niveau 0) contient la racine seulement, le deuxième niveau contient les deux fils de la racine,..., les noeuds du niveau k sont les fils des noeuds du niveau $k-1$,.... La *hauteur* d'un arbre est le nombre de niveaux de ses noeuds. C'est donc aussi le nombre de noeuds qui jalonnent la branche la plus longue. Attention, la définition de la hauteur varie en fonction des auteurs. Pour certains la hauteur d'un arbre contenant un seul noeud est 0.

Arbres binaires : hauteur, nombre de noeuds et nombre de feuilles.

Un arbre binaire est *complet* si toutes ses branches ont la même longueur et tous ses noeuds qui ne sont pas des feuilles ont deux fils. Soit A un arbre binaire complet. Le nombre de noeuds de A au niveau 0 est 1, le nombre de noeuds au niveau 1 est 2,..., et le nombre de noeuds au niveau p est donc 2^p. En particulier le nombre de feuilles est donc 2^{h-1} si h est le nombre de niveaux.



Le nombre total de noeuds d'un arbre binaire complet de h niveaux est

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

On en déduit que la hauteur $ht(A)$ (le nombre de niveaux) d'un arbre binaire A contenant n noeuds est au moins égale à

$$\lfloor \log_2 n \rfloor + 1$$

Preuve. Si A est arbre de hauteur h et comportant n noeuds, pour tout entier m, on a : $h \leq m-1 \Rightarrow n < 2^{m-1}$. Par contraposée, on a : $n \geq 2^{m-1} \Rightarrow h \geq m$. Le plus grand entier m vérifiant $n \geq 2^{m-1}$ est $\lfloor \log_2 n \rfloor + 1$. On a donc $h \geq \lfloor \log_2 n \rfloor + 1$.

La hauteur minimale $\lfloor \log_2 n \rfloor + 1$ est atteinte lorsque l'arbre est complet. Pour être efficaces, les algorithmes qui utilisent des arbres binaires font en sorte que ceux ci soient équilibrés (voir les tas ou les AVL-arbres par exemple).

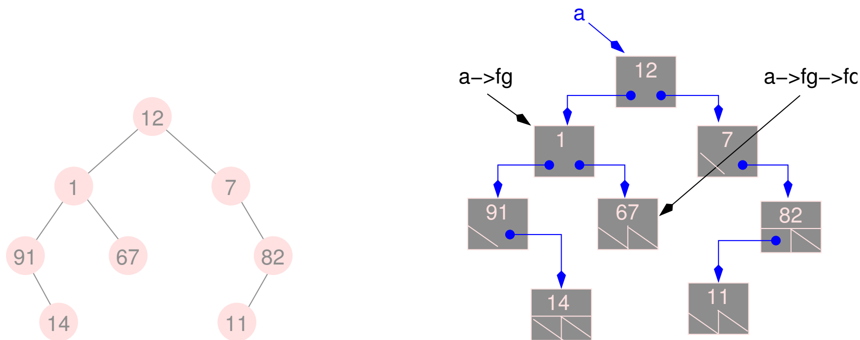
Les arbres en théorie des graphes. En théorie des graphes un arbre est un graphe *connexe* et *sans cycles*, c'est-à-dire qu'entre deux sommets quelconques du graphe il existe un chemin et un seul. On parle dans ce cas d'arbre non planté, puisqu'il n'y a pas de sommet particulier qui joue le rôle de racine. Les sommets sont voisins les uns des autres, il n'y a pas de hiérarchie qui permet de dire qu'un sommet est le père (ou le fils) d'un autre sommet. On démontre qu'un graphe de n sommets qui a cette propriété contient exactement $n-1$ arêtes, et que l'ajout d'une nouvelle arête crée un cycle, tandis que le retrait d'une arête le rend non connexe.

5.3 Arbres binaires

En C on utilise en général des pointeurs pour représenter les liens entre un noeud et ses fils dans un arbre binaire. Il existe plusieurs façons de définir le type `ARBRE`. Nous proposons la définition suivante, qui permet d'imbriquer différents types les uns dans les autres (par exemple si on veut des arbres dont les valeurs sont des listes chainées dont les valeurs sont elles-mêmes des arbres,...).

```
typedef struct noeud * ARBRE;
struct noeud
{
   TYPE_VALEUR val;
   ARBRE fg, fd;
};
```

Un arbre en C est donc désigné par l'adresse de sa racine. Il est facile d'accéder à un noeud quelconque de l'arbre à partir de la racine, en suivant les liens explicites vers le fils droit ou le fils gauche.



La construction d'un arbre consiste, à partir d'un arbre vide, à insérer des nouveaux noeuds. La fonction suivante fait l'allocation d'un noeud et l'initialisation des champs de la structure. Il est recommandé d'utiliser cette fonction chaque fois qu'un nouveau noeud doit être créé.

```
ARBRE CreerNoeud(TYPE_VALEUR v, ARBRE fg, ARBRE fd)
{
   ARBRE p;
   p = malloc(sizeof(struct noeud));
   assert(p != NULL);
   p->val = v;
   p->fg = fg;
   p->fd = fd;
   return p;
}
```

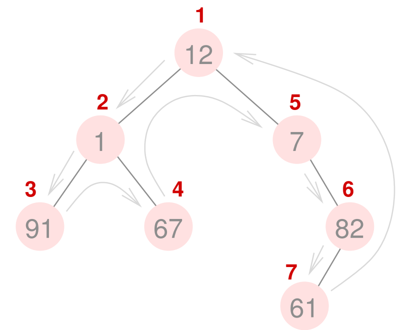
Arbres binaires : parcours. Le principe est simple : pour parcourir l'arbre a, on parcourt récursivement son sous-arbre gauche, puis son sous-arbre droit. Ainsi le sous-arbre gauche sera exploré dans sa totalité avant de commencer l'exploration du sous-arbre droit.

```
void Parcours(ARBRE a)
{
   if (a != NULL)
   {
      /* traitement avant */
      Parcours (a->fg);
      /* traitement entre */
      Parcours (a->fd);
      /* traitement apres */
   }
}
```

Dans ce schéma l'exploration descend d'abord visiter les noeuds les plus à gauche : on parle de *parcours en profondeur d'abord.* On distingue le parcours préfixe, le parcours infixe et le parcours postfixe. La différence entre ces parcours tient uniquement à l'ordre dans lequel sont traités les noeuds du sous-arbre gauche, le noeud courant, et les noeuds du sous-arbre droit.

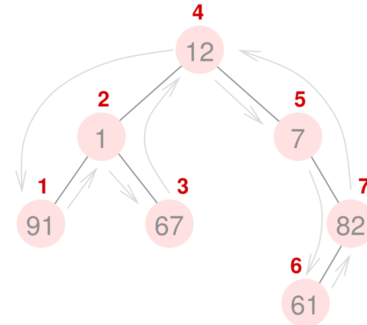
Parcours préfixe : la valeur du noeud courant est traitée *avant* les valeurs figurant dans ses sous-arbres. Si le traitement consiste simplement à afficher la valeur du noeud, le parcours préfixe du l'arbre ci-dessous produirait l'affichage 12 1 91 67 7 82 61.

```
void ParcoursPrefixe(ARBRE a)
{
   if (a != NULL)
   {
      TraiterLaValeur (a->val);
      ParcoursPrefixe (a->fg);
      ParcoursPrefixe (a->fd);
   }
}
```

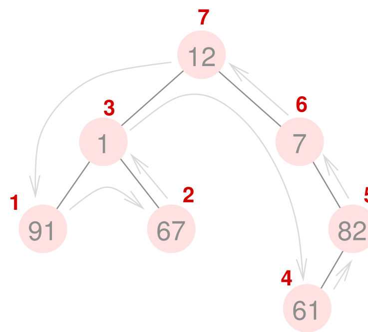


Parcours infixe : la valeur du noeud courant est traitée *après* les valeurs figurant dans son sous-arbre gauche et *avant* les valeurs figurant dans son sous-arbre droit.

```
void ParcoursInfixe(ARBRE a)
{
   if (a != NULL)
   {
      ParcoursInfixe (a->fg);
      TraiterLaValeur (a->val);
      ParcoursInfixe (a->fd);
   }
}
```



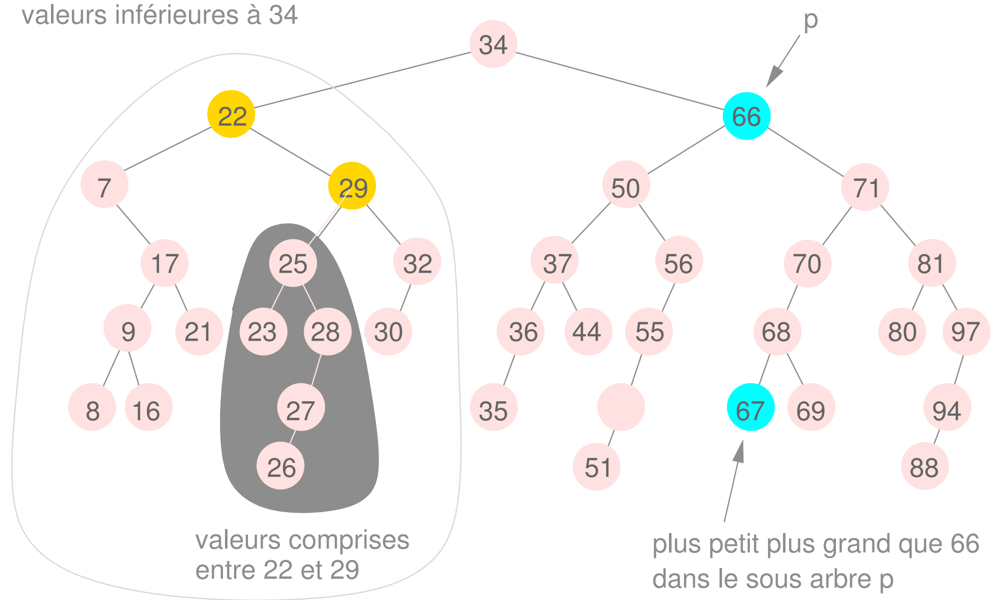
Parcours postfixé : la valeur du noeud courant est traitée *après* les valeurs de ses sous-arbres.



5.4 Arbres binaires de recherche.

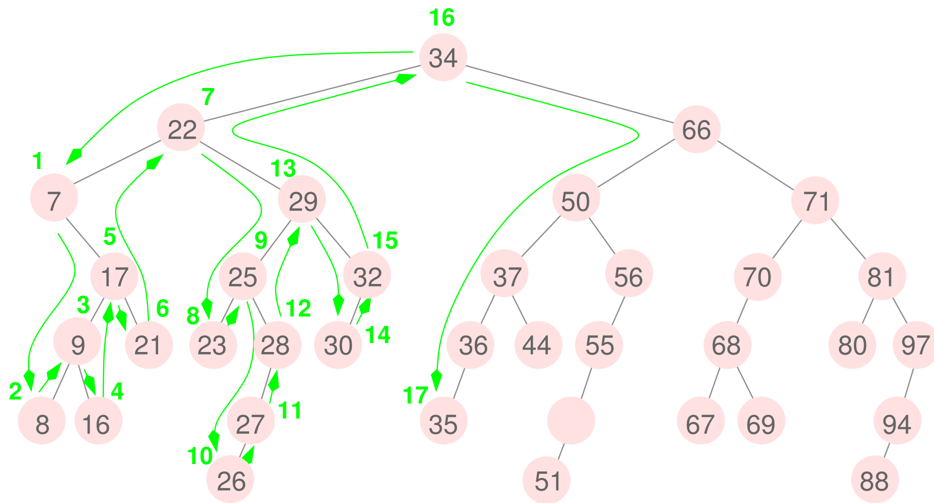
Un arbre binaire de recherche (ou ABR) est une structure de donnée qui permet de représenter un ensemble de valeurs si l'on dispose d'une relation d'ordre sur ces valeurs. Les opérations caractéristiques sur les arbres binaires de recherche sont l'**insertion**, la **suppression**, et la **recherche** d'une valeur. Ces opérations sont peu coûteuses si l'arbre n'est pas trop déséquilibré.

Soit E un ensemble muni d'une relation d'ordre, et a un arbre binaire portant des valeurs de E. a est un *arbre binaire de recherche* si pour tout noeud p de a, la valeur de p est plus grande que les valeurs figurant dans son sous-arbre gauche et plus petite que les valeurs figurant dans son sous-arbre droit.

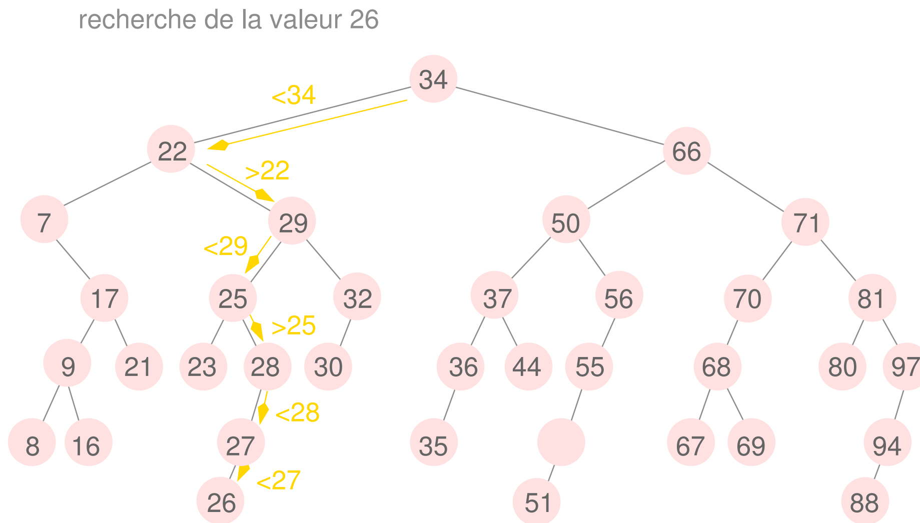


Pour accéder à la clé la plus petite (resp. la plus grande) dans un ABR il suffit de descendre sur le fils gauche (resp. sur le fils droit) autant que possible. Le dernier noeud visité, qui n'a pas de fils gauche (resp. droit), porte la valeur la plus petite (resp. la plus grande) de l'arbre.

Le parcours infixé de l'arbre produit la suite ordonnée des clés

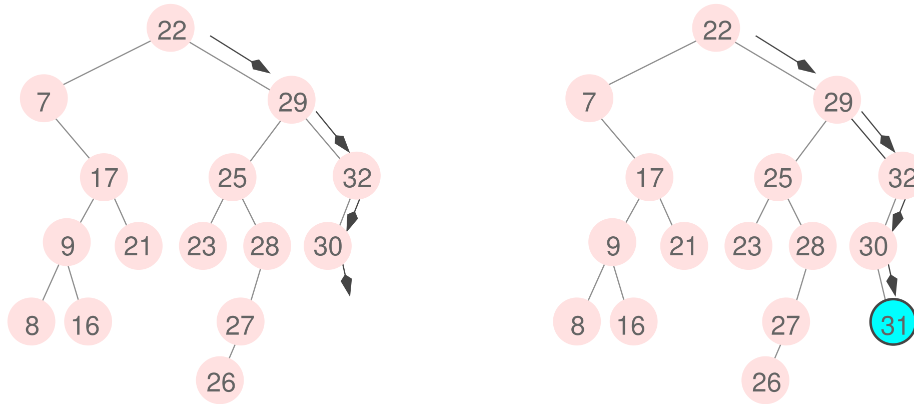


Recherche d'une valeur. La recherche d'une valeur dans un ABR consiste à parcourir une branche en partant de la racine, en descendant chaque fois sur le fils gauche ou sur le fils droit suivant que la clé portée par le noeud est plus grande ou plus petite que la valeur cherchée. La recherche s'arête dès que la valeur est rencontrée ou que l'on a atteint l'extrémité d'une branche (le fils sur lequel il aurait fallu descendre n'existe pas).

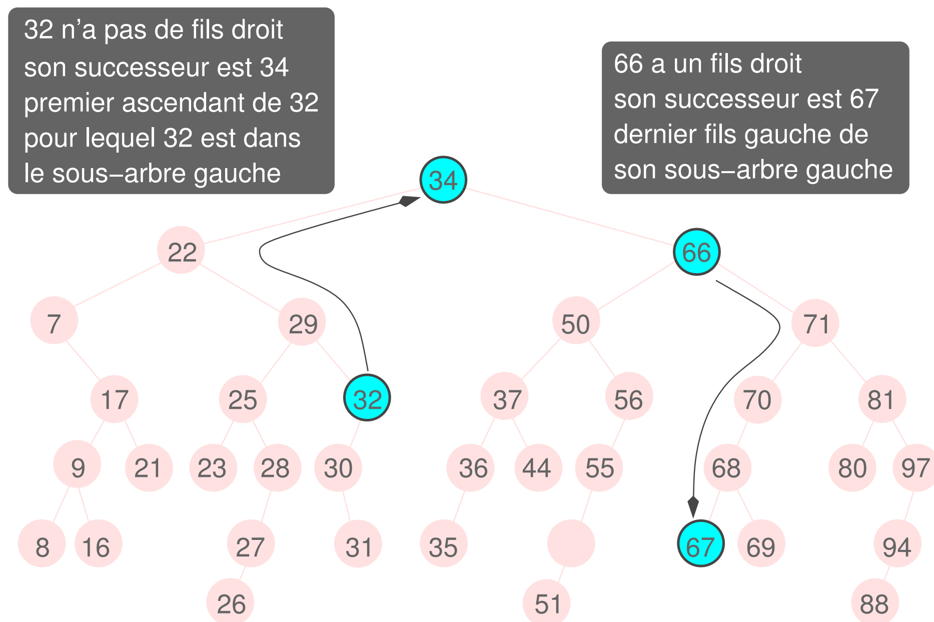



```
ARBRE recherche (TYPE_VALEUR x, ARBRE a)
{
   while ((a != NULL) && (x != a->val))
      if (x < a->val)
         a = a->fg;
      else
         a = a->fd;
   return a;
}
```

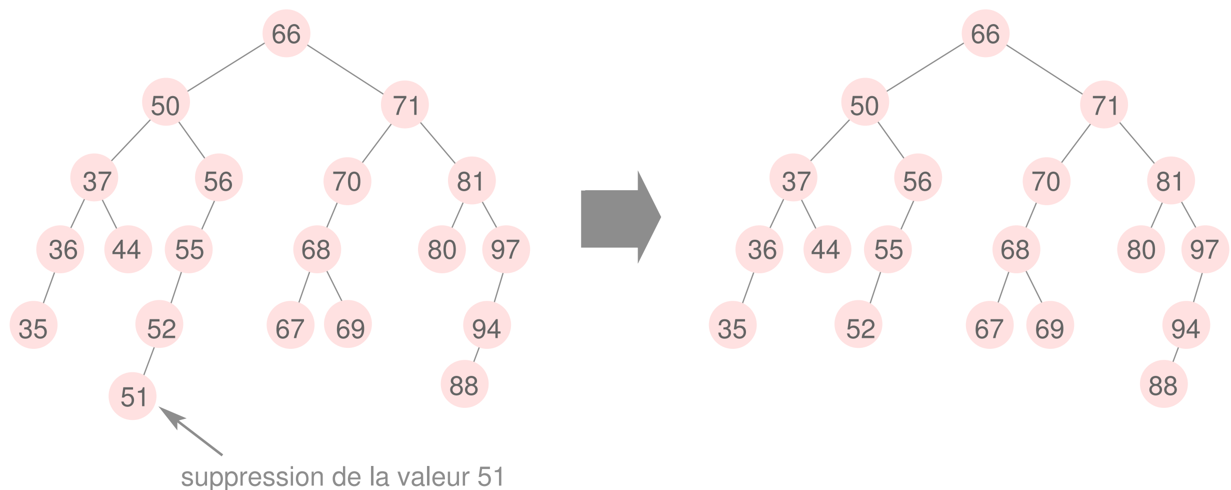
Insertion d'une nouvelle valeur. Le principe est le même que pour la recherche. Un nouveau noeud est créé avec la nouvelle valeur et inséré à l'endroit où la recherche s'est arêtée.



Recherche du successeur d'un noeud. Étant donné un noeud p d'un arbre A, le successeur de p si il existe, est le noeud de A qui porte comme valeur la plus petite des valeurs qui figurent dans A et qui sont plus grandes que la valeur de p. Si p possède un fils droit, son successeur est le noeud le plus à gauche dans son sous-arbre droit (on y accède en descendant sur le fils gauche autant que possible). Si p n'a pas de fils droit alors sont successeur est le premier de ses ascendants tel que p apparaît dans son sous-arbre gauche. Si cet ascendant n'existe pas c'est que p portait la valeur la plus grande dans l'arbre. Remarquez qu'il est nécessaire d'avoir pour chaque noeud un lien vers son père pour mener à bien cette opération.

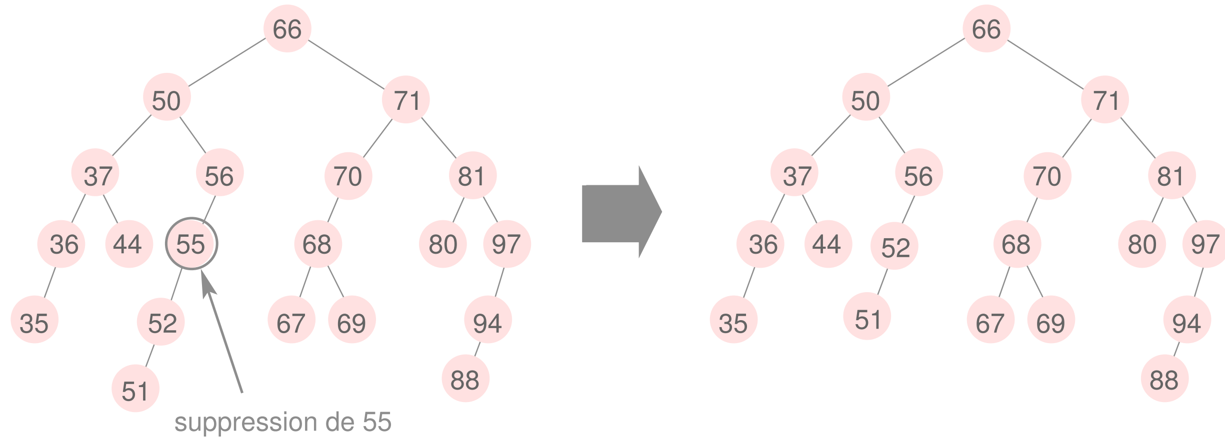


Suppression d'un noeud. L'opération dépend du nombre de fils du noeud à supprimer.

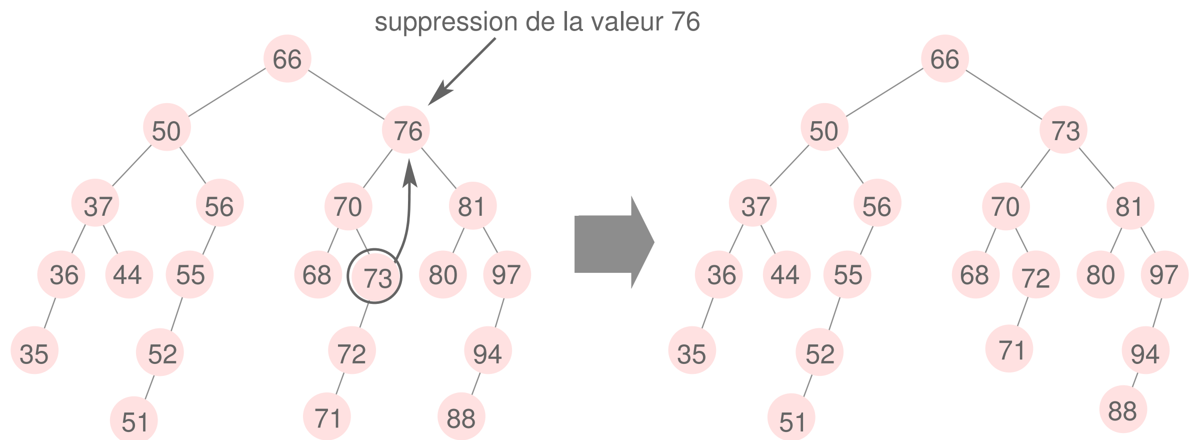


Cas 1 : le noeud à supprimer n'a pas de fils, c'est une feuille. Il suffit de *décrocher* le noeud de l'arbre, c'est-à-dire de l'enlever en modifiant le lien

du père, si il existe, vers ce fils. Si le père n'existe pas l'arbre devient l'arbre vide.



Cas 2 : le noeud à supprimer a un fils et un seul. Le noeud est *décroché* de l'arbre en remplaçant ce fils par son fils unique dans le noeud père, si il existe. Si le père n'existe pas l'arbre est réduit au fils unique du noeud supprimé.

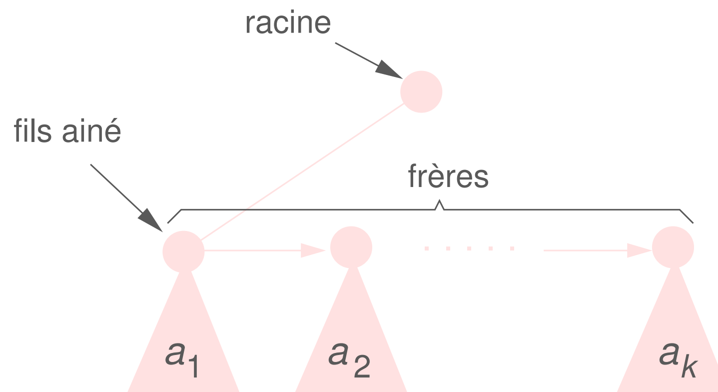


Cas 3 : le noeud à supprimer p a deux fils. On va chercher le noeud q qui porte la plus grande valeur qui figure dans son fils gauche (ou indifféremment le noeud de plus petite valeur qui figure dans son fils droit). Il suffit ensuite

de substituer la valeur de q à la valeur de p et de décrocher le noeud q. Puisque le noeud q a la valeur la plus grande dans le fils gauche, il n'a donc pas de fils droit, et peut être décroché comme on l'a fait dans les cas précédents.

5.5 Arbres généraux ou n-aires

Un arbre n-aire est un arbre dans lequel le nombre de fils d'un noeud n'est pas limité. On les représente avec des pointeurs, en liant les fils d'un même noeud entre eux, comme dans une liste chainée. Ainsi, à partir d'un noeud quelconque, on accède à son fils ainé (la liste de ses fils) et à son frère.



On définit en C les arbres n-aires de la façon suivante :

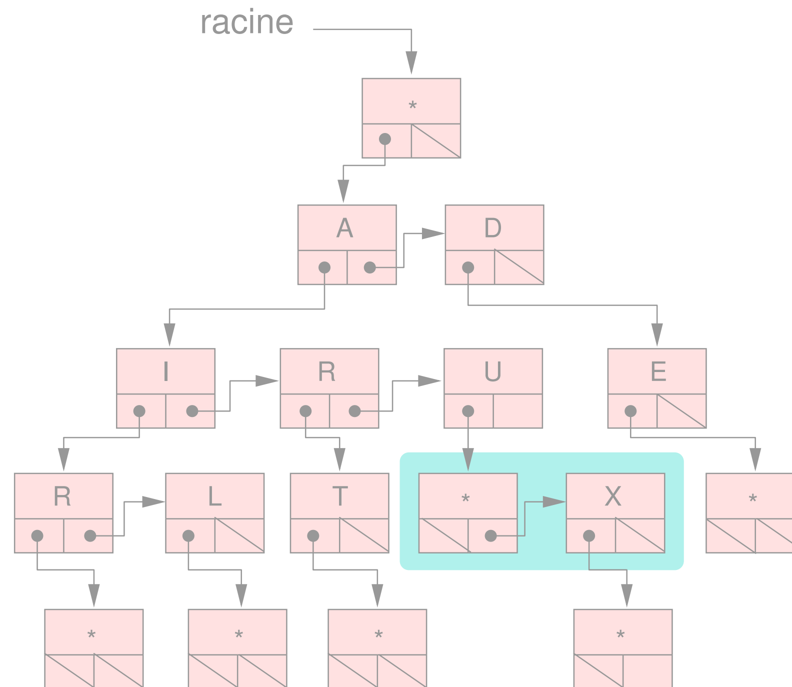
```
typedef struct noeud * ARBRE;

struct noeud
{
   TYPE_VALEUR val;
   ARBRE fa, fr;
};
```

5.6 Exercices

Les arbres lexicographiques, ou arbres en parties communes, ou dictionnaires, sont un exemple d'arbres n-aires. Ils sont utilisés pour représenter

un ensemble de mots, en optimisant l'espace mémoire occupé. Le principe consiste à représenter une seule fois les préfixes communs à plusieurs mots. Pour éliminer toute ambiguité lorsqu'un mot est le préfixe d'un autre, on signale les fins des mots à l'aide d'un caractère particulier (* dans la figure ci-dessous, mais en C le caractère de fin de chaîne `'\0'` est tout à fait adapté).



Question 1. Ecrivez les fonctions d'insertion et de recherche d'un mot dans un arbre lexicographique.

Question 2. Ecrivez une fonction qui affiche tous les mots représentés dans l'arbre.

Question 3. Ecrivez une fonction qui permet de supprimer un mot de l'arbre.

Chapitre 6

Programmation dynamique

6.1 Programmation dynamique et problèmes d'optimisation

La programmation dynamique est une méthodologie générale pour concevoir des algorithmes qui permettent de résoudre efficacement certains problèmes d'optimisation. Un problème d'optimisation admet un grand nombre de solutions. Chaque solution a une certaine valeur, et on veut identifier une solution dont la valeur est optimale (minimale ou maximale). Trouver un plus court chemin pour aller d'un point à un autre dans un réseau de transport est un problème d'optimisation

La conception d'un algorithme de programmation dynamique se décompose en quatre étapes.

1. Caractérisation de la structure d'une solution optimale.
2. Définition récursive de la valeur de la solution optimale.
3. Calcul *ascendant* de la valeur de la solution optimale.
4. Construction de la solution optimale à partir des informations obtenues à l'étape précédente.

L'étape 4 peut être omise si on a seulement besoin de la valeur de la solution optimale et non de la solution elle même.

Les sections suivantes décrivent l'utilisation de la programmation dynamique pour résoudre en temps polynomial trois problèmes d'optimisation : le calcul d'un parcours optimal dans un atelier de montage, le calcul d'une chaîne de mulplications matricielles avec un nombre minimum d'opérations

scalaires, le calcul d'un arbre binaire de recherche optimal. Ensuite, nous mettrons en évidence deux caractéristiques que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable.

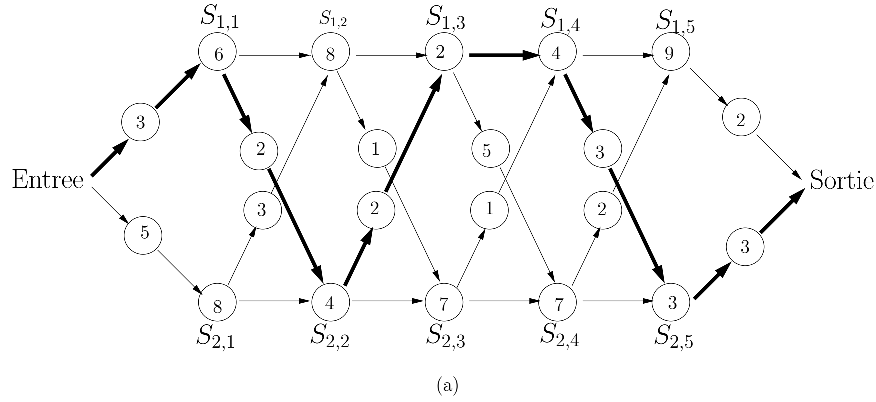
6.2 Ordonnancement optimal d'une chaîne de montage

Un constructeur automobile possède un atelier avec deux chaînes de montage comportant chacune n postes de montages. Chaque véhicule doit passer par les n postes dans l'ordre. Le constructeur cherche à déterminer quels sont les postes à sélectionner sur la chaîne 1 et sur la chaîne 2 pour minimiser le délai de transit d'une voiture à travers l'atelier. Les données du problème d'optimisation qu'il doit résoudre sont les suivantes. Pour $i = 1, 2$ et $j = 1, \ldots, n$, on note $S_{i,j}$ le $j$ème poste de la chaîne i, e_i le *temps d'entrée* d'un véhicule sur la chaîne i, $a_{i,j}$ le *temps de montage* pour le poste j sur la chaîne i, $t_{i,j}$ le *temps de transfert* d'un véhicule de la chaîne i vers l'autre chaîne après le poste $S_{i,j}$ et finallement x_i le temps de sortie d'un véhicule de la chaîne i (voir Figure 6.1a).

Chaque solution de ce problème d'optimisation est définie par le sous-ensemble de postes de la chaîne 1 utilisés (les postes restant sont choisis dans la chaîne 2). Il y a donc 2^n solutions possibles, i.e. le nombre de sous-ensembles d'un ensemble à n éléments. Par conséquent, l'approche naïve consistant à considérer tous les chemins possibles est inefficace. La programmation dynamique permet de résoudre ce problème efficacement.

La *première étape* consiste à identifier des sous-problèmes dont les solutions optimales vont nous permettre de reconstituer une solution optimale du problème initial. Les sous-problèmes à considérer ici consistent à calculer un itinéraire optimal jusqu'au poste $S_{i,j}$ pour $i = 1, 2$ et $j = 1, \ldots, n$. Par exemple, considérons un itinéraire optimal jusqu'au poste $S_{1,j}$. Si $j = 1$, il n'y a qu'un seul chemin possible. Pour $j = 2, \ldots, n$, il y a deux possibilités. Un itinéraire optimal jusqu'à $S_{1,j}$ est ou bien, un itinéraire optimal jusqu'à $S_{1,j-1}$ suivi du poste $S_{1,j}$, ou bien, un itinéraire optimal jusqu'à $S_{2,j-1}$ suivi d'un changement de chaîne et du poste $S_{1,j}$.

La *deuxième étape* consiste à définir la valeur optimale de manière récursive à partir des valeurs des solutions optimales des sous-problèmes. Soit $f_i[j]$ le délai optimal jusqu'à $S_{i,j}$ et f^* le délai optimal total. Pour traverser l'atelier, il faut atteindre ou bien $S_{1,n}$ ou bien $S_{2,n}$ et sortir de l'atelier. Par



(a)

j	1	2	3	4	5
$f_1[j]$	9	17	19	23	32
$f_2[j]$	13	15	22	29	29

$f^* = 32$

j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	2	1

$l^* = 2$

(b)

FIGURE 6.1 – Exemple d'instance du problème d'ordonnacement optimal d'une chaîne de montage : (a) les données du problème et (b) les tables de programmation dynamique.

conséquent, on a

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2).$$

Pour le poste 1 de chaque chaîne, il n'y a qu'un itinéraire possible :

$$f_1[1] = e_1 + a_{1,1}$$
$$f_2[1] = e_2 + a_{2,1}$$

Pour le poste $j = 2, \ldots, n$ de la chaîne 1, il y a deux possibilités :

$$f_1[j] = \min(f_1[j-1] + a_{1,j}\ ,\ f_2[j-1] + t_{2,j-1} + a_{1,j}), \tag{6.1}$$

et symétriquement

$$f_2[j] = \min(f_2[j-1] + a_{2,j}\ ,\ f_1[j-1] + t_{1,j-1} + a_{2,j}). \tag{6.2}$$

Les $f_i[j]$ sont les valeurs des solutions optimales des sous-problèmes. Pour pouvoir reconstruire les solutions optimales elles-mêmes, on définit $l_i[j]$ le numéro de la chaîne (1 ou 2) dont le poste $j-1$ est utilisé par un chemin

optimal jusqu'au poste $S_{i,j}$ pour $j = 2, \ldots, n$, ($l_i[1]$ n'est pas défini car aucun poste ne vient avant le poste 1).

La *troisième étape* consiste à concevoir un algorithme qui calcule en temps polynomial les valeurs $f_i[j]$ des solutions optimales et les informations $l_i[j]$ nécessaires à la construction de ces solutions. L'algorithme suivant fait ce travail en $O(n)$ en utilisant les formules récursives (6.1) et (6.2). Il revient à remplir les tables de la Figure 6.1b de la gauche vers la droite.

Plus-Rapide-Chemin(a, t, e, x, n)
1. $f_1[1] \leftarrow e_1 + a_{1,1}$
2. $f_2[1] \leftarrow e_2 + a_{2,1}$
3. **pour** $j \leftarrow 2$ à n **faire**
4. **si** $f_1[j-1] \leq f_2[j-1] + t_{2,j-1}$
5. **alors** $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
6. $l_1[j] \leftarrow 1$
7. **sinon** $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
8. $l_1[j] \leftarrow 2$
9. **si** $f_2[j-1] \leq f_1[j-1] + t_{1,j-1}$
10. **alors** $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
11. $l_2[j] \leftarrow 2$
12. **sinon** $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
13. $l_2[j] \leftarrow 1$
14. **si** $f_1[n] + x_1 \leq f_2[n] + x_2$
15. **alors** $f^* = f_1[n] + x_1$
16. $l^* = 1$
17. **sinon** $f^* = f_2[n] + x_2$
18. $l^* = 2$

Finalement, la *quatrième et dernière étape* consiste à reconstruire une solution optimale en utilisant les informations sauvegardées à l'étape 3. La procédure suivante affiche les postes utilisés par une solution optimale par ordre décroissant de numéro de poste.

Afficher-Postes(l, n)
1. $i \leftarrow l^*$
2. afficher "chaîne" i, "poste" n
3. **pour** $j \leftarrow n$ **jusqu'à** 2
4. **faire** $i \leftarrow l_i[j]$
5. afficher "chaîne" i, "poste" $j-1$

6.3 Chaîne de multiplications matricielles

On se donne une suite de n matrices $A_1, \ldots, A_n$ et on veut calculer le produit

$$A_1 A_2 \ldots A_n \tag{6.3}$$

Une fois que l'on a parenthésé cette expression de manière à supprimer l'ambiguïté liée à l'ordre dans lequel les multiplications doivent être effectuées, on peut évaluer l'expression (6.3) en utilisant comme routine l'algorithme standard de multiplication de deux matrices.

On dit d'un produit de matrices qu'il est *complètement paranthésé* dans les deux cas suivants :
– c'est une matrice isolée,
– c'est le produit de deux matrices complètement paranthésées.
Le produit de matrices est associatif par conséquent quelquesoit le parenthésage on obtient le même résultat. Par exemple, si la chaîne de matrices est A_1, A_2, A_3, A_4, le produit $A_1A_2A_3A_4$ peut être parenthésé de 5 façons différentes :

$$\begin{array}{l}(A_1(A_2(A_3A_4))),\\ (A_1((A_2A_3)A_4)),\\ ((A_1A_2)(A_3A_4)),\\ ((A_1(A_2A_3))A_4),\\ (((A_1A_2)A_3)A_4).\end{array}$$

La façon dont on paranthèse une telle chaîne de matrices peut avoir une grande importance sur le nombre d'opérations nécessaires pour effectuer le produit.

Si A est une matrice $p \times q$ et B une matrice $q \times r$, la matrice produit est une matrice $q \times r$. La complexité du calcul du produit est dominé par le nombre de multiplications scalaires qui est égal à pqr.

Pour illustrer les différences de coûts obtenus en adoptant des parenthésages différents, considérons un produit de 3 matrices $A_1,\ A_2,\ A_3$ de dimensions respectives 10×100, 100×5 et 5×50. Si on effectue les multiplications dans l'ordre donné par le parenthésage $((A_1A_2)A_3)$, le nombre d'opérations nécessaires est $10 \times 100 \times 5 = 5000$ pour obtenir le produit A_1A_2, qui est une matrice 10×5, plus $10 \times 5 \times 50 = 2500$ pour obtenir le produit $((A_1A_2)A_3)$, soit 7500 opérations en tout. Si on adopte le parenthésage $(A_1(A_2A_3))$, le nombre d'opérations nécessaires est $100 \times 5 \times 50 = 25000$ pour obtenir le pro-

duit A_2A_3, qui est une matrice 100×50, plus $10 \times 100 \times 50 = 50000$ pour obtenir le produit $(A_1(A_2A_3))$, soit 75000 opérations en tout. Par conséquent, calculer le produit en accord avec le premier parenthésage est 10 fois plus rapide.

Notre problème se formule de la façon suivante : étant donné une suite de n matrices $A_1, \ldots, A_n$, avec $p_{i-1} \times p_i$ la dimension de la matrice A_i, pour $i = 1, \ldots, n$, trouver le paranthèsage du produit $A_1A_2 \ldots A_n$ qui minimise le nombre de multiplications scalaires à effectuer.

Remarque : le nombre de parenthésages différents est une fonction exponentielle de n. Par conséquent, la stratégie qui consiste à énumérer tous les parenthésages est exclue pour de grandes valeurs de n.

6.3.1 Structure d'un parenthésage optimal

La première étape dans une approche de type programmation dynamique consiste à identifier la structure des solutions optimales.

Notons $A_{i..j}$ la matrice qui résulte de l'évaluation du produit $A_iA_{i+1} \ldots A_j$. Un parenthésage optimal de $A_1 \ldots A_n$ découpe le produit entre les matrices A_k et A_{k+1} pour un certain k compris entre 1 et $n-1$. C'est à dire que pour un certain k, on calcule d'abord le produit $A_{1..k}$ et $A_{k+1..n}$, ensuite on multiplie ces produits pour obtenir le produit final $A_{1..n}$. Le coût de ce parenthésage est la somme des coûts du calcul de $A_{1..k}$ et $A_{k+1..n}$ et du produit de ces deux matrices.

Remarquons que le parenthésage de $A_{1..k}$ doit être lui-même un parenthésage optimal de $A_1 \ldots A_k$, de même le parenthésage de $A_{k+1..n}$ doit être optimal. Par conséquent, une solution optimal d'un problème de parenthésage contient elle-même des solutions optimales de sous-problèmes de parenthésage. La présence de sous-structures optimales dans une solution optimale est l'une des caractéristiques des problèmes pour lesquels la programmation dynamique est applicable.

6.3.2 Une solution récursive

La seconde étape dans une approche de type programmation dynamique consiste à définir la valeur de la solution en fonction des solutions optimales de sous-problèmes. Dans notre cas, un sous-problème consiste à déterminer le

coût minimal d'un parenthésage de $A_i \ldots A_j$ pour $1 \leq i \leq j \leq n$. Soit $m[i,j]$ le nombre minimum de multiplications scalaires nécessaires pour obtenir le produit $A_{i..j}$. Le nombre minimum de multiplications scalaires nécessaires pour obtenir $A_{1..n}$ sera $m[1,n]$.

On peut calculer $m[i,j]$ récursivement de la façon suivante. Si $i = j$, la chaîne de matrices se réduit à une seule matrice $A_{i..i} = A_i$, aucune opération n'est nécessaire, et donc $m[i,i] = 0$ pour $i = 1, \ldots, n$. Pour calculer $m[i,j]$ quand $i < j$, on se sert de la structure des solutions optimales que nous avons précédemment mise en évidence. Supposons que la solution optimale du sous-problème découpe le produit $A_i \ldots A_j$ entre A_k et A_{k+1} avec $i \leq k < j$. Alors, $m[i,j]$ est égal au nombre minimum de multiplications scalaires pour obtenir $A_{i..k}$ et $A_{k+1..j}$ plus le nombre de multiplications nécessaires pour effectuer le produit matriciel $A_{i..k}A_{k+1..j}$, c'est à dire $p_{i-1}p_kp_j$ multiplications scalaires, on obtient

$$m[i,j] = m[i,k] + m[k,j] + p_{i-1}p_kp_j \ .$$

Cette équation récursive suppose que nous connaissions la valeur de k, ce qui n'est pas le cas. Il y a seulement $j - i$ valeurs possibles pour k : les valeurs $i, i+1, \ldots, j-1$. Puisque la solution optimale correspond à l'une de ces valeurs, il suffit de les essayer toutes et de garder la meilleure. Par conséquent, notre définition récursive pour le coût minimum d'un paranthèsage de $A_i \ldots A_j$ devient

$$m[i,j] = \begin{cases} 0 & \text{si } i = j, \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{si } i < j. \end{cases} \qquad (6.4)$$

Les valeurs $m[i,j]$ donnent les coûts des solutions optimales des sous-problèmes. Pour reconstituer une solution optimale a posteriori, nous allons stocker dans $s[i,j]$ une valeur de k qui minimise $m[i,k] + m[k,j] + p_{i-1}p_kp_j$, i.e. telle que $m[i,j] = m[i,k] + m[k,j] + p_{i-1}p_kp_j$.

6.3.3 Calcul du coût optimal

Au point où nous en sommes, on peut écrire facilement un programme récursif basé sur la relation de récurrence (6.4) pour calculer le coût de la solution optimale. Cependant, cet algorithme n'a pas une meilleure complexité qu'un algorithme énumératif brutal.

En fait, on peut remarquer qu'il existe relativement peu de sous-problèmes : un pour chaque choix de i et j qui satisfasse $1 \leq i \leq j \leq n$, c'est à dire à peu près n^2 (en fait, $n(n+1)/2+n$ exactement). Un algorithme récursif peut rencontrer plusieurs fois chacun de ces sous-problèmes dans l'arbre des appels récursifs. Cette propriété est la deuxième caractéristique des problèmes pour lesquels la programmation dynamique est applicable.

Au lieu de calculer la solution de la récurrence (6.4) récursivement, nous allons effectuer la troisième étape la troisième étape d'une approche de type programmation dynamique en calculant le coût optimal de façon *ascendante*.

L'algorithme suivant remplit la table en commençant par résoudre le problème de parenthésage sur les chaînes de matrices les plus courtes et en continuant par ordre croissant de longueur de chaînes. L'équation de récurrence (6.4) montre que l'on peut calculer le coût optimal pour une chaîne en connaissant les coûts optimaux pour les chaînes de longueurs inférieures.

Ordre-Chaîne-Matrices(p)
1. $n \leftarrow$ longueur(p)
2. **pour** $i \leftarrow 1$ à n
3.   **faire** $m[i,i] \leftarrow 0$
4. **pour** $l \leftarrow 2$ à n
5.   **faire pour** $i \leftarrow 1$ à $n-l+1$
6. $j \leftarrow i+l-1$
7. $m[i,j] \leftarrow \infty$
8.   **pour** $k \leftarrow i$ à $j-1$
9.   **faire** $q \leftarrow m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$
10.   **si** $q < m[i,j]$
11.   **alors** $m[i,j] \leftarrow q$
12. $s[i,j] \leftarrow k$
13. **retourner** m et s

Cet algorithme est en $O(n^3)$. En effet, il contient trois boucles imbriquées dans lesquelles chaque index peut prendre au plus n valeurs. De plus, le traitement est effectué à l'intérieur de ces boucles se fait en temps constant. Par conséquent, cet algorithme est beaucoup plus efficace que la méthode exponentielle qui consiste à examiner chaque parenthésage.

6.3.4 Construction d'une solution optimale

L'algorithme que nous avons décrit précédemment donne le nombre minimum de multiplications scalaires nécessaires pour calculer la chaîne de produits matriciels mais ne montre pas directement comment multiplier ces matrices en utilisant ce nombre minimum de multiplications. La dernière étape dans notre approche de type programmation dynamique consiste à reconstruire une solution optimale à partir des informations que nous avons sauvegardées à l'étape précédente.

Nous allons utiliser une table $s[\ ,\ ]$ pour déterminer la meilleure façon de multiplier les matrices. Chaque entrée $s[i,j]$ de cette table contient la valeur k telle qu'un parenthésage optimal sépare le produit $A_i A_{i+1} \dots A_j$ entre A_k et A_{k+1}. Par conséquent, nous savons que la dernière multiplication matricielle dans la solution optimale que nous voulons reconstruire est $A_{1..s[1,n]} A_{s[1,n]+1..n}$. Les multiplications précédentes peuvent être reconstituées récursivement de la même façon. La procédure suivante affiche le parenthésage optimal du produit matriciel $A_{i..j}$ étant donnée la table $s[\ ,\ ]$ calculée à l'étape précédente et les indices i et j. Pour calculer le parenthésage complet, le premier appel de cette procédure sera AFFICHAGE-PARENTHÉSAGE-OPTIMAL$(s, 1, n)$.

AFFICHAGE-PARENTHÉSAGE-OPTIMAL(s, i, j)
1. **si** $i = j$
2. **alors** Afficher "A_i"
3. **sinon** Afficher "("
4. AFFICHAGE-PARENTHÉSAGE-OPTIMAL$(s, i, s[i,j])$
5. AFFICHAGE-PARENTHÉSAGE-OPTIMAL$(s, s[i,j]+1, j)$
6. Afficher ")"

6.4 Arbre binaire de recherche optimal

Le problème d'optimisation que nous allons considérer dans cette section consiste étant donné un ensemble de clefs $\{a_1, a_2, \dots, a_n\}$ et $p_1, p_2, \dots, p_n$ les fréquences de recherche de ces clefs, à calculer un arbre binaire de recherche qui permettent de minimiser le temps de recherche de ces clefs. En remarquant que le temps nécessaire pour rechercher une clef est proportionnel à sa profondeur dans l'arbre binaire de recherche, on déduit que l'arbre binaire de recherche que nous souhaitons identifier doit minimiser la quantité

suivante

$$\text{coût}(T) = \sum_{i=1}^{n} p_i \times (\text{prof}_T(a_i) + 1)$$

où $prof_T(a_i)$ est la profondeur de la clef a_i dans l'arbre T.

clefs	a_1	a_2	a_3	a_4	a_5
fréquences	20	40	12	16	12

coût$(T_1) = 60 + 36 + 80 + 32 + 12 = 220$ coût$(T_2) = 36 + 36 + 40 + 32 + 40 = 184$

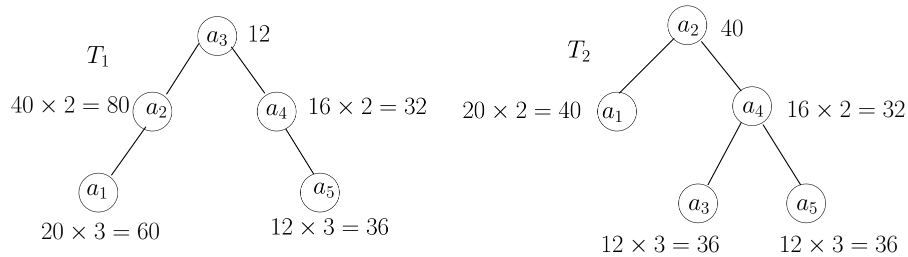


Figure 6.2 – Un exemple d'instance du problème de l'arbre binaire de recherche optimal avec deux arbres et leurs coûts respectifs.

Pour commencer, nous allons mettre en évidence la présence de sous-structures optimales dans un arbre binaire de recherche (ABR) optimal. Pour cela, nous allons introduire les notations suivantes :

- $T_{i,j}$ un ABR optimal pour les clefs $a_{i+1}, \ldots, a_j$;
- $w_{i,j} = p_{i+1} + \ldots + p_j$ la somme des fréquences des clefs de l'arbre $T_{i,j}$;
- $r_{i,j}$ la racine de l'arbre $T_{i,j}$;
- $c_{i,j}$ le coût de l'arbre $T_{i,j}$;
- $T_{i,i}$ l'arbre vide ($c_{i,i} = 0$).

Considérons un ABR optimal $T_{i,j}$ et notons k l'entier compris entre $i+1$ et j tel que la racine $r_{i,j}$ de l'arbre $T_{i,j}$ soit a_k. Le sous-arbre gauche de la racine a_k est constitué des clefs $a_{i+1}, \ldots, a_{k-1}$. De plus, ce sous-arbre T doit nécessairement être un ABR optimal sur cet ensemble de clefs car s'il existait un meilleur ABR T' sur cet ensemble de clefs alors en remplaçant T par T' dans $T_{i,j}$ on obtiendrait un arbre meilleur que $T_{i,j}$, ce qui contredirait l'optimalité de $T_{i,j}$. Le même raisonnement s'applique au sous-arbre droit et montre que l'arbre optimal $T_{i,j}$ est constitué d'une racine a_k dont le fils gauche est la racine d'un ABR optimal $T_{i,k-1}$ et le fils droit est la racine d'un ABR optimal $T_{k,j}$ (voir Figure 6.4).

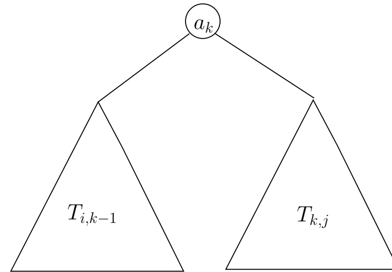


FIGURE 6.3 – Sous-structures optimales dans un ABR optimal

Voyons maintenant comment donner une définition récursive du coût d'un ABR optimal. Si a_k est la racine de T_{ij}, on a :

$$\begin{aligned} c_{i,j} &= (c_{i,k-1} + w_{i,k-1}) + p_k + (c_{k,j} + w_{k,j}) \\ &= (w_{i,k-1} + p_k + w_{k,j}) + c_{i,k-1} + c_{k,j} \\ &= w_{i,j} + c_{i,k-1} + c_{k,j} \end{aligned}$$

La première ligne s'explique par le fait que lorsque l'on place l'arbre $T_{i,k-1}$ sous la racine a_k, la profondeur de chacun de ces noeuds augmente de un, donc globalement le coût $c_{i,k-1}$ de ce sous-arbre augmente de la somme des fréquences des clefs qu'il contient, c'est à dire $w_{i,k-1}$. Idem pour la contribution du sous-arbre droit $T_{k,j}$ qui augmente de $w_{k,j}$. Finalement, il reste à compter la contribution de a_k qui est égale à p_k. La deuxième ligne est juste une réécriture de la première et la troisième utilise simplement la définition de $w_{i,j}$.

Pour finir, on remarque que la valeur de k à utiliser est celle qui minimise le coût $c_{i,k-1} + c_{k,j}$, ce qui donne la définition récursive suivante :

$$c_{i,j} = w_{i,j} + \min_{k \in \{i+1,\ldots,j\}} (c_{i,k-1} + c_{k,j})$$

L'algorithme suivant consiste à résoudre les sous-problèmes sur des sous-ensembles de clefs de longueurs l croissantes, en utilisant la formule récursive ci-dessus :

ABR-OPTIMAL(p, n)
1. Pour $i \leftarrow 0$ à n faire
2. $w_{i,i} \leftarrow 0$
3. $c_{i,i} \leftarrow 0$
4. Pour $l \leftarrow 1$ à n faire

5. Pour $i \leftarrow 0$ à $n - l$ faire
6. $j \leftarrow i + l$
7. $w_{i,j} \leftarrow w_{i,j-1} + p_j$
8. Soit m la valeur de k telle que $c_{i,k-1} + c_{k,j}$ soit minimum
9. $c_{i,j} = w_{i,j} + c_{i,m-1} + c_{m,j}$
10. $r_{i,j} \leftarrow m$

L'algorithme précédent permet d'obtenir le coût d'un ABR optimal et sauvegarde les informations nécessaires pour le contruire grâce à l'algorithme suivant.

CONST-ABR-OPT(i, j)
1. $p \leftarrow$ **CréerNoeud**$(r_{i,j}$, NULL, NULL)
2. Si $i < r_{i,j} - 1$ alors
3. p→fils-gauche = **Const-ABR-opt**$(i, r_{i,j} - 1)$
4. Si $r_{i,j} < j$ alors
5. p→fils-droit = **Const-ABR-opt**$(r_{i,j}, j)$

6.5 Applicabilité de la programmation dynamique

Dans cette section, nous présentons deux caractéristiques que doit avoir un problème d'optimisation pour que la programmation dynamique soit applicable : la présence de sous-structures optimales et le recouvrement des sous-problèmes.

6.5.1 Sous-structures optimales

La première étape dans une approche de type programmation dynamique consiste à identifier la structure des solutions optimales. On dit que le problème présente une *sous-structure optimale* si une solution optimale contient des solutions optimales de sous-problèmes.

C'était le cas dans les trois problèmes précédents. Par exemple, chaque parenthésage optimal de $A_1 \ldots A_n$ était obtenu comme le produit du parenthésage optimal de $A_1 \ldots A_k$ et de $A_{k+1} \ldots A_n$ pour un certain k et que chaque ABR optimal peut être obtenu en attachant à une racine bien choisie deux ABR optimaux pour des sous-problèmes.

6.5.2 Recouvrement des sous-problèmes

Pour que la programmation dynamique soit applicable, il faut également que l'espace des sous-problèmes soit suffisament "petit". Dans le sens où un algorithme récursif qui résoudrait le problème rencontrerait les mêmes sous-problèmes plusieurs fois, au lieu de générer toujours de nouveaux sous-problèmes. Pour que la programmation dynamique permette de concevoir un algorithme polynomial, il faut bien sûr que le nombre de sous-problèmes à considérer soit polynomial.

Chapitre 7

Graphes

Chapitre 8

Annexes mathématiques

8.1 Les notations de Landau pour la comparaison asymptotique des fonctions

On s'intéresse le plus souvent à la manière dont la complexité $T(n)$ d'un algorithme croît en fonction de n, plutôt qu'aux valeurs de T pour des valeurs particulières de n. Autrement dit, on s'intéresse au comportement asymptotique de la fonction T au voisinage de l'infini et l'on souhaite le comparer aux comportements de fonctions de références : n, $n \log n$, n^2, e^n, etc. Les notations de Landau permettent d'exprimer ces comparaisons de manière simple.

Soit $f, g : \mathbb{N} \to \mathbb{R}$ où g joue le rôle de la fonction de référence et f celui de la fonction qu'on souhaite lui comparer.

On écrit

$$f(n) = O(g(n)) \text{ si } \exists C, n_0\ \forall n \geq n_0\ f(n) \leq Cg(n).$$

La fonction g majore f, à partir d'une certaine valeur, et à une constante multiplicative près.

On écrit

$$f(n) = \Omega(g(n)) \text{ si } \exists C, n_0\ \forall n \geq n_0\ f(n) \geq Cg(n).$$

La fonction g minore f, à partir d'une certaine valeur, et à une constante multiplicative près.

On écrit

$$f(n) = \Theta(g(n)) \text{ si } \exists C_1, C_2, n_0\ \forall n \geq n_0\ C_1 g(n) \leq f(n) \geq C_2 g(n).$$

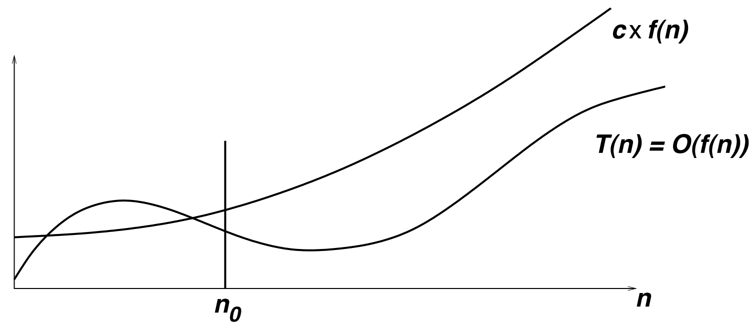


Figure 8.1 – La notation $O(f(n))$.

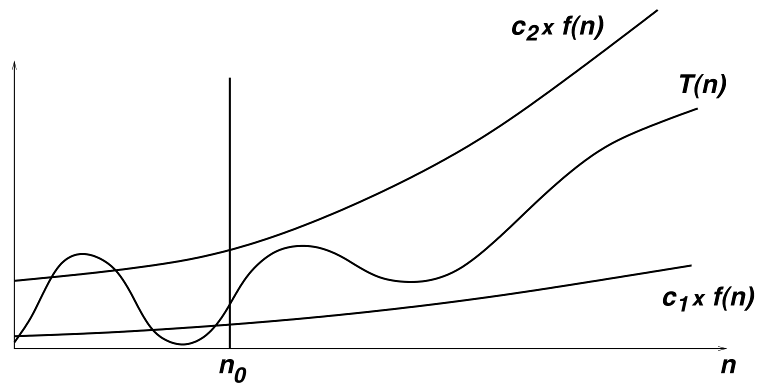


Figure 8.2 – La notation $\Theta(f(n))$.

La fonction g décrit la croissance de f, à partir d'une certaine valeur, et à une constante multiplicative près. On peut remarquer que

$$f(n) = \Theta(g(n)) \text{ ssi } f(n) = O(g(n)) \text{ et } f(n) = \Omega(g(n)).$$

Enfin, on écrit

$$f(n) = o(g(n)) \text{ si } \forall\alpha > 0\ \exists n_\alpha\ \forall n \geq n_\alpha\ f(n) \leq \alpha g(n).$$

Si g ne prend que des valeurs non nulles, cela signifie que $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Exemple 4

1. $3n^2 + 5n - 1 = O(n^2)$; en effet, $3n^2 + 5n - 1 = 3n^2(1 + \frac{5}{3n} - \frac{1}{3n^2})$ et $\frac{5}{3n} - \frac{1}{3n^2} \leq 1$ dès que $n \geq 2$; on peut donc prendre $C = 6$ et $N = 2$.
2. $n + \sin n = \Omega(n)$;
3. $n(2\log n + \log\log n - 1) = \Theta(n \log n)$;
4. $n \log n = o(n^2)$.

8.2 Comportement asymptotique des fonctions définies par récurrence

La complexité des algorithmes récursifs est naturellement décrite par des équations de récurrence. Par exemple, l'algorithme de recherche dichotomique conduit à l'équation

$$T(n) = T(\lceil \frac{n}{2} \rceil) + c,$$

l'algorithme de tri par fusion conduit à l'équation

$$T(n) = 2 \times T(\lceil \frac{n}{2} \rceil) + n$$

Théorème 1 *Soit $T : \mathbb{N} \mapsto \mathbb{R}$ défini par la formule*

$$T(n) = aT(i(n/b)) + f(n)$$

où $a \geq 1$, $b > 1$ et $f : \mathbb{N} \mapsto \mathbb{R}$ et où $i(n/b)$ est égal à $\lceil n/b \rceil$ où à $\lfloor n/b \rfloor$.

1. *S'il existe $\epsilon > 0$ pour lequel $f(n) = O(n^{\log_b a - \epsilon})$, alors $T(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = O(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log_2 n)$.*
3. *S'il existe $\epsilon > 0$ pour lequel $f(n) = \Omega(n^{\log_b a + \epsilon})$ et s'il existe $c < 1$ tel que $af(n) \leq cf(n)$ dès que n est suffisamment grand, alors $T(n) = \Theta(f(n))$.*

Ce théorème est prouvé dans le livre de référence.