
ALGORITHMIQUE ET STRUCTURES
DE DONNÉES AVANCÉES

2025-2026

RÉFÉRENCES : INTRODUCTION À L'ALGORITHMIQUE, THOMAS CORMEN, CHARLES LEISERSON, RONALD RIVEST, CLIFFORD STEIN, DUNOD, 2IEME EDITION, 1146 PAGES, 2002.

Table des matières

1 Rappel : méthodes de conception et calcul asymptotique	10
I Conception des algorithmes	10
1. Le paradigme "Divide and Conquer"	10
2. Exemple : tri fusion	11
II Les notations asymptotiques	14
III Echelle de complexité	15
IV Les récurrences	17
V Travaux dirigés	19
VI Travaux pratiques	30
2 Les arbres	34
I Arbres binaires de recherche	34
1. Parcours préfixe et postfixe	36
2. Recherche	36
3. Minimum et maximum	38
4. Successeur et prédécesseur	38
5. Insertion et Suppression	39
II Tas binaire	42
1. Définitions	42
2. Conservation de la structure de tas	44
3. Construction du tas	46
4. Applications de Tas	48
III Travaux dirigés	56
IV Travaux pratiques	71
V Projet I	72
3 Les graphes	74
I Définitions	74
II Structures de Données	75
1. Matrice d'adjacence	75
2. Liste d'adjacence	76

III	Algorithmes de parcours	76
1.	Parcours en largeur	77
2.	Parcours en profondeur	79
IV	Tri topologique	84
V	Composantes fortement connexes	86
VI	Arbres couvrants de poids minimum	88
1.	Algorithme générique	90
2.	Algorithme de Kruskal	90
3.	Algorithme de Prim	92
VII	Plus courts chemins à origine unique	96
1.	Algorithme de Dijkstra	97
2.	Algorithme de Bellman-Ford	99
VIII	Plus courts chemins pour tout couple de sommets	101
IX	Coloration de graphe	103
X	Travaux dirigés	105
XI	Travaux pratiques	110

Table des figures

1.1	Exemple de fonctionnement du tri par fusion	13
1.2	Les notation O , Ω , et Θ	15
1.3	La partie (a) montre $T(n)$, progressivement développé dans les parties (b)–(d) pour former l’arbre récursif. L’arbre entièrement développé, en partie (d), a une hauteur de $\log_4 n$ (il comprend $\log_4 n + 1$ niveaux)	31
1.4	L’arbre des appels récursifs	32
2.1	Un arbre binaire de recherche de 6 noeuds et de hauteur 2. (b) Un arbre binaire de recherche moins efficace de hauteur 4, contenant les mêmes clés.	35
2.2	Un arbre binaire de recherche.	35
2.3	La procédure PARCOURS-INFIXE(racine[T])	36
2.4	La procédure rechercher.	37
2.5	Rechercher la clé 13 dans l’arbre on suit le chemin 15-6-7-13 en partant de la racine.	37
2.6	Rechercher le Minimum	38
2.7	Le pseudo-code de ARBRE-SUCCESSEUR	38
2.8	Un arbre binaire	39
2.9	Le pseudo-code de <i>ARBRE-INSÉRER</i>	40
2.10	Insertion d’un élément de clé 13	40
2.11	La procédure permettant de supprimer un noeud donné z	41
2.12	Le pseudo code de la procédure Suppression	42
2.13	Un tas-max vu comme (a) un arbre binaire et (b) un tableau .	43
2.14	Un tas-max	44
2.15	Un tas-min	44
2.16	Hauteur d’un tas	44
2.17	Le programme <i>ENTASSER-MAX</i>	45
2.18	L’action <i>ENTASSER-MAX(A,2)</i>	45
2.19	La procédure <i>CONSTRUIRE-TAS-MAX</i>	46
2.20	<i>CONSTRUIRE-TAS-MAX</i> sur un tableau de 10 éléments . .	47

2.21	Nombre d'opérations d'échange par niveau	47
2.22	La procédure Tri par Tas	48
2.23	Exemple de l'action <i>TRI-PAR-TAS</i>	49
2.24	La procédure MAXIMUM-TAS	50
2.25	La procédure EXTRAIRE-MAX-TAS	50
2.26	Exemple de l'action <i>EXTRAIRE-MAX-TAS</i>	51
2.27	La procédure <i>AUGMENTER-CLÉ-TAS</i>	52
2.28	Exemple de l'action <i>AUGMENTER-CLÉ-TAS</i>	52
2.29	La procédure <i>INSÉRER-TAS-MAX</i>	53
2.30	Exemple de l'action <i>INSÉRER-TAS-MAX</i>	53
2.31	Exemple de construction de l'arbre de Huffman	55
2.32	La procédure de construction de l'arbre	56
2.33	Exemple 1 :"axabataaxbcctaabbxxaazazaa"	57
2.34	Exemple 2	58
3.1	(a) : graphe orienté ; (b) : le même graphe non orienté	75
3.2	Représentation d'un graphe non orienté	76
3.3	Représentation d'un graphe orienté	76
3.4	L'algorithme PL	79
3.5	Un graphe non orienté	80
3.6	L'action de PL sur un graphe non orienté	81
3.7	L'algorithme PP	82
3.8	Un graphe orienté	83
3.9	L'action de PP sur un graphe orienté	83
3.10	(a) Le résultat d'un PP. (b) Les intervalles des dates de dé- couverte et de fin de traitement	84
3.11	Un graphe orienté	85
3.12	Exemple de tri topologique	85
3.13	L'algorithme de tri topologique	86
3.14	Un graphe orienté sans circuit	86
3.15	L'algorithme de tri topologique	87
3.16	L'algorithme de recherche des composantes fortement connexes	88
3.17	Exemple d'un graphe	89
3.18	L'arbre de recouvrement minimum	89
3.19	un exemple d'un arbre de recouvrement minimum	89
3.20	Pseudo-code de l'algorithme générique	90
3.21	Pseudo-code de l'algorithme Kruskal	92
3.22	L'exécution de l'algorithme de Kruskal	93
3.23	Le pseudo code de l'algorithme de Prim	94
3.24	L'exécution de l'algorithme de Prim	95

3.25 Un graphe orienté pondéré. Les arcs en gris forment une arborescence de plus courts chemins de racine s	97
3.26 L'algorithme Dijkstra	98
3.27 La procédure Source-Unique-Initialisation(G, s)	98
3.28 La procédure Relacher	98
3.29 relâchement d'un arc (u, v) de poids $w(u, v) = 2$	99
3.30 L'exécution de l'algorithme de Dijkstra	99
3.31 L'algorithme de Bellman-Ford	100
3.32 L'exécution de l'algorithme de Bellman-Ford	101
3.33 Exemple d'un graphe avec des sommets intermédiaires	102
3.34 L'algorithme de Floyd-Warshall	103
3.35 Exemple	103
3.36 Exemple	104

Liste des tableaux

Objectifs et planification

Objectifs

Cet enseignement à pour but de présenter les structures de données avancées et leurs algorithmes de base, telles que les arbres binaires de recherche, les tas binaires, les files de priorité, et les graphes. Il s'agit en particulier,

- Etudier les structures de données avancées et leur rôle dans la conception des algorithmes.
 - Etudier les techniques algorithmiques pour la résolution des problèmes.
- Les étudiants vont apprendre les techniques avancées de conception et d'analyse d'algorithmes complexes, en particulier :
- comprendre les différentes structures de données avancées (SDA)
 - concevoir des algorithmes pour la résolution des problèmes complexes en utilisant les SDA
 - consolider la programmation en language C pour le développement des structures de données avancées

Chapitres

Ce document est structuré de la manière suivante :

- Rappel : méthodes de conception et calcul asymptotique
 - mesures de complexité et notation asymptotique
 - analyse des algorithmes itératifs et récursifs
- Structures de données élémentaires
 - arbres binaires
 - tas binaires
 - files de priorité
 - codage de Huffman
 - tri par Tas

- Algorithmes pour les graphes
 - représentation des graphes
 - parcours en largeur et en profondeur
 - tri topologique
 - composantes fortement connexes
 - arbre couvrant minimum
 - algorithme de Bellman-Ford
 - algorithme de Dijkstra

Planification

- Cours : 22 h
- TP : 22 h

Evaluation

L'évaluation comporte des épreuves individuelles (contrôle continu) (CC note sur 20) et une note de travaux pratiques évalués par les rapports et/ou un contrôle (TP sur 20) attribuée par l'enseignant au vu du travail réalisé par l'étudiant pendant les séances et une dernière note celle de l'examen final (EF note sur 20). Note finale = 20% Note Contrôle(s) + 30% Note Examen de TP/Projet + 50% Note Examen Final

Intervenants

- Mohamed BAKHOUYA : mohamed.bakhouya@uir.ac.ma
- Abderrahmane ELAMRANI : abderrahmane.elamrani@uir.ac.ma
- Abderrahmane AQACHTOUL : abderrahmane.aqachtoul@uir.ac.ma
- Ikram EN-NAJJARI : ikram.en-najjari@uir.ac.ma
- Saad NOUFEL : saad.noufel@uir.ac.ma

Chapitre 1

Rappel : méthodes de conception et calcul asymptotique

I Conception des algorithmes

Il existe de nombreuses façons de concevoir un algorithme :

- Le tri par insertion utilise une approche incrémentale : après avoir trié le sous-tableau $A[1..j - 1]$, on insère l'élément $A[j]$ au bon emplacement pour produire le sous-tableau trié $A[1..j]$.
- Une autre approche de conception, baptisée « diviser pour régner » est utilisée pour créer un algorithme de tri.
- L'un des avantages des algorithmes diviser-pour-régner est que leurs temps d'exécution sont souvent faciles à déterminer, via des techniques que nous allons voir plus tard.
- On considère généralement qu'un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur.
- Pour des entrées de taille assez grande, un algorithme en $O(n^2)$, par exemple, s'exécute plus rapidement, dans le cas le plus défavorable, qu'un algorithme en $O(n^3)$.

1. Le paradigme "Divide and Conquer"

C'est une classe de conception des algorithmes qui procède de la manière suivante. Séparer le problème en plusieurs sous-problèmes semblables au pro-

blème initial mais de taille moindre, résolvent les sous-problèmes de façon récursive, puis combinent toutes les solutions pour produire la solution du problème original :

1. On divise l'entrée en plusieurs entrées plus petites.
2. On résoud les sous-problèmes séparément d'une manière récursive.
3. On combine les sous-résultats pour former le résultat global.

C'est une méthode simple et performante. Pour analyser le coût d'un algorithme basé sur ce paradigme, on a recours aux équations de récurrence, c'est à dire le coût d'une instance est déterminé en fonction des instances plus petites. En d'autres termes, lorsqu'un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une équation de récurrence, ou récurrence. On peut alors se servir d'outils mathématiques pour résoudre la récurrence et trouver des bornes pour les performances de l'algorithme.

Supposons que l'on divise le problème en a sous-problèmes, la taille de chacun étant $1/b$ de la taille du problème initial. (Pour le tri par fusion, tant a que b valent 2, mais nous verrons beaucoup d'algorithmes diviser-pour-régner dans lesquels $a \neq b$.)

Si l'on prend un temps $D(n)$ pour diviser le problème en sous-problèmes et un temps $C(n)$ pour construire la solution finale à partir des solutions aux sous-problèmes, on obtient la récurrence :

$$\begin{cases} O(1) & \text{si } c \geq n \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{sinon} \end{cases}$$

2. Exemple : tri fusion

En utilisant le paradigme D&C :

1. On divise la liste en deux sous-listes.
2. On compte le nombre d'inversions dans les deux sous-listes séparément.
3. On compte les inversions lorsque les éléments des paires (a_i, a_j) appartiennent aux deux sous-listes.

```
Compter_Par_D&C(L)
  Si L contient 1 seul élément Alors
    Arrêter
  Sinon
```

```

Diviser L en deux sous-listes A et B
A contient les premiers [n/2] éléments et B les suivants
r_A <- Compter_Par_D&C(A)
r_B <- Compter_Par_D&C(B)
r <- Combiner_Et_Compter(A,B)
Finsi
Retourner r
Fin

Combiner_Et_Compter(A,B)
CurseurA <- Debut(A)
CurseurB <- Debut(B)
CptInversion <- 0
TantQue les deux listes sont non vides Faire
  a <- valeur(CurseurA)
  b <- valeur(CurseurB)
  Si b < a Alors
    CptInversion <- CptInversion + taille restante de A
  Finsi
  faire avancer le curseur des listes sur le plus petit (?)
FintantQue
Fin

```

La Figure 1.1 illustre un exemple de fonctionnement du tri par fusion sur le tableau $A = \langle 5, 4, 2, 7, 1 \rangle$.

Chaque étape diviser génère alors deux sous-séquences de taille $n/2$ exactement :

- Diviser : L'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant. Donc $D(n) = O(1)$.
- Régner : On résout récursivement deux sous-problèmes, chacun ayant la taille $n/2$, ce qui contribue pour $2T(n/2)$ au temps d'exécution.
- Combiner : la procédure FUSION sur un sous-tableau à n éléments prenait un temps $O(n)$, de sorte que $C(n) = O(n)$.

La complexité de l'algorithme tri par fusion peut être calculée de la manière suivante : $\begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{si } n > 1 \end{cases}$

Exercice 3 : supposez que l'on ait deux piles de cartes posées à l'endroit sur la table. Chaque pile est triée de façon à ce que la carte la plus faible soit en haut. On veut fusionner les deux piles pour obtenir une pile unique triée, dans laquelle les cartes seront à l'envers. L'étape fondamentale consiste

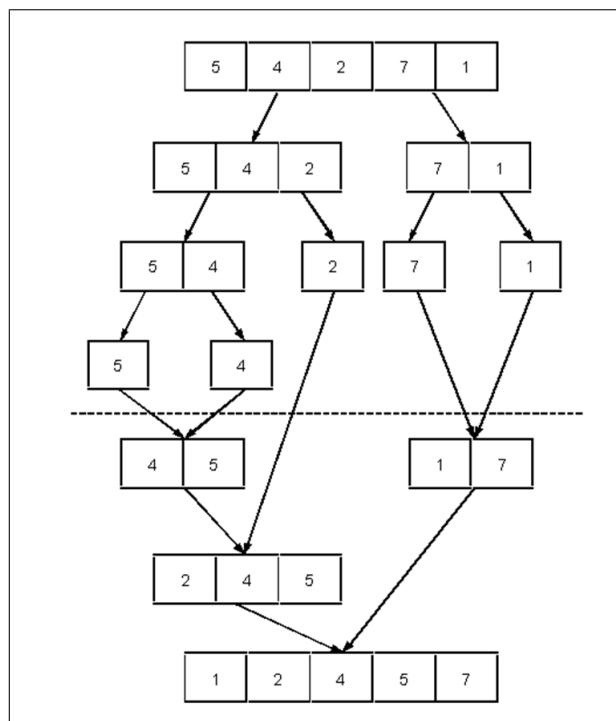


FIGURE 1.1 – Exemple de fonctionnement du tri par fusion

à choisir la plus faible des deux cartes occupant les sommets respectifs des deux piles, à la retirer de sa pile (ce qui a pour effet d'exposer une nouvelle carte), puis à la placer à l'envers sur la pile résultante. On répète cette étape jusqu'à épuisement de l'une des piles de départ, après quoi il suffit de prendre la pile qui reste et de la placer à l'envers sur la pile résultante. En utilisant cette description, donner le pseudo code de la fonction FUSIONNER.

Exercice 4 : donner le fonctionnement du tri par fusion sur le tableau $A = <5, 2, 4, 7, 1, 3, 2, 6>$. Pour trier la séquence $A = A[1], A[2], \dots, A[n]$, on fait l'appel initial $TRI-FUSION(A, 1, \text{longueur}[A])$.

Le mot asymptotique signifie qu'une tendance tend vers une droite (un maximal théorique) en s'en rapprochant de plus en plus sans jamais l'atteindre. En d'autres termes, c'est une valeur approchée qui tend de plus en plus vers la valeur véritable lorsqu'un paramètre tend vers une certaine limite.

L'objectif de ce chapitre est de comprendre les symboles suivants, dont certains ont été aperçus rapidement dans le chapitre précédent : « O , Θ , Ω », et d'apprendre comment trouver des formules de performance sans grandes difficultés. En particulier, il s'agit :

- Analyser précisément un algorithme même dans le pire des cas est impossible, d'où l'analyse asymptotique.
- Comparer des algorithmes différents sans les implémenter, sans développer des programmes ou utilisant leur efficacité asymptotique.

II Les notations asymptotiques

La notation O

Soient f et g deux fonctions de N dans R . On dit que $g(n)$ est une borne supérieure asymptotique pour $f(n)$ et l'on écrit $f(n) \in O(g(n))$ si et seulement si il existe une constante c supérieure à 0 et un n_0 tel que, pour tout $n \geq n_0$: $0 \leq f(n) \leq cg(n)$. Dans ce cas, $f(n)$ est bornée supérieurement par $g(n)$. Pour indiquer que $f(n) \in O(g(n))$, on écrit $f(n) = O(g(n))$.

La notation Ω

On dit que $g(n)$ est une borne inférieure asymptotique pour $f(n)$ et l'on écrit $f(n) \in \Omega(g(n))$ si et seulement si il existe une constante c supérieure à 0 et un n_0 tel que, pour tout $n \geq n_0$: $cg(n) \leq f(n)$. Pour indiquer que $f(n) \in \Omega(g(n))$, on écrit $f(n) = \Omega(g(n))$.

La notation Θ

On dit que $g(n)$ est une borne asymptotiquement approchée de $f(n)$ et l'on écrit $f(n) \in \Theta(g(n))$ si et seulement si il existe deux constantes strictement positives c_1 et c_2 tel que, pour n assez grand : $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. Pour indiquer que $f(n) \in \Theta(g(n))$, on écrit $f(n) = \Theta(g(n))$.

Par exemple, pour montrer que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$, on doit déterminer des constantes positives c_1 et c_2 , et n_0 telles que : $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$. On prenant $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ et $n_0 = 7$ on peut vérifier que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Pour deux fonctions quelconques $f(n)$ et $g(n)$, on a $f(n) = \Theta(g(n))$ si et seulement si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$.

La Figure 1.2 illustre les notations O , Ω , et Θ .

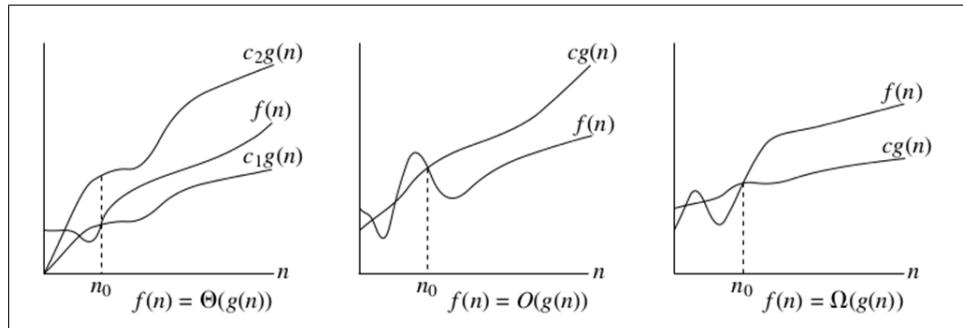


FIGURE 1.2 – Les notation O , Ω , et Θ

III Echelle de complexité

Nous donnerons ici les ordres de grandeur et leur correspondance :

- $O(1)$: Coût constant pour une instruction qui s'exécute une seule fois, ou bien en un nombre fini d'opérations.
exemple : L'accès à un élément dans un tableau, ou le calcul d'une expression.
- $O(\log(n))$: Le coût logarithmique quand on part d'un grand problème et qu'on le transforme en un sous-problème plus petit par réduction.
Exemples : Réduction dichotomique, ou opérations sur des arbres binaires de recherche.

Remarque : Même si N augmente, $\log(N)$ augmente très lentement.

- $O(n \cdot \log(n))$ Ce coût correspond aux algorithmes qui résument un problème en le divisant en sous-problèmes plus petits et en les résolvant indépendamment les uns des autres, et qui enfin combine les résultats.
Exemple : C'est le cas des tris récursifs.
- $O(n^2)$: Le coût est quadratique. Quand on traite les éléments par paire (2 boucles imbriquées par exemple).
Exemple : Manipulation de matrices, tri par comparaisons successives.
- $O(n^3)$: Le coût est cubique.
- $O(2^n)$: Coût exponentiel.

Un moyen de comparer deux fonctions est de calculer leur rapport asymptotique : $f(n) < g(n) \Leftrightarrow \lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$

Par exemple, si l'on souhaite comparer les fonctions n et n^2 : $n < n^2 \Leftrightarrow \lim_{n \rightarrow +\infty} \frac{n}{n^2} = 0$, donc $n = O(n^2)$

Exercice 1 : Calculer la complexité de l'algorithme de tri par insertion dans le cas le plus défavorable, cad la liste est triée à l'envers.

```

Pour j allant de 2 à N Faire
    clé <- A[j]
    i <- j - 1
    Tant que i > 0 et A[i] > clé Faire
        A[i+1] <- A[i]
        i <- i - 1
    FinTantQue
    A[i+1] <- clé
FinPour

```

Exercice 2 : Montrer que, pour deux constantes réelles a et b quelconques avec $b > 0$, l'on a : $(n + a)^b = \Theta(n^b)$.

Exercice 3 : Est-ce que $2^{n+1} = O(2^n)$? Est-ce que $2^{2n} = O(2^n)$?

Exercice 4 : Motrer le classement suivant des fonctions par rapport à leur vitesses de croissance asymptotique.

$$1 < 2^{100^{100}} < \log^*(\log(n)) < \log^a(n) < \ln(n) \simeq \sum_{n=1}^n 1/n < (\sqrt{2})^{\log(n)} < \log(n!) \simeq n \log(n) < n^2 \simeq \sum_{k=1}^n nk < (\log(n))^{\log(n)} \simeq n^{\log(\log(n))} < 2^n < 3^n < n!$$

Pour montrer que $O(3^n) \neq O(2^n)$, nous supposons que $O(3^n) = O(2^n)$
 $\Rightarrow \exists c, n$ tels que $3^n \leq 2^n \Rightarrow (3/2)^n \leq c$, ce qui est absurde.

IV Les récurrences

Quand un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une récurrence.

Une récurrence est une équation ou inégalité qui décrit une fonction à partir de sa valeur sur des entrées plus petites. Par exemple, l'équation de récurrence pour le tri par fusion est écrite comme suit :

$$\begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Il existe trois méthodes pour résoudre les récurrences, c'est-à-dire pour obtenir des bornes asymptotiques « Θ » ou « O » pour la solution :

- **La méthode de substitution** : on devine une borne puis on utilise une récurrence mathématique pour démontrer la validité de la conjecture.
- **La méthode de l'arbre récursif** : convertit la récurrence en un arbre dont les noeuds représentent les coûts induits à différents niveaux de la récursivité.
- **La méthode générale** : fournit des bornes pour les récurrences de la forme :
$$\begin{cases} O(1) & \text{si } c \geq n \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{sinon} \end{cases}$$

Le reste de cette section montre l'utilisation de la méthode générale pour calculer des bornes pour les récurrences. Cette méthode donne une « recette » pour résoudre les récurrences de la forme : $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, où $a \geq 1$ et $b > 1$ sont des constantes, et $f(n)$ une fonction asymptotiquement positive.

La méthode générale s'appuie sur le théorème suivant. $T(n)$ peut alors être bornée asymptotiquement de la manière suivante :

- Cas 1 : Si $f(n) = O(n^{\log_b a - \epsilon})$ pour une certaine constante $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
- Cas 2 : Si $f(n) = \Theta(n^{\log_b a})$ alors $T(n) = \Theta(n^{\log_b a} \log n)$.
- Cas 3 : Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ pour une certaine constante $\epsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une certaine constante $c < 1$ et pour tout n suffisamment grand alors $T(n) = \Theta(f(n))$.

Dans chacun des trois cas, on compare la fonction $f(n)$ à la fonction $n^{\log_b a}$. Intuitivement, la solution de la récurrence est déterminée par la plus grande des deux fonctions (Cas 1 et 3). Si les deux fonctions ont la même taille (cas 2), on multiplie par un facteur logarithmique.

Par exemple, considérons $T(n) = 9T(n/3) + n$. Pour cette récurrence, on a $a = 9$, $b = 3$ et $f(n) = n$; on a donc $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Puisque $f(n) = O(n^{\log_3 9 - \epsilon})$, où $\epsilon = 1$, on peut appliquer le cas 1 du théorème général et dire que la solution est $T(n) = \Theta(n^2)$.

Exercice 5 : Donner des bornes asymptotiques supérieures pour $T(n)$ dans chacune des récurrences suivantes et justifier les réponses :

- $T(n) = T(2n/3) + 1$
- $T(n) = 3T(n/4) + n\log n$
- $T(n) = 4T(n/2) + n$
- $T(n) = 4T(n/2) + n^2$
- $T(n) = 4T(n/2) + n^3$

Exercice 6 : calculer la complexité de l'algorithme du Tri-Fusion. Cet algorithme suit l'approche Diviser pour mieux régner qui consiste à décomposer le problème initial en sous-problèmes de petite taille puis résoudre les problèmes décomposés un par un. Nous avons dans cet algorithme, 3 étapes distinctes.

- 1^{ière} étape : diviser le problème en un certain nombre de problèmes.
- 2^{ième} étape : régner sur les sous-problèmes ou la résolution est plus triviale.
- 3^{ième} étape : combiner les solutions des sous-problèmes pour obtenir une solution au problème initial.

Exercice 7 : Utiliser la méthode générale pour montrer que la solution de la récurrence $T(n) = T(n/2) + \Theta(1)$ associée à la recherche dichotomique) est $T(n) = \Theta(\log n)$.

Exercice 8 : La méthode générale est-elle applicable à la récurrence $T(n) = 4T(n/2) + n^2\log n$? Pourquoi? Donner une borne supérieure asymptotique pour cette récurrence.

V Travaux dirigés

Exercice 1

Quelle est la valeur renvoyée par la fonction suivante ? Exprimer la réponse en fonction de n . Donner le temps d'exécution dans le pire cas.

```

fonction mystère (n)
début
r := 0 ;
pour i allant de 1 à n-1 faire
    pour j allant de i+1 à n faire
        pour k allant de 1 à j faire
            r := r+1 ;
        fin pour
    fin pour
fin pour
retourner r ;
fin

```

Le résultat de cet algorithme est la valeur : $r = r + \sum_{i+1}^n j$.

Calculer le temps d'exécution de cette fonction revient à calculer la valeur renvoyée r .

La boucle suivante effectue un calcul équivalent à $r = r + j$.

```

pour k allant de 1 à j faire
    r := r+1 ;
fin pour

```

Par conséquent, la boucle :

```

pour j allant de i+1 à n faire
    pour k allant de 1 à j faire
        r := r+1 ;
    fin pour
fin pour

```

réalise en fait :

```

pour j allant de i+1 à n faire
    r := r+j ;
fin pour

```

Cette boucle effectue un calcul équivalent à $r = r + \sum_{j=i+1}^n j$.

$$\text{Or } \sum_{j=i+1}^n j = \sum_{j=1}^n j - \sum_{j=1}^i j = \frac{n(n+1)}{2} - \frac{i(i+1)}{2}.$$

D'où la boucle :

```

pour i allant de 1 à n-1 faire
    pour j allant de i+1 à n faire
        pour k allant de 1 à j faire
            r := r+1 ;
        fin pour
    fin pour
fin pour

```

$$\begin{aligned}
r &= r + \sum_{i=1}^{n-1} \left(\frac{n(n+1)}{2} - \frac{i(i+1)}{2} \right) \\
r &= r + \sum_{i=1}^{n-1} \frac{n(n+1)}{2} - \sum_{i=1}^{n-1} \frac{i^2}{2} - \sum_{i=1}^{n-1} \frac{i}{2} \\
r &= r + \frac{n(n^2-1)}{3} \\
\text{NB : } \sum_{i=1}^n i^2 &= \frac{n(n+1)(2n+1)}{6}
\end{aligned}$$

C'est-à-dire le nombre d'exécution de l'opération d'affectation dans l'algorithme. Le temps d'exécution de l'algorithme est donc $T(n) = O(n^3)$. L'algorithme est donc borné supérieurement par n^3 .

Exercice 2 : La recherche binaire

On considère le problème suivant. Soient un tableau $A[1..n]$ et une valeur v donnée. Trouver l'indice i dans le tableau sachant que $v = A[i]$. Si v n'apparaît pas dans le tableau, retourner 0.

Si A est trié, on peut utiliser le principe de la recherche binaire : on compare v avec le milieu du tableau et on répète la recherche sur une des moitiés du tableau en éliminant l'autre moitié. Dans cet exercice, on considérera n comme une puissance de 2.

- Donner l'algorithme en version récursive, calculer son coût.
- Donner l'algorithme en version itérative, calculer son coût.

La version récursive de l'algorithme.

```

Recherche_binaire (A : tableau, v,i,j : entiers)
    -- v : valeur recherchée
    -- i : début du tableau
    -- j : fin du tableau
debut
    si i > j alors
        retourner 0 ;
    fin si

    milieu := (i+j)/2 ;

    si A[milieu] = v alors
        retourner milieu ;
    sinon si A[milieu] > v alors
        retourner Recherche_binaire(A, v, i, milieu - 1) ;
    sinon
        retourner Recherche_binaire(A, v, milieu + 1, j) ;
    fin si
fin

```

Il faut déterminer la relation de récurrence du temps d'exécution en fonction de n . Soit $T(n)$ le temps d'exécution de l'algorithme sur un tableau de taille n :

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + O(1)$$

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + O(1)$$

...

$$T(1) = T\left(\frac{n}{2^p}\right)$$

avec $n = 2^p$, donc $p = \log_2 n$. Cette méthode est efficace pour des variables de petite taille.

La version itérative de l'algorithme :

```

Recherche_binaire (A : tableau, v : entiers)
    -- v : valeur recherchée
variables
    i, j : entiers ;
debut
    i := 1 ;
    j := longueur (A) ;
    tant que i ≤ j faire
        milieu := (i+j)/2 ;
        si A[milieu] = v alors
            retourner milieu ;
        sinon si A[milieu] > v alors
            j := milieu - 1 ;
        sinon
            i := milieu + 1 ;
finsi
    fin faire

fin

```

Le coût de l'algorithme est : $T(n) = O(\log_2 n)$

Exercice 3 : La recherche du minimum et du maximum

L'algorithme suivant est basé sur le paradigme de « diviser pour régner ». Il permet de trouver la valeur maximale dans un tableau $A[1..n]$.

```

fonction maximum (A, x, y)
// revoie le maximum de A, x début du tableau, y fin du tableau
début
    si y-x ≤ 1 alors
        si A[x] > A[y] alors
            retourner A[x] ;
        sinon
            retourner A[y] ;
        fin si
    sinon
        max1 := maximum (A, x, (x+y)/2) ;

```

```

max2 := maximum(A, (x+y)/2 +1, y) ;
si max1 > max2 alors
    retourner max1 ;
sinon
    retourner max2 ;
fin si
fin si
fin

```

- Montrer que l'algorithme fonctionne.
- Donner une relation de récurrence exprimant le nombre de comparaisons utilisées dans le pire des cas. On peut supposer que n est une puissance de 2
- Quel est le temps d'exécution de l'algorithme ?

Cet algorithme fonctionne car il est récursif sur tout tableau jusqu'à 2 éléments.

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

Le temps d'exécution correspond au nombre de comparaisons : $T(n) = O(n)$

Exercice 4 : La recherche d'une valeur encadrée

Étant donné un tableau trié d'entiers $A[s..f]$ et deux entiers ("bornes") a et b , on cherche s'il existe un élément $A[i]$ du tableau tel que $a \leq A[i] \leq b$ (s'il y en a plusieurs trouvez un). Exemple, soit le tableau $A[1..5] = [3, 7, 8, 43, 556]$ et les bornes $a = 40$, $b = 50$. Dans ce cas-là la valeur encadrée existe : c'est $A[4] = 43$. Donnez (en pseudocode) un algorithme "diviser-pour-régner" qui résout ce problème.

Coupons le tableau en deux moitiés : $A[s..m]$ et $A[m + 1..f]$. Les trois cas ci-dessous correspondent à trois positions de l'élément de milieu $A[m]$ par rapport à l'intervalle $[a, b]$:

- À gauche : $A[m] < a$. Dans ce cas-là tous les éléments de la moitié gauche du tableau sont aussi inférieurs à a , et ne peuvent pas appartenir à l'intervalle demandé $[a, b]$. On va chercher la valeur encadrée dans la moitié droite seulement.
- Dedans : $a \leq A[m] \leq b$. On a déjà trouvé une valeur encadrée $A[m]$. On retourne son indice.

- À droite : $b < A[m]$. Ce cas est symétrique au premier, la recherche peut être limitée à la moitié gauche du tableau.

Cette analyse mène à la fonction récursive suivante *chercher(s, f)* qui renvoie l'indice de la valeur encadrée dans le tableau $A[s..f]$ (ou Null s'il n'y en a pas. On suppose que le tableau A , et les bornes a et b sont des variables globales.

```

Chercher (s,f)
début
    // cas de base
    si  (s = f) alors
        si A[s] appartient à [a,b] alors retourner s
        sinon retourner Null
        fin si
    fin si
    // on divise
    m=s+f/2
    si A[m] < a alors ind=Chercher (m+1,f)
    sinon
        si a <=A[m] <=b alors ind=m
        sinon ind=Chercher (s,m)
        fin si
    fin si
    retourner ind
fin

```

On a réduit un problème de taille n à un seul problème de la taille $n/2$ et des petits calculs en $O(1)$, donc la complexité satisfait la récurrence : $T(n) = T(n/2) + O(1)$. En la résolvant par Master Theorem on obtient que $T(n) = O(\log n)$.

Exercice 5 : La recherche d'un objet vainqueur

Les éléments d'un tableau donné $B = (b_1, b_2, \dots, b_n)$ sont des objets dont on peut tester l'égalité, mais qu'on ne peut pas comparer (ou non plus classer). Le vainqueur de B est l'élément présent dans le tableau strictement plus que $n/2$ fois.

1. Démontrer qu'un tableau peut contenir soit 0 soit 1.

2. Programmez une fonction booléenne $estVainqueur(x, B, s, f)$ qui teste est-ce que x est vainqueur de (bs, \dots, bf) .
3. Proposez un algorithme itératif $VainqueurNaif(B, s, f)$ qui trouve le vainqueur ou répond qu'il n'y en a pas. Quelle est sa complexité ?
4. Proposez un algorithme plus efficace de type Diviser-Pour-Régner qui trouve le vainqueur ou répond qu'il n'y en a pas.
 - On cherche à programmer la fonction $Vainqueur(B, s, f)$ qui renvoie la valeur du vainqueur de (bs, \dots, bf) ou null s'il n'y a pas de champion
 - Coupez le tableau en deux moitiés.
 - Chaque moitié peut avoir ou non son vainqueur (il y a en tout 4 cas). Qu'est-ce qu'on peut affirmer sur le vainqueur du grand tableau pour chacun de ces 4 cas. Justifiez vos réponses.
 - Donnez un algorithme récursif pour $Champ(B, s, f)$.
5. Analysez la complexité de votre algorithme Diviser-Pour-Régner

Supposons qu'un tableau contient 2 vainqueurs différents a et b (et éventuellement d'autres). Par définition de vainqueur, il y a plus de $n/2$ occurrences de a et plus $n/2$ occurrences de b - ça fait en tout plus de n objets dans un tableau de n . Contradiction. Donc on a toujours moins de 2 vainqueurs.

Voici un algo de complexité $O(n)$ (ou, plus précisément, $O(f - s)$).

```
Boolean estVainqueur(x,B,s,f)
début
    count=0
    pour i allant de s à f faire
        si x=B[i] alors count++
        fin si
        fin pour

        si 2*count>f-s+1 alors retourner vrai
        sinon retourner faux
        fin si
    fin
```

Il suffit de tester chaque objet avec la fonction précédente.

$VainqueurNaif(B, s, f)$

```

début
    pour i allant de s à f faire
        si estVainqueur(B[i],B,s,f) alors retourner B[i]
    fin pour
    retourner Null
fin

```

Il y a au pire n itérations de complexité $O(n)$ chacune, donc la complexité totale est $O(n^2)$.

L'observation clé est que le vainqueur de tableau doit être aussi vainqueurs d'une de ses moitiés (peut-être des deux). Autrement dit les seuls candidats à tester dans le grand tableau sont les vainqueurs de la moitié gauche et le vainqueur de la moitié droite.

```

Vainqueur(B,s,f)
début
    si s=f alors retourner B[s]

    m=(s+f)/2

    VGauche=Vainqueur(B,s,m)
    si VGauche !=null ET estVainqueur(VGauche,B,s,f) alors retourner VGauche

    VDroite=Vainqueur(B,m+1,f)
    si VDroite !=null ET estVainqueur(chDroite,B,s,f) alors retourner VDroite

    retourner null
fin

```

La fonction fait deux appels récursifs et encore $O(n)$ opérations. Donc $T(n) = 2T(n/2) + O(n)$, ce qui correspond au cas moyen du Master Theorem, et la complexité est $O(n \log n)$ (voir chapitre 2).

Exercice 6

Prouver que $(n + 1)^2 = O(n^2)$
 $f(n) = (n + 1)^2, g(n) = n^2$
 $(n + 1)^2 = O(n^2) \Leftrightarrow \exists c, n_0 / \forall n > n_0, f(n) \leq c \cdot g(n)$

$$(n+1)^2 = n^2 + 2n + 1 = n^2(1 + 2/n + 1/n^2) \leq n^2(1 + 2 + 1) \leq 4n^2$$

$$\Rightarrow (n+1)^2 = O(n^2)$$

Montrer que $\log(n) = O(n)$

$$n = 1 \Rightarrow 0 \leq 1$$

Supposons que $\log(n) \leq n$

$$\exists c = 1, n = 1, \forall n \geq n_0, \log(n) \leq c \cdot n$$

$$\log(n) = O(n)$$

Montrer que $3n\log(n) = O(n^2)$

Il suffit de prendre $c = 3$, et $n_0 = 1$, on a :

$$\forall n \geq 1, \log(n) \leq n,$$

$$3n \cdot \log(n) \leq 3n^2.$$

Montrer que $n^2/2 - 3n = \Theta(n^2)$

$$C_1 n^2 \leq n^2/2 - 3n \leq C_2 n^2,$$

Montrer que $6n^3 \neq \Theta(n^2)$

$$\lim(6n^3/k \cdot n^2) = \lim(n) = +\infty$$

$$\not\exists k / 6n^3 \leq k \cdot n^2,$$

$$\text{donc } 6n^3 \neq \Theta(n^2)$$

Exercice 7

Donner des bornes asymptotiques supérieures pour $T(n)$ dans chacune des récurrences suivantes. On suppose que $T(n)$ est constante pour $n \neq 2$. Rendre les bornes les plus approchées possible, et justifier les réponses.

— a. $T(n) = 4T(n/2) + n^3$

— b. $T(n) = T(n-2) + n^2$ (on suppose que $T(1) = T(2) = 1$)

— c. $T(n) = 5T(n/3) + n\sqrt{n}$

— d. $T(n) = 9T(n/3) + n$

— e. $T(n) = T(2n/3) + 1$

Résolutions des récurrences :

- a. Cas 3 : $f(n) = n^3 = \Omega(n^{2+\epsilon})$, for $\epsilon = 1 > 0$, donc $T(n) = \Theta(n^3)$
- b. On utilise la méthode de substitution, $T(n) = O(n^3)$. $T(n) = n^2 + (n-2)^2 + \dots + 1 \leq n^2 + n^2 + \dots n^2 \leq n \cdot n^2 = n^3 = O(n^3)$.
- c. Cas 3 : $n^{\log_b a} = n^{\log_3 5}$ et $f(n) = n^{3/2}$. En fait, $3/2 > \log_3 5 \Leftrightarrow 3^{3/2} > 5 \Leftrightarrow \sqrt{27} > 5$, qui est vrai, donc le cas 3 peut être appliqué, $T(n) = \Theta(n\sqrt{n})$
- d. Cas 1 : on a $a = 9$, $b = 3$ et $f(n) = n$; on a donc $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Puisque $f(n) = O(n^{\log_3 9 - \epsilon})$, où $\epsilon = 1$, la solution est $T(n) = \Theta(n^2)$.
- e. Cas 2 : $a = 1$, $b = 3/2$, $f(n) = 1$ et $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Donc la solution de la récurrence est $T(n) = \Theta(\log n)$.

Exercice 8

Pour chacune des fonctions suivantes, déterminer sa complexité asymptotique dans la notation Grand-O. Exemple : $3n$ est en $O(n)$, $3n = O(n)$.

1. $6n^3 + 10n^2 + 5n + 12 = O(n^3)$
2. $3\log_2 n + 4 = \log_2(n)$
3. $2^n + 6n^2 + 7n = O(2^n)$
4. $4\log_2 n + n = O(n)$
5. $n\log n = O(n^2)$
6. $2^n + 10 = O(2^n)$
7. $25n^4 - 10n^3 + 22n^2 = O(n^4)$
8. $n^2/2 - 3n = O(n^2)$

Exercice 9

Donner des bornes asymptotiques supérieures des récurrences suivantes et justifier votre réponse :

1. $T(n) = 7T(n/2) + O(n)$
2. $T(n) = T(n/2) + O(1)$
3. $T(n) = 2T(n/2) + O(n)$
4. $T(n) = 2T(n/2) + O(1)$

Exercice 10

Expliquer dans chacun des cas suivants pourquoi le Master Theorem ne s'applique pas :

1. $T(n) = 2^n T(n/2) + n^3$
2. $T(n) = T(n/2) + T(n/4) + n^2$
3. $T(n) = 1/2T(n/2) + n$.

Le Master Theorem ne s'applique pas pour les raisons suivantes :

1. a n'est pas une constante.
2. On a deux sous-problèmes de tailles différentes.
3. La condition $a \geq 1$ n'est pas vérifiée. .

Exercice 11

1. Déterminez la complexité de l'algorithme suivant. Justifiez votre réponse.

```
Action algoA ( m : Entier ; n : Entier )
Variable i , j : Entier
Debut
    i <- 1
    j <- 1
    TantQue ( i <= m ) OU ( j <= n ) Faire
        i <- i + 1
        j <- j + 1
    FinTantQue
Fin
```

2. Considérer les deux algorithmes A_1 et A_2 avec leurs temps d'exécution $T_1(n) = 9n^2$ et $T_2(n) = 100n + 96$ respectives.

- Déterminer la complexité asymptotique des deux algorithmes dans la notation Grand-O. Quel algorithme a la meilleure complexité ?
- Quelle est la complexité asymptotique de l'algorithme suivant ? Quelle règle avez-vous appliquée ?

```
début
    appeler A1 {Ici l'algorithme 1 est exécuté}
    appeler A2 {Ici l'algorithme 2 est exécuté}
fin
```

Exercice 12

La relation de récurrence de l'algorithme "tri fusion" exprimant le nombre de comparaisons utilisées dans le pire des cas est $T(n) = 2T(n/2) + O(n)$. On peut supposer que n est une puissance de 2, quel est le temps d'exécution de l'algorithme en utilisant les deux méthodes : méthode de l'arbre récursif et la méthode générale.

La Figure 1.3 montre un exemple de l'arbre récursif associé à la récurrence $T(n) = 3T(n/4) + cn^2$.

- La taille de sous-problème pour un nœud situé à la profondeur i est de $n/4^i$.
- La taille de sous-problème prendra la valeur $n = 1$ quand $n/4^i = 1$ ou, de manière équivalente, quand $i = \log_4 n$.
- L'arbre a $\log_4 n + 1$ niveaux, $(0, 1, 2, \dots, \log_4 n)$.

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2cn^2 + \dots + (\frac{3}{16})^{\log_4 n}cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) = \frac{(\frac{3}{16})^{\log_4 n} - 1}{\frac{3}{16} - 1} cn^2 + \Theta(n^{\log_4 3}) = O(n^2) \end{aligned}$$

La Figure 1.4 montre l'arbre des appels récursifs, sur chaque sommet étant indiqué la taille du paramètre lors de l'appel en question. On a supposé que n était une puissance de 2, donc n est de la forme $n = 2^h$ où h est la hauteur de l'arbre. C'est pourquoi $h = \log_2(n)$. Le coût total est alors égal à la somme des coûts de tous les niveaux, i.e. $n(\log_2(n) + 1) = O(n \log_2(n))$

VI Travaux pratiques

TP1 : l'objectif de ce TP est de comparer les coûts expérimentaux et théoriques des algorithmes itératifs et récursifs.

Ex1 : chercher le maximum dans un tableau

Implémenter l'algorithme permettant de chercher le maximum dans un tableau de n éléments entiers en utilisant la méthode incrémentale et la méthode « diviser pour régner ».

- Donner les courbes expérimentales correspondantes (utilisé la fonction `rand()`) sur des tableaux de données aléatoires (utilisé la fonction `clock()` de la bibliothèque `time`).
- Comparer ces courbes avec les courbes des coûts théoriques respectifs $O(n^{02})$ et $(n \log n)$ (sous Excel).

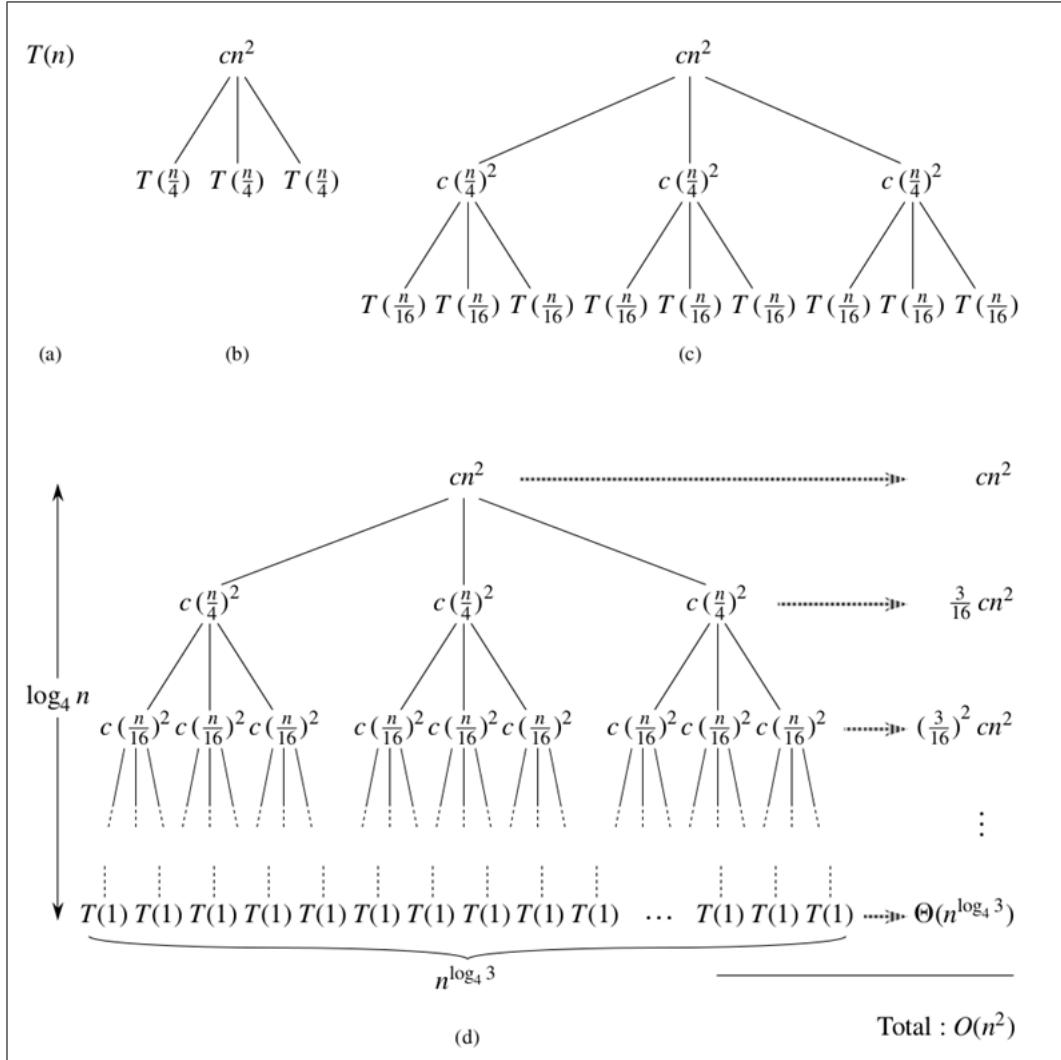


FIGURE 1.3 – La partie (a) montre $T(n)$, progressivement développé dans les parties (b)–(d) pour former l’arbre récursif. L’arbre entièrement développé, en partie (d), a une hauteur de $\log_4 n$ (il comprend $\log_4 n + 1$ niveaux)

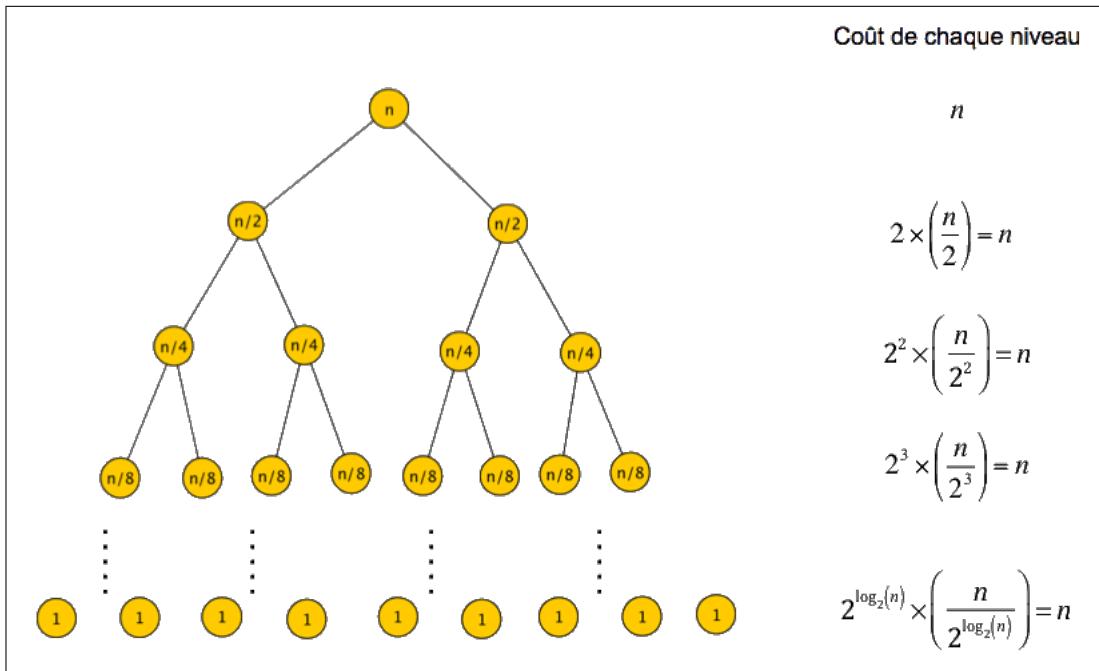


FIGURE 1.4 – L’arbre des appels récursifs

Ex2 : chercher une valeur dans un tableau

On considère le problème suivant : Soient un tableau $A[1..n]$ et une valeur v donnée. Trouver l’indice i dans le tableau sachant que $v = A[i]$. Si v n’apparaît pas dans le tableau, retourner 0. Si A est trié, on peut utiliser le principe de la recherche binaire : on compare v avec le milieu du tableau et on répète la recherche sur une des moitiés du tableau en éliminant l’autre moitié. Dans cet exercice, on considérera n comme une puissance de 2.

- Donner l’algorithme en version récursive et calculer son coût.
- Donner l’algorithme en version itérative et calculer son coût.
- Implémenter les deux algorithmes en langage C.
- Comparer le temps d’exécution avec la complexité théorique respectifs.

Ex3 : Tri par insertion/fusion

Tri par insertion : on avance progressivement dans le tableau en laissant derrière un sous tableau déjà trié. Plus précisément, je prends le premier élément du reste du tableau non encore trié et je l’insère au bon endroit dans le sous tableau déjà trié.

Tri rapide : on divise le tableau en sous tableaux d'une manière récursive jusqu'à obtenir des tableaux de taille 1, on recombine ensuite les sous tableaux tout en triant jusqu'à obtenir le tableau entier trié.

Le travail demandé :

- Implanter différents algorithmes de Tri (Tri par insertion et Tri par fusion).
- Donner les courbes expérimentales correspondantes aux coûts des différentes versions sur des tableaux de données aléatoires.
- Comparer ces courbes avec les courbes des coûts théoriques respectifs $O(n^2)$ et $O(n \log n)$ (sous Excell).

Documents à fournir (via la plateforme) pour chaque TP : rapport (max 5 pages), les sources et les jeux de tests. Ils seront rendus dans une archive (nom-prenom-TPx.zip/gz).

Chapitre 2

Les arbres

I Arbres binaires de recherche

Les arbres de recherche sont des structures de données pouvant supporter un nombre d'opérations d'ensemble dynamique, e.g., :Rechercher, Minimum et Maximum, Insérer et Supprimer, Prédécesseur et Successeur.

Les opérations basiques sur un arbre binaire de recherche dépensent un temps proportionnel à la hauteur de l'arbre :

- $\Theta(\log_2 n)$: dans le cas le plus défavorable et l'arbre binaire est complet à n noeuds.
- $\Theta(n)$: dans le cas le plus défavorable et l'arbre se réduit à une chaîne linéaire de n noeuds.

Un arbre binaire de recherche est organisé comme un arbre binaire (voire Figure 2.1).

Les clés d'un arbre binaire de recherche sont toujours stockées de manière à satisfaire à la propriété d'arbre binaire de recherche. Soit x un noeud d'un arbre binaire de recherche :

- Si y est un noeud du sous-arbre de gauche de x , alors $cle[y] \leq cle[x]$
- Si y est un noeud du sous-arbre de droite de x , alors $cle[x] \leq cle[y]$.

En d'autres termes, pour tout noeud x (Cf. Figure 2.2) :

- Tous les éléments dans le sous-arbre gauche de x sont $< x$,
- Tous les éléments dans le sous-arbre droite de x sont $> x$

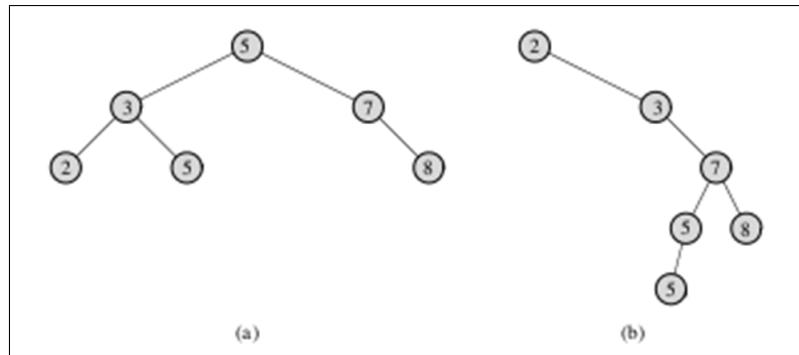


FIGURE 2.1 – Un arbre binaire de recherche de 6 noeuds et de hauteur 2.
(b) Un arbre binaire de recherche moins efficace de hauteur 4, contenant les mêmes clés.

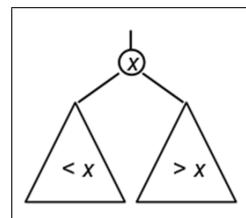


FIGURE 2.2 – Un arbre binaire de recherche.

1. Parcours préfixe et postfixe

La propriété d'arbre binaire de recherche permet d'afficher toutes les clés de l'arbre dans l'ordre trié à l'aide d'un algorithme récursif simple :

- Parcours infixé : imprimer la clé de la racine d'un sous-arbre entre les clés du sous-arbre de gauche et les clés du sous-arbre de droite.
- Parcours préfixé : imprimer la racine avant les valeurs de chacun de ses sous-arbres.
- Parcours postfixé : imprimer la racine après les valeurs de ses sous-arbres.

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. Par exemple, la Figure 2.3 présente la procédure PARCOURS-INFIXE(racine[T]).

```
PARCOURS-INFIXE(x)
1   si x ≠ NIL
2       alors PARCOURS-INFIXE(gauche[x])
3           afficher clé[x]
4           PARCOURS-INFIXE(droite[x])
```

FIGURE 2.3 – La procédure PARCOURS-INFIXE(racine[T])

Par exemple, le parcours infixé imprimera les clés de chacun des deux arbres de la Figure 2.1 dans l'ordre 2,3,5,5,7,8.

2. Recherche

La Figure 2.4 présente la procédure permettant de rechercher un noeud ayant une clé donnée dans un arbre binaire de recherche.

Étant donné un pointeur sur la racine de l'arbre et une clé k , *ARBRE-RECHERCHER* retourne un pointeur sur un noeud de clé k s'il en existe un ; sinon, elle retourne *NIL*.

La Figure 2.5 illustre l'interrogation d'un arbre binaire de recherche pour rechercher la clé 13 dans l'arbre. Le temps d'exécution est donc $O(h)$ si h est la hauteur de l'arbre.

```
ARBRE-RECHERCHER( $x, k$ )
1 si  $x = \text{NIL}$  ou  $k = clé[x]$ 
2   alors retourner  $x$ 
3 si  $k < clé[x]$ 
4   alors retourner ARBRE-RECHERCHER( $\text{gauche}[x], k$ )
5   sinon retourner ARBRE-RECHERCHER( $\text{droite}[x], k$ )
```

FIGURE 2.4 – La procédure rechercher.

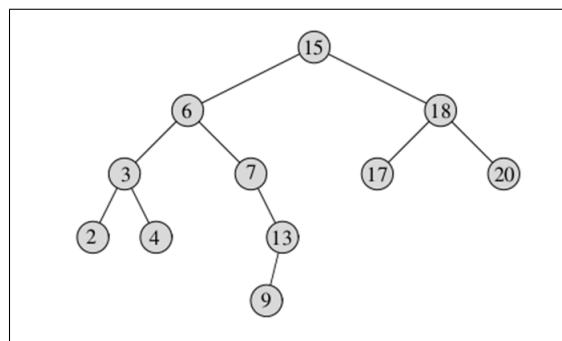


FIGURE 2.5 – Rechercher la clé 13 dans l'arbre on suit le chemin 15-6-7-13 en partant de la racine.

3. Minimum et maximum

On peut toujours trouver un élément d'un arbre binaire de recherche dont la clé est un minimum en suivant les pointeurs gauche à partir de la racine jusqu'à ce qu'on rencontre NIL.

La Figure 2.6 présente la procédure permettant de retourner un pointeur sur l'élément minimal du sous-arbre enraciné au noeud x .

```

ARBRE-MINIMUM( $x$ )
1 tant que  $gauche[x] \neq \text{NIL}$ 
2 faire  $x \leftarrow gauche[x]$ 
3 retourner  $x$ 
```

FIGURE 2.6 – Rechercher le Minimum

6 : Donner le pseudo-code de la procédure $ARBRE-MAXIMUM(x)$. Quel est son temps d'exécution.

4. Successeur et prédécesseur

La structure d'un arbre binaire de recherche permet de déterminer le successeur d'un nœud sans même effectuer de comparaison entre les clés. Si toutes les clés sont distinctes, le successeur d'un noeud x est le noeud possédant la plus petite clé supérieure à $cle[x]$. Le successeur et le prédécesseur d'un noeud x comme étant le noeud retourné par des appels à $ARBRE-SUCCESSEUR(x)$ et $ARBRE-PRÉDÉCESSEUR(x)$ respectivement (Cf. Figure 2.7).

```

ARBRE-SUCCESSEUR( $x$ )
1 si  $droite[x] \neq \text{NIL}$ 
2 alors retourner ARBRE-MINIMUM( $droite[x]$ )
3  $y \leftarrow p[x]$ 
4 tant que  $y \neq \text{NIL}$  et  $x = droite[y]$ 
5 faire  $x \leftarrow y$ 
6  $y \leftarrow p[y]$ 
7 retourner  $y$ 
```

FIGURE 2.7 – Le pseudo-code de ARBRE-SUCCESSEUR

Le code de ARBRE-SUCCESSEUR est séparé en deux cas :

- **Cas 1** : Si le sous-arbre de droite du noeud x n'est pas vide, alors le successeur de x est tout simplement le noeud le plus à gauche dans le sous-arbre de droite. e.g., le successeur du noeud de clé 15 dans la Figure 2.5 est le noeud de clé 17.
- **Cas 2** : si le sous-arbre de droite du noeud x est vide et que x a un successeur y , alors y est le premier ancêtre de x dont l'enfant de gauche est aussi un ancêtre de x . e.g., le successeur du noeud de clé 13 est le noeud de clé 15.

Question : Quel est le successeur des noeuds 15, 10, et 3 de la Figure 2.8

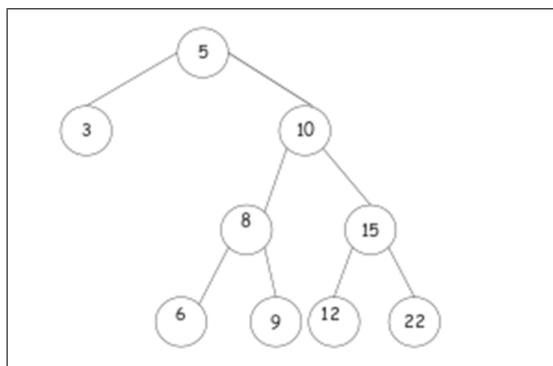


FIGURE 2.8 – Un arbre binaire

Le temps d'exécution de *ARBRE-SUCCESSEUR* sur un arbre de hauteur h est $O(h)$.

7 : Donner le pseudo-code de la procédure *ARBRE-PRÉDÉCESSEUR*. Quel est son temps d'exécution.

5. Insertion et Suppression

Les opérations d'insertion et de suppression modifient l'ensemble dynamique représenté par un arbre binaire de recherche.

a. Insertion

Pour insérer une nouvelle valeur v dans un arbre binaire de recherche T , on utilise la procédure *ARBRE-INSÉRER* (Cf. Figure 2.9). On lui passe un noeud z pour lequel $cle[z] = v$, $gauche[z] = NIL$ et $droite[z] = NIL$.

Le pointeur x parcourt le chemin et le parent de x est conservé dans le pointeur y . Après l'initialisation, la boucle tant que des lignes 3–7 fait descendre ces deux pointeurs le long de l'arbre, bifurquant à gauche ou à droite en fonction du résultat de la comparaison entre $cle[z]$ et $cle[x]$, jusqu'à ce que x prenne la valeur NIL . Ce NIL occupe la position où l'on souhaite placer l'élément d'entrée z . Les lignes 8–13 initialisent les pointeurs qui provoquent l'insertion de z .

```

ARBRE-INSÉRER( $T, z$ )
1  $y \leftarrow NIL$ 
2  $x \leftarrow racine[T]$ 
3 tant que  $x \neq NIL$ 
4   faire  $y \leftarrow x$ 
5     si  $clé[z] < clé[x]$ 
6       alors  $x \leftarrow gauche[x]$ 
7       sinon  $x \leftarrow droite[x]$ 
8    $p[z] \leftarrow y$ 
9   si  $y = NIL$ 
10    alors  $racine[T] \leftarrow z$             $\triangleright$  arbre  $T$  était vide
11    sinon si  $clé[z] < clé[y]$ 
12      alors  $gauche[y] \leftarrow z$ 
13      sinon  $droite[y] \leftarrow z$ 

```

FIGURE 2.9 – Le pseudo-code de *ARBRE-INSÉRER*

La Figure 2.10 illustre l'insertion d'un élément de clé 13. Les noeuds sur fond gris clair indiquent le chemin descendant de la racine vers la position où l'élément sera inséré. La ligne pointillée représente le lien qui est ajouté à l'arbre pour insérer l'élément

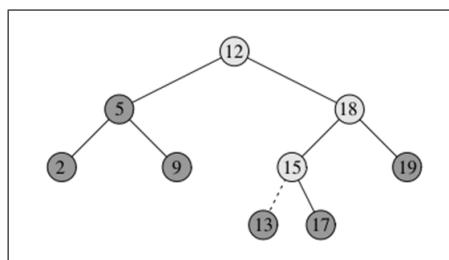


FIGURE 2.10 – Insertion d'un élément de clé 13

b. Suppression

La procédure permettant de supprimer un noeud donné z d'un arbre binaire de recherche prend comme argument un pointeur sur z . Trois cas possibles (Cf. Figure 2.11) :

- Cas a : si z n'a pas d'enfant, on modifie son parent $p[z]$ pour remplacer z par NIL dans le champ enfant.
- Cas b : si le noeud n'a qu'un seul enfant, on « détache » z en créant un nouveau lien entre son enfant et son parent.
- Cas c : si le noeud a deux enfants, on détache le successeur de z , y , qui n'a pas d'enfant gauche et on remplace la clé et les données satellites de z par la clé et les données satellite de y .

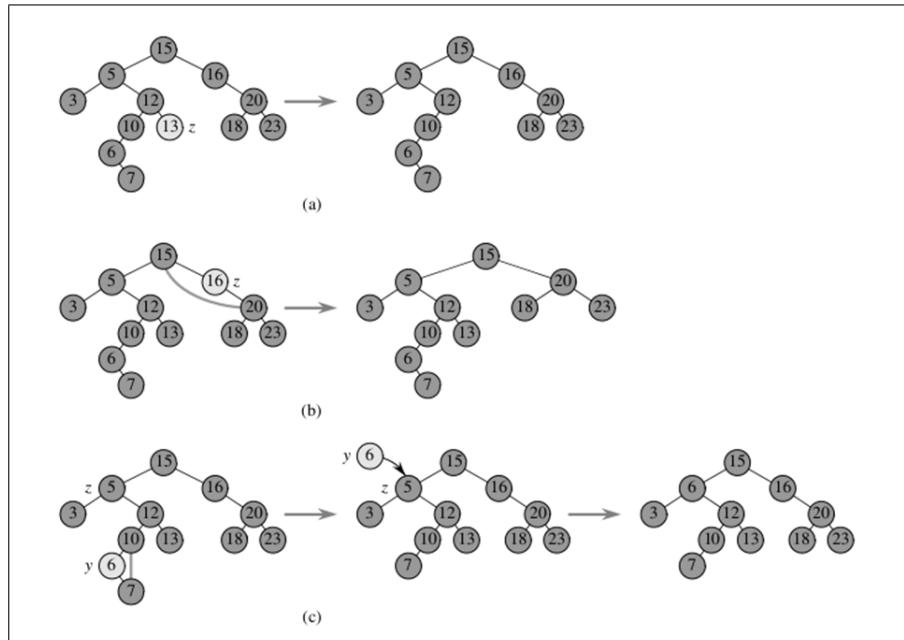


FIGURE 2.11 – La procédure permettant de supprimer un noeud donné z

Le code présenté dans la Figure 2.12 gère ces trois cas comme suit :

- Lignes 1–3 : l'algorithme détermine un noeud y à détacher. Le noeud y est soit le noeud d'entrée z (si z a au plus 1 enfant), soit le successeur de z (si z a deux enfants).
- Lignes 4–6 : x est rendu égal à l'enfant non-NIL de y ou à NIL si y n'a pas d'enfant.
- Lignes 7–13 : Le noeud y est détaché via modification des pointeurs

dans $p[y]$ et x .

- Lignes 14–16 : si le successeur de z était le noeud détaché, la clé et les données satellites de y sont déplacées dans z , écrasant alors la clé et les données satellites précédentes.

```

ARBRE-SUPPRIMER( $T, z$ )
1   si gauche[ $z$ ] = NIL ou droite[ $z$ ] = NIL
2     alors  $y \leftarrow z$ 
3     sinon  $y \leftarrow$  ARBRE-SUCCESEUR( $z$ )
4   si gauche[ $y$ ]  $\neq$  NIL
5     alors  $x \leftarrow$  gauche[ $y$ ]
6     sinon  $x \leftarrow$  droite[ $y$ ]
7   si  $x \neq$  NIL
8     alors  $p[x] \leftarrow p[y]$ 
9   si  $p[y] =$  NIL
10    alors racine[ $T$ ]  $\leftarrow x$ 
11    sinon si  $y =$  gauche[ $p[y]$ ]
12      alors gauche[ $p[y]$ ]  $\leftarrow x$ 
13      sinon droite[ $p[y]$ ]  $\leftarrow x$ 
14  si  $y \neq z$ 
15    alors clé[ $z$ ]  $\leftarrow$  clé[ $y$ ]
16    copier données satellites de  $y$  dans  $z$ 
17  retourner  $y$ 
```

FIGURE 2.12 – Le pseudo code de la procédure Suppression

On peut exécuter les opérations RECHERCHER, MINIMUM, MAXIMUM, SUCCESEUR, PRÉDÉCESSEUR, INSÉRER et SUPPRIMER en temps $O(h)$ sur un arbre binaire de recherche de hauteur h .

II Tas binaire

1. Définitions

Un tas peut être vu comme un tableau ou comme un arbre binaire partiellement complet.

Un tas est une structure de données qui :

- Permet un nouveau type de tri (Tri par tas)
- Permet l'implémentation de files de priorité
- Permet l'implémentation des méthodes de compression (Huffman)

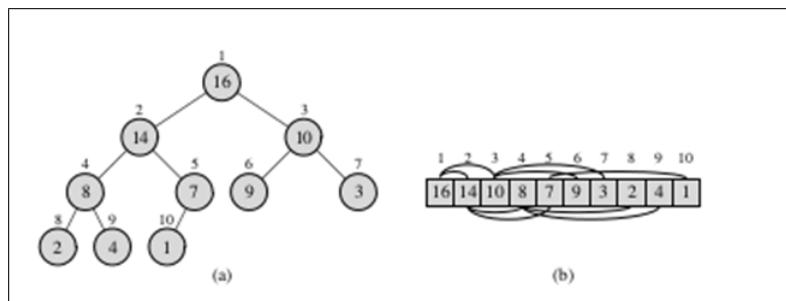


FIGURE 2.13 – Un tas-max vu comme (a) un arbre binaire et (b) un tableau

On représente un tas par un tableau A possédant 2 attributs (cf Figure 2.13) :

- $\text{Longueur}[A]$: nombre d’éléments du tableau
- $\text{Taille}[A]$: nombre d’éléments du tas rangés dans le tableau.

La racine de l’arbre est $A[1]$, et étant donné l’indice i d’un noeud, on calcule aisément les indices suivants :

- Parent(i) : retourner $\lfloor i/2 \rfloor$
- Gauche(i) : retourner $2i$
- Droite(i) : retourner $2i + 1$

On distingue deux types de tas :

- Tas max (binary maxheap) "tas", cf Figure 2.14
- $A[\text{Parent}(i)] \geq A[i]$: La valeur d’un noeud est au plus égale à celle de son parent
- La racine contient la valeur la plus grande et les valeurs sont conservées dans les nœuds internes
- Tas min (binary minheap), cf Figure 2.15
- $A[\text{Parent}(i)] \leq A[i]$: la valeur d’un noeud est au minimum égale à celle de son parent
- La racine contient la valeur la plus petite et les valeurs sont conservées dans les nœuds internes

Un tas de n noeuds a une hauteur $O(\log n)$ (cf. Figure 2.16). Les opérations proportionnelle à h sont $O(\log n)$:

- Soit h , la hauteur d’un tas de n noeuds
- Puisqu’il y a 2^i noeuds au niveau $i = 0, 1, 2, \dots, h - 2$ et au moins un noeud interne au niveau $h - 1$, on a $n \geq 1 + 2 + 4 + \dots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 (\sum_{i=0}^{n-1} 2^i = 2^n - 1)$

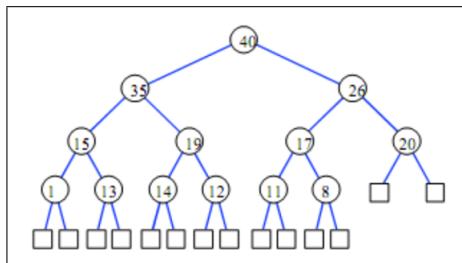


FIGURE 2.14 – Un tas-max

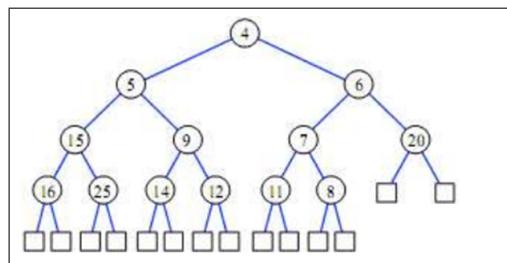


FIGURE 2.15 – Un tas-min

— Donc $n \geq 2^{h-1}$, càd $h \geq \log n + 1$

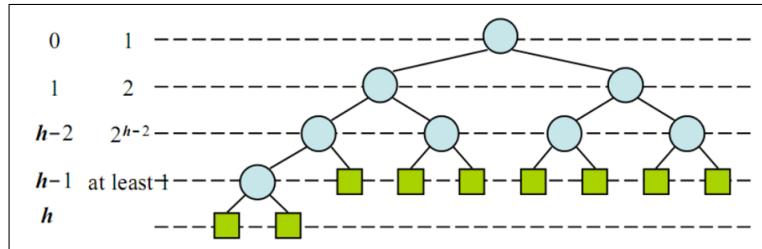


FIGURE 2.16 – Hauteur d'un tas

2. Conservation de la structure de tas

- *ENTASSER-MAX* est un sous-programme important pour la manipulation des tas max, qui prend en entrée un tableau A et un indice i .
- Quand *ENTASSER-MAX* est appelée, on suppose que les arbres binaires enracinés en $GAUCHE(i)$ et $DROITE(i)$ sont des tas max, mais

que $A[i]$ peut être plus petit que ses enfants, violant ainsi la propriété de tas max.

- Le rôle de *ENTASSER-MAX* est de faire « descendre » la valeur de $A[i]$ dans le tas max de manière que le sous-arbre enraciné en i devienne un tas max (cf. Figure 2.17).

```

ENTASSER-MAX( $A, i$ )
1    $l \leftarrow \text{GAUCHE}(i)$ 
2    $r \leftarrow \text{DROITE}(i)$ 
3   si  $l \leq \text{taille}[A]$  et  $A[l] > A[i]$ 
4     alors  $\text{max} \leftarrow l$ 
5   sinon  $\text{max} \leftarrow i$ 
6   si  $r \leq \text{taille}[A]$  et  $A[r] > A[\text{max}]$ 
7     alors  $\text{max} \leftarrow r$ 
8   si  $\text{max} \neq i$ 
9     alors échanger  $A[i] \leftrightarrow A[\text{max}]$ 
10    ENTASSER-MAX( $A, \text{max}$ )

```

FIGURE 2.17 – Le programme *ENTASSER-MAX*

La figure 2.18 illustre l'action de $\text{ENTASSER-MAX}(A, 2)$, où $\text{taille}[A] = 10$. La figure à gauche montre la configuration initiale du tas, avec la valeur $A[2]$ au noeud $i = 2$ viole la propriété de tas max puisqu'elle n'est pas supérieure à celle des deux enfants. La propriété de tas max est restaurée pour le noeud 2 dans la figure du milieu via échange de $A[2]$ avec $A[4]$, ce qui détruit la propriété de tas max pour le noeud 4. L'appel récursif $\text{ENTASSER-MAX}(A, 4)$ prend maintenant $i = 4$. Après avoir échangé $A[4]$ avec $A[9]$, comme illustré dans la figure à droite, le noeud 4 est corrigé, et l'appel récursif $\text{ENTASSER-MAX}(A, 9)$ n'engendre plus de modifications de la structure de données.

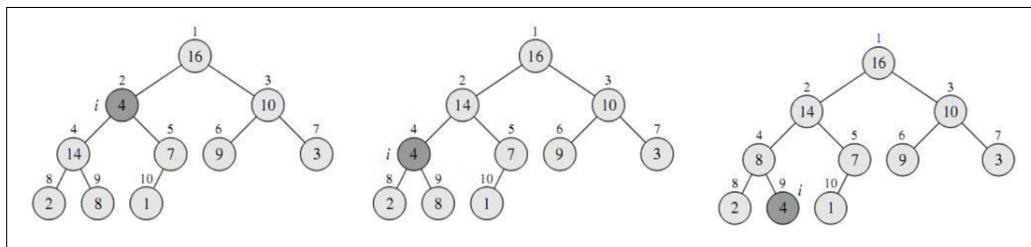


FIGURE 2.18 – L'action *ENTASSER-MAX*($A, 2$)

Le temps d'exécution de *ENTASSER-MAX* sur un noeud de hauteur h par $O(h)$.

3. Construction du tas

On peut utiliser la procédure *ENTASSER-MAX* à l'envers pour convertir un tableau $A[1..n]$, avec $n = \text{longueur}[A]$. Les éléments du sous-tableau $A[(\lfloor n/2 \rfloor + 1)..n]$ sont les feuilles de l'arbre. Chacun d'eux est donc un tas à 1 élément.

La procédure *CONSTRUIRE-TAS-MAX* parcourt les autres nœuds de l'arbre et appelle *ENTASSER-MAX* pour chacun comme le montre la Figure 2.19.

La Figure 2.20 montre le fonctionnement de *CONSTRUIRE-TAS-MAX* sur un tableau de 10 éléments.

```

CONSTRUIRE-TAS-MAX(A)
1   taille[A] ← longueur[A]
2   pour  $i \leftarrow \lfloor \text{longueur}[A]/2 \rfloor$  jusqu'à 1
3       faire ENTASSER-MAX(A, i)

```

FIGURE 2.19 – La procédure *CONSTRUIRE-TAS-MAX*

La complexité de *CONSTRUIRE-TAS-MAX* est en $O(n)$. La complexité dépendante de *Entasser-Max* avec $h[A]$, c'est-à-dire elle est en fonction de nombre d'opérations d'échange (cf Figure 2.21) :

- Pour le niveau 0, pour un nœud, h échanges (au max)
- Pour le niveau 1, pour un nœud, $h - 1$ échanges (au max)
- ...
- Pour le niveau i , pour un nœud, $h - i$ échanges (au max)

On peut calculer un majorant simple pour le temps d'exécution de *CONSTRUIRE-TAS-MAX* de la manière suivante :

- Au niveau i , il y a 2^i nœuds
- Le nombre total d'échanges pour ces 2^i nœuds est donc majoré par $2^i(h - i)$
- Le nombre total d'échanges pour tous les nœuds est donc majoré par $\sum_{i=0}^h 2^i(h - i) = \sum_{j=0}^h 2^{h-j}j = 2^h \sum_{j=0}^h 2^{-j}j$, en posant $j = h - i$
- Or, $\sum_{j=0}^h 2^{-j}j \leq 2$, donc $2^h \sum_{j=0}^h 2^{-j}j \leq 2^h \cdot 2 = 2n \rightarrow O(n)$

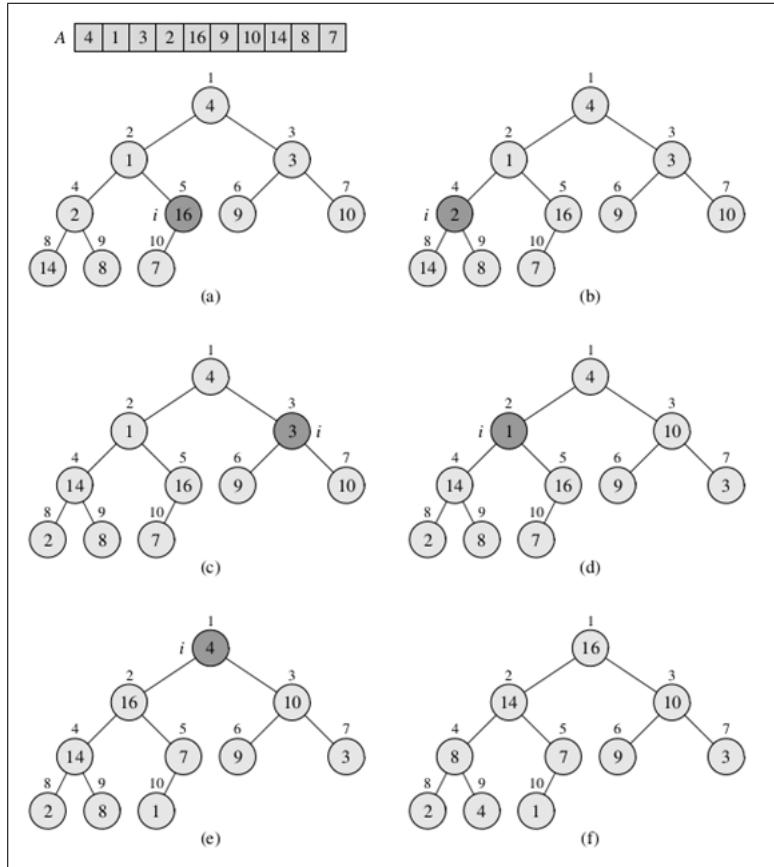
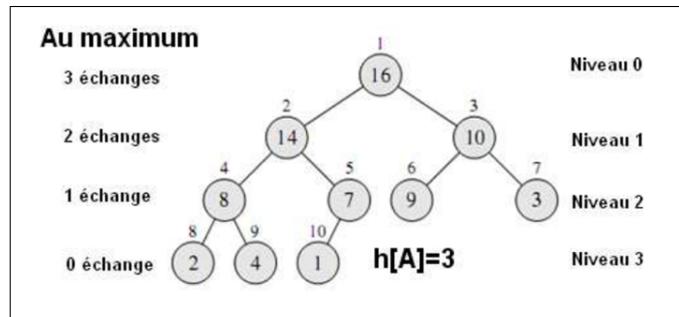
FIGURE 2.20 – *CONSTRUIRE-TAS-MAX* sur un tableau de 10 éléments

FIGURE 2.21 – Nombre d'opérations d'échange par niveau

4. Applications de Tas

La structure de données "Tas" peut être utilisée dans de nombreuses applications. Nous allons voir son application pour le tri par tas et pour construire une file de priorités.

a. Tri par Tas

Le principe de l'algorithme (cf Figure 2.22) du tris par tas est le suivant :

- Construction d'un tas max à partir d'un tableau $A[1..n]$ où $n = \text{longueur}[A]$
- Ensuite, $A[1]$ contenant l'élément de valeur la plus élevée, on peut l'échanger avec $A[n]$
- On retire le noeud "n" du tas ($\text{Taille}[A] - 1$)
- On réorganise $A[1..(n - 1)]$ pour qu'il corresponde à un tas max.
- On répète l'opération jusqu'à ce que le tas ait une taille de 1

```

TRI-PAR-TAS( $A$ )
1 CONSTRUIRE-TAS-MAX( $A$ )
2 pour  $i \leftarrow \text{longueur}[A]$  jusqu'à 2
3   faire échanger  $A[1] \leftrightarrow A[i]$ 
4    $\text{taille}[A] \leftarrow \text{taille}[A] - 1$ 
5   ENTASSER-MAX( $A, 1$ )

```

FIGURE 2.22 – La procédure Tri par Tas

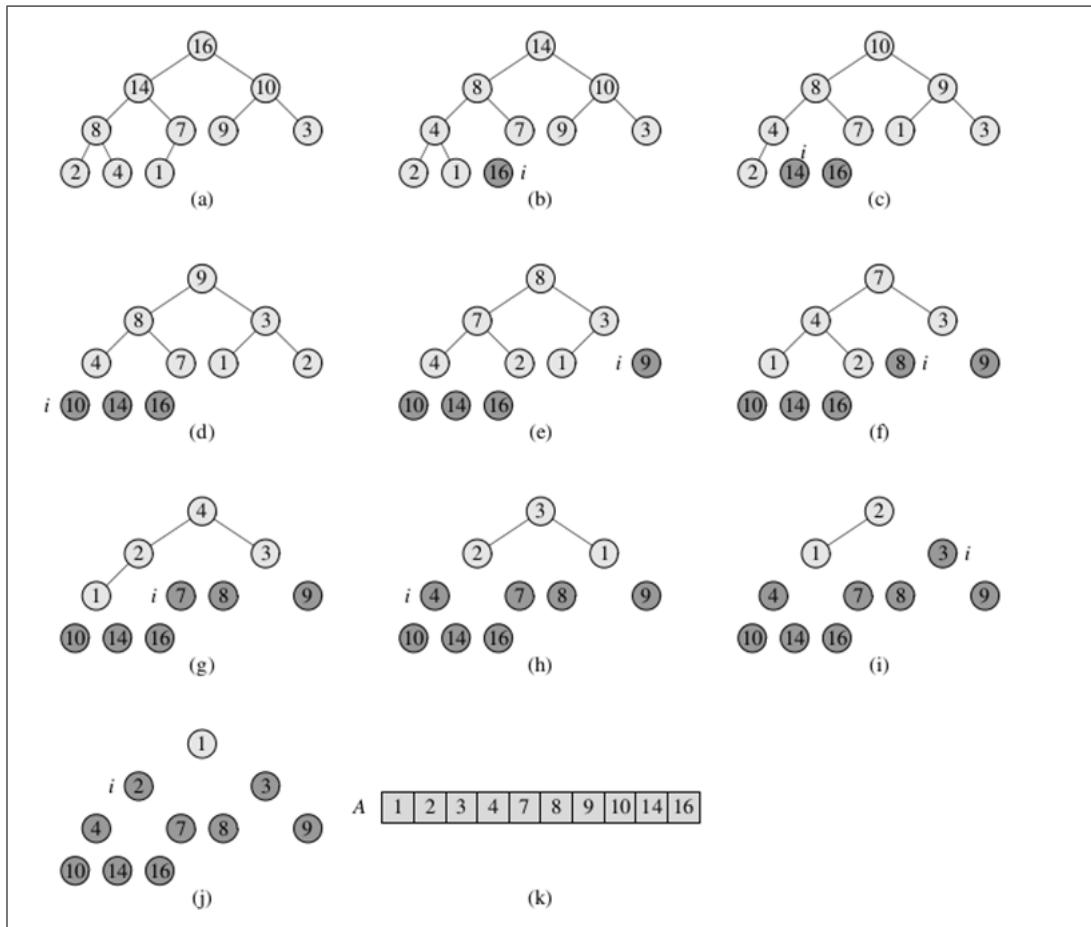
La Figure 2.23(a) illustre la structure de tas max juste après sa construction. (b)–(j) illustrent le tas max juste après chaque appel *ENTASSER-MAX* en ligne 5. La valeur de i à ce moment est montrée. Seul les noeuds légèrement ombrés restent dans le tas. (k) Le résultat du tri sur le tableau A .

La complexité du Tri par Tas est $O(n \log n)$:

- Appel à Construire-Tas-Max : $O(n)$
 - Dans la boucle "pour" il y a n appels à Entasser-Max ayant une complexité $O(\log n)$
- : Trier (9, 2, 11, 7, 4, 14, 3, 16, 8, 10, 15)

b. File de priorité

Une file de priorité est une structure de données permettant de gérer un ensemble S d'éléments auxquels on associe une "clé" qui permet la définition

FIGURE 2.23 – Exemple de l'action *TRI-PAR-TAS*

des priorités.

- Application directe de la structure de tas avec les opérations (File-Max) : Maximum(S), Extraire-Max(S), Augmenter-Clé(S,x,k), Insérer(S,x). Si on inverse l'ordre des priorités, on obtient les opérations symétriques (File-Min) : Minimum, Extraire-Min, Diminuer-Clé
- Exemple : une liste de tâches sur un ordinateur. Dans ce cas, la file de priorité gère les tâches en attente selon leur priorité. Quand une tâche est terminée, on exécute la tâche de priorité la plus élevée suivante.

La Figure 2.24 montre la procédure *MAXIMUM-TAS* qui implémente l'opération *MAXIMUM* en un temps $\Theta(1)$, i.e., retourne l'élément ayant la clé maximale. La procédure *EXTRAIRE-MAX-TAS* permet de retourner l'élément ayant la clé maximale en le supprimant de la file (cf. Figure 2.25).

```
MAXIMUM-TAS(A)
1   retourner A[1]
```

FIGURE 2.24 – La procédure MAXIMUM-TAS

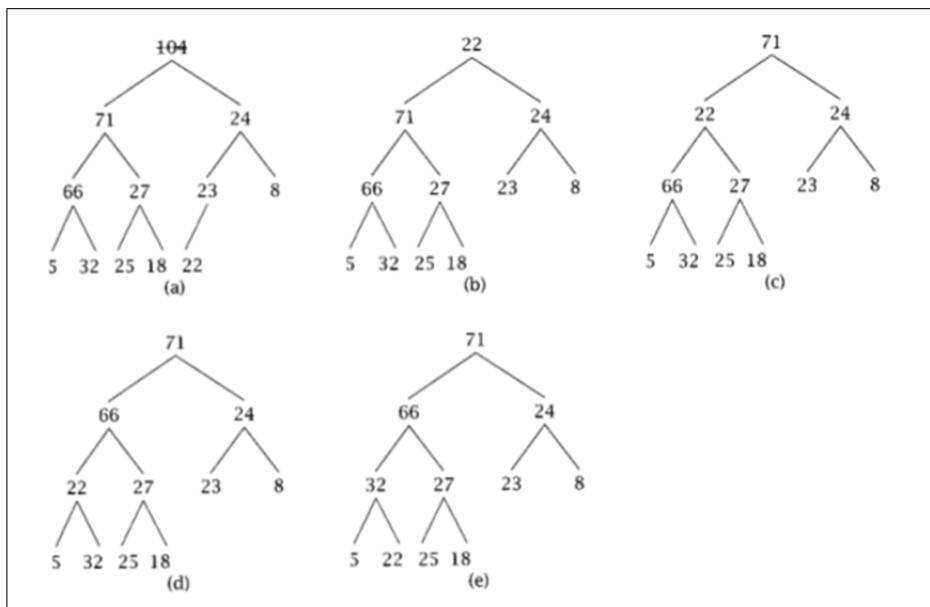
```
EXTRAIRE-MAX-TAS(A)
1   si taille[A] < 1
2     alors erreur « limite inférieure dépassée »
3   max ← A[1]
4   A[1] ← A[taille[A]]
5   taille[A] ← taille[A] - 1
6   ENTASSER-MAX(A, 1)
7   retourner max
```

FIGURE 2.25 – La procédure EXTRAIRE-MAX-TAS

La figure 2.26 illustre un exemple de l'action *EXTRAIRE-MAX-TAS*, i.e., la reconstruction du tas en déplaçant le noeud courant de haut en bas

- Echange avec le fils de clé maximale (Entasser-Max)
- Arrêt quand feuille ou clé supérieure à celles des 2 fils.

Le temps d'exécution de *EXTRAIRE-MAX-TAS* est $O(\log n)$, car elle n'effectue qu'un volume constant de travail en plus du temps $O(\lg n)$ de *ENTASSER-MAX*.

FIGURE 2.26 – Exemple de l'action *EXTRAIRE-MAX-TAS*

La procédure *AUGMENTER-CLÉ-TAS* (cf Figure 2.27) implémente l'opération AUGMENTER-CLÉ. L'élément de la file de priorité dont il faut accroître la clé est identifié par un indice i pointant vers le tableau.

La figure 2.28 illustre un exemple de fonctionnement de *AUGMENTER-CLÉ-TAS*. (a) Le tas max de la figure 2.28(a) avec un noeud dont l'indice est i en gris foncé. (b) Ce noeud voit sa clé passer à 15. (c) Après une itération de la boucle tant que des lignes 4–6, le noeud et son parent s'échangent leurs clés et l'indice i remonte pour devenir celui du parent. (d) Le tas max après une itération supplémentaire de la boucle tant que. À ce stade, $A[PARENT(i)] \geq A[i]$. La propriété de tas max étant maintenant vérifiée, la procédure se termine.

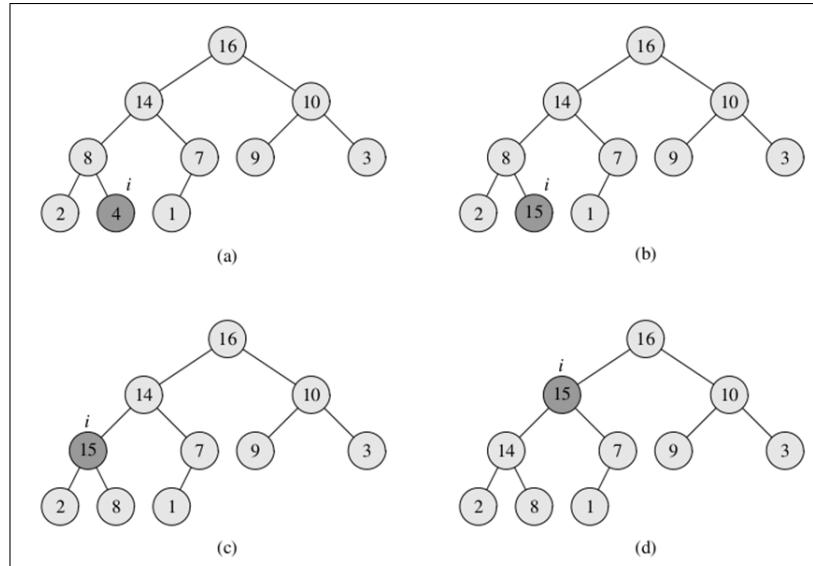
Le temps d'exécution de *AUGMENTER-CLÉ-TAS* sur un tas à n éléments est $O(\log n)$, car le chemin reliant le noeud, modifié en ligne 3, à la racine a la longueur $O(\lg n)$.

La procédure *INSÉRER-TAS-MAX* implémente l'opération *INSÉRER*. Elle prend en entrée la clé du nouvel élément à insérer dans le tas max A (cf. Figure 2.29). Le temps d'exécution de *INSÉRER-TAS-MAX* sur un tas à n

```

AUGMENTER-CLÉ-TAS( $A, i, clé$ )
1 si  $clé < A[i]$ 
2   alors erreur « nouvelle clé plus petite que clé actuelle »
3    $A[i] \leftarrow clé$ 
4   tant que  $i > 1$  et  $A[\text{PARENT}(i)] < A[i]$ 
5     faire permute  $A[i] \leftrightarrow A[\text{PARENT}(i)]$ 
6      $i \leftarrow \text{PARENT}(i)$ 

```

FIGURE 2.27 – La procédure *AUGMENTER-CLÉ-TAS*FIGURE 2.28 – Exemple de l'action *AUGMENTER-CLÉ-TAS*

éléments est $O(lgn)$. En résumé, un tas permet de faire toutes les opérations de file de priorité sur un ensemble de taille n en un temps $O(lgn)$.

```

INSÉRER-TAS-MAX( $A, clé$ )
1  $taille[A] \leftarrow taille[A] + 1$ 
2  $A[taille[A]] \leftarrow -\infty$ 
3 AUGMENTER-CLÉ-TAS( $A, taille[A], clé$ )

```

FIGURE 2.29 – La procédure *INSÉRER-TAS-MAX*

La figure 2.30 illustre un exemple de fonctionnement de la procédure *AUGMENTER-CLÉ-TAS*.

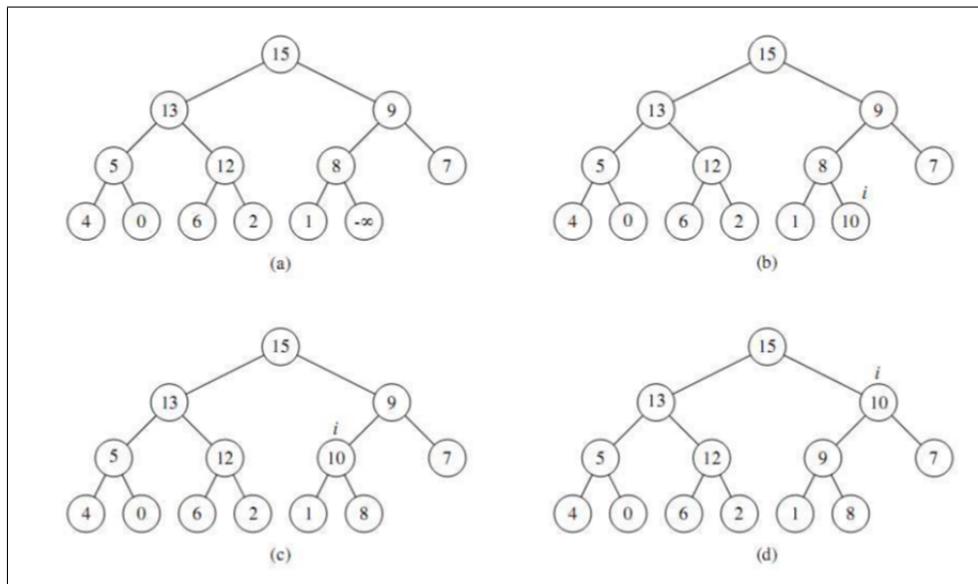


FIGURE 2.30 – Exemple de l'action *INSÉRER-TAS-MAX*

c. Codage de Huffman

La compression des données est un problème algorithmique aussi important que le tri. En fait, il est souvent pratique de compacter des données afin de sauver de l'espace sur disque et accélérer la vitesse de transfert des informations le cas échéant.

Par exemple, dans le code standard ASCII, chaque caractère est représenté par 8 bits. Il faut un nombre minimum de bits pour représenter de façon unique chaque caractère. Cette méthode serait la plus efficace si tous les caractères étaient utilisés de façon équitable.

Une autre approche consiste à trouver des codes de longueur variable. Cette codification tient compte de la fréquence d'apparition d'un caractère. Il est évident que les caractères les plus fréquemment utilisés devraient avoir des codes plus petits. Cette approche est utilisée par Huffman pour la compression de fichiers.

La méthode de compression Huffman consiste à diminuer au maximum le nombre de bits utilisés pour coder un fragment d'information par la création de l'arbre de Huffman :

- Création de la liste des fréquences d'apparition des fragments
- Trier la liste par ordre croissant de fréquences
- Construire l'arbre à partir de la liste ordonnée de noeuds

Soit un alphabet C de n caractères dont chaque caractère a une fréquence $f[c]$, l'algorithme de Huffman se déroule comme illustré à la figure 2.33.

- A chaque étape, les deux arborescences ayant les fréquences les plus basses sont fusionnés.
- Les feuilles sont représentées par des rectangles contenant un caractère et sa fréquence.
- Les noeuds internes sont représentés par des cercles contenant la somme des fréquences des enfants.
- Un arc reliant un nœud interne à ses enfants est étiqueté 0 si c'est un arc vers un enfant gauche, et 1 si c'est un arc vers un enfant droit.
- Le mot de code pour une lettre est la séquence d'étiquettes des arcs du chemin reliant la racine à la feuille correspondant à cette lettre.

L'algorithme est présenté dans la Figure 2.32. Plus précisément, le processus est décrit ci-dessous :

- Créez un noeud feuille pour chaque symbole et ajoutez-le à la file d'attente prioritaire.
- Bien qu'il y ait plus d'un noeud dans la file d'attente :
 - Supprimez les deux noeuds les plus prioritaires de la file d'attente.
 - Créez un nouveau noeud interne avec ces deux noeuds en tant qu'enfants et avec une fréquence égale à la somme de la fréquence des deux noeuds.
 - Ajoutez le nouveau noeud à la file d'attente.
- Le noeud restant est le noeud racine et l'arbre Huffman est complet.

Une fois l'arbre construit, toutes les branches gauches seront étiquetées 0 et les branches droites 1. Le code d'un caractère est obtenu en concaténant les 0 et les 1 rencontrée en partant de la racine de l'arbre à la feuille qui contient la fréquence du caractère en question.

Le décodage d'un message, codé avec l'algorithme de Huffman, est effectué en regardant à la chaîne de bits de gauche vers la droite jusqu'à ce qu'une lettre soit décodé. Cela pourrait être fait en utilisant l'arbre de Huffman en commençant par la racine. Dépendant de la valeur du bit, gauche pour 0 ou droite pour 1, on suit les branches de l'arbre jusqu'à arriver à une feuille. La feuille contient le premier caractère du message. On recommence ensuite ce processus pour les autres caractères du message.

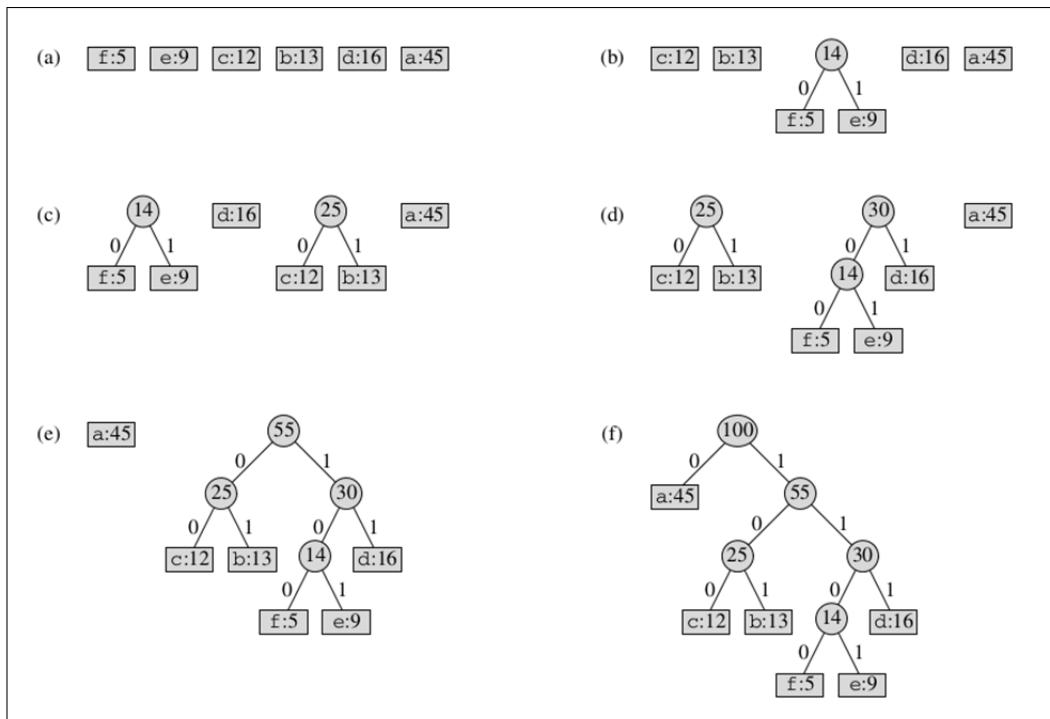


FIGURE 2.31 – Exemple de construction de l'arbre de Huffman

Exemple 1 : Appliquer le même processus sur le texte " axabataaxbcb-taabxxaaazazaa".

- 011001110101100110111101011110110011111011000100010000 (54 bits)
- 01100111-01011001-10111101-01111011-00111110-11000100-010000 (7 octets)

```

HUFFMAN(C)
1    $n \leftarrow |C|$ 
2    $Q \leftarrow C$ 
3   pour  $i \leftarrow 1$  à  $n - 1$ 
4     faire allouer un nouveau nœud  $z$ 
5      $gauche[z] \leftarrow x \leftarrow EXTRAIRE-MIN(Q)$ 
6      $droite[z] \leftarrow y \leftarrow EXTRAIRE-MIN(Q)$ 
7      $f[z] \leftarrow f[x] + f[y]$ 
8     INSÉRER( $Q, z$ )
9   retourner EXTRAIRE-MIN( $Q$ )     $\triangleright$  Retourner la racine de l'arborescence.

```

FIGURE 2.32 – La procédure de construction de l'arbre

Exemple 2 : Appliquer le même processus sur l'exemple suivant :

Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

Exemple 3 : Calculer l'arbre de Huffmann associé au message "ABACF-CAEDBAAEDAF"

III Travaux dirigés

Exercice 1 : ABR

Dessinez les arbres binaires de recherche de hauteur 2,3,4,5 et 6 pour le même ensemble de clés $S = 1, 4, 5, 10, 16, 17, 21$.

Dessins des ABR avec hauteurs différentes. On numérote les niveaux à partir de 0.

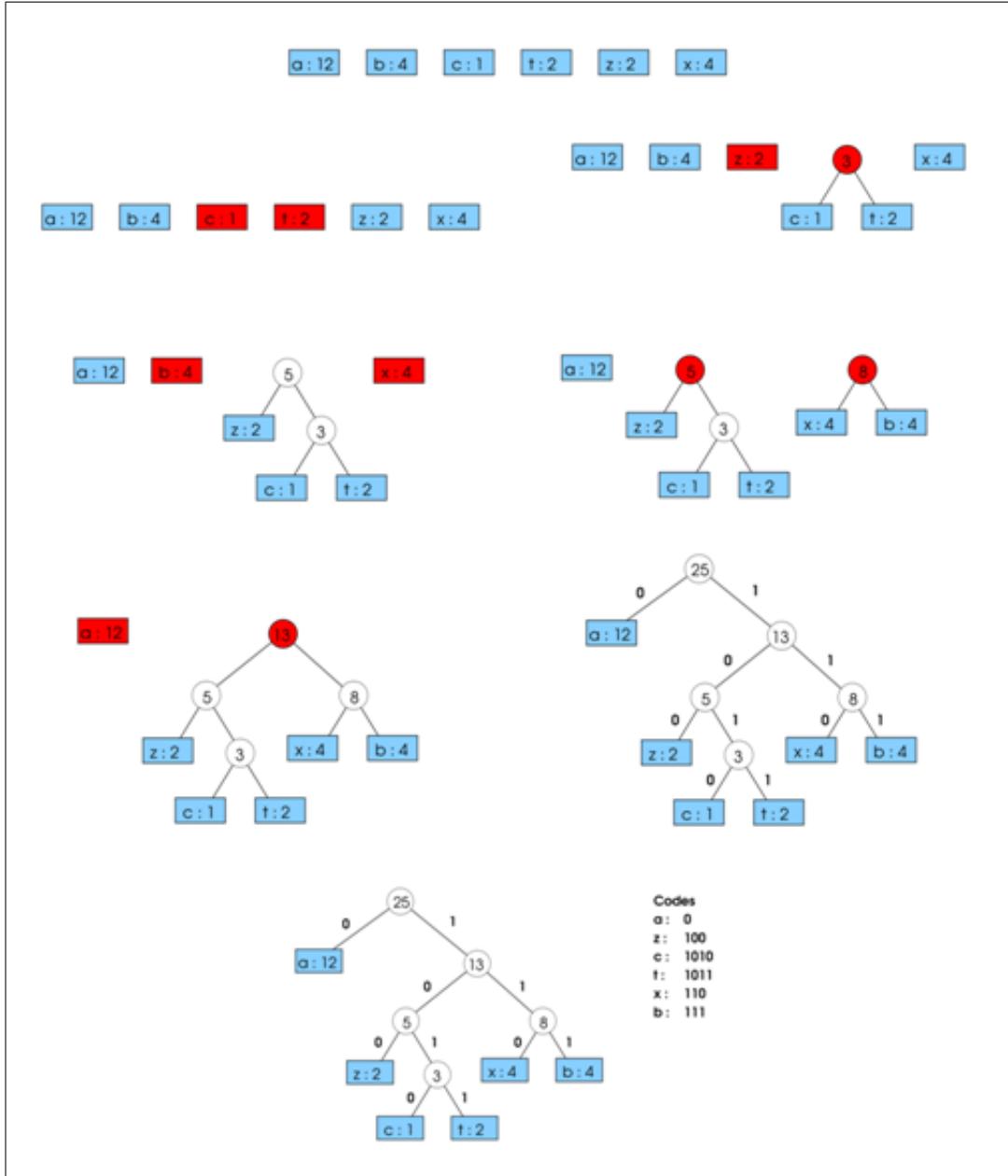


FIGURE 2.33 – Exemple 1 :"axabataaxbcctaabbxxaaazazaa"

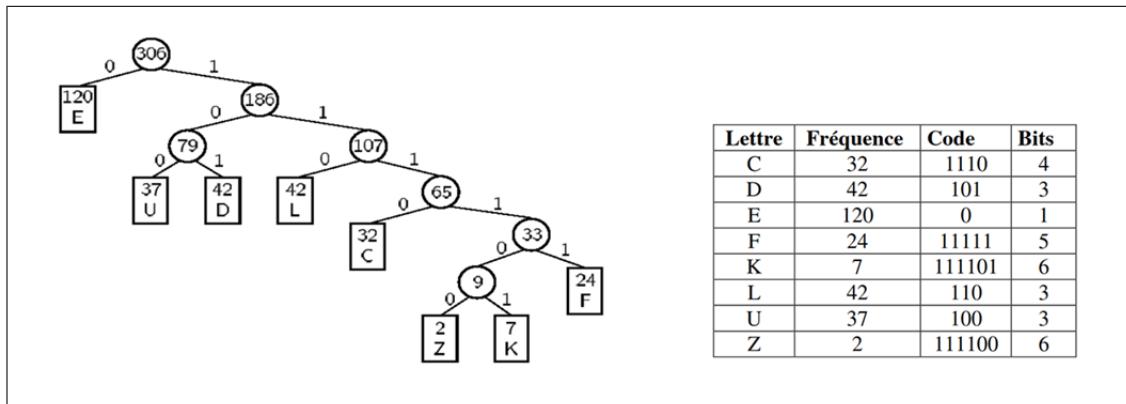
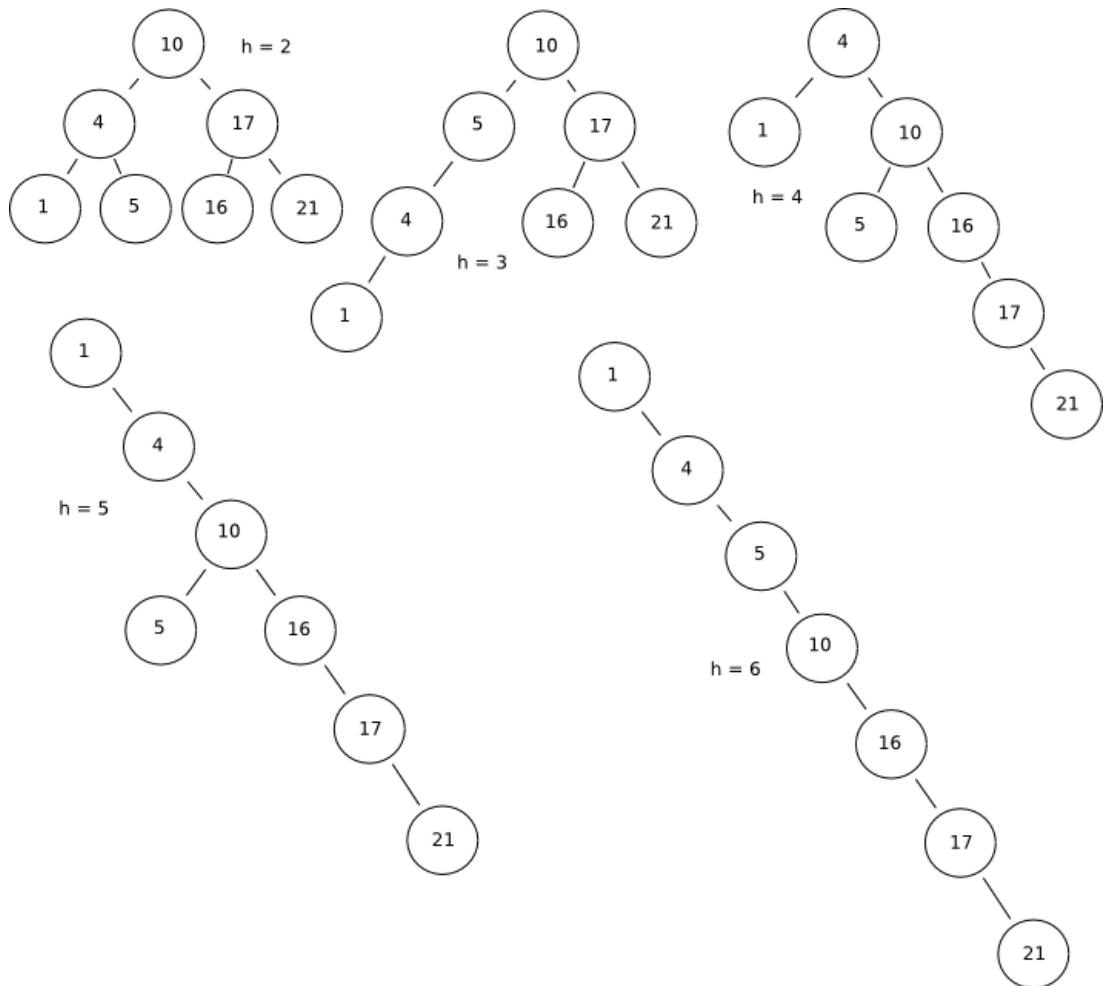
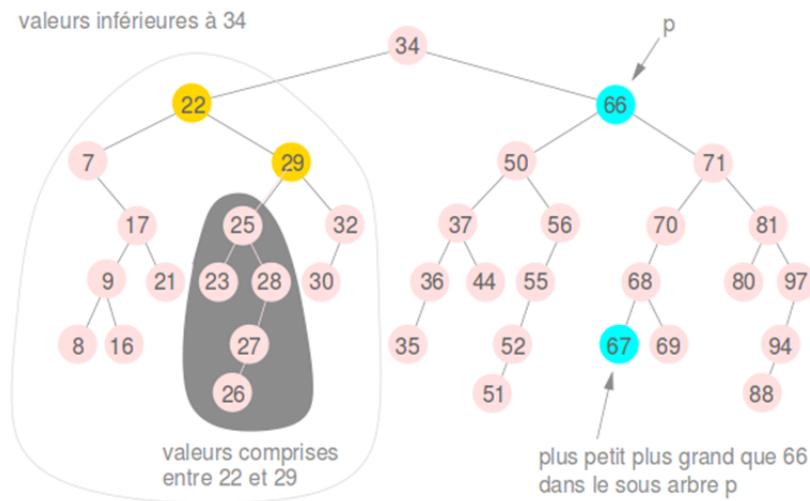


FIGURE 2.34 – Exemple 2



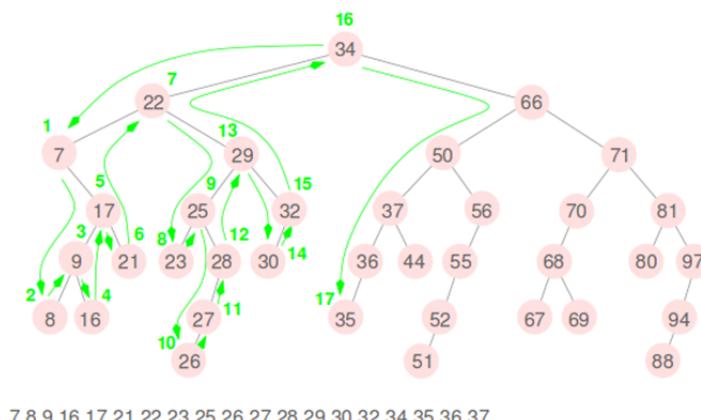
Exercice 2 : Recherche/successeur d'un noeud dans un ABR

Soint l'arbre ci-dessous :



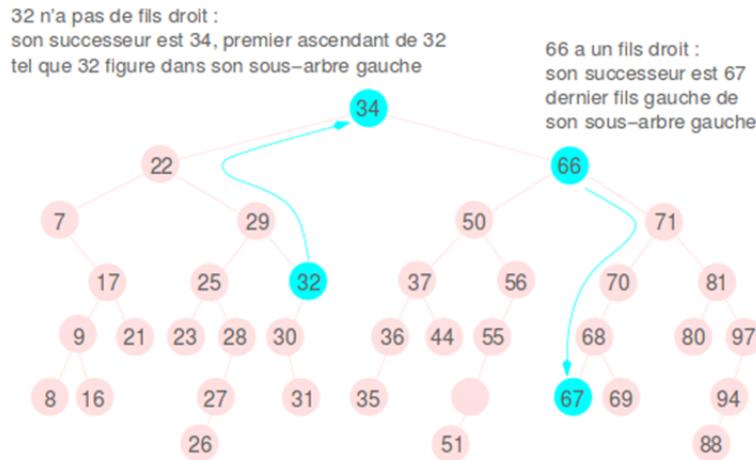
- Donner le résultat de parcours infixé de l'arbre.
- Donner le principe de la recherche d'une valeur dans ABR et illustrer le chemin suivi pour la recherche de neoud ayant la clé 27.
- Quels sont les noeuds successeurs des noeuds 32 et 66.

Question 1 : le résultat de parcours infixé de l'arbre : "7 8 9 16 17 21 22 23 25 26 27 28 29 30 32 34 35 36 37 ..."



Question 2 : 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. Le chemin suivi est : 34->22->29->25->28

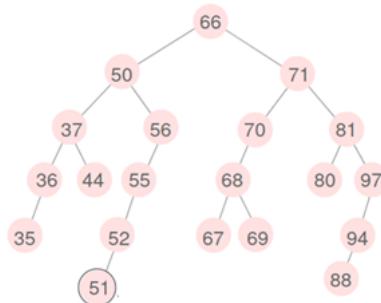
Question 3 : le successeur de noeud 32 est 34 et le successeur de noeud 66 est 67.



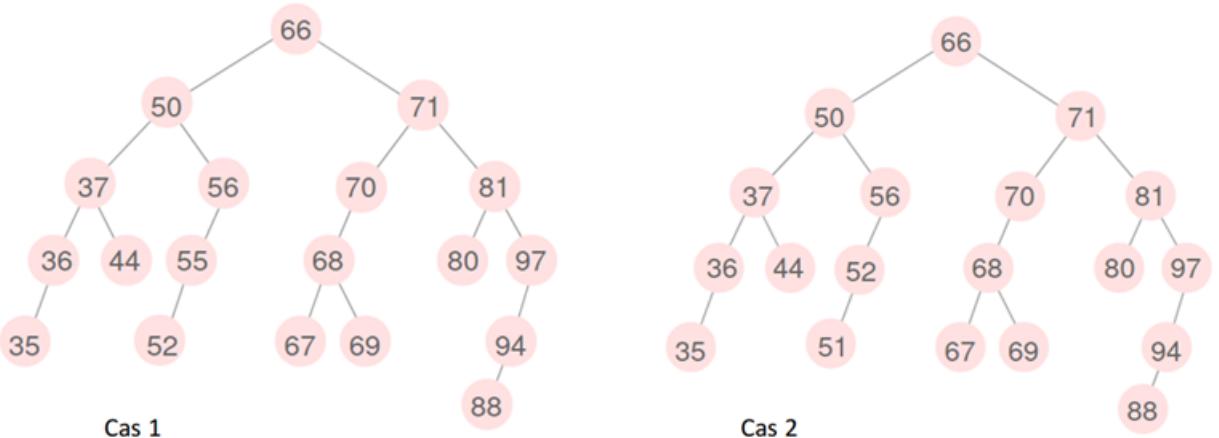
Exercice 3 : Suppression d'un noeud d'un ABR

La suppression d'un noeud dépend du nombre de fils du noeud à supprimer.
Supprimer les noeuds suivants de l'arbre de la Figure suivant :

- suppression du noeud 51
- suppression du noeud 55



- **Cas 1** : le noeud à supprimer 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 comme dans le cas 1. Il est remplacé par son fils unique dans le noeud père, si ce père existe. Sinon l'arbre est réduit au fils unique du noeud supprimé.

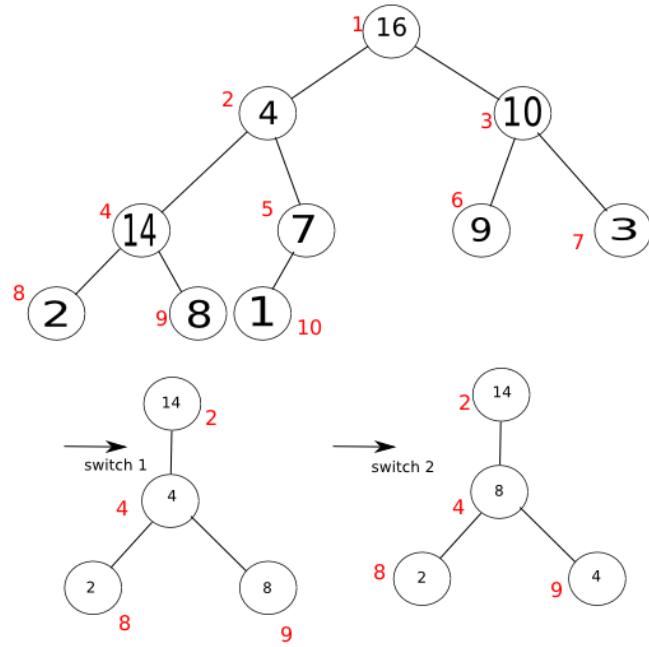


Exercice 4 : Construction d'un Tas

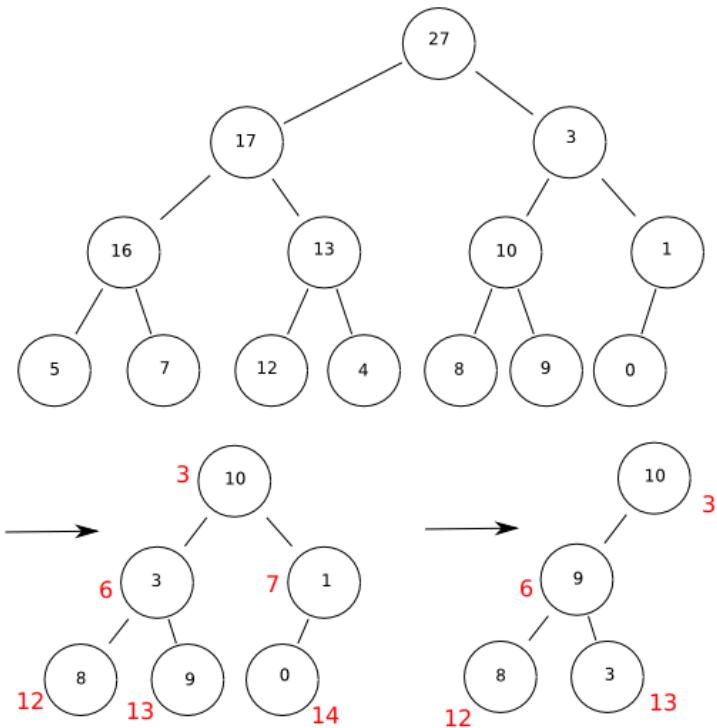
Soit un tableau A , et un indice i , $A[i]$ est la clé associée au $i^{\text{ième}}$ élément de A . Quand la fonction *EntasserMax* est appelée, on suppose que les arbres binaires enracinés en *Gauche*(i) et *Droite*(i) sont des tas max. La fonction *EntasserMax*(A, i) consiste à faire "descendre" la valeur $A[i]$ dans le tas max de manière que le sous-arbre enraciné en i devienne un tas max.

1. Illustrer l'action de *EntasserMax*($A, 2$), avec $A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$.
2. Idem pour *EntasserMax*($A, 3$) avec $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$.

Applications $A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$



$$A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$$

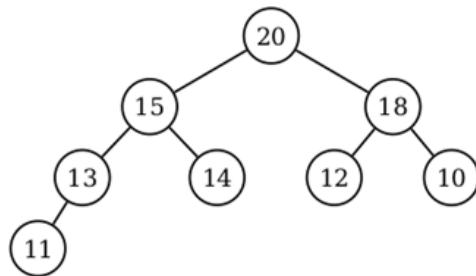


Exercice 5 : Construction/Insertion/Extraction des noeuds d'un Tas

Dans la construction d'un tas, la fonction EntasserMax est utilisée à l'envers pour convertir un tableau $A[1, \dots, n]$, avec $n = \text{longueur}(A)$, en tas max.

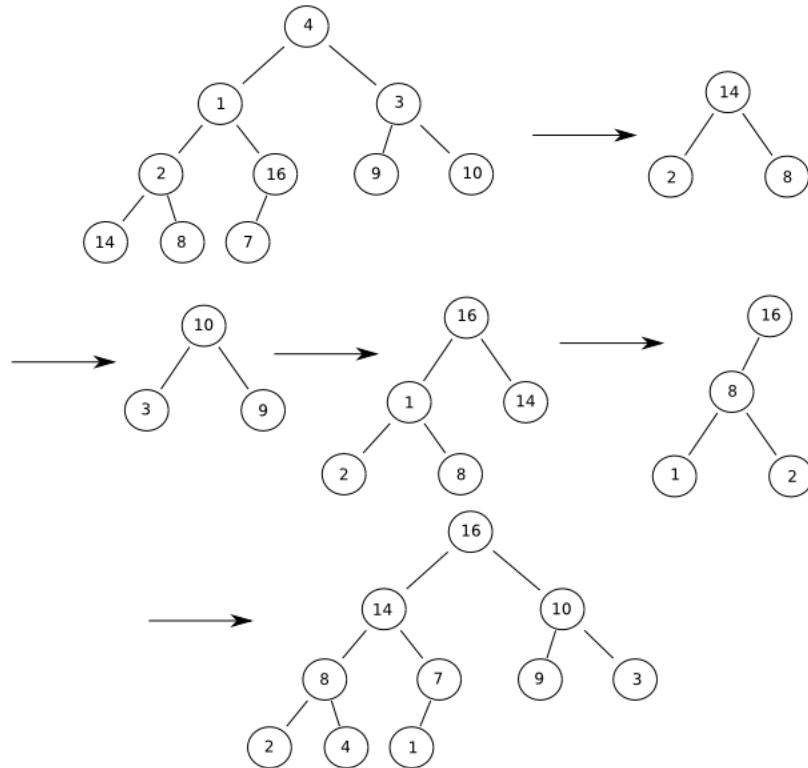
Supposant que dans un arbre binaire, qui représente un tableau à n éléments, les feuilles sont les noeuds indexés par $[n/2] + 1, [n/2] + 2, \dots, n$. Ces feuilles sont donc des tas à 1 élément.

Exemple 1 : On se donne le tas max de la figure suivante :



1. Insérer un élément de clé 17 dans ce tas. Donner le nouveau tas.
2. En repartant du tas initial, retirer l'élément de clé maximale.

Exemple 2 : on utilise le tableau suivant : $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$, pour trouver le tas Max suivant.



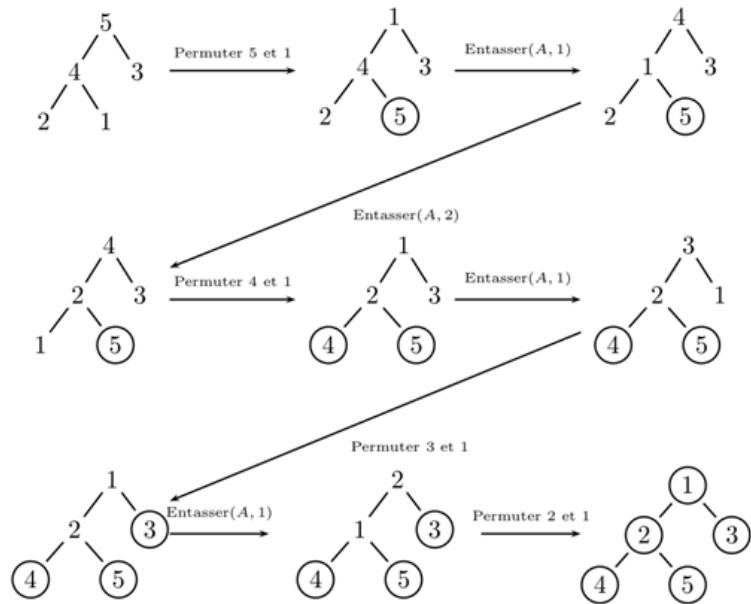
1. Illuster l'action d'insertion de la clé 15.
2. Illuster l'action d'extraction du maximum, en repartant du tas initial

Exercice 6 : Tris par Tas

Le principe du tri par tas est comme suit :

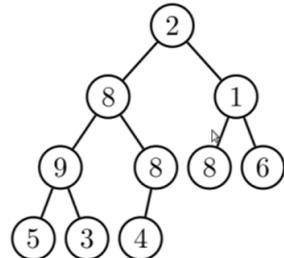
- **Etape 1** : On regarde le tableau comme un tas.
- **Etape 2** : On forme le tas en faisant appel à la fonction *EntasserMax* sur chaque noeud interne en allant du dernier au premier.
- **Etape 3** : On extrait l'élément maximum (à la racine) en l'échangeant avec le dernier élément du tableau (la dernière feuille). On sort ce dernier élément de l'arbre et on reforme le tas. On recommence cette étape avec le nouveau maximum jusqu'au dernier élément du tas.

Les trois étapes sont illustrées dans l'exemple suivant :

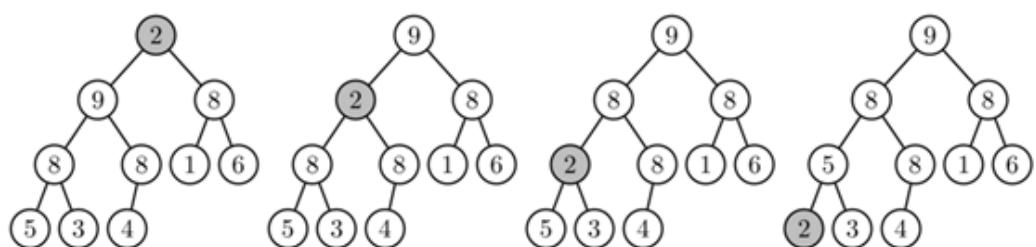
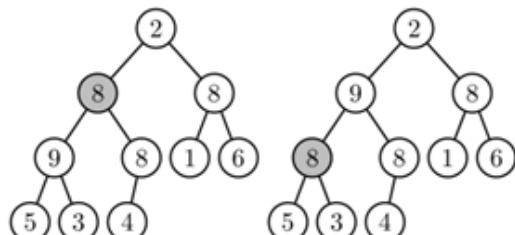
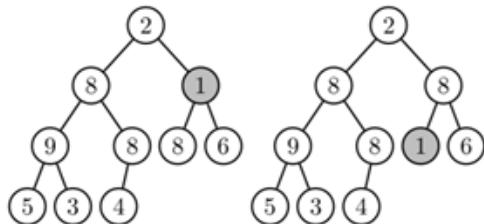
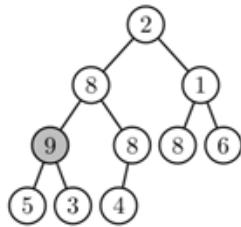
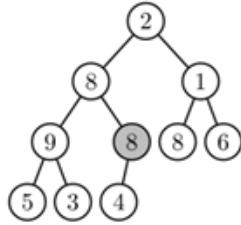


Exemple 1 : Donner le résultat de l'algorithme du tri sur le tableau suivant : [2 8 1 9 8 8 6 5 3 4].

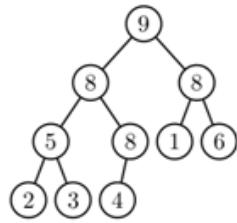
Etape 1 : On regarde le tableau comme un tas comme suit :



Etape 2 : On forme le tas en faisant appel à la fonction *EntasserMax* :



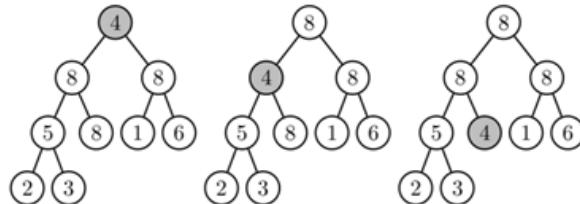
On obtient le tas suivant :



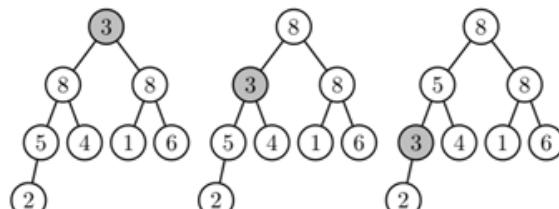
Le tableau est maintenant : [9 8 8 5 8 1 6 2 3 4].

Etape 3 : On extrait l'élément maximum en l'échangeant avec le dernier élément du tableau.

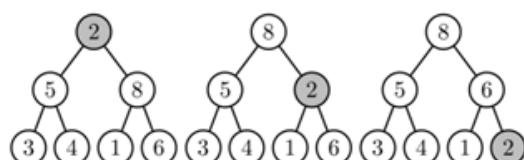
Après extraction du 9, le tableau est maintenant : [8 8 8 5 4 1 6 2 3 9], en gras, les éléments encore dans l'arbre.



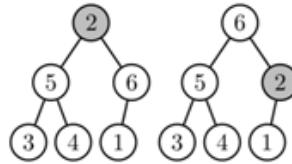
On recommence avec le nouveau maximum (ici 8), le tableau est maintenant : [8 5 8 3 4 1 6 2 8 9]



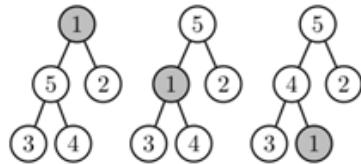
On recommence avec le nouveau maximum (ici 8), le tableau est maintenant : [8 5 6 3 4 1 2 8 8 9]



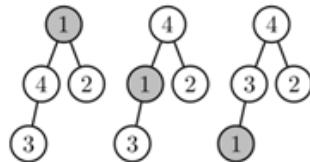
On recommence avec le nouveau maximum (ici 8), le tableau est maintenant :
[6 5 2 3 4 1 8 8 8 9]



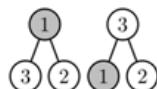
On recommence avec le nouveau maximum (ici 6), le tableau est maintenant :
[5 4 2 3 1 6 8 8 8 9]



On recommence avec le nouveau maximum (ici 5), le tableau est maintenant :
[4 3 2 1 5 6 8 8 8 9]



On recommence avec le nouveau maximum (ici 4), le tableau est maintenant :
[3 1 2 4 5 6 8 8 8 9]



On recommence avec le nouveau maximum (ici 3), le tableau est maintenant :
[2 1 3 4 5 6 8 8 8 9]

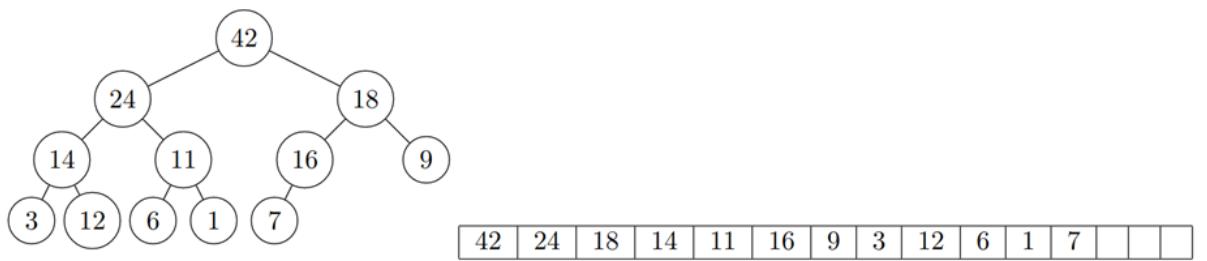


On recommence avec le nouveau maximum (ici 2), le tableau est maintenant :
[1 2 3 4 5 6 8 8 8 9]

(1)

On recommence avec le dernier maximum (ici 1), le tableau est maintenant trié : [1 2 3 4 5 6 8 8 8 9]

Exemple 2 : Donner le résultat de l'algorithme du tri sur le tas suivant en utilisant la même démarche utilisée dans l'exemple ci-dessus.



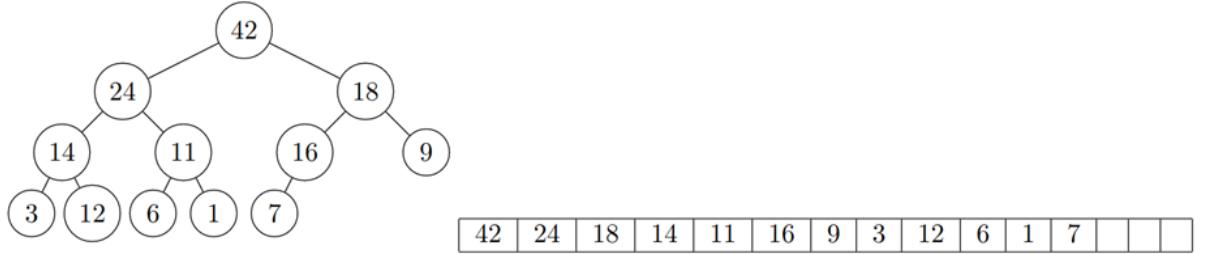
Exemple 3 : Donner le résultat de l'algorithme du tri sur le tableau [12 15 26 30 10 80 29 31 23 45] en utilisant la même démarche utilisée dans l'exemple ci-dessus.

Exercice 7 : Files de priorité

Une file de priorité est une structure de données qui permet de gérer un ensemble E d'éléments, dont chacun a une valeur clé associée. Par exemple, supposons que des gens se présentent au guichet (un seul guichet) d'une banque avec un numéro représentant leur degré de priorité. L'employé de la banque doit traiter rapidement tous ses clients selon leurs degrés de priorité. Une file de priorité max reconnaît les trois opérations suivantes : Inserer(E, x), Maximum(E), ExtraireMax(E). En fait, l'employé de la banque doit donc savoir faire rapidement les trois opérations suivantes : trouver un maximum dans la file de priorité, retirer cet élément de la file, ajouter un nouvel élément à la file.

Illustrer l'opération d'insertion d'une nouvelle personne qui arrive au guichet avec son numéro de priorité 12.

Dérouler l'algorithme permettant de supprimer le max du Tas.



Donnez l'algorithme correspondant à chaque opération. Donnez la complexité associée à chaque algorithme.

Texte des algorithmes

```
Maximum(A)
    retourner A[1]
```

```
ExtraireMax(A)
    Si taille(A) <= 1 Alors
        retourner ERREUR
    Sinon
        max <- A[1]
        A[1] <- A[taille(A)]
        taille(A) <- taille(A) - 1
        EntasserMax(A, i)
        retourner max
    Finsi
Fin
```

```
InsererTasMax(A, x)
    taille(A) <- taille(A) + 1
    A[taille(A)] <- x
    i <- taille(A)
    Tant que (i > 1) Et (A[Parent(i)] < A[i]) Faire
        permuter A[i] Et A[Parent(i)]
        i <- parent(i)
    FinTantQue
Fin
```

Complexités

- $\text{Max}(A)$ est de complexité $O(1)$.
- $\text{ExtraireMax}(A)$ est de complexité $O(\log(n))$.
- $\text{Inserer}(A, n)$ est de complexité $O(\log(n))$.

IV Travaux pratiques

L'objectif de ce TP est d'implémenter les différents algorithmes sur les structures de données, Arbres binaires et File de priorité.

TP1 : Arbres binaires

Un arbre binaire est un arbre où chaque noeud peut avoir au maximum deux branches. Pour différencier les branches, on les nomme souvent droite ou gauche, càd deux pointeurs pour lier les éléments, un pointeur pour accéder à la branche de gauche et l'autre pour accéder à la branche de droite.

- Définir une structure permettant de coder un noeud d'un arbre binaire contenant une valeur entière.
- Implémenter la fonction permettant d'insérer des éléments dans un arbre binaire.
- Implémenter la fonction permettant de rechercher un élément avec une clé v dans un arbre binaire.
- Implémenter la fonction permettant de trouver le minimum et le maximum dans un arbre binaire.
- Implémenter la fonction permettant d'afficher les éléments d'un arbre binaire, càd Infixe, Préfixe et postfixe.
- Implémenter la fonction permettant de supprimer un élément dans un arbre binaire.
- Implémenter la fonction permettant de trouver le successeur d'un élément dans un arbre binaire.
- Écrire une fonction prenant un arbre binaire et rendant le nombre de ses éléments.
- Écrire une fonction prenant un arbre binaire et rendant sa hauteur, c'est à dire le nombre d'éléments contenus dans la plus longue branche.

TP2 : Tas binaires

Un tas binaire une structure permettant de maintenir un ordre partiel sur les données via un arbre binaire. Pour simplifier le problème, considérons un tas binaire qui contient que des entiers, càd priorités des éléments. Le sommet du

tas est donc l'entier le plus grand. Coderez les fonctionnalités de base (ex., ENTASSER-MAX, CONSTRUIRE-TAS-MAX) d'un tas binaire.

TP3 : Tri par Tas

Le Tri par Tas consiste à transformer le tableau initial en tas, puis à supprimer un à un les éléments du tas pour les replacer à la fin du tableau. Implémenter cet algorithme et le comparer avec les autres algorithmes de Tri (TP3), Insertion, Fusion, ...

- Donner les courbes expérimentales correspondantes aux coûts des différentes versions sur des tableaux de données aléatoires, sous Excell.
- Comparer ces courbes avec les courbes des coûts théoriques respectifs $O(n^2)$ et $O(n \log n)$ (sous Excell).

TP4 : Files de priorité

Une file de priorité est une structure de Données très utile dans de nombreuses applications, à titre d'exemple, dans les applications d'ordonnancement des tâches ou les simulateurs d'événements. Une file de priorité permet de gérer un ensemble d'éléments E avec les opérations suivantes : $INSERER(E, x)$, $MAXIMUM(E)$, $EXTRAIRE-MAX(E)$.

- Implémenter la FP à l'aide d'un tas binaire.
- Coderez les fonctionnalités de base, ex. Maximum(S), Extraire-Max(S), Augmenter-Clé(S,x,k), Insérer(S,x).
- Comparer les performances de la FP et de la structure de données type vecteur.

Documents à fournir (via la plateforme) : rapport (max 5 pages), les sources et les jeux de tests. Ils seront rendus dans une archive (nom-prenom-TPx.zip/gz).

V Projet I

La méthode de compression Huffman consiste à diminuer au maximum le nombre de bits utilisés pour coder un fragment d'information par la création de l'arbre de Huffman. Les feuilles de l'arbre correspondent aux lettres de l'alphabet. Les noeuds interne ne contiennent pas d'information. Le chemin emprunté pour atteindre une feuille depuis la racine définit le code de la lettre sur cette feuille : à gauche 0, à droite 1.

Développer l'algorithme vu en cours permettant, à l'aide d'un arbre binaire, d'effectuer une compression des fichiers de caractères. L'objectif principal

est d'obtenir un programme final fonctionnel qui permette de compresser (éventuellement décompresser) de l'information.

- Définir la structure d'un noeud de l'arbre Huffman (ex., caractère, fréquence, fils gauche et fils droite).
- Définir la structure de la collection des noeuds de l'arbre (ou arbre Huffman), par exemple, le tableau de pointeurs des noeuds.
- La fonction permettant d'allouer un nouveau noeud de tas-min avec comme paramètre un caractère donné et sa fréquence.
- La fonction permettant de créer un tas minimum de capacité donnée.
- La fonction permettant d'échanger deux noeuds de tas-min.
- La fonction standard *entasserMin* (similaire à *entasserMax*).
- La fonction standard *extraire-Tas-Min* permettant d'extraire le noeud de valeur minimale du tas (similaire à *extraire-Tas-Max*).
- La fonction standard *inserer-Tas-Min* pour insérer un nouveau noeud dans un tas min (similaire à *inserer-Tas-Max*).
- La fonction standard permettant de construire un tas min.
- La fonction permettant d'initialiser et créer un tas min.
- La fonction principale qui construit l'arbre Huffman.
- La fonction permettant de construire un arbre Huffman et imprime les codes en parcourant l'arbre.
- Fonctions permettent d'effectuer des tests et afficher les résultats.
- Lecture du texte à partir d'un fichier source et stockage dans un tableau.
- Calcul de l'alphabet et des fréquences.
- Codage et sauvegarde des codes et du texte codé dans des fichiers.
- Les fonctions permettant le décodage à partir des codes calculés.

Date de rendu du projet : voir la plateforme

Organisation

Travail individuel.

Critères d'évaluation

Les critères qui seront utilisés pour évaluer le travail rendu sont les suivants :

- La qualité du code, càd le code rendu devra être lisible et commenté.
- La qualité et la pertinence des tests.
- La qualité du rapport rendu.

Documents à fournir (via la plateforme) :

Rapport (max 10 pages), les sources et les jeux de tests seront rendus dans une archive (nom-prenom-ProjetI.zip/gz).

Chapitre 3

Les graphes

De manière générale, un graphe est une structure permettant de représenter une structure d'un ensemble complexe en exprimant les relations entre ses éléments.

En particulier, il permet de représenter de nombreuses situations rencontrées dans de nombreuses applications :

- circuits électriques
- réseaux de transport
- réseaux d'ordinateurs
- ordonnancement d'un ensemble de tâches
- ...

La théorie des graphes est :

- une méthode de pensée
- un moyen de modélisation
- permet l'étude d'une grande variété de problèmes : des problèmes ayant un aspect algorithmique, des problèmes d'optimisation combinatoire, ...

I Définitions

Un graphe est représenté par :

- S : L'ensemble des noeuds ou sommets
- A : L'ensemble des arcs ou arrêtes

Si n est le nombre de sommets et m le nombre d'arêtes, on peut distinguer deux cas :

- Cas 1 : Le nombre d'arêtes $m = n - 1$, le graphe est un arbre

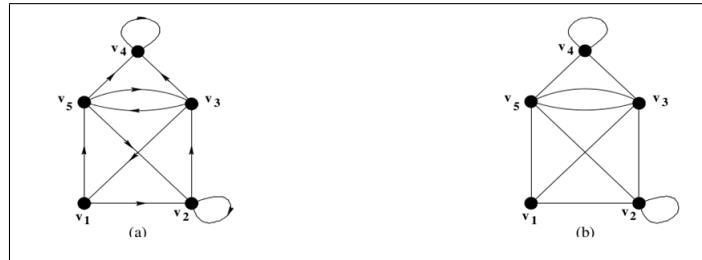


FIGURE 3.1 – (a) : graphe orienté ; (b) : le même graphe non orienté

- Cas 2 : Le nombre d'arêtes $\sum_{i=1}^n (n-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, le graphe est complètement connecté

II Structures de Données

Les graphes sont une structure de données importante en informatique. Il est donc fondamental de s'intéresser à la manière de représenter des graphes en vue de leurs manipulations algorithmiques.

Plusieurs modes de représentation peuvent être envisagés selon la nature des traitements que l'on souhaite appliquer au graphe considéré.

Dans cette partie, nous allons étudier les structures de données permettant de représenter des graphes :

- Représentation par matrice d'adjacence
- Représentation par tableau de listes d'adjacences

1. Matrice d'adjacence

C'est une matrice définie comme suit :

$$\begin{cases} A_{i,j} = 1 & \text{si } (i,j) \in E \\ A_{i,j} = 0 & \text{si } (i,j) \notin E \end{cases}$$

C'est une structure facile à utiliser, qui convient aux graphes denses, c'est à dire lorsque $|E|$ approche $O(|V|^2)$.

L'inconvénient majeur est la consommation de mémoire quelque soit le graphe, puisqu'il faut une taille en $O(|V|^2)$.

2. Liste d'adjacence

On utilise un tableau de $|V|$ listes. Pour chaque sommet v , on stocke $Adj(v)$, les sommets adjacents à v dans G .

Cette structure convient aux graphes peu denses, et la quantité de mémoire utilisée est $O(|V| + |E|)$. Toutefois, en contrepartie, sa mise en oeuvre n'est pas aisée.

La figure 3.2 (resp. 3.3) illustrent deux représentations d'un graphe non orienté (resp. orienté). (a) Un graphe non orienté G . (b) Une représentation de G par listes d'adjacences. (c) La représentation de G par matrice d'adjacences.

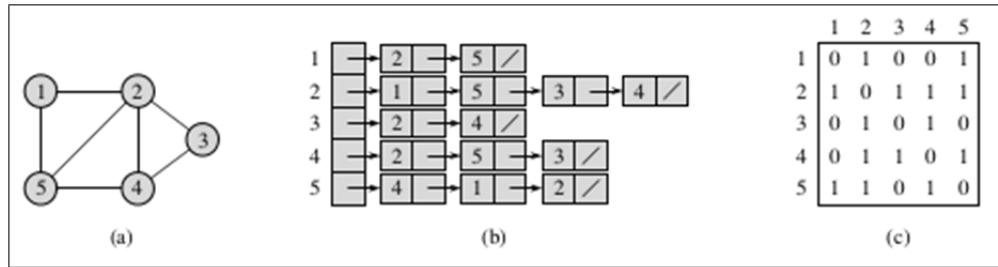


FIGURE 3.2 – Représentation d'un graphe non orienté

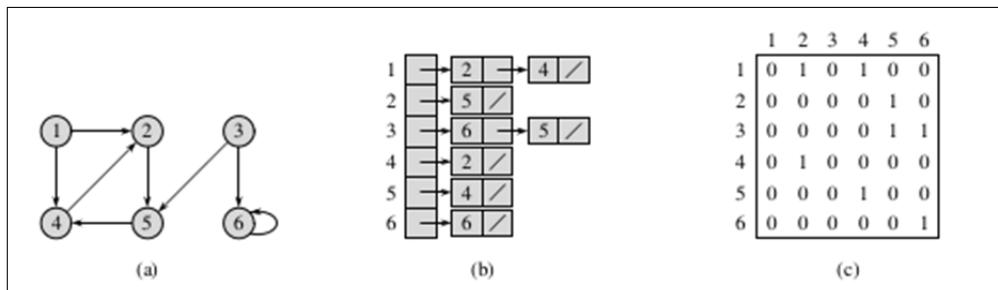


FIGURE 3.3 – Représentation d'un graphe orienté

III Algorithmes de parcours

Un parcours de graphe est une méthode systématique d'exploration des sommets et des arêtes d'un graphe. Le principe est toujours le même : on part

d'un sommet et on effectue une promenade en choisissant les arêtes.

Il existe deux algorithmes de parcours de graphes : le *parcours en largeur* et le *parcours en profondeur*. Ils sont à la base de nombreux algorithmes :

- L'algorithme du plus court chemin entre deux sommets (parcours en largeur).
- L'algorithme de construction de l'arbre couvrant minimal (parcours en largeur).
- L'algorithme de calcul des composantes connexes (parcours en profondeur).
- L'algorithme de tri topologique (parcours en profondeur).

1. Parcours en largeur

Le P.L. est la base de nombreux algorithmes importants sur les graphes. L'algorithme de Dijkstra pour calculer les plus courts chemins à origine unique et l'algorithme de Prim pour trouver l'arbre couvrant minimum utilisent des idées similaires à celles qui sont appliquées lors d'un parcours en largeur.

Il consiste à partir d'un sommet source s et de découvrir tous les sommets accessibles depuis celui-ci. Puis on procède de la même manière avec les sommets découverts.

En d'autres termes, étant donné un graphe $G = (S, A)$, et une origine s . Le P.L. emprunte systématiquement les arcs G pour découvrir tous les sommets accessibles depuis s . Il calcule la distance entre s et tous les sommets accessibles. Il construit également une arborescence de P.L. de racine s .

L'algorithme de P.L. découvre d'abord tous les sommets situés à une distance k de s , avant de découvrir tous les sommets situés à la distance $k + 1$.

Pour garder une trace de la progression, le P.L. colorie chaque sommet en *blanc*, *gris* ou *noir*. Tous les sommets sont blancs au départ. Un sommet est découvert la première fois qu'il est rencontré au cours de la recherche.

Les sommets gris ou noirs ont été déjà découverts. Si $(u, v) \in A$ et si le sommet u est noir alors le sommet v est gris ou noir. Les sommets gris peuvent avoir des sommets adjacents blancs.

La Figure 3.4 illustre l'algorithme PL on supposant que le graphe d'entrée $G = (S, A)$ est représenté par des listes d'adjacences :

- lignes 1–4 : colorient tous les sommets en blanc, donnent à $d[u]$ la valeur infinie pour chaque sommet u , et initialisent à NIL le parent de chaque sommet
- ligne 5 : colorie l'origine s en gris, car on convient qu'il est découvert au commencement
- ligne 6 : initialise $d[s]$ à 0
- ligne 7 : donne au parent de l'origine la valeur NIL
- ligne 8 : initialise F en y insérant le seul sommet s
- lignes 10–18 : se répète tant qu'il reste des sommets gris, sommets qui ont été découverts mais dont la liste d'adjacences n'a pas été entièrement examinée
 - ligne 10 : détermine le sommet gris u placé en tête de la file F
 - lignes 11–17 : considère chaque sommet v de la liste d'adjacences de u
 - ligne 18 : chaque fois qu'un sommet est ôté de la file il est colorié en noir

Pour chaque sommet du graphe, il maintient à jour plusieurs structures de données supplémentaires.

- $\text{couleur}[u]$: pour stocker la couleur de chaque sommet $u \in S$
- $\pi[u]$: pour stocker le parent de u ; $\pi[u] = NIL$ si u n'a pas de parent (par exemple, si $u = s$ ou si u n'a pas été découvert)
- $d[u]$: pour stocker la distance calculée par l'algorithme entre l'origine s et le sommet u
- F : une file FIFO pour gérer l'ensemble des sommets gris

La Figure 3.6 montre l'action de PL sur le graphe non orienté illustré dans la Figure 3.5.

Le temps d'exécution total de PL est donc $O(S + A)$, un temps linéaire par rapport à la taille de la représentation par listes d'adjacences de G .

Les propriétés du PL :

- Cet algorithme fonctionne sur un graphe orienté ou non orienté.
- Il permet de construire une arborescence de racine s .
- On découvre les sommets à une distance k et s avant les sommets à une distance $k + 1$.

Exercice 1 : Donner les valeurs d et π qui résultent d'un parcours en largeur du graphe non orienté de la Figure 3.5, en prenant pour origine le sommet u .

```

PL( $G, s$ )
1   pour chaque sommet  $u \in S[G] - \{s\}$ 
2     faire  $\text{couleur}[u] \leftarrow \text{BLANC}$ 
3      $d[u] \leftarrow \infty$ 
4      $\pi[u] \leftarrow \text{NIL}$ 
5    $\text{couleur}[s] \leftarrow \text{GRIS}$ 
6    $d[s] \leftarrow 0$ 
7    $\pi[s] \leftarrow \text{NIL}$ 
8    $F \leftarrow \{s\}$ 
9   tant que  $F \neq \emptyset$ 
10    faire  $u \leftarrow \text{tête}[F]$ 
11    pour chaque  $v \in \text{Adj}[u]$ 
12      faire si  $\text{couleur}[v] = \text{BLANC}$ 
13        alors  $\text{couleur}[v] \leftarrow \text{GRIS}$ 
14         $d[v] \leftarrow d[u] + 1$ 
15         $\pi[v] \leftarrow u$ 
16        ENFILE( $F, v$ )
17        DÉFILE( $F$ )
18     $\text{couleur}[u] \leftarrow \text{NOIR}$ 

```

FIGURE 3.4 – L'algorithme PL

2. Parcours en profondeur

C'est un algorithme de recherche qui progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S .

Le nom d'algorithme en profondeur est du au fait que, contrairement à l'algorithme de parcours en largeur, il explore en fait "à fond" les chemins un par un : pour chaque sommet, il prend le premier sommet voisin jusqu'à ce qu'un sommet n'aie plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père.

Dans le parcours en largeur, le sous-graphe prédécesseur forme une arborescence, le sous-graphe prédécesseur obtenu par un parcours en profondeur peut être composé de plusieurs arborescences, car le parcours peut être répété à partir de plusieurs origines.

La Figure 3.7 présente le pseudo code de l'algorithme PP. Le graphe d'entrée G peut être orienté ou non.

- $\text{couleur}[u]$: pour stocker la couleur de chaque sommet $u \in S$
- $\pi[u]$: pour stocker le parent de u
- $d[u]$: marque le moment où v a été découvert pour la première fois

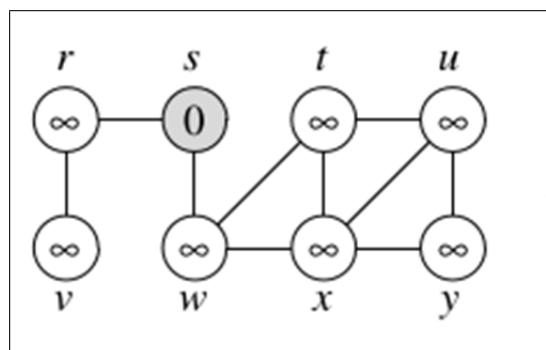


FIGURE 3.5 – Un graphe non orienté

(et colorié en gris)

- $f[u]$: enregistre le moment où le parcours a fini d'examiner la liste d'adjacences de v (et le colorie en noir). Le sommet u est BLANC avant l'instant $d[u]$, GRIS entre $d[u]$ et $f[u]$, et NOIR après
- $date$: est une variable globale qui sert à la datation

PP fonctionne de la manière suivante :

- lignes 1–3 : colorient tous les sommets en blanc et initialisent leurs champs π à *NIL*
- ligne 4 : initialise le compteur de dates
- lignes 5–7 : testent chaque sommet de S l'un après l'autre et, quand un sommet blanc est trouvé, le visitent grâce à *VISITER-PP*

À chaque appel *VISITER-PP*(u), le sommet u devient la racine d'une nouvelle arborescence de la forêt de parcours en profondeur. Quand PP se termine, chaque sommet u s'est vu affecter une date de découverte $d[u]$ et une date de fin de traitement $f[u]$.

Pour chaque appel *VISITER-PP*(u), le sommet u est blanc initialement :

- ligne 1 : colorie u en gris
- ligne 2 : incrémente la variable globale $date$
- ligne 3 : enregistre la nouvelle valeur de $date$ pour en faire la date de découverte $d[u]$
- lignes 4–7 : examinent chaque sommet v adjacent à u et visitent récursivement v s'il est blanc
- lignes 8–9 : peignent u en noir et enregistrent la date de fin de traitement dans $f[u]$.

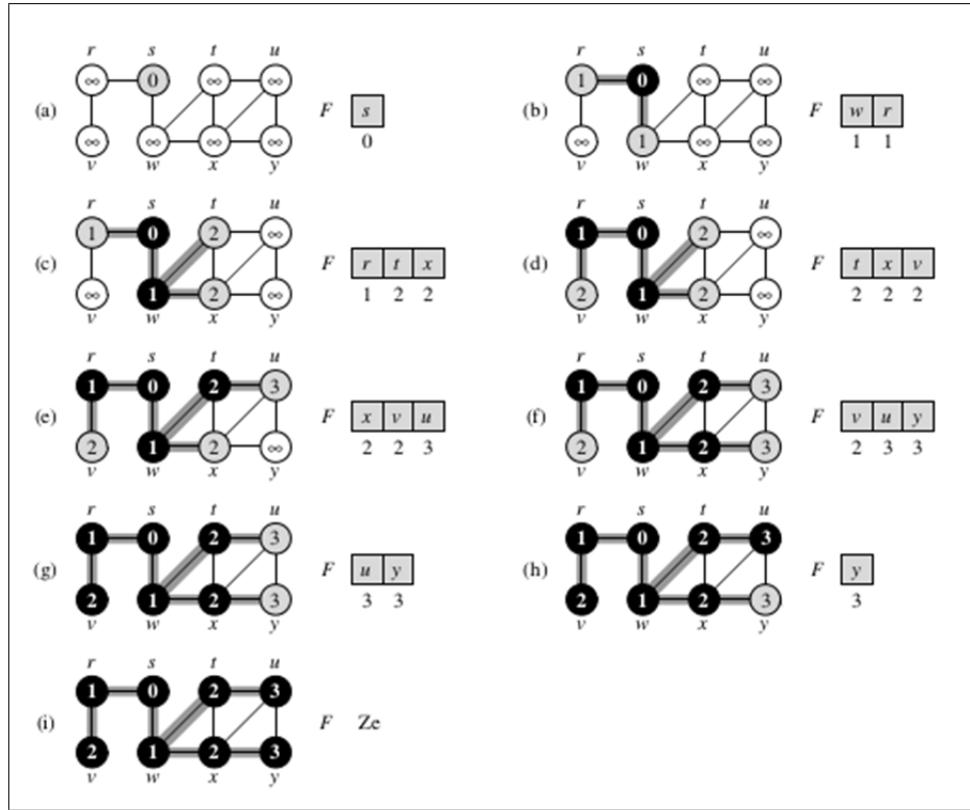


FIGURE 3.6 – L'action de PL sur un graphe non orienté

La Figure 3.9 présente l'action de PP sur le graphe orienté de la Figure 3.8. Les arcs ne faisant pas partie des arborescences (arcs qui ne sont pas des arcs de liaison) sont étiquetés R, T ou A selon que ce sont des arcs arrières, transverses ou avants. Les sommets sont étiquetés par leur dates de découverte et de fin de traitement.

Les arcs (de liaison) qui n'appartiennent pas à la forêt couvrante sont :

- soit des arcs avant, càd des arcs de transitivité dans la forêt couvrante ;
 $u \rightarrow v$ est avant ssi $d[v]f[v]$ est inclus dans $d[u]f[u]$. Ce sont des arcs qui relient (pas de liaison) un sommet u à un descendant v dans une arborescence de parcours en profondeur.
- soit des arcs arrières (ou retour) : $u \rightarrow v$ est arrière ssi $d[u]f[u]$ est inclus dans $d[v]f[v]$. Cet arc provoque l'existence d'un cycle dans le graphe. Les arcs R relient un sommet u à un ancêtre v dans une arborescence de parcours en profondeur.
- soit un arc transverse, qui lie deux arbres distincts de la forêt ; si

```

PP( $G$ )
1 pour chaque sommet  $u \in S[G]$ 
2   faire  $\text{couleur}[u] \leftarrow \text{BLANC}$ 
3    $\pi[u] \leftarrow \text{NIL}$ 
4    $\text{date} \leftarrow 0$ 
5 pour chaque sommet  $u \in S[G]$ 
6   faire si  $\text{couleur}[u] = \text{BLANC}$ 
7     alors VISITER-PP( $u$ )

VISITER-PP( $u$ )
1  $\text{couleur}[u] \leftarrow \text{GRIS} \triangleright$  sommet blanc  $u$  vient d'être découvert.
2  $\text{date} \leftarrow \text{date} + 1$ 
3  $d[u] \leftarrow \text{date}$ 
4 pour chaque  $v \in \text{Adj}[u] \triangleright$  Exploration de l'arc  $(u, v)$ .
5   faire si  $\text{couleur}[v] = \text{BLANC}$ 
6     alors  $\pi[v] \leftarrow u$ 
7       VISITER-PP( $v$ )
8    $\text{couleur}[u] \leftarrow \text{NOIR} \triangleright$  noircir  $u$ , car on en a fini avec lui.
9    $f[u] \leftarrow \text{date} \leftarrow \text{date} + 1$ 

```

FIGURE 3.7 – L'algorithme PP

$u \rightarrow v$ est transverse, les intervalles de u et v sont disjoints. De plus $d[v] < f[v] < d[u] < f[u]$: l'arbre contenant u a été construit avant celui de v .

Deux propriétés fondamentales du parcours en profondeur (Cf. Figure 3.10) :

- le sous-graphe prédécesseur G_π forme en fait une forêt, puisque la structure des arborescences de parcours en profondeur reflète exactement la structure des appels récursifs à VISITER-PP.
- les dates de découverte et de fin de traitement ont une structure parenthésée. Si l'on représente la découverte d'un sommet u par une parenthèse gauche " (u) " et la fin de son traitement par une parenthèse droite " $u)$ ", alors la succession des découvertes et des fins de traitement crée une expression bien formée, au sens où les parenthèses sont correctement imbriquées.

Exercice 2 : Donner le déroulement du parcours en profondeur sur le graphe de la Figure 3.11. On supposera que la boucle pour des lignes 5–7 de la procédure PP considère les sommets dans l'ordre alphabétique, et que chaque liste d'adjacences est ordonnée alphabétiquement. Donner les dates de découverte

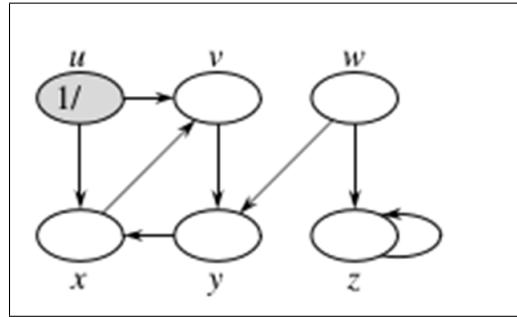


FIGURE 3.8 – Un graphe orienté

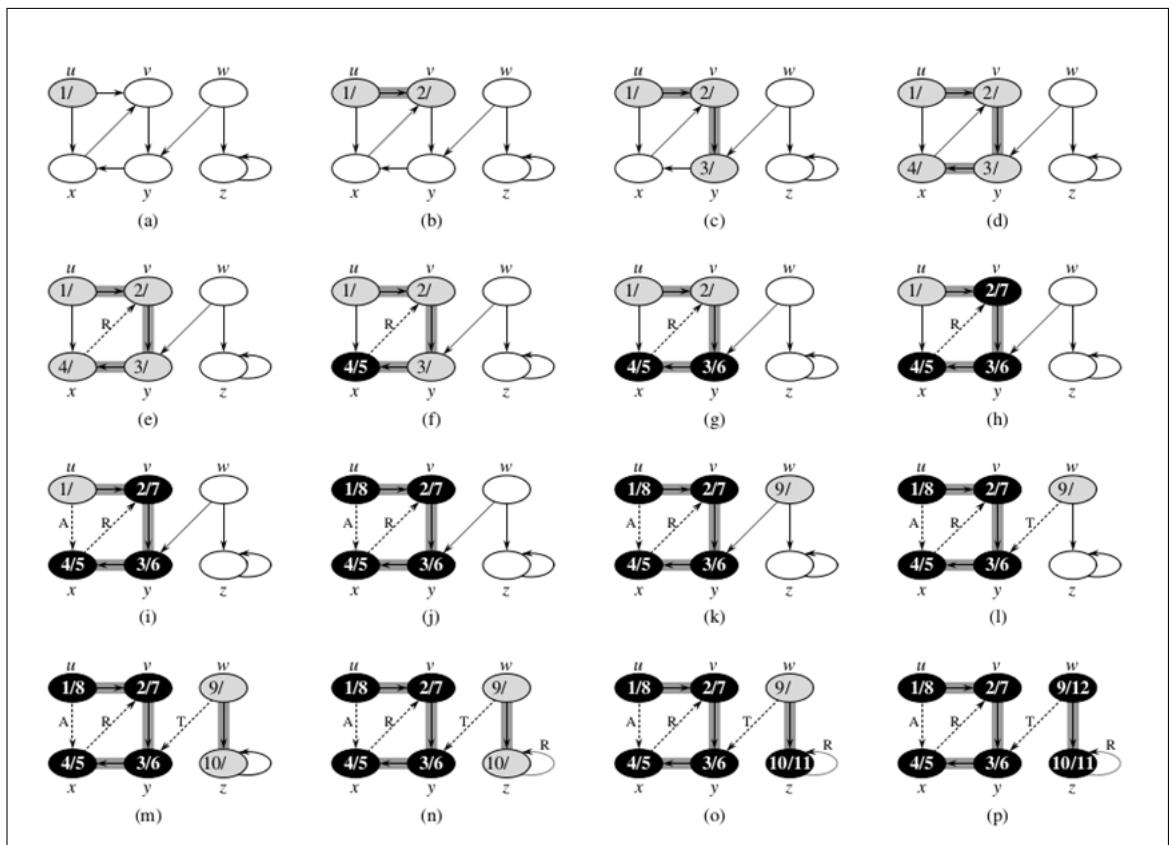


FIGURE 3.9 – L'action de PP sur un graphe orienté

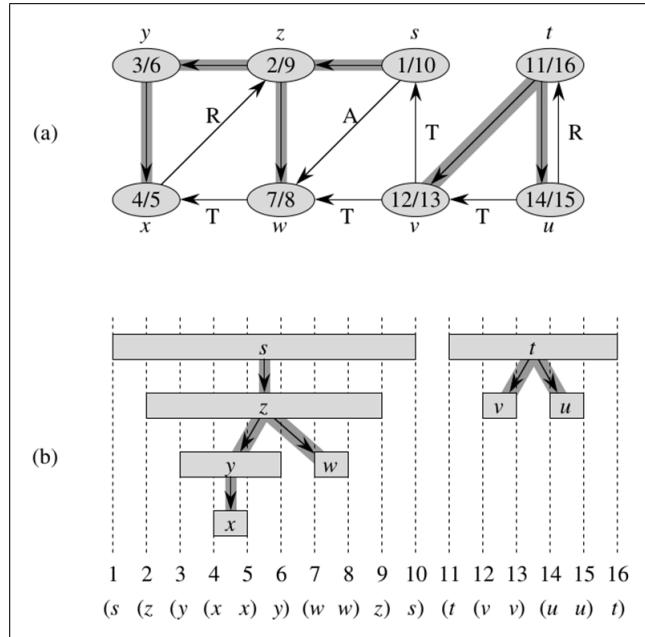


FIGURE 3.10 – (a) Le résultat d'un PP. (b) Les intervalles des dates de découverte et de fin de traitement

et de fin de traitement de chaque sommet.

Exercice 3 : Montrer que le temps d'exécution de PP est $\Theta(S + A)$.

IV Tri topologique

Nombreuses applications utilisent les graphes orientés sans circuit pour représenter des précédences entre événements.

Le tri topologique d'un graphe orienté sans circuit $G = (S, A)$ consiste à ordonner linéairement tous ses sommets de sorte que, si G contient un arc (u, v) , u apparaisse avant v dans le tri.

La Figure 3.12 donne une façon de s'habiller le matin. Un arc (u, v) du graphe orienté sans circuit indique que le vêtement u doit être enfilé avant le vêtement v . Le tri topologique de ce graphe orienté sans circuit donne donc un ordre permettant de s'habiller correctement.

Le graphe orienté sans circuit trié topologiquement peut être vue comme une

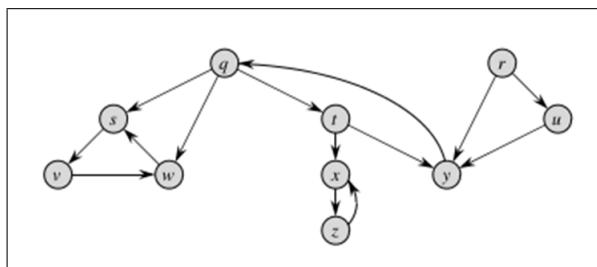


FIGURE 3.11 – Un graphe orienté

suite de sommets sur une ligne horizontale de telle façon que tous les arcs soient orientés de gauche à droite. Dans cet figure les sommets sont triés topologiquement selon l'ordre inverse des dates de fin de traitement.

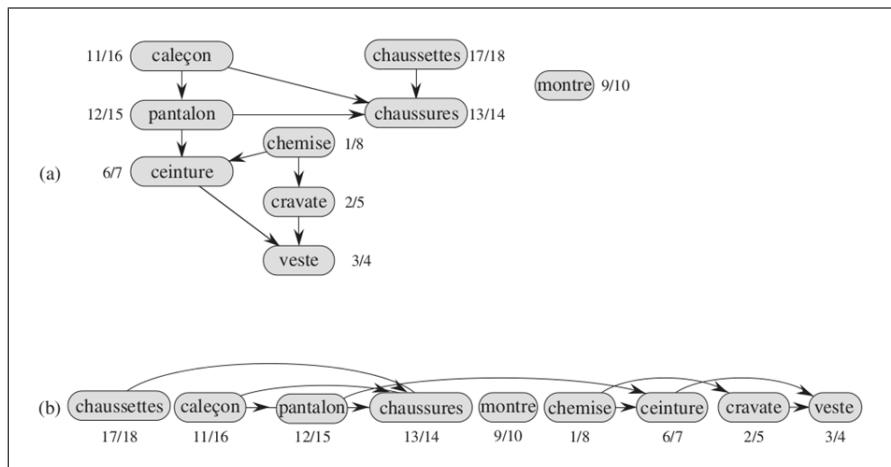


FIGURE 3.12 – Exemple de tri topologique

La Figure 3.13 présente l'algorithme permettant d'effectuer le tri topologique d'un graphe orienté sans circuit.

Exercice 4 : Donner l'ordre dans lequel TRI-TOPOLOGIQUE trie les sommets quand on l'exécute sur le graphe orienté sans circuit de la Figure 3.14, en gardant l'hypothèse de l'exercice 3.

Exercice 5 : Montrer que le temps d'exécution de tri topologique est $\Theta(S + A)$.

TRI-TOPOLOGIQUE(G)

- 1 appeler $\text{PP}(G)$ pour calculer les dates de fin de traitement $f[v]$ pour chaque sommet v
- 2 chaque fois que le traitement d'un sommet se termine, insérer le sommet début d'une liste chaînée
- 3 **retourner** la liste chaînée des sommets

FIGURE 3.13 – L'algorithme de tri topologique

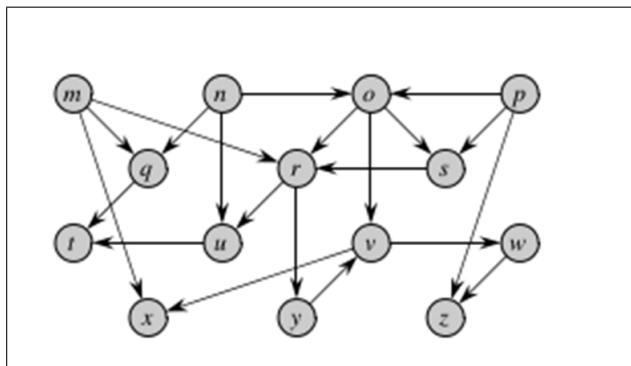


FIGURE 3.14 – Un graphe orienté sans circuit

V Composantes fortement connexes

Une composante fortement connexe d'un graphe orienté $G = (S, A)$ est un ensemble maximal de sommets $R \subseteq S$ tel que, pour chaque paire de sommets u et v de R , on ait à la fois $u \rightsquigarrow v$ et $v \rightsquigarrow u$; autrement dit, les sommets u et v sont mutuellement accessibles.

La Figure 3.15 illustre un exemple avec des composantes fortement connexes :

- (a) Les composantes fortement connexes de G sont matérialisées par des régions en gris. Chaque sommet est étiqueté par ses dates de découverte et de fin de traitement. Les arcs de liaison sont ombrés.
- (b) Le graphe transposé de G . Chaque composante fortement connexe correspond à une arborescence de parcours en profondeur. Les sommets b , c , g , et h , en gris foncé, sont les racines des arborescences de parcours en profondeur obtenus par le parcours en profondeur de graphe transposé.
- (c) Le graphe sans circuit des composantes obtenu par contraction de tous les arcs de chaque composante fortement connexe de G , de façon

que chaque composante ne contienne plus qu'un seul sommet.

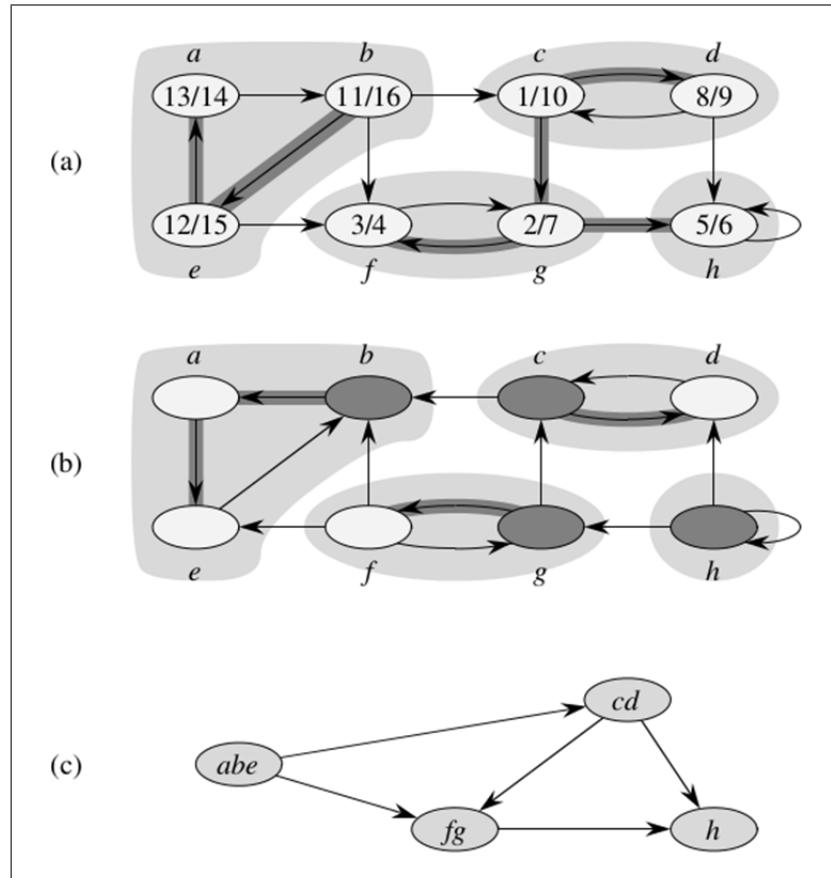


FIGURE 3.15 – L'algorithme de tri topologique

La Figure 3.16 présente l'algorithme de recherche des composantes fortement connexes d'un graphe $G = (S, A)$. Cet algorithme utilise le transposé de G .

Étant donné une représentation par listes d'adjacences de G , la création de graphe transposé demande un temps en $O(S + A)$.

Exercice 6 : Décrire le fonctionnement de la procédure COMPOSANTES-FORTEMENT-CONNEXES sur le graphe de la Figure 3.11. Plus précisément, donner les dates de fin de traitement calculées à la ligne 1 et la forêt obtenue à la ligne 3. On suppose que la boucle des lignes 5–7 de PP considère les sommets par ordre alphabétique et que les listes d'adjacences sont triées par ordre alphabétique.

COMPOSANTES-FORTEMENT-CONNEXES(G)

- 1 appeler $\text{PP}(G)$ pour calculer les dates de fin de traitement $f[u]$
pour chaque sommet u
- 2 calculer ${}^T G$
- 3 appeler $\text{PP}({}^T G)$, mais dans la boucle principale de PP , traiter les sommets
par ordre de $f[u]$ (calculés en ligne 1) décroissants
- 4 imprimer les sommets de chaque arborescence de la forêt obtenue
en ligne 3 en tant que composante fortement connexe distincte

FIGURE 3.16 – L’algorithme de recherche des composantes fortement connexes

VI Arbres couvrants de poids minimum

Lors de la phase de conception de circuits électroniques, on a souvent besoin de relier entre elles les broches de composants électriquement équivalents. Pour interconnecter un ensemble de n broches, on peut utiliser un arrangement de $n - 1$ branchements, chacun reliant deux broches. Parmi tous les arrangements possibles, celui qui utilise une longueur de branchements minimale est souvent le plus souhaitable.

Ce problème de câblage peut être modéliser à l’aide d’un graphe non orienté connexe $G = (S, A)$ où S représente l’ensemble des broches, où A l’ensemble des interconnections possibles entre paires de broches, et où pour chaque arête $(u, v) \in A$, on a un poids $w(u, v)$ qui spécifie le coût (longueur de fil nécessaire) pour connecter u et v .

On souhaite alors trouver un sous-ensemble acyclique $T \subseteq A$ qui connecte tous les sommets et dont le poids total $w(T) = \sum_{(u,v) \in T} w(u, v)$ soit minimum. Puisque T est acyclique et connecte tous les sommets, il doit former un arbre, que l’on appelle arbre couvrant car il « couvre » le graphe G .

La Figure 3.17 illustre un graph et la Figure 3.18 l’arbre de recouvrement minimum obtenu.

La Figure 3.19 illustre un autre exemple d’arbre de recouvrement minimum. Les arêtes de l’arbre couvrant minimum sont sur fond gris. Le poids total de l’arbre montré est 37. L’arbre couvrant minimal n’est pas unique : si l’on remplace l’arête (b, c) par l’arête (a, h) , on obtient un autre arbre couvrant de poids 37.

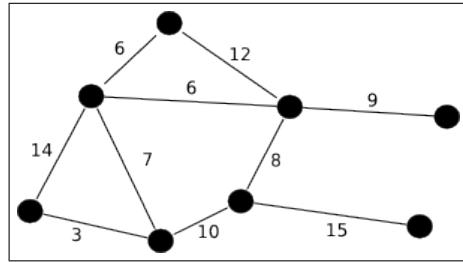


FIGURE 3.17 – Exemple d'un graphe

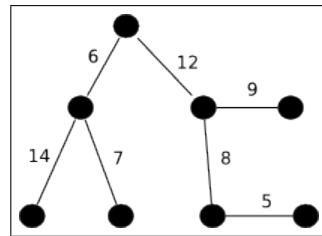


FIGURE 3.18 – L'arbre de recouvrement minimum

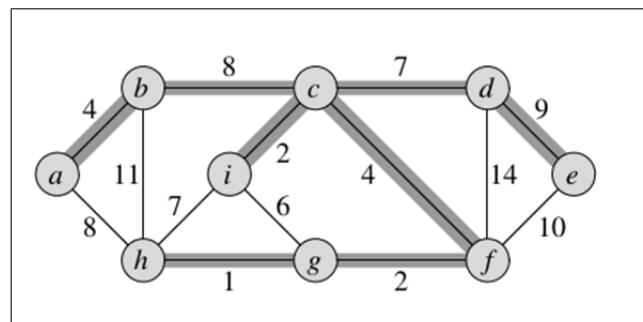


FIGURE 3.19 – un exemple d'un arbre de recouvrement minimum

Dans le reste de cette section, nous examinerons deux algorithmes permettant de trouver un arbre couvrant minimum : l'algorithme de Kruskal et l'algorithme de Prim. En utilisant des tas binaires ordinaires, on peut aisément faire en sorte qu'ils s'exécutent chacun en $O(AlgS)$.

Les deux algorithmes suivent une approche gloutonne. La stratégie gloutonne effectue le choix qui semble le meilleur à l'instant donné. Une telle stratégie n'aboutit pas forcément à des solutions globalement optimales.

Nous introduisons un algorithme « générique » d'arbre couvrant minimum, qui fait croître un arbre couvrant en lui ajoutant une arête à la fois. Après on présente deux façons d'implémenter l'algorithme générique, Kruskal et Prim.

1. Algorithme générique

Le principe de l'algorithme générique est le suivant : à chaque étape, on détermine une arête (u, v) qui peut être ajoutée à E , au sens où $E \cup (u, v)$ est également un sous-ensemble d'un arbre couvrant minimum. On appelle ce type d'arête *arête sûre* pour E , car on peut l'ajouter à E sans détruire l'invariant suivant : avant chaque itération, E est un sous-ensemble d'un arbre couvrant minimum.

```

ACM-GÉNÉRIQUE( $G, w$ )
1  $E \leftarrow \emptyset$ 
2 tant que  $E$  ne forme pas un arbre couvrant
3   faire trouver une arête  $(u, v)$  qui est sûre pour  $E$ 
4      $E \leftarrow E \cup \{(u, v)\}$ 
5 retourner  $E$ 
```

FIGURE 3.20 – Pseudo-code de l'algorithme générique

2. Algorithme de Kruskal

L'algorithme de Kruskal s'inspire directement de l'algorithme générique de l'arbre couvrant minimum.

Il trouve une arête sûre à ajouter à la forêt en cherchant, parmi toutes les arêtes reliant deux arbres quelconques de la forêt, une arête (u, v) de poids minimal. En d'autres termes, il utilise une forêt. Initialement, la forêt est

formée de tous les sommets du graphe, sans aucune arête. A chaque étape, on ajoute à la forêt une arête choisie de poids minimum, mais *seulement si cette arête ne crée pas de circuit.*

L'algorithme de Kruskal (Cf Figure 3.21 est un algorithme glouton car, à chaque étape, il ajoute à la forêt une arête de poids minimal :

- lignes 1–3 : initialisent l'ensemble E à l'ensemble vide et créent $|S|$ arbres, un pour chaque sommet
- ligne 4 : les arêtes de A sont triées par ordre de poids croissant
- lignes 5–8 : la boucle teste, pour chaque arête (u, v) , si ses extrémités u et v appartiennent au même arbre.
- Si c'est le cas, l'arête (u, v) ne peut pas être ajoutée à la forêt sans créer de cycle ; l'arête est donc rejetée.
- Sinon, l'arête (u, v) est ajoutée à E en ligne 7, et les sommets des deux arbres sont fusionnés en ligne 8.

On peut le voir selon trois étapes comme suit :

Etape 1 : E est initialisé à Vide. On trie les arcs de G en ordre croissant selon leur poids dans une liste L .

Etape 2 : On sélectionne le premier arc de L et on l'inclut dans E .

Etape 3 :

Si tous les arcs de L ont été examinés Alors
Retourner "Graphe non connecté"

Sinon

On prend le premier arc non examiné de L
S'il ne crée pas un cycle dans T
On l'ajoute à T et on va à l'étape 4.

Sinon

On supprime cet arc de L et on recommence l'étape 3

Etape 4 : On arrête le traitement si E a $(n - 1)$ arcs, sinon on retourne à l'étape 3.

Le fonctionnement de l'algorithme de Kruskal est illustré à la Figure 3.22. Les arêtes sur fond gris appartiennent à la forêt E en cours de construction. Les arêtes sont considérées par l'algorithme par ordre de poids croissant. Une flèche indique l'arête sélectionnée à chaque étape de l'algorithme. Si l'arête relie deux arbres distincts de la forêt, elle est ajoutée à la forêt, provoquant alors la fusion des deux arbres.

Exercice 7 : Montrez que le temps d'exécution de l'algorithme de Kruskal

```

ACM-KRUSKAL( $G, w$ )
1  $E \leftarrow \emptyset$ 
2 pour chaque sommet  $v \in S[G]$ 
3   faire CRÉER-ENSEMBLE( $v$ )
4   trier les arêtes de  $A$  par ordre croissant de poids  $w$ 
5   pour chaque arête  $(u, v) \in A$  pris par ordre de poids croissant
6     faire si TROUVER-ENSEMBLE( $u$ )  $\neq$  TROUVER-ENSEMBLE( $v$ )
7       alors  $E \leftarrow E \cup \{(u, v)\}$ 
8         UNION( $u, v$ )
9   retourner  $E$ 

```

FIGURE 3.21 – Pseudo-code de l'algorithme Kruskal

sur un graphe $G = (S, A)$ est en $O(A \log S)$.

3. Algorithme de Prim

Comme le montre la Figure 3.24, l'arbre démarre d'un sommet racine r arbitraire puis croît jusqu'à couvrir tous les sommets de S . Pendant l'exécution, tous les sommets qui n'appartiennent pas à l'arbre se trouvent dans une file de priorités $\min F$ basée sur un champ clé. Pour chaque sommet v , $cle[v]$ est le poids minimal d'une arête reliant v à un sommet de l'arbre ; par convention, $cle[v] = \infty$ si une telle arête n'existe pas. Le champ $\pi[v]$ désigne le parent de v dans l'arbre.

L'algorithme de Prim (Cf Figure 3.23 fonctionne comme suit :

- lignes 1–5 : initialisent la clé de chaque sommet à ∞ (exception faite de la racine r , dont la clé est initialisée à 0 pour qu'elle soit le premier sommet traité), initialisent le parent de chaque sommet à NIL et initialisent la file de priorités $\min F$ de façon qu'elle contienne tous les sommets.
- ligne 7 : identifie un sommet $u \in Q$ qui est incident pour une arête minimale traversant la coupe $(S - F, F)$ (sauf dans la première itération, où $u = r$ à cause de la ligne 4). Supprimer u de l'ensemble F a pour effet de l'ajouter à l'ensemble $S - F$ des sommets de l'arbre, et donc d'ajouter $(u, \pi[u])$ à E .
- lignes 8–11 : met à jour les champs clé et π de chaque sommet v adjacent à u mais pas dans l'arbre.

En d'autres termes, l'algorithme Prim fonctionne selon les étapes suivantes :

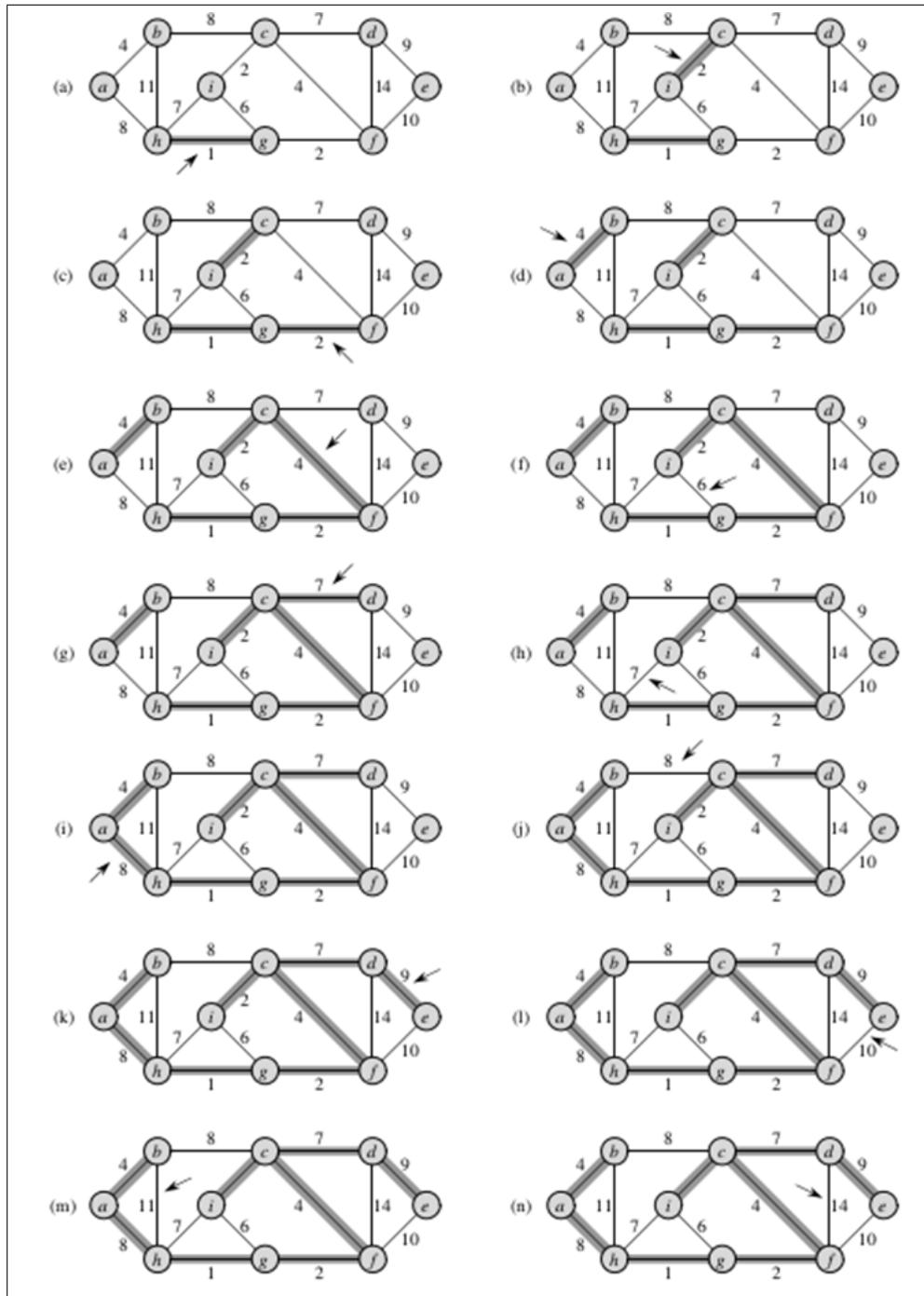


FIGURE 3.22 – L'exécution de l'algorithme de Kruskal

```

ACM-PRIM( $G, w, r$ )
1   pour chaque  $u \in S[G]$ 
2     faire  $clé[u] \leftarrow \infty$ 
3      $\pi[u] \leftarrow \text{NIL}$ 
4    $clé[r] \leftarrow 0$ 
5    $F \leftarrow S[G]$ 
6   tant que  $F \neq \emptyset$ 
7     faire  $u \leftarrow \text{EXTRAIRE-MIN}(F)$ 
8     pour chaque  $v \in Adj[u]$ 
9       faire si  $v \in F$  et  $w(u, v) < clé[v]$ 
10      alors  $\pi[v] \leftarrow u$ 
11       $clé[v] \leftarrow w(u, v)$ 

```

FIGURE 3.23 – Le pseudo code de l'algorithme de Prim

- En entrée : G
- En sortie : T

Etape 1 : On sélectionne un sommet quelconque de V et on l'insère dans T.

Etape 2 : Soit S l'ensemble de sommets de T. Si les deux ensembles S et $(V - S)$ sont connectés alors on renvoie “Graphe non convexe”.

On cherche un arc de poids minimum connectant S et $(V - S)$ s'ils ne forment pas un cycle dans T, on l'insère dans T et on va à l'étape 3.

Etape 3 : Si T a $n - 1$ arcs on retourne T. Sinon, on retourne à l'étape 2. L'exécution de l'algorithme de Prim est illustrée dans la Figure 3.24. Le sommet racine est a . Les arêtes sur fond gris appartiennent à l'arbre en cours de construction, et les sommets de l'arbre sont représentés en noir. À chaque étape de l'algorithme, les sommets de l'arbre déterminent une coupe du graphe, et une arête minimale traversant la coupe est ajoutée à l'arbre. Dans la deuxième étape, par exemple, l'algorithme a le choix entre ajouter l'arête (b, c) ou l'arête (a, h) puisque toutes les deux sont des arêtes minimales traversant la coupe.

Exercice 8 : Donnez le temps d'exécution de l'algorithme de Prim sur un graphe $G = (S, A)$ dans le cas où la file de priorités $min\ F$ est implémentée comme un tas min binaire.

Exercice 8.1 : En utilisant l'algorithme de Prim, trouver l'arbre de poids minimum dans le graphe suivant, en partant du sommet A. Appliquez l'algorithme de Kruskal sur le même graphe.

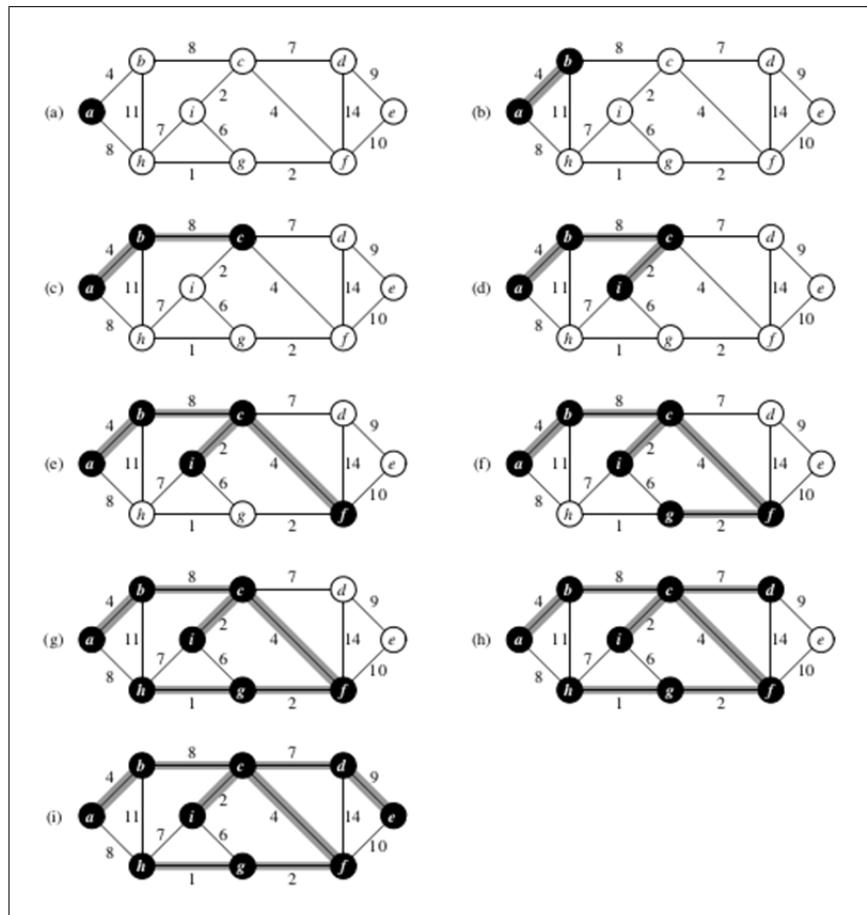
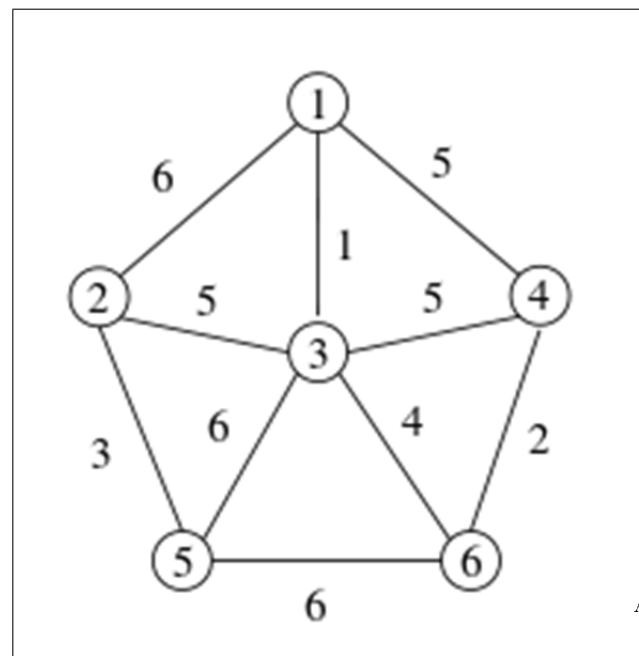


FIGURE 3.24 – L'exécution de l'algorithme de Prim



VII Plus courts chemins à origine unique

Un automobiliste souhaite trouver le plus court chemin possible entre une ville de départ et une ville destinataire. Étant donnée une carte routière du payé, avec les distances de chaque portion de route, comment peut-il déterminer la route la plus courte ? C'est un problème de plus courts chemins.

Dans un problème de plus courts chemins, on possède en entrée un graphe orienté pondéré $G = (S, A)$, avec une fonction de pondération $w : A \rightarrow R$ qui fait correspondre à chaque arc un poids à valeur réelle. La longueur du chemin $p = v_0, v_1, \dots, v_k$ est la somme des poids (longueurs) des arcs qui le constituent : $w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$. S'il y a un chemin entre u et v , la longueur du plus court chemin est définie comme suit : $\delta(u, v) = \min\{w(p) : u \rightsquigarrow^p v\}$

RQ : Les poids des arcs peuvent mesurer autre chose que des distances. On s'en sert souvent pour représenter un temps, un coût, des pénalités, ou n'importe quelle autre quantité qui s'accumule linéairement le long d'un chemin, et qu'il s'agit de minimiser.

Dans cette section, nous allons voir comment résoudre efficacement ce type de problèmes. Nous allons voir comment résoudre le problème de recherche du plus court chemin à origine unique : étant donné un graphe $G = (S, A)$, on souhaite trouver un plus court chemin depuis un sommet origine donné $s \in S$ vers n'importe quel sommet $v \in S$. D'autres problèmes peuvent être résolus par l'algorithme à origine unique, en particulier les variantes suivantes :

- Plus court chemin à destination unique : trouver un plus court chemin vers un sommet de destination à partir de n'importe quel sommet du graphe. En inversant le sens de chaque arc du graphe, on peut ramener ce problème à un problème à origine unique.
- Plus court chemin pour un couple de sommets donné : trouver un plus court chemin de u à v pour deux sommets donnés u et v . Si le problème à origine unique est résolu pour le sommet origine u , ce problème également résolu.
- Plus court chemin pour tout couple de sommets : trouver un plus court chemin de u à v pour tout couple de sommets u et v . Ce problème peut être résolu, par exemple, en exécutant un algorithme à origine unique à partir de chaque sommet.

RQ : Les plus courts chemins ne sont pas forcément uniques, pas plus que les arborescences de plus courts chemins. Par exemple, la Figure 3.25 montre un

graphe orienté pondéré et deux arborescences de plus courts chemins ayant la même racine.

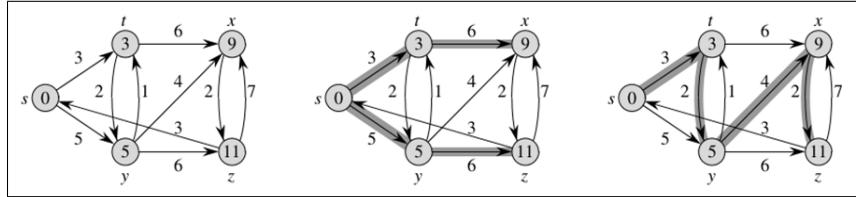


FIGURE 3.25 – Un graphe orienté pondéré. Les arcs en gris forment une arborescence de plus courts chemins de racine s

1. Algorithme de Dijkstra

Cet algorithme résout le problème de la recherche d'un plus court chemin à origine unique pour un graphe orienté pondéré $G(S, A)$ dans le cas où tous les arcs sont de poids positifs ou nuls.

L'algorithme de Dijkstra gère un ensemble E de sommets dont les longueurs finales de plus court chemin à partir de l'origine s ont été calculées.

A chaque itération, l'algorithme choisit le sommet dont l'estimation de plus court chemin est minimale.

La Figure 3.26 présente le détail de l'algorithme.

- ligne 1 : effectue l'initialisation habituelle des valeurs d et p (Cf. Figure 3.27)
- ligne 2 : initialise l'ensemble E
- ligne 3 : initialise la file de priorités $\min F$ en y incluant tous les sommets de S ; comme à ce moment-là $E = \emptyset$
- lignes 4–8 : à chaque passage dans la boucle tant que un sommet u est extrait de $F = S - E$ et inséré dans l'ensemble E
- lignes 7–8 : relâchent chaque arc (u, v) partant de u , ce qui a pour effet de mettre à jour l'estimation $d[v]$ et le prédécesseur $\pi[v]$

La Figure 3.28 présente le pseudo-code de la procédure permettant d'effectuer une étape de relâchement sur l'arc (u, v) . Une étape de relâchement peut diminuer la valeur de l'estimation de plus court chemin $d[v]$ et mettre à jour le champ prédécesseur $\pi[v]$ de v . La Figure 3.29 illustre le relâchement

```

DUJIKSTRA( $G, w, s$ )
1 SOURCE-UNIQUE-INITIALISATION( $G, s$ )
2  $E \leftarrow \emptyset$ 
3  $F \leftarrow S[G]$ 
4 tant que  $F \neq \emptyset$ 
5   faire  $u \leftarrow$  EXTRAIRE-MIN( $F$ )
6    $E \leftarrow E \cup \{u\}$ 
7   pour chaque sommet  $v \in Adj[u]$ 
8     faire RELÂCHER( $u, v, w$ )

```

FIGURE 3.26 – L’algorithme Dijkstra

```

SOURCE-UNIQUE-INITIALISATION( $G, s$ )
1 pour chaque sommet  $v \in S[G]$ 
2   faire  $d[v] \leftarrow \infty$ 
3    $\pi[v] \leftarrow \text{NIL}$ 
4  $d[s] \leftarrow 0$ 

```

FIGURE 3.27 – La procédure Source-Unique-Initialisation(G, s)

d’un arc (u, v) de poids $w(u, v) = 2$. L’estimation de plus court chemin de chaque sommet est affichée dans le sommet. Dans le cas a, comme $d[v] > d[u] + w(u, v)$ avant relâchement, la valeur de $d[v]$ décroît. Dans le cas b, $d[v] \leq d[u] + w(u, v)$ avant l’étape de relâchement, de sorte que $d[v]$ ne change pas avec le relâchement.

```

RELÂCHER( $u, v, w$ )
1 si  $d[v] > d[u] + w(u, v)$ 
2 alors  $d[v] \leftarrow d[u] + w(u, v)$ 
3  $\pi[v] \leftarrow u$ 

```

FIGURE 3.28 – La procédure Relacher

La Figure 3.30 illustre l’exécution de l’algorithme de Dijksta. Les estimations de plus court chemin sont représentées à l’intérieur des sommets, et les arcs en gris indiquent les prédécesseurs. Les sommets noirs sont dans l’ensemble E , et les sommets blancs sont dans la file de priorités $\min F = S - E$

Exercice 9 : Montrez que le temps de l’algorithme de Dijksta est en $O(S^2 + A) = O(S^2)$. L’algorithme gère la file de priorités $\min F$ en appelant trois opérations de file de priorités : INSÉRER (implicite en ligne 3), EXTRAIRE-

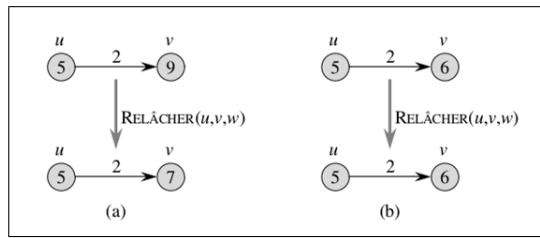
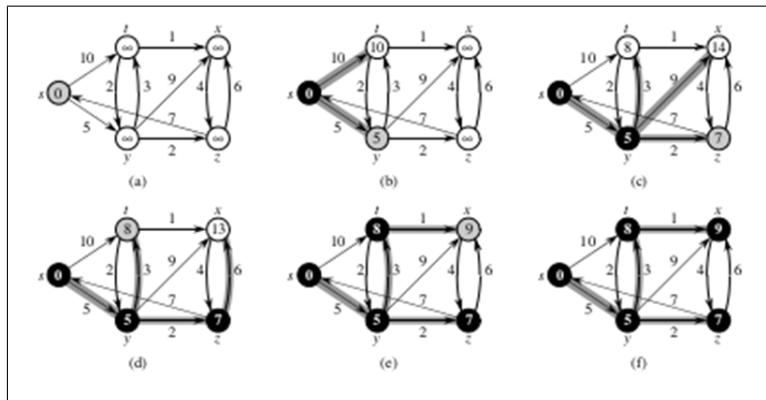

 FIGURE 3.29 – relâchement d'un arc (u, v) de poids $w(u, v) = 2$


FIGURE 3.30 – L'exécution de l'algorithme de Dijkstra

MIN (ligne 5) et DIMINUER-CLÉ (implicite dans RELÂCHER, qui est appelée en ligne 8).

Exercice 10 : Exécuter l'algorithme de Dijkstra sur le graphe orienté de la Figure 3.25, en prenant d'abord comme origine le sommet s , puis le sommet z .

2. Algorithme de Bellman-Ford

Cet algorithme permet de résoudre le problème des plus courts chemins à origine unique dans le cas général où les arcs peuvent avoir un poids négatif.

L'algorithme (Cf. Figure 3.31) utilise la technique de relâchement diminuant progressivement une estimation $d[v]$ du poids d'un plus court chemin depuis l'origine s vers chaque sommet $v \in S$ jusqu'à atteindre la valeur réelle du poids de plus court chemin $\delta(s, v)$.

L'algorithme retourne VRAI si et seulement si le graphe ne contient aucun circuit de longueur strictement négatif accessible depuis l'origine.

```

BELLMAN-FORD( $G, w, s$ )
1 SOURCE-UNIQUE-INITIALISATION( $G, s$ )
2 pour  $i \leftarrow 1$  à  $|S[G]| - 1$ 
3   faire pour chaque arc  $(u, v) \in A[G]$ 
4     faire RELÂCHER( $u, v, w$ )
5   pour chaque arc  $(u, v) \in A[G]$ 
6     faire si  $d[v] > d[u] + w(u, v)$ 
7       alors retourner FAUX
8 retourner VRAI

```

FIGURE 3.31 – L'algorithme de Bellman-Ford

La Figure 3.32 illustre l'exécution de l'algorithme de Bellman-Ford. Le sommet origine est s . Les valeurs de d sont représentées dans les sommets, et les arcs en gris indiquent les valeurs prédécesseur : si l'arc (u, v) est en gris, alors $\pi[v] = u$. Dans cet exemple particulier, chaque passage relâche les arcs dans l'ordre $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$:

- (a) : la situation juste avant le premier passage sur les arcs
- (b)–(e) : la situation après chacun des passages suivants. Les valeurs d et π donnée en partie
- (e) : sont les valeurs finales

L'algorithme de Bellman-Ford retourne VRAI pour cet exemple.

L'algorithme de Bellman-Ford s'exécute en temps $O(SA)$: l'initialisation de la ligne 1 prend $O(S)$; chacun des $|S| - 1$ passages des lignes 2–4 prend $O(A)$; la boucle pour des lignes 5–7 prend $O(A)$.

Exercice 11 : Exécuter l'algorithme de Bellman-Ford sur le graphe orienté de la Figure 3.32, en prenant z pour origine. À chaque passage, relâcher les arcs dans l'ordre $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$, et donner les valeurs de d et π après chaque passage.

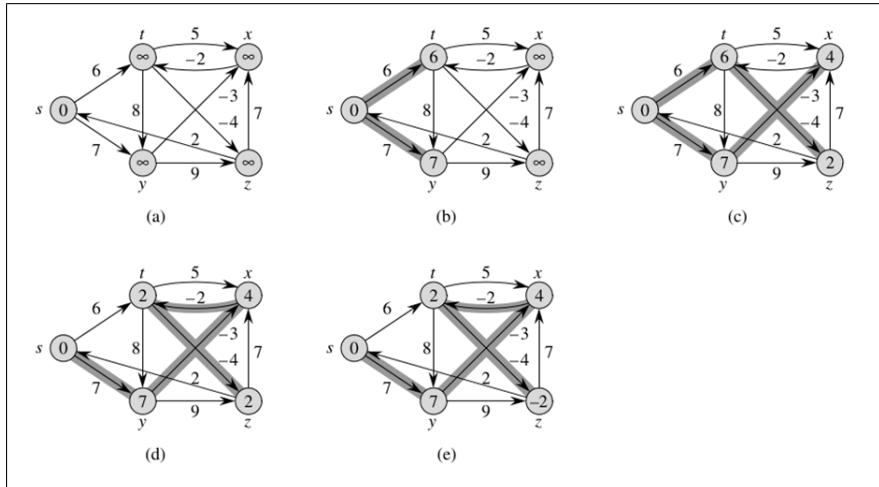


FIGURE 3.32 – L'exécution de l'algorithme de Bellman-Ford

VIII Plus courts chemins pour tout couple de sommets

Cette section traite les algorithmes permettant de rechercher des plus courts chemins entre tous les couples de sommets d'un graphe. On dispose en entrée d'un graphe orienté pondéré $G = (S, A)$ ayant la fonction de pondération $w : A \rightarrow R$ qui associe à chaque arc un poids à valeur réelle. On souhaite déterminer, pour tout couple de sommets $u, v \in S$, un plus court chemin (de longueur minimale) de u à v , où la longueur d'un chemin est la somme des poids des arcs qui le constituent.

On peut résoudre ce problème en exécutant $|S|$ fois un algorithme de plus court chemin à origine unique, en prenant à chaque fois comme origine un nouveau sommet. Si tous les poids d'arc sont positifs ou nuls, on peut utiliser l'algorithme de Dijkstra. Si la file de priorité min est implémentée sous la forme d'un tableau linéaire, le temps d'exécution est en $O(S^3)$. Si l'on autorise les arcs de poids négatif, il faut alors exécuter l'algorithme de Bellman-Ford. Le temps d'exécution résultant est en $O(S^4)$. Serait-il possible de faire mieux.

Contrairement aux algorithmes à origine unique, qui supposent que le graphe est représenté par une liste d'adjacences, la plupart des algorithmes proposés dans ce cadre utilisent une représentation par matrice d'adjacences. Nous supposons que les sommets sont numérotés $1, 2, \dots, |S|$, de sorte que l'entrée est une matrice W de type $n \times n$ qui représente les poids d'arc d'un graphe orienté $G = (S, A)$ à n sommets. Autrement dit, $W = (w_{ij})$, où,

$w_{ij} = 0$ si $i = j$, w_{ij} = le poids de l'arc (i, j) si $i \neq j$ et $(i, j) \in A$, $w_{ij} = \infty$ si $i \neq j$ et $(i, j) \notin A$.

Les arcs de poids négatif sont autorisés, mais on suppose pour l'instant que le graphe en entrée ne contient aucun circuit de longueur strictement négative. Dans le reste de cette section, nous utilisons l'algorithme de Floyd-Warshall.

L'algorithme de Floyd-Warshall s'appuie sur l'observation suivante : si l'on appelle $S = \{1, 2, \dots, n\}$ les sommets de G , on considère un sous-ensemble $\{1, 2, \dots, k\}$ de sommets pour un certain k . Pour un couple quelconque de sommets $i, j \in S$, on considère tous les chemins de i à j dont les sommets intermédiaires appartiennent tous à $\{1, 2, \dots, k\}$, et on note p un chemin de longueur minimale parmi eux. Donc, un plus court chemin du sommet i au sommet j ayant tous les sommets intermédiaires dans l'ensemble $\{1, 2, \dots, k-1\}$ est aussi un plus court chemin de i vers j ayant tous les sommets intermédiaires dans l'ensemble $\{1, 2, \dots, k\}$.

Soit $d_{ij}^{(k)}$ le poids d'un plus court chemin du sommet i au sommet j dont tous les sommets intermédiaires sont dans l'ensemble $\{1, 2, \dots, k\}$. Pour $k = 0$, un chemin de i à j sans sommet intermédiaire de rang supérieur à 0 ne possède en réalité aucun sommet intermédiaire. Il est constitué d'au plus un arc, et on a donc $d_{ij}^{(0)} = w_{ij}$. Pour tout $k \geq 1$, $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$. La matrice $D^{(n)} = d_{ij}^{(n)}$ donne le résultat final, pour tout $i, j \in S$; en effet, quel que soit le chemin, tous les sommets intermédiaires appartiennent à l'ensemble $\{1, 2, \dots, n\}$.

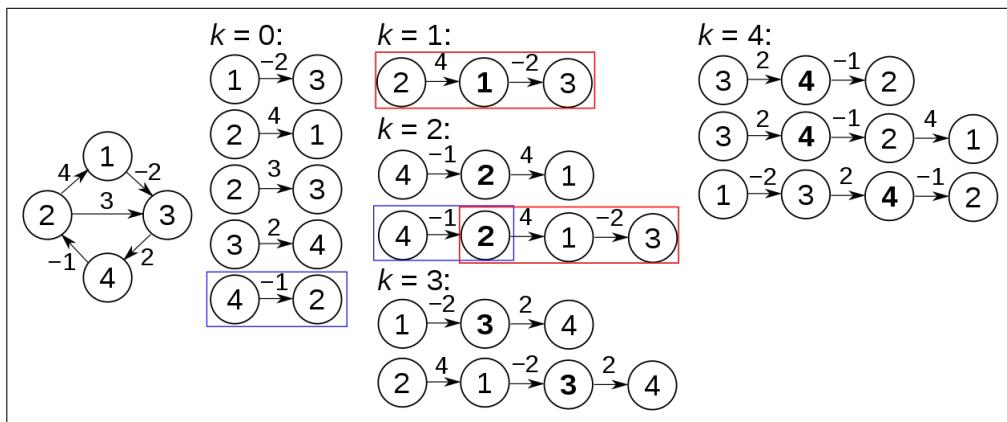


FIGURE 3.33 – Exemple d'un graphe avec des sommets intermédiaires

La Figure 3.34 présente l'algorithme e Floyd-Warshall. La figure 3.35 montre un graphe et les matrices calculées.

```

FLOYD-WARSHALL( $W$ )
1    $n \leftarrow \text{lignes}[W]$ 
2    $D^{(0)} \leftarrow W$ 
3   pour  $k \leftarrow 1$  à  $n$ 
4     faire pour  $i \leftarrow 1$  à  $n$ 
5       faire pour  $j \leftarrow 1$  à  $n$ 
6         faire  $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
7   retourner  $D^{(n)}$ 

```

FIGURE 3.34 – L'algorithme de Floyd-Warshall

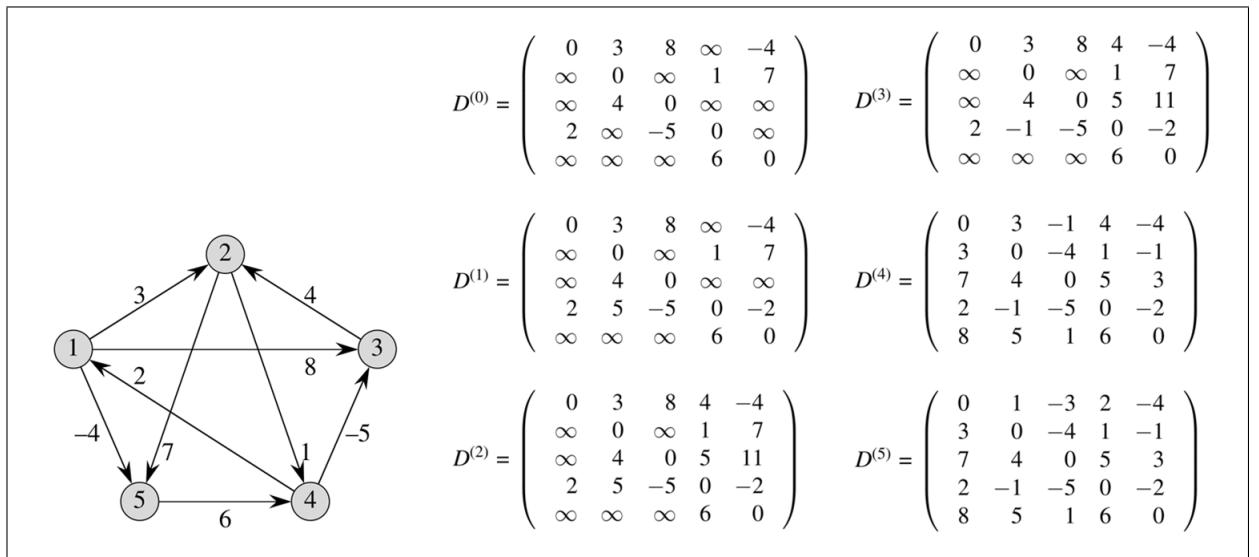


FIGURE 3.35 – Exemple

IX Coloration de graphe

Soit G un graphe représentant le réseau de communication d'un système distribué. On désire colorier ce graphe selon le principe de coloriage suivant : deux sites voisins immédiats sont de couleurs différentes. Le nombre chromatique c est le plus petit nombre de couleurs nécessaires pour colorier le graphe. Donc,

- Si un graphe a n sommets, alors $c \leq n$.
- Si un graphe de n sommets est complet, alors $c = n$.
- Si le graphe admet un sous-graphe complet de m sommets (ordre), alors $m \leq c \leq n$.

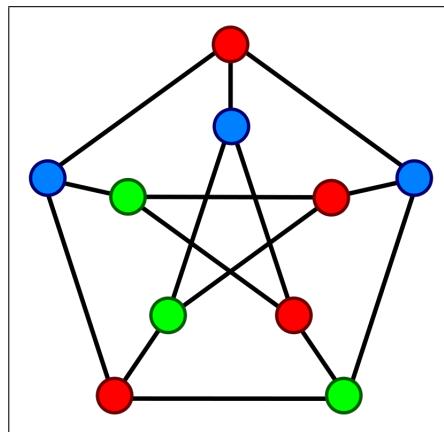


FIGURE 3.36 – Exemple

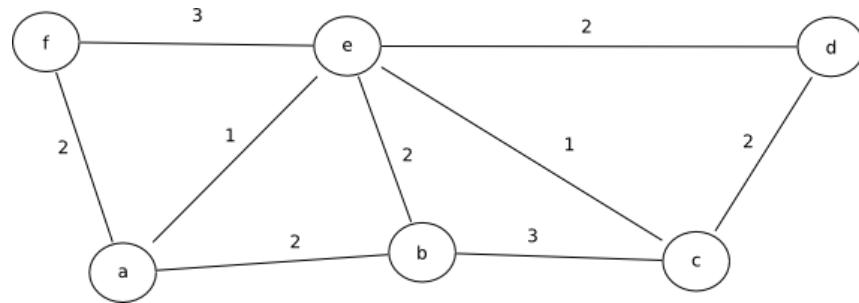
L'algorithme de Welsh-Powel permet de colorier le graphe avec le minimum de couleur. L'idée est que les sommets ayant beaucoup de voisins seront plus difficiles à colorer, et donc il faut les colorer en premier. Le principe de cet algorithme est comme suit :

1. Repérer le degré de chaque sommet.
2. Ranger les sommets par ordre de degrés décroissants (dans certains cas plusieurs possibilités).
3. Attribuer au premier sommet (A) de la liste une couleur.
4. Suivre la liste en attribuant la même couleur au premier sommet (B) qui ne soit pas adjacent à (A).
5. Suivre (si possible) la liste jusqu'au prochain sommet (C) qui ne soit adjacent ni à A ni à B.
6. Continuer jusqu'à ce que la liste soit finie.
7. Prendre une deuxième couleur pour le premier sommet (D) non encore coloré de la liste.
8. Répéter les opérations 4 à 7.
9. Continuer jusqu'à avoir coloré tous les sommets.

X Travaux dirigés

Exercice 1 : algorithme de Prim / Kruskal

Appliquez l'algorithme de Prim sur le graphe ci-dessous, pour obtenir un arbre de poids minimum. Le départ est fixé sur le sommet a .

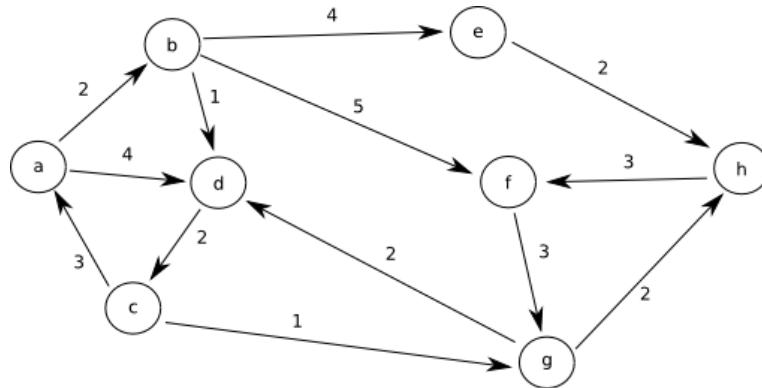


Appliquez l'algorithme de Kruskal sur le graphe précédent.

Exercice 2 : Algorithme de Dijkstra

L'algorithme de Dijkstra s'applique à un graphe orienté.

Appliquez l'algorithme sur le graphe orienté suivant, afin d'obtenir les plus courts chemins issus du sommet a .



Exercice 3

Soit la matrice suivante :

$$A = \begin{pmatrix} 0 & 6 & 7 & +\infty & +\infty \\ +\infty & 0 & 8 & 5 & -4 \\ +\infty & +\infty & 0 & -3 & 9 \\ +\infty & -2 & +\infty & 0 & +\infty \\ 2 & +\infty & +\infty & 7 & 0 \end{pmatrix}$$

Donnez le graphe associé à la matrice A.

Appliquez l'algorithme de Bellman-Ford sur le graphe obtenu en considérant le sommet 1 comme origine.

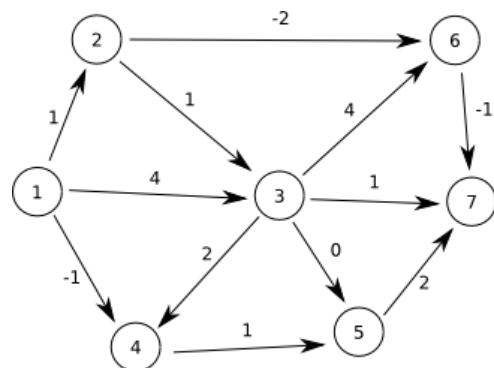
Donnez l'arborescence des plus courts chemins obtenus.

Exercice 4 : Calcul du plus long chemin

Adaptation de l'algorithme de Bellman pour calculer le plus long chemin dans un graphe orienté sans circuit.

Soit $G = (S, A, W)$ un graphe orienté pondéré, $s \in S$ est un sommet source. Notons par $L(x)$ la longueur d'un plus long chemin d'origine s et d'extrémité x.

1. Montrer qu'il existe un plus long chemin de s à x.
2. Adapter l'algorithme de Bellman pour calculer les plus longs chemins d'origine S.
3. Appliquer l'algorithme au graphe suivant.

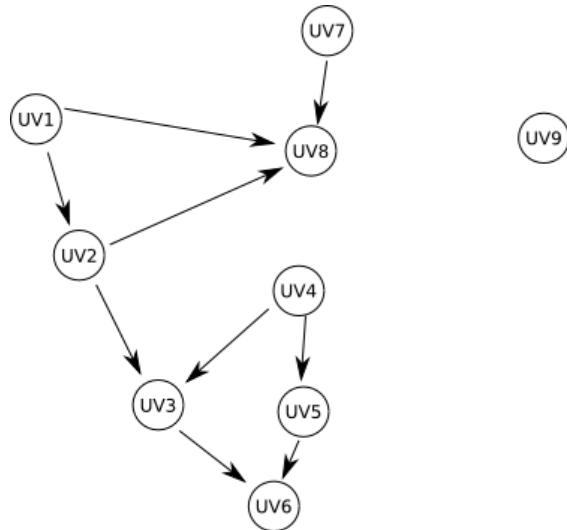


Exercice 5 : Tri topologique

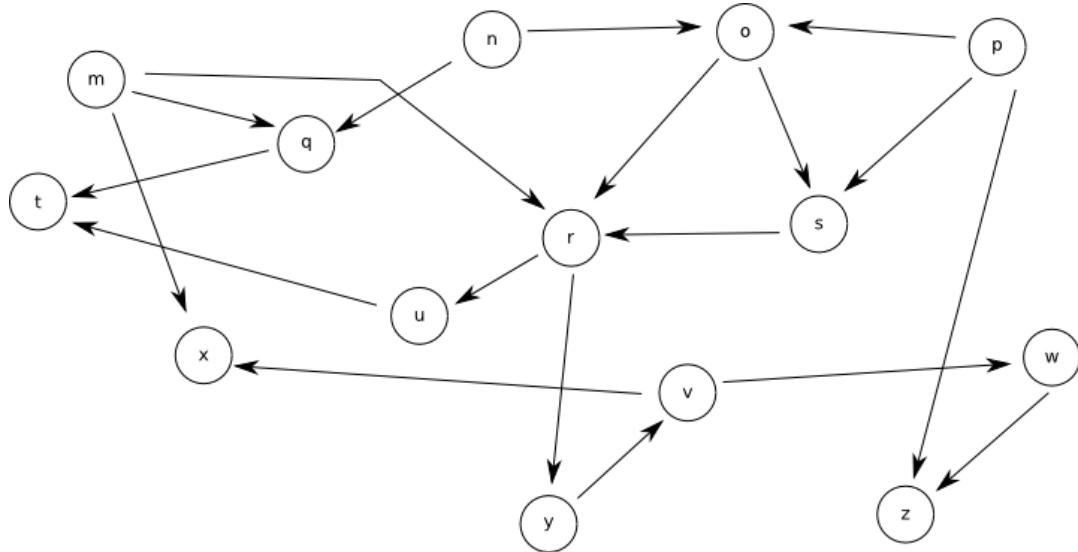
Soit $G(S, A)$ un graphe orienté sans circuit. L'algorithme de tri topologique de G consiste à ordonner linéairement tous les sommets de G tel que $\forall(u, v) \in A \times A$, u apparaît avant v dans l'ordre obtenu.

Donnez l'algorithme ainsi que son coût.

Utiliser l'algorithme PP pour le tri topologique du graphe suivant :



Utiliser l'algorithme PP pour le tri topologique du graphe suivant en commençant par le sommet m :



Exercice 6 : CFG

Une CFG (composante fortement connexe) d'un graphe orienté $G = (S, A)$ est l'ensemble maximal des sommets $U \subseteq S$ tel que $\forall(u, v) \in U$, nous avons u lié à v et v lié à u (directement ou indirectement).

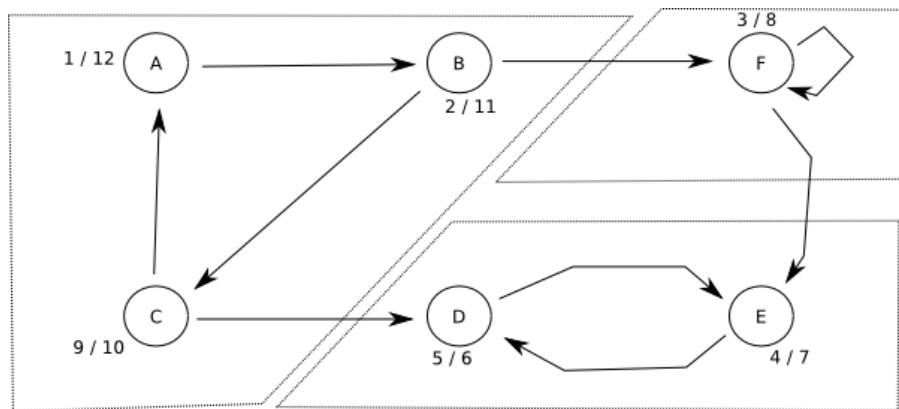
L'algorithme $\in FC$ consiste à décomposer le graphe en ses composantes fortement connexes, en utilisant le parcours en profondeur 2 fois.

Donnez cet algorithme ainsi que son coût.

Montrer que cette série de sommets :

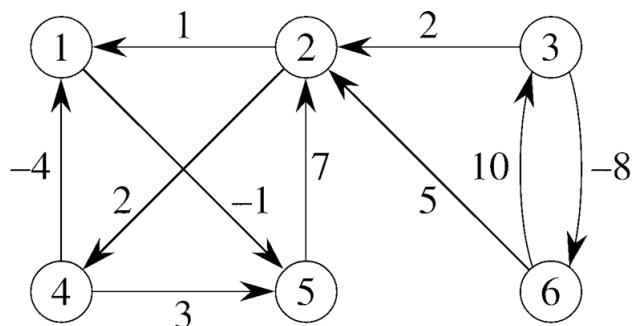
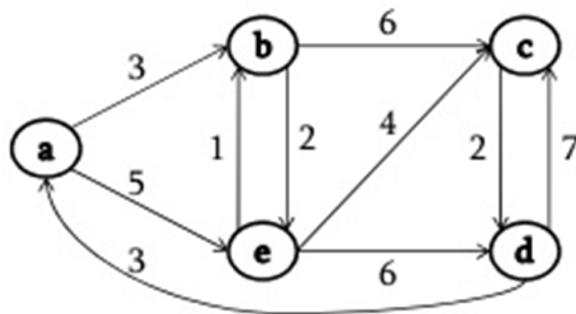
$$\mathbf{A} \rightarrow \mathbf{B} \rightarrow \mathbf{C} \ F \rightarrow \mathbf{E} \rightarrow \mathbf{D}$$

sera obtenue après l'exécution de l'algorithme sur le graphe suivant :



Exercice 7 : Algorithme de Floyd-Warshall

Utiliser l'algorithme FW pour calculer les chemins les plus courts sur les deux graphes suivants :



XI Travaux pratiques

TP1 : Algorithmes de parcours

- Implémenter le parcours en profondeur et le parcours en largeur d'un graphe quelconque donné.
- Comparer les coûts expérimentaux des deux parcours.
- Que peut-on conclure ?

TP2 : Tri topologique

- Implémenter l'algorithme permettant d'effectuer le Tri topologique d'un graphe orienté, en l'appliquant par exemple aux choix des modules en fonction de leurs ordres de précédences.
- Calculer le coût expérimental de cet algorithme et le comparer avec le coût expérimental de l'algorithme de parcours en profondeur.
- Que peut-on conclure ?

TP3 : Composantes fortement connexes

- Implémenter l'algorithme permettant de trouver et imprimer les composantes fortement connexes d'un graphe orienté.
- Calculer le coût expérimental de cet algorithme et le comparer avec le coût expérimental de l'algorithme de parcours en profondeur.
- Que peut-on conclure ?

TP4 : Arbre de recouvrement minimal

- Implémenter les deux algorithmes de construction d'un arbre couvrant minimum d'un graphe.
- Comparer les coûts expérimentaux de ces deux algorithmes.
- Que peut-on conclure ?

TP5 : Plus courts chemins

- Implémenter l'algorithme de Dijkstra.
- Implémenter l'algorithme de Bellman-Ford.
- Implémenter l'algorithme de Floyd-Warshall.

Documents à fournir (via la plateforme) : rapport (max 5 pages),

les sources et les jeux de tests. Ils seront rendus dans une archive (nom-prenom-TPx.zip/gz).