

CHAPITRE III

Structures de Données Avancées

Structure de Graphe

Le graphe est une représentation de données importante car elle permet de modéliser tous les problèmes manipulant des graphes comme données. Les réseaux de communication téléphonique, Internet et les réseaux routiers ou aériens sont des exemples concrets de graphes. La théorie des graphes permet d'étudier les graphes sous tous leurs aspects mais également des problèmes typiques de graphes tels que la recherche d'un circuit hamiltonien, et d'un circuit de poids minimum. Une grande partie de ces problèmes s'avèrent être complexes dans le sens où la complexité de calcul est exponentielle.

Dans ce qui suit, une série de concepts inhérents à la structure de graphe sont définis.

Graphe orienté

Un graphe est défini par deux ensembles : S un ensemble fini de sommets et A une relation binaire sur S ou un ensemble d'arcs. Un arc est un couple de sommets reliés entre eux dans un seul sens. Si G est un graphe orienté, on note :

$$G = \{S, A\},$$

$$S = \{S_1, S_2, \dots, S_n\}$$

$$A = \{(S_i, S_j), \text{ tel que } S_i \in S, S_j \in S \text{ et il existe un arc de } S_i \text{ à } S_j\}$$

Un exemple de graphe orienté est montré sur la figure 3.1. Dans ce graphe, les ensembles S et A sont explicités comme suit :

$$S = \{S_1, S_2, S_3, S_4, S_5\}$$

$$A = \{(S_1, S_3), (S_1, S_4), (S_2, S_1), (S_3, S_4), (S_4, S_5), (S_5, S_1), (S_5, S_5)\}$$

Ordre d'un graphe

C'est le nombre de sommets du graphe. L'ordre du graphe de l'exemple de la figure 3.1 est égal à 5.

Sommets adjacents

Deux sommets sont adjacents s'ils sont reliés par un même arc. S_3 et S_4 sont adjacents dans le graphe de l'exemple de la figure 3.1.

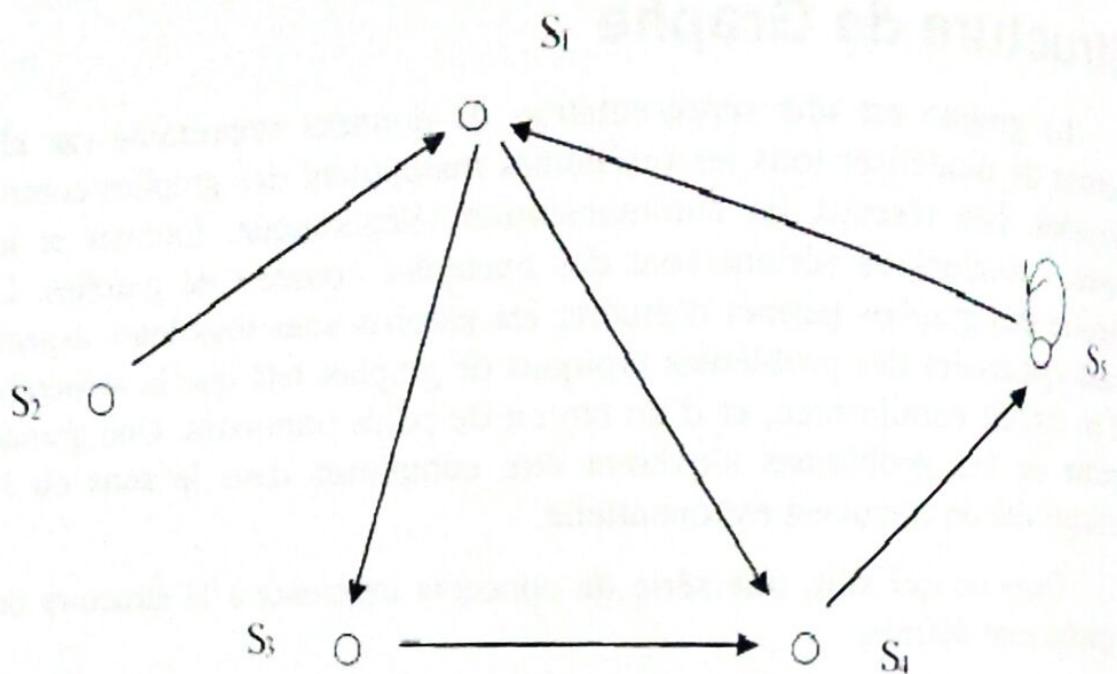


Figure 3.1. Un exemple de graphe orienté.

Boucle

Une boucle dans un graphe orienté est un arc qui relie un sommet à lui-même. Dans l'exemple précédent, (S_5, S_5) est une boucle.

Degré d'un sommet

Dans un graphe orienté, le degré sortant d'un sommet est le nombre d'arcs qui le relient à d'autres sommets. Le degré sortant de S_4 de l'exemple de la figure 3.1 est égal à 1.

Le degré entrant d'un sommet est le nombre d'arcs provenant d'autres sommets et aboutissant à ce sommet. Le degré entrant de S_4 de l'exemple de la figure 3.1 est égal à 2.

Le degré global d'un sommet est égal à la somme du degré sortant et du degré entrant.

Chemin

Un chemin allant d'un sommet S_i à un sommet S_j est une séquence de sommets telle que :

- Deux sommets consécutifs relient un arc du graphe.
- le premier sommet est S_i et le dernier est S_j .

- un arc du chemin commence par le sommet extrémité de l'arc précédent et se termine par le sommet origine de l'arc suivant.

La longueur du chemin est égale au nombre d'arcs le constituant. (S_1, S_3, S_4, S_5) est un chemin allant de S_1 à S_5 du graphe de l'exemple de la figure 3.1. Sa longueur est égale à 3.

Un chemin est élémentaire si ses sommets sont tous distincts. (S_1, S_3, S_4, S_5) est un chemin élémentaire.

Circuit

Un circuit est un chemin qui va d'un sommet et s'arrête au même sommet. Un exemple de circuit dans le graphe précédent est : $(S_1, S_3, S_4, S_5, S_1)$ qui commence par S_1 et s'arrête à S_1 . C'est un circuit élémentaire car ses sommets sont tous distincts.

Graphe non orienté

Un graphe non orienté se présente comme un graphe orienté mais dans lequel il n'existe pas de sens ou d'orientation pour les arcs, on parle alors d'arêtes. Le graphe non orienté de l'exemple précédent est montré sur la figure 3.2. Dans ce graphe, les ensembles S et A sont explicités comme suit :

$$S = \{S_1, S_2, S_3, S_4, S_5\}$$

$$A = \{(S_1, S_2), (S_1, S_3), (S_1, S_4), (S_1, S_5), (S_3, S_4), (S_4, S_5)\}$$

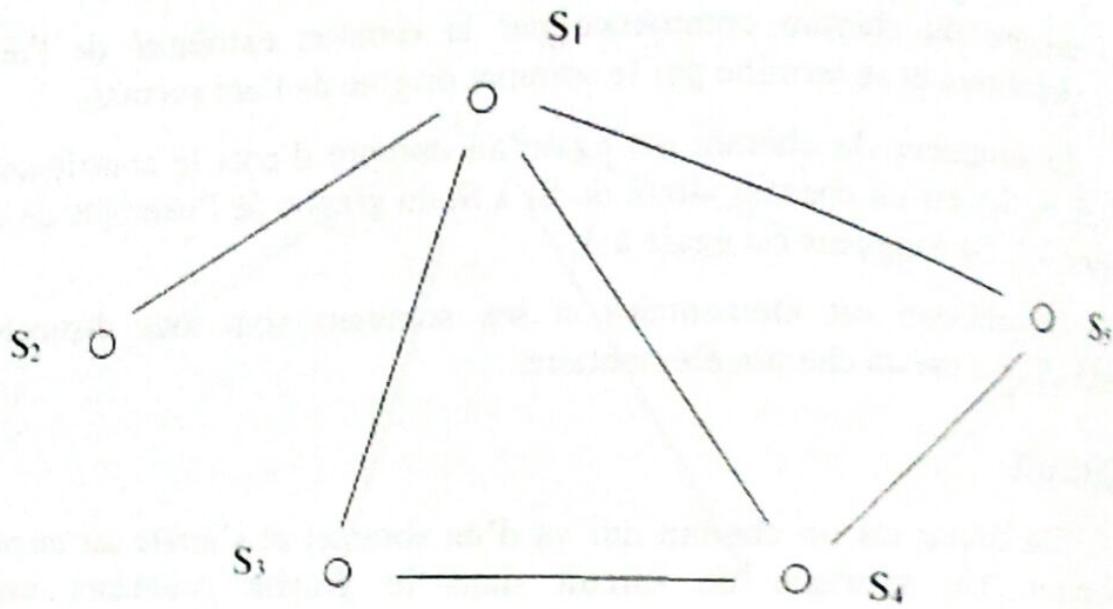


Figure 3.2. Exemple de graphe non orienté.

Ordre d'un graphe

C'est le nombre de sommets du graphe. L'ordre du graphe de l'exemple est égal à 5.

Sommets adjacents

Deux sommets sont adjacents s'ils sont reliés par une même arête. S_1 et S_4 sont adjacents dans le graphe de la figure 3.2.

Boucle

Dans un graphe non orienté, il n'existe pas de boucle.

Degré d'un sommet

La notion de degré entrant et de degré sortant pour un sommet n'existe pas dans un graphe non orienté. Le degré d'un sommet est le nombre d'arêtes dont il est l'une de ses extrémités. Le degré du sommet S_1 est égal à 4 pour le graphe de la figure 3.2.

La somme des degrés de tous les sommets d'un graphe est égal au double du nombre total d'arêtes, c'est donc un nombre pair.

Chaîne

Une chaîne dans un graphe non orienté est l'équivalent du chemin dans un graphe orienté. $(S_2, S_1, S_3, S_4, S_5)$ est un exemple de chaîne de longueur égale à 4 pour le graphe de la figure 3.2.

Une chaîne est élémentaire si tous ses sommets sont distincts. $(S_2, S_1, S_3, S_4, S_5)$ est une chaîne élémentaire.

cycle

C'est l'équivalent du circuit pour les graphes orientés. $(S_1, S_3, S_4, S_5, S_1)$ est un exemple de cycle pour le graphe de la figure 3.2. C'est un cycle élémentaire car ses sommets sont tous distincts.

Graphe connexe

C'est un graphe dans lequel chaque paire de sommets est reliée par une chaîne. L'exemple de graphe de la figure 3.2 est un graphe connexe.

Structures de données pour graphes

Un exemple concret de graphe est le réseau routier reliant certaines grandes villes algériennes comme montré sur la figure 3.3.

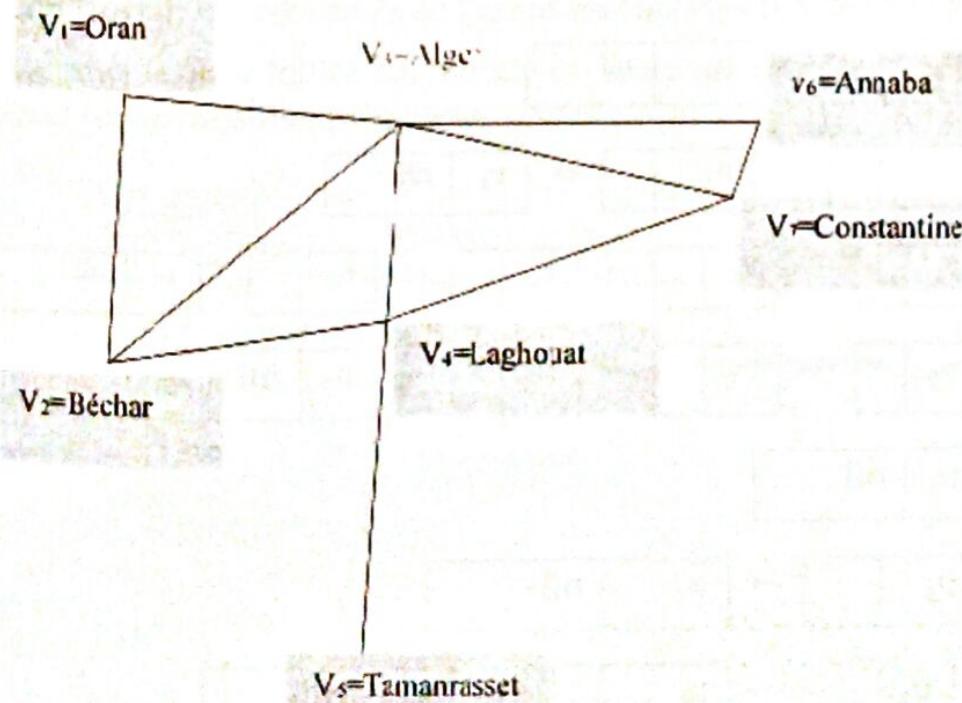


Figure 3.3. Graphe du réseau routier de quelques villes d'Algérie

$S = \{\text{Oran}, \text{Béchar}, \text{Alger}, \text{Laghouat}, \text{Tamanrasset}, \text{Annaba}, \text{Constantine}\}$ et $|A| = 10$.

Il s'agit d'un graphe non orienté dans lequel on suppose que, entre deux villes il existe une seule route qui va dans les deux sens.

Un graphe peut être représenté à l'aide d'une matrice appelée matrice d'adjacence. La matrice d'adjacence du graphe de la figure 3.3 est montrée sur la figure 3.4.

\	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	1	0	0	0	0
v_2	1	0	1	1	0	0	0
v_3	1	1	0	1	0	1	1
v_4	0	1	1	0	1	0	1
v_5	0	0	0	1	0	0	0
v_6	0	0	1	0	0	0	1
v_7	0	0	1	1	0	1	0

Figure 3.4. Matrice d'adjacence du graphe de la figure 3.3.

La matrice d'adjacence peut s'avérer parfois très coûteuse en termes d'espace mémoire. Une deuxième représentation possible qui peut contourner ce problème est de lister tous les successeurs de chacun des sommets. Pour notre exemple les listes des successeurs des sommets sont montrées sur la figure 3.5.

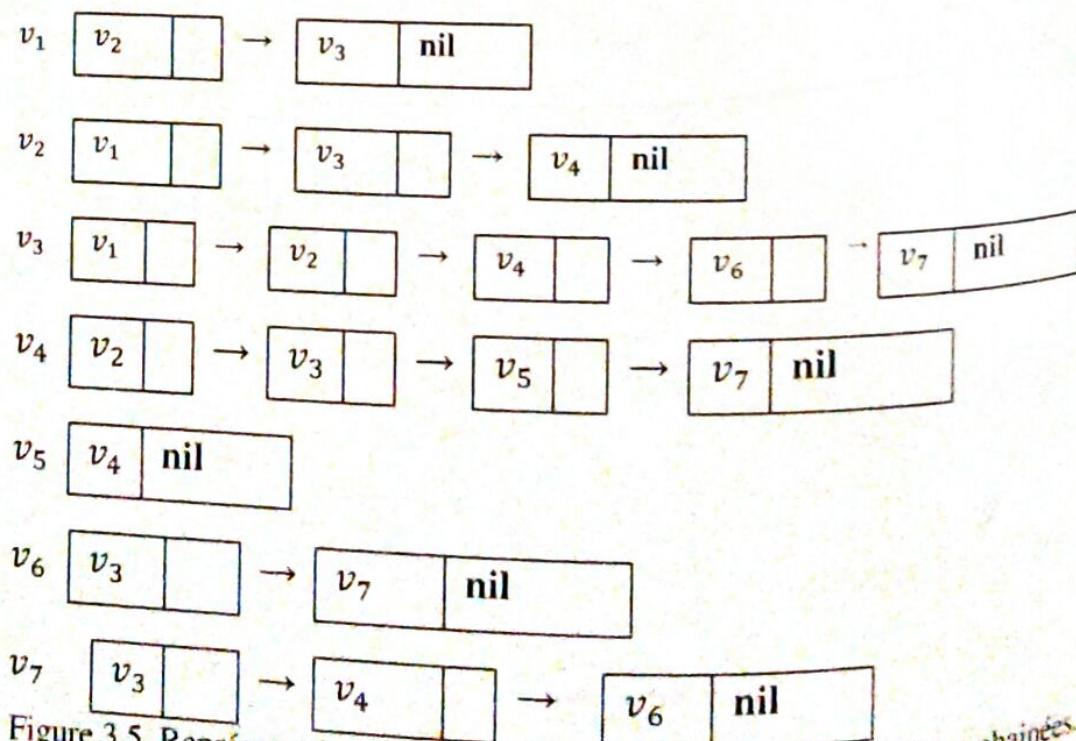


Figure 3.5. Représentation du graphe de la figure 3.3 à l'aide de listes chainées.

Le graphe de la figure 3.3 est un graphe non orienté appelé parfois multi-graphe. La matrice d'adjacence est symétrique dans ce cas et peut donc être représentée par sa moitié gauche uniquement. Dans certaines applications, des poids peuvent être associés aux arêtes, on parle dans ce cas de graphe pondéré.

Les opérations qu'on peut effectuer sur des graphes sont nombreuses, c'est là toute la raison du développement de l'algorithme des graphes. Dans le chapitre 6, quelques exemples de traitements seront présentés comme les parcours d'un graphe en profondeur d'abord et en largeur d'abord.

Structure d'arbre

Un arbre est un cas particulier de graphe non orienté connexe qui n'admet pas de cycle et est par conséquent représenté de la même manière qu'un graphe. Un sommet d'un arbre est appelé **nœud**.

Le nœud qui précède un nœud x est appelé **prédécesseur** ou **parent** de x . Les nœuds qui suivent un nœud x sont appelés **successeur** ou **fils** de x . Le **degré d'un nœud x** est égal au nombre de successeurs de x .

Un arbre possède également un nœud particulier qui n'a pas de prédécesseurs. Il s'agit de la **racine** de l'arbre et l'arbre est alors dit **enraciné**. Les nœuds extrémités de l'arbre sont appelés **feuilles** de l'arbre.

Le schéma de la figure 3.6 montre un arbre qui représente une partie du réseau routier algérien précédent.

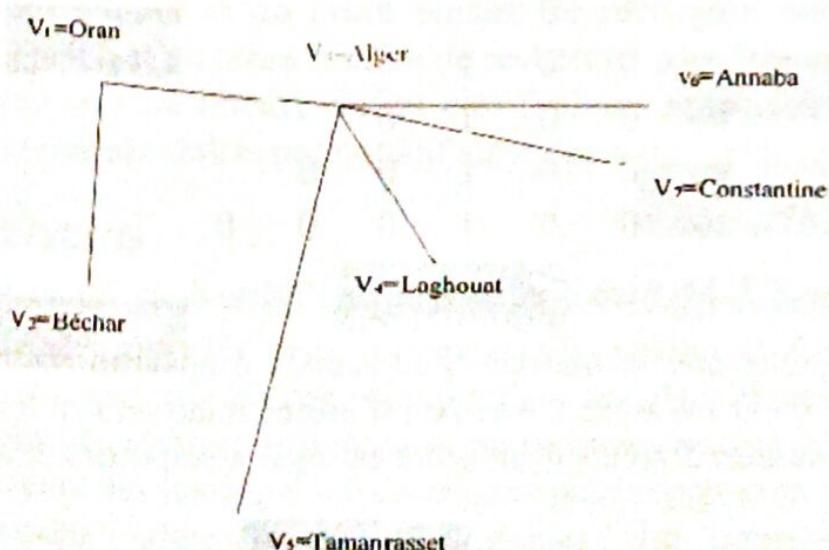


Figure 3.6. Exemple d'arbre.

Alger peut être considérée comme la racine de l'arbre alors que Béchar, Tamanrasset, Laghouat, Constantine et Annaba constituent les feuilles de l'arbre.

Profondeur d'un nœud

La notion de profondeur d'un nœud est importante dans certains traitements effectués sur les arbres tels que la recherche d'un élément dans un arbre.

La profondeur d'un nœud est souvent définie de manière récursive comme suit :

$$\begin{cases} \text{profondeur(racine)} = 0 \\ \text{profondeur}(x) = \text{profondeur}(\text{parent}(x)) + 1 \end{cases}$$

La profondeur ou hauteur de l'arbre est alors définie comme étant la plus grande profondeur de ses feuilles.

$$\text{profondeur(arbre)} = \underset{x \text{ est une feuille}}{\text{maximum}} (\text{profondeur}(x))$$

Structures de données pour arbres

La matrice d'adjacence de l'arbre de la figure 3.6 est exhibée sur la figure 3.7.

\	v_1	v_2	v_3	v_4	v_5	v_6
v_2	1					
v_3	1	0				
v_4	0	0	1			
v_5	0	0	1	0		
v_6	0	0	1	0	0	
v_7	0	0	1	0	0	0

Figure 3.7. Matrice d'adjacence de l'arbre de la figure 3.6.

Remarquons que la matrice d'adjacence d'un arbre fait apparaître beaucoup de 0 car un arbre a très peu d'arêtes relativement à un graphe. En effet, le nombre d'arêtes d'un arbre est égal à son ordre diminué de 1, c'est-à-dire :

$$|A| = |S| - 1$$

La représentation des listes des successeurs des sommets est plus adéquate. La figure 3.8 montre cette structure de données pour le graphe de la figure 3.6.

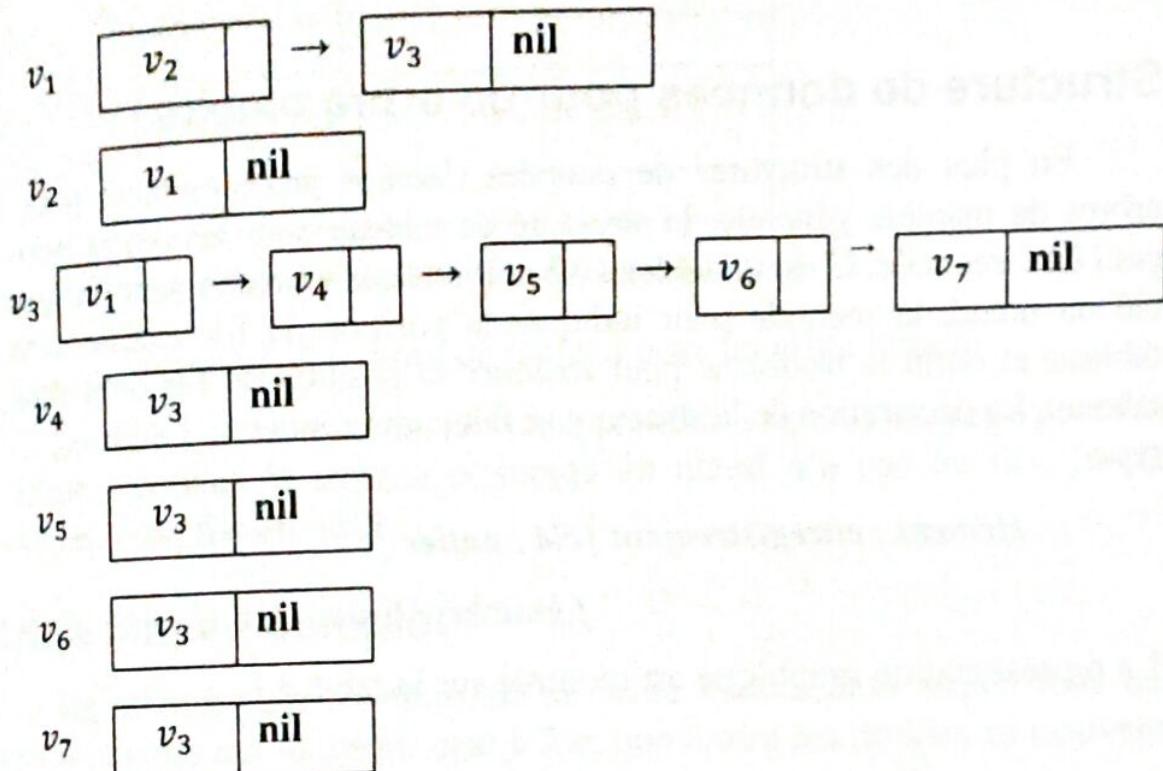


Figure 3.8. Structure de données dynamiques pour l'arbre de la figure 3.6.

Arbre binaire et arbre binaire de recherche

Un arbre binaire et un arbre binaire de recherche ont une même structure. Cependant un arbre binaire de recherche peut être vu comme un cas particulier d'arbre binaire où les clés (valeurs associées aux nœuds) respectent un certain ordre, permettant ainsi des traitements rapides.

Arbre binaire

Un arbre binaire est un arbre dans lequel chaque nœud a au plus deux successeurs appelés respectivement fils gauche et fils droit. Les arbres binaires sont utilisés par exemple dans les algorithmes de tri pour concevoir des algorithmes efficaces et performants mais également dans les compilateurs des langages informatiques pour représenter par exemple les expressions arithmétiques et logiques. La représentation de

l'expression arithmétique $\frac{-3+\sqrt{7-4*5*6}}{2*8}$ en l'occurrence, est l'arbre abstrait de la figure 3.9.

Un arbre binaire est représenté soit par une liste dynamique soit par un tableau.

Structure de données pour un arbre binaire

En plus des structures de données décrites précédemment pour les arbres de manière générale, la structure de tableau pour les arbres binaires peut être très utile. C'est un tableau à 3 colonnes, la première pour indiquer la clé du nœud, la seconde pour indiquer la position du fils gauche dans le tableau et enfin la troisième pour indiquer la position du fils droit dans le tableau. La déclaration de la structure se fait comme suit :

type

élément : enregistrement {clé : entier ;

f gauche, f droit : 1..n ;}

La représentation graphique est montrée sur la table 3.1.

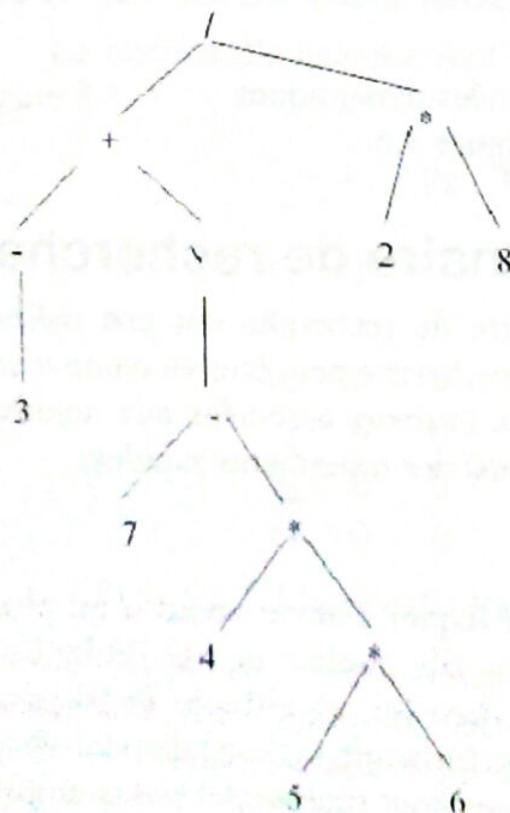


Figure 3.9. Arbre binaire représentant une expression arithmétique.

clé	f gauche	f droit
1		
2		
3		
4		
.		
.		
.		
n		

Table 3.1. Structure de tableau pour un arbre binaire

Lorsque l'arbre est vide, la racine prend la valeur 0 car cette position n'existe pas dans le tableau et lorsqu'un nœud n'a pas de fils, la clé associée à son fils est nulle.

Arbre binaire complet

Un arbre binaire complet est un arbre binaire dans lequel tous les nœuds internes ont un degré égal à 2 et que toutes les feuilles se trouvent à la même profondeur.

Propriétés d'un arbre binaire complet

Si p est la profondeur d'un arbre binaire complet, le nombre des nœuds n est égal à :

$$1 + 2 + 2^2 * \dots * 2^p = \sum_{i=0}^{i=p} 2^i = 2^{p+1}-1$$

$$n = 2^{p+1}-1$$

p s'exprime en fonction n de la manière suivante :

$$n = 2^{p+1}-1$$

$$2^{p+1} = n + 1$$

$$p + 1 = \log_2(n + 1)$$

$$p = \log_2(n + 1) - 1$$

Parcours d'un arbre binaire

Le parcours des nœuds d'un arbre binaire peut se faire selon trois stratégies : le parcours préfixé, le parcours infixé et le parcours post-fixé.

Parcours préfixé

Le parcours d'un arbre binaire selon l'ordre préfixé est défini comme suit :

- 1) Visiter la racine.
- 2) Parcourir selon l'ordre préfixé, le sous arbre dont la racine est le fils gauche de la racine de l'arbre.
- 3) Parcourir selon l'ordre préfixé, le sous arbre dont la racine est le fils droit de la racine de l'arbre.

Version récursive

La version récursive du parcours préfixé d'un arbre binaire découle directement de la définition.

Parcours préfixé en utilisant la structure de tableau

Algorithme 3.1. Parcours préfixé avec structure statique

*procédure parcours-préfixé ;
entrée : arbre : tableau [1..n] d'élément ; nœud : 0..n ;
sortie : une séquence des clés des nœuds selon l'ordre préfixé ;*

début

*si (nœud ≠ 0) alors
 début afficher (arbre[nœud].clé) ;
 parcours-préfixé (arbre[nœud].f gauche) ;
 parcours-préfixé (arbre[nœud].f droit) ;
 fin*

fin ;

Programme principal

var

racine : 0..n ;

début
parcours-préfixé (racine) ;

fin

Une racine nulle exprime un arbre vide.

Exemple

Considérer la structure d'arbre suivante :

nœud	clé	f gauche	f droit
1	9	2	3
2	13	4	5
3	3	6	7
4	20	0	0
5	17	8	9
6	6	0	0
7	42	10	11
8	4	0	0
9	23	0	0
10	7	0	0
11	11	0	0

Table 3.2. Structure de tableau pour un arbre binaire.

La valeur 0 dans le tableau indique l'absence du fils. Le schéma de l'arbre est montré sur la figure 3.10

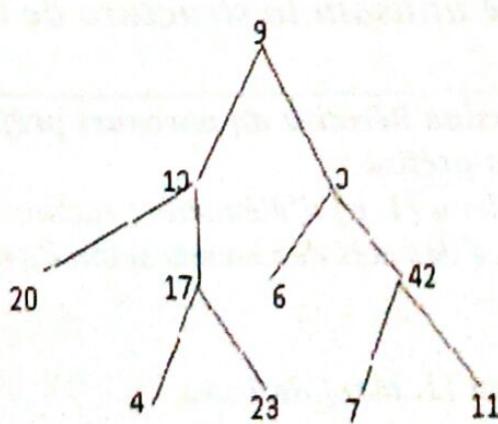


Figure 3.10 version schématique de l'arbre binaire représenté par la table 3.2.
L'ordre préfixé des nœuds de l'arbre est : 9, 13, 20, 17, 4, 23, 3, 6, 42, 7, 11.

Parcours préfixé en utilisant la structure dynamique de liste

Algorithme 3.2. Parcours préfixé avec structure dynamique

Programme principal

type

*élément : enregistrement {clé : entier ;
f gauche, f droit : ↑élément;}*

début

parcours-préfixé(racine) ;

fin

procédure parcours-préfixé ;

entrée : nœud : ↑élément;

sortie : une séquence des clés des nœuds selon l'ordre préfixé ;

début

si (nœud <> nil) alors

début afficher (nœud↑.clé) ;

parcours-préfixé (nœud↑.f gauche) ;

parcours-préfixé (nœud↑.f droit) ;

fin

fin ;

Version itérative

Parcours préfixé utilisant la structure de tableau

Algorithme 3.3. version itérative du parcours préfixé

procédure parcours-préfixé ;

entrée : arbre : tableau [1..n] d'éléments ; racine : 0..n ;

sortie : une séquence des clés des nœuds selon l'ordre préfixé ;

var

x : 0..n ;

pile : tableau [1..max] de 1..n ;

sommet : 0..n ;

début

empiler (racine) ;
tant que (pile non vide) faire

début

x := dépiler() ;

afficher (arbre[x].clé) ;

si (arbre[x].f droit ≠ 0) alors empiler (arbre[x].f droit) ;

si (arbre[x].f gauche ≠ 0) alors empiler (arbre[x].f gauche) ;

fin

fin

Le parcours préfixé en utilisant la structure de liste dynamique est facile à déduire à partir des procédures décrites précédemment.

Parcours infixé

Le parcours d'un arbre binaire selon l'ordre infixé est défini comme suit :

- Parcourir selon l'ordre infixé, le sous arbre dont la racine est le fils gauche de la racine
- Visiter la racine
- Parcourir selon l'ordre infixé, le sous arbre dont la racine est le fils droit de la racine

Comme pour le parcours préfixé, la version récursive du parcours infixé d'un arbre binaire se déduit directement à partir de la définition.

Version récursive

Parcours infixé utilisant la structure de tableau

Algorithme 3.4. Parcours infixé avec structure statique

Programme principal

var

racine : 0..n ;

début

parcours-infixé (racine) ;

fin

procédure parcours-infixé ;

entrée : arbre : tableau [1..n] d'éléments ; nœud : 0..n ;

sortie : une séquence des clés des nœuds selon l'ordre infixé ;

début

si (nœud ≠ 0) alors

début

parcours-infixé (arbre [nœud].f gauche) ;

afficher (arbre [nœud].clé) ;

parcours-infixé (arbre [nœud].f droit) ;

fin

fin ;

Le parcours infixé de l'arbre de la figure 3.10 est :

20, 13, 4, 17, 23, 9, 6, 3, 7, 42, 11.

Parcours post-fixé

Le parcours selon l'ordre post-fixé est défini comme suit :

- a. Parcourir selon l'ordre post-fixé, le sous arbre dont la racine est le fils gauche de la racine de l'arbre.
- b. Parcourir selon l'ordre post-fixé, le sous arbre dont la racine est le fils droit de la racine de l'arbre.
- c. Visiter la racine de l'arbre.

Version récursive

Parcours post-fixé utilisant la structure de tableau

Algorithme 3.5. Parcours post-fixé avec structure statique

Programme principal

var
racine : 0..n ;

début parcours-post-fixé (racine) ;

fin

procédure parcours-post-fixé ;

entrée : arbre : tableau [1..n] d'éléments ; nœud : 0..n ;

sortie : une séquence des clés des nœuds selon l'ordre post-fixé ;

début si (nœud ≠ 0) **alors**

début parcours-post-fixé (arbre [nœud].f gauche) ;

parcours-post-fixé (arbre [nœud].f droit) ;

afficher (arbre [nœud].clé) ;

fin

fin ;

Parcours post-fixé utilisant la structure dynamique de liste

Algorithme 3.6. Parcours post-fixé avec structure dynamique

Programme principal

var

racine : ↑ élément ;

début parcours-post-fixé(racine) ;

fin

procédure parcours-post-fixé(nœud : ↑ élément) ;

début si (nœud ≠ nil) **alors**

début parcours-post-fixé (nœud↑.f gauche) ;

parcours-post-fixé (nœud↑.f droit) ;

afficher (nœud↑.clé) ;

fin

fin ;

Le parcours post-fixé de l'arbre de la figure 3.10 est :

20, 4, 23, 17, 13, 6, 7, 11, 42, 3, 9.

La complexité de chacun de ces parcours est $O(n)$ car tous les nœuds de l'arbre sont visités.

Arbre binaire de recherche

Les arbres binaires sont d'une utilité importante pour l'écriture d'algorithmes efficaces. Il existe un arbre binaire particulier qui permet de maintenir un ordre de parcours de manière à rechercher facilement un élément en un temps court. Il s'agit de l'arbre binaire de recherche et dont les opérations sont :

- a. Rechercher un élément x dans l'arbre.
- b. Rechercher le minimum de l'arbre.
- c. Rechercher le maximum de l'arbre.
- d. Insérer un élément dans l'arbre.
- e. Supprimer un élément x de l'arbre.

Certes, ces opérations peuvent être menées dans un arbre binaire ordinaire. Seulement, l'avantage d'utiliser un arbre binaire de recherche réside dans le temps de calcul de ces opérations qui est plus court. On peut faire l'analogie avec la structure de tableau (ou liste) où on distingue des tableaux (ou listes) ordinaires et des tableaux (ou listes) triées.

Définition 3.1.

Un arbre binaire de recherche est un arbre binaire tel que chaque nœud de l'arbre a une clé supérieure ou égale à celles du sous-graphe gauche et inférieur ou égale à celles du sous-graphe droit.

Propriété d'un arbre binaire de recherche

Le parcours infixé d'un arbre binaire de recherche visite les nœuds de l'arbre par ordre croissant de leurs clés respectives.

Recherche d'un élément x dans un arbre binaire de recherche

La recherche d'un élément x dans un arbre binaire de recherche consiste à procéder à un parcours infixé de l'arbre et s'arrêter lorsque l'élément recherché est rencontré auquel cas il faut transmettre la position de l'élément dans l'arbre. La deuxième condition d'arrêt s'effectue lorsque l'on rencontre un élément avec une plus grande clé menant à une situation d'échec. L'algorithme est le suivant :

Algorithme 3.7. Recherche d'un élément dans un arbre binaire de recherche

Programme principal

var

racine : 0..n ; a : entier ;

début

recherche (racine, a) ;

fin

procédure recherche ;

entrée : arbre : tableau [1..n] d'éléments ; nœud : 0..n ; x : entier ;

sortie : la position de x dans l'arbre si ce dernier existe sinon un message pour signaler la non existence de x;

début

si ($nœud \neq 0$)

alors si ($x = arbre[nœud].clé$)

alors retourner ($nœud$)

sinon si ($x < arbre[nœud].clé$)

alors recherche (arbre [$nœud$].f gauche, x)

sinon recherche (arbre [$nœud$].f droit, x)

sinon afficher ('x n'existe pas dans l'arbre')

fin ;

La recherche se fait en parcourant une branche de l'arbre. Au pire, cette branche a une longueur égale à la profondeur de l'arbre. Par conséquent, la complexité de la procédure de recherche est en $O(p)$ si p est la profondeur de l'arbre. Mais comme au pire, dans le cas d'un arbre binaire complet, $p = \log_2(n + 1) - 1$ nous déduisons que la complexité de la recherche d'un élément dans un arbre binaire de recherche est $O(\log_2 n)$. Et c'est de là qu'on voit l'importance des arbres binaires de recherche car dans les arbres binaires ordinaires, cette complexité est $O(n)$. En effet dans ce cas, la recherche d'un élément consiste à parcourir tout l'arbre jusqu'à la rencontre de l'élément. Si cet élément n'existe pas dans l'arbre, tous les nœuds seront visités.

Recherche du minimum dans un arbre binaire de recherche

Le minimum des clés de l'arbre binaire de recherche se trouve au niveau de la feuille la plus à gauche. L'algorithme de détermination du minimum est le suivant :

Algorithme 3.8. Recherche du minimum dans un arbre binaire de recherche

procédure minimum ;
entrée : arbre : tableau [1..n] d'élément ; racine : 0..n ;
sortie : la clé minimale de l'arbre binaire ;

var nœud : 0..n ;
début

nœud := racine ;
si (nœud ≠ 0)
alors **début**
tant que (arbre[nœud]. F gauche) ≠ 0) **faire**
 nœud := arbre[nœud]. F gauche ;
 afficher (arbre[nœud].clé)
fin
sinon afficher('arbre vide')
fin ;

Recherche du maximum dans un arbre binaire de recherche

Le maximum des clés de l'arbre binaire de recherche se trouve au niveau de la feuille la plus à droite. L'algorithme de détermination du maximum est le suivant :

Algorithme 3.9. Recherche du maximum dans un arbre binaire de recherche

procédure maximum :

entrée : arbre : tableau [1..n] d'élément ; racine : 0..n ;

sortie : la clé maximale de l'arbre binaire ;

var nœud : 0..n ;

début

nœud := racine ;

si (nœud ≠ 0)

alors début

tant que (arbre[nœud].f droit) ≠ 0) faire

nœud := arbre[nœud].f droit ;

afficher (arbre[nœud].clé)

fin

sinon afficher('arbre vide')

fin ;

Insertion d'un élément dans un arbre binaire de recherche

L'insertion d'un élément x dans un arbre binaire de recherche consiste d'abord à rechercher la bonne position où placer x car les clés de l'arbre respectent l'ordre spécifique à l'arbre binaire de recherche. L'algorithme d'insertion dans ce cas en utilisant une structure dynamique est le suivant :

Algorithme 3.10. Insertion d'un élément dans un arbre binaire de recherche

Programme principal

var

racine : élément; a : entier ;

début

insertion (a) ;

fin

procédure insertion ;

entrée : racine : élément; x : entier ;

sortie : l'arbre binaire de recherche avec x inséré à la bonne place ;

var p : élément ; existe : booléen ;

début

nœud := racine ;

nouveau(p) ;

p[↑].clé := x ;

p[↑].fauche := nil ;

p[↑].f droit := nil ;

si (nœud ≠ nil)

alors

début existe := faux ;

tant que (nœud ≠ nil) et non existe faire

début parent := nœud ;

si (x = nœud[↑].clé) alors existe := vrai

sinon si (x < nœud[↑].clé) alors nœud := nœud[↑].f

gauche

sinon nœud := nœud[↑].f droit ;

fin ;

si (x = nœud[↑].clé)

alors afficher ('x existe déjà dans l'arbre')

sinon si x < parent[↑].clé alors parent[↑].f gauche := p

sinon parent[↑].f droit := p

fin

sinon racine := p ;

fin ;

Les procédures de recherche du minimum, recherche du maximum et insertion d'un élément dans un arbre binaire de recherche, toutes comme la recherche d'un élément parcourt une branche de l'arbre pour leur traitement respectif. Par conséquent, elles ont la même complexité que celle de la recherche d'un élément et qui est $O(\log_2 n)$.

Suppression d'un élément d'un arbre binaire de recherche

La suppression d'un élément x d'un arbre binaire de recherche consiste d'abord à rechercher la position de x dans l'arbre si ce dernier existe avant de le supprimer. La procédure de suppression par la suite n'est pas chose évidente surtout lorsque le nœud à supprimer a ses deux fils. L'algorithme est donné comme exercice.

Forêt d'arbre

Une forêt d'arbres est constituée d'un ensemble de k arbres comme montré sur la figure 3.11. Une forêt d'arbres peut modéliser par exemple le concept de cluster, classe ou grappe d'ordinateurs. La figure fait apparaître 3 clusters, qui sont : $\{v_1, v_2, v_3\}$, $\{v_4, v_5, v_6, v_7\}$ et $\{v_8, v_9, v_{10}\}$.

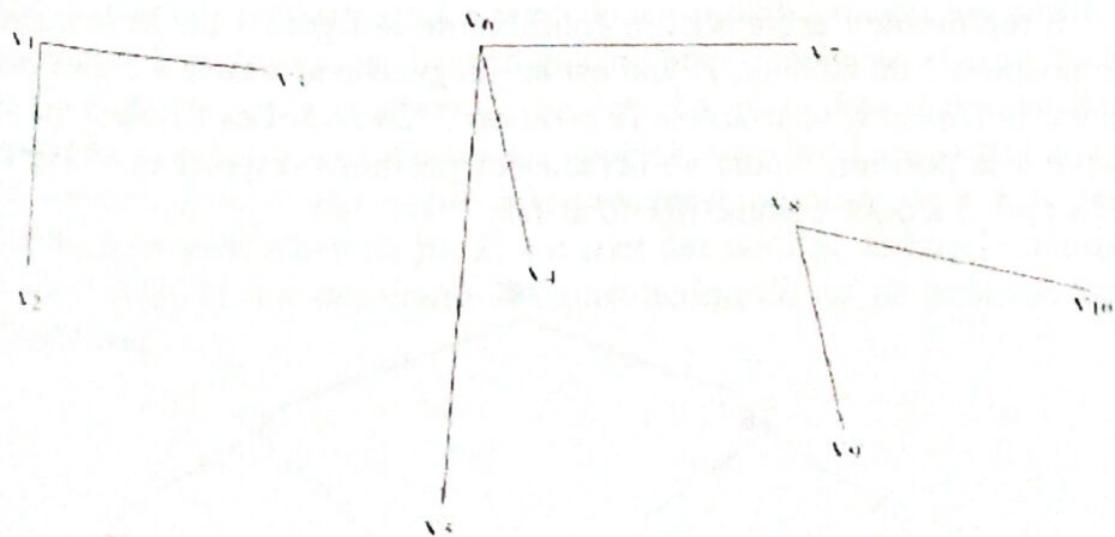


Figure 3.11. Une forêt de 3 arbres.

Structure de tas

La structure de tas est représentée à l'aide d'un tableau qui peut être converti en un arbre binaire équilibré.

Représentation d'un arbre binaire équilibré à l'aide d'un vecteur.

Un arbre binaire équilibré peut être représenté à l'aide d'un vecteur (tableau à une dimension) A de n éléments en supposant les hypothèses suivantes :

- les fils gauche et droit de l'élément $A[i]$ se trouvent respectivement au niveau des positions $2*i$ et $2*i+1$, s'ils existent pour $i=1..n/2$.
- $A[i]$ est par conséquent une feuille si $2*i > n$.

Exemple

Considérer le vecteur $A[1..10]$ suivant :

A

28	16	5	17	2	36	18	22	7	31
1	2	3	4	5	6	7	8	9	10

Il représente l'arbre binaire équilibré de la figure 3.12. 16 se trouve à la position 2 du tableau, 17 qui est le fils gauche se trouve à $2*2=4$ et 2 qui est le fils droit se trouve à la position $2*2+1=5$. Les fils de 5 qui se trouve à la position 3 sont au niveau des positions respectives $3*2=6$ et $3*2+1=7$. 5 a donc comme fils 36 et 18.

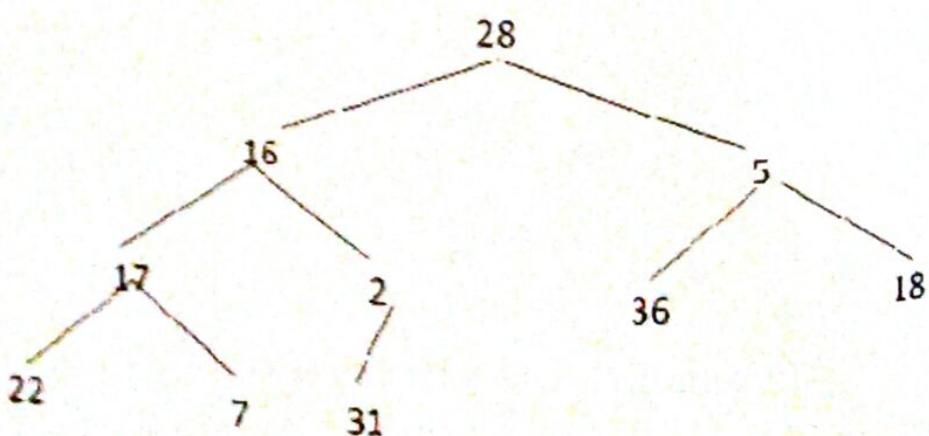


Figure 3.12. Arbre binaire représentant le vecteur A.

Définition 3.2.

Un tas est un arbre binaire dans lequel chaque clé associée à un nœud interne est supérieure aux clés associées respectivement à ses fils s'ils existent.

Caractéristiques d'un tas

En conséquence directe de la définition d'un tas, on peut déduire les deux propriétés suivantes :

- l'élément qui se trouve au niveau de la racine de l'arbre possède la plus grande clé de tous les éléments du tableau.
- Les clés se trouvant sur une chaîne allant de la racine jusqu'à une feuille sont rangées par ordre décroissant.

Exemple : L'arbre de la figure 3.10 n'est pas un tas car le nœud possédant la clé 16 est plus petit que la clé associée à son fils gauche. Pour le transformer en un tas, il faut lui appliquer la procédure construire-tas donnée ci-dessous.

Construction d'un tas

La construction d'un tas s'effectue de manière progressive de bas en haut (bottom-up technique). La procédure consiste à ranger les nœuds en parcourant le tableau A de droite à gauche pour construire un sous tas qui se développera jusqu'à atteindre un tas. La procédure qui consiste à ranger les nœuds d'un sous arbre est appelée *entasser*. La construction du tas consiste donc à faire appel à la procédure *entasser* de n à 1. Mais comme les nœuds allant de $[n/2]$ à n sont des feuilles, la boucle démarre à partir de $[n/2]$. La procédure de construction du tas se présente donc comme suit :

Algorithme 3. 11. construction d'un tas.

procédure construire-tas ;
entrée : le vecteur A[1..n] ;
sortie : structure de tas A[1..n] ;

```
var i : 1..n ;
début pour i := [n/2 ] à 1 par pas=-1 faire (* prendre la partie entière de n/2 *)
    entasser (i, n) ;

fin ;
procédure entasser ;
entrée : A[1..n] ; i,j : 1..n ;
(* i et j sont respectivement le début et la fin du sous arbre à entasser *)
sortie : structure de tas A[1..n] ;

var      temp : entier ;
imax : 1..n ;
début  imax = 0 ;
        si (2*i) <= j (* A[i] n'est pas une feuille *) alors
            si (2*i+1) <= j alors
                si (A[2*i] > A[i] ou A[2*i+1] > A[i]) alors
                    si A[2*i] > A[2*i+1] alors imax = 2*i sinon imax = 2*i+1;
                sinon si A[i] < A[2*i] alors imax = 2*i ;
                si imax ≠ 0 alors
                    début temp = A[i];
                        A[i] = A[imax];
                        A[imax] = temp;
                        entasser (imax, j);
                fin
fin ;
```

la procédure *entasser* a en entrée un sous arbre qu'elle doit transformer en un tas. Elle procède par arranger le premier nœud par rapport à ses fils. Une fois ce trio bien rangé, elle répète le même processus pour le reste du sous arbre à partir de la position qui a été modifiée (*imax*). Par conséquent le nombre d'itérations qui seront effectuées sera égal à la profondeur du sous arbre $[i..j]$. La complexité de la procédure *entasser* est donc $O(\log_2 n)$, où $\log_2 n$ exprime la plus grande profondeur. Ceci nous conduit à déduire la complexité de l'algorithme de construction du tas et qui est $O(n \log_2 n)$ car l'algorithme fait appel à la procédure *entasser* un nombre de fois égal à $[n/2]$.

L'exécution de l'algorithme sur l'arbre binaire de la figure 3.12 passe par les étapes suivantes :

A

1	2	3	4	5	6	7	8	9	10
28	16	5	17	2	36	18	22	7	31
28	16	5	17	31	36	18	22	7	2
28	16	5	22	31	36	18	17	7	2
28	16	36	22	31	5	18	17	7	2
28	31	36	22	16	5	18	17	7	2
36	31	28	22	16	5	18	17	7	2
36	31	28	22	16	5	18	17	7	2

L'arbre représentant le tas est montré sur la figure 3.13.

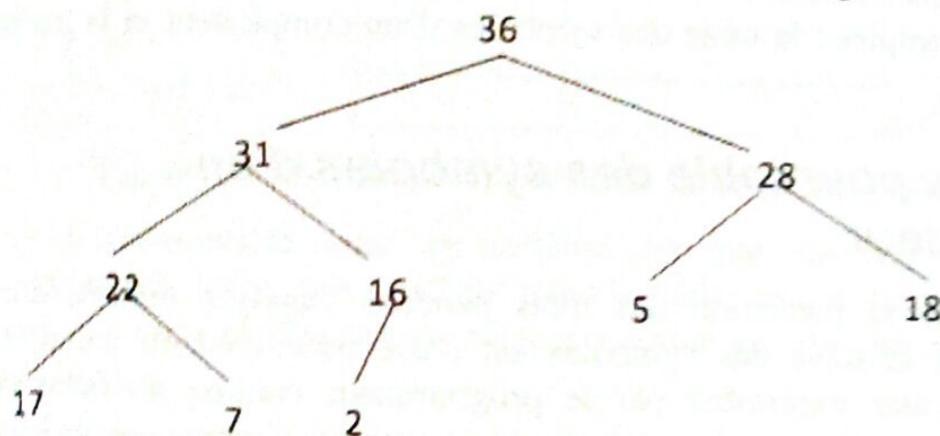


Figure 3.13. Le tas obtenu de l'exécution de l'algorithme 3.11.

Tables de Hachage

Le hachage est une technique très importante pour la gestion de l'espace mémoire et plus particulièrement pour un accès rapide à une donnée d'une grande masse stockée en mémoire. De nos jours, l'avènement de l'Internet a accentué la croissance des volumes de données qui se mesurent avec des milliers voire des millions de téraoctets. La technique du hachage est devenue un outil crucial aidant à gérer le stockage et l'accès à une donnée dans une masse faramineuse de données.

Un exemple concret d'utilisation du mécanisme de hachage est le compilateur de langages informatiques. En effet comme le nombre de

mots clés et des variables qu'elles soient engendrées par le programmeur ou le compilateur lui-même est énorme, une table de hachage contenant tous ces symboles est développée le long du processus de compilation. L'accès à un terme de la table doit se faire très rapidement car le temps de réponse du compilateur en dépend.

Une autre application du hachage s'effectue en cryptographie pour accéder par exemple aux empreintes numériques. En l'occurrence, en cryptologie, le hachage permet de retrouver rapidement une empreinte digitale stockée parmi des millions d'autres dans une grande base de données.

Dans le but de mieux illustrer la technique de hachage, nous expliquons en premier lieu la nécessité du hachage dans une grande base de données puis montrons comment développer une fonction de hachage sur deux exemples : la table des symboles d'un compilateur et le jeu du taquin.

Hachage pour table des symboles d'un compilateur

Lors de la formation des mots pendant l'analyse lexicale d'un compilateur, la table des symboles est créée pour contenir toutes les entités lexicales exprimées par le programmeur comme les noms de variable, les nombres et les chaînes de caractères. Comme ces lexèmes sont utilisés dans toutes les phases ultérieures de la compilation comme l'analyse syntaxique, la traduction du programme en code intermédiaire, l'optimisation et la génération du code objet, il est nécessaire d'accéder à ces lexèmes de manière presqu'instantanée pour que le compilateur puisse répondre très rapidement à la commande du programmeur. La technique de hachage est alors développée pour satisfaire cette contrainte de temps. Le mécanisme de hachage sans index est schématisé sur la figure 3.14.

A l'aide de la clé qui est calculée à partir de la donnée, on accède à la table. On compare la donnée avec celle présentée au niveau de la première colonne. Si elles sont égales alors on retourne sa position dans la table. Dans le cas contraire, on parle de collisions car deux données ont la même clé. La collision peut exister entre plusieurs données. On considère dans ce cas, la deuxième colonne qui lie toutes les données qui

ont la même clé pour rechercher la position de la donnée dans la table. Le chainage présente ainsi une solution au problème de collision. Si la donnée n'existe pas encore dans la table, elle est alors insérée à une place libre et reliée à la liste des données qui ont la même clé par l'intermédiaire de la deuxième colonne.

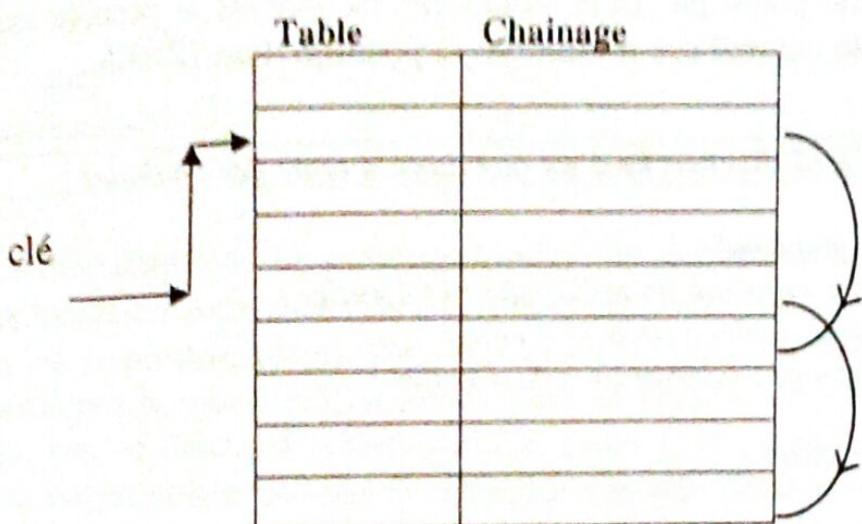


Figure 3.14. Schéma d'une table de hachage sans index.

L'inconvénient avec ce schéma est que les données peuvent apparaître de façon non contiguë dans la table, ce qui peut induire de la perte d'espace et d'autres problèmes comme un nombre très grand de collisions.

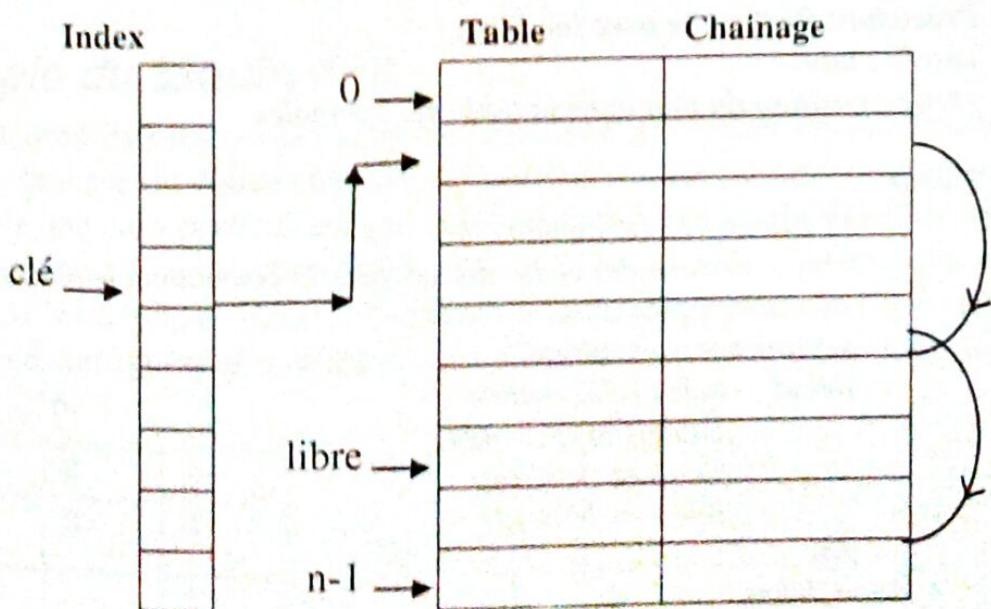


Figure 3.15. Schéma d'une table de hachage avec index.

Pour pallier à cet inconvénient, la figure 3.15 propose un schéma de hachage avec index, n étant la taille de la table. Une fois la clé calculée, on accède à l'index qui nous fournit la position de la première donnée qu'à cette clé. L'algorithme 3.12 décrit la recherche d'un mot dans la table des symboles telle effectuée par l'analyse lexicale d'un compilateur. L'algorithme passe par trois situations. Le cas où la donnée existe déjà dans la table, auquel cas il retourne sa position dans la table.

Algorithme 3.12. Recherche d'un mot dans la table des symboles

programme principal

*type index : tableau [0..taille-index-1] d'entier ;
Table : tableau [0..n-1] d'entier ;
chainage : tableau [0..n-1] d'entier ;*

var i, libre : entier ;

début

libre := 0 ;

pour i := 0 à n-1 faire chainage[i] := -1 ;

pour i := 0 à taille-index-1 faire index[i] := -1 ;

recherche ('max') ;

fin

Procédure Recherche avec hachage.

entrée : mot

sortie : position du mot dans la table des symboles

Début

(calculer la clé du mot :*)*

clé := (somme des codes des caractères constituant le mot) modulo a ;

courant := index[clé] ;

si (courant = -1) alors

début Index[clé] := libre ;

Table[libre] := mot ;

Courant := libre ;

Libre := libre + 1.

fin

sinon début

*tant que (Table[courant] ≠ mot) et (chainage[courant] ≠ -1)
faire courant := chainage[courant] ;*

```

    si (table[courant] ≠ mot) alors
        début
            Table[libre] := mot ;
            Chainage[libre] := -1 ;
            Chainage[courant] := libre ;
            Libre := libre + 1 ;
        fin ;
    fin ;
    retourner(courant)
fin

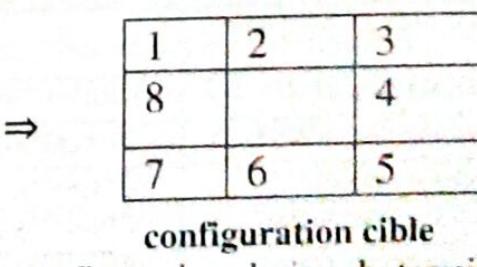
```

Le deuxième cas se présente lorsque le mot n'existe pas dans la table ni même un autre mot ayant la même clé ($\text{Index}[\text{clé}] = -1$). Enfin, le dernier cas se produit lorsque des mots ayant la même clé existent dans la table mais pas le mot que l'on est en train de rechercher. Dans ces deux derniers cas, le mot est inséré dans la table à la position libre et sa position est renvoyée comme résultat. La variable *libre* est incrémentée pour l'insertion du prochain mot qui ne se trouve pas encore dans la table.

Un grand nombre de collisions a des répercussions négatives sur le temps de recherche d'une donnée puisque, au pire des cas la recherche d'une donnée nécessitera l accès à la table, l étant la longueur maximale du chainage. Il est donc important de bien définir la clé pour réduire au maximum la longueur des chainages et par conséquent le nombre de collisions.

Exemple du taquin 3x3

Le taquin 3x3 est un jeu solitaire en forme de damier de dimension 3x3. Chaque case du damier occupe un nombre unique compris entre 1 et 8. Il existe une case particulière qui est vide et qui permet de déplacer les cases adjacentes. Les nombres apparaissent dans un ordre quelconque et il s'agit de les déplacer jusqu'à l'obtention d'un rangement de 1 à 8. Un exemple de configuration initiale et de configuration cible est montrée sur la figure 3.16.



2	8	3
1	6	4
7		5

⇒

1	2	3
8		4
7	6	5

configuration initiale

configuration cible

Figure 3.16. Exemple d'une configuration du jeu du taquin 3x3.

La résolution du jeu consiste à tester des configurations obtenues à partir de la configuration initiale si elles coïncident avec la configuration cible. Mais comme le nombre de configurations possibles est prohibitif, la résolution s'avère très compliquée. Nous avons besoin au cours du processus de résolution de stocker les configurations pour pouvoir y accéder. Dans ce qui suit, nous allons calculer le nombre de configurations qui existent au total puis dans une seconde phase, le processus de hachage qui permet de réduire le temps d'accès de manière impressionnante.

Nombre de configurations du jeu du taquin 3x3

Le nombre de configurations du jeu du taquin 3x3 est égal au nombre de combinaisons (3x3). Il est égal alors à $9! = 362880$. Ce nombre étant faramineux, le hachage est une solution pour accéder rapidement à une configuration. Bien entendu lorsque la dimension du damier augmente, le nombre de ces configurations explose.

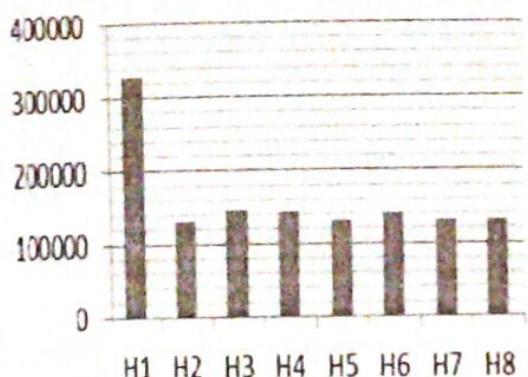
Conception d'une fonction de hachage pour le taquin 3x3

Le tableau ci-dessous montre 8 fonctions de hachage H_1 à H_8 que nous avons étudiées pour stocker de la meilleure manière possible les configurations du jeu du taquin 3x3. Il exhibe également le nombre total de collisions pour chaque fonction ainsi que le nombre de collisions maximal que peut avoir une entrée de la table de hachage. La clé est égale pour toutes ces fonctions au nombre $x_0x_1x_2\dots x_8$ où x_i est la valeur de la case i de la configuration du taquin à stocker ou à rechercher dans la table de hachage.

	Fonction de hachage	s	#collisions	#Max-collisions par entrée
$H_1(\text{clé})$	clé mod 9!		330194	28
$H_2(\text{clé})$	clé mod 362897		132052	8
$H_3(\text{clé})$	(clé mod 362897) mod 9!		146569	10
$H_4(\text{clé})$	(clé+s) mod 9!	$0*x_0+1*x_1+2*x_2+\dots+8*x_8$	145626	8
$H_5(\text{clé})$	(clé+s) mod 362897	$0*x_0+1*x_1+2*x_2+\dots+8*x_8$	133018	8
$H_6(\text{clé})$	(clé+s) mod 9!	$0^6*x_0+1^6*x_1+2^6*x_2+\dots+8^6*x_8$	142531	8
$H_7(\text{clé})$	(clé+s) mod 362897	$0^6*x_0+1^6*x_1+2^6*x_2+\dots+8^6*x_8$	131480	8
$H_8(\text{clé})$	(clé-s) mod 362897	$0^9*x_0+1^9*x_1+2^9*x_2+\dots+8^9*x_8$	131480	8

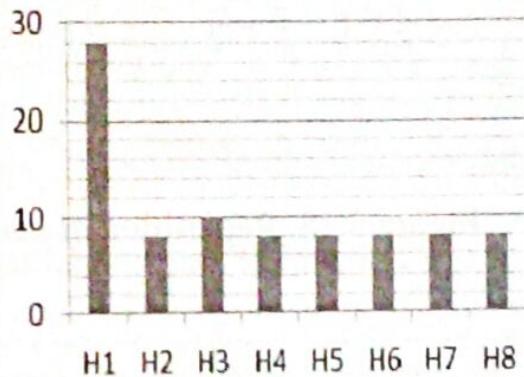
Faisons remarquer que 362897 est le premier nombre premier supérieur à $9!$. Pour éviter le problème des diviseurs communs et ainsi réduire *considérablement* le problème de collision, nous augmentons la taille de notre table de $9!$ à 362897, étant donné la différence négligeable qui existe entre ces deux nombres ($362897 - 9! = 17$). Nous représentons les résultats précédents sous forme graphique pour une meilleure lecture :

Total collisions



(a)

Collision MAX



(b)

Figure 3.17. Nombre de collisions pour chaque fonction de hachage.

La figure 3.17 montre le nombre de collisions pour chacune des fonctions $H_1..H_8$. Le graphe (a) exhibe le nombre total des collisions pour toutes les cases de la table de hachage alors que celui de (b) présente le nombre maximum de collisions par case. On remarque que, hormis H_1 et H_3 , le nombre maximal de collisions est égal à 8 pour toutes les autres fonctions. Pour H_3 , il est de 10, ce qui peut être comparable avec celui des autres mais celui de H_1 est de 29, ce qui est mauvais pour le temps d'accès à la configuration du taquin.

Distribution du nombre de collisions sur les clés

Nous avons ensuite examiné la distribution du nombre de collisions sur différentes valeurs de clé, pour chacune des 8 fonctions. Figures 3.18 jusqu'à 3.25 montrent les résultats obtenus. Là encore, à l'exception de H_1 , la variation des clés engendre à peu près le même nombre de collisions.

Exercices

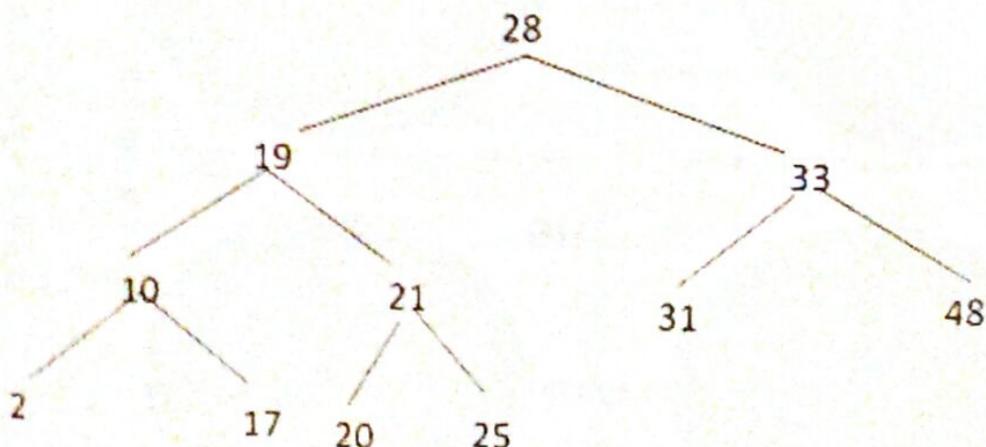
Exercice 3.1

Considérer un arbre binaire.

- 1) Écrire les versions itératives pour :
 - a. le parcours préfixé
 - b. le parcours infixé
 - c. le parcours post-fixé
- 2) Écrire une procédure pour calculer sa hauteur.
- 3) Écrire une procédure pour compter le nombre de ses feuilles.

Exercice 3.2

Considérer l'arbre binaire de recherche suivant :



- 1) insérer le nombre 43 dans l'arbre.
- 2) Ensuite supprimer le nombre 19 de l'arbre.
- 3) Écrire un algorithme de suppression d'un élément d'un arbre binaire de recherche.
- 4) Calculer la complexité de l'algorithme.

Exercice 3.3

Écrire un algorithme pour évaluer les expressions arithmétiques sur les opérateurs + et *, exprimées en notation :

- a. Pré fixée
- b. infixé
- c. post-fixé

Exercice 3.4

Soit $T = (S, A)$ un arbre binaire. Écrire un algorithme pour associer des entiers aux sommets de T en respectant la condition suivante :

- Si (v, w) est une arête et que la profondeur de v est plus petite que celle de w , alors l'entier associé à v est plus petit que celui associé à w .

Calculer la complexité de l'algorithme.

Exercice 3.5

Considérer l'arbre abstrait de la figure 3.9.

- 1) Écrire un algorithme pour évaluer une expression arithmétique représentée par un arbre abstrait.
- 2) Calculer la complexité de l'algorithme
- 3) Appliquer votre algorithme sur l'exemple de la figure 3.9.

Exercice 3.6

- 1) Écrire un algorithme pour compter le nombre de nœuds d'un arbre binaire. Calculer sa complexité.
- 2) Écrire un algorithme pour compter le nombre de nœuds d'un arbre binaire de recherche. Calculer sa complexité.
- 3) Que peut-on conclure ? Justifier votre réponse.

Exercice 3.7

Considérer l'arbre binaire de la figure 3.10.

- 1) Écrire un algorithme pour tester si un arbre binaire est un arbre binaire de recherche.
- 2) Transformer l'arbre de la figure 3.10 en un arbre binaire de recherche.
- 3) Écrire un algorithme pour transformer un arbre binaire quelconque en un arbre binaire de recherche.

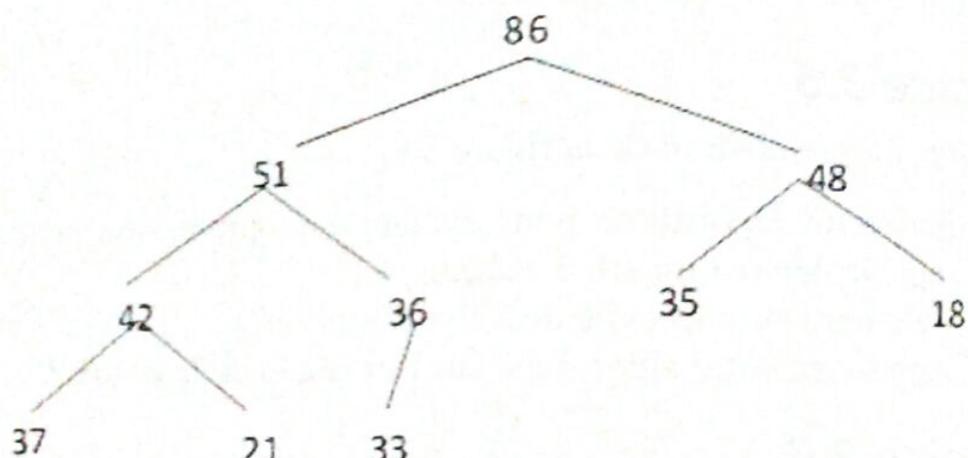
Exercice 3.8

- 1) Écrire un algorithme qui imprime les clés d'un arbre binaire d'entiers comprises entre deux entiers x et y . Calculer sa complexité.

- 2) Écrire un algorithme qui retourne un sous arbre d'un arbre binaire de recherche dont les clés sont comprises entre deux entiers x et y . Calculer sa complexité.

Exercice 3.9

Considérer l'arbre binaire suivant :



- 1) Quelles sont les propriétés de cette structure ?
- 2) Supprimer '51' de la structure tout en préservant ses propriétés.
- 3) Écrire un algorithme de suppression d'un élément de la structure tout en préservant ses propriétés.

Exercice 3.10

- 1) Construire un tas contenant les clés suivantes :
11, 73, 29, 45, 6, 31, 52, 89, 93, 9
- 2) Écrire un algorithme pour rechercher un élément dans un tas. Calculer sa complexité.
- 3) Écrire un algorithme pour insérer un élément dans un tas. Calculer sa complexité. Illustrer votre algorithme en insérant l'entier 90 dans le tas construit à la première question.

Exercice 3.11 (hachage)

Considérer la liste des mots suivants :

- Arbre
- Sommet
- Graphe

- Lecture
- Écriture
- Maximum
- Minimum

Ainsi que l'ordre alphabétique des caractères. L'ordre de 'a' est égal à 1, celui de 'b' est 2 et ainsi de suite.

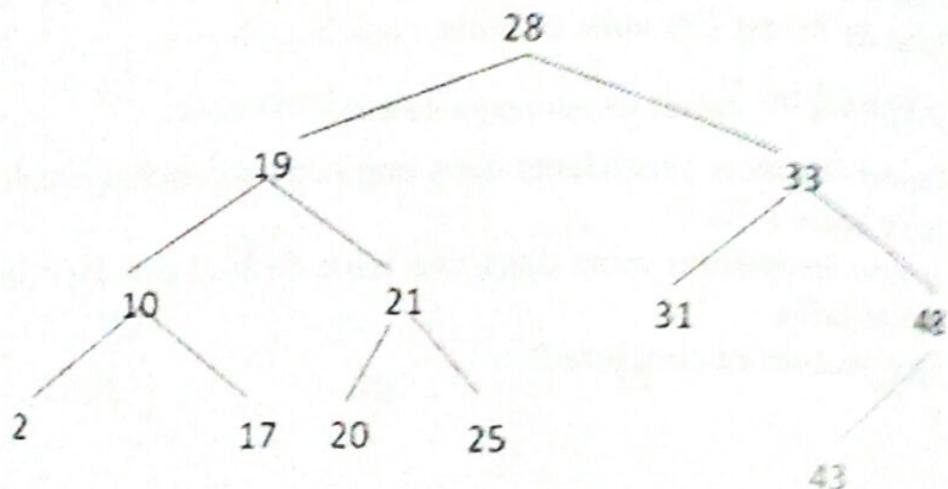
Soit $clé[mot] := \sum_{c \text{ est un caractère du mot}} ordre(c)$

- 1) Insérer les mots précédents dans une table de hachage sans index de taille égale à 20.
- 2) Insérer les mêmes mots dans une table de hachage avec index de la même taille.
- 3) Que peut-on en conclure ?

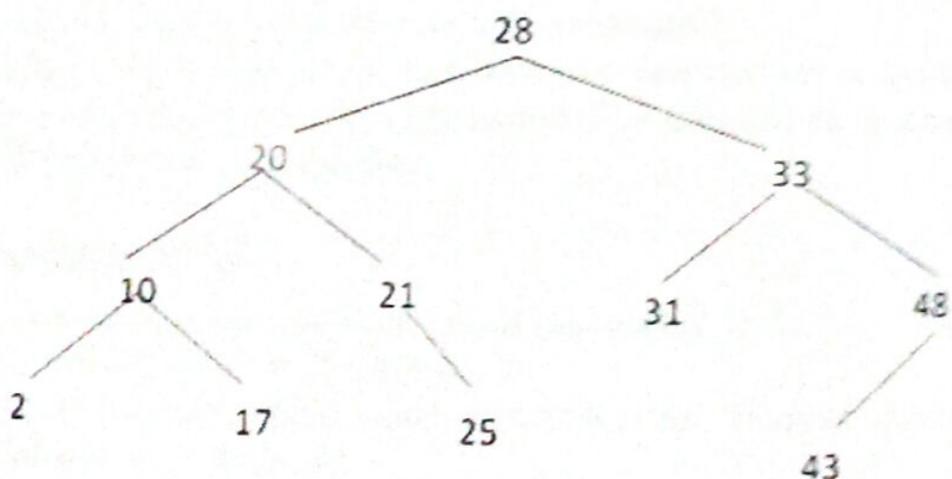
Corrigé des exercices

Exercice 3.2

- 1) insérer le nombre 43 dans l'arbre.



- 2) Ensuite supprimer le nombre 19 de l'arbre.



- 3) Écrire un algorithme de suppression d'un élément d'un arbre binaire de recherche.

L'algorithme passe par 4 étapes qui sont :

1. Recherche de la valeur x dans l'arbre.
2. Recherche de la valeur minimale du sous-arbre droit du nœud contenant x ou bien de la valeur maximale du sous-arbre gauche du nœud contenant x . Soit y , cette valeur.
3. Suppression du nœud contenant y .
4. Remplacer x par y .

Algorithme Suppression d'un élément d'un arbre binaire de recherche

Programme principal

var racine : ↑élément ; a : entier ;
début suppression (a) fin

procédure suppression ;

entrée : racine : ↑élément ; x : entier ;

sortie : l'arbre binaire de recherche sans le nœud contenant x ;

var nœud, p, prédécesseur, parent : ↑élément ; existe : booléen ;

début nœud := racine ;

existe := faux ;

tant que (nœud ≠ nil) et non existe faire

(recherche de la clé x dans l'arbre *)*

si (x = nœud↑.clé) alors existe := vrai

sinon début prédécesseur := nœud ;

si (x < nœud↑.clé) alors nœud := nœud↑.f gauche

sinon nœud := nœud↑.f droit ;

fin ;

si (nœud = nil) alors signaler ('x n'existe pas dans l'arbre')

sinon si (x = nœud↑.clé) alors

début

p := nœud↑.f droit ;

(recherche de la clé minimale du sous-arbre droit *)*

si (p ≠ nil) alors tant que (p↑.f gauche ≠ nil) faire

début parent := p ;

p := p↑.f gauche ;

fin ;

(recherche de la clé maximale du sous-arbre gauche *)*

sinon début p := nœud↑.f gauche ;

si (p ≠ nil) alors tant que (p↑.f droit ≠ nil) faire

début parent := p ;

p := p↑.f droit ;

fin ;

fin

sinon (le nœud contenant x n'a pas de successeur alors*

*suppression de ce nœud *)*

si (nœud = racine) alors racine := nil

```

sinon début prédecesseur↑.f droit := nil ;
    prédecesseur↑.f droit := nil ;
    fin ;
si (nœud↑.f gauche ≠ nil) ou (nœud↑.f droit ≠ nil) alors
début (* remplacer x par y *)
    x := p↑.clé ;
    nœud↑.clé := x ;
    (* supprimer le nœud contenant y *)
    si (p↑.f gauche = nil) alors parent↑.f gauche := nil
        sinon parent↑.f droit := nil ;
    fin
fin
fin ;

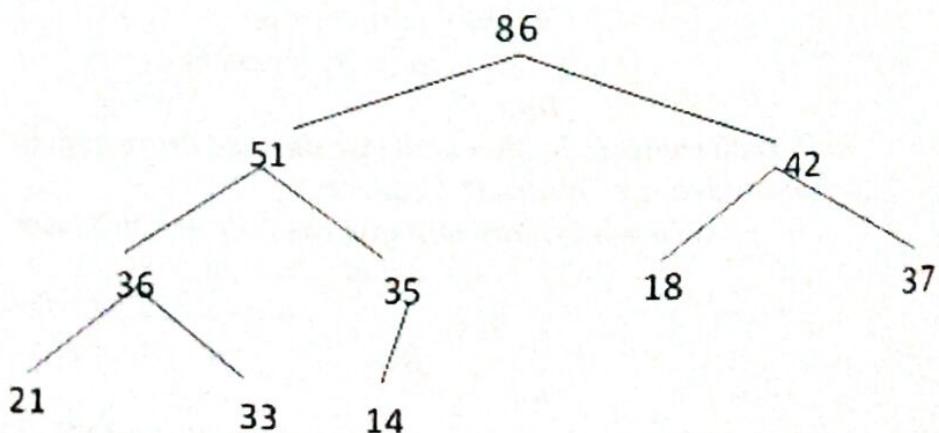
```

4) Calculer la complexité de l'algorithme.

La recherche de l'élément x à supprimer se fait en $O(\log_2(n))=p$ où p est la profondeur de l'arbre et n le nombre de nœuds. L'opération de recherche de l'élément du sous-arbre droit qui a la plus petite valeur pour remplacer x s'effectue aussi en $O(\log_2(n))$, de même pour la recherche de la clé maximale du sous-arbre gauche. La complexité est donc $O(\log_2(n))$.

Exercice 3.9

Considérer l'arbre binaire suivant :



1) Quelles sont les propriétés de cette structure ?

Comme les clés associées aux nœuds internes de l'arbre sont supérieures à celles associées aux fils, la structure est un tas.

2) Supprimer '51' de la structure tout en préservant ses propriétés.

Pour supprimer '51' de l'arbre, nous procérons comme suit :

- Déterminer la structure de tableau équivalente
- Supprimer '51' du tableau
- Construire le tas à partir du tableau obtenu

Ce qui donne :

- a) tableau

86	51	42	36	35	18	37	21	33	14
----	----	----	----	----	----	----	----	----	----

- b) suppression de '51' du tableau

86	42	36	35	18	37	21	33	14
----	----	----	----	----	----	----	----	----

- c) construction du tas

86	42	36	35	18	37	21	33	14
86	42	37	35	18	36	21	33	14

3) Écrire un algorithme de suppression d'un élément de la structure tout en préservant ses propriétés.

Algorithme :

- obtenir la structure du tas sous forme de tableau
- Supprimer l'élément du tableau
- Construire le tas à partir du tableau obtenu en 2.

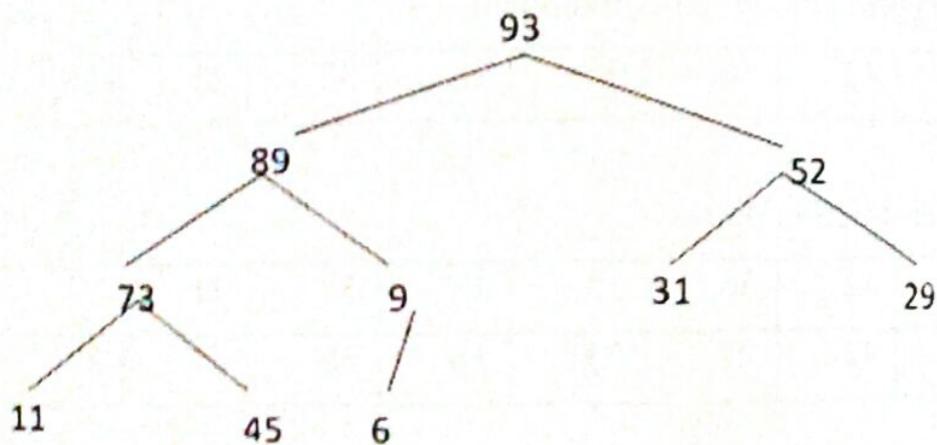
Exercice 3.10

1) Construction du tas

Les étapes des différentes itérations et plus précisément des différentes permutations de la construction du tas sont montrées ci-dessous :

11	73	29	45	6	31	52	89	93	9
				9					
			93						6
		52						45	
	93		73					73	
			89						
93	11		11						
	89								
93	89	52	73	9	31	29	11	45	6

Le tas est schématisé comme suit :



2) Recherche d'un élément dans un tas.

procédure recherche-tas ;

entrée : T : tas [1..n] ; x : entier ;

sortie : position de x dans T ;

var trouve : booléen ;

i : 1..n+1 ;

début

trouve := faux ;

i := 1 ;

tant que ($i \leq n$) et (non trouve) faire

début

si ($T[i] = x$) alors trouve := vrai ;

i := i+1 ;

fin ;

si trouve alors retourner(i-1)

sinon afficher ('x n'existe pas dans T) ;

fin ;

appel de la procédure : recherche-tas (90,1..n) ;

complexité :

Comme le vecteur n'est pas trié, au pire cas le vecteur est parcouru en entier à la recherche de l'élément. La complexité est donc O(n).

3) Insertion d'un élément dans un tas

L'idée est d'insérer l'élément à la fin du vecteur puis d'appeler la procédure construire-tas.

procédure insérer-tas (x :entier ; 1..n) ;

entrée : T : tas [1..n] ; x : entier ;

sortie : T : tas [1..n+1] ;

début

T[n+1] := x ;

Construire-tas(1..n+1) ;

fin

Complexité :

Comme toutes les branches du tas sont triées, la procédure construire-tas va toucher uniquement la branche qui contient l'élément à insérer pour l'insérer à la bonne place sur la branche correspondante. La complexité est donc $O(\log_2 n)$.

Exemple : Illustration de l'insertion de 90 dans le tas construit précédemment.

93	89	52	73	9	31	29	11	45	6	90
				90						
93	90	52	73	89	31	29	11	45	6	9

