Algorithmique et STD
L2 Informatique

A. Aïtelhadj & F. Aïtelhadj
Maitres de conférences
Département Informatique
FGEI, UMMTO

Chapitre 1

Récursivité

1. Caractéristiques principales

- Une procédure (ou fonction) récursive comporte au moins un appel à elle-même, alors qu'une procédure (ou fonction) non récursive ne peut comporter que des appels à d'autres procédures (ou fonctions).
- La récursivité permet d'écrire des programmes (algorithmes) plus facilement, et de manière élégante.
- Les programmes (algorithmes) itératifs sont souvent plus efficaces que les programmes (algorithmes) récursifs.
- Une procédure récursive n'est jamais plus rapide que son homologue itérative.

2. Découpage de la récursivité

Pour décrire un algorithme récursif, il convient d'analyser le problème afin :

- d'identifier le cas particulier ;
- d'identifier le cas général qui permet d'effectuer la récursion.

Schéma général

```
Algorithme Recur (Paramètres) ;
Si (Test-arrêt) Alors
    | Instruction du point d'arrêt
Sinon
    | Instructions ;
    | Recur (Paramètres changés);
    | Instructions
Fin Si
```

Exemple

```
Fonction ADD (N : entier) : entier;
Si (N > 1) Alors
    | ADD ← N + ADD(N − 1)
Sinon
    | ADD ← 1
Fin Si
```

Pour simplifier, on peut adopter la notation du langage C dans l'écriture des algorithmes. Par exemple, la fonction ADD peut être réécrite comme suit :

```
Fonction ADD (N : entier) : entier;
Si (N > 1) Alors
    | retourner (N + ADD(N − 1))
Sinon
    | retourner (1)
Fin Si
```

3. Récursivité terminale

Un algorithme est récursif terminal s'il ne contient aucun traitement après un appel récursif.

Schéma Général	Légende
Algo P(u)	u est la liste des paramètres
Si C(u)	C(u) est la condition sur les paramètres u
Alors	
B(u);	B(u) est le traitement de base de Algo P(u)
P(a(u))	P(a(u)) est l'appel récursif avec des paramètres u transformés
Sinon	
T(u)	T(u) est le traitement de terminaison (arrêt de l'algorithme)
Finsi	

Remarque

Il est toujours possible de dérécursifier un algorithme récursif.

- S'il est récursif non-terminal, on utilise les piles pour sauvegarder le contexte.

- S'il est récursif terminal, il est inutile d'utiliser les piles ; l'algorithme se transforme alors comme suit :

Schéma général de l'algorithme P transformé en l'algorithme P' ci-après

```
Algo P'(u);
Tant que (C(u)) faire
    | B(u);
    | u ← a(u)
Fait
T(u)
```

4. Exemples de fonctions récursives non-terminales

- La fonction ADD présentée précédemment en 2 est récursive non-terminale.

- La fonction suivante qui fait l'addition de deux entiers est également récursive non-terminale.

```
Fonction PLUS (a, b : entier) : entier;
Si (b = 0) Alors
   | retourner (a)
Sinon
  | retourner (1 + PLUS (a, b − 1))
Fin Si
```

L'exécution de la fonction PLUS(4,2) est montrée dans le tableau suivant :

Appel	Exécution	Valeur retournée
PLUS(4,2)	1 + PLUS(4,1)	-
PLUS(4,1)	1 + PLUS(4,0)	-
PLUS(4,0)	PLUS ← 4	4 est le résultat au 1^{er} retour 1 + 4 : 5 est le résultat au 2^{eme} retour 1 + 5 : 6 est le résultat au dernier retour correspondant à l'appel PLUS(4,2)

- Enfin, la fonction Somme_cubic suivante qui calcule la somme cubique des N premiers entiers est également une fonction récursive non-terminale.

```
Fonction Somme_cubic( N : entier) : entier;
Si (N = 0) Alors
   | retourner (0)
Sinon
   | retourner (Somme_cubic (N − 1)+ N^3)
Fin Si
```

L'exécution de la fonction Somme_cubic (3) est répertoriée dans le tableau ci-après :

Appel	Exécution	Valeur retournée
Somme_cubic (3)	Somme_cubic (2) + 3^3	-
Somme_cubic (2)	Somme_cubic (1) + 2^3	-
Somme_cubic (1)	Somme_cubic (0) + 1^3	-
Somme_cubic (0)	Somme_cubic ← 0	0 est le résultat au 1^{er} retour
		0 + 1^3 : 1 est le résultat au 2^{eme} retour
		1 + 2^3 : 9 est le résultat au 3^{eme} retour
		9 + 3^3 : 36 est le résultat au dernier retour
		qui est la valeur correspondant
		à l'appel Somme_cubic (3)

5. Exemples de fonctions récursives terminales

- La fonction qui calcule le PGCD de manière récursive est une fonction récursive terminale. Un pseudo code de cette fonction est le suivant :

```
Fonction PGCD (A, B : entier) : entier ;
Si (A ≠ B) Alors
   | Si (A > B) Alors
   |    | retourner PGCD (A − B, B)
   | Sinon
   |    | reourner PGCD (A, B − A)
   | Fin Si
Sinon
   | retourner (A)
Fin Si
```

Cette fonction est récursive terminale; le pseudo code de la fonction itérative équivalente est donné par la fonction PGCD1 suivante :

```
Fonction PGCD1 (A, B : entier) : entier ;
Tant que (A ≠ B) faire
    | Si (A > B) Alors
    |     | A ← A − B
    | Sinon
    |     | B ← B − A
    | Fin Si
Fait
PGCD1 ← A
```

- Un autre exemple de fonction récursive terminale est la fonction récursive booléenne (prédicat) qui indique si un nombre entier naturel est pair ou non (impair). Elle consiste en le pseudo code suivant :

```
Fonction Parité (N : entier) : booléen ;
Si (N = 0) Alors
  | retourner (vrai)
Sinon
  | Si (N = 1) Alors
  |     | retourner (faux)
  | Sinon
  |     | retourner Parité(N − 2)
  | Fin Si
Fin Si
```

Ce code est équivalent au code suivant :

```
Fonction Parité (N : entier) : booléen ;
Si (N > 1) Alors
  | retourner Parité(N − 2)
Sinon
  | Si (N = 0) Alors
  |     | retourner (vrai)
  | Sinon
  |     | retourner (faux)
  | Fin Si
Fin Si
```

ce dernier, produit le code itératif suivant :

```
Fonction Parité (N : entier) : booléen ;
Tant que (N > 1) faire
    | N ← N − 2
Fait
Si (N = 0) Alors
    | Parité ← vrai
Sinon
    | Parité ← faux
Fin Si
```

Chapitre 2

Complexités algorithmiques

1. Notion de complexité algorithmique

La théorie de la complexité repose sur la définition de classes de complexité qui permettent de classer les problèmes en fonction de la complexité des algorithmes qui existent pour les résoudre.

Juger de l'efficacité d'un algorithme requiert deux facteurs prépondérants : la complexité temporelle et la complexité spatiale.

La complexité temporelle équivaut au temps nécessaire pour l'exécution de l'algorithme.

La complexité spatiale, dite aussi encombrement mémoire, représente la place (espace mémoire) occupée par toutes les structures de données manipulées par l'algorithme.

On s'intéresse ici, plus particulièrement, à la complexité temporelle.

2. Principe d'évaluation

On associe à chaque instruction de l'algorithme une valeur positive ou nulle représentant le nombre d'exécutions. On distingue :

- Le nombre d'exécutions de chaque primitive non définie dans l'algorithme proprement dit :
  - Primitives de la machine séquentielle (MS)
  - Primitives de traitement de structures abstraites de données comme les piles, files, etc.
- Le nombre d'exécutions des autres instructions :
  - Contrôle comme : tantque, si, répéter, pour, etc.
  - Opérations logiques comme : et, ou, non, etc.

- Opérations arithmétiques; il est souhaitable de distinguer les opérateurs additifs (+, -) des opérateurs multiplicatifs (/, *)
- Affectation
- Accès aux tableaux; sur certains machines, ils sont très coûteux
- E/S; on range dans cette catégorie Lecture/Écriture qui sont très coûteuses
- Encombrement mémoire (complexité spatiale) qui exprime la taille des données

On distingue entre complexité théorique (CT) et complexité pratique (CP).

3. Complexité théorique (CT)

La complexité théorique vise à donner une idée sur le comportement général de l'algorithme. On parle dans ce cas de comportement asymptotique de l'algorithme.

Si N représente le nombre d'éléments à traiter le comportement peut être défini comme suit dans le tableau ci-après :

Notation	Type de complexité
$O(1)$	complexité indépendante de la taille de la donnée
$O(\log(N))$	complexité logarithmique
$O(N)$	complexité linéaire ou séquentielle
$O(N \cdot \log(N))$	complexité quasi-quadratique ou linéarithmique
$O(N^2)$	complexité quadratique
$O(N^3)$	complexité cubique
$O(N^p)$	complexité polynomiale
$O(N^{\log(N)})$	complexité quasi-polynomiale
$O(2^N)$	complexité exponentielle
$O(N!)$	complexité factorielle

Dans ce cas, N doit être suffisamment grand !! pour marquer les différences entre algorithmes de classes différentes. En effet, par exemple si $N \longrightarrow +\infty$ alors $\frac{\log(N)}{N} \longrightarrow 0$. La classe à comportement linéaire l'emporte sur la classe à comportement logarithmique lorsque N tend vers l'infini.

Remarque $O(f(N))$ est la notation universelle adoptée pour la complexité théorique (CT). La fonction f peut être linéaire, quadratique, polynomiale, exponentielle, etc., comme celles du tableau ci-dessus.
Il existe d'autres notations comme Ω, o et Θ, mais auxquelles on ne fait presque pas appel dans le cadre de ce cours. La signification exacte de ces symboles est la suivante :

- $f = O(g) \Leftrightarrow f(N) \leq c \cdot g(N), c > 0$
- $f = \Omega(g) \Leftrightarrow f(N) \geq c \cdot g(N), c > 0$
- $f = o(g) \Leftrightarrow \lim\limits_{N \to +\infty} \dfrac{f(N)}{g(N)} = 0$

- $f = \Theta(g) \Leftrightarrow f = O(g)$ et $g = O(f)$

4. Complexité pratique (CP)

Contrairement à la complexité théorique, la complexité pratique vise à une analyse plus fine de nature à pouvoir mettre en valeur les différences entre algorithmes appartenant à une même classe d'algorithmes. Son évaluation est relativement inexacte du fait qu'on ne dispose pas toujours de chiffres exacts sur les coûts de chaque opération. Il s'agit donc tout juste d'une estimation grossière.

Remarque Dans chaque type ou classe de complexité, on peut parler de comportement moyen, de comportement au mieux, ainsi que de comportement au pire.

Par exemple, la recherche dans une table par accès séquentiel (linéaire) a un comportement linéaire noté $O(N)$ comme indiqué dans le tableau précédent.
Au pire, on va parcourir toute la table; donc N accès. Au mieux, l'élément recherché est le premier dans la table; donc un (1) seul accès. En moyenne, on a $\frac{N+1}{2}$; le comportement asymptotique est donc de l'ordre $O(N)$ correspondant à une complexité théorique (CT) linéaire.

Dans ce contexte, avec une complexité CP $= \frac{N}{2}$, le comportement général asymptotique est $O(N)$, c'est-à-dire linéaire.

En outre, si CP $= N^2 + 4N + 3$, le comportement général asymptotique est $O(N^2)$. Autrement dit, on prend uniquement le monôme de plus haut degré de coefficient 1.

5. Exemple d'application

L'algorithme suivant calcule la racine carrée d'un nombre quelconque $x \neq 0$ par la méthode de Newton.

Algorithme Newton;
Constante epsilon $= 10^{-6}$;
Variable $x, racine$: réel;
(1) lire (x); *racine* $\leftarrow 1$;
(-) répéter
(2) $racine \leftarrow \frac{\frac{|x|}{racine}+racine}{2}$
(3) jusqu'à $|\frac{|x|}{square(racine)} - 1| < epsilon$;
(4) si $x > 0$
(5) alors écrire (`racine réelle :´, $racine$)
(6) sinon écrire (`racine complexe :´, `i´, $racine$)

Les détails concernant l'évaluation de la complexité algorithmique de l'algorithme ci-dessus sont donnés dans le tableau suivant :

numéro	contrôle	affectation	arithmétique	MS	E/S
(1)		1			1
(2)		N	$3N$	N	
(3)	N		$2N$	$3N$	
(4)	1				
(5)/(6)					1
total	$N+1$	$N+1$	$5N$	$4N$	2

Au total on a CP $= 11N + 4$. Donc, la complexité théorique est à tendance linéaire, c'est-à-dire $O(N)$.
L'encombrement mémoire ou complexité spatiale consiste en 3 variables, à savoir, *epsilon*, x, et *racine*.

6. Méthode de calcul de la complexité

On s'intéresse plus particulièrement à la complexité théorique qui indique le comportement général de l'algorithme indépendamment de la machine.

Règles de calcul de la complexité

On ne mesure pas vraiment le temps de calcul, car il dépend de la machine. On évalue, néanmoins, le nombre d'opérations élémentaires (additions, multiplications, etc.) à exécuter. Par convention, on attribue 1 unité à chaque opération quelle que soit sa nature. On obtient donc une estimation du temps de calcul à une constante multiplicative près.
Bien qu'on ait simplifié le processus de calcul, il convient également de fixer les règles à respecter pour chaque catégorie d'instruction.

Règle 1 : Composition séquentielle
Si I_1 et I_2 ont respectivement pour complexités $O(f(N))$ et $O(g(N))$, le bloc séquentiel $\mathbf{I_1; I_2}$ a pour complexité $O(\max(f(N), g(N)))$

Règle 2 : Instruction conditionnelle if (C) {$\mathbf{I_1}$} else {$\mathbf{I_2}$}
Si C, I_1 et I_2 ont respectivement pour complexités $O(f(N))$, $O(g(N))$ et $O(h(N))$, l'instruction conditionnelle **if (C) {$\mathbf{I_1}$} else {$\mathbf{I_2}$}** a pour complexité $O(\max(f(N), g(N), h(N)))$

Règle 3 : Boucle for
En supposant que l'instruction I a pour complexité $O(f(N))$, et qu'elle n'a aucun effet sur i et N, alors la complexité de la boucle
for (int $i = 0; i < N; i++$**){I}** est $O(N \cdot f(N))$

Règle 4 : Boucle While (C){I}
Si C, I et la boucle While ont respectivement pour complexités $O(f(N))$, $O(g(N))$ et $O(h(N))$, alors la complexité de l'instruction **While (C){I}** est $O(h(N) \cdot \max(f(N), g(N)))$

Calcul de la complexité d'un algorithme récursif

La complexité d'un algorithme récursif peut toujours s'exprimer par une équation récurrente. La réesolution des équations de récurrence n'est pas toujours triviale,

mais peut toujours être approchée.

- **Exemple 1**
  Calcul simplifié de la complexité en termes de nombre de multiplications effectuées dans un algorithme récursif qui calcule la fonction factorielle d'un nombre entier N.

```
Fonction fact( N : entier) : entier ;
Si (N = 0) Alors
   | retourner (1)
Sinon
  | retourner (N*fact(N − 1))
Fin Si
```

Ainsi, on a :

- C(0) = 0 pas de multiplication;
- Appel récursif. $\forall N \geq 1$, C(N) = 1 + C($N-1$) $\Rightarrow$ C(N)= N. Ce qui correspond à la réalité; $N!$ est bien évidemment calculée en N multiplications.

- **Exemple 2**
  On se propose d'étudier un cas plus complet et détaillé du calcul de la complexité d'un algorithme récursif. On s'appuie sur une version récursive de la fonction de Fibonacci exprimée par le code suivant :

```
Fonction fibo( N : entier) : entier;
(1) : Si ((N = 0) ou (N = 1)) Alors
          | (2) : retourner 1
      Sinon
        | (3) : retourner (fibo(N − 1) + fibo(N − 2))
      Fin Si
```

numéro	contrôle	logique	affectation	arithmétique	appels
(1)	2	1			
(2)			1		
(3)			1	1	C($N-1$)+C($N-2$)
total	2	1	2	1	C($N-1$)+C($N-2$)

Au total on a : C(N) = 6 + C($N-1$) + C($N-2$)
C(N)$\geq$ 6 + 2*C($N-2$), car C est une suite croissante
C(N)$\geq$ 6 + 2*(6 + 2*C($N-4$))

C(N)$\geq$ 6+ 2*6 + 6*(6 + C($N-6$))
C(N)$\geq$...
C(N)$\geq$ 6 + 2*6 + 4*6 + ...+ $2^{\frac{N}{2}}$* 6
C(N)$\geq$ 6* $\sum_{i=0}^{\frac{N}{2}} 2^i$ = 6* $(2^{\frac{N}{2}+1} - 1)$
Donc, C(N)= $\Omega(2^{\frac{N}{2}})$

De même, on a également

C(N) = 6 + C($N-1$) + C($N-2$)
C(N)$\leq$ 6+ 2*C($N-1$), car C est une suite croissante
C(N)$\leq$ 6 + 2*(6 + 2*C($N-2$))
C(N)$\leq$ 6 + 2*6 + 4*(6+ C($N-3$))
C(N)$\leq$...
C(N)$\leq$ 6 + 2*6 + 4*6 + ...+ 2^N * 6
C(N)$\leq$ 6* $\sum_{i=0}^{N} 2^i$ = 6 * $(2^{N+1} - 1)$
Donc, C(N)= $O(2^N)$

Chapitre 3

Structures de données séquentielles

3.1 Pile

- La pile est un ensemble, éventuellement vide, d'éléments empilés les uns sur les autres, et dont le seul élément accessible est celui du sommet.
- Les piles sont aussi appelées listes LIFO (Last In First Out), stacks, push-down lists (listes refoulées). Elles sont très utilisées dans les systèmes informatiques.
- Les opérations habituellement effectuées sur les piles sont :
  - Initialisation de la pile;
  - Vérification si la pile est vide;
  - Accès au sommet de pile;
  - Retrait de l'élément situé au sommet de la pile (dépiler);
  - Ajout d'un élément au sommet de la pile (empiler).
- Trois façons pour représenter et gérer la pile :
  - Certains langages de programmation (par exemple l'assembleur x86) permettent d'implanter et de gérer la pile. Il suffit de la déclarer et d'utiliser les primitives de manipulation PUSH (empiler) et POP (dépiler) disponibles.
  - Utilisation par un vecteur : écriture des procédures empiler, dépiler, initialisation et écriture de la fonction testant si la pile est vide, etc.
  - Utilisation d'une liste et écriture des procédures adéquates (empiler, dépiler, etc.) pour sa gestion.

3.1.1 Concrétisation de la pile par un vecteur

- **Déclaration**
  PILE : array [1..MAXI] of ELEMENT;
  SOMMET: 0..MAXI;

- **Initialisation**
  SOMMET := 0;

- **Retrait de l'élément du sommet de pile (Dépiler)**
  IF SOMMET > 0 THEN
  BEGIN
  X := PILE [SOMMET];
  SOMMET := SOMMET-1
  END
  ELSE ERREUR ("PILE VIDE")

- **Ajout d'un élément au sommet de pile (Empiler)**
  IF SOMMET < MAXI THEN
  BEGIN
  SOMMET := SOMMET + 1;
  PILE [SOMMET] := X
  END
  ELSE ERREUR ("DÉBORDEMENT");

- **Pile vide**
  IF SOMMET = 0 THEN pile_vide = TRUE

3.1.2 Concrétisation de la pile par une liste linéaire chainée

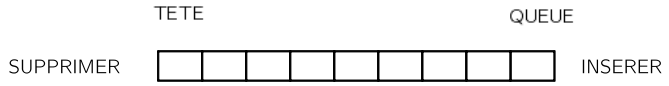
- Une pile sera représentée par une liste linéaire chainée, donc par un pointeur P qui indiquera l'élément sommet de la pile (tête ou sommet de liste).

- Lorsque la pile est vide ce pointeur aura pour valeur Nil.

- Comme on n'accède qu'à l'élément du sommet de la pile, les traitements adaptés au besoin seront efficaces, cependant à cause des pointeurs, une telle représentation prendra deux fois plus de place en mémoire qu'avec la représentation séquentielle qui utilise un vecteur. Ce que l'on gagne en allocation dynamique on le perd en place mémoire.

3.2 File

- Une file est une suite d'éléments dans laquelle toutes les insertions se font à une extrémité (en fin de file) et toutes les suppressions se font à l'autre extrémité (en début de file), comme c'est illustré schématiquement par la figure suivante.



  Les files sont appelées FIFO (First In First Out) et fonctionnent comme les files d'attente (le premier arrivé est le premier sorti ou servi).

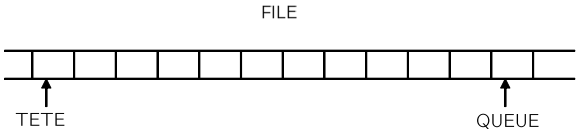
- Les opérations de gestion de la file sont les suivantes :

- Initialiser la file;
- Vérifier si la file est vide;
- Obtenir la valeur de l'élément de tête;
- Supprimer l'élément de tête (Défiler);
- Ajouter un élément en queue (Enfiler).

- Tout comme les piles, les files peuvent être représentées de plusieurs façons; une première représentation utilise un vecteur, tandis qu'une autre fait appel aux listes linéaires chainées.

3.2.1 Représentation de la file par un vecteur

Deux variables supplémentaires sont définies pour contrôler la file. Ces variables nommées QUEUE et TÊTE permettent de conserver les indices des éléments en queue et en tête de la file.



- **L'opération d'insertion**

```
IF QUEUE = N THEN WRITE ("DÉBORDEMENT")
ELSE
    BEGIN
      QUEUE := QUEUE + 1;
      FILE [QUEUE] := ELEMENT;
      IF TETE = 0 THEN TETE := 1
    END;
```

- **L'opération de suppression**

```
IF TETE = 0 THEN WRITE ("FILE VIDE")
ELSE
    BEGIN
      ELEMENT := FILE [TETE];
      IF TETE = QUEUE THEN
        BEGIN
          TETE := 0;
          QUEUE := 0
        END
      ELSE TETE := TETE + 1
    END;
```

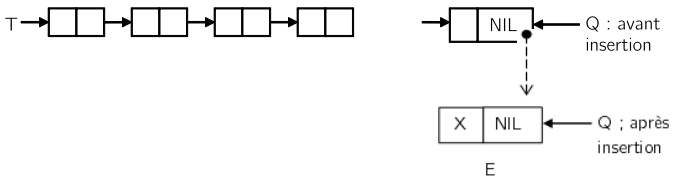
3.2.2 Représentation de la file par une liste linéaire chainée

- Une liste linéaire chainée peut également être utilisée pour représenter une file. On utilise dans ce cas deux pointeurs T et Q qui indiquent respectivement la tête et la queue de la file.

- **Insertion**
  L'insertion d'un élément dans la file se fera simplement par la séquence d'instructions suivante :

```
NEW (E);          "création d'une nouvelle cellule mémoire"
E↑.info := X;  "mettre la valeur X dans la partie information"
Q↑.pt := E;         "effectuer le chainage"
E↑.pt := NIL;    "pointeur du dernier élément de la file = NIL"
Q := E;               "référencer le dernier élément de la file par Q"
```

- Cette séquence d'instructions est illustrée schématiquement par la figure suivante :



- **Suppression**
  La suppression d'un élément de la file se fera selon l'instruction suivante :
  IF T = NIL THEN ("FILE VIDE")
  ELSE T := T↑.pt;

  Attention! La suppression ici est logique. Autrement dit, l'élément supprimé reste physiquement en mémoire. Pour libérer la mémoire et supprimer physiquement l'élément en question, il faut reconsidérer la branche ELSE comme suit :
  ELSE E := T; T := T↑.pt; DISPOSE (E).

- Une file vide sera représentée par T ayant pour valeur NIL et Q pointant sur T.

3.2.3 Conclusion

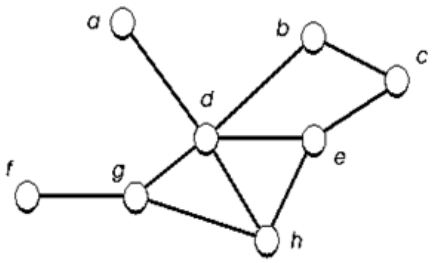
La représentation dynamique des piles et des files est très valable lorsqu'on ne connaît pas le nombre d'éléments présents en moyenne (ou maximum) dans la pile et dans la file.

Chapitre 4

Structures de données hiérarchiques

4.1 Notions élémentaires de la théorie des graphes

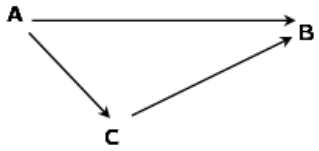
Un exemple de graphe à 8 sommets, comportant 10 arêtes est donné par la figure suivante :



- S = {*a, b, c, d, e, f, g, h*} est l'ensemble des sommets du graphe.
- A = {*ad, bc, bd, de, ec, eh, hd, fg, dg, gh*} est l'ensemble des arêtes.
- *dbced, gdhg, dehd, etc.*, sont des cycles (circuits).
- *adg* est un chemin qui mène de a vers g ; *adb* est aussi un chemin de *a* vers *b*.
- La longueur d'un chemin est le nombre d'arcs (arêtes) qui constituent le chemin en question.
- *adgf* n'est pas un circuit, puisqu'il n'y a pas de chemin qui mène de *a* vers *a*.

Un graphe orienté ou digraphe est un graphe dont les arêtes sont orientées (marquées par des flèches). Dans ce cas, ces arêtes s'appellent arcs.

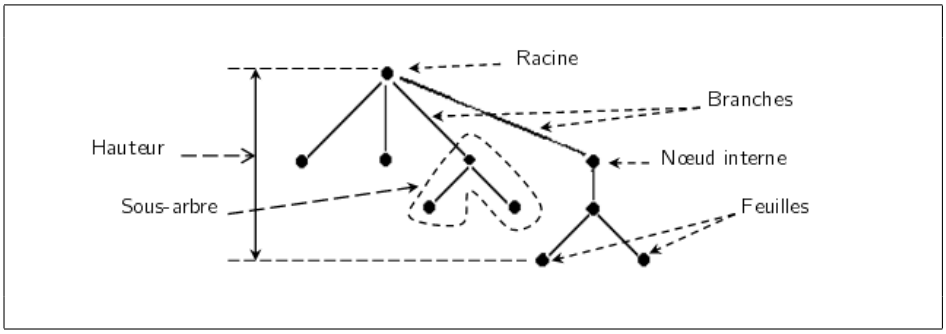
Ci-après, le schéma d'un graphe orienté sans circuit :



4.2 Notions générales sur les arbres

Un arbre est un graphe connexe sans cycle (circuit) tel que :

- Il existe un sommet (nœud) et un seul où il n'arrive aucun arc. Ce sommet s'appelle la racine.
- Il arrive un arc et un seul en tout autre sommet.
- Les sommets qui n'ont pas de successeurs sont appelés feuilles.
- A chaque nœud ou sommet on peut affecter un niveau (profondeur ou hauteur) qui est la longueur du chemin qui joint la racine à ce nœud augmentée de 1.



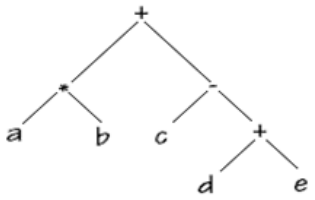
- La racine est de niveau 1. Certains auteurs affectent le niveau 0 à la racine.
- Le degré d'un nœud est égal au nombre de ses nœuds fils.
- Le degré d'un arbre est égal au plus grand des degrés de ses nœuds.
- La taille d'un arbre est égale au nombre total de ses nœuds.
- La hauteur d'un arbre est le nombre de nœuds du chemin le plus long dans l'arbre.
- La hauteur de l'arbre vide est égale à 0.

4.2.1 Arbre binaire

Définitions

- Un arbre binaire (ou binaire-unaire) est un arbre dans lequel chaque nœud a au plus deux fils.

  Par exemple, sur la figure ci-dessous, un arbre binaire peut être utilisé pour représenter l'expression a * b + (c - (d + e)).



- Un arbre binaire entier est un arbre binaire dont tous les nœuds possèdent zéro ou deux fils.
- Un arbre binaire parfait est un arbre binaire entier dont toutes les feuilles sont à la même distance de la racine.
- L'arbre binaire complet est un arbre binaire dont les feuilles ont pour profondeur n ou $n-1$ pour n donné.
- Un arbre binaire est dégénéré si chacun de ses nœuds internes (non feuilles) n'a qu'un seul descendant.
- Un facteur d'équilibre d'un nœud x est défini comme étant la différence entre la hauteur du sous-arbre droit et celle du sous-arbre gauche.
  On le note comme suit :
  FE (x) = h (fg (x)) - h (fd(x)) ou FE (x) = h (fd (x)) - h (fg(x)).
- Un arbre binaire de recherche (ABR) est un arbre binaire ordonné tq :

$$\text{fg } (x) < x < \text{fd } (x).$$

  On peut aussi imaginer la relation inverse, i.e. fg $(x) > x >$ fd (x).

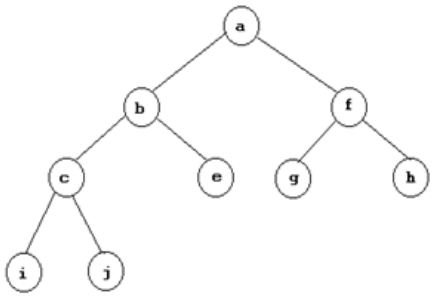
- Un ABR est dit équilibré si pour tout nœud x de l'arbre on a :

$$| \text{ FE}(x) | \leq 1.$$

Parcours d'arbres binaires

Trois parcours en profondeur bien connus et couramment utilisés dans le cadre des arbres binaires sont :

Le parcours préordre (prefixé) :	r T1 T2
Le parcours inordre (infixé ou symétrique) :	T1 r T2
Le parcours postordre (postfixé ou suffixé) :	T1 T2 r



En parcourant l'arbre de l'exemple de la figure ci-dessus en préordre, inordre et postordre, on obtient respectivement les trois résultats suivants :

Préordre : a, b, c, i, j, e, f, g, h.
Inordre : i, c, j, b, e, a, g, f, h.
Postordre: i, j, c, e, b, g, h, f, a.

Algorithmes récursifs de parcours d'arbres binaires

- **Parcours préfixé**

```
procédure préfix (b : arbre_bin) ;
Si (b ≠ NIL) Alors
    écrire (b↑.info) ;
    préfix (b↑.gauche) ;
    préfix (b↑.droit)
Fin Si
```

- **Parcours infixé**

```
procédure infix (b : arbre_bin) ;
Si (b ≠ NIL) Alors
    infix (b↑.gauche) ;
    écrire (b↑.info) ;
    infix (b↑.droit)
Fin Si
```

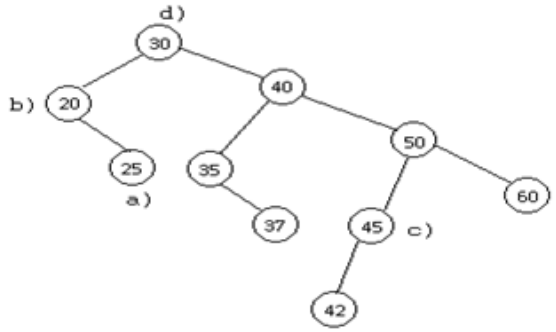
- **Parcours postfixé**

```
procédure postfix (b : arbre_bin) ;
Si (b ≠ NIL) Alors
    | postfix (b↑.gauche) ;
    | postfix (b↑.droit)
    | écrire (b↑.info)
Fin Si
```

Opérations sur l'ABR (arbre binaire de recherche)

- Recherche : À chaque étape, soit l'élément recherché est trouvé, soit un sous-arbre est éliminé. (Recherche dichotomique, complexité $\log_2$ (N)). N étant la taille de l'arbre (nombre de nœuds de l'arbre).
- Insertion: L'élément inséré est toujours une feuille.
- Suppression : Plus délicate.

Par exemple, soit à réaliser les opérations de suppression notées respectivement a), b), c) et d) comme mentionné sur l'ABR de la figure ci-dessous :



- Soit n le pointeur sur l'élément à supprimer;
- # signifie possède un descendant. NIL : pas de descendant. Fg (fils gauche); Fd (fils droit).

	Fg	Fd	Action
a)	NIL	NIL	Remplacer n par NIL (pointeur de 20 à NIL)
b)	NIL	#	Remplacer n par Fd (n) (pointeur de 30 sur 25)
c)	#	NIL	Remplacer n par Fg (n) (pointeur de 50 sur 42)
d)	#	#	1. Rechercher le plus petit descendant du sous-arbre droit, soit p 2. Remplacer Info (n) par Info (p) 3. Remplacer p par Fd (p)

35 est le plus petit descendant du sous-arbre droit. Soit donc à remplacer 30 par 35. Remplacer ensuite 35 par 37 le Fd (35).

On peut alternativement réaliser l'action d) avec le plus grand descendant du sous-arbre gauche, soit 25. On aura ainsi à remplacer 30 par 25 et ensuite à remplacer 25 par NIL qui est Fg (20).

Remarques

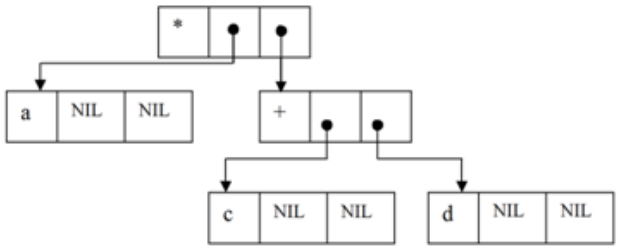
- En ce qui concerne l'application de l'action d), on a le choix, mais une fois le choix effectué, il faut le respecter. Autrement dit, si on opte pour le plus petit descendant droit (ou bien le plus grand descendant gauche), on doit rester sur ce choix. L'algorithme doit fonctionner avec une seule option.

- Avec l'exemple précédent, les actions a), b), c) et d) ont été réalisées indépendamment les unes des autres.
  Si elles avaient été réalisées successivement, l'une après l'autre, le résultat serait différent. A titre indicatif, si l'action b) avait été réalisée suite à l'action a), on aurait obtenu un résultat autre que celui qu'on avait obtenu dans l'exemple ci-dessus.

Implémentations et représentations internes des arbres binaires

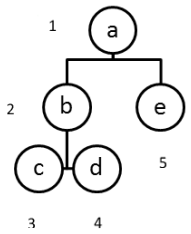
- On peut implémenter un arbre binaire en utilisant une structure de données dynamique ou une structure de données statique.

  La représentation mémoire de l'arbre abstrait de l'expression a * (c + d) par une liste chaînée non linéaire est illustrée par la figure suivante :



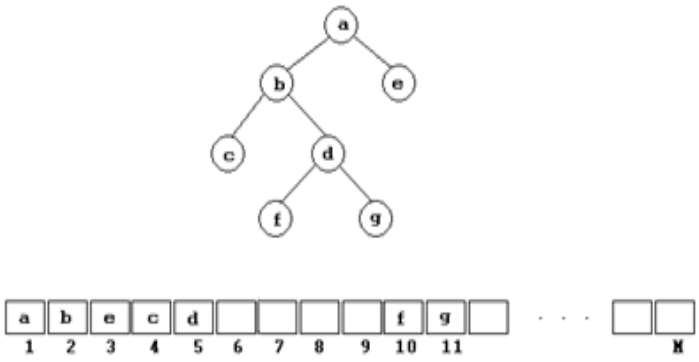
- Il existe deux catégories de représentations internes statiques : la représentation statique standard et la représentation statique séquentielle :

  – Le modèle statique standard permet de représenter explicitement les nœuds fils comme sur la figure ci-dessous :



index	1	2	3	4	5
fils gauche	2	3	0	0	0
noeud	a	b	c	d	e
fils droit	5	4	0	0	0

– La représentation statique séquentielle (représentation implicite des fils). Par exemple, le nœud à la position k est le père implicite des nœuds aux positions 2k et 2k + 1.



4.2.2 Arbre général

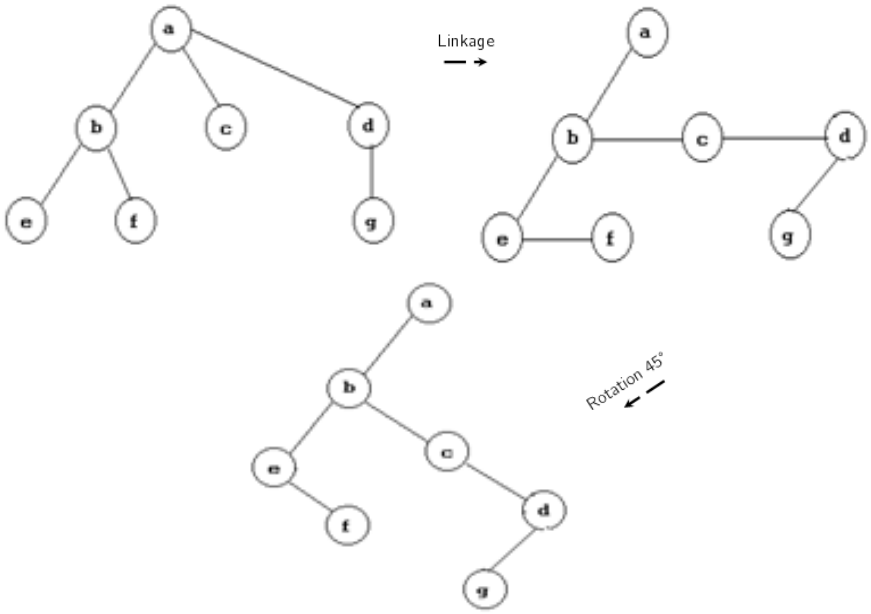
Caractéristique

Tout comme l'arbre binaire, un arbre général est représenté par une structure arborescence dont chaque nœud contient une seule clé. Mais, contrairement à l'arbre binaire, les nœuds d'un arbre général possèdent un nombre quelconque (potentiellement infini) de fils.
On l'appelle également arbre n-aire de taille n non bornée. A ne pas confondre avec un arbre n-aire avec n borné, qui est une forme de généralisation de l'arbre binaire et qui n'est pas étudié dans ce cours.

La manipulation d'un arbre général en machine nécessite qu'il soit implémenté sous forme d'un arbre binaire qui est généralement plus facile à manipuler. Ainsi, dans ce qui suit, on montre comment on peut passer d'un arbre général à un arbre binaire.

Transformation d'un arbre général en un arbre binaire

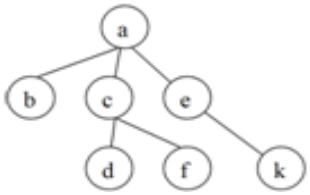
- Lier les nœuds frères dans une liste linéaire chaînée;
- Rotation de 45° dans le sens des aiguilles d'une montre.



Parcours

Les parcours les plus communément utilisés sont :

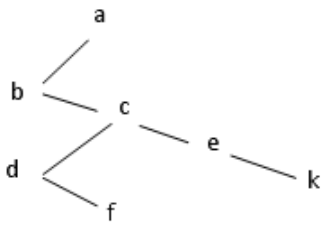
Préordre :	r T1 T2 ... Tn
Inordre :	T1 r T2 ... Tn
Postordre :	T1 T2 ... Tn r



Les trois parcours de l'arbre de la figure ci-dessus donnent respectivement :

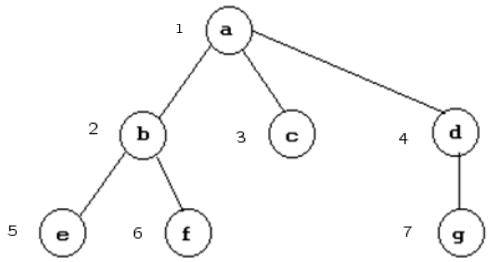
Préordre : a b c d f e k
Inordre : b a d c f e k
Postordre : b d f c k e a

Si l'on transforme cet arbre en un arbre binaire on obtient l'arbre de la figure suivante :



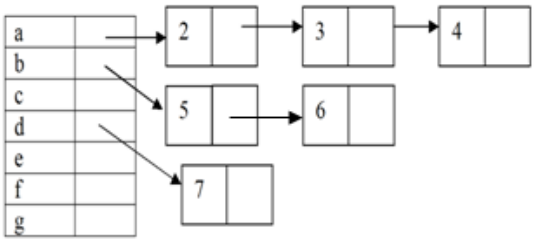
Représentations internes d'un arbre général

- On peut représenter un arbre général de plusieurs manières différentes. On peut utiliser la représentation statique standard; mais, le tableau représentatif risque d'être creux (perte d'espace) comme c'est le cas avec la figure suivante :



index	nœud	fils 1	fils 2	fils 3
1	a	2	3	4
2	b	5	6	-
3	c	-	-	-
4	d	7	-	-
5	e	-	-	-
6	f	-	-	-
7	g	-	-	-

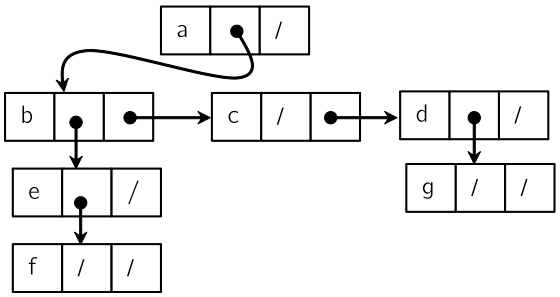
- On peut remédier à la perte d'espace engendrée par la représentation statique standard en adoptant un autre modèle de représentation. On utilise, à cet effet, un tableau de nœuds (statique). Ensuite, à chaque nœud est associée la liste de ses fils (dynamique). C'est la représentation semi statique ou semi dynamique de la figure suivante :



- On peut également adopter une tout autre représentation nommée représentation statique dense dite aussi représentation par index parent comme illustré par la table de la figure ci-après :

index parent	-	1	1	1	2	2	4
nœud	a	b	c	d	e	f	g
index nœud	1	2	3	4	5	6	7

- Enfin, on peut représenter l'arbre général de manière purement dynamique par une liste chainée après l'avoir transformé en un arbre binaire. Ci-après le modèle de représentation :



Arbre n-aire

Définition

Un arbre *n-aire* admet, pour chaque nœud, au plus *n* descendants directs (fils). C'est, en quelque sorte, une généralisation de l'arbre binaire de recherche (ABR).

La structure d'un nœud d'un arbre *n-aire* est définie comme suit :

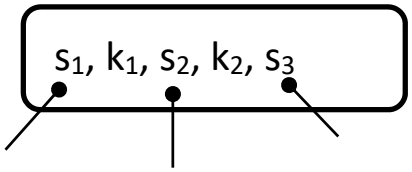
(s_1, k_1, s_2, ...k_{n-1}, s_n) avec (n – 1) clés ordonnées $k_1 < k_2$ ....< k_{n-1} et s_1, s_2, ... s_n n descendants directs (fils).

. Les clés des éléments $\in s_1 \leq k_1$;

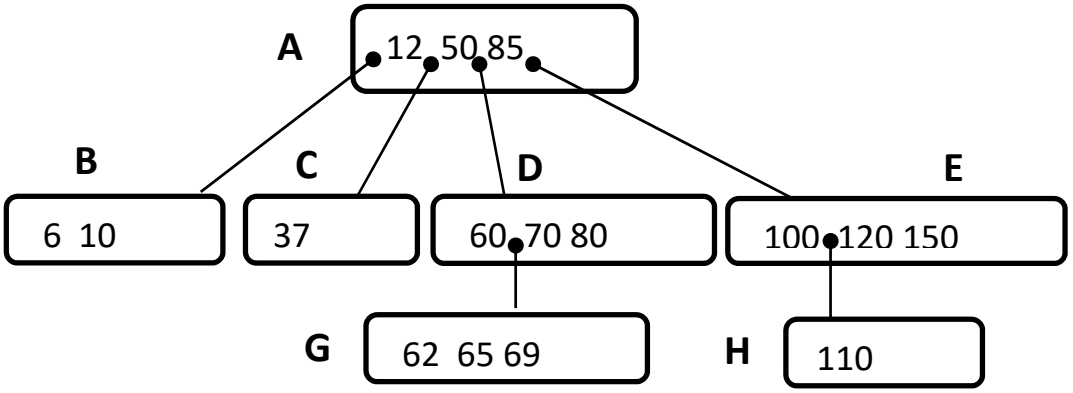
. k_{j-1} < Les clés des éléments $\in s_j \leq k_j$ pour j = 2, 3, ...n ;

. les clés des éléments $\in s_n > k_{n-1}$.

Par exemple, le nœud suivant contient 2 clés : k_1 et k_2 et 3 pointeurs s_1, s_2 et s_3 vers les 3 descendants directs du nœud en question.



L'arbre suivant est un arbre 3-aire ou *m*-aire avec $m = 3$.



Opérations sur l'arbre n-aire

- **Recherche** : C'est une généralisation de l'ABR : $O\ (Log_{degré}\ (N))$.

On rappelle que *degré* est le nombre maximum de fils d'un nœud.

- **Insertion**

  Rechercher la clé.

. Si clé non trouvée (on est sur une feuille, soit *p*) et *p* non complet, c'est-à-dire : le nombre de clés dans *p* < *degré* − 1, on insère la clé dans *p*.

. Si clé non trouvée et nœud *p* complet, on fait les opérations suivantes :

- Allocation d'un nouveau nœud ;
- Insérer la clé dans ce nouveau nœud ;
- Placer le nœud en question comme fils du nœud surchargé.

- **Suppression**
  - Logique : Laisser la clé au niveau du nœud et marquer celui-ci. Le nœud reste utilisé pour l'algorithme de recherche ;
  - Physique (coûteuse) : technique similaire à celle des ABR.

    . Si la clé à supprimer possède un sous-arbre gauche ou droit vide, alors simplement supprimer la clé et tasser le nœud. Si c'est la seule clé dans le nœud, libérer le nœud.

    . Si la clé à supprimer a des sous-arbres gauche et droit tous les deux non vides, alors trouver la clé successeur (qui doit avoir un sous-arbre gauche vide). Remplacer la clé à supprimer par la clé successeur et tasser le nœud qui contenait cette dernière. Si la clé successeur est la seule dans le nœud alors libèrer le nœud.

    Ce dernier point peut être réalisé en considérant la clé prédécesseur (qui a un sous-arbre droit vide). On remplace alors la clé à supprimer par la clé prédécesseur et on tasse le nœud qui contenait cette dernière. Si la clé prédécesseur est la seule clé dans le nœud on libère ce nœud.

On applique alors la procédure sur l'arbre 4-aire donné ci-dessus comme suit.

1- Insertion de la clé 11. On l'insère directement au niveau du nœud-feuille B qui est un nœud-feuille incomplet.

2- Insertion de la clé 61. Le nœud G est complet (surchargé) alors on alloue un nouveau nœud et on y place la clé 61 à l'endroit approprié tel que 61 < 62. 62 est une clé dans G. On place ce nouveau nœud comme fils du nœud G sachant que 61 < 62.

3- Suppression des clés 6, 10 et 37. On supprime les clés 6 et 10 et on tasse le nœud B. Il reste uniquement la clé 11 dans B.

En supprimant la clé 37, le nœud C se vide, on libère alors ce nœud C.

4- Suppression de la clé 85. On est dans le cas où sous-arbres droit et gauche sont non vides. Ainsi, la clé successeur qui a un sous-arbre gauche vide, c'est 100. On remplace donc 85 par 100 et on tasse le nœud E qui contenait la clé 100.

5- Suppression de 50 du nœud A, en supposant que la clé 37, n'a pas été supprimée du nœud C. Ainsi, la clé prédécesseur, c'est 37. Donc, on remplace 50 par 37. Mais, comme 37 est la seule clé dans C, alors on libère le nœud C.