UNIVERSITÉ DE LORRAINE | IUT nancy Charlemagne Département Informatique

Algorithmique Avancée

PUBLIC CONCERNÉ : formation initiale, 2ème année

DATE : 2016/2017

IUT NANCY-CHARLEMAGNE
2 TER BD CHARLEMAGNE - CS 55227
54052 NANCY CEDEX
TÉL. 03 54 50 38 20/22
FAX 03 54 50 38 21
iutnc-info@univ-lorraine.fr
http://iut-charlemagne.univ-lorraine.fr

Algorithmique Avancée

Chapitre 1 : Les Problèmes Récursifs

I. Introduction à la récursivité

II . Les problèmes définis par récurrence

Exercice 1 : Factorielle n

Exercice 2 : Nième ligne du triangle de Pascal

Exercice 3 : conversion en binaire d'un entier positif

Exercice 4 : Nombre de Fibonacci

Exercice 5 : tours de Hanoï

Exercice 6 : expressions fonctionnelles

III. Les algorithmes de recherche avec retour arrière

Exercice 7 : recherche de chemins dans un graphe avec circuits

Exercice 8 : problème des 8 reines

IV. Diviser pour régner

Exercice 9 : recherche dichotomique dans un tableau trié

Exercice 10 : recherche de points d'accumulations dans un espace à deux dimensions

Exercice 11 : un exemple de tri par dichotomie: le tri rapide (ou méthode de Hibbard)

V. Les algorithmes liés aux structures d'informations récursives

1. **Les listes**
   Exemple : impression des éléments d'une liste
   Exercice 12: retourner une liste ***(optionnel)***
2. **Les arbres** (*Ils seront traités dans le chapitre trois*)

Chapitre 2 : La complexité des algorithmes

Les différents types de complexité
L'ordre de grandeur
Exercices

Chapitre 3 : Les arbres

Les arbres binaires
Les Arbres n-aires

Chapitre 1

Les Problèmes Récursifs

I. Introduction à la récursivité

La récursivité est une technique de résolution de problème utilisée en informatique. Elle permet d'écrire des solutions courtes pour des problèmes complexes : on réduit le problème initial à un problème similaire mais de taille plus petite. Elle permet de trouver une solution élégante et claire à certains problèmes, comme celui des Tours de Hanoï.

Une fonction qui contient un appel à elle-même est dite récursive. L'algorithme correspondant est aussi dit récursif. Tout algorithme récursif comporte une instruction (ou un bloc d'instructions) nommée *condition d'arrêt*, qui garantit la fin des appels récursifs et qui porte sur les paramètres.

Pour tout problème récursif, il faut :

- préciser la condition d'arrêt,
- définir une formule de récurrence.

Une fonction récursive F contient :

- des instructions résolvant directement les cas particuliers (appelés aussi cas triviaux);
- des instructions qui partagent F en des sous problèmes tous de même nature que F mais de complexité moindre ;
- des instructions calculant les solutions des sous problèmes (des appels à F)
- des instructions combinant les solutions intermédiaires pour calculer la solution de F.

Algorithme général :

```
fonction F_récursive (paramètres)
début
   si   <condition d'arrêt> alors
            instructions résolvant directement les cas particuliers
  sinon
            instructions
            appel(s) récursif(s) à F_récursive(paramètres modifiés)
            instructions
            (l'ordre peut changer selon les problèmes)
   fsi
fin
```

Remarque
1 - Il faut noter que les paramètres passés à la fonction récursive ***doivent*** changer d'une étape à une autre, sinon les appels récursifs ne se termineront jamais. Ils sont généralement « plus simples » d'un appel au suivant et doivent modifier la condition d'arrêt au bout d'un temps fini.

2 - Il ne faut jamais mettre l'appel récursif en première instruction de l'algorithme récursif, sinon on aura un appel récursif infini.

Classes d'algorithmes

On peut distinguer 4 classes d'algorithmes récursifs en fonction des types de problèmes à résoudre et des solutions proposées :

- les problèmes définis par récurrence
- les recherches avec retour arrière
- les techniques de découpage du type « diviser pour régner »
- les problèmes liés aux structures d'informations récursives.

Nous étudierons des exemples dans chacune de ces classes.

II. Les problèmes définis par récurrence

Exercice 1 : factorielle n

Ecrire une fonction qui calcule factorielle n .

Rappel de la définition de factorielle n : $fact(n) = n * fact(n\text{-}1),\ n>0$
$fact(n) = 1,\ n = 0$

Exercice 2 : triangle de Pascal

Ecrire une fonction qui calcule la $n^{ième}$ ligne du triangle de Pascal.

Exemple :
ligne 1 : 1
ligne 2 : 1 1
ligne 3 : 1 2 1
ligne 4 : 1 3 3 1
ligne 5 : 1 4 6 4 1

donnée: un entier, n
résultat: un tableau d'entiers, ligne

cas triviaux:
n = 1 ligne[1] = 1

n = 2 ligne[1] = 1
ligne[2] = 1

cas général:
n > 2 ligne[1] = 1
ligne[n] = 1
ligne[j] = ligne_précédente[j-1] + ligne_précédente[j] pour 2≤j<n
où ligne_précédente est la (n-1) ième ligne du triangle de Pascal

Exercice 3 : conversion en binaire

Ecrire une fonction récursive qui imprime la conversion en binaire d'un entier positif.

Exercice 4 : nombre de Fibonacci

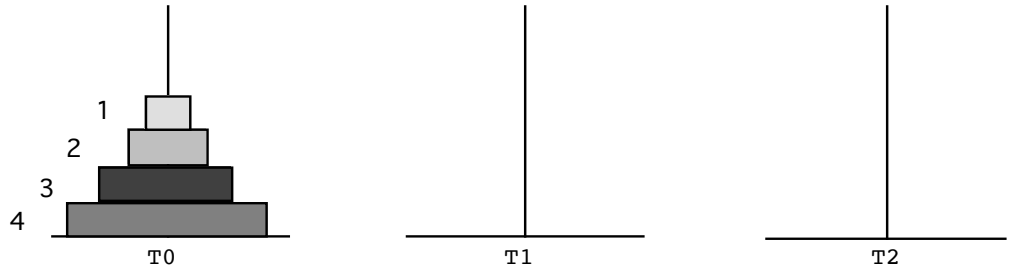
Ecrire une fonction récursive qui calcule le $n^{ième}$ nombre de Fibonacci.

Rappel : Le nombre de Fibonacci F_n est défini comme suit :

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n \geq 2 \end{cases}$$

Exercice 5 : les tours de Hanoï

Soient trois tours. Sur l'une d'entre elles sont empilés des disques par taille décroissante:



Le but du jeu consiste à déplacer la pyramide de *n* disques de la tour de départ vers une des deux tours libres.

Règles de déplacement :

- on ne peut déplacer qu'un seul disque à la fois,
- un disque ne peut pas se trouver sur un disque de diamètre inférieur.

Ecrire une fonction récursive indiquant la suite des déplacements permettant d'arriver à l'état de satisfaction.

Les paramètres de la fonction sont : le nombre de disques à déplacer, le nom de la tour de départ, le nom de la tour d'arrivée et le nom de la tour intermédiaire.

Exercice 6 : expressions fonctionnelles

La technique la plus classique pour écrire des expressions fonctionnelles consiste à utiliser un système de parenthèses, de virgules et de symboles pour les fonctions et les variables.

Exemples : $f(x)$ $g(x,y)$ $h(x,g(x,y))$

Une autre technique consiste à supprimer les parenthèses et les virgules et à utiliser l'arité du symbole fonctionnel pour interpréter la suite de caractères.

Exemples :

symbole	*f*	*x*	*g*	*x*	*y*	*h*	*x*	*g*	*x*	*y*
arité	1	0	2	0	0	2	0	2	0	0

Ecrire l'algorithme qui rétablit les parenthèses et les virgules. On suppose qu'on dispose d'une fonction qui lit un symbole et d'une fonction qui donne l'arité correspondant à un symbole:

```
fonction lireSymbole ( ) : chaîne
fonction aritéSymbole (symbole : chaîne) : entier
```

donnée : expression fonctionnelle sans parenthèse ni virgule

résultat : impression de l'expression fonctionnelle avec parenthèses et virgules

III. Les algorithmes de recherche avec retour arrière

(*les techniques de backtracking*)

Le principe des algorithmes de recherche avec retour arrière consiste à faire des choix et prendre des décisions puis de remettre en question ces décisions afin d'éviter une impasse. Il s'agit de revenir légèrement en arrière sur des décisions prises afin de sortir d'un blocage. L'idée est d'essayer chaque possibilité (combinaison) jusqu'à trouver la bonne.

Pendant la recherche, si on essaie une alternative qui ne satisfait plus une contrainte, on effectue un retour arrière à un point où d'autres alternatives s'offraient à nous, et on essaie la possibilité suivante. Si on n'a plus de tels points la recherche échoue. Le point fort du « backtracking » est que beaucoup de ses réalisations évitent d'essayer un maximum de combinaisons partielles, diminuant ainsi le temps d'exécution.

Algorithme général (informel)

(*Trouver une seule solution pour un problème donné*)

```
fonction rechercherSolution (...) : booléen
début
   si solution complète alors
      infructueux ← faux
   sinon
      infructueux ← vrai
      initialiser la sélection des candidats
      tant que (infructueux) et (il y a des candidats)
         choix d'un candidat
         si  acceptable alors
                enregistrer
                infructueux ← rechercherSolution (...)
                  si  infructueux alors
                           effacer enregistrement
                  fsi
            fsi
         ftant
   fsi
   retourne infructueux
fin
```

Le choix d'un candidat doit satisfaire certaines conditions. Si ces conditions sont satisfaites alors, ce candidat est acceptable. Dans ce cas, le candidat doit être enregistré. Il s'agit d'une construction partielle de la solution. Si on n'a pas atteint la solution finale, on applique l'algorithme de nouveau. Si après les appels récursifs on arrive à un blocage (infructueux), c'est que cette solution partielle n'est pas la bonne (le chemin construit n'amènera pas à la solution finale), et dans ce cas, on enlève le candidat choisi à la dernière étape de la solution partielle, et on continue avec un autre candidat.

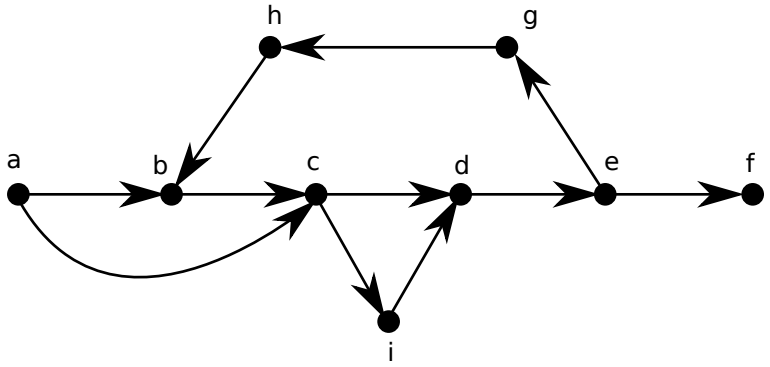
L'algorithme informel suivant montre le cas de la recherche de toutes les solutions.

```
fonction chercherToutesSolutions (...)
début
   si solution complète alors
      imprimer la solution
   sinon
      initialiser la sélection des candidats
      pour chaque candidat faire
         choix du candidat
         si  acceptable alors
             enregistrer
             chercherToutesSolutions (...)
              effacer enregistrement
          fsi
      fpour
   fsi
fin
```

Exercice 7 : recherche de chemins dans un graphe avec circuits

Rechercher tous les chemins dans un graphe avec circuits.

Exemple de graphe :



Un graphe est un ensemble de *nœuds* (ou *sommets*) et de *relations* (ou *arcs*) entre ces nœuds. Le graphe donné en exemple a pour nœuds : a, b, c, d, e, f, g, h et i. Le nœud début du graphe est *a* et le nœud fin *f*. Un chemin commence en *a*, se termine en *f* et ne passe jamais 2 fois par le même nœud. Les chemins du graphe donné en exemple sont les suivants : a b c d e f, a b c i d e f, a c d e f, a c i d e f.

Méthode de résolution :

On construit les chemins un à un, nœud par nœud. Si, à un instant donné, on se trouve au nœud *extrémité* qui est l'extrémité d'un début de chemin, on le complète en choisissant un des nœuds successeurs de *extrémité*. Le nœud choisi ne doit pas déjà appartenir au chemin. Si c'est le cas et qu'il n'y a pas d'autre choix possible, on remet en question le choix de *extrémité* ($\Rightarrow$ retour arrière).

Exercice 8 : problème des 8 reines

Placer 8 reines sur un échiquier de façon à ce qu'aucune ne soit en prise avec une autre (même ligne, même colonne, même diagonale).

Supposons que l'on place les reines colonne par colonne. La $i^{ème}$ reine sera placée dans la $i^{ème}$ colonne. Soit *placerReine(i)* la fonction chargée de placer la $i^{ème}$ reine (pour i de 1 à 8). Si cette fonction tombe en échec, on remet en question les choix précédents. On s'arrête quand toutes les reines sont bien placées.

IV. « Diviser pour régner »

Exercice 9 : recherche dichotomique

Ecrire l'algorithme de la fonction qui recherche l'indice d'un élément donné dans un tableau trié. La recherche sera dichotomique.

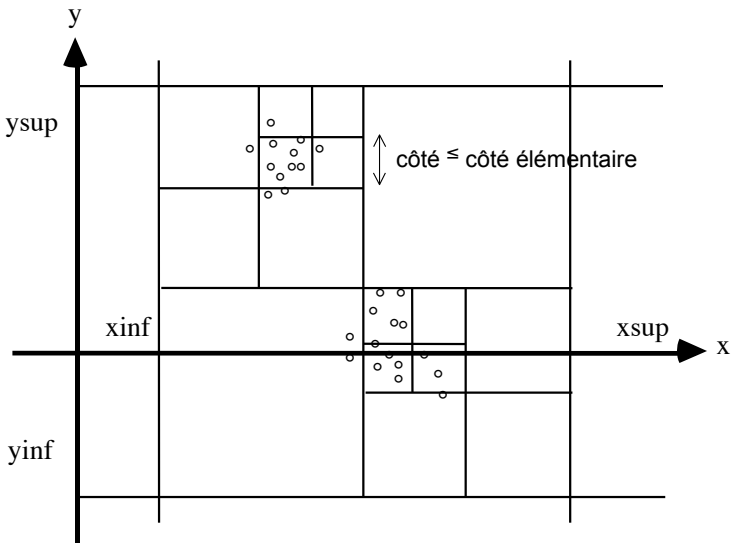
données :

- un tableau trié de n éléments, *tab*,
- les bornes inférieur (*borne_inf*) et supérieur (*borne_sup*) du tableau,
- l'élément cherché, *élément*.

résultat : l'indice où se trouve l'élément dans *tab*, s'il y est, -1 sinon.

Exercice 10 : recherche de points d'accumulation

Etant donné un nuage de points dans un espace à 2 dimensions (cf. figure ci-dessous), on cherche les **points d'accumulation** de ce nuage. On suppose que l'espace de recherche, noté E ∈R X R, est borné et forme un carré. On partage E en 4 carrés de dimensions égales. Si un carré est assez petit (de côté ≤ *côté_élémentaire*, *côté_élémentaire* donné), et contient au moins *nb_min* points (*nb_min* donné), il est un **point d'accumulation.** Le processus de partage est répété tant que cela est nécessaire.



nb_min = 5 ; E est borné par xinf, xsup, yinf et ysup.
On trouve 3 points d'accumulation.

Ecrire un algorithme récursif fournissant l'ensemble des points d'accumulation par partages successifs. Pour chaque point d'accumulation trouvé, on imprime les coordonnées de son milieu. Le nuage de points à considérer est donné sous la forme d'une liste de points, un point étant caractérisé par ses coordonnées. Les bornes de l'espace de recherche sont données par les valeurs minimales et maximales en abscisses et en ordonnées.

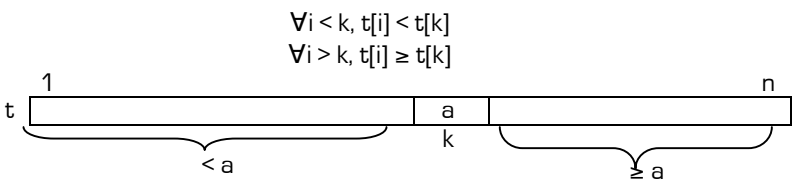
Exercice 11 : tri par dichotomie

On cherche à trier un tableau *t* d'entiers par dichotomie. Le principe est le suivant:

- partager le tableau *t* en 2 sous-tableaux *t1* et *t2* tels que si *x* est un élément de *t1* et *y* un élément de *t2* alors $x \leq y$,
- recommencer avec *t1* et *t2* jusqu'à obtention de sous-tableaux à un élément.

Les différentes méthodes de tri par dichotomie se distinguent par le partage. Nous allons étudier le *tri rapide* ou *méthode de Hibbard*.

Le partage se fait ainsi : choisir un élément du tableau, par exemple le premier, et le placer de telle sorte que tous les éléments plus petits que lui se trouvent placés avant et tous les éléments plus grands après. Si *a* est cet élément et *k* sa place, on a :



On obtient ainsi 2 sous-tableaux :

- de 1 à k-1
- de k+1 à n

et un élément bien placé.

On réitère le processus sur chacun des 2 sous-tableaux.

Question 1 : Ecrire l'algorithme itératif de la fonction qui place les éléments par rapport à la 1ère valeur du tableau sans utiliser de tableau intermédiaire et en parcourant le tableau une seule fois.

Question 2 : Ecrire l'algorithme récursif de la fonction qui trie les éléments du tableau.

V. Les algorithmes liés aux structures de données récursives

La principale structure de données récursive est l'arbre que nous étudierons dans un chapitre suivant. Il est également possible de manipuler les listes de manière récursive.

Les listes

Une liste est une structure de données récursive car elle est soit vide, soit formée du premier élément de la liste suivi d'une autre liste (la liste privée de son premier élément) :

liste = liste vide | valeur liste

Exemple

Ecrire une fonction récursive qui imprime les éléments d'une liste.

```
fonction imprimerListe ( l : liste (V), p :Place)
  début
        si non finliste(l,p)  alors
                    écrire (val(l,p))
                    psuiv ← suc(l, p)
                    imprimerListe  ( l, psuiv )
        fsi
  fin

Lexique
        l : liste (V), liste dont on veut imprimer les éléments
        p : Place, place courante, tête(l) au 1er appel
        psuiv : Place, place suivant p
```

Exercice (*optionnel*) : inversion de liste

Ecrire l'algorithme de la fonction récursive qui crée et retourne la liste inverse d'une liste donnée.

Exemple :

liste	2 → 4 → 6 → 9 → nil
liste inversée	9 → 6 → 4 → 2 → nil

Chapitre 2

Complexité des algorithmes

1. Position du problème

Pour résoudre un problème donné, on peut trouver plusieurs algorithmes qui fournissent la même solution. On cherche généralement l'algorithme le plus performant et le plus efficace. Afin de déterminer l'efficacité d'un algorithme à résoudre un problème donné, il faut effectuer une analyse de cet algorithme. Il s'agit d'évaluer les ressources utilisées par l'algorithme. Les ressources peuvent être le temps, la mémoire, le nombre de processeurs, la bande passante d'un réseau de communication, etc. Généralement, la complexité temporelle est la plus importante. On exprime le coût d'un algorithme en fonction de la taille des données.

Il faut noter que lorsqu'on analyse l'efficacité d'un algorithme en terme de temps, on ne s'intéresse pas à déterminer le nombre de cycles du microprocesseur nécessaire pour l'exécution de l'algorithme, puisque ceci dépendra de la machine utilisée pour l'exécution de l'algorithme. De plus, on ne voudrait pas compter chaque instruction exécutée, puisque ce nombre peut dépendre du langage de programmation utilisé pour implémenter l'algorithme. La mesure de l'efficacité d'un algorithme (et donc sa complexité) doit être indépendante de la machine, du langage de programmation, du programmeur, et tous les détails liés à l'implémentation.

Objectif : proposer des méthodes qui, pour la résolution d'un problème donné, permettent d'estimer le coût d'un algorithme, de comparer deux algorithmes différents (sans avoir à les programmer effectivement)

L'analyse de la complexité consiste à déterminer une fonction associant un coût en unité de temps à chaque entrée soumise à l'algorithme. En fait, on peut se contenter d'associer un coût à un paramètre entier *n* qui résume la taille des données.

Exemples :
- pour la recherche d'un élément dans un tableau : *n* est la taille du tableau,
- pour le tri d'une liste : *n* est la taille de la liste,
- pour le calcul d'un terme d'une suite définie par une relation de récurrence (par exemple *Fibonacci*) : *n* est l'indice du terme.

2. Les différents types de complexité

On définit trois mesures de complexité : la complexité dans le meilleur des cas, la complexité en moyenne et la complexité dans le pire des cas. Quand on parle de complexité d'un algorithme sans préciser laquelle, c'est souvent de la complexité temporelle dans le pire des cas. La complexité dans le meilleur des cas n'est pas très utilisée. La complexité en moyenne est celle qui révèle le mieux le comportement « réel » de l'algorithme, cependant elle est la complexité la plus difficile à calculer (Il faut avoir un modèle de répartition des données nécessitant en général des techniques mathématiques non élémentaires).

Exemple 1
Soit l'algorithme suivant :

pour j de 1 à n faire	*n* fois
x ← x + 3*j	*c*
fpour	

On note par *c* le coût de l'instruction `x ← x + 3*j`
Le coût total de cet algorithme (on dit aussi la complexité de l'algorithme) est :
$n \times c = c \,.\, n$

Exemple 2
Soit l'algorithme suivant :

pour j de 1 à n faire	*n* fois
x ← j * 2	c_1
fpour	
pour i de 1 à n faire	*n* fois
pour j de 1 à n faire	*n* fois
x ← x + i*x+j	c_2
fpour	
fpour	

c_1 est le coût de l'instruction `(x ← j * 2)`
c_2 est le coût de l'instruction `(x ← x + i*x+j)`
La complexité de cet algorithme est : $n \times c_1 + n \times n \times c_2 = c_2 \,.\, n^2 + c_1.n$

3. Ordre de grandeur

En pratique, on ne s'intéresse pas à calculer exactement la complexité mais on se contente de calculer son ordre de grandeur voire de borner celui-ci. Pour l'exemple 2, la complexité exacte de l'algorithme est : $c_2.n^2 + c_1.n$, on ne retiendra pas les coefficients c1 et c2. En effet, pour des valeurs de *n* très grandes, l'influence des coefficients sur le coût final est négligeable. C'est pour cela qu'on dit que la complexité est indépendante de la machine. De plus, on ne retiendra que le monôme ayant la puissance la plus élevée. Pour l'algorithme de l'exemple 2, on retiendra que l'ordre de grandeur est n^2. On dit que l'algorithme a une complexité en $O(n^2)$·. L'algorithme de l'exemple 1 a une complexité en $O(n)$. Cette information nous permet de comparer les deux algorithmes et dire que l'algorithme 1 est plus rapide que l'algorithme 2 : plus la puissance du monôme est élevé plus la complexité de l'algorithme est importante.

Propriétés

- Si $a < b$, alors un algorithme qui est en $\boldsymbol{O(n^b)}$ a une complexité plus importante qu'un algorithme en $\boldsymbol{O(n^a)}$
- Les algorithmes ayant une complexité en ***log n*** sont **plus rapides** que les algorithmes ayant une complexité en $\boldsymbol{n^\alpha}$ ($\alpha > 0$).
- Les algorithmes ayant une complexité en $\boldsymbol{n^k}$ (pour tout k>0) sont **plus rapides** que les algorithmes ayant une complexité en $\boldsymbol{c^n}$(pour $c > 0$)

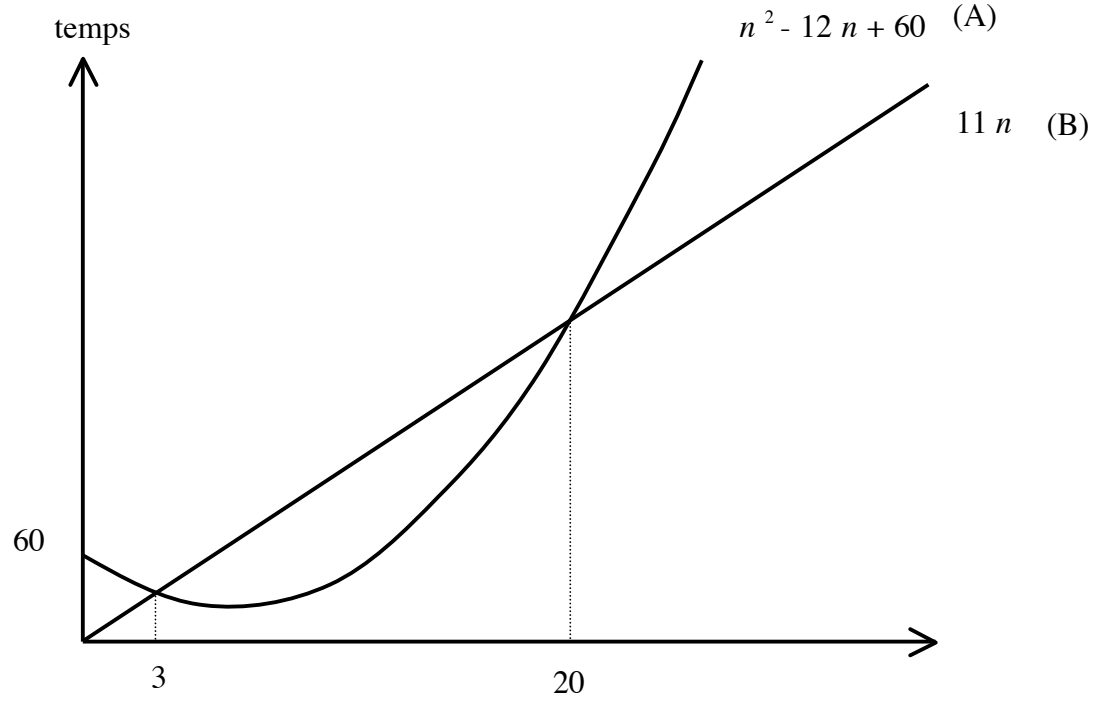
· On dit aussi que le coût de l'algorithme 2 est dominé par n^2.

Remarque

Il faut noter que ces propriétés ne sont valables que pour une valeur de n suffisamment grande. Dans la figure suivante, on présente graphiquement la complexité de deux algorithmes :

- Algorithme A de complexité $n^2 - 12\,n + 60$
- Algorithme B de complexité $11\,n$.

Pour les valeurs de n < 20 , l'algorithme A est plus performant que B. Mais pour toute valeur de n > 20, c'est l'algorithme B qui l'emporte. L'algorithme A est en $O(n^2)$ et l'algorithme B est en $O(n)$.



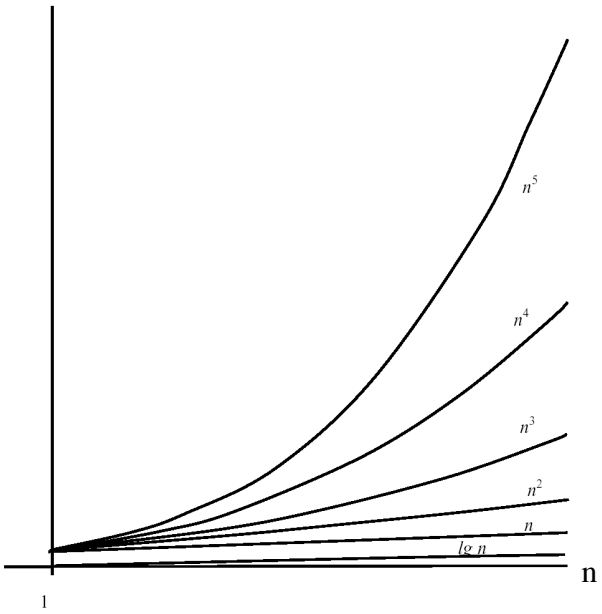
Rappel sur les logarithmes

On note la fonction logarithme par ***log***. Le logarithme de base b est noté $\boldsymbol{log_b}$ .On note souvent le logarithme **binaire** ou de base 2 par ***lg***. On distingue aussi le logarithme **naturel** ou **népérien** (*log* de base *e*) qu'on note par ***ln***. Pour exprimer un logarithme d'une base en fonction d'un logarithme d'une autre base, on utilise la formule suivante :

$$log_b(x) = log_a(x)/log_a(b)$$

Exemples de croissance de fonctions

Le graphe suivant montre différentes fonctions qu'on utilise souvent pour la complexité des algorithmes. L'axe horizontal représente la taille des données n et l'axe vertical le temps. Les algorithmes en $O(log\ n)$ sont les plus rapides. Ensuite, nous avons les algorithmes en $O(n)$. On dit d'un algorithme en $O(n)$ qu'il est *linéaire*. Un algorithme en $O(n^2)$ est dit *quadratique*. Plus généralement, un algorithme en $O(n^p)$ est dit *polynomial*. Enfin, un algorithme ayant une complexité en $O(c^n)$ est dit exponentiel.



Importance des ordres de grandeurs

Le tableau suivant montre l'importance des ordres de grandeurs. On présente 5 algorithmes avec la complexité de chacun exprimé en fonction de la taille des données *n*. Pour les grandes valeurs de *n* , on voit bien l'importance des ordres de grandeur caractérisant chaque algorithme. Un algorithme en *O(n)* est plus rapide qu'un algorithme en *O(n log n),* etc. Notez le cas particulier de l'algorithme 5 qui est un algorithme ayant une complexité exponentielle. Même avec une valeur de $n = 100$, relativement petite, il faut un temps astronomique pour terminer l'exécution de l'algorithme. C'est pour cela qu'il est fortement conseillé d'éviter les algorithmes exponentiels.

Algorithme	1	2	3	4	5
Fonction de temps (microsec.)	33 *n*	46 *n log n*	13 n^2	3.4 n^3	2^n
Taille des données (*n*)	Temps de résolution				
10	0.00033 sec	0.0015 sec	0.0013 sec	0.0034 sec	0.001 sec
100	0.003 sec	0.03 sec	0.13 sec	3.4 sec	$4 . 10^{16}$ années
1000	0.033 sec	.45 sec	13 sec	0.94 heures.	
10 000	0.33 sec	6.1 sec.	22 min.	39 jours	
100 000	3.3 sec	1.3 min	1.5 jours	108 années	

Le tableau suivant montre que l'ordre de grandeur est plus important que les coefficients. On compare deux algorithmes sur deux machines différentes (des années 70s) : (1) Cray-1 : (80MHz x 2) ; (2) TRS-80 : (1.77MHz). Le temps d'exécution d'une instruction de l'algorithme sur Cray-1 nécessite 3 nanosecondes et sur TRS-80, elle nécessite 19 500 000 nanosecondes !! On s'attend à ce que Cray-1 donne de meilleure performance vu sa rapidité par rapport à TRS-80. Néanmoins, ceci n'est pas vrai dès que la taille des données devient grande. En effet, dès que *n* dépasse 1000, l'algorithme exécuté sur TRS-80 qui a une complexité en *O(n)* dépasse rapidement les performances de l'algorithme exécuté sur Cray-1 qui a une complexité en $O(n^3)$. Cet exemple montre que les coefficients de la fonction coût d'un algorithme n'ont aucune influence sur la complexité d'un algorithme. L'exemple montre aussi que si un algorithme a une complexité importante, même si on l'exécute sur machine très rapide cela ne rendra pas l'algorithme plus rapide.

Taille des données	Cray-1	TRS-80
n	$3\,n^3$ nanosecondes	19 500 000 n nanosecondes
10	3 microsecondes	.2 secondes
100	3 millisecondes	2 secondes
1000	3 secondes	20 secondes
2 500	50 secondes	50 secondes
10 000	49 minutes	3.2 minutes
1 000 000	95 années	5.4 heures

Exercices

1- Calculer la complexité de l'algorithme de la fonction Factorielle (présenté dans l'exercice 1)
On suppose que le coût des instructions non récursives est 0.

2- Calculer la complexité de l'algorithme de résolution des tours de Hanoï. (présenté dans l'exercice 5)
On compte les déplacements comme opération élémentaire.

3- Calculer la complexité de l'algorithme de recherche dichotomique (présenté dans l'exercice 9).
On suppose que le coût des instructions non récursives est 1.

Chapitre 3

Première partie : Les arbres binaires

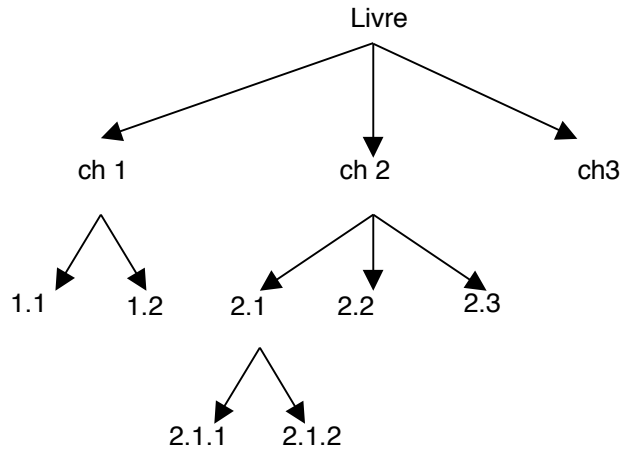
I - Présentation de la notion d'arbres binaires : niveau logique et opérations

1) Terminologie de base

Un **arbre** est un ensemble d'éléments appelés nœuds (l'un d'entre eux est appelé **racine**) liés par une relation induisant une **structure hiérarchique** parmi ces nœuds. A chaque nœud est associée une **valeur** de type quelconque.

Exemple d'arbre :
une table des matières et sa représentation arborescente

Livre
ch 1
1.1
1.2
ch 2
2.1
2.1.1
2.1.2
2.2
2.3
ch 3



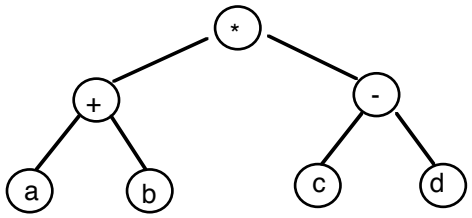
On peut définir un arbre comme formé d'un nœud appelé racine de l'arbre et d'un nombre fini d'arbres appelés **sous arbres** de la racine. Un nœud auquel n'est attaché aucun sous arbre est appelé **feuille**. Les racines des sous-barbes d'un nœud n sont appelées **fils** de n.

Les nœuds d'un arbre qui ne sont ni des feuilles ni la racine sont appelés **nœuds internes**. Les nœuds d'un arbre sont reliés entre eux 2 à 2 par des axes appelés branches de l'arbre. Chaque nœud excepté la racine admet un ascendant ou **père**. Les fils d'un même père sont dits des **frères**.

Si *n*1, *n*2,..., *n*k est une suite de nœuds d'un arbre telle que ni est le parent de ni + 1 pour 1< = i <= k, cette suite est appelée **chemin** entre le nœud ni et le nœud nk. La longueur d'un chemin est égale au nombre de nœuds qu'il contient moins un. S'il existe un chemin entre les nœuds a et b, on dit que a est un ascendant ou un ancêtre de b et réciproquement que b est un descendant de a. Un sous arbre d'un arbre est un nœud accompagné de toute sa descendance. La **hauteur d'un nœud** dans un arbre est la longueur du plus long chemin que l'on puisse mener entre ce nœud et une feuille de l'arbre. La **hauteur d'un arbre** est la hauteur de sa racine. La **profondeur** d'un nœud est la longueur du chemin entre la racine et ce nœud. Il est parfois utile d'inclure l'arbre vide parmi l'ensemble des arbres : c'est un arbre sans nœud.

Un **arbre est dit binaire** si chaque nœud a au plus 2 fils.

Exemple d'arbre binaire
L'expression arithmétique ((a + b) * (c - d)) peut être représenté par l'arbre binaire suivant :



Remarque : un nœud dans un arbre est comparable à une place dans une liste.

2) Définition formelle d'un arbre binaire

Soit V un ensemble de valeurs. On appelle **arbre binaire sur V** et on note arbin(V) l'ensemble des arbres binaires dont les valeurs sont des éléments de V. Baptisons ArbreBinaireV le type arbin (V).

Un arbre binaire sur V est défini par :

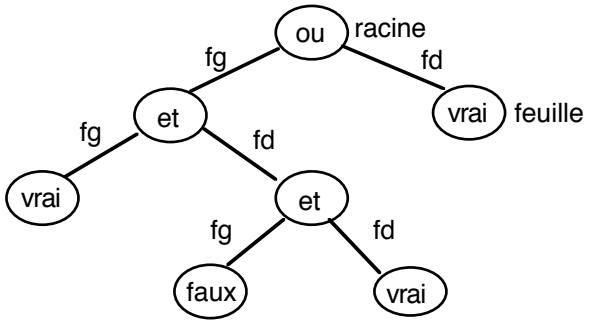
- **Nœud** : ensemble des nœuds, y compris *nil* qui est un nœud fictif

et les fonctions :

• **racine** :	ArbreBinaireV	→	Nœud
• **val** :	ArbreBinaireV x Nœud - { nil }	→	V
• **fg** :	ArbreBinaireV x Nœud - { nil }	→	Nœud
• **fd** :	ArbreBinaireV x Nœud - { nil }	→	Nœud
• **noeudvide** :	ArbreBinaireV x Nœud	→	booléen

La fonction *racine* donne le 1er nœud de l'arbre, la fonction *val* la valeur associée à un nœud, la fonction *fg* le nœud qui se trouve à gauche du nœud courant (le fils gauche), la fonction *fd* le nœud qui se trouve à droite du nœud courant (le fils droit). La fonction *noeudvide* rend vrai si le nœud est le nœud *nil*.

Exemple :



Les fils gauches et fils droits des feuilles sont nil.

3) Opérations de parcours

Les opérations de parcours sont : **racine, val, fg, fd, noeudvide.**

4) Opérations de construction

Les opérateurs de construction d'arbre binaire sont les suivants :

- **créerarb : V → ArbreBinaireV**

  création d'un arbre réduit à un seul nœud

- **adjfg : ArbreBinaireV x Nœud - { nil } x V →**

  adjfg (a,n,v) : adjonction d'un fils gauche au nœud n de l'arbre a

- **adjfd : ArbreBinaireV x Noeud - { nil } x V →**

  adjfd (a,n,v) : adjonction d'un fils droit au noeud n de l'arbre a

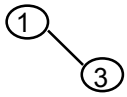
Les opérations *adjfg* et *adjfd* modifient l'arbre donné en paramètre.

Exemple

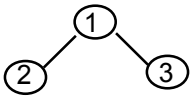
Soit ArbreBinaireEntier = arbin (entier), a : ArbreBinaireEntier, n : Nœud

a <- créerarb (1) donne l'arbre suivant :

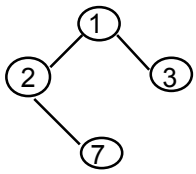
adjfd (a, racine (a), 3) donne



adjfg (a, racine (a), 2) donne



n <- fg (a, racine (a))
adjfd (a,n,7) donne



5) Opérations de mise à jour

Les opérations de mise à jour sont les suivantes :

- **chgarb : ArbreBinaireV x Nœud - { nil } x V →**

  chgarb (a,n,v) : changement de la valeur du noeud n de l'arbre a

- **adjfg : ArbreBinaireV x Nœud - { nil } x V →**
- **adjfd : ArbreBinaireV x Nœud - { nil } x V →**

  ces deux opérations sont à la fois des opérations de mise à jour et de construction

- **supfg : ArbreBinaireV x Nœud- { nil } →**

  supfg (a,n) : suppression du sous arbre dont la racine est le fils gauche du nœud n de l'arbre a

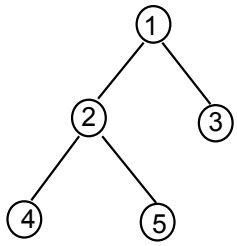
- **supfd : ArbreBinaireV x Nœud - { nil } →**

  supfd (a,n) : suppression du sous arbre dont la racine est le fils droit du nœud n de l'arbre a

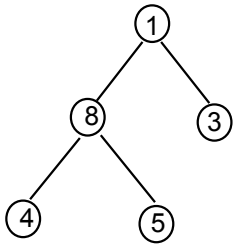
Les opérations chgarb, adjfg, adjfd, supfg et supfd modifient l'arbre donné en paramètre.

Exemple

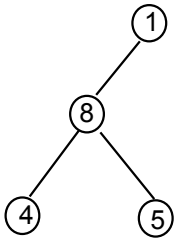
Soit a :



chgarb (a,fg (a, racine(a)),8) donne a :



supfd (a, racine(a)) donne a :



supfg (a, racine (a)) donne a :

6) Une structure de données récursive

Un arbre est une **structure de données récursive** car tout nœud de l'arbre peut être considéré comme la racine d'un autre arbre.

Toutes les fonctions utilisant, manipulant ou créant des arbres pourront avoir une définition récursive.

On dit qu'une fonction **f a une définition récursive** si sa définition algorithmique utilise un (ou plusieurs) appel(s) de f. Par abus de langage, on dit souvent de la fonction elle-même qu'elle est récursive.

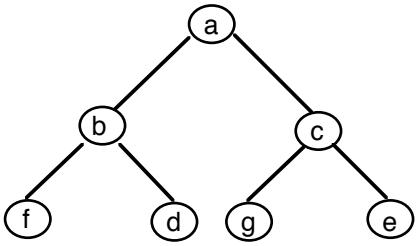
II - Parcours d'arbres

On peut parcourir un arbre de plusieurs façons :

- en ordre préfixé
- en ordre infixé (ou symétrique)
- en ordre postfixé

- en ordre préfixé : on traite d'abord la racine de l'arbre puis son fils gauche puis son fils droit
- en ordre infixé : on traite d'abord le fils gauche puis la racine puis le fils droit
- en ordre postfixé : on traite d'abord le fils gauche puis le fils droit puis la racine

Exemple 1 :
Soit l'arbre suivant



On souhaite imprimer les valeurs associées à chaque nœud :

en ordre **préfixé** : a b f d c g e

en ordre **infixé** : f b d a g c e

en ordre **postfixé** : f d b g e c a

Exemple 2 :
On souhaite écrire la fonction qui imprime en ordre préfixé les valeurs associées aux nœuds d'un arbre donné.

```
fonction imprimerPréfixé (arbre : ArbreBinaireChaîne, noeud : Noeud)
   début
        si non noeudvide (arbre,noeud) alors
                écrire(val (arbre, noeud))
                imprimerPréfixé (arbre, fg (arbre, noeud))
                imprimerPréfixé (arbre, fd (arbre, noeud))
        fsi
  fin
```

où ArbreBinaireChaîne = arbin (chaîne)

à dérouler sur l'exemple 1

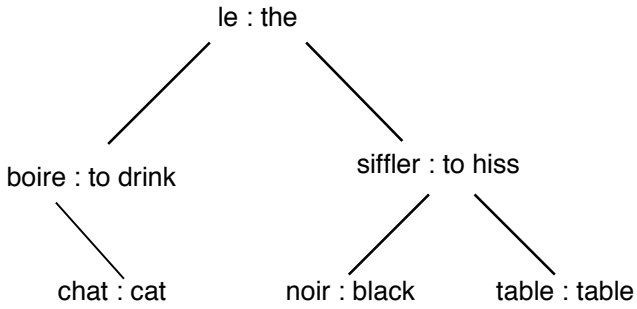
Remarque : le paramètre *nœud* est nécessaire pour pouvoir utiliser la récursivité, il est la racine du sous arbre en cours de traitement. A l'appel de la fonction, *nœud* est la racine de *arbre*.

Exercice :
Même question en ordre infixé puis en ordre postfixé

III - Exercice : algorithmes logiques sur les arbres

Un lexique français-anglais est un ensemble de couples de mots dont le 2ème est une traduction du 1er (traduction supposée unique). La représentation logique habituelle est une table. Mais comme la probabilité de rechercher un mot n'est pas uniforme, on peut préférer une représentation par un arbre binaire. Ceci permet de ranger les mots les plus utilisés près de la racine et d'optimiser ainsi le temps moyen d'accès aux mots. Les mots sont placés aux nœuds de telle façon que, pour tout nœud n de valeur v, on trouve à gauche de n tous les mots plus petits (d'après l'ordre lexicographique) que v et à droite de n tous les mots plus grands que v.

Exemple



Soit *Lexique* le type de cet arbre représentant un lexique :

Lexique = arbin (MotTraduction = < français (chaîne), anglais (chaîne) >)

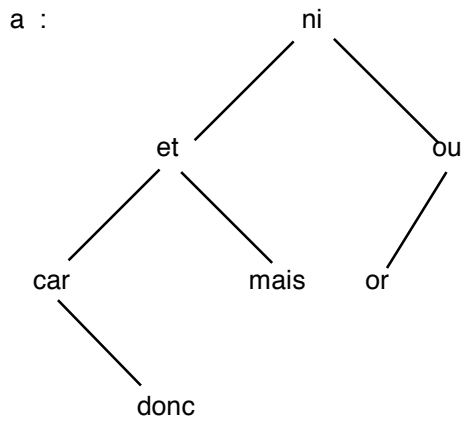
a) Ecrire une fonction qui donne la traduction d'un mot s'il existe dans le lexique ou la chaîne vide.
b) Ecrire une fonction qui permet d'insérer un mot et sa traduction dans le lexique. L'insertion se fera à un nœud qui n'a pas encore de fils gauche en cas d'adjonction à gauche ou pas de fils droit en cas d'adjonction à droite. On suppose que le mot à insérer n'existe pas déjà dans le lexique et que le lexique n'est pas vide. (Exemple : rouge, red)

IV - Représentation physique

1) Représentation chaînée d'un arbre binaire dans un tableau ou fichier direct

Nous présentons la représentation par tableau, celles des fichiers s'en déduisant par analogie.
L'arbre est représenté par un tableau de variables composites à 3 champs : val, fg et fd. Les nœuds sont représentés par des indices allant de 1 à nombre de nœud. L'indice 0 correspond au nœud vide. Par convention, nous avons choisi de mémoriser la racine de l'arbre dans le champ fg de l'élément d'indice 0.

Exemple



Représentation de a

	0	1	2	3	4	5	6	7
champ val		ni	et	car	donc	mais	ou	or
champ fg	1	2	3	0	0	0	7	0
champ fd		6	5	4	0	0	0	0

tableau a

Déclarations
a (tableau <val (Valeur), fg (entier), fd (entier)> [0..BS])
où Valeur est le type des éléments de l'arbre

Expression des fonctions logiques

racine (a) ⇔ a[0].fg

val (a, n) ⇔ a[n].val

fg (a, n) ⇔ a[n].fg

fd (a, n) ⇔ a[n].fd

noeudvide (a, n) ⇔ n = 0

a ← créerarb (v) ⇔
- a [1].val ← v
- a [1].fg ← 0
- a [1].fd ← 0
- a [0].fg ←1
- initialisation de l'espace libre

adjfg (a,n,v) ⇔
- recherche d'une place libre pl dans a
- a [pl].val ← v
- a [pl].fg ← 0
- a [pl].fd ← 0
- a [n].fg ← pl

adjfd (a,n,v) ⇔
- recherche d'une place libre pl dans a
- a[pl].val ← v
- a[pl].fg ← 0
- a[pl].fd ← 0
- a[n].fd ← pl

supfg (a,n) ⇔
- a [n].fg ← 0
- restitution de la place libérée

supfd (a,n) ⇔
- a [n].fd ← 0
- restitution de la place libérée

chgarb (a,n,v) ⇔
- a[n].val ← v

La gestion de l'espace libre peut se faire de manière similaire à celle proposée pour les listes dans le cas de la représentation d'une liste chaînée par tableau ou fichier direct.

2) Représentation à l'aide de pointeurs

Un nœud est un pointeur vers un élément de type composite à 3 champs : la valeur associée au nœud, un pointeur vers le fils gauche et un pointeur vers le fils droit. Un pointeur est à *nil* lorsqu'il représente un nœud vide. Un arbre est représenté par un pointeur qui est la racine.

Déclarations nécessaires en C pour un arbre a :

```
typedef struct t_ElemType  {
              Valeur val ;
              struct t_ElemType * fg ;
              struct t_ElemType * fd ;
              } ElemType ;
ElemType * a ;
```

où Valeur est le type des valeurs associées aux noeuds.

Expressions des fonctions logiques en C

racine (a) ⇔ a

val (a, n) ⇔ n→val

fg (a, n) ⇔ n→fg

fd (a, n) ⇔ n→fd

noeudvide (a, n) ⇔ n = = NULL

a ← créerarb (v) ⇔
- a = malloc (sizeof (ElemType)) ;
- a →val = v ;
- a →fd = NULL ;
- a →fg = NULL ;

adjfg (a,n,v) ⇔
- nouvn = malloc (sizeof (ElemType)) ;
- nouvn →val = v ;
- nouvn →fg = NULL ;
- nouvn →fd = NULL ;
- n →fg = nouvn ;

adjfd (a,n,v) ⇔
- nouvn = malloc (sizeof (ElemType)) ;
- nouvn →val = v ;
- nouvn →fg = NULL ;
- nouvn →fd = NULL ;
- n →fd = nouvn ;

supfg (a,n) ⇔
- n →fg = NULL ;
- restitution de la place libérée

supfd (a,n) ⇔
- n →fd = NULL ;
- restitution de la place libérée

chgarb (a,n,v) ⇔ n →val = v ;

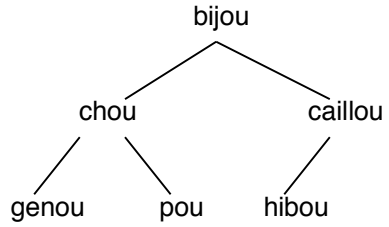
La restitution de la place libérée peut se faire par un appel à une fonction de libération de place. Cette fonction parcourerait le sous-arbre de racine fg(a, n) ou fd(a, n) pour libérer la place occupée par chaque noeud.

3) Représentation contiguë

Elle peut être réalisée dans un tableau ou un fichier direct. Nous présentons ici le cas des tableaux.
Les noeuds sont représentés par des indices, la fonction d'accès *val* par un tableau, les fonctions d'accès *fg* et *fd* par calcul. Les fils de l'élément d'indice i sont aux indices respectifs 2i et 2i + 1. La racine se trouve à l'indice 1.

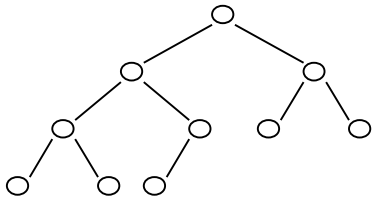
Exemple

a :



	1	2	3	4	5	6
a	bijou	chou	caillou	genou	pou	hibou

Cette représentation n'est intéressante que pour un arbre binaire "plein" ou presque "plein". Un arbre binaire est dit "plein" lorsque tous ses noeuds ont 2 fils sauf les noeuds du dernier niveau et éventuellement les noeuds les plus à droite de l'avant dernier niveau.
Exemple d'arbre binaire "plein" :



Pour un arbre binaire "plein", cette méthode est économe en place puisqu'elle n'exige que la place en mémoire nécessaire à la représentation des n valeurs.

Déclarations :
a [tableau Valeur [1..BS]]

Expressions des fonctions logiques :

racine (a) ⇔ a ← 1
val (a, n) ⇔ a [n]
fg (a, n) ⇔ 2 * n
fd (a, n) ⇔ 2 * n + 1
noeudvide (a, n) ⇔ a [n] = vnil
où vnil est une valeur particulière qui indique que le noeud n est vide.

a ← créerarb (v)	⇔	a [1] ← v pour i de 2 à BS faire a [i] ← vnil fpour
adjfg (a,n,v)	⇔	a [2n] ← v
adjfd (a,n,v)	⇔	a [2n+1] ← v
supfg (a,n)	⇔	a [2n] ← vnil
supfd (a,n)	⇔	a [2n+1] ← vnil
chgarb (a,n,v)	⇔	a [n] ← v

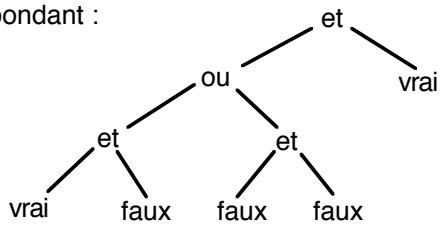
V – Exercice

Question 1

Ecrire une fonction qui évalue une expression booléenne (ne contenant que des opérateurs ET et OU) représentée sous forme d'arbre binaire. On suppose l'arbre non vide. Les valeurs associées à chaque nœud sont des chaînes de caractères.

Exemple d'expression :
(((vrai et faux) ou (faux et faux)) et vrai)
On suppose qu'il y a un blanc avant et après chaque parenthèse.

arbre correspondant :



Question 2

Ecrire une fonction qui crée l'arbre d'une expression booléenne (ne contenant que des opérateurs "et" et "ou") entièrement parenthèsée. L'expression est saisie au fur et à mesure.

On suppose disposer de la fonction suivante :

fonction enraciner (a1: arbin(V), v : V, a2: arbin(V)) : arbin(V)

qui, à partir de 2 arbres a1 et a2 et d'une valeur v, crée un arbre dont le nœud racine a pour valeur v, dont le sous arbre gauche est a1 et le sous arbre droit a2.

Question 3

Ecrire la fonction *enraciner* en utilisant et en écrivant les fonctions récursives *constfg* et *constfd* suivantes:

fonction constfg (a InOut: ArbreBinaire, na : Noeud, b : ArbreBinaire, nb : Noeud)
/* ajoute à l'arbre a à gauche du noeud na le sous-arbre de b débutant au noeud nb */

idem fonction constfd en remplaçant à gauche par à droite.

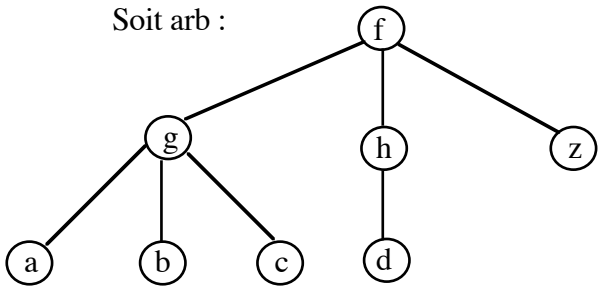
Remarque:
Si on choisit la représentation chaînée à l'aide de pointeurs, la fonction *enraciner* peut s'écrire plus simplement.

Deuxième partie : Les Arbres n-aires

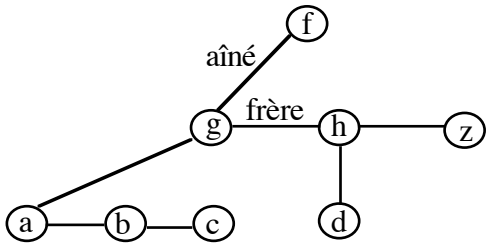
I - Définition et opérations

Dans un arbre n-aire, les nœuds ont un nombre quelconque de fils.

Exemple



Il est plus approprié de représenter cet arbre n-aire de la manière suivante :



On retrouve ainsi une représentation similaire à celle d'un arbre binaire. Pour un nœud, on a accès à son 1er fils et à son 1er frère.

Notons ArbreNaireV = arbnaire (V) le type arbre n-aire sur V.

Opérations sur les arbres n-aires :

racine :ArbreNaireV	→	Nœud	
val :	ArbreNaireV x Nœud - { nil }	→	V
aîné :	ArbreNaireV x Nœud - { nil }	→	Nœud
frère :	ArbreNaireV x Nœud - { nil }	→	Nœud
noeudvide :	ArbreNaireV x Nœud	→	booléen
créerarb :	V	→	ArbreNaireV
adjaîné:	ArbreNaireV x Nœud x V	→	
adjfrère :	ArbreNaireV x Nœud x V	→	
chgarb :	ArbreNaireV x Nœud x V	→	
supaîné :	ArbreNaireV x Nœud	→	
supfrère :	ArbreNaireV x Nœud	→	

Les opérations chgarb, adjaîné, adjfrère, supaîné et supfrère modifient l'arbre donné en paramètre.

II - Parcours

parcours préfixé : on traite d'abord un nœud puis ses fils

parcours postfixé : on traite d'abord les fils puis le nœud père

III - Exercice

Question 1

Ecrire une fonction imprimant les valeurs des nœuds d'un arbre n-aire en le parcourant de manière postfixée.

Exemple : imprimerPostfixé (arb, racine (arb)) donne a b c g d h z f

Question 2

Même question en le parcourant de manière préfixée.

IV - Représentation physique

Nous ne les détaillerons pas. Elles sont comparables à celles des arbres binaires et s'en déduisent facilement.