# Université de Strasbourg

## *UFR de Mathématique et d'Informatique*

## L2 Informatique
## Semestres S3 et S4

---

# Programmation I et II

---

## *Structures de Données et Algorithmes 1 et 2*

---

**Fiches d'exercices**

**année 2008-2009**

# 1. Algorithmique et programmation de base, notations

## *Constructions de base*

**Exercice 1** (*définitions séquentielles*)**.** Spécifier l'établissement d'une petite facture pour un seul type d'articles. On veut afficher le prix unitaire, la quantité, le prix brut, la remise, la TVA, le prix net, en ayant comme données le prix unitaire, la quantité, le taux de remise. Le taux de TVA est fixé à 19,6%. Ensuite, écrire en C un programme pour lire les données, faire les calculs et afficher les résultats avec des libellés clairs.
***Notations*** :
- définition simple : $x$ = *expr* **avec** $x_1 = expr_1, \ldots, x_n = expr_n$
  ou bien : $x$ = **soit** $x_1 = expr_1, \ldots, x_n = expr_n$ **dans** *expr*
- lecture en séquence sur organe standard : $(x_1, \ldots, x_n)$ = *donnée*
- écriture en séquence *par effet de bord* sur organe standard : *écrire*$(y_1, \ldots, y_n)$
- résultat général (fonction main() de C) : principal = $expr_1, \ldots, expr_n$

**Exercice 2** (*définitions conditionnelles*)**.** Spécifier la résolution complète de l'équation du second degré $ax^2 + bx + c = 0$, où les coefficients $a$, $b$ et $c$ sont des données flottantes, avec $a$ non nul. On distinguera selon la valeur du discriminant le cas réel, où l'on affiche les deux racines, et le cas complexe où l'on affiche la partie réelle et la partie imaginaire des racines. Ecrire en C un programme pour lire les données, faire les calculs et afficher les résultats avec des libellés clairs.
***Notations*** :
- définition conditionnelle : $x$ = **si** *cond* **alors** $expr_1$ **sinon** $expr_2$ **fsi**

**Exercice 3** (*définitions itératives simples, récurrence*)**.** Spécifier et programmer en C l'affichage de la factorielle de l'entier naturel $n$ entré au clavier.
***Notations*** :
- définition itérative simple : $x$ = **init** $x_0$ **pour** $i$ **de** $d$ **à** $f$ **pas** $p$ **répéter** *expr*[$x$, $i$] **frépéter**

**Exercice 4** (*définitions itératives simples*)**.** Spécifier et programmer en C l'établissement de la table de multiplication d'un entier naturel quelconque par les entiers de 1 à 10.

**Exercice 5** (*définitions itératives générales, récurrence*)**.** Spécifier et programmer en C l'affichage d'une approximation à $10^{-7}$ près de la racine carrée d'un nombre $a$ entré au clavier.
***Indication*** : utiliser la suite $u_0 = 1$, $u_n = 1/2(u_{n-1} + a/u_{n-1})$.
***Notations*** :
- définition itérative générale : $x$ = **init** $x_0$ **tantque** *cond*[$x$] **répéter** *expr*[$x$] **frépéter**

**Exercice 6** (*définitions itératives générales, récurrence, n-uplets*)**.** Spécifier et programmer en C l'affichage du plus petit entier tel que sa factorielle soit supérieure à un entier $M$ entré au clavier.
***Notations*** :
- définition itérative générale : $x$ = **init** $x_0$ **tantque** *cond*[$x$] **répéter** *expr*[$x$] **frépéter**
- n-*uplet* : $(x_1, \ldots, x_n)$

## *Fonctions et programmes*

**Exercice 7** (*spécifications itératives et programmation*)**.** Spécifier itérativement et écrire en C des fonctions pour calculer de manière approchée $exp(x)$, $sin(x)$ et $cos(x)$ pour $x$ flottant quelconque, et $ln(1 + x)$ pour $|x| < 1$.

**Exercice 8** (*récursivité et itération*)**.** Spécifier et écrire en C une fonction renvoyant $x^n$, pour $x$ flottant et $n$ entier naturel fournis en arguments. En produire deux spécifications récursives, l'une diminuant $n$ de 1 à chaque étape, l'autre le divisant par 2. Les transformer en spécifications itératives. Programmer en C les différentes versions.

**Exercice 9** (*récursivité double, suite double*)**.** Spécifier et écrire en C une fonction $u(n)$ renvoyant le $n$-ième terme de la suite de Fibonacci définie par $u_0 = 0$, $u_1 = 1$, $u_n = u_{n-1} + u_{n-2}$ pour $n > 1$. On en donnera deux versions, l'une récursive, l'autre itérative obtenue en introduisant une suite double. Définir et programmer l'affichage d'une table donnant, pour $n$ de 1 à 20, le rapport $u_n/\phi^n$, où $\phi = (1 + \sqrt{5}) / 2$ est le *nombre d'Or*.

**Exercice 10** (*résultats multiples*, *entrées-sorties*, *lecture à l'avance, suppression de la récursivité*)**.** Spécifier et écrire en C une fonction renvoyant la somme et le produit d'entiers relatifs entrés au clavier jusqu'à la frappe de l'entier -100, qui est exclu des calculs. En produire deux versions récursives, l'une non terminale et l'autre terminale, puis une version itérative. Définir et écrire en C une fonction qui calcule les résultats et les affiche au lieu de les renvoyer. ***Indication*** : utiliser une lecture à l'avance.

**Exercice 11** (*approximation numérique, passage de fonction en paramètre*)**.** Spécifier et écrire en C une fonction calculant dans l'intervalle $[a, b]$ une approximation du zéro d'une fonction $f$ dérivable strictement croissante telle que $f(a)f(b) < 0$. On s'appuiera sur une méthode classique : tangente (Newton), point fixe ou dichotomique.

**Exercice 12** (*intégration, passage de fonction en paramètre*)**.** Spécifier et écrire en C une fonction calculant une approximation par la *méthode des trapèzes* de l'intégrale de Riemann dans l'intervalle $[a, b]$ d'une fonction $f$ continue.

# 2. Généralités sur la spécification et la programmation

*N.B.* Contenu type d'un *dossier de programmation* : énoncé informel, spécification formelle, programme, jeu d'essai, résultats prévus, résultats obtenus, complexité, commentaires.

## *Spécification de types et d'opérations, programmation*

**Exercice 1** (*spécification algébrique et implantation des booléens*)**.** Spécifier algébriquement les *booléens* de sorte *Bool* et leurs opérations classiques : *vrai*, *faux*, *non*, *et*, *ou, ouex, egb* (égalité). En représentant en C *Bool* par `unsigned char`, montrer comment y programmer ces opérations.

**Exercice 2** (*spécification algébrique et implantation des entiers*)**.** En supposant connus les booléens, spécifier algébriquement les entiers naturels de sorte *Nat*, et leurs opérations classiques +, *, - (renvoyant 0 quand la fonction n'est habituellement pas définie), *egn* (égalité), <, <=, *pair*, *impair*, à partir des constructeurs 0 et *S*, rendant le successeur d'un entier. En représentant en C *Nat* par `unsigned int`, montrer comment se transposent les opérations ci-dessus.

**Exercice 3** (*spécification algébrique et implantation d'un stock, preuve de correction d'implantation*)**.** Spécifier algébriquement une petite *gestion de stock* d'au maximum 1000 produits, dont les références vont de 1 à 1000, avec une sorte *Stock* et les opérations suivantes : création et initialisation de tout le stock, entrée d'une nouvelle référence de produit avec sa quantité, suppression d'une référence, entrée ou sortie d'une quantité d'un produit, réajustement de la quantité d'un produit, interrogation du stock. On écrira soigneusement les *préconditions*. Faire une représentation en C de la sorte *Stock* par un type de pointeurs sur un entier relatif pour allouer dynamiquement un tableau, puis programmer les opérations *avec mutations* et les offrir dans un menu. Prouver la correction totale de l'implantation par rapport à la spécification.

**Exercice 4** (*spécification et programmation des tableaux*, *effets de bord*)**.** La sorte *S* et l'entier *n* > 0 étant considérés comme des paramètres globaux, spécifier algébriquement les tableaux contenant en permanence *n* objets de sorte *S*, indicés par les entiers de 0 à *n*-1, avec les opérations *gent*(*a*) pour engendrer un tableau dont chaque case contient la même constante *a*, *modt*(*t, i, a*) pour remplacer par *a* le contenu de la case *i* du tableau *t*, *élém*(*t, i*) pour accéder à la valeur rangée à la case *i* de *t*, encore notée *t*[*i*], et *éch*(*t, i, j*) pour échanger les valeurs *t*[*i*] et *t*[*j*]. Proposer en C une implantation de tels tableaux et programmer les opérations avec mutations. Proposer ensuite une spécification et une représentation où *n* peut varier selon le tableau.

**Exercice 5** (*drapeau hollandais*)**.** Spécifier à l'aide des opérations de l'Exercice 4 une fonction résolvant avec un tableau de caractères 'B', 'W' ou 'R' en paramètre le problème du *drapeau hollandais*. Il s'agit de transformer le tableau initial en un tableau trié dans l'ordre 'B', 'W', 'R', en effectuant un seul parcours de gauche à droite, avec, à chaque étape, au maximum un échange de deux valeurs. Programmer cette opération comme une fonction C effectuant une mutation. Définir et programmer des opérations de lecture *lect* et d'affichage *afft* d'un tel tableau, ainsi qu'une opération *main* pour appeler et tester l'opération du drapeau hollandais.

**Exercice 6** (*récursivité et itération*, *paramètres passés par adresse*)**.** Définir formellement avec les opérations de l'Exercice 4 et écrire en C une fonction renvoyant le *minimum* et le *maximum* des valeurs d'un tableau d'entiers naturels. On veut une version séquentielle et une autre dichotomique. Les spécifier formellement et les programmer récursivement puis itérativement.

**Exercice 7** (*tri de tableau par sélection simple*)**.** Définir à l'aide des opérations de l'Exercice 4 et écrire en C une fonction triant en croissant « sur place » un tableau d'entiers naturels par *sélection simple*. Dans cette méthode, à chaque étape, lors d'un parcours de gauche à droite de la portion de tableau restant à trier, on sélectionne un plus petit élément que l'on place à la première place libre du tableau grâce à un échange. On pourra définir une fonction donnant la place du minimum sélectionné à chaque étape.

**Exercice 8** (*spécification algébrique et implantation des nombres complexes*)**.** Spécifier algébriquement et programmer en C les *nombres complexes* sur les flottants avec leurs opérations classiques. Implanter en coordonnées cartésiennes, puis en coordonnées polaires.

**Exercice 9** (*ensembles finis représentés par des entiers*)**.**
**a.** L'entier naturel *n* étant un paramètre global (<= 32), spécifier une bibliothèque de base pour gérer les (sous-)ensembles finis de sorte *Ens* d'entiers pris dans [0, n-1], avec les opérations classiques : *ev*, ensemble vide, *i*, insertion, *d*, effacement, *dans*, appartenance, *uni*, union, *inter*, intersection, *diff*, différence, *incl*, inclusion, *ege* : égalité. Spécifier aussi une opération d'affichage *affe* du contenu d'un ensemble.
**b.** Implanter et programmer en C cette bibliothèque en représentant tout ensemble juste par un entier naturel sans signe sur 32 bits. Programmer les opérations ci-dessus *sans mutation*, en utilisant des opérations arithmétiques (division par 2, modulo 2…) avec récursivité. Prévoir pour les tests des opérations auxiliaires.
**c.** Avec la même représentation, reprendre la programmation des opérations en utilisant uniquement des décalages dans des chaînes de bits, des opérations logiques et des filtres binaires ad hoc.

**Exercice 10** (*gestion des multiensembles finis*)**.** Spécifier une bibliothèque de base pour gérer les multiensembles finis de caractères ASCII avec les opérations classiques sur les multiensembles. Implanter et programmer en C cette bibliothèque avec une représentation contiguë des multiensembles. On suppose connues les opérations usuelles sur les caractères. Pour chaque opération, choisir et discuter une réalisation avec ou sans mutations.

## 3. Piles, files, listes

**Exercice 1** (*implantation des piles d'entiers*)**.** Reprendre la spécification algébrique des *piles d'entiers*, de hauteur bornée par le paramètre *n*, avec leurs opérations classiques. Les implanter en C de manière contiguë, puis de manière chaînée.

**Exercice 2** (*implantation de piles de booléens*)**.** Spécifier algébriquement les *piles de booléens*, de hauteur bornée par 31, avec leurs opérations classiques. Les implanter en C sous la forme d'un simple entier naturel et programmer de manière fonctionnelle pure les opérations dans cette représentation. Prouver que cette implantation est correcte totalement.

**Exercice 3** (*implantation des piles par blocs*)**.** Spécifier algébriquement et programmer en C les piles d'entiers naturels non bornées avec leurs opérations classiques. Implanter ces piles de manière contiguë, et programmer avec mutations, avec allocation et désallocation dynamiques de blocs de taille fixe quand c'est nécessaire.

**Exercice 4** (*gestion d'un couple de piles*)**.** Spécifier la gestion d'un couple de piles d'entiers dont la somme des hauteurs est globalement bornée par le paramètre global *N*. On aura intérêt à spécifier directement les couples sans réutiliser la spécification connue des piles. Programmer en C avec une implantation contiguë et avec mutations.

**Exercice 5** (*évaluation d'expressions arithmétiques avec deux piles*)**.** Spécifier, puis programmer en C une fonction pour évaluer, en calculs sur les entiers, une expression arithmétique. Celle-ci se présente sous la forme d'une suite d'entiers naturels sur un seul chiffre décimal, d'opérateurs binaires parmi +, -, *, /, et de parenthèses ( et ). La suite est supposée correctement et complètement parenthésée. Tous les caractères sont entrés successivement au clavier. On définira cette évaluation de manière récursive sur les entrées, puis itérative, en utilisant une pile pour les opérateurs et une autre pour les opérandes, mais on ne programmera qu'une seule version. On suppose connues les opérations usuelles sur les caractères.

**Exercice 6** (*évaluation d'expressions arithmétiques avec une seule pile*)**.** Reprendre l'Exercice 5, avec une seule pile contenant opérandes et opérateurs, codés de manière convenable.

**Exercice 7** (*gestion de files PAPS*)**.** Spécifier une bibliothèque de base pour gérer des files d'éléments de sorte *S* en paramètre avec une politique premier arrivé-premier servi. On traitera le cas des files de taille bornée et celui des files de taille non bornée. Programmer en C avec mutations dans une implantation contiguë pour les files bornées, et chaînée avec simple pointeur sur la queue pour les files non bornées.

**Exercice 8** (*files PAPS circulaires, Flavius Josèphe*)**.** Spécifier une « boîte à outils » sur les files PAPS d'entiers naturels exactement adaptée à la résolution du problème de la *légende de Flavius Josèphe*, en mettant en évidence des décalages circulaires à gauche d'une position, puis de *p* positions. Représenter ces files de manière simplement chaînée et circulaire, puis de manière contiguë. Programmer successivement dans ces deux représentations en C avec mutations.

**Exercice 9** (*spécification de la suppression d'éléments d'une liste*)**.** A partir de la spécification de base des listes d'entiers naturels, spécifier sans précondition les opérations de suppression de *tous*

les éléments d'une liste vérifiant une propriété exprimée par une fonction à résultat booléen passée en paramètre. Idem pour la suppression et le remplacement par un entier donné en paramètre. Implanter les listes de manière chaînée et programmer les opérations ci-dessus *avec modifications* de manière récursive puis itérative.

**Exercice 10** (*implantation des listes par chaînage*)**.** Implanter les listes d'entiers naturels de manière simplement chaînée avec des cellules à deux champs (valeur et pointeur sur le successeur), et programmer avec mutations en C, successivement avec deux variantes, selon qu'on implante la liste vide par un simple pointeur de valeur `NULL` ou bien par un pointeur sur une cellule contenant un entier quelconque (par exemple la longueur de la liste) et un pointeur de valeur `NULL`.

**Exercice 11** (*gestion de listes triées*)**.** Spécifier une bibliothèque pour les *listes d'entiers naturels triées* de manière croissante, avec entre autres des opérations de lecture et d'écriture de listes, une opération d'interclassement de deux listes triées, puis de $n$ listes triées, considérées comme un tableau (Fiche 2, Exercice 4) de $n$ listes. Implanter le tout de manière simplement chaînée et programmer en C de manière récursive.

**Exercice 12** (*spécification algébrique et implantation des chaînes*)**.** En supposant connues les opérations sur les caractères, spécifier algébriquement les *chaînes de caractères* de longueur maximale $N > 0$ (paramètre global) avec les opérations : chaîne vide, adjonction en tête, tête, longueur, corps, $k$-ième caractère, modification du $k$-ième caractère, adjonction en queue, queue, corps inverse, sous-chaîne, concaténation, égalité, comparaison en ordre lexicographique, lecture, écriture, copie. En implantant les chaînes comme des tableaux, écrire en C les fonctions correspondantes avec et sans mutation, ainsi qu'une opération de désallocation de chaîne. Reprendre cette programmation avec une implantation des chaînes de manière chaînée, chaque chaînon contenant un seul caractère.

**Exercice 13** (*fonctions sur les chaînes*)**.** Utiliser les opérations de l'Exercice 12 pour spécifier et écrire en C une fonction donnant le *miroir* d'une chaîne, selon deux versions récursives, dont une récursive terminale que l'on transformera en une version itérative. Définir et programmer une fonction testant si une chaîne est un *palindrome*, une fonction engendrant toutes les *permutations* d'une chaîne et une fonction testant si une chaîne est un *anagramme* d'une autre chaîne.

**Exercice 14** (*gestion des ensembles finis*)**.** Soit en paramètres une sorte $S$ quelconque munie d'une égalité et d'un ordre total <=. Spécifier une bibliothèque de base pour gérer les ensembles finis d'objets de sorte $S$ contenant les opérations classiques sur les ensembles. Programmer en C cette bibliothèque avec une *représentation chaînée* des ensembles où leurs éléments sont ordonnés. Pour chaque opération, choisir et discuter une réalisation avec ou sans mutations.

# 4. Notions de complexité

**Exercice 1.** Dire ce que définissent les fonctions suivantes sur Nat, puis calculer leur complexité, pour rec1 en nombre de multiplications par 2 et pour rec2 en nombre d'additions :

rec1(n) = **si** n ==0 **alors** 1 **sinon** 2*rec1(n-1) **fsi**

rec2(n) = **si** n ==0 **alors** 1 **sinon** rec2(n-1) + rec2(n-1) **fsi**

**Exercice 2.** Evaluer en nombre de comparaisons, puis en nombre d'échanges, la complexité du programme du drapeau hollandais (Fiche 2, Exercice 5), puis de celui du tri par sélection simple (Fiche 2, Exercice 7) en fonction du nombre *n* d'éléments à trier.

**Exercice 3.** Evaluer la complexité de la procédure `proc` en nombre d'appels de `inst()` :

```
void proc(int n)
{
   int i, j, k;
   for (i = 1; i <= n - 1; i++)
      for (j = i + 1; j <= n; j++)
         for (k = 1; k <= j; k++) inst();
}
```

**Exercice 4** (*calcul d'une puissance*)**.** Donner une forme logarithmique simple du nombre de bits de la représentation binaire minimale d'un entier naturel *n*. Evaluer en nombre de multiplications la complexité dans le pire des cas des 2 versions récursives du calcul de $x^n$ (Fiche 1, Exercice 8).

**Exercice 5** (*suite de Fibonacci*)**.** Etudier la complexité des deux versions du calcul du terme général $u_n$ d'une suite de Fibonacci (Fiche 1, Exercice 9) en nombre d'additions. Pour la version naïve, on pourra minorer par un nombre connu.

**Exercice 6.** (*minimum et maximum*)**.** Evaluer et comparer en nombre de comparaisons d'éléments de tableaux la complexité des deux versions de l'Exercice 6 de la Fiche 2.

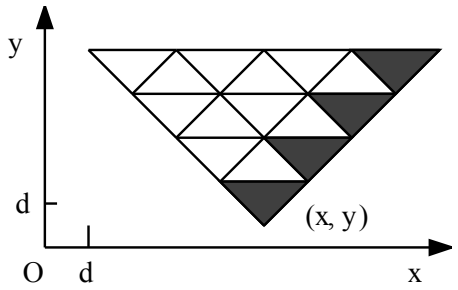
**Exercice 7** (*dessins*)**.** On travaille dans un repère euclidien du plan. On suppose connues les primitives `placer(x, y)` et `tracer(x, y)`, qui placent le crayon au point `(x, y)`, la première sans tracer et la seconde en traçant le segment joignant le point courant au point `(x, y)`.

**a.** Soit la procédure récursive suivante, écrite en C :

```
void triangles1(float x, float y, float d, float h)
{
   if (y + d < h)
   {
      tracer(x - d, y + d);
      triangles1(x - d, y + d, d, h);
      tracer(x + d, y + d);
      triangles1(x + d, y + d, d, h);
      tracer(x, y);
   }
```

```
}
```

La séquence `placer(x, y); triangles1(x, y, d, h)`, où `d > 0` et `h > 0`, trace la figure suivante (ne pas tenir compte pour l'instant du contenu grisé de certains triangles) :




**a.1.** On note $T1(y)$ le nombre d'exécutions de la primitive `tracer` pour l'appel `triangles1(x, y, d, h)` avec `d > 0`. Evaluer $T1(y)$ en fonction de $T1(y + d)$.
**a.2.** Soit $k$ tel que $y + k.d < h \leq y + (k + 1).d$. On pose $ti = T1(y + i.d)$, pour $i = 0, ..., k$. Montrer que $ti = 3.(2^{k-i} - 1)$, pour $i = 0, ..., k$.
**a.3.** En déduire la valeur de $T1(y)$ en fonction de $k$. Comment expliquer ce mauvais résultat ?
**b.** On cherche à améliorer l'efficacité de la procédure précédente.
**b.1.** Ecrire en C une procédure `bande(x, y, d, h)` traçant à partir du point `(x, y)` les triangles gris dans la figure et ramenant le crayon en ce point.
**b.2.** Ecrire ensuite une procédure récursive `triangles2(x, y, d, h)`, qui, en utilisant la procédure `bande`, trace chaque triangle de la figure une et une seule fois,.
**b.3.** On note $B(y)$ et $T2(y)$ les nombres d'exécutions de `tracer` pour les appels `bande(x, y, d, h)` et `triangles2(x, y, d, h)`, avec `d > 0`. Soit $k$ tel que $y + k.d < h \leq y + (k + 1).d$. On pose $bi = B(y + i.d)$ et $ti = T2(y + i.d)$, pour $i = 0, ..., k$. Evaluer $bi$ et $ti$, pour $i = 0, ..., k$.
**b.4.** En déduire $T2(y)$ en fonction de $k$. Comparer avec $T1(y)$. En dénombrant les segments élémentaires, montrer que la complexité $T2(y)$ est *optimale* en nombre de tracés de segments.

**Exercice 8.** Soient $f$ et $g$ deux fonctions des entiers naturels dans les réels strictement positifs.
**a.** Montrer que la relation binaire $f \in O(g)$ est réflexive et transitive. Montrer que la relation binaire $f \in \Theta(g)$ est une relation d'équivalence.
**b.** Montrer que $f \in O(g)$ implique $a.f \in O(g)$ et que $f \in \Theta(g)$ implique $a.f \in \Theta(g)$, où $a$ est un réel strictement positif quelconque.

**Exercice 9.** Soient $f1$, $f2$, $g1$ et $g2$ des fonctions des entiers naturels dans les réels positifs.
**a.** Montrer que $f1 + f2 \in O(\max(g1, g2))$ et $f1 * f2 \in O(g1 * g2)$ si $f1 \in O(g1)$ et $f2 \in O(g2)$.
**b.** Montrer que $f1 - f2 \in O(f1)$ si $f1 - f2 \geq 0$.
**c.** Montrer que $f1 + f2 \in \Theta(\max(g1, g2))$ et $f1 * f2 \in \Theta(g1 * g2)$ si $f1 \in \Theta(g1)$ et $f2 \in \Theta(g2)$.

**Exercice 10.** Soient $f$ et $g$ deux fonctions des entiers dans les réels strictement positifs.
**a.** Montrer que $\lim_{n\to\infty} f(n)/g(n) = a \neq 0$ implique $f \in \Theta(g)$.
**b.** Montrer que $\lim_{n\to\infty} f(n)/g(n) = 0$ implique $f \in O(g)$ et $f$ n'est pas en $\Theta(g)$.
**c.** Montrer que $\lim_{n\to\infty} f(n)/g(n) = \infty$ implique $g \in O(f)$ et $f$ n'est pas en $\Theta(g)$.

# 5. Arbres : spécification et programmation

**Exercice 1** (*arbres binaires : implantation chaînée*)**.** Reprendre la spécification des *arbres binaires étiquetés* avec toutes les opérations de base, et les implanter par chaînage en C, comme dans le cours, avec comme étiquettes des entiers naturels. Définir et programmer également l'affichage d'un arbre sous forme de terme.

**Exercice 2** (*arbres binomiaux*)**.** Les *arbres binomiaux* sont des arbres généraux étiquetés ordonnés définis inductivement comme suit (noter qu'il n'y a pas d'arbre vide) :

- un arbre réduit à un seul nœud est binomial de hauteur nulle ;
- un arbre binomial $A$ de hauteur $p > 0$ est construit, à partir de deux sous-arbres binomiaux $A1$ et $A2$ de hauteur $p$ - 1, à l'aide d'un lien tel que $A1$ est le premier fils de $A$, et $A2$ un arbre dont l'étiquette à la racine est celle de $A$ et dont les fils sont les fils de $A$ sauf le premier.

Par convention, la profondeur d'un nœud dans un tel arbre est égale au *nombre de liens* entre la racine et ce nœud, et la hauteur d'un arbre binomial est la profondeur maximale de ses nœuds.

**a.** Dessiner des arbres binomiaux non étiquetés, de la hauteur 0 à la hauteur 4.

**b.** Spécifier algébriquement la sorte Arbinom des arbres binomiaux étiquetés par la sorte S, avec les opérations f, créant un arbre réduit à une feuille, l, liant deux arbres binomiaux selon le principe ci-dessus, r, donnant l'étiquette à la racine d'un arbre, ef testant si un arbre est une feuille, pf et af, donnant respectivement le premier fils et l'arbre des autres fils d'un arbre binomial, h donnant la hauteur d'un arbre.

**c.** Spécifier, par rapport à f et l, les opérations nn, nf, ni et nh donnant le nombre de nœuds, de feuilles, de nœuds internes, et de nœuds à une profondeur donnée d'un arbre binomial.

**d.** Démontrer par induction que, lorsque A est de hauteur p, son nombre de nœuds est $2^p$, et son nombre de nœuds à la profondeur k, $0 \leq k \leq p$, est un coefficient du binôme de Newton. Définir simplement nn et nh en fonction de h.

**e.** A l'aide d'un schéma, expliquer comment transformer un arbre binomial en arbre binaire étiqueté. Proposer alors une définition des arbres binomiaux de la question (b) utilisant la spécification des arbres binaires étiquetés, en exprimant les générateurs d'arbres binomiaux en fonction des générateurs d'arbres binaires.

**f.** Implanter les arbres binomiaux étiquetés par Nat en C, et programmer les opérations ci-dessus, en reprenant l'implantation des arbres binaires étiquetés.

**Exercice 3** (*arbres quadratiques*)**.** On suppose connue une spécification algébrique des booléens de sorte Bool, notés 1 ou 0, avec leurs opérations habituelles.

**a.** Spécifier algébriquement les *arbres quadratiques* non bornés de sorte Arbq, étiquetés à chaque feuille par un booléen et non étiquetés aux autres noeuds, avec les fonctions f, renvoyant un arbre réduit à une feuille, e, enracinant 4 arbres quadratiques, ne, no, so, se, renvoyant respectivement les 4 sous-arbres, h, la hauteur d'un arbre, c, l'étiquette (1 ou 0) d'une feuille, et estf, testant la réduction d'un arbre à une feuille.

**b.** Pour $n = 2^k$ fixé, une image carrée noir et blanc de $n \times n$ pixels peut être représentée par un arbre quadratique étiqueté aux feuilles par 1, pour noir, ou 0, pour blanc. Un tel arbre représentant une image est défini récursivement de la manière suivante :

- si $k = 0$ ou si l'image est complètement noire (resp. blanche), l'arbre est réduit à une feuille noire (resp. blanche) ;

- sinon, l'image est partagée en 4 images carrées de $2^{k-1} \times 2^{k-1}$ pixels chacune et l'arbre possède 4 fils, appelés respectivement *ne*, *no*, *so* et *se*, pour rappeler les 4 directions à partir du centre.

Donner un exemple d'image et d'arbre correspondant avec $n = 4$. Pour $n$ donné en paramètre général, on dira qu'un arbre quadratique est *normal* s'il correspond bien à cette définition, c'est-à-dire s'il est de hauteur bornée (borne à déterminer) et si aucun de ses sous-arbres ne possède 4 fils réduits à des feuilles de même couleur.

Spécifier algébriquement une fonction estn testant si un arbre quadratique est normal, et une opération normal de conversion d'un arbre quadratique de hauteur bornée en arbre normal.

**c.** En combinant si nécessaire les pixels par les opérations booléennes, spécifier sur les arbres quadratiques normaux des fonctions égalité, union, intersection, inclusion des parties noires, correspondant à des opérations sur les images.

**d.** Dans une image $n \times n$, on suppose qu'un pixel est repéré par des coordonnées entières $(x, y)$ où $0 \leq x, y \leq n - 1$, avec comme origine (0, 0), le pixel en bas à gauche (au sud-ouest). Définir récursivement une opération noir permettant de savoir à partir de l'arbre quadratique normal d'une image si un pixel de coordonnées données est noir. Définir ensuite une opération colorer permettant de noircir ou blanchir un pixel de coordonnées données.

**e.** Proposer une implantation chaînée pour les arbres quadratiques, et la décrire en C. Programmer en C avec mutation les opérations de la question (a) ainsi que conv, union et colorer.

**Exercice 4** (*gestion de faune*)**.** Une *faune* est un arbre binaire dont les feuilles contiennent des noms d'animaux et leurs caractères, tandis que les nœuds internes contiennent des questions, à réponse par oui ou par non, permettant de scinder en deux parties l'ensemble des animaux. Ainsi, pour chaque question, le sous-arbre gauche correspond à la réponse positive et le sous-arbre droit à la réponse négative.

**a.** Spécifier algébriquement des faunes vues comme des arbres binaires, avec des fonctions élémentaires, puis de haut niveau, pour la gestion et l'interrogation interactives. L'opération essentielle est la *recherche-insertion* d'un animal $x$ : si la suite des réponses fournies par un utilisateur aux questions de la machine conduit à un animal $y$ qui, manifestement, n'est pas $x$, il faut introduire $x$ dans la faune à une feuille, avec un caractère discriminant entre $x$ et $y$, et la question correspondante.

**b.** Concevoir une opération interactive pour détruire une sous- arborescence à partir d'un certain nœud, et la remplacer par une autre qui serait déjà construite.

**c.** Concevoir une hiérarchie de menus pour appeler interactivement les différentes fonctions offertes à un utilisateur, ainsi que le programme de guidage correspondant.

**d.** Programmer en C les différentes fonctions et le programme de guidage.