8 Les listes

Listes et tableaux : structures linéaires contenant des éléments dans un certain ordre (avec occurrences multiples possibles).
Différences : (a) accès direct, en $O(1)$, pour les tableaux (*random access*) et en $O(n)$ pour les listes ; (b) structure figée pour les tableaux et flexibles pour les listes. Analogie CD *versus* bande magnétique.
La structure de liste généralise la structure d'entiers-bâtons : à la place des bâtons (tous égaux) on a des éléments (quelconques).
La notation $(a\ b\ c)$ représente une liste de trois éléments, dont a est le premier.

Type abstrait. Nom : `liste`.
Types abstraits importés : `booléen`, `typélt`.
Opérations primitives.
Les constructeurs sont :
`l_vide` : $\longrightarrow$ `liste`
`cons` : `typélt` $\times$ `liste` $\longrightarrow$ `liste`
Les accès sont :
`est_vide` : `liste` $\longrightarrow$ `booléen`
`prem` : `liste` $\longrightarrow$ `typélt`
`reste` : `liste` $\longrightarrow$ `liste`

Axiomes :

[Ax-1] `est_vide`(`l_vide`) = `vrai`

[Ax-2] `est_vide`(`cons`(x, L)) = `faux`

[Ax-3] `prem`(`l_vide`) = `Erreur_typélt`, où `Erreur_typélt` est une constante de type `typélt`.

[Ax-4] `prem`(`cons`(x, L)) $= x$

[Ax-5] `reste`(`l_vide`) = `l_vide`

[Ax-6] `reste`(`cons`(x, L)) $= L$

Implantation par listes chaînées. Fait appel à un enregistrement à deux champs : un champ pour `prem`, un champ pour `reste`.

Exercice 1 *Donner une implantation en C par listes chaînées du type liste de* `TYPELT`. `TYPELT` *est défini par* `#define TYPELT int` *s'il s'agit du type liste d'entiers.*

Exercice 2 *Pour chacune des opérations suivantes, on demande de suivre la démarche algorithmique définie au chapitre 4, à savoir, pour rappel : (1) définir le profil, (2) traiter un ou plusieurs exemples, (3) définir des axiomes, (4) traduire ces axiomes en un algorithme récursif, (5) donner un algorithme itératif, (6) traduire l'algorithme récursif en fonction C, (7) traduire l'algorithme itératif en fonction C, (8) mettre au point un test de ces fonctions :*

Q1 `longueur`(L) *donne le nombre d'éléments de la liste* L.

Q2 `appartient`(x, L) *teste si une valeur* x *de type* `typélt` *appartient à la liste* L.

Q3 `nème_élément`(n, L) *donne le* $n^{ème}$ *élément de* L. *Exemple :*
`nème_élément`$(3, (a\ b\ a\ c)) = a$ *et* `nème_élément`$(8, (a\ b\ a\ c))$ = `Erreur_typélt`.

Q4 `nème_reste`(n, L) *donne la liste* `reste`$^n(L)$. *Exemple :*
`nème_reste`$(2, (a\ b\ a\ c)) = (a\ c)$ *et* `nème_reste`$(8, (a\ b\ a\ c)) = ()$.

Q5 `insérer_élément`(x, n, L) *est la liste obtenue en insérant dans* L *le* `typélt` x *à la* $n^{ème}$ *place. Exemple :*
`insérer_élément`$(z, 2, (a\ b\ c)) = (a\ z\ b\ c)$.

Q6 `supprimer_élément`(x, L) *est la liste obtenue en supprimant toutes les occurrences de* x *dans* L. *Exemple :*
`supprimer_élément`$(a, (a\ b\ a\ a\ c\ a)) = (b\ c)$.

Q7 `nb_occurrences`(x, L) *retourne le nombre d'occurrences de* x *dans* L. *Exemple :*
`nb_occurrences`$(b, (a\ b\ b\ c\ b)) = 3$.

Q8 `égales`(L_1, L_2) *teste si* L_1 *est égal à* L_2. *Exemple :*
`égales`$((a\ b\ c), (a\ c\ b))$ = `faux` *et* `égales`$((a\ b\ c), (a\ b\ c))$ = `vrai`.

Q9 `dernier`(L) *est le dernier élément de la liste non vide* L *(donne une erreur si* L *est vide). Exemple :*
`dernier`$((a\ b\ c)) = c$.

Q10 `renverser`(L) *est la liste obtenue en renversant dans* L *la liste de de ses éléments. Exemple :*
`renverser`$((a\ b\ c)) = (c\ b\ a)$.

Q11 `snoc`(x, L) *est la liste obtenue en ajoutant le* `typélt` x *à la fin de* L. *Exemple :*
`snoc`$(z, (a\ b\ c)) = (a\ b\ c\ z)$.

Q12 `concaténer`(L_1, L_2) *est la liste obtenue par concaténation de* L_1 *et* L_2. *Exemple :*
`concaténer`$((a\ b\ c), (d\ e)) = (a\ b\ c\ d\ e)$.

Q13 `insérer_liste`(L_1, n, L_2) *est la liste obtenue en insérant dans* L_2 *la liste* L_1 *à la* $n^{ème}$ *place. Exemple :*
`insérer_liste`$((y\ z), 2, (a\ b\ c)) = (a\ y\ z\ b\ c)$.

Q14 `dé_bégayer`(L) *est la liste obtenue en supprimant les répétitions d'éléments successifs. Exemple :*
`dé_bégayer`$(a\ b\ a\ a\ c\ b\ b) = (a\ b\ a\ c\ b)$.

Q15 `sous_liste_de`(L_1, L_2) *teste si* L_1 *est une sous-liste de* L_2, *c'est-à-dire si* L_2 *contient successivement les éléments de* L_1 *dans l'ordre. Exemples :*
`sous_liste_de`$((b\ c), (a\ b\ c\ d))$ = `vrai` *et* `sous_liste_de`$((b\ c), (a\ b\ e\ c\ d))$ = `faux`.

Q16 `sous_séquence_de`(L_1, L_2) *teste si* L_1 *est une sous-séquence de* L_2*, c'est-à-dire si* L_2 *contient les éléments de* L_1 *dans l'ordre, avec éventuellement des éléments propres à* L_2 *insérés. Exemple :*
`sous_séquence_de`$((b\ c), (a\ b\ e\ c\ d)) =$ `vrai`*. (si* L_1 *est une sous-liste de* L_2 *alors* L_1 *est une sous-séquence de* L_2 *mais la réciproque est fausse).*

---

9 Piles, files et autres structures linéaires

Les piles. Structure proche des listes, mais on ne se permet pas, en général, d'interclasser ou de supprimer un élément en-dehors du sommet (cf. implantation).
Type abstrait : le même, à un renommage près de celui des listes : `cons` $\mapsto$ `empiler`, `l_vide` $\mapsto$ `pile_vide`, `prem` $\mapsto$ `sommet`, `reste` $\mapsto$ `dépiler`, (`est_vide` reste `est_vide` : le même nom d'opération est utilisé pour deux opérations différentes ayant des types en entrée différents).
Implantation à l'aide de tableaux. Une pile p sera représentée par un tableau tab à une dimension de taille N, dont les éléments sont des `typélt` et par un entier naturel h donnant la hauteur de la pile : $p = (tab, h)$. L'implantation des opérations se fera de la façon suivante :
– `pile_vide`() correspond à l'affectation $h := 0$.
– $p :=$ `empiler`(x, p) se fait en testant si $h < N$ et, dans l'affirmative, à faire les affectations $tab[h] := x$ et $h := h + 1$. Si au contraire $h \geqslant N$, l'empilage est impossible (la pile est pleine).
– `sommet`(p) est $tab[h-1]$.
– $p :=$ `dépiler`(p) se fait en testant si $h > 0$. Si c'est le cas, l'instruction à effectuer sera $h := h - 1$. Dans le cas contraire, on ne fait rien (cf. l'axiome `dépiler`(`pile_vide`) = `pile_vide`).

Exercice 3 *Donner une implantation en C du type des piles d'entiers, en s'appuyant sur le principe décrit ci-dessus.*

Application à l'exécution des programmes récursifs (exemple de la factorielle).
Application : vérifier qu'une chaîne de caractères est bien parenthésée.

Exercice 4 `dépiler_jusqu'à`(x, P) *donne la liste obtenue en enlevant le sommet jusqu'à ce que ce sommet soit* x*. Si* x *n'appartient pas à la pile, cela donnera la pile vide. Donner le profil, un jeu d'axiomes, un algorithme récursif et un algorithme itératif pour la fonction* `dépiler_jusqu'à`*.*

Exercice 5 *Le type abstrait défini ci-dessus ne tient pas compte du fait que l'implantation peut se heurter à un tableau trop petit, insuffisant pour stocker tous les éléments de la pile. Pour pallier à ce problème, on peut proposer deux solutions :*

(a) Refuser d'empiler un élément quand la pile est pleine (et provoquer une erreur), ce qui suppose également d'avoir une opération d'accès testant si la pile est pleine ou non.

(b) Créer un nouveau tableau, par exemple 2 fois plus grand que le premier, recopier les premiers éléments dans ce nouveau tableau et le substituter à l'ancien.

Choisir la solution (a) et définir un nouveau type abstrait `pile` *(opérations primitives, axiomes) tenant compte de ce problème.*

Les files. Intuition : les files d'attente (dans une boutique ou dans une imprimante).

Les constructeurs du type abstrait `file` sont :
`file_vide` : $\longrightarrow$ `file`
`enfiler` : `typélt` $\times$ `file` $\longrightarrow$ `file`

Les accès de `file` sont :
`est_vide` : `file` $\longrightarrow$ `booléen`
`premier` : `file` $\longrightarrow$ `typélt`
`défiler` : `file` $\longrightarrow$ `file`

Exercice 6 *Donner un jeu d'axiomes pour le type abstrait* `file`*.*

On peut implanter une file f par la donnée d'un tableau à une dimentsion tab de taille N et dont les éléments sont des `typélt`, et de deux indices *début* et *fin* : $f = (tab, \textit{début}, \textit{fin})$. Si *début* $\leqslant$ *fin*, les éléments de la file sont $tab[\textit{début}]$, $tab[\textit{début}+1]$, ... $tab[fin]$. Si *début* $>$ *fin*, les éléments de la file sont $tab[\textit{début}]$, $tab[\textit{début}+1]$, ... $tab[N-1]$, $tab[0]$, $tab[1]$, ... $tab[fin]$. Le premier élément de la file est $tab[fin]$.
On peut aussi implanter les files à l'aide de listes. Dans ce cas, on peut considérer par exemple que le premier élément de la file f est `prem`(f) (`prem` étant l'opération primitive sur les listes).

Exercice 7 *Donner une implantation en C du type des files de chaînes de caractères, en s'appuyant sur le principe décrit ci-dessus.*

Autres structures linéaires simples. Les ensembles finis. Les multi-ensembles finis. Les listes hétérogènes. Les listes circulaires. Les listes bidirectionnelles.