  
Cahier de TD de SDD  
: structures séquentielles

Listes chaînées

Exercice 1.1 – Parcourir intégralement une liste 🟅

Concevoir deux algorithmes, l’un itératif, l’autre récursif, qui retournent la longueur[[1]](#footnote-1) d’une LSC[[2]](#footnote-2).

Exercice 1.2 – Parcourir une liste jusqu’à un certain point 🟅

Concevoir deux algorithmes, l’un itératif, l’autre récursif, qui retournent la position[[3]](#footnote-3) d’un élément[[4]](#footnote-4) donné dans une liste (retournent 0 si la liste ne comporte aucune occurrence[[5]](#footnote-5) de l’élément).

Exercice 1.3 – Analyser le contenu d’une liste 🟅

Concevoir deux algorithmes, l’un itératif, l’autre récursif, qui retournent le nombre d’occurrences d’un élément donné dans une liste.

Exercice 1.4 – Ajouter un élément dans une liste 🟅🟅

Concevoir un algorithme qui ajoute un élément donné à une position donnée dans une LSC. Si la position est supérieure à la longueur de la liste, ajouter l’élément en queue[[6]](#footnote-6) de liste. Si la position donnée est 0, ajouter l’élément en tête[[7]](#footnote-7) de liste. Adapter l’algorithme pour le cas d’une LDC[[8]](#footnote-8).

Exercice 1.5 – Concaténer deux listes 🟅🟅

Concevoir un algorithme itératif qui concatène[[9]](#footnote-9) deux listes données en argument modifié[[10]](#footnote-10).

Aucune copie d’élément ne doit être effectuée. Optimiser en tenant compte des cas où l’un des deux arguments, ou les deux, sont des listes vides. Réfléchir à l’effet de l’algorithme sur une même liste donnée pour les deux arguments. Même question avec deux sous-listes[[11]](#footnote-11) d’une même liste.

Exercice 1.6 – Construire une LDC à partir d’une LSC 🟅🟅🟅

Concevoir deux algorithmes, l’un itératif, l’autre récursif, qui retournent une LDC dont les éléments sont la copie, en ordre inverse, des éléments d’une LSC donnée. LA LDC DOIT ETRE TRAITEE COMME UNE DONNEE MODIFIEE. OPTIMISER.

Exercice 1.7 – Décomposer une LSC 🟅🟅🟅

Concevoir un algorithme qui produit deux listes à partir d’une liste, en travaillant uniquement par référence. La première des deux listes produites est constituée des nœuds des éléments pairs de la liste d’entrée, la seconde, ceux des éléments impairs. L’ordre de placement des éléments de ces deux listes (relation « précède en position »), est celui de la liste initiale. La liste initiale devient, à l’issue du traitement, la liste vide[[12]](#footnote-12). Les 3 listes doivent être des données modifiées.

Exercice 1.8 – Pivoter une LSC circulaire 🟅🟅🟅

Ecrire un algorithme qui effectue la rotation[[13]](#footnote-13) d’un nombre de pas donnés d'une LSC circulaire. LA LSC DOIT ETRE TRAITEE COMME UNE DONNEE MODIFIEE. OPTIMISER.

Problème 1.9 – Mesurer la longueur d’une LSC généralisée 🟅🟅🟅🟅

En utilisant la fonction de concaténation[[14]](#footnote-14) d’une liste sur l’une des ses sous-listes, on produit une structure qui comporte une partie linéaire suivie d’une partie circulaire

Concevoir un algorithme performant qui retourne sous forme de données modifiées, la longueur de la partie linéaire et celle de la partie circulaire (son périmètre).

Problème 1.10 – Concevoir une structure chaînée complexe 🟅🟅🟅🟅

Concevoir une structure de liste circulaire de listes circulaires dotée des fonctionnalités d’ajout et de retrait d’élément à une position donnée.

Une position est déterminée par un couple de paramètres indiquant la position relative en nombre de pas de rotation[[15]](#footnote-15), le premier pour repérer la place de la liste principale qui contient la liste secondaire visée, le second pour la place visée dans la liste secondaire.

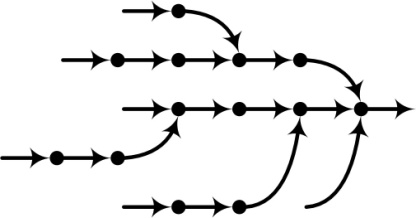
La structure doit maintenir le périmètre respectif de chacune des listes (principale et secondaires) pour assurer une performance optimale (éviter les tours inutiles).

Pour les plus avancés, compléter avec les fonctionnalités suivantes :

* Extraire sous forme de liste circulaire une coupe transversale constituée des copies des éléments situés à la même position dans chacune des listes secondaires.
* Pivoter l’ensemble de la structure, c'est-à-dire la liste principale et chaque liste secondaire, en fonction d’une position relative par rapport à la position actuelle (0, 0).

Problème 1.11 – Point de confluence de deux listes 🟅🟅🟅🟅

Concevoir un algorithme itératif qui retourne l’adresse du point de confluence[[16]](#footnote-16) de deux LSC s’il existe, une adresse nulle sinon. Donner une première version naïve en complexité quadratique, une seconde version performante en complexité linéaire.



*Exemples de 6 listes confluentes entre-elles (ici 4 points de confluence)*

Exercices supplémentaires pour les plus courageux

Exercice 1.12 – Supprimer un élément d’une liste 🟅🟅

Concevoir deux algorithmes, l’un itératif, l’autre récursif, qui suppriment toutes les occurrences d’un élément donné dans une LSC. Adapter l’algorithme pour le cas d’une LDC.

Exercice 1.13 – Permuter deux places dans une LSC 🟅🟅🟅

Concevoir un algorithme qui échange les positions de deux places d’une LSC. Il est interdit de permuter les éléments : il faut effectivement permuter les deux places sans modifier leur contenu respectif (travail par référence vs. travail par valeur).

Exercice 1.14 – Inverser une LSC 🟅🟅🟅🟅

Concevoir un algorithme récursif qui inverse l’ordre une LSC sans recopier ses éléments.

Piles et Files

Exercice 2.1 – Principe de fonctionnement 🟅

Donner le contenu de la pile pour chaque opération de la suite Q\*UES\*\*\*TI\*ON\*FAC\*\*\*IL\*\*\*E\*\*. Chaque lettre provoque un empilement et chaque astérisque un dépilement.

Donner le contenu de la file pour chaque opération de la suite Q\*UES\*\*\*TI\*ON\*FAC\*\*\*IL\*\*\*E\*\*. Chaque lettre provoque un empilement et chaque astérisque un dépilement.

Exercice 2.2 – Implémenter pile et file à l’aide d’une LSC 🟅🟅🟅

Réaliser une structure de pile et ses fonctions de manipulation, à l’aide d’une LSC.

Même exercice avec une file.

Exercice 2.3 – Inverser file et pile 🟅🟅

Concevoir un algorithme qui inverse l’ordre des éléments contenus dans une file donnée.

Travailler sans effectuer de copie des éléments.

Même question avec une pile.

Exercice 2.4 – Parcours itératif d’un arbre 🟅🟅

Concevoir deux algorithmes de parcours itératif d’un arbre, le premier en largeur, le second en profondeur.

Exercice 2.5 – Simulation de la récursivité 🟅🟅🟅

1. Etant donné la méthode récursive suivante de calcul d’une puissance entière d’un entier :



En C (pour information) :

**unsigned** power1**(unsigned** a**, unsigned** n**) {**

**return** n **==** 0 **?** 1 **:** a **\*** power1**(**a**,** n **-** 1**);**

**}**

Concevoir un algorithme qui reproduit cet algorithme en simulant la récursivité à l’aide d’une pile.

Remarque : les en-têtes C des structures et opérations à utiliser pour cet exercice sont fournis :

lStack \*newEmptyStack(); // création d’une nouvelle pile

int isEmptyStack(lStack \*ps); // test qui retourne 1 si la pile est vide, -1 si elle est nulle, 0 sinon

void push(void \*val, lStack \*ps); // empilement de val dans \*ps

int pop(void \*\*pval, lStack \*ps); // dépilement de \*pval de \*ps

1. Même question avec la méthode de l’exponentiation rapide :



En C (pour information) :

**unsigned** power2(**unsigned** a, **unsigned** n**)**

**{**

**if (**n **==** 0**) return** 1**;**

**if (**n **%** 2 **==** 0**)**

**{**

**unsigned** sqrt **=** power**(a,** n **/** 2**);**

**return** sqrt **\*** sqrt**;**

**}**

**else return** a **\*** power**(a,** n **-** 1**);**

**}**

Exercice 2.6 – Application détecteur de palindrome 🟅🟅

On appelle palindrome un mot m qui se lit de la même façon de gauche à droite et de droite à gauche. Par exemple les mots "elle" et "radar" sont des palindromes. Un mot étant considéré comme une liste de caractères, trouver une représentation chaînée judicieuse pour les mots et écrire une procédure qui reconnaît si un mot est un palindrome.

Problème 2.7 – Application traducteur d’expression arithmétique 🟅🟅🟅

Ecrire un algorithme pour convertir une expression infixée valide avec parenthèses en expression postfixée.

1. Longueur d’une liste – son nombre de maillons (ou places, ou encore nœuds). [↑](#footnote-ref-1)
2. LSC – Liste Simplement Chaînée [↑](#footnote-ref-2)
3. Position – place relative par rapport à celle de la tête de liste (position 1) [↑](#footnote-ref-3)
4. Elément – valeur, contenu d’un nœud [↑](#footnote-ref-4)
5. Occurrence – une copie particulière d’un élément (sous-entendu, parmi plusieurs copies) [↑](#footnote-ref-5)
6. Queue de liste – sa dernière place [↑](#footnote-ref-6)
7. Tête de liste – sa première place [↑](#footnote-ref-7)
8. LDC – Liste Doublement Chaînée [↑](#footnote-ref-8)
9. Concaténation – Liaison de la queue d’une première liste à la tête d’une seconde liste [↑](#footnote-ref-9)
10. Donnée modifiée – variable passée à la fonction de telle manière à ce que la fonction puisse en modifier la valeur. En C, cela s’implémente avec une indirection de pointeur (opérateur &). [↑](#footnote-ref-10)
11. Sous-liste – Une sous-liste est un pointeur vers un maillon particulier d’une liste, ce dernier constituant alors la tête de cette sous-liste. Le nombre de sous-listes d’une liste est donc égal à sa longueur. [↑](#footnote-ref-11)
12. Liste vide – liste qui ne comporte aucun nœud (représentée en C par un pointeur nul) [↑](#footnote-ref-12)
13. Rotation – Au même titre que l’on préfèrera le terme de « périmètre » à celui de « longueur » pour une LSC circulaire, on préfèrera le terme « rotation » à celui « d’itération ». Il va de soi que la rotation d’une liste circulaire d’un certain nombre de pas, obéit à une logique d’arithmétique modulaire. Il est en effet inutile d’effectuer « des tours pour rien ». [↑](#footnote-ref-13)
14. Voir exercice 2.1 [↑](#footnote-ref-14)
15. Voir exercice 2.4 [↑](#footnote-ref-15)
16. Maillon commun aux deux listes : s’il existe, il est nécessairement unique. [↑](#footnote-ref-16)