Pratique de la programmation et projet
TP 7 : Listes (simplement) chaînées

Frédéric Vivien

1 Définitions des listes chaînées

1.1 Définitions

Une **liste chaînée** est une structure de données dans laquelle les objets sont arrangés linéairement, l'ordre linéaire étant déterminé par des pointeurs sur les éléments.

Chaque élément de la liste contient :

1. Une donnée appelée *clé*.
2. Un champ *successeur* qui est un pointeur sur l'élément suivant dans la liste chaînée.
3. Un nombre quelconque (qui peut être nul) d'autres champs de données.

Si le champ *successeur* d'un élément vaut NIL, cet élément n'a pas de successeur et est donc le dernier élément ou la **queue** de la liste. Le premier élément de la liste est appelé la **tête** de la liste. Une liste L est manipulée via un pointeur vers son premier élément, que l'on notera TÊTE(L). Si TÊTE(L) vaut NIL, la liste est vide.

La figure 1 présente un exemple de liste chaînée et montre les conséquences des opérations INSERTION et SUPPRESSION sur une telle structure de données. Dans cet exemple, les éléments de la liste ne contiennent pas d'autres champs de données que la clé.

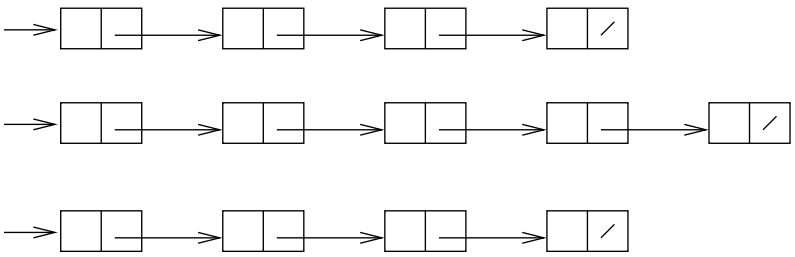

FIG. 1 – Exemple de liste chaînée : a) initialement la liste chaînée contient les valeurs 9, 6, 4 et 1 ; b) état de la liste chaînée après l'opération INSERTION(5) ; c) état de la liste chaînée après l'opération SUPPRESSION(4).

Une liste chaînée peut prendre plusieurs formes :

- **Liste doublement chaînée** : en plus du champ *successeur*, chaque élément contient un champ *prédécesseur* qui est un pointeur sur l'élément précédent dans la liste. Si le champ *prédécesseur* d'un élément vaut NIL, cet élément n'a pas de prédécesseur et est donc le premir élément ou la **tête** de la liste. Une liste qui n'est pas doublement chaînée est dite **simplement chaînée**.
  La figure 2 présente un exemple de liste doublement chaînée et montre les conséquences des opérations INSERTION et SUPPRESSION sur une telle structure de données.
- **Triée** ou **non triée** : suivant que l'ordre linéaire des éléments dans la liste correspond ou non à l'ordre linéaire des *clés* de ces éléments. Le lieu de l'insertion d'un élément dans une liste triée dépend donc

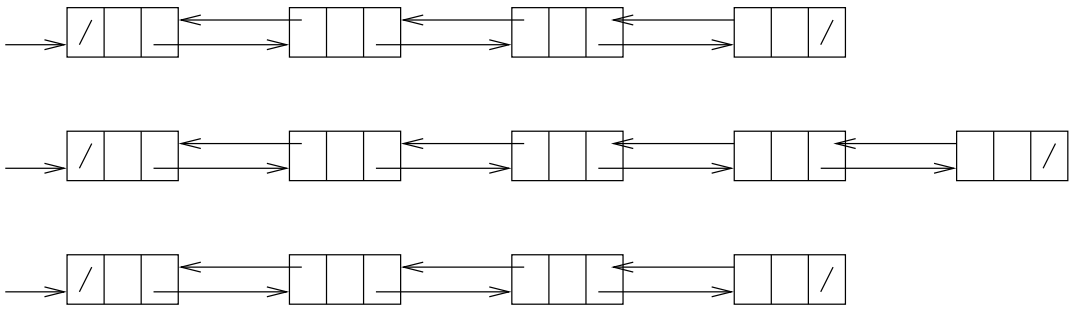

FIG. 2 – Exemple de liste doublement chaînée : a) initialement la liste contient les valeurs 9, 6, 4 et 1 ; b) état de la liste après l'opération INSERTION(5) ; c) état de la liste après l'opération SUPPRESSION(4).

de la valeur de la clé de cet élément. Par contre, une insertion dans une liste non triée a *toujours lieu* en tête de la liste.

– **Circulaire** : si le champ *précécesseur* de la tête de la liste pointe sur la queue, et si le champ *successeur* de la queue pointe sur la tête. La liste est alors vue comme un anneau.

1.2 Algorithmes de manipulation des listes chaînées

1.2.1 Recherche

L'algorithme RECHERCHE-LISTE(L, k) trouve le premier élément de clé k dans la liste L par une simple recherche linéaire, et retourne un pointeur sur cet élément. Si la liste ne contient aucun objet de clé k, l'algorithme renvoie NIL.

RECHERCHE-LISTE(L, k)
  $x \leftarrow$ TÊTE(L)
  tant que $x \neq$ NIL et *clé*(x) $\neq k$ **faire**
    $x \leftarrow$ *successeur*(x)
  renvoyer x

Cet algorithme manipule aussi bien des listes simplement que doublement chaînées.

1.2.2 Insertion

Étant donné un élément x et une liste L, l'algorithme INSERTION-LISTE insère x en tête de L.

INSERTION-LISTE(L, x)
  successeur(x) $\leftarrow$ TÊTE(L)
  si TÊTE(L) $\neq$ NIL **alors** *prédecesseur*(TÊTE(L)) $\leftarrow x$
  TÊTE(L) $\leftarrow x$
  prédecesseur(x) $\leftarrow$ NIL

Cet algorithme est écrit pour les listes doublement chaînées. Il suffit d'ignorer les deux instructions concernant le champ *prédécesseur* pour obtenir l'algorithme équivalent pour les listes simplement chaînées.

1.2.3 Suppression

L'algorithme SUPPRESSION-LISTE élimine un élément x d'une liste chaînée L. Cet algorithme a besoin d'un pointeur sur l'élément x à supprimer. Si on ne possède que la clé de cet élément, il faut préalablement utiliser l'algorithme RECHERCHE-LISTE pour obtenir le pointeur nécessaire.

SUPPRESSION-LISTE(L, x)
  si *prédécesseur*(x) $\neq$ NIL
    alors *successeur*(*prédécesseur*(x)) $\leftarrow$ *successeur*(x)
    sinon TÊTE(L) $\leftarrow$ *successeur*(x)
  si *successeur*(x) $\neq$ NIL
    alors *prédécesseur*(*successeur*(x)) $\leftarrow$ *prédécesseur*(x)

Cet algorithme est écrit pour les listes doublement chaînées. L'algorithme équivalent pour les listes simplement chaînées est plus compliqué puisqu'avec les listes simplement chaînées nous n'avons pas de moyen simple de récupérer un pointeur sur l'élément qui précède celui à supprimer...

SUPPRESSION-LISTE(L, x)
  si x = TÊTE(L)
    alors TÊTE(L) $\leftarrow$ *successeur*(x)
    sinon $y \leftarrow$ TÊTE(L)
          tant que *successeur*(y) $\neq x$ **faire** $y \leftarrow$ *successeur*(y)
          successeur(y) $\leftarrow$ *successeur*(x)

2 Implémentation des listes simplement chaînées en C

2.1 Structure de données

On définit les listes chaînées au moyen du type suivant :

```
typedef struct cell {
int           valeur;
struct cell * suivant;
} Cellule;
```

2.2 Fonctions

Écrivez les fonctions suivantes :

1. `is_empty` : teste si une liste est vide.
2. `car` : retourne la valeur contenue dans la tête de la liste argument.
3. `cdr` : retourne le reste de la liste argument.
4. `longueur` : calcule la longueur d'une liste.
5. `print_list` : affiche le contenu d'une liste.
6. `appartient` : teste l'appartenance d'un élément à une liste.
7. `egal` : test l'égalité de deux listes (égalité des contenus).
8. `dernier` : renvoie la valeur du dernier élément de la liste.
9. `cons` : rajoute un élément en tête d'une liste.
10. `rplaca` : remplace la valeur en tête de la liste.
11. `rplacd` : remplace le reste de la liste.
12. `copy` : crée une liste copie de la liste argument.
13. `concat` : concatène deux listes, vous écrirez deux versions de cette fonction, l'une avec remplacement physique et l'autre sans (autrement dit, l'une des deux versions modifie les listes passées en argument, mais pas l'autre version).
14. `reverse` : renverse une liste (écrivez deux versions de cette fonction : une avec et une sans accumulateur).