Ensembles et Séquences vs. Listes Chaînées

Florent Bouchez Tichadou

24 septembre 2018

Nous avons vu précédemment que l'on pouvait utiliser des tableaux (bas-niveau) pour stocker des ensembles ou des séquences, structures de données de haut-niveau. Les tableaux ont de bonnes propriétés telles qu'un stockage compact en mémoire et un accès aux éléments en temps constant à partir d'un indice, mais ont également des limitations comme une taille fixe et la difficulté d'insertion d'éléments : décalage nécessaire de tous les éléments quand l'ordre est important, ce qui est le cas des séquences.

Dans ce document nous allons découvrir une autre structure de données de bas-niveau qui se détache de ces limitations (et par contre en comporte d'autres comme nous le verrons) : les listes chaînées.

1 Modèle mémoire

Avant de présenter les listes chaînées, il est important de bien comprendre le fonctionnement de la mémoire : celle-ci s'apparente à un tableau gigantesque, et l'on peut stocker une valeur dans chaque case. L'indice d'une case s'appelle l'*adresse mémoire*, et le *contenu* est une valeur en général sur 32 bits ou 64 bits.

Du point de vue de la mémoire, le contenu n'a pas de type. C'est le programme qui connaît le type de chaque valeur et est ainsi capable de l'interpréter correctement. Par exemple, la valeur binaire `01000001 01101100 01100111 01101111` peut être interprétée comme la valeur entière 1097623407, un booléen `True`, la chaîne de caractères `"Algo"`, ou... l'adresse mémoire `0x416c676f` (représentation hexadécimale).

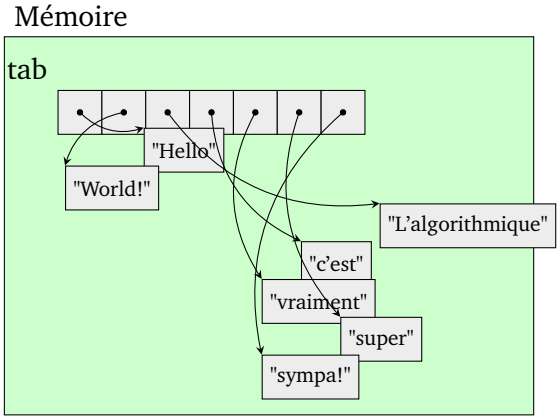
Il est ainsi possible de stocker en mémoire des adresses mémoire. C'est par exemple le cas pour un tableau de chaînes de caractères : le tableau ne contient pas directement les caractères mais les adresses de début des chaînes (voir Figure 1a). Lorsqu'on veut accéder à une chaîne à partir de son indice dans le tableau, le programme n'a qu'à « suivre » le lien stocké dans le tableau, i.e., aller à l'adresse mémoire indiquée.

Nous allons utiliser cette technique pour construire une liste chaînée, c'est-à-dire un objet constitué de multiples éléments appelés *cellules*, tel que chaque cellule contient deux champs : l'un stocke l'élément et l'autre l'adresse de la cellule qui la suit (voir Figure 1b). À partir de l'adresse de la première cellule (appelée usuellement « tête »), il est donc possible d'accéder à toutes les cellules en suivant les liens jusqu'en bout de chaîne (appelée « queue », et pointant vers l'adresse spéciale `Nil` (ou `Null`)).

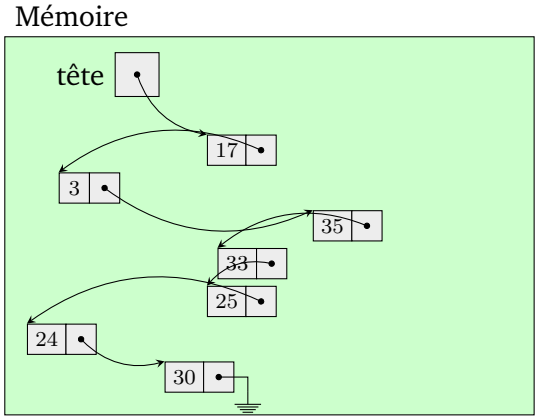
2 Listes chaînées

À l'inverse des tableaux, les éléments d'une liste chaînée ne sont pas stockés de manière contiguë dans la mémoire. Il est donc très facile de rajouter un nouvel élément à une liste chaînée : il suffit de réserver un espace pour une nouvelle cellule dans la mémoire et de modifier les *liens de chaînage* (adresses mémoires) pour que la liste « passe » par la nouvelle cellule.

Algorithmiquement, nous avons besoin d'un mécanisme pour passer d'une cellule à une autre. Nous préférons nous abstraire d'une implantation à base d'adresses mémoire (« pointeurs »), qui dépend du langage de programmation utilisé. Nous allons donc utiliser le mécanisme générique des *références* pour définir nos cellules.



(a) Tableau de *strings*



(b) Liste chaînée d'entiers

FIGURE 1 – Représentations en mémoire de tableaux d'adresses et de listes chaînées.

Déclaration de types et variables.

```
type Cellule : { val : élément,
                 suivant : référence d'une Cellule }
type Liste chaînée :
                 { tête : référence d'une Cellule }
cel : Cellule
liste : Liste chaînée
```

Création d'une liste à un élément.

```
cel ← nouvelle Cellule
cel.valeur ← 42
cel.suivant ← Nil
liste.tête← cel
```

3 Séquences et listes chaînées

Une liste chaînée peut contenir un nombre arbitrairement grand d'éléments (dans la limite de la mémoire disponible). Sa longueur (nombre d'éléments) est dynamique et il est facile d'ajouter un élément en conservant l'ordre existant. Cette structure de donnée bas-niveau est une très bonne candidate pour stocker des séquences haut-niveau ! C'est également un bon choix pour y stocker des ensembles si l'on sait par exemple à l'avance que la longueur est très variable.

De manière bas-niveau, une séquence peut donc être directement représentée par une liste chaînée, c'est-à-dire une référence sur cellule : la « tête » de la liste. En fonction des besoins on peut éventuellement garder d'autres références comme par exemple la « queue » de la liste (dernière cellule), ou une référence vers la cellule précédente (liste doublement chaînée, voir exercices à la fin du document).

Déclaration d'une séquence

```
type Séquence : Liste chaînée
S : Séquence
cel : Cellule
```

Affichage d'une séquence

```
cel ← S.tête
tant que cel ≠ Nil faire
    afficher (cel.valeur)
    cel ← cel.suivant
```

Les sections suivantes détaillent les implantations bas-niveau des opérations usuelles sur les séquences.

3.1 Création de séquence, accès aux éléments et successeurs

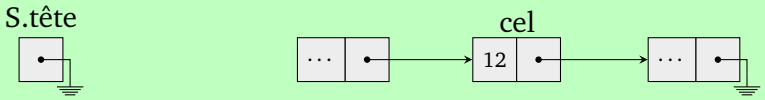
Pour créer une séquence vide, il nous suffit d'une référence sur. . . rien ! Et oui, une séquence vide ne contient aucune cellule !
Pour accéder à un élément, lorsque l'on a une référence vers sa cellule, on consulte son champ `valeur`.
Pour accéder au successeur d'un élément, toujours en partant de sa cellule, on utilise le champ `suivant`.

```
S : Séquence
S.tête ← Nil                    /* Séquence vide */
```

```
afficher ("La cellule contient : ", cel.valeur)
cel.val ← cel.val+12            /* modification */
cel ← cel.suivant               /* on passe à la suivante */
```

Exemple

Représentations mémoire d'une séquence vide et d'une cellule contenant 12.



Il est à noter que dans une liste chaînée, il n'est pas possible d'accéder directement au $i^{\text{ème}}$ élément comme dans un tableau : il faut partir de la tête et suivre les liens `suivant` autant de fois que nécessaire (cf. section suivante).

3.2 Itération sur tous les éléments et longueur

Pour parcourir une séquence représentée par une liste chaînée, nous accédons à la tête de la liste, puis aux éléments suivants les uns après les autres grâce aux liens de chaînage. Pour obtenir la longueur, nous utilisons le même schéma de parcours en incrémentant un compteur. [1]

Parcours et affichage

```
cel ← S.tête
tant que cel ≠ Nil faire
    afficher (cel.valeur)
    cel ← cel.suivant
```

Calcul de la longueur

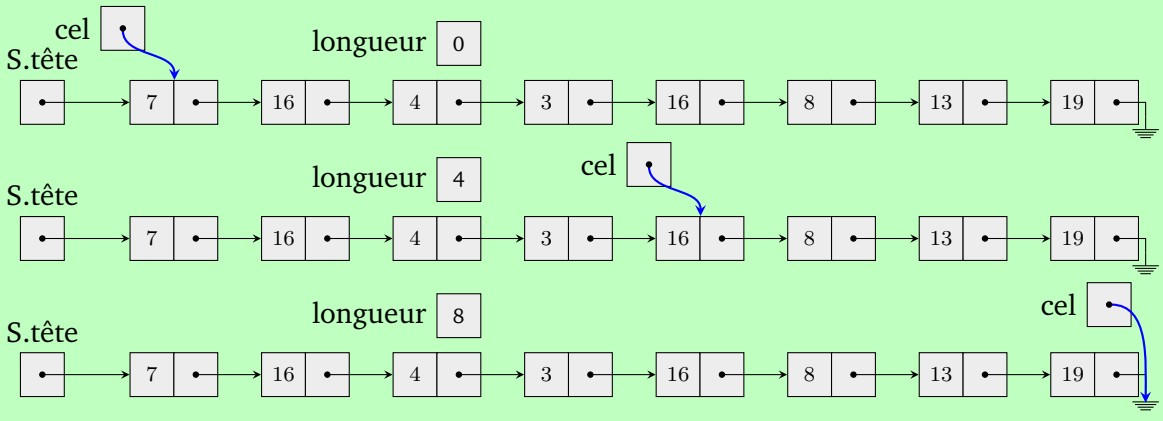
```
cel ← S.tête ;   longueur← 0
tant que cel ≠ Nil faire
    longueur ← longueur+1
    cel ← cel.suivant
```

Accès au $i^{\text{ème}}$ élément

```
cel ← S.tête
faire i fois
    cel ← cel.suivant
```

Exemple

Dessin de l'invariant suivant durant un calcul de longueur : « `longueur` contient le nombre de cellules avant celle pointée par `cel` dans la séquence. »



1. Si c'est une opération très souvent effectuée, il est aussi possible de sauvegarder dans la structure de liste un champ `longueur` en plus de la `tête`.

3.2.1 Recherche simple d'un élément

La recherche d'un élément, ou test d'appartenance à une séquence, est une application directe des schémas d'itération sur liste chaînée. Nous proposons ici trois versions qui renvoient toutes Vrai si la valeur recherchée a été trouvée dans la séquence, et Faux sinon. La première utilise un booléen qui se « souvient » si l'élément a été vu ou non dans la séquence, tandis que les deuxième et troisième s'arrêtent dès que l'élément a été trouvé. La troisième est la manière la plus élégante, mais ne fonctionne que si l'on est dans une fonction, car le « retourner » permet alors de sortir de la boucle. La deuxième montre un schéma classique où la sortie de boucle dépend de deux conditions : on peut soit être en fin de séquence, soit avoir trouvé l'élément. Il faut alors re-tester l'une des conditions en sortie de boucle pour connaître l'état de sortie.

Recherche simple

```
recherche (S, v)
    cel ← S.tête
    trouvé ← Faux
    tant que cel ≠ Nil faire
        si cel.valeur = v alors
            trouvé ← Vrai
        cel ← cel.suivant
    retourner trouvé
```

Boucle avec double condition

```
recherche (S, v)
    cel ← S.tête
    tant que cel ≠ Nil et
     cel.valeur ≠ v faire
        cel ← cel.suivant
    retourner cel ≠ Nil
```

Sortie de boucle avec retourner

```
recherche (S, v)
    cel ← S.tête
    tant que cel ≠ Nil faire
        si cel.valeur = v alors
            retourner Vrai
        cel ← cel.suivant
    retourner Faux
```

Important

Pour le deuxième algorithme de recherche l'ordre des comparaisons dans la boucle « tant que » est **très** important : **il faut absolument vérifier que** `cel` **n'est pas Nil avant de tester** `cel.valeur`. Dans le cas contraire, on cherche le champ d'un objet potentiellement inexistant ce qui est une faute algorithmique, et dans le cas d'un programme réel source de bugs infâmes. Pour la même raison, on ne peut en sortie de boucle que tester si `cel` est Nil et non l'autre condition.

3.3 Ajout d'un élément

Nous sommes dans des séquences donc l'ordre est important. Distinguons ici plusieurs cas :

1. l'ajout en début de séquence, i.e., en tête de liste ;
2. l'ajout après un élément, i.e., après une cellule donnée ;
3. l'ajout en fin de séquence, i.e., en queue de liste.

Pour ajouter en tête de liste, on crée une nouvelle cellule dont le successeur est l'actuelle tête, puis la tête devient cette cellule.
L'ajout après un élément particulier est facile si l'on a une référence vers la cellule c de cet élément : de la même manière que pour l'ajout en tête, on crée une nouvelle cellule dont le successeur est c.suivant, puis on met à jour c.suivant pour pointer vers cette cellule.
Enfin, l'ajout en queue nécessite tout d'abord de trouver la queue, l'ajout se fera ensuite comme au point précédent (dans ce cas, c.suivant est Nil), sauf si la liste est vide (il n'y a pas de queue), auquel cas on crée simplement une cellule qui devient la tête (et également la queue).

Attention : dans tous les cas, *l'ordre* des modifications est important ! Si l'on commence par exemple par faire pointer `S.tête` vers la nouvelle cellule, on perd l'ancienne tête (et le reste de la liste avec !).

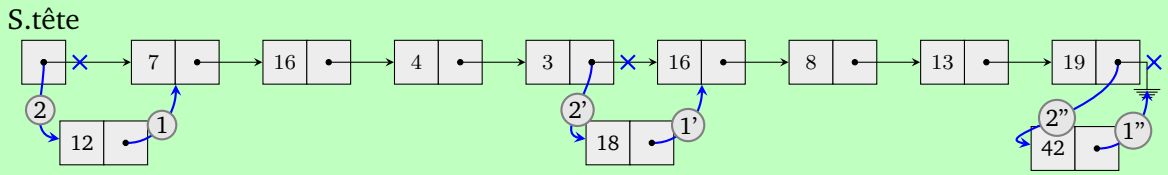
```
ajouter_debut (S,x)
    cel ← nouvelle Cellule
    cel.valeur ← x
    cel.suivant ← S.tête
    S.tête← cel

ajouter_après (S,x,pred)
    cel ← nouvelle Cellule
    cel.valeur ← x
    cel.suivant ← pred.suivant
    pred.suivant← cel
```

```
ajouter_fin (S,x)
    si S.tête = Nil alors
        ajouter_debut (S,x)
    sinon
        queue ← S.tête
        tant que queue.suivant ≠ Nil faire
            queue ← queue.suivant
        ajouter_après (S,x,queue)
```

Exemple

Ajout de 12, 18 et 42 respectivement au début, après 3, et en fin de séquence. Les numéros sur les liens montrent l'ordre de modification des liens de chaînage. Les croix montrent les liens qui disparaissent suite aux modifications numérotées 2.



3.4 Recherche et suppression d'un élément

Pour supprimer un élément, il faut faire pointer la cellule précédente vers la cellule suivante, puis libérer la cellule de l'élément (afin de ne pas saturer la mémoire). Le principal problème vient ici du fait que s'il est facile d'accéder à la cellule suivante, en revanche ce n'est pas directement possible d'obtenir la précédente : il faut repartir de la tête et parcourir la liste !

Il est donc vivement conseillé, si l'on sait que l'on va peut-être devoir supprimer un élément, de conserver une référence sur le prédécesseur afin d'éviter un nouveau parcours de la liste. Nous présentons ici une fonction recherche améliorée qui renvoie une référence sur la cellule de l'élément recherché (ou Nil si non trouvé), et renvoie également la référence de la cellule précédente (ou Nil si en tête de liste).

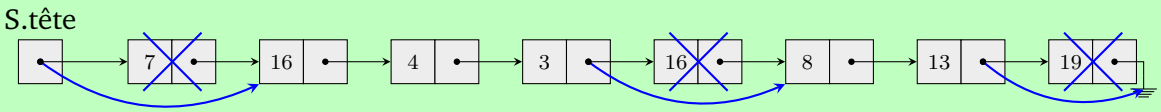
Note : encore une fois, l'ordre des comparaisons dans la boucle « tant que » est *très* important : d'aborder vérifier que `cel` n'est pas Nil *avant* de tester `cel.valeur`. De même, lors de la suppression, on ne libère la cellule qu'une fois qu'on a fini d'accéder à ses champs.

```
recherche (S, x)
    cel ← S.tête
    pred ← Nil
    tant que cel ≠ Nil et cel.valeur ≠ x faire
        pred ← cel
        cel ← cel.suivant
    retourner (cel, pred)
```

```
supprimer (S, x)
    (cel, pred) ← recherche (S, x)
    si cel = Nil alors
        Erreur ("Élément non trouvé !")
    si pred = Nil alors  S.tête ← cel.suivant
    sinon  pred.suivant ← cel.suivant
    libérer (cel)
```

Exemple

Suppression de la tête, d'un élément au milieu, et de la queue d'une liste chaînée.



3.5 Échange de deux éléments

Pour échanger deux éléments, nous devons modifier tous les liens de chaînages entre les prédécesseurs et les successeurs des cellules de ces éléments. Encore une fois, si l'on sait que l'on va devoir faire un échange, il est important de conserver une référence vers les prédécesseurs pour ne pas parcourir à nouveau la liste.

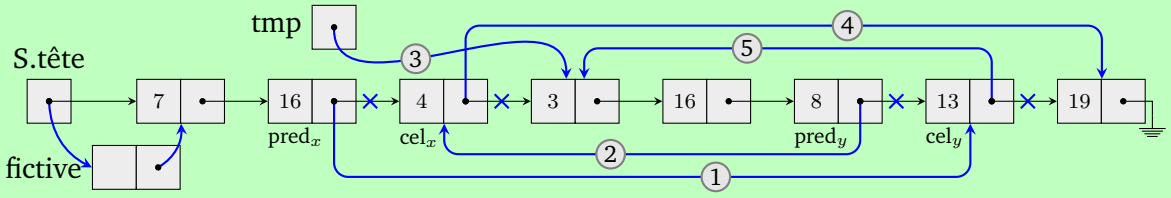
Nous avons ici un cas particulier assez pénible à traiter : si l'un des deux éléments occupe la cellule de tête. Pour éviter d'écrire trois fois un code assez délicat car souvent source d'erreurs, nous commençons par créer une cellule « fictive » que nous plaçons en tête de liste. Ainsi, nous savons que les prédécesseurs ne seront pas nuls (attention à choisir pour la cellule fictive une valeur qui ne sera jamais cherchée, ou à modifier la fonction de recherche pour qu'elle ignore cette cellule).

Note : autre cas particulier : si les deux éléments sont consécutifs. L'algorithme ci-dessous gère correctement ce cas, je vous laisse vous en convaincre par vous même en traçant l'exécution sur un exemple.

```
échange (S, x, y)
    fictive ← nouvelle Cellule
    fictive.suivant ← S.tête
    S.tête ← fictive
    (cel_x, pred_x) ← recherche (S, x)
    (cel_y, pred_y) ← recherche (S, y)
    pred_x.suivant ← cel_y
    pred_y.suivant ← cel_x
    tmp ← cel_x.suivant
    cel_x.suivant ← cel_y.suivant
    si cel_x = pred_y alors
        cel_y.suivant ← cel_x ;
    sinon
        cel_y.suivant ← tmp ;
    S.tête ← fictive.suivant
    libérer (fictive)
```

Exemple

Échange des éléments 4 et 13.



4 Exercices

4.1 Exercices à revisiter

Les listes chaînées sont simplement une possibilité d'implantation bas-niveau des structures de données « ensembles » et « séquences ».

Il est donc normalement possible de reprendre tous les algorithmes bas-niveau du document « Ensembles et Séquences vs. Tableaux », mais de transformer les algorithmes bas-niveau utilisant des tableaux en des algorithmes bas-niveau utilisant des listes chaînées.
Cela vous permet également de vérifier que vous avez compris la différence haut-niveau / bas-niveau : si votre algorithme haut-niveau ne peut être implanté en bas-niveau avec une liste chaînée, c'est que vous avez supposé que vos ensembles/séquences étaient stockés dans un tableau. . .

4.2 Exercices particuliers listes chaînées

Les exercices suivants pourraient bien entendu également être réalisés avec des tableaux, mais les listes chaînées sont bien adaptées (ou font travailler des notions intéressantes).

Exercice 1 (Insérer avant)

Nous avons vu dans ce document comment insérer une nouvelle cellule après une cellule donnée. Donnez l'algorithme pour réaliser l'insertion *avant* une cellule donnée.

Exercice 2 (Séparer en deux)

Soit une séquence S sous forme de liste chaînée. On veut séparer la séquences en S et $S2$ de sorte qu'un élément sur deux de la séquence initiale reste dans S, et les autres éléments soient dans $S2$. On veut conserver l'ordre initial et réaliser la séparation uniquement par modification des liens de chaînages (pas de suppression ni création de cellule).

1. Dessinez sur un exemple l'état initial de la séquence, et l'état final (liens d'une autre couleur).
2. Re-dessinez l'exemple avec l'état *en cours d'exécution* de la séparation : une partie de la séquence est déjà séparée, le reste est en attente de traitement. Ce dessin est votre invariant de boucle.
3. Identifiez sur votre précédent dessin la/les modifications à faire pour qu'une cellule de plus soit traitée. Ceci représente le corps de votre boucle et doit satisfaire l'invariant (se remettre dans le même « état » pour se préparer à l'itération suivante).
4. Déduisez-en le corps de votre boucle. Il ne reste maintenant qu'à déterminer l'initialisation (ce qu'on fait avant la boucle) et la condition d'arrêt (test de boucle).

Recommencer l'exercice mais en effectuant une séparation en fonction des valeurs des éléments que l'on suppose entiers : tous les $\geqslant 0$ restent dans S tandis que les < 0 doivent être dans $S2$.

Exercice 3 (File d'attente)

On cherche à modéliser une file d'attente : des clients rentrent à la poste (par exemple) et attendent leur tour.

On utilise pour cela une liste de type FIFO (*First In First Out*) : le premier arrivé est le premier sorti. Quand un nouveau client arrive, il doit être inséré en fin de chaîne via la fonction `arrivée`. Quand un guichet est libre, on prend le client en tête de chaîne grâce à la fonction `servir`.

1. Donnez une implantation des fonctions `arrivée (nom: string)` et `servir ()`.
2. L'ajout en queue impose de rechercher d'abord la queue d'une liste chaînée ce qui est coûteux. On aimerait garder en même une référence vers la queue. Modifiez la structure de donnée ainsi que la fonction `arrivée` pour que le coût soit indépendant de la taille de la file d'attente.

Exercice 4 (Dans les deux sens)

On considère un jeu de carte à deux joueurs. Ce jeu commence avec une ligne de cinq cartes tirées au hasard face visible. Chaque joueur peut, à tour de rôle, soit enlever la carte de début ou de fin de ligne, soit ajouter une carte (prise dans la pioche face cachée) au début ou à la fin de la ligne. On cherche ici à proposer un simulateur du jeu.

On veut implanter la ligne de cartes par une liste chaînée, qui serait efficace pour l'ajout et la suppression pour une des extrémités, mais très mauvaise pour l'autre extrémité.

1. Proposez une extension pour avoir des listes *doublement* chaînées : chaque cellule possède une référence vers la suivant mais également la précédent.
2. Donnez les algorithmes d'ajout et de suppression en début ou fin de séquence pour cette structure de données.
3. Supposons qu'on veuille également insérer et supprimer à n'importe quel endroit. Donnez les algorithmes nécessaires.

Exercice 5 (Le crêpier)

La crêperie du coin emploie un excellent crêpier, mais qui a des habitudes un peu maniaques. En particulier, une fois qu'il a fini de cuire sa pile de crêpes, il faut qu'elle soit bien ordonnée : de la plus grande en bas à la plus petite en haut. Pour modifier sa pile, le crêpier ne dispose que de sa spatule qu'il peut insérer n'importe où dans la pile, et retourner d'un coup toutes les crêpes qui se trouvent au dessus.

1. Donnez un algorithme haut-niveau (qui peut utiliser par exemple la phrase « retourner les trois premières crêpes ») qui ordonne correctement une pile de crêpes.
2. Donnez une implantation bas-niveau de votre algorithme en utilisant des listes chaînées.
3. Les crêpes ont toujours une face un peu plus jolie. Ordonnez la pile de manière à ce que la jolie face soit toujours vers le haut.