TD3 : Structures de données

L'objectif de cette séance est d'explorer les principes des structures de données, y inclus ses opérations. Ce TD vous aidera à devenir capable de :

- Designer une structure de données à partir d'une description
- Calculer la complexité des opérations d'une structure de données
- Modifier des structures de données pour améliorer la complexité de certaines opérations
- Comparer differentes structures de données et spécifier quand appliquer une structure
- Analyser une opération donnée en code C, sa complexité, et si elle s'exécute correctement

Instructions :

- Les exercices sont tous à faire, sauf les exercices difficiles marqués par (⚙).
- Lire bien toutes les informations fournies avant de faire chaque exercice du TD.
- Fournissez toujours des justifications valides pour toutes vos réponses.
- Si un exercice demande d'écrire un algorithme, écrivez votre solution en pseudo-code.

1 Structures de données

Exercice 1 : Analyse d'un code C (*). Considérez le code suivant :

```c
typedef struct _m_int {
    int val;
    struct _m_int * suiv;
} m_int;

int main() {
    // Insertion
    m_int tete = {0, NULL};
    for (int i=10; i>0; i--) {
        m_int nv = {i, NULL};
        nv.suiv = tete.suiv;
        tete.suiv = &nv;
    }
    // Affichage
    m_int* p = &tete;
    while (p) {
        printf("%d ", p->val);
        p = p->suiv;
    }
    printf("\n");
    return 0;
}
```

1. Pourquoi est-ce que le code ne s'exécute pas correctement ?
2. Quelle est le résultat attendu, si le code était correctement implémenté ?
3. Comment peut-on modifier le code à fin de résoudre le problème ?
   Ce n'est pas nécessaire d'écrire du code, juste décrire les modifications.

Exercice 2 : Construction d'une liste (*). Considérez la séquence d'opérations ci-dessous. Rappelez que la INSERERPOSITION(`lst`, i, v) insère un élément avec valeur v à la position i de la liste `lst` (et retourne un pointeur sur le maillon), et SUPPRIMERAPRES(`m`) supprime le maillon suivant de m.

Algorithme 1

1: `lst` ← `NULL`
2: INSERERPOSITION(`lst`, 0, 5)
3: INSERERPOSITION(`lst`, 1, 2)
4: m ← INSERERPOSITION(`lst`, 1, 1)
5: INSERERPOSITION(`lst`, 3, 4)
6: SUPPRIMERAPRES(m)
7: INSERERPOSITION(`lst`, 3, 3)

1. Quel est le résultat de l'exécution des opérations ci-dessus ?
2. Combien de fois l'attribut `suiv` d'un élément est-il accédé ou modifié ? (Cela permet d'estimer le nombre d'opérations)
3. Si la même séquence d'opérations est exécutée sur un tableau, combien de fois un élément du tableau est-il modifié ?

   Remarque: les tableaux dynamiques n'ont pas d'opération SUPPRIMERAPRES, mais nous pouvons la remplacer par une opération qui supprime la position correspondante.

Exercice 3 : Suppression d'éléments dans une liste (**). Dans cet exercice, nous considérons la suppression d'éléments dans une liste chaînée.

1. Décrivez (en pseudo-code) un algorithme qui prend en paramètre un pointeur vers une liste chaînée et un pointeur vers un maillon et supprime ce maillon.
2. Quelle est la complexité (dans le pire cas) de cet algorithme ?
3. Est-il possible de améliorer cette complexité (sans changer la structure de données) ? Justifiez la réponse.

Dans une variante de la liste chaînée, appelé *doublement chaînée*, chaque maillon stocke deux pointeurs : un vers l'élément suivant et un vers l'élément précédent.

4. Écrivez un algorithme de suppression pour les listes doublement chaînées avec une complexité améliorée. Comme dans l'exercice ci-dessus, l'algorithme prend en paramètre un pointeur vers une liste (doublement chaînée) et un pointeur vers un maillon, et supprime le maillon donné.

Exercice 4 : Mise en œuvre d'une file (***). Une file est une structure de données basée sur le principe *FIFO* (premier entré, premier sorti, en anglais *first in first out*). Une file stocke une séquence d'éléments dans un ordre spécifique et nous permet d'insérer des éléments et de retirer le plus ancien élément disponible, ce qui signifie que les éléments sont retirés dans l'ordre où ils sont arrivés.

Cette structure de données permet trois opérations :

- Insérer : insère un élément dans la file,
- Retirer : supprime le premier élément de la file (le plus ancien), et retourne sa valeur,
- Regarder : retourne la valeur du premier élément sans le supprimer.

Par exemple, si on insère les éléments 1, 2 et 3 dans une file, puis si on retire un élément, l'élément 1 sera retiré, car il a été inséré en premier, puis si on retire un autre élément, c'est l'élément 2 qui sera retiré, et ensuite l'élément 3.

1. Décrivez (en pseudo-code) comment implémenter une file à l'aide d'une liste chaînée standard, de sorte que les opérations de retirer et regarder aient une complexité de $\Theta(1)$.
2. Écrivez le pseudo-code de l'opération d'insertion avec une complexité de $\Theta(1)$, en utilisant un pointeur supplémentaire sur un élément de la liste.

Exercice 5 : Tri fusion sur des tableaux (***).

1. Écrivez une version du tri par fusion pour les tableaux.
2. Combien de mémoire supplémentaire cet algorithme alloue-t-il ?
3. (⚙) Décrire comment modifier l'algorithme pour qu'il n'alloue que $\Theta(n)$ octets de mémoire.

2 Pour aller plus loin

Exercice 6 : Gaspillage dans les tableaux dynamiques (⚙). L'un des inconvénients des tableaux dynamiques est qu'ils allouent plus d'espace que nécessaire pour les éléments qui y sont stockés.

Pour cet exercice, nous ne considérerons que des insertions et nous supposerons que la capacité de mémoire allouée est doublée lors de l'insertion d'un élément dans un tableau complet.

1. Quel est le nombre maximum de places vides dans un tableau dynamique en fonction de n, le nombre d'éléments ?
2. Quel est l'impact de la stratégie de réallocation sur ce nombre ? Par exemple, combien d'espaces vides y a-t-il si la capacité est multipliée par 1,5 (et non par 2) lors de la réallocation ?
3. Quelle est la complexité totale de n opérations d'insertion, si nous partons d'un tableau de n éléments et utilisons la stratégie de réallocation vu en cours ?

Une façon d'éliminer les espaces vides gaspillés consiste à n'allouer que la quantité de mémoire nécessaire. Dans ce cas, chaque fois que nous insérons un élément, nous devons réallouer la mémoire.

4. Quelle est la complexité d'effectuer n insertions avec cette stratégie ?

Exercice 7 : Recherche dans une liste chaînée triée (⚙). Dans cet exercice, nous considérons la recherche dans une liste chaînée où les valeurs sont triées. Une liste chaînée de n éléments est donnée en entrée, et nous ne considérons pas les insertions ou les suppressions.

1. Quelle est la complexité de la recherche d'un élément dans la liste ?
2. Supposons que, en plus du pointeur habituel vers l'élément suivant, chaque maillon stocke un pointeur vers l'élément situé 10 positions après (par exemple, le maillon à la position 1 a des pointeurs vers les positions 2 et 11).

   Comment la complexité de la recherche d'un élément change-t-elle si nous utilisons ce nouveau pointeur ?
3. Dans une autre variante, nous disposons de ce deuxième pointeur, mais il pointe sur l'élément situé $\sqrt{n}$ positions après lui.

   Pouvez-vous améliorer la complexité de la recherche en utilisant ce nouveau pointeur ?
4. Dans une dernière variation, chaque maillon a maintenant $\log(n)$ pointeurs vers les éléments 2^i positions après, pour chaque i, c'est-à-dire, chaque maillon a des pointeurs vers les éléments situés $1, 2, 4, \ldots$ positions après.

   Écrivez un algorithme de recherche avec complexité $\Theta(\log n)$ dans ce cas.