P31 Développement efficace - épisode 3

1 - Listes chaînées

Dans une liste chaînée, chaque élément est stocké indépendamment dans un `noeud`. Chaque `noeud` contient l'adresse du `noeud` suivant dans la liste, ce qui permet de parcourir la liste en partant du premier `noeud` et en suivant les adresses successives.

Contrairement à un `vector`, le nombre d'éléments d'une liste chaînée n'est pas limité par un espace mémoire alloué à l'avance. Les nouveaux `noeuds` sont ajoutés au fur et à mesure et "chaînés" aux éléments existants.

Par rapport à un `vector` : - l'accès à un élément en fonction de son indice nécessite de parcourir la liste depuis le début jusqu'à l'élément voulu - les éléments sont éparpillés en mémoire, ce qui limite l'efficacité de la mémoire cache lors des parcours - l'ajout et le suppression d'éléments sont simples car il n'y a pas à déplacer les éléments existants

> ***Créer les fichiers `containers/linked_list.h` et `containers/linked_list.c`.***

Comme avant, le fichier `.h` contiendra la définition de la structure et les déclarations des fonctions et le fichier `.c` contiendra le corps de ces fonctions.

Il y a deux structures à définir :

- `linked_list_node` : un noeud de la liste, qui contient :
  - `data` : un pointeur vers la zone mémoire contenant l'élément correspondant au noeud
  - `next` : un pointeur vers le noeud suivant de la liste
- `linked_list` : une liste chaînée qui contient :
  - `head` : l'adresse du premier noeud de la liste
  - `value_size` : la taille des éléments (en octets)

> ***Définir l'ensemble des fonctions suivantes :***

- initialisation et libération :

```
void ll_init(linked_list *ll, size_t value_size);
// il est probablement plus prudent d'écrire cette fonction après avoir écrit
// les fonctions de manipulation des données
void ll_free(linked_list *ll);
```

- accès aux propriétés courantes :

```
size_t ll_size(linked_list const *ll);
size_t ll_value_size(linked_list const *ll);
```

- manipulation des données :

```
// retourne un pointeur sur le noeud d'index index
linked_list_node const *ll_get_node(linked_list const *ll, size_t index);
// copie la mémoire pointée par value dans les données du noeud n
void ll_set_node_value(linked_list const *ll, linked_list_node *n, void const *value);
// copie la mémoire pointée par value dans un nouveau noeud à la fin de la liste ll,
// puis retourne un pointeur sur ce noeud
linked_list_node *ll_push_back(linked_list *ll, void const *value);
// copie la mémoire pointée par value dans un nouveau noeud au début de la liste ll,
// puis retourne un pointeur sur ce noeud
linked_list_node *ll_push_front(linked_list *ll, void const *value);
// copie la mémoire pointée par value dans un nouveau noeud inséré après le noeud n
// dans la liste ll, puis retourne un pointeur sur ce noeud
linked_list_node *ll_insert_after_node(linked_list *ll, linked_list_node *n,
```

```
                                        void const *value);
// copie la mémoire pointée par value dans un nouveau noeud inséré avant le noeud n
// dans la liste ll, puis retourne un pointeur sur ce noeud
linked_list_node *ll_insert_before_node(linked_list *ll, linked_list_node *n,
                                        void const *value);
// supprime le noeud n de la liste ll
void ll_erase_node(linked_list *ll, linked_list_node *n);
```

- recherche

```
linked_list_node const *find(linked_list const *ll, void const *value,
                             bool (*equals)(void const *a, void const *b));
```

2 - Tri par insertion

> ***Créer les fichiers `algos/insertion_sort.h` et `algos/insertion_sort.c`, puis y définir les deux fonctions suivantes :***
>
> ```
> void insertion_sort_vec(vector *v,
>                         int (*compare)(void const *a, void const *b));
> void insertion_sort_ll(linked_list *ll,
>                        int (*compare)(void const *a, void const *b));
> ```

Comme leur nom le laisse deviner, on attend de ces fonctions qu'elles trient les éléments d'un `vector` ou d'une liste chaînée en utilisant l'algorithme de tri par insertion.

> ***S'assurer que ces fonctions de tri fonctionnent comme attendu sur ces deux types de conteneur avec des types primitifs. Mesurer les temps d'exécution.***

Intuitivement, le tri par insertion est plus efficace sur une liste chaînée que sur un `vector` car il n'y a pas besoin de déplacer les éléments existants lors de l'insertion. Cependant, la dispertion en mémoire des éléments de la liste chaînée nuit à l'efficacité des parcours, ce qui peut la rendre dans la pratique moins efficace qu'un `vector` dans certains cas.

> ***Dans le fichier `utils.h`, définir une structure de données bien plus lourde en mémoire qu'un simple type primitif ou qu'un rectangle. Par exemple, on peut y stocker plusieurs tableaux de taille fixe d'entiers, de floats, etc.***
> ***On y mettra également un simple entier que l'on pourra initialiser aléatoirement et dont on se servira dans la fonction de comparaison entre deux instances de cette structure.***

Que peut-on constater sur les temps d'exécution du tri par insertion sur un `vector` et sur une liste chaînée de cette nouvelle structure ?