EXPLIQUER – STRUCTURE DE DONNEES – LAURA & CAMILLE – du 01-03 au 04-03-2021

**Introduction**

Définition

=> Stocker des éléments auxquels le programmeur veut pouvoir accéder plus tard

On appelle les différentes utilisations possibles de la structure de données des *opérations*

Une opération courante : la *lecture* => on veut récupérer un élément stocké dans la structure. Autres opérations : l'*insertion* (qui rajoute un nouvel élément dans la structure), ou la *suppression* (qui en enlève)

=> Connaissance des coûts (=la complexité) des opérations effectuées pour évaluer la complexité globale de l'algorithme

**Table de hachage et ses différentes implémentations**

Pour traduire les algorithmes qui manipulent des types de données dans un langage compréhensible pour l'ordinateur on dit qu'on implémente un algorithme. Il est donc aussi nécessaire d'implémenter les types de données comme les listes, les piles ou les files afin qu'ils soient utilisables par les ordinateurs. Ce sont des types abstraits de données. L'implémentation d'un type de données dépend du langage de programmation.

Qu'est-ce qu'une table de hachage ?

*Table de hachage* : un bon compromis entre le tableau et la liste. Ce sont des tableaux. On y stocke des données à un emplacement déterminé par une fonction de hachage

Avec les tables de hachage, on peut accéder à une case du tableau rien qu'avec le nom (alors que d'habitude, on utilise les index)

Pour ce faire, il faut pouvoir faire l'association entre chaque nom et un numéro de case de tableau

=> le principe : écrire une fonction qui prend en entrée une chaîne de caractères, fait des calculs avec, puis retourne en sortie un numéro correspondant à cette chaîne = l'indice de la case dans le tableau

Pourquoi utiliser une table de hachage ?

Ici, on dit que l'algorithme de recherche d'un élément a une complexité en O(n) car il faut parcourir toute la liste chaînée pour retrouver un élément donné, dans le pire des cas où celui-ci est à la fin

Une bonne fonction de hachage doit avoir les propriétés suivantes :

* Efficacement calculable
* Doit distribuer uniformément les clés
* Doit produire peu de collisions, c'est-à-dire qu'elle doit éviter de renvoyer le même nombre pour deux clés différentes
* En cas de collision, on peut utiliser l'adressage ouvert (recherche d'une autre case libre dans le tableau) ou bien le chaînage (combinaison avec une liste chaînée)

**Tableau et liste**

Liste

* L'ordre dans lequel les éléments sont dans la liste est important
* Une *liste* est "soit vide, soit un élément suivi d'une *liste*"
* La liste est une structure à laquelle il est très facile d'ajouter ou d'enlever (par filtrage, par exemple) des éléments. Il est bon de l'utiliser quand vous n'avez aucune idée du nombre exact d'éléments que vous allez manipuler. La seule méthode d'accès est le parcours de tous les éléments (ou du moins, ceux du début de la liste, on peut arrêter le parcours en cours de route)

Tableau

* *Tableau* : stocke des éléments du même type (homogènes) ensemble, dans des cases, chaque case étant étiquetée d'un numéro (généralement *indice*). Pour accéder à un élément particulier d'un tableau, on donne son indice
* Le calcul de la position de chaque élément dans ce tableau est facilité, en ajoutant un **décalage à une valeur de base**, c’est-à-dire l'**emplacement mémoire** du premier élément du tableau (généralement désigné par le nom du tableau).
* On considère que la taille d'un tableau est toujours connue, que tous les tableaux sont initialisés dès leur création et donc qu'elle est toujours O(N)
* Le tableau est très efficace quand le nombre d'éléments ne change pas et qu'on veut l'accès arbitraire

**Tuple**

*Tuple* : collection d'éléments, ordonnés, non modifiables, répétables

Une liste non-mutable. Une fois créé, un tuple ne peut en aucune manière être modifié

=> Entouré de parenthèses plutôt que de crochets

=> Les éléments ont un ordre défini, tout comme ceux d'une liste

=> on ne peut pas ajouter, enlever, rechercher d'éléments dans un tuple

=> il n'y a pas de méthodes index

=> on peut toutefois utiliser *in* pour vérifier l'existence d'un élément dans un tuple

=> plus rapides que les listes

=> le découpage fonctionne aussi, tout comme pour une liste

**Ensemble**

*L'ensemble* (set) : collection d'éléments, non ordonnés, modifiables, non répétables

Type abstrait qui peut stocker certaines valeurs, sans ordre particulier, sans répétition, pas de données en double

**File, pile, dèque**

Pile

* Une *pile* ou LIFO (dernier, premier sorti) est un **type de données abstrai**t, qui sert de collection d'éléments, avec deux opérations principales : *push*, qui ajoute un élément à la collection, et *pop*, qui supprime le dernier élément qui a été ajouté.
* Les piles sont utilisées pour maintenir les appels de fonction (la dernière fonction appelée doit terminer l'exécution en premier).
* Le principe est de stocker des données au fur et à mesure les unes au-dessus des autres pour pouvoir les récupérer plus tard.

File

* Une *file d'attente* ou FIFO (premier entré, premier sorti) est un **type de données abstrait** qui sert de collection d'éléments, avec deux opérations principales : mettre en file d'attente, le processus d'ajout d'un élément à la collection (l'élément est ajouté par l'arrière) et de queue, le processus de suppression du premier élément ajouté (élément retiré de la face avant).
* En programmation, les files sont utiles pour mettre en attente des informations dans l'ordre dans lequel elles sont arrivées
* *File d'attente circulaire* : l'avantage de cette structure est qu'elle réduit le gaspillage d'espace en cas d'implémentation de tableau car l'insertion du (n+1)ème élément se fait au 0ème index s'il est vide

Dèque (doubly ended queue)

C’est un conteneur se comportant comme une liste avec des ajouts et retraits rapides à chaque extrémité.