Mathématiques spéciales le 17 Novembre 2017

TP n°3 - Structures de données linéaires

Lors de l'étude de la complexité d'un algorithme, les calculs se basent sur certaines opérations élémentaires que l'on s'est fixées, le plus souvent sur le choix des opérations élémentaires les plus présentes ou les plus représentatives de l'algorithme. Il est bien-sûr légitime de se demander si certaines opérations élémentaires sont plus coûteuse en temps ou en mémoire que d'autres. De même, lorsqu'on stocke en mémoire un ensemble cohérent de données (comme une liste en Python par exemple) l'accès à une donnée ou l'écriture d'une donnée dans cet ensemble a un coût différent selon sa "situation" dans l'ensemble.

Ainsi on va étudier la notion informatique de **structure de données** qui permet de spécifier, selon les besoins du programmeur, le coût de chaque opérations élémentaires sur l'ensemble des données de la structure telle que l'accès ou l'écriture d'une donnée.

1. Structure de données

a. Définitions

En informatique, une **structure de données** est une façon de représenter en mémoire un ensemble de données en spécifiant :

— la manière d'attribuer une certaine quantité de mémoire à cette structure ;

— la manière d'accéder qu'elle contient.

Plus précisément, on dit que :

— La structure de données est **statique** si la quantité de mémoire attribuée à la structure lors de sa création est fixe et ne peut être modifiée (par exemple, la classe `tuple` ou `str` en Python).

— Dans le cas contraire, on dit que la structure de données est **dynamique** (par exemple, la classe `list` en Python) ;

De plus, une structure de données est dite **mutable** si on peut modifier les données contenues dans la structure après sa création : les classes `tuple` et `str` ne sont pas mutable en Python tandis que la classe `list` l'est.

Nous reviendrons sur la notion de *classe* dans un prochain T.P.

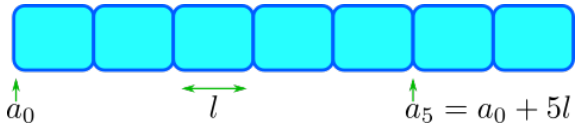
Voici quelques exemples de structures de données usuelles en informatique :

- Les structures de données linéaires représentent les données sous forme de suites finies : chaque élément dans une structure linéaire, à part dernier, possède un successeur ; il existe plusieurs type de structures linéaires dont les listes, les tableaux, les piles et les files
- les tableaux multidimensionnels (matrices) ;
- Les structures d'arbres (penser à la structure arborescente des fichiers en mémoire) ;
- Les structures de graphes ou structures relationnelles (penser aux bases de données).

b. Les tableaux et les listes chaînées

Les tableaux

Un tableau est une structure de données linéaires dans laquelle on stocke une suite de données de même type à des emplacements mémoire consécutifs. Du fait que les données partagent le même type, chaque emplacement dans la tableau a une taille constante l.

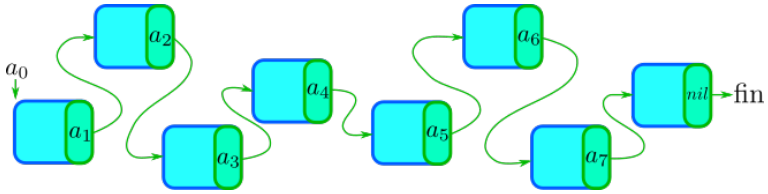


Ainsi, à la création du tableau, l'adresse a_0 du début de l'emplacement d'indice 0 est fixé, puis on accède à l'adresse de l'emplacement d'indice k en calculant $a_k = a_0 + kl$. L'accès à un emplacement se fait alors en temps constant : on obtient toujours l'adresse d'un emplacement du tableau en 2 opérations élémentaires (une addition, une multiplication).

Le désavantage de cette structure est qu'elle est statique : on ne peut pas ajouter d'emplacement à la fin du tableau car l'espace mémoire après le dernier emplacement n'est pas forcément libre.

Les listes chaînées

Une liste chaînée est une structure de données linéaires dans laquelle on stocke une suite de données de même type, et à chaque emplacement d'une donnée, on associe un pointeur contenant l'adresse en mémoire de l'emplacement suivant.



Contrairement à la structure de tableau, on ne peut pas connaître à l'avance l'adresse de l'emplacement de donnée d'indice k d'une liste chaînée. Pour l'obtenir, il faut parcourir les k emplacements précédents en partant de l'emplacement d'indice 0. Ainsi, l'accès à un emplacement dans une liste chaînée se fait en temps linéaire : en effet, il faut faire k opérations élémentaires pour obtenir l'adresse de l'emplacement d'indice k (k lectures).

L'avantage de cette structure par rapport à celle de tableau est qu'elle est dynamique : on peut ajouter un emplacement supplémentaire en modifiant un pointeur !

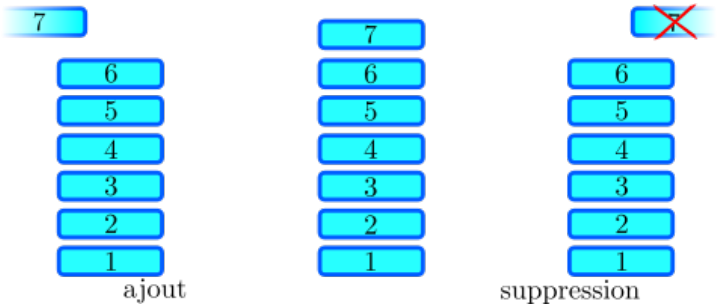
Attention : Malgré son nom, la classe `list` en Python ne correspond pas à une structure de liste chaînée ! Sa structure est beaucoup plus complexe et allie (pour simplifier) les avantages des deux structures linéaires que nous avons vus précédemment.

c. Les piles et les files

Les piles et les files sont des structures de données dynamiques et qui possèdent toutes deux des conditions d'ajout et de suppression des données particulières.

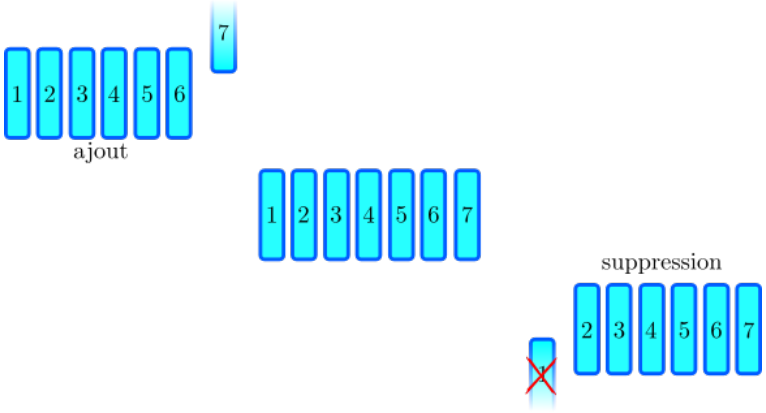
Les piles

L'ajout et la suppression des éléments d'une pile reposent sur le principe du **LIFO** (Last In, First Out) : on peut comparer ce fonctionnement à celle d'une pile d'assiette dans l'évier : la dernière assiette sale ajoutée à la pile sera la première à sortir propre !



Les files

L'ajout et la suppression des éléments d'une file reposent sur le principe du **FIFO** (First In, First Out) : on peut comparer ce fonctionnement à celle d'une file d'attente au supermarché : la premier client arrivé dans la file sera la premier parti !



2. Exercices sur les piles et les files

Dans cette partie, nous allons utiliser les méthodes des listes en Python pour "simuler" des piles et des files. Lors d'un prochain T.P., nous implémenterons directement nos propres de structures de piles et de files grâce à la programmation orientée objet.

a. Rappels sur les listes en Python

On rappelle la liste des méthodes de la classe `list`. On considère un objet `L` de type `list` :

Méthodes qui modifient la liste

- `L.append(x)` : ajoute l'élément `x` en fin de liste ;
- `L.extend(N)` : ajoute la liste `N` en fin de liste ;
- `L.insert(i,x)` : insère l'élément `x` à l'indice `i` - le précédent (avant insertion) élément d'indice `i` et les suivants voit alors leur indice augmenter de 1 ;
- `L.pop(i)` : supprime l'élément d'indice `i` et retourne sa valeur - si aucun indice n'est indiquée en argument, c'est le dernier élément de la liste qui est supprimé et retourné ;
- `L.sort()` : trie la liste par ordre croissant (autant que faire se peut !) ;
- `L.reverse()` : inverse l'ordre des éléments de la liste ;

Méthodes qui ne modifient pas la liste

- `L.index(x)` : retourne l'indice de la première occurrence de `x` dans la liste ;
- `L.count(x)` : retourne le nombre d'occurrences de `x` dans la liste ;

b. Simulation d'une pile

On peut très facilement simuler une structure de pile grâce à Python : il suffit d'utiliser les méthodes `append(x)` pour l'ajout de `x` en "haut" de la pile et la méthode `pop()` (sans argument) pour la suppression de l'élément en "haut" de la pile. Pour l'exercice suivant, on simule la structure de pile, on a donc seulement accès à l'ajout et à la suppression en haut de pile : ainsi, on ne pourra utiliser pour manipuler les données que les méthodes `append` et `pop` (sans argument) ... et les fonctions créées dans les questions d'avant !.

Exercice 1.

On simule ici une pile `P` par une liste Python.

1. Écrire une fonction `Pcreer_pile_vide(P)` qui retourne une pile vide (et donc une liste vide `[]` en fait !)
2. Écrire une fonction `Pest_vide(P)` qui retourne `True` si la pile `P` est vide, `False` sinon.
3. Écrire une fonction `Ppeek(P)` qui retourne le dernier élément de la pile `P` sans modifier celle-ci.
4. Écrire une fonction `Phauteur(P)` qui retourne la hauteur de la pile `P` sans la modifier et sans utiliser `len`.
5. Écrire une fonction `Pinverse(P)` qui retourne une copie de la pile `P` dont les éléments ont été inversés (le haut en bas et le bas en haut !), sans modifier `P`.
6. Écrire une fonction `Pcopie(P)` qui retourne une copie de `P`, sans modifier `P`.
7. Écrire une fonction `Pcirc(P)` qui retourne une copie de la pile `P` dont on a placé le dernier élément tout en bas (en dessous des autres).
8. Écrire une fonction `Phautbas(P)` qui retourne une copie de la pile `P` dont on a placé le dernier élément tout en bas et le premier élément tout en haut.

c. Simulation d'une file

De la même façon, on simule une file grâce aux méthodes `x` pour ajouter `x` en bout de file et `pop(0)` pour supprimer l'élément en tête de file

Exercice 2.

On simule ici une file `F` par une liste Python.

1. Écrire une fonction `Fcreer_file_vide(P)` qui retourne une file vide (et donc une liste vide `[]` en fait !)
2. Écrire une fonction `Fest_vide(F)` qui vérifie si la file `F` est vide.
3. Écrire une fonction `Fpeek(F)` qui retourne le dernier élément de la file `F` sans modifier celle-ci.
4. Écrire une fonction `Flongueur(F)` qui retourne la longueur de la file `F` sans la modifier et sans utiliser `len`.

5. Écrire une fonction `Fcopie(F)` qui retourne une copie de `F`, sans modifier `F`.
6. Écrire une fonction `Finverse(F)` qui retourne une copie de la file `F` dont les éléments ont été inversés, sans modifier `F`.
7. Écrire une fonction `Fcirc(F)` qui retourne une copie de la file `F` dont on a placé le dernier élément tout en tête de file.

d. Parenthésage et pile

Exercice 3.

Le but de cet exercice est de coder une fonction de vérification du parenthésage d'une formule : par exemple, l'expression $(x * y) - z$ est correctement parenthésée alors que $(x * (z - y)$ ne l'est pas.

Écrire une fonction `Parentheses(chn)` qui prend en argument une chaîne de caractères `chn` (qui contiendra les expressions dont le parenthésage est à vérifier, par ex : `'(x-(((y+(z))))'`) et qui renvoie `True` si le parenthésage est correct, `False` sinon.

Dans cette fonction, la chaîne `chn` est analysée caractère par caractère. On génère alors une pile contenant les parenthèses ouvrantes et on compare chaque parenthèse fermante avec le sommet, s'il existe, de la pile.