|  |
| --- |
| Capacités attendues |
| * Spécifier une structure de données par son interface. * Distinguer interface et implémentation. * Écrire plusieurs implémentations d’une même structure de données. * Distinguer des structures par le jeu des méthodes qui les caractérisent. * Choisir une structure de données adaptée à la situation à modéliser. * Distinguer la recherche d’une valeur dans une liste et dans un dictionnaire. |

# Introduction

## Les structures de données abstraites

Les algorithmes opèrent sur des données qui peuvent être de différentes natures. La première version d’un algorithme est autant que possible indépendante d’une implémentation particulière, c’est-à-dire que la représentation des données n’est pas fixée.

A ce premier niveau, les données sont considérées de manière abstraite, on se donne une notation pour les décrire ainsi que l’ensemble des opérations que l’on peut appliquer et les propriétés de ces opérations. On parle alors de type abstrait de données.

Pourquoi avoir recourt à cette notion de type abstrait ? Tout simplement pour définir des types de données indépendant d’un langage de programmation. On les distingue des données de type primitif que sont les entiers, les booléens, les flottants …

Dans ce chapitre, nous étudierons 2 types de structures de données abstraites :

* Les structures linéaires : les listes, les piles et les files
* Les structures à accès par clé : les dictionnaires

Remarque : Ces structures de données peuvent être parfois déjà implémentés dans les langages de programmation comme type de données mais ce n’est pas toujours le cas !

## Interface

Les structures de données possèdent un ensemble de routines (procédures ou fonctions) permettant d’ajouter, d’effacer, d’accéder aux données. Cet ensemble de routines est appelé interface.

L’interface est généralement constituée des 4 routines élémentaire dites CRUD :

* Create : ajout d’une donnée.
* Read : lecture d’une donnée.
* Update : modification d’une donnée.
* Delete : suppression d’une donnée.

A ces opérations se rajoute une autre routine tout aussi importante : la recherche d’une donnée. En effet il faut d’abord trouver la donnée dans la structure avant de pouvoir la lire, la modifier ou la supprimer.

# Les listes

## Définition

Une liste est une structure de donnée qui correspond à une suite finie d’éléments repérés par leur rang (index). Les éléments sont ordonnés et leur place à une grande importance. Une liste est évolutive, on peut ajouter ou supprimer n’importe quel élément.

Voici les opérations qui peuvent être effectuées sur une liste créée :

* INSERER(L, x, i) qui repère la donnée de rang i-1 puis décale de 1 rang à droite tous les éléments situés derrière lui et insère x au rang i.
* SUPPRIMER(L, i) qui repère la donnée de rang i (en commençant par le premier élément) puis décale de 1 rang à gauche tous les éléments situés derrière lui.

## Interface

**Type abstrait :** Liste

**Données :** éléments de type T

**Opérations**

CREER\_LISTE\_VIDE() qui retourne un objet de type Liste.

INSERER(L, e, i) qui repère la donnée de rang i-1 puis décale de 1 rang à droite tous les éléments situés derrière lui et insère x au rang i.

SUPPRIMER(L, i) qui repère la donnée de rang i (en commençant par le premier élément) puis décale de 1 rang à gauche tous les éléments situés derrière lui.

RECHERCHER(L, e) qui cherche l’élément e dans la liste L et retourne son rang.

LIRE(L, i) qui retourne l’élément situé au rang i.

MODIFIER(L, i, e) qui écrase l’élément de rang i et le remplace par l’élément e.

LONGUEUR(L) qui retourne le nombre d’éléments contenus dans la liste L.

**Condition :** Le rang i doit être compris entre 1 et la longueur de la liste L.

* 1. **Exemple**

Prenons la suite d’instructions suivantes :

L = CREER\_LISTE\_VIDE()

INSERER(L,’A’,1)

INSERER(L,’O’,2)

INSERER(L,’B’,1)

INSERER(L,’V’,3)

INSERER(L,’R’,2)

A la fin de ces instructions, la liste L contient 5 élèments de type « caractère » et L = {‘B’, ‘R’, ‘A’, ‘V’, ‘O’}.

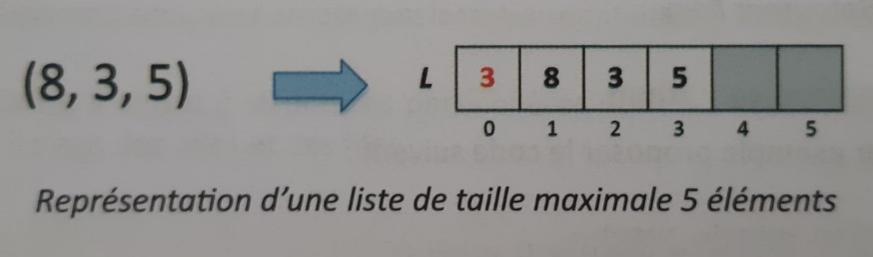
Application

## Implémentation

Il existe plusieurs façons de concevoir une liste. La plus simple consiste à utiliser un tableau de taille fixe dont chaque élément est identifié par son indice. Tous les langages de programmation permettent de réaliser des tableaux. Pour le langage Python, le plus simple est d’utiliser des listes.

On peut par exemple réaliser une liste capable de contenir n éléments avec un tableau L[0 … n] pouvant contenir (n+1) éléments :

* La première case du tableau (d’indice 0) contient le nombre d’éléments présents dans la liste.
* Les cases suivantes du tableau (d’indices 1 à n), contiennent les éléments de la liste ou sont vides.



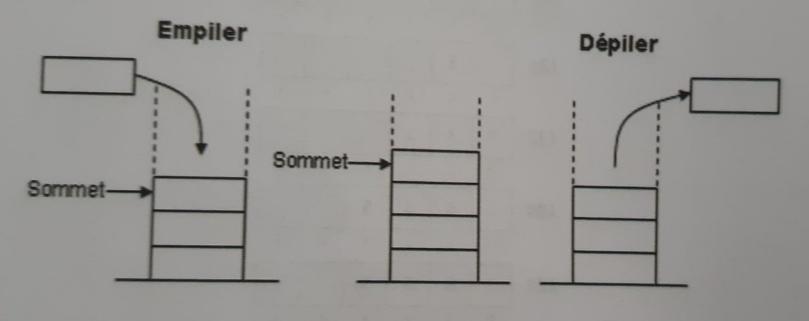
Remarques :

* Si (L[0] == 0) alors la liste est vide. A chaque fois qu’on insère un élément dans la liste, on augmente L[0] d’une unité.
* Lorsque (L[0] == n+1), la liste est pleine. De la même façon, lorsque l’on supprime un élément, on diminue L[0] d’une unité.

Application

# Les piles

## Définition

Il s’agit d’une structure de données qui donne accès en priorité aux dernières données ajoutées. Ainsi, la dernière information ajoutée sera la première à sortir. Autrement dit, on ne peut accéder qu’à l’objet situé au sommet de la pile. On décrit souvent ce comportement par l’expression « dernier entré, premier sorti » ou encore LIFO : Last In, First Out.

Remarque : Le rangement des assiettes convient à cette description. En effet l’ordre dans lequel les assiettes sont dépilées est l’inverse de celui dans lequel elles ont été empilées, puisque seule l’assiette supérieure est accessible.

Les 2 opérations élémentaires que l’on a besoin avec cette structure sont :

* EMPILER(P, x) qui correspond à l’insertion de la donnée x au sommet de la pile P si la pile n’est pas pleine.
* DEPILER(P) qui retire la dernière donnée de la pile P et la retourne si la pile n’est pas vide.

## Interface

**Type abstrait :** Pile

**Données :** éléments de type T

**Opérations**

CREER\_PILE\_VIDE() qui retourne un objet de type Pile.

EMPILER(P, e) qui correspond à l’insertion de la donnée e au sommet de la pile P.

DEPILER(P) qui retire la dernière donnée de la pile P et la retourne si la pile n’est pas vide.

SOMMET(P) qui retourne l’élément qui se trouve au sommet de la pile sans le retirer de la pile.

NB\_ELEMENTS(P) qui retourne le nombre d’élément de la pile.

EST\_VIDE(P) qui retourne un objet de type booléen : True si la pile est vide ou False sinon.

EST\_PLEIN(P) qui retourne un objet de type booléen : True si la pile est pleine ou False sinon.

**Conditions :** DEPILER et SOMMET sont définies si et seulement si la pile P est non vide.

## Exemple

Prenons la suite d’instructions suivantes :

P = CREER\_PILE\_VIDE()

EMPILER(P,3)

EMPILER(P,2)

N = DEPILER(P)

EMPILER(P,5)

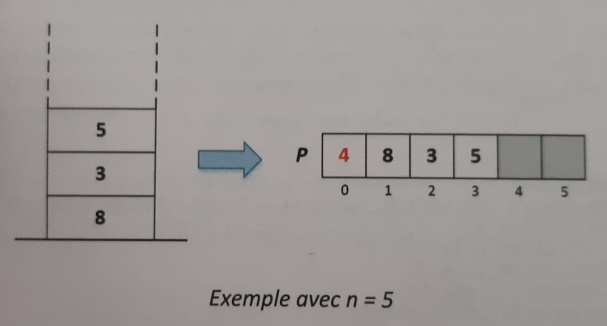
EMPILER(P,7)

EMPILER(P,9)

A la fin de ces instructions, le pile P contient 4 éléments de type « entier » et P = (9,7,5,3) et N = 2.

Application

## Implémentation

On peut aussi représenter une pile capable de contenir n éléments avec un tableau P[0 … n] pouvant contenir (n+1) éléments :

* La première case du tableau (d’indice 0) contient l’indice de la prochaine case vide (c’est l’indice qui correspondra au prochain élément à insérer dans la pile).
* Les cases suivantes du tableau (d’indices 1 à n), contiennent les éléments de la pile ou sont vides.

La dernière case non vide du tableau est le sommet de la pile.

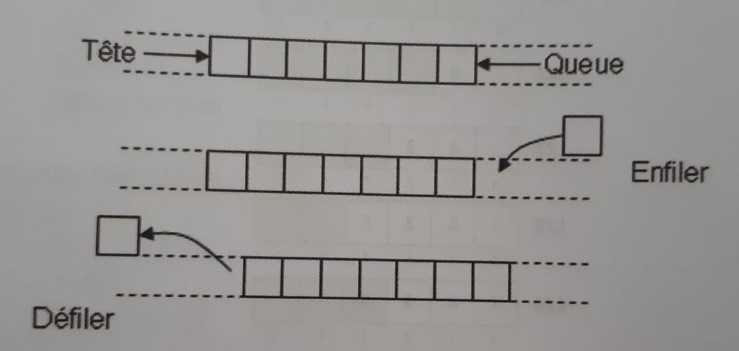
Remarques :

* Si (P[0] == 1) alors la pile est vide. A chaque fois qu’on insère un élément, on augment P[0] d’une unité.
* Lorsque (P[0]==n+1), la pile est pleine.

# Les files

## Définition

Une file est une structure de données dans laquelle on accède aux éléments suivant la règle du premier arrivé premier sorti. Autrement dit, on ne peut accéder qu’à l’objet situé au début de la file. On décrit souvent ce comportement par l’expression « premier entré, premier sorti » ou encore FIFO : First In, First Out. La file comporte une tête et une queue.



Remarque : La file de client qui attend à un guichet ou à une caisse qui convient à cette description. En effet le client qui passe en premier est celui qui est arrivé le premier.

Les 2 opérations élémentaires que l’on a besoin avec cette structure sont :

* ENFILER(F, x) qui correspond à l’insertion de la donnée x à la queue de la file F.
* DEFILER(F) qui retire la donnée de tête de la file F et la retourne si la file n’est pas vide.

## Interface

**Type abstrait :** File

**Données :** éléments de type T

**Opérations**

CREER\_FILE\_VIDE() qui retourne un objet de type File.

ENFILER(F, e) qui insère la donnée e en queue de la file F.

DEFILER(P) qui retire la donnée de tête de la file F et la retourne si la file n’est pas vide.

EST\_VIDE(P) qui retourne un objet de type booléen : True si la file est vide ou False si la file n’est pas vide.

**Condition :** DEFILER est définie si et seulement si la file F est non vide.

## Exemple

Prenons la suite d’instructions suivantes :

F = CREER\_FILE\_VIDE()

ENFILER(F, 21)

ENFILER(F, 22)

ENFILER(F, 23)

N = DEFILER(F)

ENFILER(F, 24)

ENFILER(F, 25)

N = DEFILER(F)

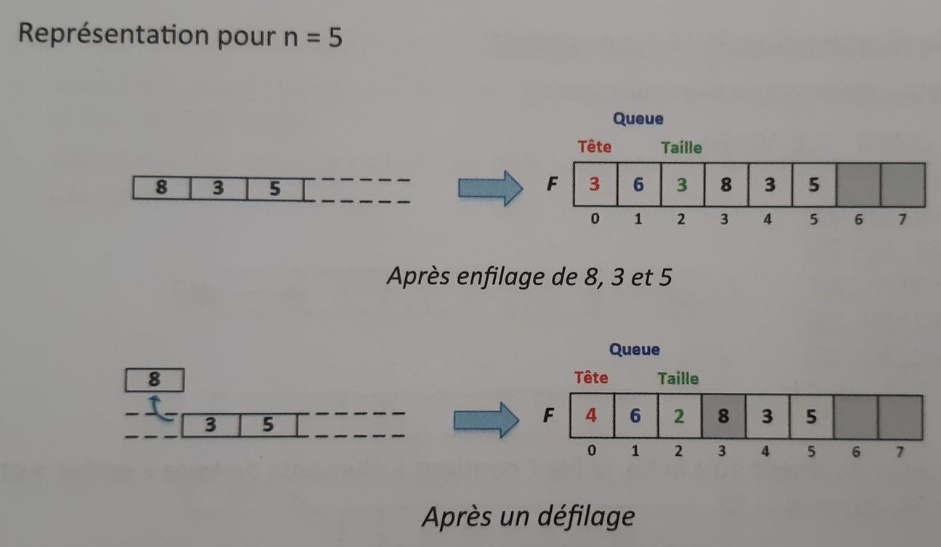
A la fin de ces instructions, la file F contient 3 éléments et F = (23, 24, 25) et N = 22.

Application

## Implémentation

On peut représenter une file capable de contenir n éléments avec un tableau F[0 … (n+2)] avec (n+3) éléments :

* La première case du tableau (d’indice 0) contient l’indice de la tête de la file.
* La deuxième case du tableau (d’indice 1) contient l’indice de la queue de la file, c’est-à-dire de la prochaine case disponible pour la queue.
* La troisième case du tableau (d’indice 2) contient le nombre d’éléments présents dans la file.
* Les cases suivantes du tableau (d’indice 3 à n+2) contiennent les éléments de la file ou sont vides.



Remarques :

* Si (taille == 0) la file est vide et si (taille == n) la file est pleine.
* A chaque fois qu’on enfile un élément, on augmente la taille d’une unité ainsi que la queue.
* A chaque fois qu’on défile un élément, on diminue la taille d’une unité et on augmente la tête d’une unité.
* Dès que les indices de tête ou de queue dépasse la longueur du tableau, ils repartent au début du tableau (on appelle cela un tableau avec une gestion circulaire).

# Les dictionnaires

* 1. **Définition**

Un dictionnaire est une structure de données qui permet d’associer une valeur à une clé. Cette clé peut être un mot ou un entier. L’ensemble clé-valeur est appelée entrée. On peut définir par exemple une structure dictionnaire pour un annuaire téléphonique en posant 2 clés : nom et numéro de téléphone. On pourrait ainsi avoir :

Nom : Toto

Numéro de téléphone : 06 12 34 56 78

On peut rappeler que les dictionnaires ne comportent pas d’ordre. Contrairement aux listes, on ne peut pas retrouver un élément via sa position, mais uniquement via sa clé. Il s’agit d’une structure de données très commune et très pratique.

Remarque : Cette structure de données est appelée dictionnaire car dans un dictionnaire, on associe un mot (la clé) à sa définition (la valeur).

Il faut noter que Python possède nativement cette structure de données, ce qui n’est pas forcément vrai dans d’autres langages de programmation. Il faut alors l’implémenter si on souhaite l’utiliser.

* 1. **Interface**

**Type abstrait :** Dictionnaire

**Données :** éléments de type T

**Opérations**

CREER\_DICO\_VIDE() qui retourne un objet de type Dictionnaire.

INSERER(D, cle, valeur) qui insère l’entrée clé-valeur dans le dictionnaire D.

SUPPRIMER(D, cle) qui supprime l’entrée dont la clé est cle.

LIRE(D, cle) qui retourne un objet de type T correspond à l’entrée dont la clé est cle.

RECHERCHER(D, cle) qui retourne un objet de type Booléen : retourne True s’il existe une entrée dont la clé est cle dans le dictionnaire D et retourne False sinon.

**Conditions :** LIRE(L, cle) et SUPPRIMER(L, cle) sont défini si et seulement si RECHERCHER(D, cle) = True.

* 1. **Implémentation**

L’implémentation des dictionnaires en Python a déjà été vu en classe de première. L’exemple ci-après montre un dictionnaire constitué de 3 entrées.

