Séquence 7 – Structures de données linéaire : Liste, Pile, File

Objectifs

1. Distinguer Listes, Piles et Files par le jeu des méthodes qui les caractérisent.
2. Distinguer les modes LIFO et FIFO des Piles et des Files
3. Choisir une structure de données adaptée à la situation à modéliser.
4. Distinguer la recherche d'une valeur dans une liste et dans un dictionnaire.
5. Spécifier une structure de données par son interface.
6. Distinguer interface et implémentation.
7. Écrire plusieurs implémentations d'une même structure de données.

Cette séquence s'appuie sur :

- https://pixees.fr/informatiquelycee/n_site/nsi_term_structDo_liste.html
- https://www.maxicours.com/se/cours/utiliser-une-pile-pour-evaluer-une-notation-en-polonais-inverse/

1 Types abstraits

De nombreux algorithmes "classiques" manipulent des structures de données plus complexes que des simples nombres.
Indépendamment de tous langages de programmation, chaque type abstrait de structure de données est défini par son interface.

L' interface .....

.....

L'implémentation, ...

.....

.....

Nous allons ici voir quelques-unes de ces structures de données : les listes, les piles et les files. Ces trois types abstraits de structures sont qualifiés de linéaires.

Lire livre P. 94

2 Les listes

Une liste est une structure de données permettant de regrouper des données. Une liste L est composée de 2 parties : sa tête (souvent noté `car`), qui correspond au dernier élément ajouté à la liste, et sa queue (souvent noté `cdr`) qui correspond au reste de la liste. Le langage de programmation Lisp (inventé par John McCarthy en 1958) a été un des premiers langages de programmation à introduire cette notion de liste (Lisp signifie "list processing"). Voici son interface (ou opérations qui peuvent être effectuées) :

- ..... une liste vide (`L=vide()` on a créé une liste `L` vide)
- ..... si une liste est vide (`estVide(L)` renvoie vrai si la liste `L` est vide)
- ..... un élément en tête de liste (`ajouteEnTete(x,L)` avec `L` une liste et `x` l'élément à ajouter)
- ..... la tête `x` d'une liste `L` et renvoyer cette tête `x` (`supprEnTete(L)`)
- ..... le nombre d'éléments présents dans une liste (`compte(L)` renvoie le nombre d'éléments présents dans la liste `L`)
- La fonction `cons` permet d'obtenir une nouvelle liste à partir d'une liste et d'un élément (`L1 = cons(x,L)`). Il est possible "d'enchaîner" les `cons` et d'obtenir ce genre de structure : `cons(x, cons(y, cons(z,L)))`

Exemples :
Voici une série d'instructions (les instructions ci-dessous s'enchaînent):

- `L=vide()` => on a créé une liste vide
- `estVide(L)` => renvoie vrai
- `ajoutEnTete(3,L)` => La liste L contient maintenant l'élément 3
- `estVide(L)` => renvoie faux
- `ajoutEnTete(5,L)` => la tête de la liste L correspond à 5, la queue contient l'élément 3
- `ajoutEnTete(8,L)` => la tête de la liste L correspond à 8, la queue contient les éléments 3 et 5

- `t = supprEnTete(L)` => la variable t vaut 8, la tête de L correspond à 5 et la queue contient l'élément 3
- `L1 = vide()` => on a créé une autre liste vide
- `L2 = cons(8, cons(5, cons(3, L1)))` => La tête de L2 correspond à 8 et la queue contient les éléments 3 et 5

A faire vous même 1. Manipulation de l' interface

Voici une série d'instructions (les instructions ci-dessous s'enchaînent), expliquez ce qui se passe à chacune des étapes :

- `L = vide()`
- `ajoutEnTete(10,L)`
- `ajoutEnTete(9,L)`
- `ajoutEnTete(7,L)`
- `L1 = vide()`
- `L2 = cons(5, cons(4, cons(3, cons (2, cons(1, cons(0,L1))))))`

A faire vous même 2. Implémentation

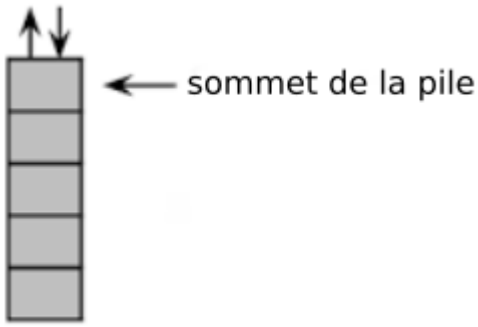
- Créez un objet python qui se nommera `nœud`. Cet objet aura 2 attributs : `valeur` (un entier) et `noeud_suivant` (un objet de type `nœud`). Ces attributs sont passés en paramètre à la création et peuvent être égale à `None` par défaut.
- Créez un objet python qui se nommera `liste_abstraite`. Cet objet aura 1 attribut : `noeud` (un objet de type `noeud`). Cet attribut est passé en paramètre à la création et peut être égale à `None` par défaut.
- Développez une fonction python nommée `vide` qui créé et retourne une `liste_abstraite` vide.
- Développez une fonction python nommée `est_vide` qui retourne `True` si la liste passée en paramètre est vide sinon `False`.
- Développez une fonction python nommée `ajout_en_tete` (voir ci-dessus) qui ajoute un nœud en tête de liste (attention au cas où la liste n'est pas vide).
- Développez une fonction python nommée `suppr_en_tete`
- Développez une fonction python nommée `cons` conforme à la description ci-dessus.
- Développez une fonction python nommée `compte` conforme à la description ci-dessus.
- Avec ce kit, reproduisez les instructions du A faire vous-même 1

ACTIVITE AUTOUR DU JEU LES TOURS DE HANOI :

http://championmath.free.fr/tourhanoi.htm

3 Les piles (ou stack)

On retrouve dans les piles une partie des propriétés vues sur les listes. Dans les piles, il est uniquement possible de manipuler le dernier élément introduit dans la pile. On prend souvent l'analogie avec une pile d'assiettes : dans une pile d'assiettes la seule assiette directement accessible et la dernière assiette qui a été déposée sur la pile.



Les piles .....

.....

.....

.....

Voici les opérations que l'on peut réaliser sur une pile, on peut :

- ..... si une pile est vide (`pile_vide`)
- ..... un nouvel élément sur la pile (`push`)
- ..... l'élément au sommet de la pile tout en le supprimant. On dit que l'on dépile (`pop`)
- ..... à l'élément situé au sommet de la pile sans le supprimer de la pile (`sommet`)
- ..... le nombre d'éléments présents dans la pile (`taille`)

Exemples :
Soit une pile `P` composée des éléments suivants : 12, 14, 8, 7, 19 et 22 (le sommet de la pile est 22)
Pour chaque exemple ci-dessous on repart de la pile d'origine :

- `pop(P)` renvoie 22 et la pile `P` est maintenant composée des éléments suivants : 12, 14, 8, 7 et 19 (le sommet de la pile est 19)
- `push(P,42)` la pile `P` est maintenant composée des éléments suivants : 12, 14, 8, 7, 19, 22 et 42
- `sommet(P)` renvoie 22, la pile P n'est pas modifiée
- si on applique `pop(P)` 6 fois de suite, `pile_vide(P)` renvoie vrai
- Après avoir appliqué `pop(P)` une fois, `taille(P)` renvoie 5

A faire vous même 3. Manipulation

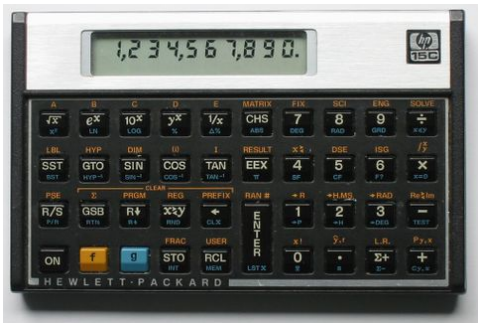
Soit une pile `P` composée des éléments suivants : 15, 11, 32, 45 et 67 (le sommet de la pile est 67). Quel est l'effet de l'instruction `pop(P)` ?

A faire vous même 4.

P. 104 ex 5 : Créer une classe `Stack` en vous basant sur l'objet `liste_abstraite`. Suivez les instructions données dans le livre. Si besoin, complétez cette classe et développez les fonctions décrites ci-dessus. N'oubliez pas de renseigner les docstring avec, notamment, un(des) jeu(x) de tests.

3.1 LA NOTATION EN POLONAIS INVERSE

Il y a 35 ans, HP commercialisait des calculatrices qui utilisait la notation polonaise inverse.



3.1.1 Les notations infixée, préfixée et postfixée

Pour écrire des expressions algébriques, on peut utiliser différentes notations : infixée (ou infixe), préfixée (ou préfixe) ou postfixée (ou postfixe).
Les notations infixée, préfixée et postfixée d'une expression algébrique se distinguent par la position que prennent les opérateurs et les opérandes.

Vocabulaire :
Une expression algébrique est une suite de chiffres avec des opérations du type +, –, *, / ou ^ (pour les puissances).
Un opérateur est un symbole qui représente une opération.

Remarque :
Les opérateurs sont +, –, *, /, ** ou ^ (pour la puissance).
Un opérande est un nombre.

Exemple :
4 + 5 * 3 est une expression algébrique.
Les opérateurs sont + et *. Les opérandes sont 4, 5 et 3.

La notation infixée
La notation infixée est celle qu'on utilise couramment en mathématiques. On l'utilise lorsqu'on réalise des calculs. Voir exemple ci-dessus.

La notation préfixée
Dans la notation préfixée, on écrit l'opérateur puis les opérandes.
Exemples

- `+ … 3 … 4`
- `+ … 1 … 2 … × … 3 … ^ … 4`

La notation postfixée
Dans la notation postfixée, on écrit les opérandes puis les opérateurs.
Exemples

- `3 … 4 … +`
- `1 … 2 … + … 3 … × … 4 …^`

3.1.2 La notation en polonais inverse

La notation en polonais inverse, en abrégé NPI (RPN en anglais pour Reverse Polish Notation) est un autre type de notation, qui est qualifié de postfixé. Cette notation permet d'écrire de façon non ambiguë les calculs, sans parenthèses et en utilisant des piles.

Rappel
Une pile est un type abstrait de structure de données, dans lequel on empile et dépile les données : les dernières données ajoutées à une pile sont les premières à sortir.

Principe
On note `a … b … operateur` dans la notation en polonais inverse, avec `a` et `b` deux opérandes.

Exemples
- Le calcul classique (infixée) `3 … + … 4` se note `3 … 4 … +` en NPI.
- Le calcul classique (infixée) `( ( 1 … + … 2) … × … 3 )`^{4} se note `1 … 2 … + … 3 … × … 4 … ^` en NPI.

Remarques
- Le but n'est pas ici d'effectuer la conversion de la notation infixée vers la notation postfixée.
- L'algorithme qui permet cette conversion se nomme l'algorithme de Shunting-yard ou algorithme de la gare de triage.

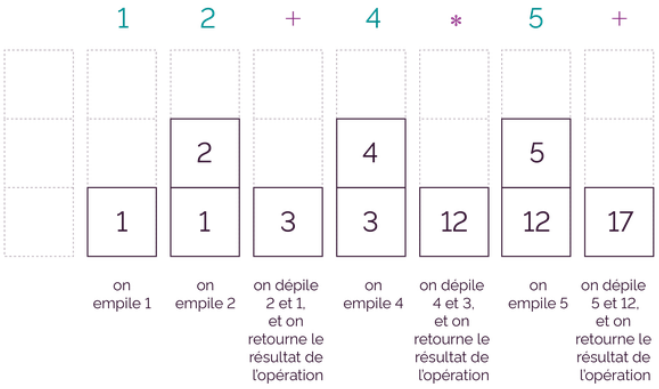
3.1.3 L'évaluation d'une expression postfixée (NPI)
À l'aide d'un algorithme, on souhaite évaluer une expression qui a été conçue avec une notation postfixée telle que le polonais inverse (NPS).
Évaluer une expression postfixée consiste à déterminer la valeur décimale de cette expression.

Principe de l'algorithme
Pour évaluer une expression postfixée, c'est-à-dire obtenir la valeur décimale de cette expression, l'algorithme commence par lire un par un les caractères de l'expression postfixée.
- Si le caractère lu est un opérande (nombre), alors on l'empile en haut de la pile.
- Si le caractère lu est un opérateur (symbole mathématique), alors :
  - on dépile les deux opérandes qui se trouvent en haut de la pile ;
  - on calcule le résultat en appliquant l'opérateur sur les deux opérandes dépilés ;
  - et on empile le résultat en haut de la pile.

Remarque
On empile les opérandes et non les opérateurs.
Une fois tous les caractères lus, la pile ne contient qu'un seul élément qui correspond au résultat final.

Exemple
On souhaite évaluer l'expression postfixée `1 … 2 … + … 4 … * … 5 … +`, c'est-à-dire obtenir sa valeur décimale.

On lit un à un les caractères de l'expression postfixée `1 … 2 … + … 4 … * … 5 … +` pour l'évaluer.
- Les opérateurs sont les éléments `+` et `*`.
- Les opérandes sont les éléments `1`, `2`, `4` et `5`.
- Le premier opérateur `+` permet de dépiler les deux opérandes `1` et `2`, et d'empiler le résultat de l'opération `1 + 2`.

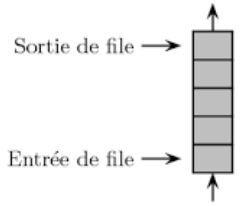


La pile associée à l'expression postfixée 1 2 + 4 * 5 + renverra la valeur décimale 17.

A faire vous même 5. Une calculatrice virtuelle en NPI
Installez le paquet : grpn
Lancez ce programme et effectuez les deux calculs suivants :
- $3+4$
- $((1+2)\times 3)^4$

A faire vous même 6.
- P. 105 ex 11

4 Les files (ou queue)
Comme les piles, les files ont des points communs avec les listes. Différences majeures : dans une file on ajoute des éléments à une extrémité de la file et on supprime des éléments à l'autre extrémité. On prend souvent l'analogie de la file d'attente devant un magasin pour décrire une file de données.



Les files sont basées .....

.....

.....

Voici les opérations que l'on peut réaliser sur une file, on peut :

- ..... si une file est vide (`file_vide`)
- ..... un nouvel élément à la file (`ajout`)
- ..... l'élément situé en bout de file tout en le supprimant (`retire`)
- ..... à l'élément situé en bout de file sans le supprimer de la file (`premier`)
- ..... le nombre d'éléments présents dans la file (`taille`)

Exemples :
Soit une file F composée des éléments suivants : 12, 14, 8, 7, 19 et 22 (le premier élément rentré dans la file est 22 ; le dernier élément rentré dans la file est 12). Pour chaque exemple ci-dessous on repart de la file d'origine :

- `ajout(F,42)` la file `F` est maintenant composée des éléments suivants : 42, 12, 14, 8, 7, 19 et 22 (le premier élément rentré dans la file est 22 ; le dernier élément rentré dans la file est 42)
- `retire(F)` la file `F` est maintenant composée des éléments suivants : 12, 14, 8, 7, et 19 (le premier élément rentré dans la file est 19 ; le dernier élément rentré dans la file est 12)
- `premier(F)` renvoie 22, la file F n'est pas modifiée
- si on applique `retire(F)` 6 fois de suite, `file_vide(F)` renvoie vrai
- Après avoir appliqué `retire(F)` une fois, `taille(F)` renvoie 5.

A faire vous même 7.

Soit une file F composée des éléments suivants : 1, 12, 24, 17, 21 et 72 (le premier élément rentré dans la file est 72 ; le dernier élément rentré dans la file est 1). Quel est l'effet de l'instruction `ajout(F,25)` ?

A faire vous même 8.

- S' inspirer de P. 104 ex 5 déjà fait et créer une classe `file`
- En vous basant sur l'objet `liste_abstraite`, développez l'objet file et les fonctions décrites ci-dessus.
- N'oubliez pas de renseigner les docstring avec, notamment, un jeu de test.

5 Types abstraits et représentation concrète des données

Les listes, les piles ou les files sont des "vues de l'esprit" présent uniquement dans la tête des informaticiens, on dit que ce sont des types abstraits de données (ou plus simplement des types abstraits).
Nous avons évoqué ci-dessus la manipulation des types de données (liste, pile et file) par des algorithmes, mais, au-delà de la beauté intellectuelle de réfléchir sur ces algorithmes, le but de l'opération est souvent, à un moment ou un autre, de "traduire" ces algorithmes dans un langage compréhensible pour un ordinateur (Python, Java, C,...). On dit alors que l'on implémente un algorithme.

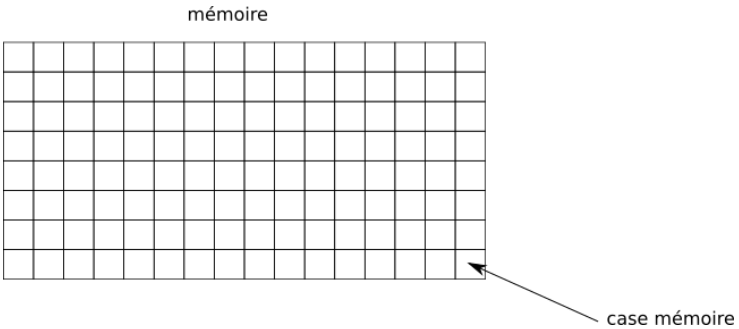
Il est donc aussi nécessaire d'implémenter les types de données comme les listes, les piles ou les files afin qu'ils soient utilisables par les ordinateurs.

L'implémentation d'un type de données dépend du langage de programmation. Il faut, quel que soit le langage utilisé, que le programmeur retrouve les fonctions qui ont été définies pour le type abstrait. Certains types abstraits ne sont pas forcément implémentés dans un langage donné, si le programmeur veut utiliser ce type abstrait, il faudra qu'il le programme par lui-même en utilisant les "outils" fournis par son langage de programmation.
Pour implémenter les listes (ou les piles et les files), beaucoup de langages de programmation utilisent 2 structures : les tableaux et les listes chaînées.

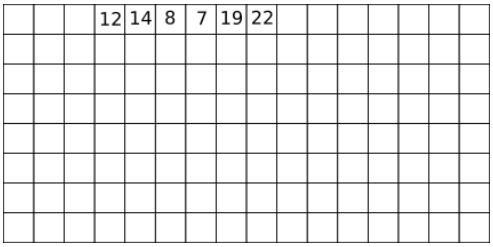
5.1 Les tableaux

5.1.1 Tableaux fixes

Un tableau est une suite contiguë de cases mémoires (les adresses des cases mémoire se suivent) :



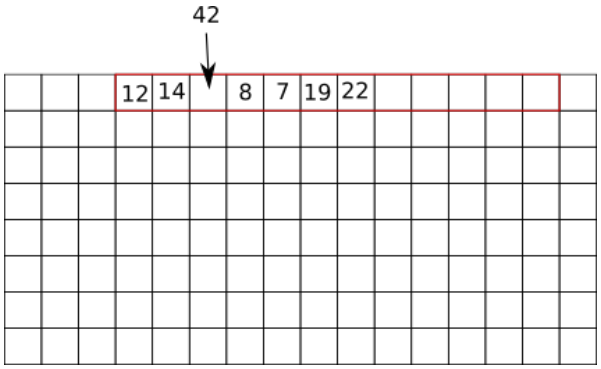
Le système réserve une plage d'adresse mémoire afin de stocker des éléments.



La taille d'un tableau est fixe : une fois que l'on a défini le nombre d'éléments que le tableau peut accueillir, il n'est pas possible modifier sa taille. Si l'on veut insérer une donnée, on doit créer un nouveau tableau plus grand et déplacer les éléments du premier tableau vers le second tout en ajoutant la donnée au bon endroit !

5.1.2 Tableaux dynamiques

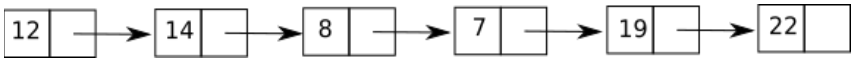
Dans certains langages de programmation, on trouve une version "évoluée" des tableaux : les tableaux dynamiques. Les tableaux dynamiques ont une taille qui peut varier. Il est donc relativement simple d'insérer des éléments dans le tableau. Ce type de tableaux permet d'implémenter facilement le type abstrait liste (de même pour les piles et les files)
À noter que les "listes Python" (listes Python) sont des tableaux dynamiques. Attention de ne pas confondre avec le type abstrait liste défini ci-dessus, ce sont de "faux amis".



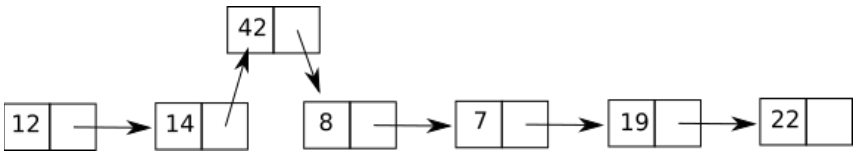
5.2 Les listes chaînées

Autre type de structure que l'on rencontre souvent et qui permet d'implémenter les listes, les piles et les files : les listes chaînées.

Dans une liste chaînée, à chaque élément de la liste on associe 2 cases mémoire : la première case contient l'élément et la deuxième contient l'adresse mémoire de l'élément suivant.



Il est relativement facile d'insérer un élément dans une liste chaînée :



6 Tableaux associatif

Nous allons maintenant étudier un autre type abstrait de données : les dictionnaires aussi appelés tableau associatif. On retrouve une structure qui ressemble, à première vue, beaucoup à un tableau (à chaque élément on associe un indice de position). Mais au lieu d'associer chaque élément à un indice de position, dans un dictionnaire, on associe chaque élément (on parle de valeur dans un dictionnaire) à une clef, on dit qu'un dictionnaire contient des couples clef:valeur (chaque clé est associée à une valeur).

Exemples de couples clé:valeur
`prenom:Kevin, nom:Durand, date-naissance:17-05-2005.`
`prenom`, `nom` et `date` sont des clés ; `Kevin`, `Durand` et `17-05-2005` sont des valeurs.
Voici les opérations que l'on peut effectuer sur le type abstrait dictionnaire :

- `ajout` : on associe une nouvelle valeur à une nouvelle clé
- `modif` : on modifie un couple clé:valeur en remplaçant la valeur courante par une autre valeur (la clé restant identique)

- `suppr` : on supprime une clé (et donc la valeur qui lui est associée)
- `rech` : on recherche une valeur à l'aide de la clé associée à cette valeur.

Exemples :

Soit le dictionnaire `D` composé des couples clé:valeur suivants => `prenom:Kevin`, `nom:Durand`, `date-naissance:17-05-2005`.

Pour chaque exemple ci-dessous on repart du dictionnaire d'origine :

- `ajout(D,tel:06060606)` ; le dictionnaire `D` est maintenant composé des couples suivants : `prenom:Kevin, nom:Durand, date-naissance:17-05-2005, tel:06060606`
- `modif(D,nom:Dupont)` ; le dictionnaire `D` est maintenant composé des couples suivants : `prenom:Kevin, nom:Dupont, date-naissance:17-05-2005`
- `suppr(D,date-naissance)` ; le dictionnaire `D` est maintenant composé des couples suivants : `prenom:Kevin, nom:Durand`
- `rech(D,prenom)` ; la fonction retourne `Kevin`

L'implémentation des dictionnaires dans les langages de programmation peut se faire à l'aide des tables de hachage. Les tables de hachages ainsi que les fonctions de hachages qui sont utilisées pour construire les tables de hachages, ne sont pas au programme de NSI. Cependant, l'utilisation des fonctions de hachages est omniprésente en informatique, il serait donc bon, pour votre "culture générale informatique", de connaître le principe des fonctions de hachages.

A faire vous même 9. Pour les plus motivés

Lire ce texte qui vous permettra de comprendre le principe des fonctions de hachages : c'est quoi le hachage .

A faire vous même 10. Pour les plus motivés

Pour avoir quelques idées sur le principe des tables de hachages, je vous recommande le visionnage de cette vidéo : wandida : les tables de hachage

Si vous avez visionné la vidéo de wandida, vous avez déjà compris que l'algorithme de recherche dans une table de hachage a une complexité `O(1)` (le temps de recherche ne dépend pas du nombre d'éléments présents dans la table de hachage), alors que la complexité de l'algorithme de recherche dans un tableau non trié est `O(n)`.
Comme l'implémentation des dictionnaires s'appuie sur les tables de hachage, on peut dire que l'algorithme de recherche d'un élément dans un dictionnaire a une complexité `O(1)` alors que l'algorithme de recherche d'un élément dans un tableau non trié a une complexité `O(n)`.
Python propose une implémentation des dictionnaires, nous avons déjà étudié cette implémentation l'année dernière, n'hésitez pas à vous référer à la ressource proposée l'an passé : les dictionnaires en Python

7 Structure de données Python

Lire P. 98 du livre.
Attention ! Deux types n'ont pas encore été abordés : le type `set` (ensemble). Et le type `collections.deque` (file).

A faire vous même 11.

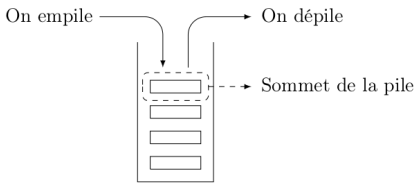
- Faire QCM P. 102 ex 1
- Faire QCM P. 102 ex 2
- Faire QCM P. 102 ex 3
- Faire QCM P. 102 ex 4

A faire vous même 12.

- P. 106 ex 10

8 Exercice type bac : Des piles et des crêpes

On rappelle qu'une pile est une structure de données abstraite fondée sur le principe « dernier arrivé, premier sorti » :



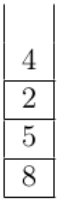
Une pile

On munit la structure de données Pile de quatre fonctions primitives définies dans le tableau ci-dessous. :

```
Structure de données abstraite : Pile
Utilise : Éléments, Booléen
Opérations :
— creer_pile_vide : ∅ -→ Pile
    creer_pile_vide() renvoie une pile vide

— est_vide : Pile -→ Booléen
    est_vide(pile) renvoie True si pile est vide, False sinon

— empiler : Pile, Élément -→ Rien
    empiler(pile, element) ajoute element au sommet de la pile

— depiler : Pile -→ Élément
    depiler(pile) renvoie l'élément au sommet de la pile en le retirant de la pile
```

8.1 Manipulation de pile

On suppose dans cette question que le contenu de la pile `P` est le suivant (les éléments étant empilés par le haut) :



Q1. Quel sera le contenu de la pile `Q` après exécution de la suite d'instructions suivante ?

```
Q = creer_pile_vide()
while not est_vide(P):
        empiler(Q, depiler(P))
```

On appelle *hauteur* d'une pile le nombre d'éléments qu'elle contient. La fonction `hauteur_pile` prend en paramètre une pile `P` et renvoie sa hauteur. Après appel de cette fonction, la pile `P` doit avoir retrouvé son état d'origine.

Exemple : si `P` est la pile de la question 1 : `hauteur_pile(P) = 4`.

Q2. Compléter le programme Python suivant implémentant la fonction `hauteur_pile` en remplaçant les ... aux lignes 5, 9 et 11 par les bonnes instructions.

```
1. def hauteur_pile(P):
2.     Q = creer_pile_vide()
3.     n = 0
4.     while not (est_vide(P)):
5.         ...
6.         x = depiler(P)
7.         empiler (Q, x)
8.     while not (est_vide(Q)):
9.         ...
10.                empiler (P, x)
11.        return ...
```

Q3. Créer une fonction `max_pile` ayant pour paramètres une pile `P` et un entier `i`. Cette fonction renvoie la position `j` de l'élément maximum parmi les `i` derniers éléments empilés de la pile `P`.
Après appel de cette fonction, la pile `P` devra avoir retrouvé son état d'origine. La position du sommet de la pile est 1.
Exemple : si P est la pile de la question Q1 : `max_pile(P, 2) = 1`

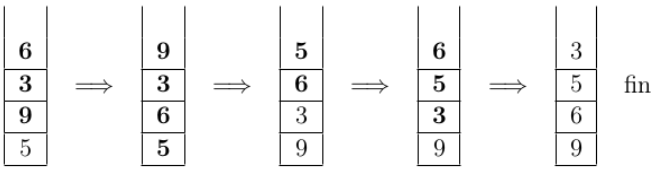
Q4. Créer une fonction retourner ayant pour paramètres une pile `P` et un entier `j`. Cette fonction inverse l'ordre des `j` derniers éléments empilés et ne renvoie rien. On pourra utiliser deux piles auxiliaires.

Exemple : si `P` est la pile de la question Q1, après l'appel de `retourner(P, 3)`, l'état de la pile `P` sera :

8.2 L'objectif de cette partie est de trier une pile de crêpes.

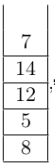
On modélise une pile de crêpes par une pile d'entiers représentant le diamètre de chaque crêpe. On souhaite réordonner les crêpes de la plus grande (placée en bas de la pile) à la plus petite (placée en haut de la pile).
On dispose uniquement d'une spatule que l'on peut insérer dans la pile de crêpes de façon à retourner l'ensemble des crêpes qui lui sont au-dessus.
Le principe est le suivant :
— On recherche la plus grande crêpe.
— On retourne la pile à partir de cette crêpe de façon à mettre cette plus grande crêpe tout en haut de la pile.
— On retourne l'ensemble de la pile de façon à ce que cette plus grande crêpe se retrouve tout en bas.
— La plus grande crêpe étant à sa place, on recommence le principe avec le reste de la pile.

Exemple :



Q5. Créer la fonction `tri_crepes` ayant pour paramètre une pile `P`. Cette fonction trie la pile `P` selon la méthode du tri crêpes et ne renvoie rien. On utilisera les fonctions créées dans les questions précédentes.

Exemple :

Si la pile `P` est 

, après l'appel de `tri_crepes(P)`, la pile `P` devient : 

.