# Structure de pile

## *De l'abstrait et du concret*

$\psi$

stephane@gonnord.org - http://blog.psi945.fr

Lundi 28 nov. et 5 déc. 2016

# Plan

# Sommaire

# Types et structures de données

- Des types :
  - les entiers de Python 2...

# Types et structures de données

- Des types :
  - les entiers de Python 2... et ceux de Python 3 ; et ceux de C, de C++, de Java...

# Types et structures de données

- Des types :
  - les entiers de Python 2... et ceux de Python 3 ; et ceux de C, de C++, de Java...
  - les flottants de Python, de C...

# Types et structures de données

- Des types :
  - les entiers de Python 2... et ceux de Python 3 ; et ceux de C, de C++, de Java...
  - les flottants de Python, de C...
  - les tableaux de Python, ceux en C, C++, etc.

# Types et structures de données

- Des types :
  - les entiers de Python 2... et ceux de Python 3 ; et ceux de C, de C++, de Java...
  - les flottants de Python, de C...
  - les tableaux de Python, ceux en C, C++, etc.
- Des structures de données :
  - piles ;
  - arbres (binaires de recherche) ;
  - files ;
  - tas ;
  - dictionnaires ;
  - ...

# Types et structures de données

- Des types :
  - les entiers de Python 2... et ceux de Python 3 ; et ceux de C, de C++, de Java...
  - les flottants de Python, de C...
  - les tableaux de Python, ceux en C, C++, etc.
- Des structures de données :
  - piles ;
  - arbres (binaires de recherche) ;
  - files ;
  - tas ;
  - dictionnaires ;
  - ...
- Niveaux d'abstraction différents.

# Différents besoins, différentes structures

Une structure de données répond à des besoins particuliers, spécifiques... pour « ranger » des données.

- Ajout, récupération des données les plus récentes : PILES.

# Différents besoins, différentes structures

Une structure de données répond à des besoins particuliers, spécifiques... pour « ranger » des données.

- Ajout, récupération des données les plus récentes : PILES.
- Ajout, récupération des données les plus vieilles : FILES.

# Différents besoins, différentes structures

Une structure de données répond à des besoins particuliers, spécifiques... pour « ranger » des données.

- Ajout, récupération des données les plus récentes : PILES.
- Ajout, récupération des données les plus vieilles : FILES.
- Ajout, suppression, consultation de données *ordonnées* : ARBRES BINAIRES DE RECHERCHE.

# Différents besoins, différentes structures

Une structure de données répond à des besoins particuliers, spécifiques... pour « ranger » des données.

- Ajout, récupération des données les plus récentes : PILES.
- Ajout, récupération des données les plus vieilles : FILES.
- Ajout, suppression, consultation de données *ordonnées* : ARBRES BINAIRES DE RECHERCHE.
- Ajout, suppression du plus gros(/petit) élément parmi des données *ordonnées* :

# Différents besoins, différentes structures

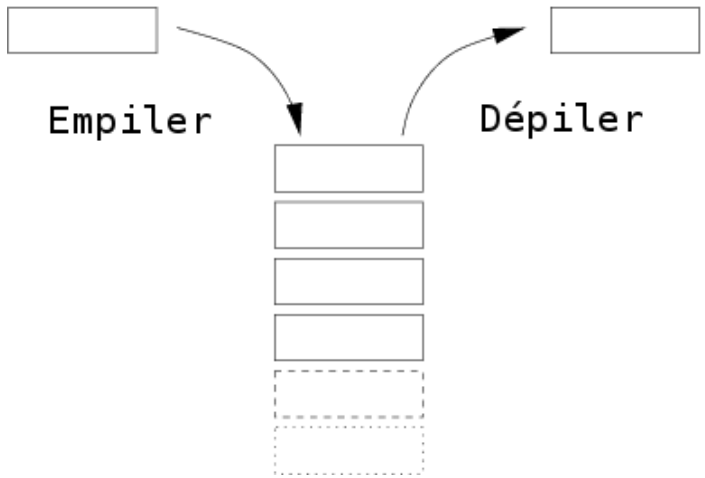
Une structure de données répond à des besoins particuliers, spécifiques... pour « ranger » des données.

- Ajout, récupération des données les plus récentes : PILES.
- Ajout, récupération des données les plus vieilles : FILES.
- Ajout, suppression, consultation de données *ordonnées* : ARBRES BINAIRES DE RECHERCHE.
- Ajout, suppression du plus gros(/petit) élément parmi des données *ordonnées* : TAS.
- etc...

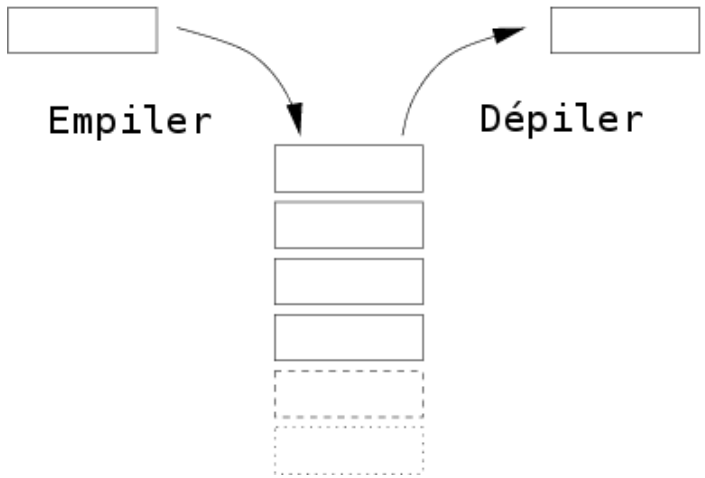
*La complexité (annoncée) des différentes opérations guide le choix de la structure de données.*

# Sommaire

# Dans le cas des piles...



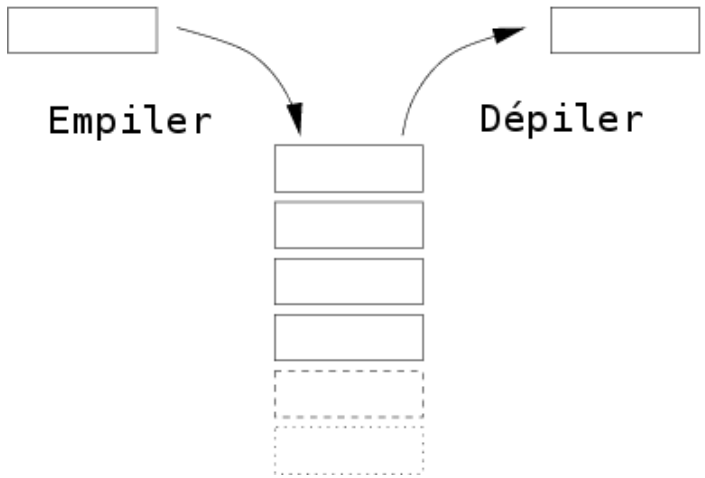
# Dans le cas des piles...




Opérations supportées :

- empiler ;
- dépiler ;

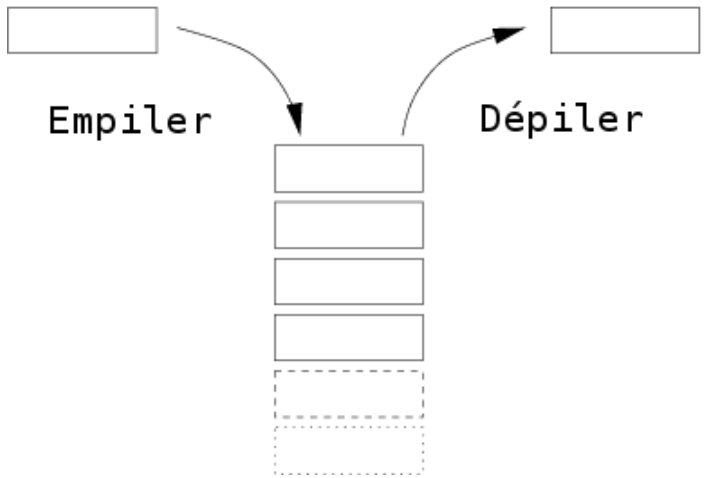
# Dans le cas des piles...




Opérations supportées :

- empiler ;
- dépiler ;
- créer une pile ;
- tester sa vacuité ;

# Dans le cas des piles...




Opérations supportées :

- empiler ;
- dépiler ;
- créer une pile ;
- tester sa vacuité ;
- optionnellement : lire le haut de pile ; évaluer la hauteur.

# Sommaire

# Quelques utilisations

- Bas niveau :
  - gestion de la récursivité (retours d'appels) ;
  - calcul en polonaise inversée (jadis...).

# Quelques utilisations

- Bas niveau :
  - gestion de la récursivité (retours d'appels) ;
  - calcul en polonaise inversée (jadis...).
- Haut niveau :
  - alternative à la récursivité (tri rapide) ;
  - tri par insertion via deux piles ;
  - backtracking : parcours d'un graphe en profondeur, problème du cavalier, du serpent : récursivité ou TODO-list

# Sommaire

# Dans un tableau de taille fixe

- Création, vacuité :

```
LongueurPile = 2*10**4
def creer_pile():
    return [0] + ([None] * LongueurPile)
def est_vide(pile):
    return pile[0] == 0
```

# Dans un tableau de taille fixe

- Création, vacuité :

```
LongueurPile = 2*10**4
def creer_pile():
    return [0] + ([None] * LongueurPile)
def est_vide(pile):
    return pile[0] == 0
```

- Push/pop :

```
def empiler(x, pile):
    assert pile[0] < LongueurPile
    pile[0] += 1
    pile[ pile[0] ] = x
def depiler(pile):
    pile[0] -= 1
    return pile[ pile[0] + 1 ]
```

# Sommaire

# Dans un tableau de taille variable

- Les listes Python... sont des piles !
- Création, test de vacuité :

```
def creer_pile():
    return []
def est_vide(pile):
    return pile == []
```

# Dans un tableau de taille variable

- Les listes Python... sont des piles !
- Création, test de vacuité :

```
def creer_pile():
    return []
def est_vide(pile):
    return pile == []
```

- Push/pop :

```
def empiler(x, pile):
    pile.append(x)
def depiler(pile):
    return pile.pop()
```

# Dans un tableau de taille variable

- Les listes Python... sont des piles !
- Création, test de vacuité :

```
def creer_pile():
    return []
def est_vide(pile):
    return pile == []
```

- Push/pop :

```
def empiler(x, pile):
    pile.append(x)
def depiler(pile):
    return pile.pop()
```

- Complexités ?

# Sommaire

# Un peu de programmation objet (culturel)

```
class Pile3(object):
    """ Pile implémentée dans une classe
    Il manque (volontairement) une méthode "longueur" """
    def __init__(self):
        self.contenu = []
    def est_vide(self):
        return self.contenu == []
    def empiler(self, a):
        self.contenu.append(a)
    def depiler(self):
        assert self.contenu != []
        return self.contenu.pop()
    def __repr__(self):
        return " ; ".join([str(x) for x in self.contenu])
```

# Quelques exercices

- Écrire un interpréteur d'expressions en notation polonaise inversée :

```
>>> interpreterNPI( ['12', '6', '5', '*', '+'] )
42
```

# Quelques exercices

- Écrire un interpréteur d'expressions en notation polonaise inversée :

```
>>> interpreterNPI( ['12', '6', '5', '*', '+'] )
42
```

- En pseudo-code, comment parcourir les sommets d'un graphes ?

# Quelques exercices

- Écrire un interpréteur d'expressions en notation polonaise inversée :

```
>>> interpreterNPI( ['12', '6', '5', '*', '+'] )
42
```

- En pseudo-code, comment parcourir les sommets d'un graphes ?
  - Qu'est-ce que ça signifie ?
  - Comment représenter un graphe ?

# Quelques exercices

- Écrire un interpréteur d'expressions en notation polonaise inversée :

```
>>> interpreterNPI( ['12', '6', '5', '*', '+'] )
42
```

- En pseudo-code, comment parcourir les sommets d'un graphes ?
  - Qu'est-ce que ça signifie ?
  - Comment représenter un graphe ?
- Fonctions d'échanges d'éléments de la pile :

```
swap2( p )          swap_n( p, n )
```

# Quelques exercices

- Écrire un interpréteur d'expressions en notation polonaise inversée :

```
>>> interpreterNPI( ['12', '6', '5', '*', '+'] )
42
```

- En pseudo-code, comment parcourir les sommets d'un graphes ?
  - Qu'est-ce que ça signifie ?
  - Comment représenter un graphe ?
- Fonctions d'échanges d'éléments de la pile :

`swap2( p )` `swap_n( p, n )`

- Permutation circulaire des *n* éléments en haut de pile.

`roll_n( p, n )`

# Quelques exercices

- Écrire un interpréteur d'expressions en notation polonaise inversée :

```
>>> interpreterNPI( ['12', '6', '5', '*', '+'] )
42
```

- En pseudo-code, comment parcourir les sommets d'un graphes ?
  - Qu'est-ce que ça signifie ?
  - Comment représenter un graphe ?
- Fonctions d'échanges d'éléments de la pile :

`swap2( p )` `swap_n( p, n )`

- Permutation circulaire des *n* éléments en haut de pile.

`roll_n( p, n )`

- Superposition de deux piles.

# C’est fini !

Merci de votre attention.

**Table 1**

| NUMBER OF ITEMS | MERGE SORT | QUICKSORT |
|---|---|---|
| 500 | 2 min 8 sec | 1 min 21 sec |
| 1,000 | 4 min 48 sec | 3 min 8 sec |
| 1,500 | 8 min 15 sec* | 5 min 6 sec |
| 2,000 | 11 min 0 sec* | 6 min 47 sec |

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.