L3 SID APU
Cours 5
Structures de données élémentaires

Thomas Pellegrini, équipe SAMoVA, IRIT,
thomas.pellegrini@irit.fr

IRIT - UPS

2016-2017

Introduction

programme = algorithme + structure de données

Introduction

programme = algorithme + structure de données

Introduction

programme = algorithme + structure de données

- Comment représenter des ensembles dynamiques ?
- A l'aide de structures de données simples qui utilisent des *pointeurs*
- Nous allons voir les plus simples : piles (*stack*), files (*queue*), listes chaînées (*linked lists*)

IRIT CNRS - INPT - UPS - UT1 - UT2J

UNIVERSITÉ TOULOUSE III PAUL SABATIER Université de Toulouse

Ce qui est dispo en Python

- Il y a plusieurs SD prêtes à l'emploi en Python : *lists, tuples, dictionaries, strings, sets and frozensets*
- On parle de types ou de SD « builtin »

Lists : crochets, *brackets*

```
l = [1, 2, "a"]
```

Tuples (n-uplets en français) : parenthèses

```
t = (1, 2, "a")
```

Dictionnaires : accolades

```
d = {"a":1, "b":2}
```

IRIT CNRS - INPT - UPS - UT1 - UT2J

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Ce qui est dispo en Python

Strings : guillemets simples ou double *single or double quotes*

```
s = 'Yves'
```

Sets : accolades

```
a = {1, 2, 3, 4}
```

Sets : accolades

```
a = {1, 2, 3, 4}
```

Frozensets

```
a = frozenset([1, 2, 3, 4])
```

IRIT CNRS - INPT - UPS - UT1 - UT2J

Université Toulouse III Paul Sabatier Université de Toulouse

Lists : méthodes

```
>>> l = [1, 2, 3]
>>> l[0]
1
>>> l.append(1)
>>> l
[1, 2, 3, 1]
>>> l.extend([4,5,6])
>>> l
[1, 2, 3, 1, 4, 5, 6]
>>> l.append([4,5,6])
>>> l
[1, 2, 3, 1, 4, 5, 6, [4, 5, 6]]
>>> len(l)
8
>>> max([1, 2, 3])
3
```

Lists : méthodes

```
>>> my_list = ['a','b','c','b','a']
>>> my_list.index('b')
1
>>> my_list.index('b', 2)
3
>>> my_list.remove('a')
>>> my_list
['b', 'c', 'b', 'a']
>>> my_list.pop()
'a'
>>> my_list
['b', 'c', 'b']
>>> my_list.count('b')
2
>>> my_list.sort()
>>> my_list
['b', 'b', 'c']
```

Lists : méthodes

```
>>> my_list.sort(reverse=True)
>>> my_list
['c', 'b', 'b']
>>> my_list = ['b','c','a']
>>> my_list.reverse()
>>> my_list
>>> ['a', 'c', 'b']
```

IRIT CNRS - INPT - UPS - UT1 - UT2J

UNIVERSITÉ TOULOUSE III PAUL SABATIER Université de Toulouse

Lists : méthodes

```
>>> my_list = [1]
>>> my_list += [2]
>>> my_list
[1, 2]
>>> my_list += [3, 4]
>>> my_list
[1, 2, 3, 4]
```

CNRS - INPT - UPS - UT1 - UT2J

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Lists : méthodes

```
>>> my_list = [1, 2]
>>> my_list = my_list * 2
>>> my_list
[1, 2, 1, 2]
>>> my_list = [0]*10
>>> my_list
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> my_list = [[0]*10]*3
>>> my_list[0][0]
0
>>> my_list[0][0]=1
>>> my_list
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Lists : slicing

```
>>> a = [0, 1, 2, 3, 4, 5]
>>> a[2:]
[2, 3, 4, 5]
>>> a[:2]
[0, 1]
>>> a[-1]
5
>>> a[-1]
4
>>> a[2:-1]
[2, 3, 4]
>>> a[::2]
[0, 2, 4]
>>> a[::-1]
[5, 4, 3, 2, 1, 0]
```

Lists

```
a = ['a', 'c', 'b']

for el in a:
    print(el)

for i, v in enumerate(a):
    print(i, v)

a2 = [el + '!' for el in a]

a = [1, 2, 3]
a2 = [el + 1 for el in a]
```

Tuples

```
>>> l = (1, 2, 3)
>>> l[0]
1
>>> l = 1, 2, 3
>>> singleton = (1, )
>>> (1,) * 5
(1, 1, 1, 1, 1)
>>> s1 = (1,0)
>>> s1 += (1,)
>>> s1
(1, 0, 1)
```

Tuples

Similaire aux listes mais :

- Plus rapides car non-mutables
- Utiles pour protéger des données (non-mutables)
- Utiles pour déclarer des dictionnaires

```
>>> d = dict([('jan', 1), ('feb', 2), ('march', 3)])
>>> d['feb']
2
>>> def rien(bla, bla):
    ...
    return a, b, c
>>> s = rien(2,3)
>>> type(s)
tuple
>>> a, b, c = s # unpacking
```

Tuples

```
>>> t= (1,2)
>>> len(t)
2
>>> t = (1,2,3,4,5)
>>> t[2:] # slicing
(3, 4, 5)
>>> t = (1, 2, 3, 4, 5)
>>> newt = t # vraie copie, pas comme list,
# car non-mutable
>>> t[0] = 5
>>> newt
(1, 2, 3, 4, 5)
```

dicts

```
>>> d = {'first':'string value', 'second':[1,2]}
>>> d.keys()
['first', 'second']
>>> d.values()
['string value', [1, 2]]
>>> d.items()
[('second', [1, 2]), ('first', 'string value')]
>>> d.has_key('first')
True
>>> 'first' in d
True
```

- On ne peut pas avoir de doublons dans les clés
- Les éléments dans un dict ne sont pas ordonnés

```
>>> d.get('first')
'string value'
>>> d.get('firs', 'pas dans le dico !')
'pas dans le dico !'
>>> d.pop('first')
'string value'
>>> d
{'second': [1, 2]}
>>> d1 = {'a': [1,2]}
>>> d2 = d1
>>> d2['a'] = [1,2,3,4]
>>> d1['a]
[1,2,3,4]
>>> d2 = d1.copy() # shallow copy
>>> d2 = d1.deepcopy() # deep copy
>>> d2.clear()
{}
>>> d = {'a':1, 'b':2, 'c':3}
>>> del d['a']
>>> d.clear()
```

CNRS - INPT - UPS - UT1 - UT2J

UNIVERSITÉ TOULOUSE III PAUL SABATIER

```
>>> d1 = {'a':1}
>>> d2 = {'a':2; 'b':2}
>>> d1.update(d2)
>>> d1['a']
2
>>> d2['b']
2


>>> [x for x in t.itervalues()]
['string value', [1, 2]]
>>>
>>> [x for x in t.iterkeys()]
['first', 'csecond']
>>> [x for x in t.iteritems()]
[('a', 'string value'), ('b', [1, 2])]
```

IRIT CNRS - INPT - UPS - UT1 - UT2J

UNIVERSITÉ TOULOUSE III PAUL SABATIER Université de Toulouse

```python
import collections
print 'Regular dictionary:'
d = {}
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'
d['d'] = 'D'
d['e'] = 'E'
for k, v in d.items():
    print k, v

print '\nOrderedDict:'
d = collections.OrderedDict()
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'
d['d'] = 'D'
d['e'] = 'E'
for k, v in d.items():
    print k, v
```

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1,\
'spam': 500}
>>> cles = dishes.viewkeys()
>>> valeurs = dishes.viewvalues()
>>> list(cles)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(valeurs)
[2, 1, 1, 500]
>>> # 'cles' et 'valeurs' sont des 'vues'
# elles refletent les changements
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(cles)
['spam', 'bacon']

>>> # operations ensemblistes
>>> cles & {'eggs', 'bacon', 'salad'}
{'bacon'}
```

IRIT CNRS - INPT - UPS - UT1 - UT2J

Université Toulouse III Paul Sabatier

sets

```
>>> a = set([1, 2, 3, 4])
>>> b = set([3, 4, 5, 6])
>>> a | b # Union
{1, 2, 3, 4, 5, 6}
>>> a & b # Intersection
{3, 4}
>>> a < b # Subset
False
>>> a - b # Difference
{1, 2}
>>> a ^ b # Difference symetrique
{1, 2, 5, 6}
```

- Attention : pas ordonnés, se méfier comme avec les dict !

```
>>> a = set([1, 2, 3])
>>> b = set([2, 3, 4])
>>> c = a.intersection(b) # eq. c = a & b
>>> a.intersection(b)
set([2, 3])
>>> c.issubset(a) # eq. c <= a
True
>>> c.issuperset(a) # eq. c >= a
False
>>> a.difference(b) # eq. a - b
set([1])
>>> a.symmetric_difference(b) # eq. a ^ b
set([1, 4])
```

Recap

- Lists, strings et tuples : séquences ordonnées d'objets
- Strings : seulement des caractères, lists et tuples : tout type d'objets
- Dictionnaires et sets : séquences non-ordonnées d'objets

- Lists, sets, dictionnaires : mutables
- Tuples, strings, frozensets : non mutables

- On parle de « méthodes » pour désigner des fonctions définies pour un type d'objet

IRIT CNRS - INPT - UPS - UT1 - UT2J

Université Toulouse III Paul Sabatier

Piles

Définition

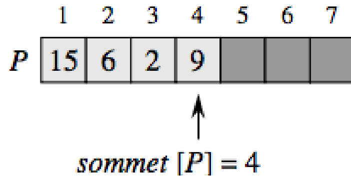
- Pile = liste qui ne permet des insertions ou des suppressions qu'à une seule extrémité, le sommet de la pile.
- EMPILER : insertion, DEPILER : suppression
- Propriété : un objet ne peut être dépilé qu'après avoir supprimé tous les objets au-dessus de lui
- *LIFO : Last In, First Out*

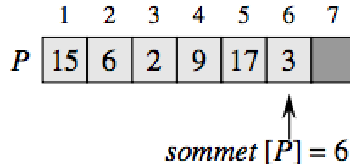
Piles

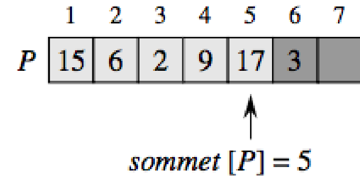
Exemples

- La fonction « annuler la frappe » (*undo*) mémorise les modifications apportées au texte dans une pile
- Les algorithmes récursifs utilisent implicitement une pile d'appels
- Pile d'exécution : les appels et retours de fonctions. La machine stocke l'adresse de la ligne de la fonction A pour exécuter une fonction B si besoin, puis revient à l'exécution de la fonction A

Piles







- Pile = tableau (ou liste) de n éléments $P[1 \ldots n]$
- Un attribut : P.sommet qui indexe l'élément le plus récemment empilé
- $\rightarrow$ Pile = tableau $P[1 \ldots P.sommet]$
- $P[1]$: élément à la base de la pile
- $P[P.sommet]$: élément au sommet de la pile
- P.sommet = 0 $\Rightarrow$ P est vide, on va définir une méthode « PILE-VIDE » qui teste si P est vide ou non

Piles : opérations

```
1: FONCTION PILE-VIDE(P)
2:     SI P.sommet == 0 ALORS
3:         RETOURNER VRAI ;
4:     SINON
5:         RETOURNER FAUX ;
```

Piles : opérations

```
1: FONCTION PILE-VIDE(P)
2:    SI P.sommet == 0 ALORS
3:       RETOURNER VRAI ;
4:    SINON
5:       RETOURNER FAUX ;
```

```
1: FONCTION EMPILER(P, x)
2:    P.sommet ← P.sommet + 1 ;
3:    P[P.sommet] ← x ;
```

Piles : opérations

```
1: FONCTION PILE-VIDE(P)
2:     SI P.sommet == 0 ALORS
3:         RETOURNER VRAI ;
4:     SINON
5:         RETOURNER FAUX ;
```

```
1: FONCTION EMPILER(P, x)
2:     P.sommet ← P.sommet + 1 ;
3:     P[P.sommet] ← x ;
```

```
1: FONCTION DEPILER(P)
2:     SI PILE-VIDE(P) ALORS
3:         ERREUR ”débordement négatif” ;
4:     SINON
5:         P.sommet = P.sommet-1 ;
6:         RETOURNER P[P.sommet+1]
```

Piles : opérations

```
1: FONCTION PILE-VIDE(P)
2:    SI P.sommet == 0 ALORS
3:       RETOURNER VRAI ;
4:    SINON
5:       RETOURNER FAUX ;
```

```
1: FONCTION EMPILER(P, x)
2:    P.sommet ← P.sommet + 1 ;
3:    P[P.sommet] ← x ;
```

```
1: FONCTION DEPILER(P)
2:    SI PILE-VIDE(P) ALORS
3:       ERREUR "débordement négatif" ;
4:    SINON
5:       P.sommet = P.sommet-1 ;
6:       RETOURNER P[P.sommet+1]
```

Chacune des trois opérations consomme un temps $O(1)$

Piles : première implémentation en Python

```
class Pile(object):
    def __init__(self):
        self.sommet = -1
        self.P = []

    def estVide(self):
        return self.sommet == -1

    def empiler(self, x):
        self.sommet =self.sommet +  1
        self.P.append(x)

    def depiler(self):
        if self.estVide():
            print('erreur, pile vide !')
        else:
            self.sommet = self.sommet - 1
            return self.P[self.sommet+1]
```

Piles : programme principal

```
a = Pile()
print(a.estVide())
a.depiler()
a.empiler(1)
a.empiler(2)
a.empiler(3)
print(a.estVide())
print(a.depiler())
```

Sortie : ?

IRIT CNRS - INPT - UPS - UT1 - UT2J

UNIVERSITÉ TOULOUSE III PAUL SABATIER Université de Toulouse

Piles : deuxième implémentation en Python

```
class Pile(object):
    def __init__(self):
        self.P = []

    def estVide(self):
        return self.P == []

    def empiler(self, x):
        self.P.append(x)

    def depiler(self):
        if self.estVide():
            print('erreur, pile vide !')
        else:
            return self.P.pop()

    def taille(self):
        return len(self.P)
```

Files

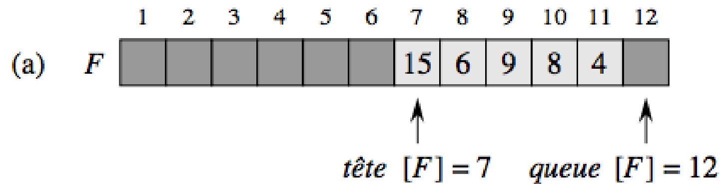
Définition

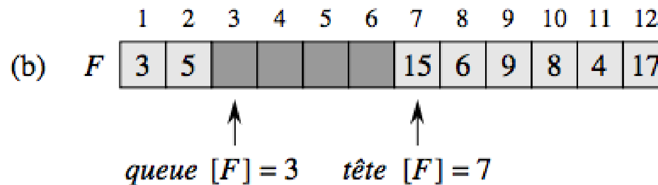
- File = liste qui ne permet des insertions qu'à une extrémité (queue, *tail*), et des suppressions qu'à l'autre extrémité (tête, *head*)
- ENFILER : opération d'ajout, DEFILER : opération de suppression
- *FIFO* : First In, First Out

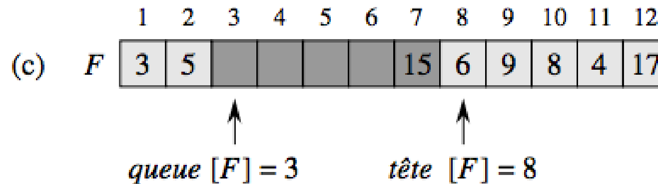
Exemple

- Principale application : les **buffers** ou mémoire tampon, *i.e.* espace mémoire temporaire
- Imprimantes : elles traitent les documents à imprimer en fonction de l'ordre dans lequel les requêtes arrivent, en les mettant dans une file d'attente

Files







- File = tableau (ou liste) de $n - 1$ éléments : $F[1 \ldots n]$
- Deux attributs : F.tete et F.queue
- Les éléments se trouvent indicés par F.tete, F.tete+1, . . ., F.queue-1
- F.tete = F.queue $\Rightarrow$ F est vide
- F.tete = F.queue + 1 $\Rightarrow$ F est pleine

Files : opérations

A vous d'écrire les fonctions : FILE-VIDE, FILE-PLEINE, ENFILER, DEFILER