

Introduction à l'Algorithmique

TC1-1A-ECL

Ecole Centrale de Lyon
Département Mathématiques-Informatique

Structures de données remarquables et algorithmes

CHAPITRE I : Partie II

Algorithmes

2016-2017

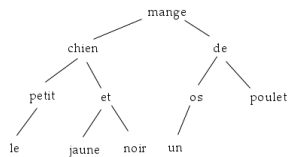
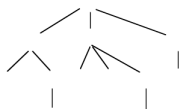
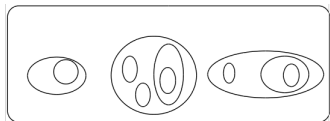
SDs et Stratégies remarquables

Structures importantes :

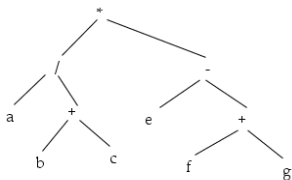
- Arbres
- Graphes
- ABOH / AVL / TAS
- Etc...

Parcours dans ces structures

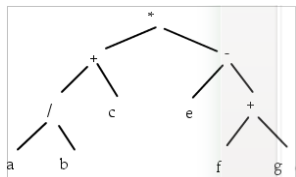
Les arbres



$$(a/(b+c) * (e-(f+g)))$$



$$(a/b+c) * (e-(f+g))$$



Les arbres (suite)

Quelques définitions

- arbre, sous-arbre, noeud, racine
- descendant (fils, fille), ascendant (père, mère)
- feuille, mot des feuilles,
- hauteur (niveau)
- arbre n-aire, binaire, ternaire, quaternaire, ...
- Relation d'ordre, ABOH, TAS (Heap), AVL, etc

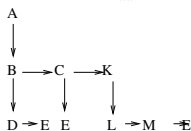
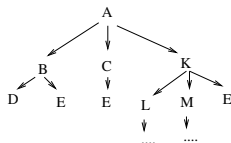
Définition récursive d'un arbre binaire :

Un arbre est

- soit vide
- soit un élément , un arbre (gauche) et un arbre (droit)

Arbres : représentation

- Représentation des arbres par : matrice, liste , table (tableau), ...
- Exemple : représentation d'un arbre **ternaire** par matrices / tableaux / listes
 - ➔ Abstraction **Forêt** (d'arbres) : notions de Fils et Frère (utilisation pour les dicos)



Abstraction Forêt

A	B C K
B	D E
C	E
D	A
E	
K	L M E
L
M
....	

Représentation par Listes

☞ L'abstraction Forêt permet en fait de présenter un arbre n-aire par un binaire.

Représentation d'Arbres binaires

Une représentation par tableaux (listes) :

Convention :

Si l'indice (Place) du noeud est = i

Alors la Place désignant son fils gauche est à $2i$ et

la Place désignant son fils droit est à $2i + 1$.

- La hauteur d'un arbre (binaire) h
- Arbre binaire **équilibré**
- La hauteur h d'un arbre **équilibré** (*compacte*) est telle que
$$h = \lfloor \log(NB_ele) \rfloor + 1 \text{ d'où } NB_ele = 2^h - 1$$

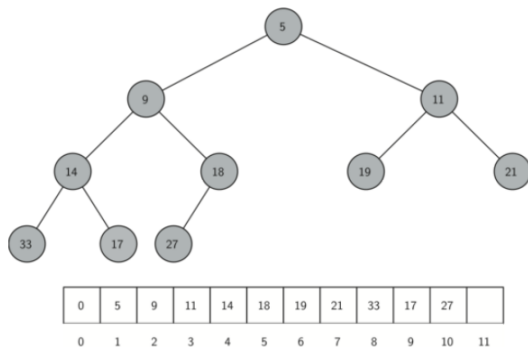
☞ **Important** : question d'équilibre de cette représentation.

☞ Techniques employées pour équilibrer les arbres, les **AVL** (v. plus loin)

Représentation d'Arbres binaires (suite)

Un **exemple** de cette représentation :

Un arbre binaire *complet* et sa représentation par table / liste

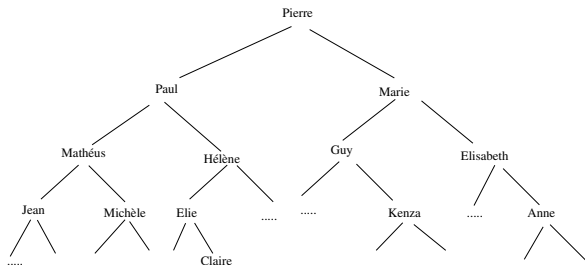


☞ Relation d'ordre : on est en présence d'un *Heap* minimal (v. *Heap* plus loin)

☞ Problème de compacité : pas trous entre les éléments !

Représentation d'Arbres binaires (suite)

Une autre représentation possible : exemple *généalogie*

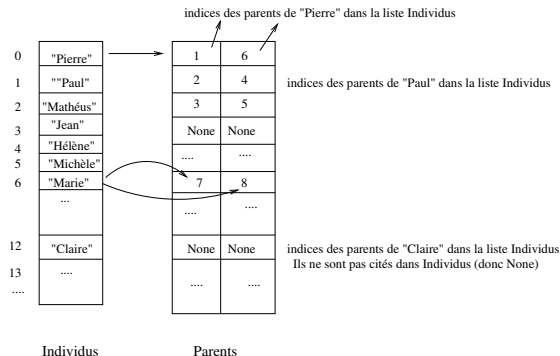


- Exemples de questions posées (les développer sur la représentation concrète) :
 - *Trouver les ancêtres de (p. ex.) Hélène.*
 - *Insérer un nouvel enfant ;*
 - *Afficher l'arborescence, etc...*
- Une représentation physique possible :

... ~>

Représentation d'Arbres binaires (suite)

Représentation par listes :



☞ Comparaison des deux représentations.

→ La 2nde est est plus élaborée (temps d'accès plus long) mais plus efficace.

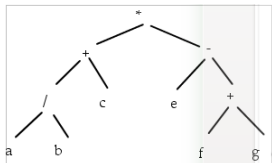
Parcours d'arbre binaire

- Soit : **R** = Racine, **D** = sous-arbre Droit et **G** = sous-arbre Gauche
- Il y a trois types (principaux) de parcours : observer la place de **R**
 - Préfixé : **R** G D : pré-ordre
 - Infixé : G **R** D : mi-ordre
 - Postfixé : G D **R** : post-ordre

Exemples :

Les noeuds visités selon le mode de parcours :

- Infixé : $a/b+c*e-f+g \rightarrow ((a/b)+c) * (e-(f+g))$
- Postfixé : $a b / c + e f g + - *$
- Préfixé : $* + / a b c - e + f g$



☞ On remarque que lors d'une évaluation (manuelle) de l'expression, seul le mode *infixé* nécessite des parenthèses pour lever des ambiguïtés éventuelles.

Parcours d'arbre binaire (suite)

Les algorithmes de parcours d'arbre binaire en Python :

```
def Prefixe(R) :
    if not vide(R) :
        traiter(R);           # R
        Pefixe (gauche(R));  # G
        Pefixe (droit(R));   # D

def Infixe(R) :
    if not vide(R) :
        Infixe (gauche(R));  # G
        traiter(R);         # R
        Infixe (droit(R));   # D

def Postfixe(R) :
    if not vide(R) :
        Postfixe (gauche(R)); # G
        Postfixe (droit(R));  # D
        traiter(R);          # R
```

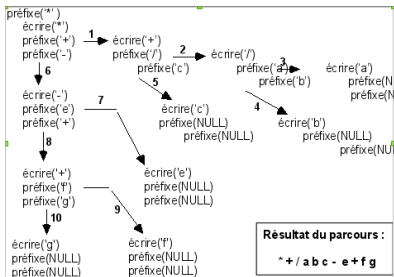
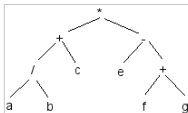
☞ Le paramètre R représente initialement la racine puis différents noeuds aux cours des appels.

→ Voir plus loin le codage complet en Python.

N.B. : le *Type de Données Abstrait* (TDA) est un outil apprécié pour la définition algébrique et formelle de la structure arbre (voir plus loin en Annexes).

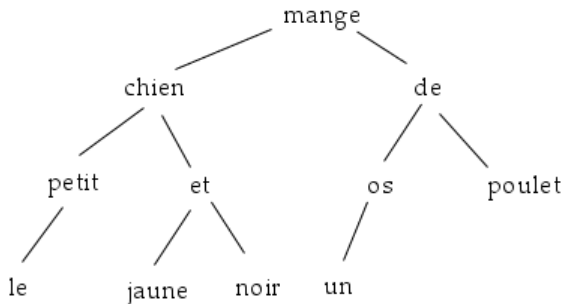
Parcours d'arbre binaire (suite)

Trace du parcours préfixé (pré-ordre) de l'arbre d'une expressions :



Parcours d'arbre binaire (suite)

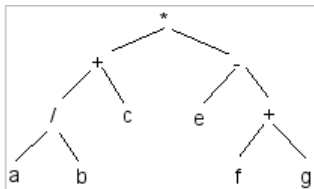
Question : quel type de parcours pour retrouver les informations dans l'ordre dans l'arbre binaire suivant ?



Arbres binaires : un exemple Python

Représentation et parcours Python de l'arbre de l'expression arithmétique précédente :

- On représente cet arbre par une structure similaire à l'exemple *généalogie* ci-dessus.



L'arbre est réalisé par 2 données :

- "Arbre" (tableau à gauche) et
- "Fils" (celui de droite).

indice	Noeuds	Fils
0	'*'	(1,6)
1	'+'	(2,5)
2	'/'	(3,4)
3	'a'	(None, None)
4	'b'	(None, None)
5	'c'	(None, None)
6	'-'	(7,8)
7	'e'	(None, None)
8	'+'	(9,10)
9	'f'	(None, None)
10	'g'	(None, None)

☞ Dans le code python qui suit, l'arbre sera une donnée globale.

Arbres binaires : un exemple Python (suite)

```
# Les fonctions reçoivent un indice, La racine commence toujours en 0
```

```
def prefixe(S) :  
    global Arbre  
    if S != None :  
        traiter(S)  
        prefixe(gauche(S))  
        prefixe(droite(S))
```

```
def infixe(S) :  
    global Arbre  
    if S != None :  
        infixe(gauche(S))  
        traiter(S)  
        infixe(droite(S))
```

```
def prefixe(S) :  
    global Arbre  
    if S != None :  
        traiter(S)  
        prefixe(gauche(S))  
        prefixe(droite(S))
```

```
def postfixe(S) :  
    global Arbre  
    if S != None :  
        postfixe(gauche(S))  
        postfixe(droite(S))  
        traiter(S)
```


Arbres binaires : un exemple Python (suite)

```
# N'est appelé que si S existe (un indice)
def traiter(S) :
    global Arbre
    assert(S) # ← vérification. (utilisation d'une exception possible)
    print(Arbre[S], end= ' ')

# N'est appelé que si S existe (un indice)
def gauche(S) :
    global Fils
    return Fils[S][0]

# N'est appelé que si S existe (un indice)
def droite(S) :
    global Fils
    return Fils[S][1]

# Liste principale + matrice de G/D
def go() :
    global Arbre
    global Fils

    Arbre = ['*', '+', '/', 'a', 'b', 'c', '-', 'e', '+', 'f', 'g']

    Fils = [(1,6),      # 0
            (2,5),      # 1
            (3,4),      # 2
            (None, None), # 3
            (None, None), # 4
            (None, None), # 5
            (7,8),      # 6
            (None, None), # 7
            (9,10),     # 8
            (None, None), # 9
```

Arbres binaires : un exemple Python (suite)

```

        (None, None), # 10
    ]

    infixe(0); print()
    prefixe(0); print()
    postfixe(0); print()

# -----
#          TEST (rapide) : sans construire __main__
# -----

go()

# Trace :
a / b + c * e - f + g
* + / a b c - e + f g
a b / c + e f g + - *

```

- On reprendra cette expression plus loin pour une **évaluation** arithmétique.

Exercice : donner une solution pour la représentation $2i, 2i + 1, 2i + 2$ (indices : 0..)

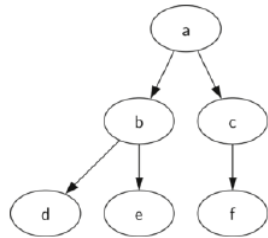
→ Idée d'équilibrage de la représentation $2i, 2i + 1, 2i + 2$

Arbres : représentation par listes imbriquées

Représentation alternative : une liste de listes contient l'arbre.

```

un_arbre=
  ['a',           # racine
   ['b',         # a-b
    ['d',       # a-b-d
     [], []    # 2 ss-arbres vides de 'd'
    ],
   ['e',       # a-b-e
    [], []    # 2 ss-arbres vides de 'e'
   ]
  ],
  ['c',         # fin de la partie gauche
   ['f',       # a-c
    [], []    # 2 ss-arbres vides de 'f'
   ],
   []         # fils droit de 'c'
  ]
 ]
  
```



- La racine de l'arbre est accessible par `un_arbre[0]`,
- Le fils gauche par `un_arbre[1]` et le fils droit par `un_arbre[2]`.

Arbres : représentation par listes imbriquées (suite)

- La même convention s'applique aux sous-arbres.

```
un_arbre[0]
# 'a'

un_arbre[1]
# ['b', ['d', [], []], ['e', [], []]]

un_arbre[2]
# ['c', ['f', [], []], []]

un_arbre[1][0]
# 'b'
```

☞ Un arbre n-aire peut adapter cette même représentation.

- La définition récursive des listes s'applique aux différentes représentations des arbres où les sous-arbres ont la même structure que l'arbre entier.

- Rappel : pour un arbre binaire, cette définition est :

- soit *Vide*
- soit une *information*, *Arbre*, *Arbre*

→ C-à-d. une information puis un sous-arbre (gauche) puis un sous-arbre (droit).

Python : Fonctions usuelles sur arbres binaires

Quelques fonctions usuelles pour les arbres binaires repr. par **listes imbriquées** :

```
def arbre_binaire(r):    return [r, [], []]

def get_val_racine(arbre):
    assert(arbre !=[])
    return arbre[0]

def set_val_racine(arbre ,new_val):
    assert(arbre !=[])
    arbre[0] = new_val

def get_fils_gauche(arbre):
    assert(arbre !=[])
    return arbre[1]

def get_fils_droit(arbre):
    assert(arbre !=[])
    return arbre[2]

def inserer_gauche(arbre , new_gauche):
    def ins_gauche(A,new_gch) :
        if not A[1] : A[1]=[new_gch,[],[]]
        else :    ins_gauche(A[1], new_gch)
    assert(len(arbre)>1)
    ins_gauche(arbre, new_gauche)
    return arbre

def inserer_droit(arbre , new_droite):
    def ins_droit(A,new_dt) :
        if not A[2] : A[2]=[new_dt,[],[]]
        else :    ins_droit(A[2], new_dt)
    assert(len(arbre)>1)
    ins_droit(arbre, new_droite)
    return arbre
```

Python : Fonctions usuelles sur arbres binaires (suite)

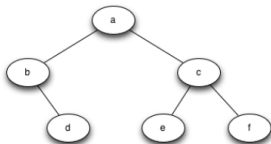
Quelques tests (dessiner l'arbre !) :

```
r = arbre_binaire(3)
print(r)
# [3, [], []]
inserer_gauche(r, 4)
print(r)
# [3, [4, [], []], []]
inserer_gauche(r, 5)
print(r)
# [3, [4, [5, [], []], []], []]
inserer_droit(r, 6)
print(r)
# [3, [4, [5, [], []], []], [6, [], []]]
inserer_droit(r, 7)
print(r)
# [3, [4, [5, [], []], []], [6, [], [7, [], []]]]
l = get_fils_gauche(r)
print(l)
# [4, [5, [], []], []]
set_val_racine(l, 9)
print(l)
# [9, [5, [], []], []]
inserer_gauche(l, 11)
print(l)
# [9, [5, [11, [], []], []], []]
print(r)
# [3, [9, [5, [11, [], []], []], []], [6, [], [7, [], []]]]
print(get_fils_droit(get_fils_droit(r)))
#[7, [], []]
```

Python : Fonctions usuelles sur arbres binaires (suite)

Un autre exemple :

construction de l'arbre ci-dessous à l'aide des opérateurs précédents :



```

a = arbre_binaire('a')
b= inserer_gauche(a, 'b') #la valeur renvoyé est l'arbre complet (a)
print(a)
# ['a', ['b', [], []], []]

print(b) # Affichera l'arbre a car b récupère la totalité de l'arbre
# ['a', ['b', [], []], []]

inserer_droit(a, 'c')
inserer_droit(get_fils_gauche(a), 'd')
c=get_fils_droit(a)
inserer_gauche(c, 'e')
inserer_droit(get_fils_droit(a), 'f')

print(a)
# ['a', ['b', [], ['d', [], []]], ['c', ['e', [], []], ['f', [], []]]]
  
```

Python : Fonctions usuelles sur arbres binaires (suite)

Un autre exemple :

afficher (dessiner) l'arbre précédent à l'écran.

- Chaque information (noeud ou feuille) sera affichée sur une ligne.
- L'idée est de descendre le plus à droite possible dans l'arbre pour afficher la feuille la plus à droite sur une ligne puis écrire l'information du parent de cette feuille-là sur la ligne suivante avant d'aller à gauche.

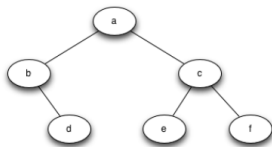
```
def affiche_arbre(arbre) :  
    def affiche_arbre_bis(arbre, decalage) :  
        if arbre != []:  
            affiche_arbre_bis(get_fils_droit(arbre), decalage+3)  
            print(' '*decalage, arbre[0])  
            affiche_arbre_bis(get_fils_gauche(arbre), decalage+3)  
    affiche_arbre_bis(arbre, 1)
```

→ Noter que `affiche_arbre(arbre)` joue le rôle du (*wrapper*) et prépare le paramètre *décalage* (espaces à gauche) pour `affiche_arbre_bis(arbre, decalage)`.

Python : Fonctions usuelles sur arbres binaires (suite)

Trace pour l'arbre **a** ci-dessus (tourner la tête 90 degrés à gauche !):

```
a=.....  
affiche_arbre(a)  
  
# On obtient :  
      f  
     c  
    e  
   a  
  d  
 b
```



Quelques autres algorithmes en Python sur les arbres

Quelques algorithmes sur les arbres binaires :

- 1 La fonction `taille` qui calcule le nombre de noeuds d'un arbre binaire.
- 2 La fonction `feuille` qui test si un noeud d'un arbre binaire est une feuille.
- 3 La fonction `nb_feuilles` qui calcule le nombre de feuilles d'un arbre binaire.
- 4 La fonction `hauteur` qui calcule la hauteur d'un arbre binaire.
NB : La hauteur est définie par le maximum de nombre d'arcs allant de la racine jusqu'à toute feuille + 1.
- 5 La fonction `recherche` qui cherche l'élément X dans un arbre binaire.
- 6 La fonction `isomorphe` qui vérifie si deux arbres contiennent les mêmes informations dans le même ordre.

Quelques autres algorithmes en Python sur les arbres (suite)

☞ Les algorithmes suivant font abstraction de la représentation de l'arbre.

- La fonction *taille* qui calcule le nombre de noeuds d'un arbre binaire.

```
def taille(R) :  
    if vide(R) : return 0  
    else : return 1+taille(gauche(R))+taille(droit(R));
```

→ Quel est le type de ce parcours ?

- La fonction *feuille* : teste si un noeud d'un arbre binaire est une feuille.

```
def feuille(R) :  
    if vide(R) : return False;  
    else : return vide(gauche(R)) and vide(droit(R));
```

→ Quel est le type du parcours ?

Quelques autres algorithmes en Python sur les arbres (suite)

- La fonction *nb_feuilles* : calcule le nombre de feuilles d'un arbre binaire.

```
def nb_feuilles(R) :  
    if vide(R) : return 0  
    if feuille(R) : return 1  
    else : return 1+nb_feuilles(gauche(R))+nb_feuilles(droit(R));
```

→ Quel est le type du parcours ?

- La fonction hauteur qui calcule la hauteur d'un arbre binaire.

NB : La hauteur est définie par le maximum de nombre d'arcs allant de la racine (jusqu'à toute feuille) + 1.

```
def hauteur(R) :  
    if vide(R) : return 0  
    else : return max(hauteur(gauche(R)), hauteur(droit(R)))+1
```

- Avec *taille* et *hauteur*, on peut vérifier la *compacité*
- Ensuite, soit on ré-équilibre ; soit on utilise les AVL (v. + loin).

Quelques autres algorithmes en Python sur les arbres (suite)

- La fonction *recherche* : cherche l'élément X dans un arbre binaire.

```
def recherche(R, Val) :  
    if vide(R) : return False  
    elif info(R) == val : return True  
    else : return recherche(gauche(R)) or recherche(droit(R))
```

→ Quel est le type du parcours ?

- La fonction *isomorpohe* : vérifie si deux arbres contiennent les mêmes informations dans le même ordre.

```
def isomorpohe(R1, R2) :  
    if vide(R1) and vide(R2) : return True  
    elif not vide(R1) and not vide(R2) :  
        return info(R1) == info(R2) and isomorpohe(gauche(R1), gauche(R2)) and  
            isomorpohe(droit(R1), droit(R2))  
    else : return False
```

ABOH en Python

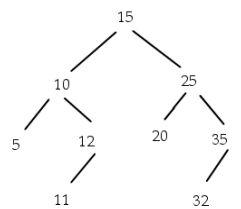
Variante d'arbre : Arbre Binaire Ordonné horizontalement (ABOH)

→ Appelé également Arbre Binaire de Recherche (ABR).

- Relation vérifiée sur chaque noeud d'un ABOH :

$$\text{Max}(\text{info}(ss_arbre_G)) < \text{info}(R) < \text{Min}(\text{info}(ss_arbre_D))$$

- Exemple :



👉 Où se trouve l'élément minimum / maximum (dans un tel arbre) ?

ABOH en Python (suite)

Quelques algorithmes sur les ABOH :

- La fonction de recherche d'un élément X dans un ABOH :

```
def recherche_ABOH(R, Val) :  
    if vide(R) : return False  
    elif info(R) == val : return True  
    elif info(R) > val : return recherche_ABOH(gauche(R))  
    else : return recherche_ABOH(droit(R))
```

- Complexité : si h = la hauteur de l'arbre alors

$h = \lfloor \log(N) \rfloor + 1$ (N = nombre de noeuds) si **arbre équilibré et compacte**.

- La version itérative (transformation aisée de la récursivité) :

```
def recherche_ABOH_iter(R, Val) :  
    trouve=False  
    while not vide(R) and not trouve :  
        if info(R) == val : trouve=True  
        elif info(R) > val : R=gauche(R)  
        else : R=droit(R)  
    return trouve
```

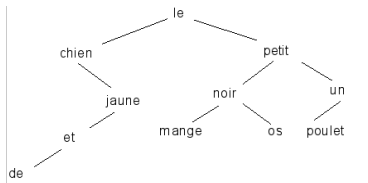
ABOH en Python (suite)

- Les ABOHs sont rarement utilisés pour des insertions.
 - Raison : risque de déséquilibre !
 - Néanmoins, l'insertion doit être prévue :

```
def insere_ABOH(R, Val) :                               # R est un objet mutable
    if vide(R) : return ajouter(R, Val)                # ajout du noeud (Val, vide, vide)
    elif info(R) > val : return insere_ABOH(gauche(R), Val)
    else : return insere_ABOH(droit(R), Val)
```

→ D'autres versions (p.ex. on peut renvoyer un arbre).

- Exemple : est-ce l'ABOH obtenu en insérant les mots de la phrase suivante :
"le petit chien jaune et noir mange un os de poulet" (utilité ?)



Exercice sur les Arbres

Exercice sur les ABOHs :

- Proposer / décide d'une représentation d'ABOH sous Python, P. EX.
 - Par une liste principale + liste des fils
 - Par un tableau (liste) et la relation $i, 2i, 2i + 1, \dots$ Puis
- Lire un texte et constituer l'indexe des mots (comme à la fin d'un livre).

Il est besoin de construire un dictionnaire où chaque lettre de l'alphabet est une entrée pour les mots commençant par celle-ci.

Sur les arbres en général :

- Reprendre l'expression arithmétique vue plus haut (arbres binaires) pour une **évaluation** arithmétique.
- Donner une relation (type $2i, 2i + 1, 2i + 2$) pour un arbre ternaire, quaternaire et quinquenaire.
 - Pour un arbre quaternaire, voir BE2.

Exercice sur les Arbres (suite)

- Pour $k = 3$, la représentation ci-dessus permet de passer :
 - de l'indice *pere* du père aux fils : $3 * \text{pere} - 1, 3 * \text{pere}, 3 * \text{pere} + 1$ (for j in range(k) : $3 * \text{pere} + j - 1$)
 - de l'indice *fils* au père : $\text{pere} = \text{ fils} // 3$; if $\text{ fils} \% 3 == 2$: $\text{pere} += 1$
 - Pour $k = 3$, l'indice 0 de la liste non utilisé.
 - Pour $k = 4$ et l'indice *pere* donné (cf. BE2), le premier fils est à l'indice $4 \text{pere} + 1$, le 2e à l'indice $4 \text{pere} + 2$, le 3e à $4 \text{pere} + 3$ et le dernier à $4 \text{pere} + 4$.
 - Pour retrouver le parent (*pere*) d'un noeud d'indice *fils* :


```
pere=fils // 4
if fils % 4 == 0 : pere -= 1
```
 - Pour $k = 5$ et l'indice *pere* donné, les fils sont en indice $5 \text{pere} - 3, 5 \text{pere} - 2, 5 \text{pere} - 1, 5 \text{pere}, 5 \text{pere} + 1$
 - Pour retrouver le parent (*pere*) d'un noeud d'indice *fils* :


```
pere=fils // 5
if fils % 5 > 1 : pere += 1
```
- ☞ **Rappelons** que pour certains k , l'indice 0 **n'est pas** utilisé.
- Vous pouvez chercher la relation *pere-fils* pour $k > 5$.

Pile et File en Python

- D'autres structures (utilitaires) importantes.
- Voir TAS (plus loin) pour les classes utilisées.

Pile : la classe *Queue* implante une (sous-classe) *Pile* (gestion *LIFO*) :

```
class Queue.LifoQueue(maxsize=0)
```

```
import queue
pile = queue.LifoQueue()
for i in range(5): pile.put(i)
while not pile.empty(): print(pile.get(), end=' ')
print()
# Donne 4 3 2 1 0
```

→ L'élément inséré en dernier est en tête : gestion LIFO (Last-In-First-Out)

- La classe Queue propose en fait plusieurs structures de données (avec des applications en programmation concurrente).

Pile et File en Python (suite)

File (d'attente) : gestion FIFO

- La classe *Queue* implante également une (sous-classe) *file* :

```
class Queue.Queue(maxsize=0)
```

☞ Rappelons qu'une Pile est facilement réalisable avec une simple liste !

- Opérateurs habituels (communs aux *Queue* ou *LifoQueue*) avec notation Objets :
 - **empty()** : test de vacuité
 - **full()** : plein (si on a donné une taille max à la création)
 - **get()** : récupère l'élément et l'enlève de la Pile/File ; exception *queue.Empty* si vide
 - **put()** : ajout (selon *Queue* ou *LifoQueue*) ; exception *queue.Full* si plein (voir *full()*)
 - **qsize()** : nombre actuel d'éléments.

Pile et File en Python (suite)

☞ La classe `file` est utilisable en particulier dans les Threads (structure dite *Thread-Safe*).

Il y a de nombreuses utilisations de cette structure en programmation concurrente.

● Exemple d'utilisation d'une file :

```
import queue
file = queue.Queue()
for i in range(5) : file.put(i)
while not file.empty(): print(file.get(), end=' ')
print()
# 0 1 2 3 4
```

→ Premier arrivé, premier servi : gestion FIFO

☞ Noter `Queue.Queue()` pour une *File* et `Queue.LifoQueue()` pour une *Pile*.

Autres implantations de Pile et File en Python

Pile (LIFO) :

- Certains pensent que sous Python, le type Pile (avec le schéma *last-in, first-out*) est "superflu" puisque une liste remplit les mêmes fonctions :

```

stack = []           # fonction d'initialisation
stack = [3, 4, 5]
stack.append(6)     # fonction empiler(6)
stack.append(7)

stack
# [3, 4, 5, 6, 7]

stack.pop()        # fonction dépiler() qui renvoie sommet(). Attention au TDA
# 7

stack
# [3, 4, 5, 6]

stack[-1]          # fonction sommet(). La pile n'est pas modifiée
# 6

```

→ On peut être d'accord !

Autres implantations de Pile et File en Python (suite)

File (FIFO) :

- Comme pour les Piles, on peut utiliser les listes pour implanter les *Files* (d'attente, avec le schéma *First-in, First-out*).
- Par exemple :

```
"""Module de gestion d'une queue FIFO."""  
  
queue = [] # initialisation  
  
def enqueue():  
    queue.append(int(input("Entrez un entier : ")))  
  
def dequeue():  
    if len(queue) == 0:  
        print("\nImpossible : la queue est vide !")  
    else:  
        print("\nÉlément '%d' supprimé" % queue.pop(0))  
  
def afficheQueue():  
    print("\nqueue :", queue)
```

Autres implantations de Pile et File en Python (suite)

Mieux ? :

- Le type **Deque** de *collections.deque* "double ended queue" (liste à 2 entrées) :

```

from collections import deque

queue_a_2_tetes_haha = deque(["Eric", "Jean", "Michael"])
queue_a_2_tetes_haha.append("Pierre")      # enfiler("Pierre")
queue_a_2_tetes_haha.append("Marie")      # enfiler("Marie")

queue_a_2_tetes_haha.popleft()             # défiler(): renvoie sommet(). Différent du TDA
# 'Eric'

queue_a_2_tetes_haha.popleft()             # défiler(): renvoie sommet(). différent du TDA
# 'Jean'

queue                                     # Ce qui reste
# deque(['Michael', 'Pierre', 'Marie'])

queue_a_2_tetes_haha.pop()                 # On dirait popright = pop comme pour une liste !
# "Marie"

```


Autres implantations de Pile et File en Python (suite)

- Les fonctions de *deque* (extrait de la documentation Python) :

append(x) : Add x to the right side of the deque.

appendleft(x) : Add x to the left side of the deque.

extend(iterable) : Extend the right side of the deque by appending elements from the iterable argument.

extendleft(iterable) : Extend the left side of the deque by appending elements from iterable.

Note : the series of left appends results in reversing the order of elements in the iterable argument.

insert(i, x) : Insert x into the deque at position i.

pop() : Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

popleft() : Remove and return an element from the left side of the deque.

If no elements are present, raises an IndexError.

remove(value) : Remove the first occurrence of value. If not found, raises a ValueError.

copy() : Create a shallow copy of the deque.

reverse() : Reverse the elements of the deque in-place and then return None.

rotate(n) : Rotate the deque n steps to the right. If n is negative, rotate to the left.

Rotating one step to the right is equivalent to : *d.appendleft(d.pop())*.

clear() : Remove all elements from the deque leaving it with length 0.

count(x) : Count the number of deque elements equal to x.

Autres implantations de Pile et File en Python (suite)

index(x[, start[, stop]]) : Return the position of x in the deque (at or after index start and before index stop).

Returns the first match or raises ValueError if not found.

- Deque objects also provide one read-only attribute :
maxlen : Maximum size of a deque or None if unbounded.
- In addition to the above, deques support iteration, pickling, len(d), reversed(d), copy.copy(d), copy.deepcopy(d), membership testing with the in operator, and subscript references such as d[-1].
- Indexed access is $O(1)$ at both ends but slows to $O(n)$ in the middle. For fast random access, use lists instead.
- Starting in version 3.5, deques support `__add__()`, `__mul__()`, and `__imul__()`.

Autres implantations de Pile et File en Python (suite)

Exemple :

```

from collections import deque
d = deque('ghi')           # création
for elem in d:           # Itération
    print(elem.upper())
    """ G H I """

d.append('j')             # Ajout à Droite (= à la fin par défaut)
d.appendleft('f')        # Ajout à Gauche (utile pour une insertion en tête d'une File/Pile)
d                         # montrer la représentation de d
# deque(['f', 'g', 'h', 'i', 'j'])

d.pop()                  # enlever et renvoyer l'élément le plus à droite
# 'j'
d.popleft()              # enlever et renvoyer l'élément le plus à gauche
# 'f'

list(d)                  # présenter "d" comme une liste
# ['g', 'h', 'i']
d[0]                     # l'élément le plus à gauche
# 'g'
d[-1]                    # l'élément le plus à droite
# 'i'

list(reversed(d))        # présenter "d" comme une liste mais à l'envers !
# ['i', 'h', 'g']
'h' in d                 # recherche dans "d"
# True

```

Autres implantations de Pile et File en Python (suite)

```
d.extend('ijkl')           # ajouter plusieurs éléments en même temps
d
# deque(['g', 'h', 'i', 'j', 'k', 'l'])

d.rotate(1)               # rotation à droite
d
# deque(['l', 'g', 'h', 'i', 'j', 'k'])
d.rotate(-1)              # rotation à gauche
d
# deque(['g', 'h', 'i', 'j', 'k', 'l'])

deque(reversed(d))        # créer un nouveau deque avec "d" à l'envers
# deque(['l', 'k', 'j', 'i', 'h', 'g'])

d.clear()                  # vider "d"
d.pop()                    # On ne peut pas enlever un élément de "d" vide
# IndexError: pop from an empty deque

d.extendleft('abc')       # extendleft() met à l'envers l'ordre des éléments ajoutés
d
# deque(['c', 'b', 'a'])
```

Collection et container

- site <https://docs.python.org/3/library/collections.html>

Et <https://docs.python.org/3.1/library/collections.html#module-collections>

8.3. *collections* – Container datatypes

Source code: *Lib/collections/___init___*.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.

namedtuple() factory function for creating tuple subclasses with named fields

deque list-like container with fast appends and pops on either end

ChainMap dict-like class for creating a single view of multiple mappings

Counter dict subclass for counting hashable objects

OrderedDict dict subclass that remembers the order entries were added

defaultdict dict subclass that calls a factory function to supply missing values

UserDict wrapper around dictionary objects for easier dict subclassing

UserList wrapper around list objects for easier list subclassing

UserString wrapper around string objects for easier string subclassing

Changed in version 3.3: Moved Collections Abstract Base Classes to the collections.abc module. For backwards compatibility, they continue to be visible in this module as well.

TAS ou Heap

- Un TAS (HEAP) est un arbre binaire ordonné **verticalement**.

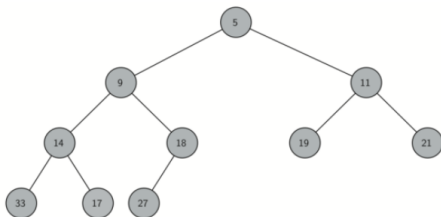
- Dans un tel arbre, la relation d'ordre est (cf. *min_heap*) :

$$\text{info}(\text{noeud}) \leq \min \text{info}(\text{gauche}) \text{ et } \text{info}(\text{noeud}) \leq \min \text{info}(\text{droit})$$

- Pour le *max_heap*, on aura symétriquement

$$\text{info}(\text{noeud}) \geq \max \text{info}(\text{gauche}) \text{ et } \text{info}(\text{noeud}) > \max \text{info}(\text{droit})$$

- Par défaut, on considère les *min_heaps* comme dans figure suivante :



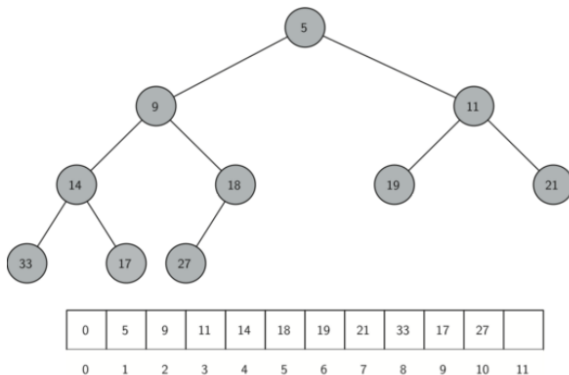
TAS ou Heap (suite)

- Un *min_heap* binaire (*binary heap*) est également appelé une *file binaire de priorités* (*binary priority queue*).
 - Une file binaire de priorités est (en général) implantée par un TAS.
- L'appellation *file de priorité* vient du fait que les informations stockées représentent (habituellement) des priorités de traitement (comme devant un guichet)
- Un TAS est en général représenté en mémoire par un tableau avec les indices $2i$ et $2i + 1$ (i doit commencer à 1).
- Un TAS peut être implantée à l'aide des listes triées.
 - Dans ce cas, l'insertion sera $O(N)$ et le tri $O(N \cdot \log(N))$.
 - Par contre, dans les Tas binaires (qui sont par définition ordonnés), l'insertion et la suppression (enfiler / défiler) sont $O(\log(N))$.

TAS ou Heap (suite)

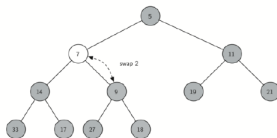
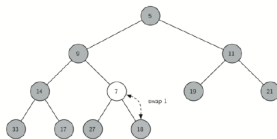
Heap et sa représentation par tableau (liste) :

- Un arbre binaire complet et sa représentation par table / liste



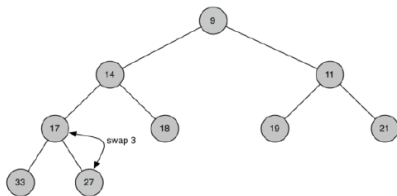
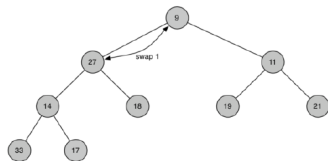
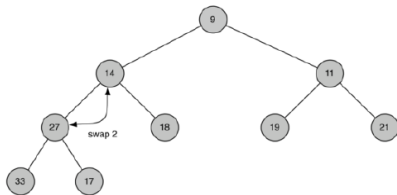
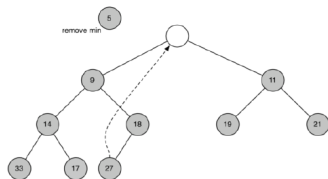
TAS ou Heap (suite)

- Exemple d'insertion de 7



TAS ou Heap (suite)

- Exemple du retrait du min et la réorganisation :

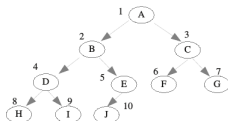


TAS en Python

TAS (ou *Heap Queue*) en Python : une variante d'une File de priorité.

- Le module *heapq* implante la méthode *min-heap-sort* pour le tri des listes Python.
- Un TAS est un arbre binaire avec la relation d'ordre $<$ (**minimum** au sommet).
- Son implantation utilise un tableau/une liste (soit *heap*) où $\forall k \geq 0$:

$$\text{heap}[k] \leq \text{heap}[2k + 1] \quad \text{et} \quad \text{heap}[k] \leq \text{heap}[2k + 2].$$
- ☞ Dans Python, l'élément d'indice 0 existe ! (voir son implantation).
- Les éléments absents sont considérés comme infinis.
- Dans un tel arbre, le minimum est toujours au sommet.



indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	..
valeur	*	A	B	C	D	E	F	G	H	I	J	...			

TAS en Python (suite)

On peut transformer une liste L en *heapq* par *heapq.heapify(L)*.

Par exemple :

```
import heapq

heap = []
heapq.heappush(heap, (1, 'one'))
heapq.heappush(heap, (10, 'ten'))
heapq.heappush(heap, (5, 'five'))

for x in heap:
    print(x)

print()
# (1, 'one')
# (10, 'ten')
# (5, 'five')

heapq.heappop(heap)      # Enlever le minimum
# (1, 'one')

for x in heap:
    print(x)
print()

#(5, 'five')
# (10, 'ten')

print(heap[0])          # Le minimum actuel
# (5, 'five')
```

TAS en Python (suite)

- Le même exemple modifié : remarquer la destruction du TAS (par un ajout sauvage !) puis sa reconstruction :

```
import heapq

heap = [(1, 'one'), (10, 'ten'), (5, 'five')]
heapq.heapify(heap)
for x in heap:
    print(x)

# (1, 'one')
# (10, 'ten')
# (5, 'five')

heap[0] = (9, 'nine')      # On modifie le 2e élément
for x in heap:            # ZZ : CE N'EST PLUS UN TAS
    print(x)

# (9, 'nine')
# (10, 'ten')
# (5, 'five')

heapq.heapify(heap)      # ZZ : CE REDEVIENT UN TAS
for x in heap:
    print(x)

# (5, 'five')
# (10, 'ten')
# (9, 'nine')
```

TAS en Python (suite)

- La différence d'un *heapq* par rapport à une File de priorité est
 - *heapq* est (seulement) un *min-heap*
 - Toute intervention (sauvage) sur le tableau sous-jacent risque de détruire le TAS
 - ➔ Mais on peut réparer (cf. l'exemple ci-dessus).
- Exemple avec *Heapq* (de la bibliothèque) : `min_heap` et `max_heap` :

```
import heapq
listForTree = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
heapq.heapify(listForTree)           # Pour min_heap (par défaut)
heapq._heapify_max(listForTree)     # Pour max_heap

listForTree
# [15, 11, 14, 9, 10, 13, 7, 8, 4, 2, 5, 12, 6, 3, 1]
```

TAS en Python (suite)

TAS : une autre possibilité dans Python :

- Sous Python, la classe **Queue** propose la sous-classe *PriorityQueue* qui implante un TAS.
- La classe *Queue* elle-même implante une file (FIFO) utilisable (en particulier dans les *Threads*).
- Sous Python, **PriorityQueue** est implantée à base du module **heapq** mais propose également une interface (classique) **queue**.
 - ➔ Cette interface est en grande partie la différence majeure entre les deux : une conception Objet (et sa notation pour *PriorityQueue*) vs. le passage d'une liste en paramètre (que les opérateurs de *heapq* peuvent modifier).
 - ➔ Autrement dit, les fonctions de *heapq* demandent un paramètre vs. une utilisation par la notation objet pour *PriorityQueue*

TAS en Python (suite)

Exemples (avec queue et sa sous-classe *PriorityQueue*)

```
import queue as Q

q = Q.PriorityQueue()
q.put(10)
q.put(1)
q.put(5)
while not q.empty():
    print(q.get())

# 1 5 10
```

☞ Remarquer l'ordre des insertions et celui d'affichage.

- Si on veut ajouter des tuples :

```
import queue as Q

q = Q.PriorityQueue()
q.put((10, 'ten'))
q.put((1, 'one'))
q.put((5, 'five'))
while not q.empty(): print(q.get())

# (1, 'one') (5, 'five') (10, 'ten')
```

→ La comparaison se fait sur les premiers éléments des couples.

En savoir plus : Transformation de la récursivité

Rappels et compléments sur la récursivité (induction)

- Avantages

- Analyse naturelle et intuitive de multiples problèmes
- Apport de la preuve des algorithmes facilité
- Quasi obligation d'utilisation dans certains problèmes (en particulier traitant des structures récursives comme les arbres, les graphes...)
- Similarité avec l'Induction

- Inconvénients

- Plus coûteux en espace mémoire et en temps de calcul si mal écrit.
- Moins accessible : demande un effort d'exercices

En savoir plus : Transformation de la récursivité (suite)

- Solution aux inconvénients :

Il existe des techniques fiables de transformation des schémas récursifs en schémas itératifs.

→ Ces techniques sont prouvées justes et complètes.

- La démarche à suivre :

- écrire un algorithme récursif

- le tester, valider, prouver (dans les applications critiques)

- lui appliquer les techniques de transformation pour obtenir une version itérative

☞ Un soucis : ces techniques ne sont pas à 100% automatisables (pour l'instant !)

→ Dans des cas non triviaux, une intervention humaine peut être nécessaire.

Récursivité terminale

- Il existe un lien direct entre un schéma itératif et celle récursive terminale.

Exemple : schéma de la récursivité terminale :

```
f_rec(X) =
  si p(X) alors a(X)
  sinon
    b(X)
    f_rec(nouveau(X))
  Finsi;
```

- Un exemple utilisant ce schéma : affichage d'une chaîne caractère par caractère

```
afficher(Ch) =
  si Ch="" alors RIEN
  sinon
    écrire(tête(Ch))
    afficher(Reste(Ch))
```

- La traduction du schéma récursif terminal sera de la forme :

```
f_iter(X) =
  Tant que p(X) = faux faire
    b(X);
    X := nouveau(X);
  Fin Tq;
  a(X);
```

Récursivité terminale (suite)

Remarque :

- Les schémas *Tant que* et *répéter* sont équivalents :

→ On propose un schéma *répéter* équivalent :

```
f_iter(X) =  
  si p(X) = faux alors  
    répéter  
      b(X);  
      X := nouveau(X);  
    jusqu'à p(X)=vrai;  
  a(X);
```

- On en déduit :

L'évaluation d'une fonction récursive terminale est une itération.

Récursivité terminale (suite)

- Remarque : un schéma récursif terminal est de la forme :

```
P(X) =  
  si cond(X) alors  
    action_A(X)  
    P(nouveau(X))  
  sinon  
    action_B(X)  
  Finsi ;
```

- Cet algorithme peut être résumé par l'expression de la logique des prédicats :

$$P(x) : [Cond(x) \wedge Action_A(x) \wedge P(\rho(x))] \vee [\overline{Cond(x)} \wedge Action_B(x)]$$

Où la fonction $\rho(x)$ (notée *nouveau(x)*) modifie la valeur de x pour la faire converger vers $Cond(x) = vrai$

- De cette expression, on peut tirer l'arbre de développement suivant :

Récursivité terminale (suite)

- La séquence d'actions effectuées :

$cond(x_0), A(x_0), cond(x_1), A(x_1), cond(x_2), A(x_2),$

$\dots, cond(x_n), A(x_n), \overline{cond(x_{n+1})}, B(x_{n+1})$

Où x_{i+1} est obtenue par $\rho(x_i)$

- Dont on peut déduire le schéma :

$i \leftarrow 0$

Tant que $cond(x_i)$

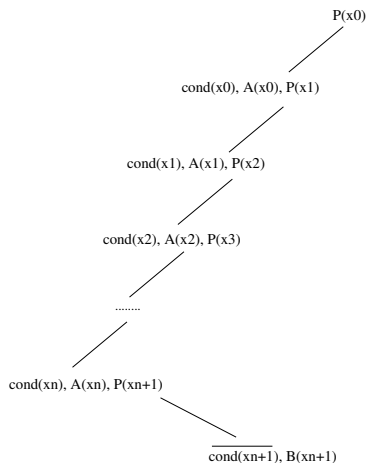
$A(x_i)$

$x_{i+1} \leftarrow \rho(x_i)$

Fin TQ

// ici : on a : $\overline{cond(x_{n+1})}$

$B(x_{n+1})$



Récursivité terminale (suite)

Avec Python :

```
def p1_rec (i) :  
    if i > 0 :  
        action1(i)                # ne contenant pas d'appel à p1_rec  
        p1_rec(i-1)              # (i non modifié par action1)  
    else :  
        action2(i)                # ne contenant pas d'appel à p1_rec
```

- La solution itérative supprime la récursivité terminale.

```
def p1_iter (i) :  
    j=i  
    while j > 0 :  
        action1(j)  
        j = j-1  
    action2(j);
```

Récursivité non terminale

Cas de récursivité non terminale :

```
def p2_rec (i) :  
    if i > 0 :  
        p2_rec(i-1)  
        action1(i)           # ne contenant pas d'appel à p2_rec  
    else :  
        action2(i)         # ne contenant pas d'appel à p2_rec
```

- La version itérative utilise une **pile** (comme une *pile d'assiettes*) avec une mode de gestion :
le dernier arrivé est le premier servi.
- Les opérateurs habituellement définis sur les piles sont :
 - *vide* : la constante pile vide
 - *empiler(élément, pile)* : procédure d'ajout d'un élément au sommet ,
 - *dépiler(pile)* : procédure d'extraction du sommet (fonction partielle ?)
 - *sommet(pile)* : fonction d'interrogation de l'élément du sommet (partielle ?),
 - ...

Récursivité non terminale (suite)

Rappel du schéma récursif :

```
def p2_rec (i) :  
    if i > 0 :  
        p2_rec(i-1)  
        action1(i)           # ne contenant pas d'appel à p2_rec  
    else :  
        action2(i)         # ne contenant pas d'appel à p2_rec
```

• Et le schéma itératif équivalent :

```
def p2_iter (i) :  
    j=i;  
    pile = [];  
    while i > 0 :  
        empiler(j, pile)  
        j=j-1  
  
    while pile != vide :  
        j=sommet(pile)  
        depiler(pile)  
        action1(j)  
    action2(j)
```

Récursivité non terminale (suite)

- Remarque : pour un schéma récursif non terminal de la forme :

```

P(X) =
  si cond(X) alors
    P(nouveau(X))
  action_A(X)
  sinon
    action_B(X)
  Finsi ;
  
```

L'expression associée en logique des prédicats sera (cf. la récursivité terminale) :

$$P(x) : [Cond(x) \wedge P(\rho(x)) \wedge Action_A(x)] \vee [\overline{Cond(x)} \wedge Action_B(x)]$$

Où la fonction $\rho(x)$ (notée *nouveau(x)*) modifie la valeur de x pour la faire converger vers $Cond(x) = \text{vrai}$

- ☞ On constate clairement que $Action_A(x)$ a besoin de l'ancienne valeur de x modifiée par $\rho(x)$. Par exemple, si $\rho(x) : x = x + 1$, il suffira de restaurer l'ancienne valeur de x en faisant $x - 1$ mais dans le cas général, on doit stocker les différentes valeurs de x dans une pile de sorte que l'on puisse les récupérer et leur appliquer $Action_A$.

- Une étude de l'arbre de développement de ce schéma montre que :

Récursivité non terminale (suite)

- La séquence d'actions effectuées :

$cond(x_0), cond(x_1), cond(x_2), \dots, cond(x_n)$

$\overline{cond(x_{n+1})}, B(x_{n+1})$ ←-- la 1e action est $B(x_{n+1})$

$A(x_n), A(x_{n-1}), \dots, A(x_2), A(x_1), A(x_0)$

Où x_{i+1} est obtenue par $\rho(x_i)$

- Dont on peut déduire le schéma :

$i \leftarrow 0$; $Pile \leftarrow vide$

Tant que $cond(x_i)$

empiler(x_i , $Pile$)

$x_{i+1} \leftarrow \rho(x_i)$

Fin TQ

// ici : on a : $\overline{cond(x_{n+1})}$

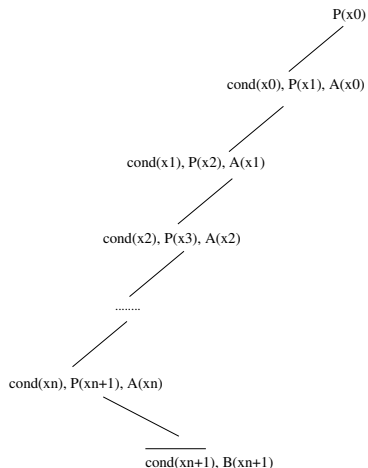
$B(x_{n+1})$ ←-- la 1e action

Tant que $est_vide(Pile) = faux$

$x_i \leftarrow sommet(Pile)$; $depiler(Pile)$

$A(x_i)$ ←-- la 1e valeur est $A(x_n)$

Fin TQ



Récursivité non terminale : exemples

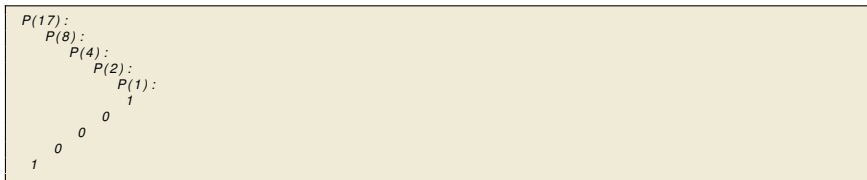
Exemple : conversion décimale-binaire

- On souhaite afficher la séquence de bits (0 ou 1) qui représente la forme binaire d'un entier positif X.
- Par exemple, l'entier 5 donnera lieu à la séquence 101.
- Le code ci-dessous est équivalent aux divisions successives que l'on ferait pour atteindre manuellement le même objectif.

```
def affiche_bin_rec(X)
    if X==1 : print('1')
    else :
        affiche_bin_rec(X//2)
        if (X %2 == 0) : print('0')
        else : print('1')
```

Récursivité non terminale : exemples (suite)

- Une trace pour $X=17$ donnera la chaîne de bits 10001.
- La trace des appels ressemble à :



- Résultat pour $17 = 10001$
- Selon le schéma général de la transformation d'une récursivité non terminale, la version itérative de cette fonction sera (on utilise une liste Python pour représenter la Pile) :

```

def affiche_bin_iter(X) :
    Pile=[]
    while (X > 1) :
        Pile.append(X)
        X=X//2
    if (X==1) : print("1")
    while (Pile != []) :
        X=Pile.pop() # cette fonction fait sommet + dépile !
        if (X % 2 == 0) : print("0")
        else : print("1")
  
```

Récursivité non terminale : exemples (suite)

Exemple factorielle :

```
def FACT(N) :  
  if (N>1) :  
    temp= FACT(N-1)  
    Val= temp * N  
  else :  
    Val =1  
  return Val;
```

- La multiplication ne peut avoir lieu que lorsque la valeur Fact(N-1) est disponible, ce qui fait de cet appel un cas non terminal.

- Le schéma itératif :

```
def FACT_iter (N) :  
  Pile=[]  
  while (N>1) :  
    Pile.append(N)  
    N=N-1  
  
  val=1  
  while (Pile != []) :  
    N = Pile.pop()      # cette fonction fait sommet + dépile !  
    val= val * N  
  
  return val;
```

Récursivité non terminale : exemples (suite)

Cas particulier d'énumération de valeurs

- Ci-dessus, on a indiqué que dans certains cas numériques, on peut éviter la Pile.
- Exemple de transformation de la fonction factorielle sans utiliser une pile. Dans certains cas, on peut éviter d'utiliser une pile si son utilité est simplement de contenir différentes valeurs successives (ou régulier, p.e. les nombres pairs) d'un entier.
- Rappel de l'exemple factorielle où l'on constate bien la place de l'appel récursif :

```
def FACTORIELLE (N) :  
  if (N>1) :  
    temp= FACTORIELLE (N-1)  
    Val= N * temp  
  else :  
    Val =1  
  return Val
```

- La multiplication ne peut avoir lieu que lorsque la valeur $FACTORIELLE(N-1)$ est disponible, ce qui fait de cet appel un cas non terminal.

Récursivité non terminale : exemples (suite)

- La version itérative obtenue par une transformation qui utilise une Pile.

```
def FACTORIELLE_iter(N) :
  Pile = []
  while (N>1) :
    Pile.append(N)
    N=N-1
  val=1
  while (Pile != []):
    N=Pile.pop()
    val= val * N
  return val;
}
```

- Du fait de faire N-1 fois '-1' sur N puis autant de fois retirer ces valeurs de la Pile avant de les multiplier, on peut proposer une solution qui n'utilise pas une pile :

```
def FACTORIELLE_iter_sans_pile(N) :
  while (N>1) :
    N=N-1
    compteur = compteur+1 # compte (init 0) est le nbr de fois où on a fait l'action "N=N-1"

  val =1;          # N vaut 1
  while (compteur > 0) :
    N=N+1
    val= val * N
    compteur = compteur -1
  return val;
```


Exemples non triviaux

Exemple du schéma Hanoï :

- L'exemple des tours de Hanoï est doublement intéressant car il contient à la fois une récursivité terminale et une non terminale.

```
def HANOI(N, dep, arr, inter) :  
    if (n > 0) :  
        HANOI(n-1, dep, inter, arr)  
        print( dep, "→", arr)  
        HANOI(n-1, inter, arr, dep)
```

- On procède en 2 étapes :
 - élimination de la récursivité terminale puis celle de la récursivité non terminale.

Hanoï : élimination de la récursivité terminale

- En suivant les indications ci-dessus, on obtient la version hanoi_iter1 :

```
def hanoi_iter1(N, dep, arr, inter) :  
    while (N>0) :  
        hanoi_iter1(N-1, dep, inter, arr)  
        print( dep, "→", arr)  
        N=N-1  
        permuter(dep, inter)
```

Exemples non triviaux (suite)

- On constate qu'il est nécessaire de permuter *dep* et *inter* du fait de l'appel récursif terminal *hanoi_rec(N-1, inter, arr, dep)*.

☞ Noter que l'appel récursif non terminal s'inscrit dans une boucle `while`.

→ Dans les cas similaires, on a recours à la suppression du *while* (ou *for*, etc.)

Hanoï : élimination de la récursivité non terminale

- Pour simplifier cette transformation, considérons la version suivante de *hanoi_iter1* où la boucle `while` a été remplacée par un *if + goto*.

```
void hanoi_iter1_avec_if_et_label(N, dep, arr, inter) :  
label:  
    if (N>0) :  
        hanoi_iter1_avec_if_et_label(N-1, dep, inter, arr)  
        print( dep, "→", arr)  
        N=N-1  
        permuter(dep,inter)  
        goto label
```

- De cette manière, on peut appliquer le schéma de transformation vu ci-dessus.

Exemples non triviaux (suite)

- En utilisant une pile, on obtient (simplifions : **supposons** disposer des labels en Python) :

```

void hanoi_iter2_avec_label(N, dep, arr, inter) :
    Pile = []
    label : while (N>0) :
        Pile.append([N, dep, arr, inter])
        N=N-1
        permuter(arr, inter)      # du fait de l'appel récursif

    if (Pile != []):
        data=Pile.pop()          # devenu "if" à cause du goto label ci-dessous
        print( dep, "→", arr)   # Mêmes remarques que ci-dessous
        N=data[0]-1             # équivalent à N = N - 1
        permuter(dep, inter)
        goto label

```

- On peut maintenant donner la version qui nous débarrasse du label (remplacé par *while*) :

```

void hanoi_iter2(N, dep, arr, inter) :
    Pile = []
    while True :
        while (N>0) :
            Pile.append([N, dep, arr, inter])
            N=N-1
            permuter(arr, inter)  #du fait de l'appel récursif

        if (Pile != []):
            data=Pile.pop()
            print( dep, "→", arr)
            data[0]-1             # équivalent à N = N - 1
            permuter(dep, inter)

    if (N <= 0 or Pile==[]) : break

```

Exemples non triviaux (suite)

Exemple des primes :

- On rappelle l'exemple de distribution des primes pour une transformation.
- On veut distribuer une prime de N francs entre M les employés d'une entreprise.
- Soit $e_1 \dots e_m$ la série ordonnée qui représente le personnel dans l'ordre de leur ancienneté dont la distribution doit tenir compte : le montant affecté à e_i doit être $\geq e_{i+1}$.
- Certaines primes peuvent être nulles.
- Déterminer le montant affecté à chaque personne ainsi que le nombre de manières de répartir N en M sommants. .

Rappel de la solution récursive :

- La solution récursive est rappelée en séparant les différentes sections du code Python :
 - Les tests ont été séparés et placés dans une fonction `cond()` pour plus de clarté.

Exemples non triviaux (suite)

```

def cond(Montant_restant, Empl_restants, Last_montant) :
    if (Empl_restants < 1) or (Montant_restant < 0) : return False
    elif Empl_restants == 1 :
        if Montant_restant > Last_montant : return False
        else : return False # Montant_restant <= Last_montant
    elif Montant_restant == 0 : return False # Empl_restants > 1
    return True

def prime3(Montant_restant, Empl_restants, Last_montant) :
    Res=0
    if cond(Montant_restant, Empl_restants, Last_montant) :
        # » Action A(x) «
        Min_ = min(Last_montant, Montant_restant) # min de 2 valeurs

        N = 0 # Pour la boucle, ne fait pas partie de A(x), noter les "vraies" actions
        # Partie récursive non terminale
        for K in reversed(range(1, Min_+1)) :
            # K donné à l'employé actuel, on va distribuer le reste
            temp = prime3(Montant_restant-K, Empl_restants-1, K)

            # » Action B(x) «
            N += temp

        Res=N
    else :
        # » Action D(x) «
        if (Empl_restants < 1) or (Montant_restant < 0) : Res=0
        elif Empl_restants == 1 :
            if Montant_restant > Last_montant : Res=0
            else : Res=1 # Montant_restant <= Last_montant
        elif Montant_restant == 0 : Res=1 # Empl_restants > 1
        # » Action E(x) «
    return Res

```

Exemples non triviaux (suite)

- La schéma récursif généralisé est alors de la forme :

```
def P_rec_non_terminale(X) :  
    if cond(X) :  
        Action A(x)  
        for K in range(Inf..Sup) :  
            P_rec_non_terminale(nouveau(X))    # le vecteur X modifié  
            Action B(x)  
  
    else :  
        Action D(x)  
  
    Action E(x)
```

Exemples non triviaux (suite)

- Le schéma itératif correspondant sera (voir aussi le schéma Hanoi) :

```
def P_iteratif(X) :  
    Pile=[]  
    while True :  
        if cond(X) :           # Condition pour empiler des valeurs  
            Action A(x)  
  
            for K in range(Inf..Sup) :  
                # Empiler différentes configurations issues du vecteur X  
                # On les traite ensuite (noter nouveaux(X) empilé)  
                empiler(Pile , nouveau(X))  
  
            Action D(x)  
  
            Pile_vider = (Pile == [])  
            if (Pile != []) :   # Simple précaution avant de dépiler  
                X = sommet(Pile)  
                depiler(Pile)  
  
            Action B(x)  
            if Pile_vider : break # Un test (Pile != []) ne suffit pas car on  
                                # vient peut être de dépiler et vidé la Pile  
    Action E(x)
```

Exemples non triviaux (suite)

```

def prime_iter(Montant_restant, Empl_restants, Last_montant) :
    Res = 0
    Pile = []
    while True :
        if cond(Montant_restant, Empl_restants, Last_montant) :

            # Action A(x)
            Min_ = min(Last_montant, Montant_restant)

            # Appel Récursif non terminale transformée
            for k in reversed(range(1, Min_+1)) :
                Pile.append([Montant_restant-k, Empl_restants-1, k])

            # Action D(x)
            N=0
            if (Empl_restants < 1) or (Montant_restant < 0) : N = 0
            elif Empl_restants == 1 :
                if Montant_restant > Last_montant : N = 0
                else : N = 1 # Montant_restant <= Last_montant
            elif Montant_restant == 0 : N = 1 # Empl_restants > 1

            Pile_vide= (Pile == [])
            if (Pile != []) :
                [Montant_restant, Empl_restants, Last_montant] = Pile.pop()

            # Action B(x)
            Res += N
            if Pile_vide : break

            # Action E(x)
            return Res

print(prime_iter(10,5,10)) # 30 OK

```


Exemples non triviaux (suite)

Remarques :

- Il n'y a pas de transformation automatisée à 100%, sauf dans les cas simples.
- Sachant qu'il y a une grande diversité d'algorithmes récursifs (il y a autant de versions que d'algorithmes différents), il faut adapter le schéma itératif aux particularités du schéma récursif en main.

Addendum : Parcours itératif d'arbre binaire

S'obtient par la transformation de schéma récursif.

Exemple : traduction itérative du parcours récursif infixé.

```
Procédure Infixe ( R : Arbre_bin ) =  
  Début  
    Si non est_vide(R) alors  
      Infixe (gauche(R));  
      traiter (info(R));  
      Infixe (droit(R));  
    Fin si;  
  Fin Infixe ;
```

● La première étape de transformation supprime la récursivité terminale :

```
Procédure infixe1 (Racine : Arbre_bin) =  
  R ← Racine : Arbre_bin;  
  Début  
    Tant que non est_vide(R)  
      infixe1 (gauche(R))  
      traiter (info(R));  
      R ← droit(R);  
    Fin Tq;  
  Fin infixe1;
```

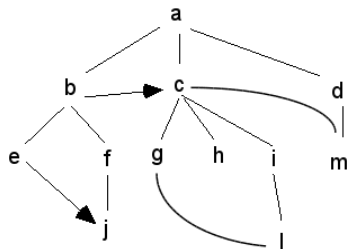
Addendum : Parcours itératif d'arbre binaire (suite)

- La seconde étape supprime la récursivité non terminale à l'aide d'une Pile.
- On utilisera les opérateurs habituels de la Pile :

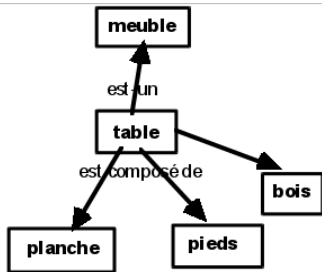
```
Procédure infixe2(Racine : Arbre_bin) =  
R ← Racine : Arbre_bin;  
P ← Pile_vide : Pile;  
Début  
  Tant que non_est_vide(R) ET non Est_vide(P)  
    Tant que non_est_vide(R)  
      Empiler(P, R);  
      R ← gauche(R);  
    Fin Tq;  
    Dépiler(P, R);  
    traiter(info(R));  
    R ← droit(R);  
  Fin Tq;  
Fin infixe2
```

Graphes

- Exemples :



*un graphe représentant des liens
entre différents nœuds (villes)*



*un graphe représentant des
liens d'héritage et de composition*

Graphes (suite)

Quelques exemples d'applications :

- Recherche de chemin : Dijkstra, Floyd Warshall, etc
 - Table de routage : chemins entre les routeurs, recherche de meilleures voies.
- Efficacité des pipelines : FLUX, MST
- Messagerie : le voyageurs de commerce , trajet d'un facteur
- Réseaux de communication : MST
- Gestion du trafic : problème de FLUX, chemins d'encombrement minimum
- Navigation aérienne (les avions dans des couloirs au ciel !)
- Le système de transport fermé (circuit fermé) : livraison de marchandises, TSP.
- Câblage de circuits imprimés
- etc...

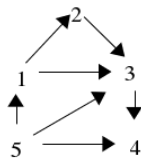
Graphes (suite)

Quelques définitions

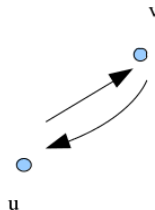
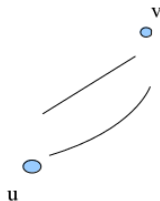
- Un graphe $G=(V, E)$

V : ensemble de noeuds (vertex)

$E \in (V \times V)$: ensemble d'arêtes ou arcs (edges)



Un Digraphe

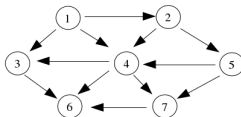


Graphes (suite)

- Chaque **arc** = une paire $(v, w) \in E, v, w \in V$
- Si (v, w) est **ordonné**, on aura un graphe **orienté** (*digraphe*)
 → *directed graph*
- w est **adjacent** de v si $(v, w) \in E$
- Dans un graphe non orienté (undirected), $(v, w) \sim (w, v)$ v et w sont adjacents
- Dans certains problèmes, les noeuds représentent les *variables* et les arcs les *relations*.
- **Poids** (*weight*, pondération) : valeur d'un arc/arête
- **Chemin** (branche, path) : w_1, w_2, \dots, w_n tel que $(w_i, w_{i+1}) \in E$
 $\text{chemin}((X_1, \dots, X_k), (V, E)) \equiv (X_1, X_2) \in E \wedge \dots \wedge (X_{k-1}, X_k) \in E$
- **Longueur** d'un chemin contenant N noeuds = nombre d'arcs = $n-1$
- Un noeud Y est **accessible** depuis un noeud X s'il existe un chemin de X vers Y :
 $(X, Y) \in E \vee \exists Z_1, \dots, Z_k : \text{chemin}((X, Z_1, \dots, Z_k, Y), (V, E))$
- Un **arbre** : un graphe acyclique connexe
- Un graphe est **dense** si $|E| = O(|V|^2)$ voir les annexes pour un complément

Graphes (suite)

• Exemple de graphe



Graphe G1 : graphe orienté du trafic

Représentation d'un graphe :

- Par une matrice carrée
- Par un tableau/liste principal + tableau/liste des adjacents
- Par un Dictionnaire
- etc.

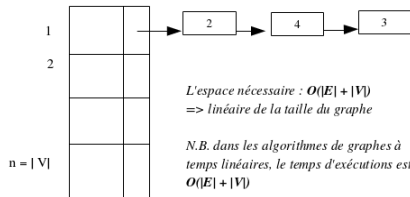
- Selon la nature du graphe (information contenue, orienté ou non, valué ou non), les structures peuvent être plus ou moins complexes.

Représentation abstraite des graphes

- Représentation par une matrice (ou liste de listes)

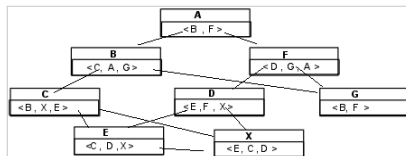
	A	B	C	D	E	F	G
A							
B			V				
C		F					
D		15			3		
E				-1			
F							
G							

- Représentation par un dictionnaire (avec listes d'adjacents)



Représentation abstraite des graphes (suite)

- Un exemple de graphe ...



- .. Et sa représentation par un dictionnaire ou liste de listes :

→ La colonne 1 contient toute information

noeud et ses données *liste d'indices des adjacents*

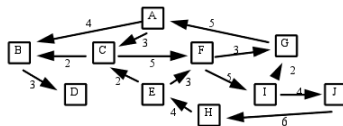
	<i>Nom et données du noeud numéro 1</i>	<i>< i, k, ></i>
1		
i		
k		
N		

☞ De manière similaire, au lieu de dupliquer les informations des noeuds, on utilise la synonymie (de Python) : ne pas copier les listes en profondeur.

Représentation abstraite des graphes (suite)

- Une représentation de graphe $G=(V, E)$ par un tableau / liste principale (contient les informations) et une matrice d'adjacents + couts.

1	A	1	2	4
2	B	1	3	3
3	C	2	4	3
4	D			
5	E			
6	F			
7	G			
...				



La table gauche représente l'ensemble des noeuds V et la table droite représente l'ensemble E . Dans chacune des lignes de la table E , on a l'arc $e_i \in E = (V_{debut} \times V_{fin})$ suivi du cout de cet arc.

Représentation abstraite des graphes (suite)

Plus concrètement :

- Le graphe $G(V,E)$ (avec V : ensemble des noeuds, E : ensemble d'arcs/arêtes) est représenté par ces deux tables.
- Habituellement, la table V contient toutes les informations sur les noeuds (p. ex. une ville, sa population, sa superficie, ...).
- La table E n'a pas besoin de répéter ces informations et se contente de contenir le nom du successeur, ou mieux, l'indice du successeur dans la table V .
- Les flèches rouges dans la figure ci-contre renvoient vers le noeud successeur dans V : ce renvoi se fait via le nom du *successeur_i* (le même nom que dans V) ou via l'indice du successeur dans V .
- Si une pondération des arcs (arêtes) est présente, chaque case de E sera un couple (*noeud_succ*, *poids*).

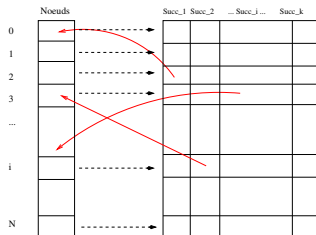


Table V

Table E (les arcs/arêtes)

Parcours général de graphes

- Deux types majeurs de parcours :

1- En profondeur : avantages en inconvénients

2- En largeur : avantages et inconvénients

Il y a également des parcours ad-hoc (B&B, A^* , etc)

- Des deux types de parcours notables, on peut obtenir des parcours variés (en particulier dans le cadre d'un parcours en largeur).

- **Remarque** : comme pour les arbres et selon le "moment" où l'on traite l'information d'un noeud, on a différents types de traitements : pré-ordre, post-ordre et mi-ordre.

Le principe de parcours récursif en profondeur (pré-ordre)

- Traiter chaque noeud puis traiter son premier adjacent récursivement avant de traiter les autres adjacents,
- Éviter de retraiter un noeud en marquant les noeuds visités

Parcours général de graphes (suite)

Parcours en profondeur :

- Ce parcours est semblable au parcours en profondeur des arbres.
- Il est en général assez performant mais si le graphe contient un cycle "à gauche", alors aucune réponse ne pourra être produite.
- Pseudo code du parcours en profondeur (version de base, **sans marquage**) :

```
Procédure profondeur ( G : ref Noeud ) =  
Début  
  Si Non est_vide(G) alors  
    traiter(noeud_courant(G)); // un traitement quelconque  
    Pour X dans adjacents(noeud_courant(G))  
      profondeur (X);  
    Fin pour;  
  Fin si;  
Fin profonde ;
```

- Ce parcours ne mémorise rien et peut donc entrer dans une boucle infinie en cas de circuit ou une boucle (loop).

Parcours général de graphes (suite)

- Parcours récursif en profondeur **avec marquage**

```
Procédure profondeur (G : ref Noeud) =  
Début  
  Si Non est_vider(G) alors  
    marquer(noeud_courant(G));           # on marque tout de suite et ...  
    traiter(noeud_courant(G));          #traitement quelconque  
    Pour X dans adjacents(noeud_courant(G))  
      Si X n'est pas marqué             # ... on contrôle ici  
        Alors profondeur(X);  
      Fin si;  
    Fin pour;  
  Fin si;  
Fin profondeur ;
```

- Ce parcours marque les noeuds déjà visités pour ne pas les réessayer.
- Les mécanismes de marquage :
 - marquage à l'intérieur du noeud
 - marquage par une structure de données externe au graphe (un tableau, ...)

Parcours général de graphes (suite)

- Une autre manière de parcourir le graphe en marquant les noeuds est :

```

Procédure profondeur (G : ref Noeud) =
Début
  Si Non_est_vider(G) ET noeud_courant(G) n'est pas marqué           # On contrôle en entrant
  Alors
    marquer(noeud_courant(G));           # et on marque ici
    traiter(noeud_courant(G));           # traitement quelconque
    Pour X dans adjacents(noeud_courant(G))
      profondeur(X);
    Fin pour;
  Fin si;
Fin profondeur ;
  
```

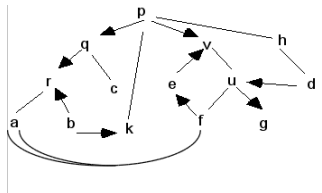
- Trace de parcours en profondeur de **p** à **u** :

profondeur(p) → profondeur(q) →

profondeur(r) → profondeur(a) →

profondeur(f) → profondeur(e) →

profondeur(v) → profondeur(u)



Parcours général de graphes (suite)

● Principe de l'algorithme itératif en Profondeur :

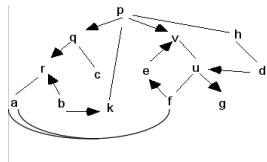
➔ On a besoin d'une pile (LIFO) pour simuler la pile machine (expliquer).

```

Procédure profondeur_iteratif(G)
  Pile=vide
  empiler(noeud_courant(G))
  Tant que NON est_vide(Pile)
    Noeud ← dépiler(Pile)
    Si NON est_marqué(X)           # Utile si un noeud se trouve 2 fois dans la Pile
      marquer(Noeud)
      traiter(Noeud)
    Pour X dans adjacents(Noeud)  # A considérer dans l'ordre voulu (p.ex. inversé = de gche à dte, V. Trace)
      Si NON est_marqué(X)
        Alors empiler(Pile, X)   # empiler dans le désordre
    fin pour
  Fin Tant que
  Fin profondeur_iteratif
  
```

● Trace de parcours en profondeur de **p** à **d** (on souligne si traité) :

[p] → [h,v,k,q] → [h,v,k,c,r] → [h,v,k,c,r] → [h,v,k,c,a] → [h,v,k,c,f]
 → [h,v,k,c,e,u] → [h,v,k,c,e,g,v] ➔ v se trouve 2 fois dans la pile sans être marqué
 → [h,v,k,c,e, g] → [h,v,k,c,e] → [h,v,k, c] → [h,v, k] → [h, v]
 ➔ v : la 2e fois ! déjà traité. puis
 → [h] → [d] → []



Parcours général de graphes (suite)

Le principe de parcours en largeur

- traiter par niveau : traiter chaque noeud
- puis traiter chacun de ses successeurs avant de traiter les successeur du prochain niveau.

☞ **Par contre** : contrairement au parcours en profondeur,

s'il existe un cycle dans le graphe, des réponses seront néanmoins produites.

● Ce parcours s'adapte mieux à une solution itérative.

→ On utilise une file d'attente pour la construction de la liste des noeuds à traiter.

● Les opérations sur une file :

- enfiler(X, File)
- defiler(File)
- premier(File)
- ...

Parcours général de graphes (suite)

Pseudo algorithme récursif de parcours en largeur (graphe connexe) :

- Remarque : certains noeuds sont traités plusieurs fois, voir marquage

```
File :file d'attente = File_vider;  
  
Procédure largeur_récurif (G : ref Noeud) =  
Début  
  Si Non_est_vider(G) alors  
    traiter(noeud_courant(G)); //une opération quelconque  
    Pour X dans adjacents(noeud_courant(G))  
      File=Enfiler(X, File);  
    Fin pour;  
    X = Sommet(File); // Respect du TDA File  
    File = Défiler(File);  
    largeur_récurif (X);  
  Fin si;  
Fin largeur_récurif ;
```

☞ N.B. : pas de marquage

→ en l'absence de marquage, certains noeuds sont traités plusieurs fois.

Parcours général de graphes (suite)

Cas de graphe connexe : tout noeud est connecté aux autres.

➔ on peut travailler (se laisser guider) directement par la file d'attente

```

File : file d'attente = File_vider;
enfiler(la racine du graphe G, File)

Procédure largeur_récurif (G, File) =
Début
  Si Non_est_vider(File) alors
    X=Sommet(File);
    File=Défiler(File);           // car défiler ne renvoie pas un élément (cf. TDA File)
    traiter(noeud_courant(X));   //un traitement quelconque
    Pour X dans adjacents(noeud_courant(G))
      File=Enfiler(X, File);
    Fin pour;
    largeur_récurif (G, File);
  Fin si;
Fin largeur_récurif ;
  
```

- Initialement, il faut enfiler le noeud de départ.

☞ Le marquage (pour éviter de re-traiter un noeud) se fait comme pour le parcours en profondeur (voir ci-après).

Parcours général de graphes (suite)

- On peut récupérer le chemin par le mécanisme *Coming-From*.
- Pour un graphe connexe, la version suivante récupère le chemin par ce mécanisme.

```

File : file d'attente =File_vide;
enfiler(la racine du graphe G, File)
CF : tableau indicé par les noeuds initialisé à 0
CF[Départ]=Départ

Procédure chemin_largeur_récuratif (File) =
Début
  Si Non_est_vide(File) alors
    X=Sommet(File);
    File=Défiler(File);           // car défiler ne renvoie pas un élément (cf. TDA File)
    traiter(noeud_courant(X));    //une opération quelconque
  Pour X dans adjacents(noeud_courant(G))
    File=enfiler(X, File);
    CF[X]=noeud_courant(G)
  Fin pour;
  largeur_récuratif (File);
Fin si;
Fin largeur_récuratif ;

```

- Exercice : comment extraire ensuite le chemin ?
- Note : voir la version Python du problème de la monnaie.

Parcours général de graphes (suite)

L'algorithme itératif de parcours en largeur

```
File : file d'attente=File_vider;  
  
Procédure largeur_itératif ( G : graphe)  
  File=vider  
  Début  
    Si Non est_vider(G) alors  
      Enfiler(noeud_courant(G), File);  
    Fin si;  
    Tant que Non est_vider(File)  
      N = Sommet(File);  
      File = Défiler(File);  
      traiter(N); // ou visiter(N)  
      Pour X dans adjacents(N)  
        File = Enfiler(X, File);  
      Fin pour;  
    Fin Tant que;  
  Fin largeur_itératif ;
```

☞ N.B. : pas de marquage

Parcours général de graphes (suite)

Trace de parcours en largeur

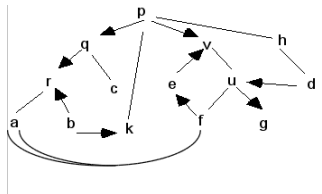
- Parcours en largeur de p à u

(noter le cycle $P \rightarrow \dots \rightarrow P \dots$) :

largeur(p) \rightarrow largeur(q) \rightarrow largeur(k) \rightarrow

largeur(v) \rightarrow largeur(h) \rightarrow largeur(r) \rightarrow

largeur(c) \rightarrow largeur(p) \rightarrow largeur(u)



- L'évolution de la File lors de ce parcours : ajout tant que *sommet* $\neq u$:

$[\] \rightarrow [p] \rightarrow [q, k, v, h] \rightarrow [k, v, h, r, c] \rightarrow [v, h, r, c, p] \rightarrow [h, r, c, p, u]$

$\rightarrow [r, c, p, u, p, d] \rightarrow [c, p, u, p, d, a] \rightarrow [p, u, p, d, a] \rightarrow [u, p, d, a, q, k, v, h]$

\rightarrow Destination atteinte.

Parcours général de graphes (suite)

L'algorithme itératif de parcours en largeur avec marquage

Ici, **la File peut contenir** des doublons qui ne seront pas traités une seconde fois.

```
File : file d'attente=File_vider;
Procédure largeur_itératif_marquage ( G : graphe)
Début
  Si Non est_vider(G)
  Alors
    Enfiler(noeud_courant(G), File);
  Fin si;
  Tant que Non est_vider(File)
  N = Sommet(File);
  File = Défiler(File);
  Si est_marqué(N)
  Alors passer à l'itération suivante ;
  Sinon marquer(N);
  Fin si;
  traiter(N); // ou visiter(N)
  Pour X dans adjacents(N);
    File=Enfiler(X, File);
  Fin pour;
  Fin Tant que;
Fin largeur_itératif_marquage ;
```


Parcours général de graphes (suite)

Une variante parcours en largeur itératif (avec marquage)

- Une seconde version avec marquage :
 - on marque les noeuds non marqués placés dans la file.
- **La file ne contiendra pas de doublon.**

```
File : file d'attente=File_vider;
Procédure premier_chemin_largeur_itératif ( G : graphe)
Début
  Si Non_est_vider(G) alors
    Enfiler(noeud_courant(G), File);
    marquer(noeud_courant(G));
  Fin si;
  Tant que Non_est_vider(File)
    N = Sommet(File);
    File = Défiler(File);
    traiter(N); // ou visiter(N)
    Pour X dans adjacents(N) non marqués
      File = Enfiler(X, File);
      marquer(X);
    Fin pour;
  Fin Tant que;
Fin premier_chemin_largeur_itératif ;
```

AES et graphes

- Reprendre de la première partie l'algorithme à essais successifs et l'appliquer pour la recherche en profondeur d'un chemin dans un graphe.
- Rappel :

Prédicat *Essais_successifs* (renvoie un booléen);

Données : G : un graphe d'états;

$Noeud_courant$: le noeud (ou état) courant que l'on traite dans cet appel

Résultat : Succès ou Echec (un booléen)

début

si $Noeud_courant$ est un état final **alors**

 renvoyer **Succès**

sinon

pour tous les $Noeud_suivant$ **successeurs de** $Noeud_courant$ **dans** G **faire**

si $Prometteur(G, Noeud_suivant)$ **alors**

si $Essais_successifs(G, Noeud_suivant) = Succès$ **alors**

 renvoyer **Succès**

fin

fin

fin

 renvoyer **Échec**

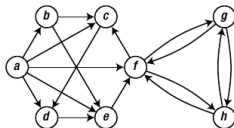
fin

fin

Représentation par ensembles d'adjacents

☞ Pléthore de représentations possibles.

- Cas de graphe non valué (non pondéré) représenté par une liste d'ensembles.



```

a, b, c, d, e, f, g, h = range(8)
N = [
    {b, c, d, e, f}, # a
    {c, e}, # b
    {d}, # c
    {e}, # d
    {f}, # e
    {c, g, h}, # f
    {f, h}, # g
    {f, g} # h
]
  
```

Représentation par listes d'adjacents

- Une autre présentation des graphes comme liste (ou array) d'adjacents.

On dit array car les listes de Python sont des arrays dynamiques.

```

a, b, c, d, e, f, g, h = range(8)
N = [
    [b, c, d, e, f], # a
    [c, e], # b
    [d], # c
    [e], # d
    [f], # e
    [c, g, h], # f
    [f, h], # g
    [f, g] # h
]
b in N[a] # Neighborhood membership
# True
len(N[f]) # Degree
#3

```

- Une variation dans la représentation du graphe est de représenter les ensembles de voisins comme une liste triée. En particulier, si on ne modifie pas trop le graphe.

→ Permet une méthode dichotomique dans la recherche d'un voisin.

N.B. : les ensembles (set) prédéfinis de Python sont plus pratiques dans ce cas.

Représentation par dict : graphe valué

- Pour tenir compte des valeurs, utiliser les `dicts` de Python où le voisin sera la clé et la valeur permet de réaliser un graphe valué (avec arcs pondérés). Par exemple :

```
a, b, c, d, e, f, g, h = range(8)
N = [
    {b:2, c:1, d:3, e:9, f:4}, # a
    {c:4, e:3},             # b
    {d:8},                  # c
    {e:7},                  # d
    {f:5},                  # e
    {c:2, g:2, h:2},       # f
    {f:1, h:6},            # g
    {f:9, g:8}             # h
]
```

- Le `dict` d'adjacence peut être utilisé comme ci-dessus, avec la valeur en plus.

```
b in N[a]    # Neighborhood membership
# True
len(N[f])    # Degree
#3
N[a][b]     # Edge weight for (a, b)
#2
```

Représentation par dict : graphe valué (suite)

Dans cette représentation (par Dicts) :

- L'utilisation des dicts pour les graphes est intéressante dans la mesure où on utilise un ensemble (set).

De plus, cette représentation est compatible avec les anciennes versions de Python qui n'avait pas le type `set` (mais avaient ensembles par construction, avec `'{'}`).

- Même si on n'a pas besoin de représenter un poids (la valeur) dans les noeuds, on peut continuer à utiliser les *dicts* et utiliser *None* ou toute autre valeur pour la pondération.

Représentation de graphes par dict d'ensemble

- A la représentation ci-dessus par `set`, `list` ou `dict` indexée par le numéro du noeud, on peut préférer une autre représentation plus souple est d'utiliser un `dict` où la clé peut être *hashée* (clé *hashable*) et où les valeurs seront des ensembles d'adjacents.
- Dans l'exemple ci-dessous, les noeuds sont des caractères :

```
N = {  
    'a': set('bcdef'),  
    'b': set('ce'),  
    'c': set('d'),  
    'd': set('e'),  
    'e': set('f'),  
    'f': set('cgh'),  
    'g': set('fh'),  
    'h': set('fg')  
}
```

- Notons que si on enlève les constructeurs `set (. .)`, les adjacents d'un noeud serait une chaîne de caractère non mutable.
→ Ça marche aussi ! Le choix dépend de ce que l'on veut faire du graphe et comment le graphe nous est fourni (sous forme de texte ?).

Représentation par matrice d'adjacence

```
a, b, c, d, e, f, g, h = range(8)
```

```
# a b c d e f g h
```

```
N = [[0,1,1,1,1,1,0,0], # a
```

```
      [0,0,1,0,1,0,0,0], # b
```

```
      [0,0,0,1,0,0,0,0], # c
```

```
      [0,0,0,0,1,0,0,0], # d
```

```
      [0,0,0,0,0,1,0,0], # e
```

```
      [0,0,1,0,0,0,1,1], # f
```

```
      [0,0,0,0,0,1,0,1], # g
```

```
      [0,0,0,0,0,1,1,0]] # h
```

```
N[a][b] # Neighborhood membership
```

```
# 1
```

```
sum(N[f]) # Degree
```

```
# 3
```

- Ici, un '1' atteste la présence d'un arc et '0' son absence.

→ On peut bien entendu utiliser True / False.

Représentation par matrice d'adjacence (suite)

- Pour représenter un graphe valué (pondéré), on peut utiliser le format suivant où `inf` représente l'absence d'arc.
 - Notons que les '0' représentant un cout (pondération) nul permettent de conserver le caractère réflexif de la fonction `arc` : il y a un arc de tout noeud à lui même de cout nul.
- Ce mécanisme simplifie souvent les algorithmes.

```
a, b, c, d, e, f, g, h = range(8)
inf = float('inf')

#   a   b   c   d   e   f   g   h
W=[[ 0,  2,  1,  3,  9,  4, inf, inf], # a
   [inf, 0,  4, inf, 3, inf, inf, inf], # b
   [inf, inf, 0,  8, inf, inf, inf, inf], # c
   [inf, inf, inf, 0,  7, inf, inf, inf], # d
   [inf, inf, inf, inf, 0,  5, inf, inf], # e
   [inf, inf,  2, inf, inf, 0,  2,  2], # f
   [inf, inf, inf, inf, inf, inf, 1,  0,  6], # g
   [inf, inf, inf, inf, inf, 9,  8,  0]] # h
```

Représentation par matrice d'adjacence (suite)

- Traces :

```
W[a][b] < inf    # Neighborhood membership
# True

W[c][e] < inf    # Neighborhood membership
# False

sum(1 for w in W[a] if w < inf) - 1 # Degree : le '-1' est pour les '0'
# 5
```

- La matrice ci-dessus simplifie l'accès aux pondérations, mais la vérification et la recherche du degré d'un noeud ou même l'itération sur les voisins sont réalisés différemment : on doit prendre en compte la valeur `inf`.

Parcours de graphes en Python

- **Exemple** : création d'un graphe orienté non pondéré à l'aide d'un **dictionnaire**.

A -> B

A -> C

B -> C

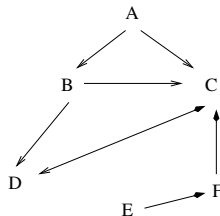
B -> D

C -> D

D -> C

E -> F

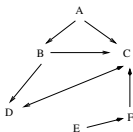
F -> C



```
graph = {'A': ['B', 'C'],  
        'B': ['C', 'D'],  
        'C': ['D'],  
        'D': ['C'],  
        'E': ['F'],  
        'F': ['C']}
```

Parcours de graphes en Python (suite)

- Recherche de chemin :

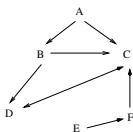


```
def chemin(graph, depart, arrivee, path=[]):  
    path = path + [depart]  
    if depart == arrivee :  
        return path  
    if not graph.has_key(depart):  
        return None  
    for noeud in graph[depart]:  
        if noeud not in path:  
            nouv_path = chemin(graph, noeud, arrivee, path)  
            if nouv_path: return nouv_path  
    return None
```

```
chemin(graph, 'A', 'D')  
# ['A', 'B', 'C', 'D']
```

Parcours de graphes en Python (suite)

- Recherche de tous les chemins depuis un noeud :



```

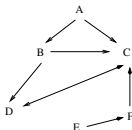
def tous_chemins(graph, depart, arrivee, path=[]):
    path = path + [depart]
    if depart == arrivee :
        return [path]
    if not graph.has_key(depart):
        return []
    paths = []
    for noeud in graph[depart]:
        if noeud not in path:
            nouv_paths = tous_chemins(graph, noeud, arrivee, path)
            for nouv_path in nouv_paths:
                paths.append(nouv_path)
    return paths
  
```

```

tous_chemins(graph, 'A', 'D')
# [['A', 'B', 'C', 'D'], ['A', 'B', 'D'], ['A', 'C', 'D']]
  
```

Parcours de graphes en Python (suite)

- Chemin le plus court (en nombre d'arcs) par une stratégie B&B :



```
def chemin_le_plus_court(graph, depart, arrivee, path=[]):
    path = path + [depart]
    if depart == arrivee :
        return path
    if not graph.has_key(depart):
        return None
    shortest = None
    for noeud in graph[depart]:
        if noeud not in path:
            nouv_path = chemin_le_plus_court(graph, noeud, arrivee, path)
            if nouv_path:
                if not shortest or len(nouv_path) < len(shortest):
                    shortest = nouv_path
    return shortest

chemin_le_plus_court(graph, 'A', 'D')
# ['A', 'C', 'D']
```

Complément : bibliothèques de Graphe de Python

- Quelques bibliothèques de graphes pour Python :
 - NetworkX : <http://networkx.lanl.gov>
 - python-graph : <http://code.google.com/p/python-graph>
 - Graphine : <https://gitorious.org/graphine/pages/Home>
 - Graph-tool : <http://graph-tool.skewed.de>

Et aussi :

- Pygr (<https://github.com/cjlee112/pygr>)
- Gato, un toolbox d'animation de graphes (<http://gato.sourceforge.net>) ;
- PADS : une collection d'algorithmes de graphes
(<http://www.ics.uci.edu/~eppstein/PADS>).

Python et Graph_tool

- Les modules *NetworkX*, *graph_tool* et *igraph* sous Python proposent des graphes.
- *Graph_tool* est réalisé en interne en *C++* à l'aide de *Boost*. Il est difficile de l'installer sous Python. Ici, on voit juste un exemple.
- Voir aussi <https://wiki.python.org/moin/PythonGraphApi> pour les graphes sous Python.

```
from graph_tool.all import *  
  
g = Graph()           # Par défaut, un graphe orienté  
ug = Graph(directed=False) # Pour un graphe non orienté
```

- On peut changer un graphe orienté en non orienté et inversement à la volée à l'aide de la fonction *set_directed()*.
- On peut tester cette propriété du graphe par la fonction *is_directed()*.

```
ug = Graph()  
ug.set_directed(False)  
assert(ug.is_directed() == False)
```


Python et Graph_tool (suite)

- On ajoute des noeuds avec `add_vertex()`.

```
v1 = g.add_vertex()    # renvoie la description du noeud dans v1
v2 = g.add_vertex()
```

- On peut ajouter des arcs par `add_edge()`. On relie les noeuds `v1` et `v2` :

```
e = g.add_edge(v1, v2)
```

- On peut demander à dessiner un graphe :

```
graph_draw(g, vertex_text=g.vertex_index, vertex_font_size=18, output_size=(200, 200), output="two-nodes.png")
```

- On peut créer un graphe à partir d'un autre :

```
g1 = Graph()
# ... remplir g1
g2 = Graph(g1)           # g1 et g2 sont des copies
```

- La copie est ici réelle (dite "copie profonde").
- Ce ne sont pas deux références sur le même graphe.

Complément : Complexité d'opérations sous Python

Complexité en temps :



- This page documents the time-complexity (aka "Big O" or "Big Oh") of various operations in current CPython.
- Other Python implementations (or older or still-under development versions of CPython) may have slightly different performance characteristics. However, it is generally safe to assume that they are not slower by more than a factor of $O(\log n)$. Generally, 'n' is the number of elements currently in the container. 'k' is either the value of a parameter or the number of elements in the parameter.

Complément : Complexité d'opérations sous Python (suite)

Cas des listes :

- The Average Case assumes parameters generated uniformly at random.
- Internally, a list is represented as an array ; the largest costs come from growing beyond the current allocation size (because everything must move), or from inserting or deleting somewhere near the beginning (because everything after that must move). If you need to add/remove at both ends, consider using a `collections.deque` instead.

Complément : Complexité d'opérations sous Python (suite)

Operation	Average Case	 Amortized Worst Case
Copy	$O(n)$	$O(n)$
Append[1]	$O(1)$	$O(1)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(k+n)$
Extend[1]	$O(k)$	$O(k)$
 Sort	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
x in s	$O(n)$	
min(s), max(s)	$O(n)$	
Get Length	$O(1)$	$O(1)$

Complément : Complexité d'opérations sous Python (suite)

collections.deque

- A deque (double-ended queue) is represented internally as a doubly linked list. (Well, a list of arrays rather than objects, for greater efficiency.) Both ends are accessible, but even looking at the middle is slow, and adding to or removing from the middle is slower still.

Complément : Complexité d'opérations sous Python (suite)

Operation	Average Case	Amortized Worst Case
Copy	$O(n)$	$O(n)$
append	$O(1)$	$O(1)$
appendleft	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
popleft	$O(1)$	$O(1)$
extend	$O(k)$	$O(k)$
extendleft	$O(k)$	$O(k)$
rotate	$O(k)$	$O(k)$
remove	$O(n)$	$O(n)$

Complément : Complexité d'opérations sous Python (suite)

Ensembles :

- See dict – the implementation is intentionally very similar.

Operation	Average case	Worst Case
<code>x in s</code>	$O(1)$	$O(n)$
Union <code>s t</code>	$O(\text{len}(s)+\text{len}(t))$	
Intersection <code>s&t</code>	$O(\min(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$
Multiple intersection <code>s1&s2&...&sn</code>		$(n-1)*O(l)$ where l is $\max(\text{len}(s_1), \dots, \text{len}(s_n))$
Difference <code>s-t</code>	$O(\text{len}(s))$	
<code>s.difference_update(t)</code>	$O(\text{len}(t))$	
Symmetric Difference <code>s^t</code>	$O(\text{len}(s))$	$O(\text{len}(s) * \text{len}(t))$
<code>s.symmetric_difference_update(t)</code>	$O(\text{len}(t))$	$O(\text{len}(t) * \text{len}(s))$

→ Dans ce tableau, il y a une note sur la ligne "intersection s & t" :
 'replace "min" with "max" if t is not a set'.

Complément : Complexité d'opérations sous Python (suite)

Dicts :

- The Average Case times listed for dict objects assume that the hash function for the objects is sufficiently robust to make collisions uncommon. The Average Case assumes the keys used in parameters are selected uniformly at random from the set of all keys.
- Note that there is a fast-path for dicts that (in practice) only deal with str keys ; this doesn't affect the algorithmic complexity, but it can significantly affect the constant factors : how quickly a typical program finishes.

Operation	Average Case	Amortized Worst Case
Copy[2]	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(n)$
Set Item[1]	$O(1)$	$O(n)$
Delete Item	$O(1)$	$O(n)$
Iteration[2]	$O(n)$	$O(n)$

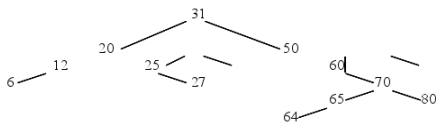
Complément : Complexité d'opérations sous Python (suite)

Quelques notes :

- sur les fonctions 'append' et 'extend' des listes, et 'Set item' des dicts : These operations rely on the "Amortized" part of "Amortized Worst Case". Individual actions may take surprisingly long, depending on the history of the container.
- sur les fonctions 'copy' et 'iteration' des Dicts : For these operations, the worst case n is the maximum size the container ever achieved, rather than just the current size. For example, if N objects are added to a dictionary, then $N-1$ are deleted, the dictionary will still be sized for N objects (at least) until another insertion is made.

Addendum : Équilibrage d'un ABOH

- Exemple : on a un ABOH que l'on veut équilibrer :



- La hauteur (le niveau) H d'un ABOH équilibré (complet) contenant N éléments : $H = \text{Log}(N)$
- Ceci n'est pas applicable à l'exemple ci-dessus car cet ABOH n'est pas complet $H = 6$.

Addendum : Équilibrage d'un ABOH (suite)

Équilibrage :

L'équilibrage se fait à l'aide d'un tableau. On vide l'ABOH dans T puis on réorganise pour créer un autre ABOH équilibré.

Par un parcours infixé, on vide l'arbre dans le tableau (N=12) :

6	12	20	25	27	31	50	60	64	65	70	80
1	2	3	4	5	6	7	8	9	10	11	12

- Algorithme : on remonte un nouvel ABOH à partir du Tableau

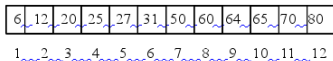
Addendum : Équilibrage d'un ABOH (suite)

```

Début
T: Tableau; A, New_A: arbre;
Vider A dans T
equilibre(T,New_A)
Fin;
procédure equilibre(T: Tableau, A: ES arbre)=
Début
  Soient Inf et Sup les bornes de T
  Si (Taille(T)=0) alors retourne — Inf > Sup
  Sinsi (Taille(T)=1) alors — Inf=Sup
    A=allouer un noeud
    A.info ← T[inf];
    A.gche ←A.drte ← NULL;
  Sinon — Inf < Sup
    m ← (Inf + Sup) /2;
    A=allouer un noeud;
    A.info ← T[m];
    equilibre(T[Inf .. m-1], A.gche);
    equilibre(T[m+1 .. Sup], A.drte);
  Fin Si;

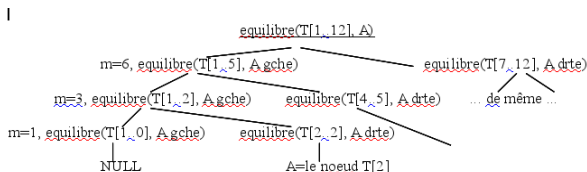
```

- Pour l'exemple ci-dessus :

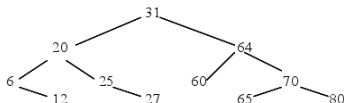


Addendum : Équilibrage d'un ABOH (suite)

On procèdera comme suite (la trace de la branche gauche)) :



- Ce qui donne (autre forme de trace donnant l'ABOH) :



Addendum : Équilibrage d'un ABOH (suite)

→ La hauteur de ce nouvel arbre est $\log(12) \leq 4$.

- Un autre solution dépendante du type des informations dans les noeuds consiste à remplacer le choix de la moyenne des bornes par la moyenne des valeurs (possible pour les entiers, réels, caractères .. et les types composites si leur clés le permet) .

Remarque :

- Si l'ABOH est présenté sous forme d'un tableau, on applique à l'ABOH une numérotation où si la valeur associée à une racine est I , alors, son fils gauche aura l'indice $2I$ et le fils droit $2I+1$.
- Ce qui permet dans les langages sans pointeurs. Cette numérotation n'est pas utilisée dans l'algorithme de rééquilibrage sauf si l'arbre est représenté par un tableau en mémoire.
- Exercice : code pour l'évaluation d'une expression propositionnelle représentée par un arbre :

Addendum : Équilibrage d'un ABOH (suite)

```
— solution a la question Arbre du ctrl-cont de mai 95. il n'y
— a pas de programme, juste un test des declarations
with text_io; use text_io;
procedure arbre is
Type Genre is (op, cst);
Type Operateur is (ou, et, non);
Type Valeur is (vrai, faux);

Type Boite(g : Genre := cst) is record — initialise a valeur au pif
  Case g is
    When op => O : Operateur;
    When cst => V : Valeur;
  End Case;
End record;

Type Noeud;
Type Arbre is access Noeud;
```

Addendum : Équilibrage d'un ABOH (suite)

Type Noeud is record

Info : Boite;

Gauche, Droite : Arbre;

End record;

Function Evaluate(A : Arbre) return boolean is

Begin

Case A.info.g is

When op =>

Case A.info.O is

When ou => return evaluate(A.Gauche) Or evaluate(A.Droite);

When et => return evaluate(A.Gauche) And evaluate(A.Droite);

When non => return evaluate(A.Gauche) ;

when others => null;

End Case;

When cst =>

Case A.info.V is

Addendum : Équilibrage d'un ABOH (suite)

```
When vrai => return true;
When faux => return false;
when others => null;
End Case;
When others => null;
End Case;
End Evaluate;

begin
null;
end arbre;

voir aussi /home/alex/ADA-ALL/ADA-AUT/ADA/Sujets.dir/SOLUTIONS/arbre.ada
Aussi /home/alex/ECL-ALL/AG-ALL/AC-05-06/2005/ARBRES-code-doc

ZZZ voir /home/alex/GSU-ALL/FIT3-plus-complet/ALGOSD-2/Arbres/Arbres.doc

Pour une implantation avec des tableaux. ==> Python
```

Addendum : TDA arbre binaire

Sorte Arbre_bin, Place

Utilise Bool, Element

Opérations :

Vide : \rightarrow Arbre_bin

est_vide : Arbre_bin \rightarrow bool

racine : Arbre_bin \rightarrow place

gauche : Arbre_bin \rightarrow Arbre_bin

droit : Arbre_bin \rightarrow Arbre_bin

recherche : Arbre_bin \times Elément \rightarrow place

inserer : Arbre_bin \times Elément \rightarrow Arbre_bin

.....

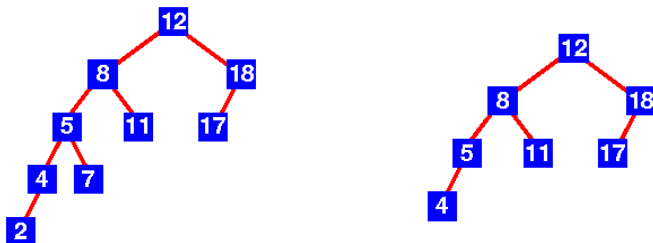
- Préconditions et axiomes
- les opérateurs habituels de Place

Addendum : AVL

- Les arbres AVL sont des ABOHs (arbres binaires ordonnés Horizontalement, ou arbres binaires de recherche) automatiquement équilibrés.
- Dans un arbre AVL, les hauteurs des deux sous-arbres d'un même noeud diffèrent au plus de un.
 - De ce fait, la recherche, l'insertion et la suppression d'un élément sont toutes en $O(\log(N))$.
 - L'insertion et la suppression peuvent nécessiter un équilibrage.
- Le terme **AVL** est dû aux noms de ses deux inventeurs *G. Adelson-Velsky* et *E. Landis* (1962).
- Le facteur d'équilibrage d'un noeud est la différence entre la hauteur de son sous-arbre droit et celle de son sous-arbre gauche.
- Un noeud dont le facteur d'équilibrage est 1, 0, ou -1 est considéré comme équilibré.

Addendum : AVL (suite)

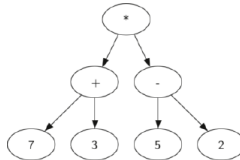
- Un noeud avec tout autre facteur est considéré comme déséquilibré et devra être rééquilibré.
- Exemple d'un ABOH (non équilibré qui ne garantie pas $O(\log(N))$) et son AVL :



Pour AVL : `/home/alex/ECL-ALL/AC-ALL/AC-05-06/2005/ARBRES-code-doc/Avl.cpp`
 et `/home/alex/ECL-ALL/AC-ALL/AC-05-06/2005/ARBRES-code-doc/TDA-Aboh.cpp`
`/home/alex/ECL-ALL/AC-ALL/AC-05-06/2005/Bien-GSU-from-CD/ALGOSD-2/Arbres`

Addendum : évaluation d'expressions arithmétique

- Soit l'arbre binaire de l'expression $((7 + 3) * (5 - 2))$:



- Écrire les algos pour évaluer cette expressions.
- Ensuite, introduire le traitement des variables.

Addendum : évaluation d'expressions arithmétique (suite)

Solution :

- Le code ci-dessus lit une expression, construit son arbre syntaxique.
- On peut ensuite à l'évaluation de cette expression.
- Construction de l'arbre syntaxique :

```

import queue
def build_parse_tree(fp_exp):
    fp_list = fp_exp.split()
    p_stack = queue.Queue()
    e_tree = BinaryTree('')          # A définir
    p_stack.put(e_tree)
    current_tree = e_tree
    for i in fp_list:
        if i == '(':
            current_tree.insert_left('')
            p_stack.put(current_tree)
            current_tree = current_tree.get_left_child()
        elif i not in ['+', '-', '*', '/', ')']:
            current_tree.set_root_val(int(i))
            parent = p_stack.get()
            current_tree = parent
        elif i in ['+', '-', '*', '/']:
            current_tree.set_root_val(i)
            current_tree.insert_right('')
            p_stack.put(current_tree)
            current_tree = current_tree.get_right_child()

```

Addendum : évaluation d'expressions arithmétique (suite)

```

elif i == ')':
    current_tree = p_stack.get()
else:
    raise ValueError
return e_tree

```

● L'évaluation

```

import operator
def evaluate(parse_tree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.truediv}
    left = parse_tree.get_left_child()
    right = parse_tree.get_right_child()
    if left and right:
        fn = opers[parse_tree.get_root_val()]
        return fn(evaluate(left), evaluate(right))
    else:
        return parse_tree.get_root_val()

pt = build_parse_tree("( ( 10 + 5 ) * 3 )")
pt.postorder()           # défini plus loin

```

Addendum : évaluation d'expressions arithmétique (suite)

- Il y a aussi cette version qui n'a pas besoin de Pile.

```
import operator
def postorder_eval(tree):
    ops = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postorder_eval(tree.get_left_child())
        res2 = postorder_eval(tree.get_right_child())
        if res1 and res2:
            return ops[tree.get_root_val()](res1, res2)
        else:
            return tree.get_root_val()
```

- Affichage de l'arbre de l'expression après sa construction par la fonction ci-dessous :

```
def print_exp(Arbre):
    str_val = ""
    if Arbre:
        str_val = '(' + print_exp(Arbre.get_fils_gauche())
        str_val = str_val + str(Arbre.get_val_racine())
        str_val = str_val + print_exp(Arbre.get_fils_droit()) + ')'
    return str_val
```


Addendum : évaluation d'expressions arithmétique (suite)

☞ Voir le site <https://docs.python.org/3/library/collections.html> pour les containers. Voir en particulier les exemples donnés pour "OrderedDict dict subclass that remembers the order entries were added". Aussi voir "defaultdict"

<code>namedtuple()</code>	factory function for creating tuple subclasses with named fields
<code>deque</code>	list-like container with fast appends and pops on either end
<code>ChainMap</code>	dict-like class for creating a single view of multiple mappings
<code>Counter</code>	dict subclass for counting hashable objects
<code>OrderedDict</code>	dict subclass that remembers the order entries were added
<code>defaultdict</code>	dict subclass that calls a factory function to supply missing values
<code>UserDict</code>	wrapper around dictionary objects for easier dict subclassing
<code>UserList</code>	wrapper around list objects for easier list subclassing
<code>UserString</code>	wrapper around string objects for easier string subclassing

- Ex Ordered Dict :

Addendum : évaluation d'expressions arithmétique (suite)

```
# regular unsorted dictionary
d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}

# dictionary sorted by key
OrderedDict(sorted(d.items(), key=lambda t: t[0]))
# OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

# dictionary sorted by value
OrderedDict(sorted(d.items(), key=lambda t: t[1]))
# OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

# dictionary sorted by length of the key string
OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
# OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

Addendum : TDA (Lecture optionnelle)

- Petite introduction aux types de données abstraits (TDA).
- Pourquoi les TDAs ?
- Le but : moyen formel de spécification et de définition de structures de données.
- La définition d'un TDA comporte différentes informations :
 - **sorte** : les type que l'on veut définir (par exemple T)
 - **utilise** : les type utilisés pour définir T
 - **opérations** : le profil des opérateurs définis sur T (totales et partielles).

- Une fonction est notée par :

$$\text{nom} : \text{type_de_par1} \times \dots \times \text{type_de_parn} \rightarrow \text{type_du_résultat}$$

- Par exemple, l'opérateur **and** sur les booléens :

$$\text{and} : \text{booléen} \times \text{booléen} \rightarrow \text{booléen}$$
$$\text{not} : \text{booléen} \rightarrow \text{booléen}$$

Addendum : TDA (Lecture optionnelle) (suite)

- La définition du i ème élément d'une liste quelconque :

accès : liste \times indice \rightarrow élément

\rightarrow veu dire que la fonction est partielle :

\rightarrow indice entre les bornes (inf) et supérieure (sup) du tableau/de la liste.

- Les constantes sont considérés comme des fonction sans argument (fonctions *0-aires*)

vrai : \rightarrow booléen

faux : \rightarrow booléen

- Le type de la valeur renvoyée par un opérateur peut être :

(1) le type en cours de définition ou

(2) un type mentionné dans la section **utilise**.

1 : l'opérateur est une loi de composition interne (dit opérateur **interne**),

2 : il s'agit d'une loi de composition externe (dit **observateur**).

Addendum : TDA (Lecture optionnelle) (suite)

- **préconditions** : conditions à respecter sur les fonctions partielles.

accès(liste, indice) est définie si indice \in {inf..sup}

- **axiomes** : pour définir les propriétés des opérateurs.

→ Des "vérités" sur les opérateurs : équations/logique des prédicats,

- Exemples :

$$\text{and}(\text{vrai}, X) = X \quad X : \text{booléen}$$

$$\text{and}(\text{faux}, X) = \text{faux}$$

$$\text{not}(\text{not}(X)) = X$$

- Le symbole "=" ici est utilisé dans le sens *équationnel*.
- On peut également se servir des axiomes comme des règles de réécriture.

Par exemple : $\forall X : \text{bool}(X) \implies \text{not}(X) \vee \text{and}(\text{vrai}, X) = \text{vrai}$

- Règles pour les axiomes : croiser tout ! (parfois très long)
- Dans la pratique, on combine uniquement les observateurs avec les internes
- appelée *complétude suffisante*.

Le TDA Bool

TDA bool

Signature :

False : \rightarrow bool

True : \rightarrow bool

And : bool x bool \rightarrow bool

Or : bool x bool \rightarrow bool

Xor : bool x bool \rightarrow bool

Not : bool \rightarrow bool

Implies : bool x bool \rightarrow bool

Equiv : bool x bool \rightarrow bool

...

Le symbole \Rightarrow

Le symbole \Leftrightarrow (et non \equiv qui n'est pas un connecteur)

Le TDA Bool (suite)

Axiomes :

pour $a, b : \text{bool}$

$$\text{Not}(\text{True}) = \text{False}$$

$$\text{Not}(\text{False}) = \text{True}$$

$$\text{And}(\text{True}, b) = b$$

$$\text{And}(\text{False}, b) = \text{False}$$

☞ ajouter les cas symétriques (cf. $\text{And}(b, \text{False})$)

$$\text{Or}(\text{True}, b) = \text{True}$$

$$\text{Or}(\text{False}, b) = b$$

☞ ajouter les cas symétriques (cf. $\text{Or}(b, \text{False})$)

$$\text{Xor}(a, b) = \text{Or}(\text{And}(a, \text{Not}(b)), \text{And}(\text{Not}(a), b))$$

Le TDA Bool (suite)

$\text{Implies}(\text{True}, \text{True}) = \text{True}$

$\text{Implies}(\text{True}, \text{False}) = \text{False}$

$\text{Implies}(\text{False}, b) = \text{True}$

☞ On peut écrire également $\text{Implies}(a, b) = \text{Or}(\text{Not}(a), b)$

$\text{Equiv}(a, b) = \text{And}(\text{Implies}(a, b), \text{Implies}(b, a)) = \text{Not}(\text{Xor}(a, b))$

...

- On peut bien entendu utiliser des expressions logiques équivalentes.
→ Par exemple, on peut tout réécrire à l'aide de NAND et NOR.

Le TDA Bool (suite)

A propos de la partie Axiomes : la *complétude suffisante* (ou minimale)

- Habituellement, on constitue les *Axiomes* en croisant les opérations **externes** (celles qui ne renvoient pas *set*) avec les **internes** (celles qui renvoient *set*).
 - A l'inverse, la complétude maximale est de croiser tous les opérateurs, deux à deux.
- On peut imposer (et ajouter dans la définition) des préconditions aux opérations.
 - Ces préconditions seront nécessaires et utiles pour préciser la sémantique des fonctions *partielles* (voir plus loin).
 - Par exemple, exiger que l'indice d'un tableau soit dans un intervalle défini (les bornes du tableau).

TDA entier

Cette section ainsi que les autres TDAs suivants sont facultatives.

Exemple du TDA *entier* :

sorte entier

utilise : booléen

opérateurs :

zéro : \rightarrow entier # 0

succ : entier \rightarrow entier # incrément

pred : entier \rightarrow entier # décréement

add : entier \times entier \rightarrow entier

soust : entier \times entier \rightarrow entier

mult : entier \times entier \rightarrow entier

div : entier \times entier \rightarrow entier # fonction partielle

égal : entier \times entier \rightarrow booléen

....

TDA entier (suite)

préconditions : pour les fonctions partielles,

div(x,y) est défini si $y \neq \text{zéro}$. x,y : entier

ou encore $\text{div}(x,y)$ est défini si $\text{not égal}(y,\text{zéro})$.

axiomes

pour X,Y : entier

$$\text{succ}(\text{pred}(X)) = X$$

$$\text{pred}(\text{succ}(X)) = X$$

$$\text{add}(0,X) = X$$

$$\text{add}(\text{succ}(X),Y) = \text{succ}(\text{add}(X,Y))$$

$$\text{égal}(0,0) = \text{vrai}$$

$$\text{égal}(\text{succ}(X),\text{succ}(Y)) = \text{égal}(X,Y)$$

...

NB : complétude suffisante : il suffit seulement combiner égal avec les autres.

TDA entier (suite)

Remarques sur la gestion des préconditions :

- Qui teste les préconditions des fonction partielles :
l'appelant de la fonction ou la fonction elle-même ?
- Les deux approches ont leurs avantages et inconvénients.
- Il est souvent difficile pour la fonction elle-même de juger de l'action à entreprendre lorsqu'une précondition n'est pas respectée.
- De plus, en cas d'erreur, comment doit-elle régler le problème de l'envoi d'un objet du type calculé par la fonction ?
- L'utilisateur préfère souvent ne pas avoir à tester des valeurs avant d'appeler une fonction.
- Au niveau de l'implantation, ce sont les moyens du langage hôte qui permettent de décider.
- Le traitement des exceptions est un bon candidat pour la seconde approche.
- En cas d'erreur : simple affichage d'un message suivi d'un arrêt du programme jusque un traitement poussé par les exceptions.
- On étudiera plus loin le traitement des préconditions en Python à l'aide des *exceptions*.

Le TDA Liste simple

Sort Liste, Place, Iter_Data

Utilise Element, Bool, Entier , Flot

Opérations :

Vide : \rightarrow Liste

est_vide : Liste \rightarrow Bool

tete : Liste \rightarrow Element

reste : Liste \rightarrow Liste

cons : Element \times Liste \rightarrow Liste

premier : Liste \rightarrow Place

dernier : Liste \rightarrow Place

est_dernier : Liste \times Place \rightarrow Bool

inser_fin : Element \times Liste \rightarrow Liste

existe : Element \times Liste \rightarrow Bool

recherche Element \times liste \rightarrow Place

lg : Liste \rightarrow Entier

nieme Entier \times Liste \rightarrow Place

Le TDA Liste simple (suite)

$\text{nb_occ Element} \times \text{Liste} \rightarrow \text{Entier}$

$\text{concat Liste} \times \text{Liste} \rightarrow \text{Liste}$

$\text{copy Liste} \rightarrow \text{Liste}$

$\text{affiche} : \text{Liste} \rightarrow \text{Flot}$

Pour *place* :

$\text{contenu} : \text{Liste} \times \text{Place} \rightarrow \text{Element}$ – contenu d'une place

$\text{modifier} : \text{Liste} \times \text{Place} \times \text{Element} \rightarrow \text{Place}$ – Liste modifiée

$\text{next} : \text{Liste} \times \text{Place} \rightarrow \text{Place}$

$\text{before} : \text{Liste} \times \text{Place} \rightarrow \text{Place}$

$\text{valide} : \text{Liste} \times \text{Place} \rightarrow \text{Bool}$ – Place valide (éventuellement)

L'itérateur : (permet les expressions "pour tout X dans L faire Action")

$\text{init_iterateur} : \text{Liste} \rightarrow \text{iter_Data}$

$\text{itérateur} : \text{iter_Data} \rightarrow \text{Place}$

- L'opérateur "itérateur" modifie son paramètre.

On doit donc écrire une procédure à deux paramètres ou bien transformer "itérateur" en :

$\text{itérateur} : \text{iter_Data} \rightarrow \text{iter_Data}$ – avancer Place_crt

Le TDA Liste simple (suite)

Place_crt : iter_Data \rightarrow Place – extraire place_crt

Les préconditions

tete(l) : non est_vide(l)

reste(l) : non est_vide(l)

premier(l) : non vide (l)

dernier(l) : non est_vide(l)

next(l,p) : non est_dernier(l,p)

nieme(i,l) : (i > 0) & (lg(l) >= i)

nieme_place(i,l) : (i > 0) & (lg(l) >= i)

.....

Les axiomes : croiser les observateurs et les internes. Conserver les combinaisons significatives.

Le TDA Liste simple (suite)

Remarques : les cinq premiers opérateurs sont considérés comme le minimum permettant de représenter n'importe quel autre.

- La définition de l'itérateur simplifie grandement les écritures.
- De même, on préfère un résultat de type Place à celui de type Element.
 - Les traductions et l'implantation plus efficaces et plus concises.
- Par exemple, étant donné l'itérateur, l'opérateur "existe(L,E)" s'écrit :

pour tout X dans L

si égal(X,E) alors renvoie vrai fin si

Ou encore :

pour tout X dans L jusqu'à trouve

si égal(X,E) alors trouve <- vrai fin si

☞ La gestion des préconditions des TDA se fait par les exceptions.

Spécification TDA Ensemble (Set)

TDA set (ensemble de *data*)

Utilise : *nat, bool, data*

Opérations :

vide : \rightarrow set

egal : $\text{set} \times \text{set} \rightarrow \text{bool}$

ajout : $\text{data} \times \text{set} \rightarrow \text{set}$

constructeur

supprime : $\text{data} \times \text{set} \rightarrow \text{set}$

membre : $\text{data} \times \text{set} \rightarrow \text{bool}$

est_vide : $\text{set} \rightarrow \text{bool}$

card : $\text{set} \rightarrow \text{nat}$

le nbr d'éléments d'un ensemble

union, intersection, difference, sous_ensemble ... et autres fonctions

Equations (axiomes) : $d, d1 \in \text{data}, s, s1 \in \text{set}$

egal(vide, vide) = True

egal(ajout(d,s), ajout(d,s1)) = egal(s,s1)

ajout(d,s) représente un ens. non vide.

supprime(d,vide) = vide

Spécification TDA Ensemble (Set) (suite)

$\text{supprime}(d, \text{ajout}(d, s)) = s$

$\text{membre}(d, \text{vide}) = \text{false}$

$\text{membre}(d, \text{ajout}(d, s)) = \text{true}$

$\text{est_vide}(\text{vide}) = \text{true}$

$\text{est_vide}(\text{ajout}(d, s)) = \text{false}$

$\text{card}(\text{vide}) = 0$

$\text{card}(\text{ajout}(d, s)) = 1 + \text{card}(s)$

→ On accepte les doublons mais on ne les compte pas !

↳ Le traitement des doublons dans ajout / supprimer

.....

☞ Remarques : il s'agit ici d'un ensemble : une collection ordonnée sans doublon.

→ On remarque par exemple que la fonction *card* en tient compte (sinon, on rechercherait un autre exemplaire éventuel de *d* dans *s*)

Pour rendre compte du fait que cette collection est sans doublon, on peut préciser le constructeur *ajout* : supposons disposer d'une fonction opérationnelle **add(d,s)** qui insère effectivement (implantation) *d* dans *s* et renvoie l'ensemble résultant.

Spécification TDA Ensemble (Set) (suite)

ajout(d, vide) = **add**(d, vide)

ajout(d, ajout(d1, s)) = **add**(d, ajout(d1, s)) si (\neg *membre*(d, ajout(d1, s)))

- Rappel : les axiomes permettent (à un programmeur) de créer le type **set** de n'importe quel type et de proposer les fonctions nécessaire à son utilisation (\sim approche objet).

→ Par exemple, dans le TDA **set**, la fonction **egal** peut être :

Prédicat `egal(S1, S2) =`

Si (`est_vide(S1)`) : `est_vide(S2)`

lire : *Si (...) alors renvoie est_vide(S2)*

Sinon

`assert(\neg est_vide(S1))`

ce que l'on sait ICI sur l'état du calcul

Si (`est_vide(S2)`) : `false`

Sinon

`assert S1 = (d,R1) et S2 = (d,R2)`

ICI : S1 et S2 ne sont pas vides

`egal(R1,R2)`

On renvoie le résultat de egal(R1,R2)

- Dans certaines notation des axiomes, on suppose une lecture **logique** binaire :

→ Si une situation n'est pas représentée, on suppose que le résultat est faux (pour un opérateur booléen) et non-défini pour une action (situation à éviter si possible).

Spécification TDA Ensemble (Set) (suite)

→ Ce qui correspond à l'hypothèse du *monde fermé*.

→ Exemple : les axiomes de **egal** pourrait être décrits par :

$\text{egal}(\text{vide}, \text{vide}) = \text{True}$

$\text{egal}(\text{ajout}(d,s), \text{ajout}(d,s1)) = \text{egal}(s,s1)$

$\text{egal}(\text{vide}, \text{ajout}(d,s)) = \text{False}$

$\text{egal}(\text{ajout}(d,s), \text{vide}) = \text{False}$

$\text{ajout}(d,s)$ représente un ens. non vide.

Dans la version donnée dans le TDA (les 2 premiers axiomes ci-dessus), on a supposé que les autres cas de figure donneront *False*

☞ Donner la fonction *egal* (en pseudo code) et discuter de sa table de vérité.

TDA Pile

- Une pile st souvent implantée à l'aide d'un table ou une liste.
- Le mode de gestion est : le dernier arrivé est le premier servi.
- Les opérateurs habituellement définis sur les piles sont donnés par la signature (le profile) de ces opérateurs qui en même temps nous renseignent sur la manière (syntaxe) de leur utilisation :

pile_vider : \rightarrow Pile

la constante pile vide

empiler : Pile \times élément \rightarrow Pile

ajout d'un élément au sommet

dépiler : Pile \rightarrow Pile

(ou \rightarrow) suppression du sommet (pop()),

sommet : Pile \rightarrow élément

consultation de l'élément du sommet (top())

est_vider : Pile \rightarrow booléen

test de la vacuité de la pile (empty())

taille : Pile \rightarrow entier

le nbr d'éléments (size())

- La fonction *sommet* est partielle (notation \rightarrow) et provoque une erreur (ou une exception) si la pile est vide.

TDA Pile (suite)

- On notera que dépiler ne renvoie pas un élément mais retire simplement l'élément (éventuel) qui se trouve au sommet de la Pile. Ceci vient du caractère fonctionnel de ces opérateurs (une fonction renvoie une seule chose !).
- Pour exprimer formellement ces propriétés, on complète la signature par une partie préconditions qui expliquent dans quelles conditions peut-on utiliser ces fonctions.
- Par exemple, pour indiquer que la fonction sommet doit être appelée sur une pile non vide :

Pré conditions : *sommet : non est_vide(Pile)*

- Ensuite, on complète la définition par les axiomes qui précisent les propriétés des opérateurs. Pour ce faire, on croise l'ensemble pile_vide, empiler, dépiler (dit les opérateurs internes) avec est_vide , sommet, taille (opérateurs externes) pour obtenir une complétude suffisante.

TDA Pile (suite)

- Axiomes : $P : \text{pile}, E : \text{élément}$ déclarations pour la suite
 - est_vide(pile_vide)=Vrai
 - est_vide(empiler(P,E))=Faux
 - est_vide(dépiler(P)) = (taille(P)==1) # peut être exprimé d'autres manières
 - sommet(empiler(P,E))=E
 - sommet(pile_vide) n'est pas évoqué car la précondition de sommet n'est pas respectée.
 - sommet(dépiler(P)) n'apporte pas d'information intéressante.
 - taille(pile_vide)=0
 - taille(empiler(P,E))=taille(P)+1
 - taille(dépiler(P)) = si taille(P)<2 alors 0 sinon taille(P)-1
 - sommet(empiler(P,E))=E
- Des expressions telles que "*Si dépiler(P)) != pile_vide alors sommet(dépiler(P))*" donne le 1er élément du reste de P" n'apporte aucune aide particulière.

TDA Pile (suite)

- A l'aide de ce TDA, on est en mesure de savoir par exemple que la fonction dépiler peut être appelée même sur une pile vide.
- Parfois et au besoin, on va plus loin que la complétude suffisante et on croise tous les opérateurs internes avec eux mêmes.
- Par exemple, on peut noter :
 - dépiler(empiler(pile_vide, E))=pile_vide. # évident !
 - dépiler(pile_vide)=pile_vide. # dépiler une pile vide qui reste vide !

TDA Graphe

sorte : graphe, place

utilise : liste, booléen, Data

opérateurs : (minimum)

vide : \rightarrow graphe

recherche : graphe \times Data \rightarrow Place

insere : graphe \times Place \times Liste \rightarrow graphe # Liste d'Adj

adj : graphe \times Place \rightarrow liste

est_vide : graphe \rightarrow Booléen

contenu : graphe \times Place \rightarrow Data

modifie : graphe \times Place \times Data \rightarrow Place

- Pour un graphe orienté, on remplace l'opérateur Adj par :

succ : graphe \times place \rightarrow liste

pred : graphe \times place \rightarrow liste

- Abstraction à l'aide des listes/Dictionnaires/

PrD from Fred Vivien

→ A faire.

Caractéristiques de la PrD

Définition des sous-problèmes et condition de **sous-problème optimal** :

- On peut caractériser une solution optimale à un sous-problème en termes de solutions à ses sous-problèmes (donc sous-sous-problèmes).

- Par exemple, si on doit multiplier une chaîne de matrices $A_0 \dots A_{n-1}$

(1) Les sous-problèmes seront les différents parenthésages **optimaux** de la séquence

$$A_i \times A_{i+1} \times \dots \times A_j$$

→ On découpe donc en sous-problèmes :

pour $A_i \times A_{i+1} \times \dots \times A_j$, on doit trouver

$$(A_i \times \dots \times A_k)(A_{k+1} \times \dots \times A_j) \quad \text{avec } k \in \{i, i+1, \dots, j-1\}$$

Caractéristiques de la PrD (suite)

- **Caractérisation des solutions optimales et condition de sous-structure optimale :**

(2) Quel que soit le choix de k , $(A_i \times \dots \times A_k)$ et $(A_{k+1} \times \dots \times A_j)$ doivent être résolus de façon optimale pour avoir un optimum global.

☞ Si cela ne devait pas être le cas, une solution globale optimale aurait une solution non optimale à un de ses sous-problèmes ! Mais cela est impossible car on devrait pouvoir remplacer la solution non optimale par une optimale !

→ Cette observation détermine un problème d'optimisation :

la PrD est un outil d'optimisation.

- Supposons placer k là où on aura un nombre minimal de multiplications et obtenir N_{ij} une solution.

→ N_{ij} sera exprimé en termes de solutions optimales aux sous-problèmes.

Caractéristiques de la PrD (suite)

Conception de la PrD :

- Sachant que chaque A_i est une matrice $d_i \times d_{i+1}$, de ci-dessus, on déduit :

$$N_{ij} = \min_{i \leq k < j} \{N_{ik} + N_{k+1j} + d_i d_{k+1} d_{j+1}\} \quad \text{pour } i : 0..n - 1$$

avec $N_{ii} = 0$ (une seule matrice)

- N_{ij} = minimum du nombre de multiplications nécessaires pour chaque sous-expression
+ le nombre de multiplications pour faire la dernière multiplication de matrices.

- **Exemple :**

$$(A_i \times A_i \times \dots \times A_j)$$

$$(A_i \times \dots \times A_k) (A_{k+1} \times \dots \times A_j) \text{ avec } k \in \{i, i + 1, \dots, j - 1\}$$

| -trouver min-| | -trouver min-|

+ | - la multiplication de ces deux matrices -|

Caractéristiques de la PrD (suite)

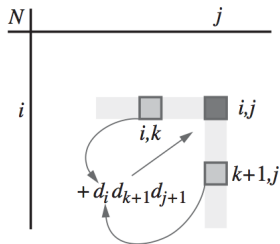
Rappel : $N_{ij} = \min_{i \leq k < j} \{N_{ik} + N_{k+1j} + d_i d_{k+1} d_{j+1}\}$ pour $i : 0..n - 1$ avec $N_{ii} = 0$ (une seule matrice)

- L'expression N_{ij} ressemble à celle d'une stratégie Diviser-Régner mais seulement en apparence.

→ Ici, les sous-problèmes **ne sont pas indépendants** et **partagent des sous-sous-problèmes** qui nous empêchent de couper le problème initial en sous-problèmes indépendants.

- On peut calculer les N_{ij} de manière *Ascendante (Bottom-up)* et les stocker dans une table (cf. PrD) :

1. On initialise $N_{ij} = 0$ pour $i = 0..n - 1$ puis
 2. L'équation générale de N_{ij} calcule $N_{i+1, i+1}$ qui ne dépend que de N_{ii} et de $N_{i+1, i+1}$ qui seront à ce moment-là disponibles.
 3. Avec $N_{i, i+1}$, on aura $N_{i, i+2}$...
- N_{ij} est donc calculé de manière ascendante et $N_{0, n-1}$ sera la réponse finale.



Caractéristiques de la PrD (suite)

- Ci-dessous l'algorithme dont la complexité est évaluée à $O(n^3)$.

Fonction **ChaineMatrices**($d_0 \dots d_n$) :

Entrée : une séquence $d_0 \dots d_n$ d'entiers

Sortie : pour $i, j = 0 \dots n - 1$, le nombre minimum de multiplications $N_{i,j}$ nécessaires pour calculer le produit $A_i.A_{i+1} \dots A_j$ où A_k est une matrice $d_k \times d_{k+1}$

Début

Pour $i \leftarrow 0$ à $n - 1$:

$N_{i,i} \leftarrow 0$

Pour $b \leftarrow 1$ à $n - 1$:

Pour $i \leftarrow 0$ à $n - b - 1$:

$j \leftarrow i + b$

$N_{i,j} \leftarrow \infty$

Pour $k \leftarrow i$ à $j - 1$:

$N_{i,j} \leftarrow \min(N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1})$

Fin ChaineMatrices

Caractéristiques de la PrD (suite)

Résumons les étapes de définition d'un schéma PrD :

- (I) **Sous-problèmes simples** : le problème initial doit pouvoir être décomposé en sous-problèmes avec une *structure similaire* au problème initial (même formule d'optimalité).
De plus il doit y avoir une façon *simple* de définir les sous-problèmes (avec peu d'indices et de variables).

- (II) **Optimalité des sous-problèmes** : une solution optimale au problème initial doit être une composition des solutions optimales aux sous-problèmes avec une *composition simple*.
→ On ne devrait pas pouvoir trouver une solution optimale globale contenant une solution *non-optimale* aux sous-problèmes.

- (III) **Recouvrement des sous-problèmes** : les solutions optimales aux sous-problèmes indépendants peuvent contenir des *sous-problèmes communs* (des *sous-sous-problèmes* communs).
Ces recouvrements permettent d'améliorer l'efficacité de l'algorithme PrD en mémorisant les solutions à ces sous-sous-problèmes (pour ne pas refaire des calculs).
Un bon exemple est Fib(N) utilisant Fib(N-1) et Fib(N-2) déjà calculées et stockées.
On pourra utiliser des tableaux / matrices / etc.

Stratégie Programmation Dynamique

- La Programmation Dynamique (Prd) est similaire à Diviser-Régner et découpe le problème P_k en sous problèmes $P_j, j < k$.
- La résolution des sous problèmes P_j a en général besoin d'un faible nombre d'information (par rapport à P_k).
- **Mais contrairement** à l'approche descendante du Diviser pour Régner, la technique de la *PrD* procède ensuite par une approche Ascendante (*Bottom-up*).
- Comparée à une approche *Ascendant* (par exemple la multiplication binaire, voir plus loin) et au calcul de la médiane (*Descendant*, vu plus haut) :
 - On résout les instances simples et petites, on stock les résultats.
 - Plus tard, lorsque l'on a besoin de ces valeurs déjà calculées, on les utilise (pas de re-calculation) pour calculer les instances plus importantes.
- N.B. : le terme *PrD* vient de la théorie du contrôle où « programmation » veut dire que l'on utilise un tableau dans lequel les solutions sont construites.

Stratégie Programmation Dynamique (suite)

Quelques Exemples de PrD :

- Coefficient Binomial
- Algorithme de Floyd (les chemins les plus courts entre toutes paires de noeuds d'un graphe valué)
- Recherche dans un AVL (arbre binaire compacte et équilibré)
- Voyageur de commerce (*TSP : traveler sales Person*)
- Fib (en partant de 1 et jusqu'à N) est un exemple simple
- Factorielle est un cas trivial de PrD
- Distance d'édition (*Levenshtein*)
- Rendue optimale de Monnaies,
- etc.

Stratégie Programmation Dynamique (suite)

Un exemple :

Le calcul de *Fibonacci* (version récursive puis utilisation d'un tableau qui stock les termes précédents).

→ Ce qui caractérise PrD est l'approche Ascendante

• Les étapes de PrD :

1. Définir une propriété récursive qui donne la solution à une instance du problème
2. Résoudre une instance du problème de façon ascendante en résolvant d'abord les instances plus petites.

Rappel : $Fib(0) = Fib(1) = 1$

et $Fib(N) = Fib(N-1) + Fib(N-2)$ si $N > 1$

Stratégie Programmation Dynamique (suite)

- Le code Python de la stratégie PrD sans le stockage des valeurs.

Le paramètre *Crt* est incrémenté depuis 2 jusqu'à arriver à *N* auquel cas *Fib(N)* est simplement la somme des deux derniers paramètres :

```
def fib_aux(N, Crt, F_moins_1, F_moins_2) :
    if Crt == N : return F_moins_1 + F_moins_2
    else : return fib_aux(N, Crt+1, F_moins_1 + F_moins_2, F_moins_1)

def fib(N) :
    if N < 2 : return 1
    else : return fib_aux(N, 2, 1, 1)

print(fib(10)) # 89
```

- Une version avec stockage.

Le paramètre *tab* contient les termes *Fib(1)* .. *Fib(N)* :

```
def fib_tab_aux(N, tab) :
    if N == 0 : return tab[-1]+tab[-2]
    else : return fib_tab_aux(N-1, tab+[tab[-1]+tab[-2]])

def fib_tab(N) :
    if N < 2 : return 1
    else : return fib_tab_aux(N-2, [1,1])

print(fib_tab(10)) # 89
```

Stratégie Programmation Dynamique (suite)

Exemple (binomial) :

$$\text{Coefficient binomial : } C_k^n = \binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{pour } 0 \leq k \leq n$$

Pour résoudre ce problème :

1. On définit une propriété récursive qui donne la solution à une instance du problème
2. On résout le problème de façon ascendante en résolvant d'abord les instances plus petites.

• L'approche *PrD* de résolution de ce problème établit la propriété récursive :

$$C_k^n = \binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{pour } 0 < k < n \\ 1 & \text{pour } k = 0 \text{ ou } k = n \end{cases}$$

• Ce qui donnera l'algorithme basique suivante (voir amélioration dans bin2) :

```

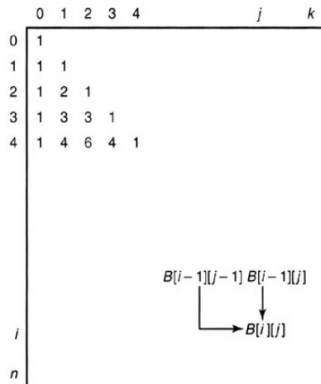
fonction bin(n, k : entiers)
  Si      k=0 OU n=k
  Alors  1
  Sinon  bin(n-1, k-1) + bin(n-1, k)           // Découpage (en n-1) puis assemblage (addition)
  Finsi
  
```

Stratégie Programmation Dynamique (suite)

Amélioration : on utilisera une matrice B pour stocker les calculs intermédiaires :

Le contenu de cette matrice pour le calcul de $bin(4, 2)$:

$bin(4, 2)$	$B[0][0] = 1$
Calcul de la	$B[1][0] = 1$
ligne 1	$B[1][1] = 1$
Calcul de la	$B[2][0] = 1$
ligne 2	$B[2][1] = B[1][0] + B[1][1] = 1+1 = 2$ $B[2][2] = 1$
Calcul de la	$B[3][0] = 1$
ligne 3	$B[3][1] = B[2][0] + B[2][1] = 1+2 = 3$ $B[3][2] = B[2][1] + B[2][2] = 2+1 = 3$
Calcul de la	$B[4][0] = 1$
ligne 4	$B[4][1] = B[3][0] + B[3][1] = 1+3 = 4$ $B[4][2] = B[3][1] + B[3][2] = 3+3 = 6$



Stratégie Programmation Dynamique (suite)

L'algorithme amélioré du calcul des valeurs de la matrice B devient :

```
def bin2(n,k) :
    B=[ [ 0 for i in range(k+1) ] for j in range(n+1) ]      # n+1 lignes de k+1 colonnes : n x k
    for i in range(n+1) :
        for j in range(min(i,k)+1) :
            if (j == 0 or j == i) : B[i][j] = 1
            else : B[i][j] = B[i-1][j-1] + B[i-1][j]
    print(B); print()
    return B[n][k]

print(bin2(4,2))      # 6
```

- La complexité de *bin2* : pour une valeur de *i*, on note le nombre de passages dans la boucle *j*

val de <i>i</i>	0	1	2	3	4	5	<i>k</i> -1	<i>k</i>	<i>k</i> +1	...	<i>n</i>
nbr. de passes dans la boucle <i>j</i>	1	2	3	4	5	6	<i>k</i>	<i>k</i> +1	<i>k</i> +1	...	<i>k</i> +1

Le total : $1 + 2 + \dots + k$ jusqu'à $i = k - 1$ + $(n - k + 1)$ fois $(k + 1) = (2n - k + 2)(k + 1)/2 = \Theta(nk)$.

→ Améliore grandement la version basique qui utilise la factorielle classique.

N.B. : avec la version *PrD* de la factorielle dans *bin*, on obtiendra aussi une complexité $\Theta(nk)$.

Exercice : faire de même en exploitant l'égalité
$$\binom{n}{k} = \binom{n}{n-k}$$

Stratégie Programmation Dynamique (suite)

Exemple 3 : Distance de *Levenshtein* :

- La distance de *Levenshtein* est une métrique permettant de calculer une **distance d'édition** entre deux mots en vue d'une (proposition de) correction orthographique.
- Cette distance représente le nombre minimum d'opérations d'édition (suppression, remplacement et insertion de caractères) nécessaires pour rendre identiques les deux chaînes de caractères.
- **Exemples** de la distance (d) :
 - pour 2 mots identiques ("Ecole" et "Ecole"), $d=0$
 - pour "Ecole" et "Ekole", $d=1$ (remplacement de 'k' par 'c').
 - Notons qu'une suppression de 'k' suivie d'une insertion de 'c' donnera $d=2$ mais qui n'est pas minimale.
 - pour la correction de "klavié" vers "clavier", on a $d=3$:
 - un remplacement de 'k' par 'c', un remplacement de 'é' par 'e' et l'ajout de 'r'.

Stratégie Programmation Dynamique (suite)

- Les éditeurs de texte proposent de remplacer un mot erroné par un autre, ils proposent en général des mots pris dans un dictionnaire et dans l'ordre de cette distance.
- La **complexité** de ce calcul est le produit des longueurs des 2 mots comparés.
- **Variantes** de la distance d'édition :
 - Dans une variante (*Daereau-Levenshtein*), on peut également procéder à intervertir (transposer) deux caractères adjacents.
 - La distance de **Hamming** est une variante qui ne permet que la substitution.
- La distance de *Levenshtein* a donné lieu à une variante appliquée en séquençage d'ADN (*Smith-Waterman*).
- L'application du principe de la *PrD* au calcul de la distance de *Levenshtein* de 2 mots est de construire une matrice contenant la distance entre tous les préfixes des deux mots.
 - Ainsi, la dernière valeur calculée donnera la distance entre les deux mots.

Stratégie Programmation Dynamique (suite)

Le pseudo-Algorithmme de Levenshtien

```

Procédure levenshtein(mot1, mot2: tableau de caractères) // chaque mots commence à l'indice 1
locale: M : matrice[0..taille mot1][0..taille mot2] // la matrice M commence à l'indice 0
Pour i=0.. taille mot1 M[i][0]=i; Fin pour // dist. des préfixes du mot1 aux mots vides
Pour j=0.. taille mot2 M[0][j]=j; Fin pour // dist. des préfixes du mot2 aux mots vides
Pour j=1.. taille mot2 // le 1er car. de chaque mot est à l'indice 1
  Pour i=1.. taille mot1
    Si (mot1[i] = mot2[j])
      Alors M[i][j]= M[i-1,j-1] // aucune opération nécessaire
    Sinon m[i][j]= min(M[i-1][j]+1, // suppression
      M[i][j-1]+1, // insertion
      M[i-1][j-1]+1) // substitution
    Fin si
  Fin pour
Fin pour
m[taille mot1][taille mot2] est le résultat
Fin Levenshtien

```

- L'intérêt de la stratégie PrD est de calculer les distances successives à l'aide de précédentes (en phase ascendante).
- La phase descendante est ici réduite aux initialisations.

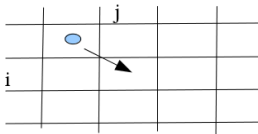
Stratégie Programmation Dynamique (suite)

Exemple : les mots "klavié" vs "clavier"

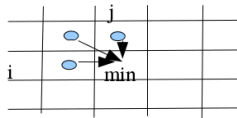
	k	l	a	v	i	é
0	1	2	3	4	5	6
c	1	1	2	3	4	5
l	2	2	1	2	3	4
a	3	3	2	1	2	3
v	4	4	3	2	1	2
i	5	5	4	3	2	1
e	6	6	5	4	3	2
r	7	7	6	5	4	3

3 ← il faudra 3 opérations pour rendre "klavié" identique à "clavier"

- Pour comprendre le remplissage de la matrice M, considérons les deux cas de figures de comparaison de 2 lettres (identiques ou différentes) :



$$\text{mot2}[j] = \text{mot1}[i] : \text{mat}[i][j] = \text{mat}[i-1][j-1] + 1$$



$$\text{mot2}[j] \neq \text{mot1}[i] : \text{mat}[i][j] = \min(\text{des 3 cases}) + 1$$

Stratégie Programmation Dynamique (suite)

Exemple 4 : algorithme de Floyd

Propos : calcul des chemins entre toutes paires de noeuds d'un graphe.

Utilisation : trafic aérien (en l'absence de ligne directe), routage (adresses) dans les réseaux et routage de messages dans un réseau global, ...

- Dans ce graphe (valué orienté), on cherche le meilleur chemin entre v_1 et v_3 .

- On a les candidats avec leurs valeurs :

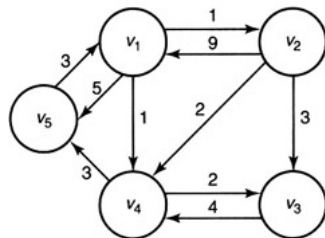
$$\text{longueur}(\langle v_1, v_2, v_3 \rangle) = 1 + 3 = 4$$

$$\text{longueur}(\langle v_1, v_4, v_3 \rangle) = 1 + 2 = 3$$

$$\text{longueur}(\langle v_1, v_2, v_4, v_3 \rangle) = 1 + 2 + 2 = 5$$

Le chemin $\langle v_1, v_4, v_3 \rangle$ semble **le plus court**.

→ Un problème d'optimisation ../..



Stratégie Programmation Dynamique (suite)

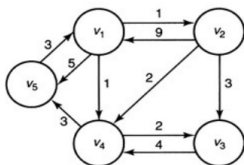
- La recherche du chemin le plus court est un problème d'optimisation :
 - Il existe plusieurs chemins (candidats) entre deux noeuds (de valeurs différentes) et on souhaite calculer le plus court (de valeur optimale)
 - Il peut exister plusieurs chemins de longueur minimale entre 2 noeuds.
→ Le choix est dans ce cas aléatoire.
- Un algorithme basique peut calculer TOUS les chemins entre TOUS couples de noeuds puis de choisir le plus court.
- Un tel algorithme (pour un graphe fortement connexe) est de **complexité exponentielle** $(N-2)!$ pour N noeuds.
- A l'aide de la PrD, on peut obtenir une complexité cubique.

Stratégie Programmation Dynamique (suite)

Démarche : on définit une matrice W des poids (distances directes) avec la convention :

- $W[i][j]$ = la distance (arc) entre i et j
- $W[i][j] = 0$ si $i = j$
- $W[i][j] = \infty$ si i et j non directement connectés

• Pour le graphe du problème, on aura la matrice initiale des distances W et D = la matrice des chemins les plus courts.



	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

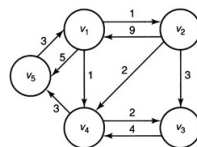
• On notera $D^{(k)}[i][j]$ = la longueur du chemin le plus court entre i et j utilisant seulement les noeuds $\{v_1, v_2, \dots, v_k\}$.

Stratégie Programmation Dynamique (suite)

- Par définition, on a (pour N noeuds) :

- $D^{(0)}[i][j]$ = chemin passant par aucun autre noeud que j et j = la matrice W
- $D^{(n)}[i][j]$ = chemin passant par n'importe quel autre noeud du graphe
= la matrice finale D (meilleurs chemins)

Exemples :



$$\circ D^{(0)}[2][5] = \text{longueur}(\langle v2, v5 \rangle) = \infty$$

le chemin le plus court entre $v2 - v5$ en passant par $v0$ = direct ($v0$ fictif)

$$\circ D^{(1)}[2][5] = \text{minimum}(\text{longueur}(\langle v2, v1, v5 \rangle), \text{longueur}(\langle v2, v5 \rangle)) \\ = \text{minimum}(14, \infty) = 14$$

$$\circ D^{(2)}[2][5] = D^{(1)}[2][5] = 14$$

aucun chemin partant de $v2$ ne peut repasser par $v2$ (vrai pour tout graphe)

$$\circ D^{(3)}[2][5] = D^{(2)}[2][5] = 14 \text{ l'inclusion de } v3 \text{ (seul) n'apporte rien de plus.}$$

→ ($\langle v2, v3, v5 \rangle = \langle v2, v5 \rangle$, on ne passe pas par $v4$).

Stratégie Programmation Dynamique (suite)

- Donc, on obtiendra la matrice \mathbf{D} depuis \mathbf{W} en procédant (principe de la *PrD*) :
 1. Définir une propriété récursive (calcul) permettant d'obtenir $D^{(k)}$ depuis $D^{(k-1)}$.
 2. Résoudre une instance du problème (ascendant) en répétant l'étape (1) pour $k=1 \dots n$.
 → Ce qui créé la séquence : $D^{(0)}, D^{(1)}, D^{(2)}, \dots, D^{(n)}$.
- Deux cas peuvent se présenter pour l'étape 1 :

Cas 1 : au moins un des chemins les plus courts de i à j n'utilisera pas v_k dans $\{v_1, \dots, v_k\}$:

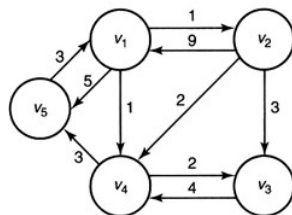
$$D^{(k)}[i][j] = D^{(k-1)}[i][j]. \quad (1)$$

On a les candidats avec leurs valeurs :

Par exemple, dans le graphe précédent :

$$D^{(5)}[1][3] = D^{(4)}[1][3] = 3$$

Car l'inclusion de v_5 à $\{v_1, v_4, v_3\}$ n'apporte rien au chemin le plus court qui reste $\langle v_1, v_4, v_3 \rangle$.

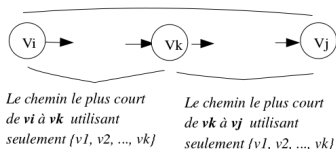


Stratégie Programmation Dynamique (suite)

Cas 2 : les chemins les plus courts entre v_i et v_j utilisant seulement $\{v_1, v_2, \dots, v_k\}$ utilisent v_k .

Dans ce cas, tout chemin le plus court est décrit dans le schéma ci-contre.

Sachant que v_k ne peut pas être un noeud intermédiaire du sous chemin v_i à v_k , ce sous chemin utilise seulement les noeuds $\{v_1, v_2, \dots, v_{k-1}\}$.



- Ce qui implique que le sous-chemin sera de la même longueur que $D^{(k-1)}[i][k]$.

$$\text{D'où : } D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]. \quad (2)$$

- Exemple : $D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$.
- Sachant que nous serons dans l'un des 2 cas ci-dessus, la valeur de $D^{(k)}[i][j]$ est le minimum de la valeur à droite des équations (1) et (2).

- Ce qui veut dire que l'on détermine $D^{(k)}[i][j]$ depuis $D^{(k-1)}[i][j]$ par :

$$D^{(k)}[i][j] = \text{minimum}(D^{(k-1)}[i][k], D^{(k-1)}[i][k] + D^{(k-1)}[k][j]) .$$

→ Ceci accomplit le cas 1 ci dessus.

Stratégie Programmation Dynamique (suite)

- Pour l'étape 2, on développera la propriété récursive de l'étape 1 pour créer la séquence $D^{(0)}, D^{(1)}, D^{(2)}, \dots, D^{(n)}$.

- Exemple de $D^{(2)}[5][4] = \min(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4])$
(rappel : $D^0 = W$ la matrice initiale) :

$$D^{(1)}[2][4] = \min(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4])$$

$$= \min(2, 9 + 1) = 2$$

$$D^{(1)}[5][2] = \min(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2])$$

$$= \min(\infty, 3 + 1) = 4$$

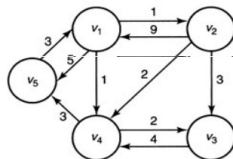
$$D^{(1)}[5][4] = \min(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4])$$

$$= \min(\infty, 3 + 1) = 4$$

$$D^{(2)}[5][4] = \min(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4])$$

$$= \min(4, 3 + 2) = 4$$

- Après avoir calculé tous les D^2 , on continuera jusqu'à D^5 .
- Ces calculs donneront la matrice D ci-contre.



	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

Stratégie Programmation Dynamique (suite)

Algorithme de Floyd

```

Procédure floyd (n, W[[]], D[[]])      # les matrices W[[]] et D[[]]
  D = W
  pour k = 1 .. n
    pour i = 1 .. n
      pour j = 1 .. n
        D[i][j] = min(D[i][j], D[i][k] + D[k][j])  # même principe utilisé dans Dijkstra
  
```

- N.B. : on peut remarquer la trame de l'algorithme de *Dijkstra*.
- La complexité de cet algorithme : $\Theta(n^3)$
- La matrice finale (D) des chemins les plus courts :

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

Stratégie Programmation Dynamique (suite)

- Si on veut obtenir les chemins (trajets) :

On enregistre dans la nouvelle matrice $CF[u][v]$ le plus grand numéro du noeud intermédiaire sur le chemin le plus court $u \rightarrow v$. $CF[u][v] = 0$ si ce noeud n'existe pas.

```

Procédure floyd2(n, W[[]], D[[]], CF[[]])
  pour i = 1 .. n
  pour j = 1 .. n
    CF[i][j] = 0;
  D = W
  pour k = 1 .. n
    pour i = 1 .. n
      pour j = 1 .. n
        Si (D[i][k] + D[k][j] < D[i][j])
          CF[i][j] = k
          D[i][j] = D[i][k] + D[k][j]

```

- Affichage du chemin :

```

Procédure affiche_path(u, v) # trajet pour aller de u à v
  Si (CF[u][v] != 0)
    affiche_path(u, CF[u][v])
    print(" → ", CF[u][v])
    affiche_path(CF[u][v], v)

```

Stratégie Programmation Dynamique (suite)

La PrD et la question d'optimisation

- L'algorithme de **Floyd** permet de déterminer le chemin le plus court et de le calculer.
- La construction de la solution optimale devient donc la 3e étape du développement d'un problème d'optimisation.
- Les étapes de PrD appliquée à un problème d'optimisation devient :
 1. Définir une propriété récursive qui donne la solution à une instance du problème
 2. Calculer la valeur de la solution optimale de chaque instance
 3. Construire la solution optimale de façon ascendante en utilisant d'abord les instances plus petites.
- Comme toute autre méthode, la PrD peut ne pas être adaptée à un problème d'optimisation car :

Le principe d'optimalité : *une solution optimale à un problème contient des solutions optimales à toutes les sous instances de ce problème (et vice versa).*

Stratégie Programmation Dynamique (suite)

- Ce principe a été appliqué au problème des chemins les plus courts :
le chemin le plus court $v_i \rightarrow v_j$ en passant par le noeud intermédiaire v_k est optimale si les chemins partiels $v_i \rightarrow v_k$ et $v_k \rightarrow v_j$ sont optimaux (les plus courts ici).
- **Il faudra donc d'abord démontrer** l'application du principe d'optimalité avant de faire l'hypothèse d'optimalité de la solution avec PrD.

- **Contre exemple** : calculer les chemins **les plus longs** sur le graphe ci-contre.

N.B. : on se restreint à ne pas exploiter le circuit $(V2, V3)$ dans ce graphe.

Sinon, on peut passer une infinité de fois dans ce cycle.

On a le chemin le plus long optimal entre $v1 \rightarrow v4$: $\langle v1, v3, v2, v4 \rangle = 7$

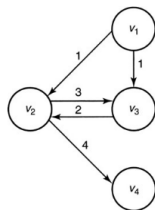
Pependant, le sous-chemin $v1 \rightarrow v3$ n'est pas optimal car :

longueur($\langle v1, v3 \rangle$) = 1 et *longueur*($\langle v1, v2, v3 \rangle$) = 4 (sans circuit)

- Or, la solution optimale finale $\langle v1, v3, v2, v4 \rangle$ ne contient pas $\langle v1, v2, v3 \rangle$.

→ Le principe d'optimalité ne s'applique pas ici.

- Un exemple d'application du principe est la recherche optimale dans les ABOHs (AVLs).
- La solution optimale passera parfois par une optimisation des données.



Algorithme de Roy-Warshall

PrD et TSP

PrD et Sac à Dos

Quelques applications des graphes

Exemples d'applications

- Efficacité des pipelines :
 - transport et distribution d'eau/pétrole/gaz/etc.
 - On peut s'intéresser au FLUX ou à MST pour simplement assurer la distribution
 - Les valeurs des arcs (du graphe) : coût ou capacité.
- Table de routage :
 - calcul des plus courts chemins entre les routeurs eux-mêmes pour savoir comment atteindre une destination par la meilleure voie.
- Messagerie :
 - visiter des points de livraison pour prendre/déposer des colis.
 - le voyageurs de commerce
 - trajet d'un facteur
- Réseaux de communication :
 - Les réseaux avec leurs équipements (lignes tél, relais, Satellites, ...) à situer de façon optimale (MST)
- Gestion du trafic : Meilleure route entre 2 points de la ville en fonction du trafic
 - problème de FLUX
 - Rechercher des chemins d'encombrement minimum

Quelques applications des graphes (suite)

- Navigation aérienne (les avions dans des couloirs au ciel !)
 - problème : le vent, coût du survol d'un espace, le trafic...
 - Le MST (chemin de poids minimum entre 2 points)
 - Le système de transport fermé (circuit fermé)
 - délivrer des pièces, amener des marchandises (ou en emporter d'un magasin)
 - TSP (voir aussi CLP)
- Câblage de circuits imprimés Connexion des broches des puces (les broches standard, cartes différentes
 - MST permet de les connecter Aussi, le problème de placement des puces et les pistes les plus courtes (cf. chemins sous contraintes) Problème de largeur de pistes, taille des cartes, ...
- etc...

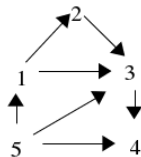
Compléments sur les graphes

Quelques définitions

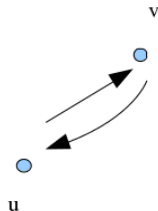
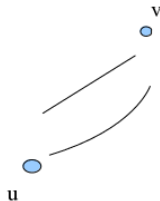
- Un graphe $G=(V, E)$

V : ensemble de noeuds (vertex)

$E \in (V \times V)$: ensemble d'arêtes ou arcs (edges)



Un Digraphe



Compléments sur les graphes (suite)

- Chaque **arc**=une paire $(v, w) \in E, v, w \in V$
- Si (v, w) est **ordonné**, on aura un graphe **orienté** (*digraphe*)
→ *directed graph*
- w est **adjacent** de v si $(v, w) \in E$
- Dans un graphe non orienté (undirected), $(v,w) \sim (w,v)$ v et w sont adjacents
- Dans certains problèmes, les noeuds représentent les *variables* et les arcs les *relations*.
- Dans d'autres, les noeuds peuvent être des opérateurs/constantes/identificateurs et les arêtes un lien vers les opérandes.
- **Poids** (*weight*, pondération) : valeur d'un arc/arête
- **Chemin** (branche, path) : w_1, w_2, \dots, w_n tel que $(w_i, w_{i+1}) \in E$
 $chemin((X_1, \dots, X_k), (V, E)) \equiv (X_1, X_2) \in E \wedge \dots \wedge (X_{k-1}, X_k) \in E$
- **Longueur** d'un chemin contenant N noeuds=nombre d'arcs= $n-1$
- Un noeud Y est **accessible** depuis un noeud X s'il existe un chemin de X vers Y :
 $(X, Y) \in E \vee \exists Z_1, \dots, Z_k : chemin((X, Z_1, \dots, Z_k, Y), (V, E))$

Compléments sur les graphes (suite)

- (w, w) est de longueur 0 si $(w, w) \notin E$ (il n'y a pas d'arc de w à w).
- Si le graphe contient un arc (v, v) , le chemin (v, v) = une **boucle** (loop)
 - une boucle ne concerne qu'un noeud
- Un **chemin simple** = un chemin de w_1 à w_n avec des noeuds distincts mais éventuellement (à la fin), w_1 peut être égale à w_n .

Par exemple : $a \rightarrow b \rightarrow c \rightarrow d$ ou $a \rightarrow b \rightarrow c \rightarrow a$

- **Réseau** : Un graphe connexe et sans boucle
- Un **cycle** : un chemin de longueur ≥ 1 tel que $w_1 = w_n$.

→ Un cycle peut être simple si le chemin est simple :

Dans un graphe non orienté, pour un cycle, les arêtes sont distinctes et le chemin uvu n'est pas un cycle car (u, v) et (v, u) sont les mêmes arêtes.

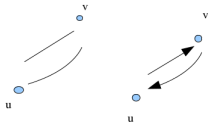
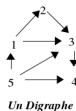
Mais dans un graphe orienté, (u, v) et (v, u) sont deux arcs distincts.

Dans ce cas : uvu est un cycle.

Compléments sur les graphes (suite)

Dans les 2 dessins ci-contre :

- o celui de gauche N'EST pas un cycle (dans un graphe NON orienté),
- o celui de droite est un cycle (dans un graphe orienté).



- Un graphe orienté est **acyclique** s'il n'a pas de cycle ;
- Un **DAG** : directed acyclique graphe.
- Un graphe non orienté est **connexe** s'il y a un chemin entre toute paire de noeuds :

$$\forall X, Y \in V, \text{accessible}(X, Y, (V, E))$$

→ Un graphe orienté et connexe est appelé **fortement connexe** (voir exemple de réseau aéroports/ferroviaire) ci-dessous.

→ Un graphe peut ne pas être connexe (des variables sans relation peuvent exister)

- Si un graphe orienté n'est pas fortement connexe mais son graphe non orienté sous-jacent (sans les directions) est connexe, on aura un graphe faiblement connexe.
- **Relation d'ordre** dans les graphes.

Compléments sur les graphes (suite)

- Un graphe est **complet** s'il existe un arc (arête) entre toute paire de noeuds.
Exemple : dans un réseau d'aéroports/ferroviaire : si toute paire de villes est desservie par un train/avion direct (un arc/arête).
- Graphe **valué** : les arcs peuvent avoir un poids : temps, distance, prix...
→ Un tel graphe (avec coût) est considéré orienté car les poids peuvent être différents d'une direction à une autre (entre 2 mêmes noeuds), e.g. les taxes locales, ...
- Le réseau d'aéroports/ferroviaire est-il fortement connexe ?
→ Oui si l'on peut aller de n'importe quel point à un autre.
 - Exemple de trafic :
chaque carrefour=un noeud ;
chaque rue=un arc.
Les poids : limite de vitesse ou capacité des rues (nombre de voies).
- Un **arbre** : un graphe acyclique connexe
- Un graphe est **dense** si $|E| = O(|V|^2)$

Tab Mat

1

Les arbres

- Quelques définitions
- Représentation des arbres
- Représentation d' Arbres binaires
- Parcours d'arbre binaire
- Arbres binaires : un exemple Python
- Arbres : représentation par listes imbriquées
- Python : Fonctions usuelles sur arbres binaires
- Arbres : quelques algorithmes
- ABOH en Python
- Exercice sur les Arbres

2

Pile et File en Python

3

Complément : alternatives aux Pile et File en Python

4

Collection et container

5

TAS ou Heap

- TAS en Python

6

En savoir plus : Transformation de la récursivité

- Récursivité terminale
- Récursivité non terminale
- Récursivité non terminale : exemples
- Exemples plus élaborés
- Addendum : Parcours itératif d'arbre binaire

7

Graphes

- Représentation abstraite des graphes
- Parcours général de graphes

Tab Mat (suite)

- AES et graphes

8 Graphes en Python

- Représentation des graphes en Python
- Parcours de graphes en Python
- Les bibliothèques de Graphe de Python
- Python et le module graph_tool

9 Complément : Complexité en Python

10 Addendums à voir Cours 3-4

- Addendum : Équilibrage d'un ABOH
- Addendum : TDA arbre binaire
- Addendum : AVL
- Finish : Addendum : évaluation d'expressions arithmétique

11 Addendum : TDA

- TDA Bool
- TDA entier*
- TDA Liste*
- TDA Ensemble*
- TDA Pile*

12 TDA Graphe et MAP (ABOH)

- TDA Graphe

13 Programmation Dynamique

- Caractéristiques de la PrD
- Algorithme de Roy-Warshall
- PrD et le TSP

Tab Mat (suite)

- PrD et le Sac à Dos

- 14 Annexes
 - Compléments sur les graphes