Algorithmique et Programmation 2

Structures de données en python

Tristan Cazenave

Professeur au LAMSADE à Dauphine

Recherche sur l'intelligence artificielle pour les jeux et l'optimisation

Algorithmes de Monte-Carlo et Deep Learning

Tristan.Cazenave@dauphine.fr

Objectifs

- Notions de complexité
- Tris
- Structures de données :
  - Tableaux
  - Listes chaînées
  - Piles et files
  - Arbres
  - Arbres binaires de recherche

Notions de complexité

- Notion de problème
- Notion d'algorithme
- Tri par sélection
- Analyse des algorithmes
- Tri par insertion
- Complexité des algorithmes

Notion de problème

- Un problème comprend des données en entrée et spécifie une sortie désirée.
- Exemple de problème :
- Étant donnés un tableau de nombres et un nombre, trouver si le nombre fait partie du tableau de nombres.

Notion de problème

- Autre exemple :
- Trier une séquence finie de nombres

  [a1, a2, ..., an]
- Entrée : une séquence de nombres

  [a1, a2, ..., an]
- Sortie : une permutation de la séquence d'entrée [a1', a2', ..., an'] telle que a1' ≤ a2' ≤ ... ≤ an'

Notion de problème

- Un problème peut avoir des instances.
- Définition : L'instance d'un problème est constituée des entrées satisfaisant les contraintes imposées par l'énoncé du problème.
- Exemple : la séquence [10, 4, 2, 9, 8, 3] est une instance du problème du tri.
- Un algorithme de tri appliqué à cette instance donnera en sortie la séquence [2,3,4,8,9,10].

Notion de problème

- Une notion importante est la taille de l'instance.
- En général la taille de l'instance est le nombre d'éléments de l'instance.
- La taille de l'instance précédente pour le problème du tri est par exemple de 6.

Notion de problème

- Pour certains problèmes la taille de l'instance peut être mesurée différemment.
- Pour la multiplication d'entiers, la mesure de la taille est le nombre de bits nécessaires à la représentation mémoire de l'entrée.

Notion d'algorithme

- Définition : Un algorithme est une suite d'instructions simples à exécuter pour résoudre en temps fini un problème posé.
- Définition : Un algorithme est dit correct s'il se termine avec la sortie juste pour toute instance.
- Si un algorithme est correct on dit qu'il résout le problème.

Notion d'algorithme

- Un algorithme a un temps d'exécution. Ce temps d'exécution croît en général avec la taille de l'instance.
- Exemple : trier un tableau de 50 éléments sera en général plus rapide que trier un tableau de 100 éléments, mais pas toujours.
- Il est intéressant d'évaluer le temps d'exécution dans le pire des cas.

Notion d'algorithme

- On peut pour cela trouver une fonction qui donne une limite supérieure au temps d'exécution de l'algorithme pour les instances de grandes tailles.
- On utilise alors la notation O pour exprimer cette borne supérieure asymptotique.

Notion d'algorithme

- Définition : Pour une fonction g donnée, on note O(g) l'ensemble des fonctions :

  $O(g) = \{f \mid \exists c \in N, \exists n0 \in N, \forall n \geq n0 \in N,$

  $f(n) \leq c.g(n)\}$

- On note par abus de langage $f=O(g)$ ssi $f(n) \in O(g)$.

Notion d'algorithme

- Exemple : Pour le problème de recherche un élément dans un tableau non trié.
- Si le nombre d'éléments du tableau est n, la complexité de la recherche dans le pire des cas est O(n) car il faut parcourir le tableau et tester tous ses éléments.
- Exemple graphique de fonction f et de fonction g la majorant asymptotiquement.

Tri par sélection

- Principe :
- Pour chaque élément en partant du début de la liste on cherche le plus petit élément dans le reste de la liste et on l'échange avec l'élément courant.

Tri par sélection

[10,3,4,2,8,6,1,7,9,5]

[1,3,4,2,8,6,10,7,9,5]

[1,2,4,3,8,6,10,7,9,5]

[1,2,3,4,8,6,10,7,9,5]

[1,2,3,4,8,6,10,7,9,5]

[1,2,3,4,5,6,10,7,9,8]

[1,2,3,4,5,6,10,7,9,8]

Tri par sélection

```
l = [10,3,4,2,8,6,1,7,9,5]
for i in range(0, len(l)):
    indice = i
    for j in range(i+1,len(l)):
        if l[j] < l[indice]:
            indice = j
    tmp = l[i]
    l[i] = l[indice]
    l[indice] = tmp
print(l)
```

Analyse des algorithmes

- Analyser un algorithme, c'est prévoir les ressources nécessaires à son exécution (mémoire et temps de calcul)
- S'il existe plusieurs algorithmes pour un problème, analyser ces algorithmes permet de choisir le plus efficace.
- Pour effectuer l'analyse il faut modéliser la machine utilisée
- Hypothèse : on utilise une machine à accès aléatoire (RAM), à processeur unique (instructions exécutées l'une après l'autre).

Analyse des algorithmes

- l'analyse d'algorithmes même simples peut être délicate
- un algorithme peut se comporter différemment pour chaque valeur d'entrée possible
- pour analyser un algorithme on doit définir le temps d'exécution.
- Intuitivement le temps d'exécution d'un algorithme sur une entrée particulière est le nombre d'étapes exécutées.

Tri par insertion

- Principe : s'inspire de la manière dont on trie un paquet de cartes.
- Au départ la main est vide, on prend la première carte, puis on tire une carte et on l'insère à la bonne place dans la main.
- Pour trouver la bonne place, on regarde les cartes de droite à gauche et dès que l'on rencontre une carte plus petite on insère la carte à droite de cette carte.
- Donc lorsqu'on tire la jième carte, les j-1 premières cartes sont triées

Tri par insertion

[10,3,4,2,8,6,1,7,9,5]

[3,10,4,2,8,6,1,7,9,5]

[3,4,10,2,8,6,1,7,9,5]

[2,3,4,10,8,6,1,7,9,5]

[2,3,4,8,10,6,1,7,9,5]

[2,3,4,6,8,10,1,7,9,5]

[1,2,3,4,6,8,10,7,9,5]

[1,2,3,4,6,7,8,10,9,5]

[1,2,3,4,6,7,8,9,10,5]

[1,2,3,4,5,6,7,8,9,10]

Tri par insertion

```
l = [10,3,4,2,8,6,1,7,9,5]
for j in range(1, len(l)):
    valeur = l[j]
    i = j - 1
    while i >= 0 and l[i] > valeur:
        l[i + 1] = l[i]
        i = i-1
    l[i+1] = valeur
print(l)
```

Complexité du tri par insertion

- On suppose que chaque ligne du pseudo-code prend un temps constant ci.
- On doit déterminer pour chaque ligne le nombre de fois qu'elle sera exécutée.
- Pour un j fixé, on note t_j le nombre de fois que le test de la boucle while est exécuté pour cette valeur de j.

Tri par insertion

	coût	nombre de fois
for j in range(1, len(l)):	c1	n
valeur = l[j]	c2	n-1
i = j - 1	c3	n-1
while i >= 0 and l[i] > valeur:	c4	$t_1+t_2+...+t_{n-1}$
l[i + 1] = l[i]	c5	$(t_1-1)+(t_2-1)+...+(t_{n-1}-1)$
i = i-1	c6	$(t_1-1)+(t_2-1)+...+(t_{n-1}-1)$
l[i+1] = valeur	c7	n-1

Tri par insertion

- Le temps d'exécution pour une instance de taille n est T(n) :
- $T(n) = c1.n + c2.(n-1) + c3.(n-1) + c4.(t_1+t_2+..+t_{n-1}) + c5.((t_1-1)+..+(t_{n-1}-1)) + c6.((t_1-1)+..+(t_{n-1}-1)) + c7.(n-1)$

Tri par insertion

- Pour une instance d'une taille donnée, le temps d'exécution dépend de la nature de l'entrée.
- Cas le plus favorable :
- Le tableau est déjà trié. Dans ce cas on a toujours l[i] < valeur dans la boucle while.
- Le test n'est effectué qu'une seule fois pour chaque j. Donc $\forall j\ t_j = 1$.

Tri par insertion

- $T(n) = c1.n + c2.(n-1) + c3.(n-1) + c4.(n-1) + c5.(0) + c6.(0) + c7.(n-1)$
- soit
- $T(n) = n.(c1+c2+c3+c4+c7) - (c2+c3+c4+c7)$
- On obtient donc une fonction linéaire de n de la forme an+b (où a et b sont des constantes).

Tri par insertion

- Cas le moins favorable :
- Le tableau est trié en ordre décroissant inverse.
- Il faut alors comparer la valeur avec chaque élément du sous tableau Tab[1..j-1] déjà trié.
- On a donc $\forall j\ t_j = j$.

Tri par insertion

$T(n) = c1.n + c2.(n-1) + c3.(n-1) + c4.(2+3+..+n) + c5.(1+2+..+(n-1)) + c6.(1+2+..+(n-1)) + c7.(n-1)$

Or on sait que $1 + 2 + 3 + ... + n-1 = n.(n-1) / 2$

Donc :

$T(n) = c1.n + c2.(n-1) + c3.(n-1) + c4.(n.(n+1) / 2 - 1) + c5.(n.(n-1) / 2) + c6.(n.(n-1) / 2) + c7.(n-1)$

Tri par insertion

$T(n) = n^2.(c4 / 2 + c5 / 2 + c6 / 2) + n.(c1 + c2 + c3 + c4 / 2 – c5 / 2 – c6 / 2 + c7) – (c2 + c3 + c4 + c7)$

On obtient une fonction quadratique de n de la forme

$a.n^2 + b.n + c$

Complexité d'un algorithme

- Définition : on appelle opération élémentaire une opération dont le temps d'exécution peut être borné supérieurement par une constante ne dépendant que de l'implantation spécifique (matériel, langage, compilateur).
- Exemples :
  - opération arithmétique de base (+,-,/,*),
  - test binaire (a>b),
  - affectation.

Complexité d'un algorithme

- La complexité d'un algorithme est le nombre d'opérations élémentaires nécessaires à l'exécution de l'algorithme.
- La complexité dépend en général de la taille de l'instance du problème.

Complexité d'un algorithme

- Analyse dans le meilleur des cas : cas où les données sont les plus favorables pour le problème. Nombre d'opérations élémentaires minimum.
- Analyse au pire des cas : cas où les données sont les plus mal disposées pour le problème. Nombre d'opérations élémentaires maximum.
- Analyse en moyenne : cas où on étudie la complexité de l'algorithme quand les données sont disposées de façon aléatoire. Nombre d'opérations élémentaires moyen.
- On étudiera principalement le pire des cas.

Complexité d'un algorithme

- Motivations :
- Le temps d'exécution dans le pire des cas est une borne supérieure du temps d'exécution d'une entrée quelconque de taille donnée. On a donc une garantie que l'algorithme ne prendra jamais plus de temps que le pire des cas pour une instance de même taille.
- Pour certains algorithmes le pire cas survient souvent. Par exemple lorsqu'on cherche si un élément est dans un ensemble, le cas où l'élément n'est pas dans l'ensemble peut arriver souvent et c'est le pire des cas.

Complexité d'un algorithme

- Il n'est pas rare que le cas moyen soit à peu près aussi mauvais que le pire des cas.
- Par exemple pour le tri par insertion :
- La complexité dépend fortement du nombre de fois où on exécute la boucle while, c'est à dire de tj.
- meilleur cas -> tj = 1
- pire des cas -> tj = j
- en moyenne l'élément valeur = Tab[j] se placera au milieu du tableau, d'où tj = j/2
- Si on calcule T(n) on obtient une fonction quadratique avec des constantes différentes du pire des cas.

Complexité d'un algorithme

- On étudiera parfois la complexité en moyenne, mais elle est souvent difficile à déterminer.
- Qu'est ce qu'une entrée moyenne ?
- Toutes les entrées sont équiprobables

Deux types d'étude de la complexité

- Étude Empirique :
- Calculer pour un grand nombre d'instances de taille donnée le temps nécessaire à la résolution.
- On estime le temps moyen de résolution en faisant la moyenne des temps d'exécution.

Deux types d'étude de la complexité

- Inconvénients :
- nécessite matériel et temps pour tester un grand nombre de cas
- il faut s'assurer que l'on génère les différentes instances que l'on peut rencontrer
- cela suppose l'équiprobabilité des instances
- pas de garantie sur le temps maximum à l'exécution

Deux types d'étude de la complexité

- Avantages :
- Donne une bonne idée du temps de réponse moyen de l'algorithme
- Sur des gros problèmes dont les instances sont complexes, c'est l'approche utilisée en pratique.

Deux types d'étude de la complexité

- Étude théorique de la complexité :
- évaluer le nombre d'opérations élémentaires nécessaires à l'exécution de l'algorithme.
- comme ce nombre est difficile à obtenir, on étudie la complexité au pire des cas.
- On trouve un majorant du nombre d'opérations qui dépend de la taille du problème posé.

Deux types d'étude de la complexité

- On compare les algorithmes pour les problèmes de grande taille.
- On étudie l'efficacité asymptotique :
- Algo1 est meilleur que Algo2 si $\forall n \geq n0$:

  $T\ Algo1\ (n) \leq T\ Algo2\ (n)$
- Pour cela on a besoin d'un certain nombre de notations et de propriétés.

Notations et propriétés asymptotiques

- Les notations utilisées pour décrire le temps d'exécution asymptotique d'un algorithme sont définies en terme de fonctions sur N :

  T : N -> R

  n -> T(n)

Notations et propriétés asymptotiques

- Notation Θ
- Définition : pour une fonction g(n), on note $\Theta(g(n))$ l'ensemble des fonctions :

  $\Theta(g) = \{f \mid \exists\, c1, c2 \in N*, \exists\, n0 \in N, \forall n \geq n0 \in N, 0 \leq c1.g(n) \leq f(n) \leq c2.g(n)\}$
- $f(n) \in \Theta(g(n))$ si f(n) peut être prise en sandwich entre c1.g(n) et c2.g(n) à partir d'un certain rang pour n.
- Par abus de notation, on écrit $f(n) = \Theta(g(n))$ pour dire que $f(n) \in \Theta(g(n))$.

Notations et propriétés asymptotiques

- Exercice : montrez que $3n^2 - 10n = \Theta(n^2)$

Notations et propriétés asymptotiques

- Montrons que $3n^2 - 10n = \Theta(n^2)$
- On cherche à déterminer c1, c2 et n0 tels que :
- $c1.n^2 \leq 3n^2 - 10n \leq c2.n^2 \; \forall n \geq n0$
- comme $n^2 > 0$ pour $n > 1$ on divise tout par n^2 :
- $c1 \leq 3 - 10/n \leq c2 \; \forall n \geq n0$

Notations et propriétés asymptotiques

- partie gauche : $c1 \leq 3 - 10/n$
- Si on choisit c1=2, on a
- $2 \leq 3 - 10/n$
- $10/n \leq 1$
- $n \geq 10$
- c'est donc vérifié avec c1=2 pour n0=10

Notations et propriétés asymptotiques

- partie droite : $3 - 10/n \leq c2$
- Si on choisit c2=3, on a
- $3 - 10/n \leq 3$
- $10/n \geq 0$
- qui est vérifié pour n0=10

Notations et propriétés asymptotiques

- Donc

  $\forall n \geq 10 \qquad 2.n^2 \leq 3n^2 - 10n \leq 3.n^2$

- Remarque : il suffit d'exhiber un cas

Notations et propriétés asymptotiques

- Exercice : Montrer que $6n^3 = \Theta(n^2)$

Notations et propriétés asymptotiques

- Supposons que $6n^3 = \Theta(n^2)$
- on doit avoir $6n^3 \leq c2.n^2 \ \forall n \geq n0$
- donc $6n^2 \leq c2.n$
- or pour n arbitrairement grand c'est impossible car c2 est une constante.
- => de tels c2 et n0 n'existent pas
- donc $6n^3$ est différent de $\Theta(n^2)$

Notations et propriétés asymptotiques

- Intuitivement, les termes d'ordre inférieur d'une fonction positive asymptotiquement peuvent être ignorés ainsi que le coefficient du terme d'ordre max.
- Pour appliquer la définition, il suffit de choisir c1 un peu plus petit que le coefficient associé au terme de plus grand ordre et pour c2 une valeur légèrement plus grande.

Application au tri par insertion

- On a montré que dans le pire des cas, le temps d'exécution du tri par insertion est

  $T(n) = a.n^2 + b.n + c$ avec a, b et c constantes.
- Montrons que $T(n) = \Theta(n^2)$
- On dira que la complexité au pire des cas du tri par insertion est en $\Theta(n^2)$.

Application au tri par insertion

- Pour cela montrons que :
- Soit p(x) un polynôme de degré k quelconque à coefficients positifs, alors $p(n) \in \Theta(n^k)$.
- Il faut montrer $\exists$ c1, c2, n0 $\in \mathbb{N}*$ tels que :

$\forall n \geq n0 \qquad c1.n^k \leq p(n) \leq c2.n^k$

Application au tri par insertion

- soit $p(x) = \Sigma\ ai\ x^i$ avec $ai \geq 0$ et $ak > 0$
- posons $c1 = ak$
- de façon évidente on a :
- $c1.n^k \leq p(n) \quad \forall n \geq 1$

Application au tri par insertion

- posons $c2 = \Sigma\ ai$
- on a bien
- $p(n) \leq c2.n^k \qquad \forall n \geq 1$
- cqfd.

Notation O

- Quand on dispose d'une borne supérieure asymptotique, on utilise la notation O
- Définition : pour une fonction g(n), on note O(g(n)) l'ensemble des fonctions :
- $O(g) = \{f \mid \exists c \in N*, \exists n0 \in N, \forall n \geq n0 \in N,$

  $0 \leq f(n) \leq c.g(n)\}$

Notation O

- La notation O vise à donner une borne sup à une fonction, on écrit f (n) = O(g(n))
- Exemple graphique de f(n) et c.g(n)
- Remarque :

  si f (n) = Θ(g(n)) alors f (n) = O(g(n))

Notation O

- Exemple : comme $3n^2 - 10n = \Theta(n^2)$

  alors $3n^2 - 10n = O(n^2)$
- Montrons que $10n = O(n^2)$
- $10n \leq n^2$ pour $n0 = 10$
- La définition est vérifiée pour $c=1$ et $n0 = 10$.

Utilisation en théorie de la complexité

- Puisque la notation O décrit une borne sup, quand on l'utilise pour borner le temps d'exécution d'un algorithme dans le pire des cas, on borne aussi le temps d'exécution pour une entrée quelconque.
- Application : Pour le tri par insertion, la borne en $O(n^2)$ obtenue pour le pire cas s'applique à toute instance.

Notation Ω

- Cette notation fournit une borne asymptotique inférieure.
- Définition : pour une fonction g(n), on note Ω(g(n)) l'ensemble des fonctions :
- $\Omega(g) = \{f \mid \exists c \in N*, \exists n0 \in N, \forall n \geq n0 \in N, 0 \leq c.g(n) \leq f(n)\}$

Notation Ω

- Théorème : Pour deux fonctions f(n) et g(n) on a

  f (n) = Θ(g(n)) ssi f (n) = O(g(n)) et f (n) = Ω(g(n)) .
- Application à la complexité : comme la notation Ω décrit une borne inférieure quand on l'utilise pour borner le temps d'exécution d'un algorithme dans le meilleur des cas, on borne également le temps d'exécution pour toute entrée.

Complexité d'un algorithme

- On cherche en général à déterminer la complexité théorique d'un algorithme dans le pire des cas.
- On utilise la notation O puisque c'est celle qui donne une borne sup.

Ordres de grandeur de complexités

Complexité croissante	type
$O(1)$	constante
$O(\ln(n))$	logarithmique
$O(n)$	linéaire
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(n^k)$	polynomiale
$O(e^n)$	exponentielle

Méthodes de calcul de la complexité

- Consiste à déterminer un majorant du nombre d'opérations élémentaires nécessaires à l'exécution d'un algorithme.
- Les opérations arithmétiques de base, les comparaisons (>, <, =, =) sont des opérations élémentaires en O(1).
- La complexité d'une affectation est en O(1).
- La complexité d'une opération de lecture ou d'écriture est en O(1)

Méthodes de calcul de la complexité

- La complexité d'une suite d'instructions correspond à la complexité la plus forte parmi toutes les instructions de la suite.
- Instruction conditionnelle :

  Si condition alors I1 sinon I2.

  La complexité est la somme de la complexité de l'évaluation de la condition et de la complexité la plus grande entre I1 et I2.

Méthodes de calcul de la complexité

- La complexité d'une boucle est la somme sur toutes les itérations de la complexité des instructions rencontrées dans le corps de la boucle plus la complexité du test de sortie de la boucle.
- Procédure récursive : pour déterminer sa complexité, il faut lui associer une fonction de complexité inconnue T(n) et cherche une équation de récurrence qu'il faut alors résoudre.

Comparaison de l'efficacité de différents algorithmes

- Problème : Soit p un polynôme de degré n, à coefficients réels, dont on veut calculer la valeur en un point réel.
- $p(x) = a0 + a1\ x + a2\ x^2 + ... + an\ x^n$

Comparaison de l'efficacité de différents algorithmes

```
x = 3
poly = [1,3,4,2]  # 1 + 3x + 4x^2 + 2x^3
p = 0
for i in range (len(poly)):
    pi = 1
    for j in range (i):
        pi = pi * x
    p = p + poly [i] * pi
print (p)
```

- Complexité ?

Comparaison de l'efficacité de différents algorithmes

```
x = 3
poly = [1,3,4,2]  # 1 + 3x + 4x^2 + 2x^3
p = 0
for i in range (len(poly)):
    pi = 1
    for j in range (i):
        pi = pi * x
    p = p + poly [i] * pi
print (p)
```

- Complexité en $O(n^2)$ si polynôme de degré n

Comparaison de l'efficacité de différents algorithmes

```
x = 3
poly = [1,3,4,2]  # 1 + 3x + 4x^2 + 2x^3
p = 0
q = 1
for i in range (len(poly)):
    p = p + poly [i] * q
    q = q * x
print (p)
```

- Complexité en O(n) si polynôme de degré n

Algorithme de Horner

- Principe :
- Effectuer les calculs en partant de a_n
- Puis $a_n * x + a_{n-1}$
- Puis $(a_n * x + a_{n-1}) * x + a_{n-2}$
- etc.

Comparaison de l'efficacité de différents algorithmes

```
x = 3
poly = [1,3,4,2] # 1 + 3x + 4x^2 + 2x^3
i = len (poly) - 1
p = poly [i]
while i > 0:
  i = i - 1
  p = p * x + poly [i]
print (p)
```

- Complexité en O(n) si polynôme de degré n
- Il demande moins d'opérations que l'algorithme précédent

Comparaison de l'efficacité de différents algorithmes

- Attention : Il faut réfléchir avant de mettre en place un algorithme pour un problème donné. Les algorithmes prévus pour résoudre un problème diffèrent souvent énormément par leur efficacité.
- Illustration sur le problème du tri :
- Le tri par insertion est en $O(n^2)$.
- Le tri fusion est en $O(n * \ln(n))$

Comparaison de l'efficacité de différents algorithmes

- Supposons que le tri fusion soit programmé sur un ordinateur personnel exécutant 10^9 instructions par secondes et que le tri insertion soit programmé sur un super ordinateur exécutant 10^{11} instructions par secondes.
- Pour trier un tableau de 10^6 valeurs on aura :
- super ordinateur :

  $(10^6)^2 / 10^{11} = 10$ secondes
- ordinateur personnel :

  $10^6 * \ln(10^6) / 10^9 = 0.02$ secondes.

Structures de données
listes, piles, files

- Il est souvent utile de regrouper des données différentes dans une même structure.

- Nous allons montrer comment représenter ces structures et comment les organiser simplement à l'aide de tableaux, de listes, de piles et de files.

Exemple de l'annuaire

- Un annuaire est un ensemble d'enregistrements.
- Chaque enregistrement fournit des informations sur un élément de l'annuaire.

Exemple de l'annuaire

- Par exemple :

- Un annuaire des produits vendus par une société ; les informations relatives à chaque produit pourraient être son code, son libellé, son prix, une brève description, etc.

Exemple de l'annuaire

- Autres exemples :

- Un annuaire des employés d'une société ; les informations relatives à chaque employé pourraient être son adresse électronique, son nom, son prénom, le département auquel il est rattaché, etc.

- Un annuaire des patients de la Sécurité Sociale ; les informations relatives à chaque patient pourraient être son numéro de SS, son nom, son prénom, son adresse, etc.

Exemple de l'annuaire

- Les mêmes informations sont présentes dans chaque enregistrement d'un même annuaire.
- Pour représenter un enregistrement, on utilise un constructeur de type de données qui s'appelle une classe.

Exemple de l'annuaire

```
class Patient (object):
    def __init__(self, entier):
        self.numero = entier
        self.nom = None
        self.prenom = None
```

Exemple de l'annuaire

- Soit unpatient une variable de type patient.

- On la déclare par :

  unpatient = Patient (12345678)

- On accède aux différents champs de la variable par :

  unpatient.numéro
  unpatient.nom
  unpatient.prénom

Exemple de l'annuaire

- Parmi les champs d'un enregistrement, il y a un champ particulier qui permet d'identifier l'enregistrement :
- Deux enregistrements différents ne peuvent avoir la même valeur pour ce champ.
- Dans les exemples précédents, il s'agit du code produit, de l'adresse électronique et du numéro de Sécurité Sociale.
- Dans la suite, nous l'appellerons id (pour identifiant).

Opérations sur un annuaire

- Les opérations usuelles sur un annuaire :

- La recherche des informations relatives à un enregistrement désigné par son identifiant. Cette opération doit prévoir que l'enregistrement peut ne pas exister.

- L'ajout d'un enregistrement désigné par son identifiant. Cette opération doit interdire l'ajout si un enregistrement de même identifiant est déjà présent.

Opérations sur un annuaire

- La suppression d'un enregistrement désigné par son identifiant. La suppression n'a lieu que si l'enregistrement existe.
- La modification d'un enregistrement désigné par son identifiant avec de nouvelles informations. La modification n'a lieu que si l'enregistrement existe.

Gestion à l'aide d'un tableau

- Prenons le cas de l'annuaire de la Sécurité Sociale, l'identifiant est un entier.
- Utilisons cet identifiant comme indice de l'enregistrement dans le tableau.
- Certains numéros n'ont pas d'enregistrement associé => Dans ce cas l'élément à cet indice dans le tableau prend la valeur None.

Gestion à l'aide d'un tableau

```
t = []
for i in range (1000000):
    t.append (None)
unpatient = Patient (12345)
unpatient.nom = "Cazenave"
unpatient.prenom = "Tristan"
t[unpatient.numero] = unpatient
print (t[unpatient.numero].nom)
print (t[55])
```

Gestion à l'aide d'un tableau

```
def chercher (id):
    return t [id]

a = chercher (unpatient.numero)
print a.nom
```

Gestion à l'aide d'un tableau

```
def ajouter(patient):
    if t [patient.numero] != None:
        return False
    else:
        t [patient.numero] = patient
    return True


p = Patient (65432)
p.nom = "Turing"
p.prenom = "Alan"
b = ajouter (p)
print (b)
print (t [65432].nom)
```

Gestion à l'aide d'un tableau

```
def supprimer (id):
    if t [id] == None:
        return False
    else:
        t [id] = None
    return True

supprimer (65432)
print t [65432]
```

Gestion à l'aide d'un tableau

- A priori, cette gestion semble tout à fait satisfaisante puisque toutes les opérations se font en temps constant.
- Cependant cette simplicité masque un inconvénient rédhibitoire : la place occupée.

Gestion à l'aide d'un tableau

- Le tableau est indicé par l'ensemble des identifiants possibles.
- Pour les numéros de sécurité sociale, cela conduit à un tableau de 10^{15} entrées (ce qui dépasse la taille de la mémoire vive qui est de l'ordre de 10^{9} octets)
- Or on estime le nombre de numéros attribués à 10^{8} .
- Il est nécessaire d'utiliser une structure qui prenne moins de place.

Gestion à l'aide d'un tableau séquentiel

- Une autre possibilité consiste à prévoir un tableau dont la taille peu augmenter avec le nombre d'enregistrements.
- Les enregistrements sont stockés par indice croissant au fur et à mesure de leur ajout.
- Un compteur sommet permet de savoir où s'arrêtent les enregistrements.

Gestion à l'aide d'un tableau séquentiel

- La recherche se fait "séquentiellement" en parcourant le tableau du premier indice au sommet.
- Ce parcours est interrompu si on a rencontré l'enregistrement.
- Exercice : écrire une fonction qui cherche un enregistrement à partir de son identifiant

Gestion à l'aide d'un tableau séquentiel

```
t = []

def chercher (id):
    for i in range (len (t)):
        if (t [i].numero == id):
            return i
    return -1
```

Gestion à l'aide d'un tableau séquentiel

- Pour ajouter un élément on utilise append qui fait grandir la taille du tableau
- Si l'enregistrement existe déjà on ne le rajoute pas.
- Exercice : écrire une fonction ajouter

Gestion à l'aide d'un tableau séquentiel

```
def ajouter(patient):
    i = chercher (patient.numero)
    if (i != -1):
        return False
    else:
        t.append (patient)
        return True
```

Gestion à l'aide d'un tableau séquentiel

- Pour supprimer un élément de ce tableau :
- On cherche si l'enregistrement est dans le tableau.
- Si c'est le cas on doit déplacer vers le bas les éléments situés au dessus de l'enregistrement supprimé.
- Exercice : écrire une fonction Supprimer.

Gestion à l'aide d'un tableau séquentiel

```
def supprimer (id):
    i = chercher (id)
    if (i == -1):
        return False
    else:
        for j in range (i + 1, len(t)):
            t [j -1] = t [j]
        t.pop ()
        return True
```

Gestion à l'aide d'un tableau séquentiel

```
p = Patient (65432)
p.nom = "Turing"
p.prenom = "Alan"
b = ajouter (p)

a = chercher (65432)
print t[a].nom

supprimer (65432)
print sommet
```

Gestion à l'aide d'un tableau séquentiel

Inconvénient de la représentation par un tableau séquentiel :

- La suppression d'un élément nécessite la recopie de tous les éléments suivants.
- Pour implémenter un tableau on doit utiliser une grande zone de mémoire contiguë.

Gestion à l'aide d'une liste

- Pour éviter les inconvénients du tableau séquentiel on utilise des listes
- La suppression ne demande pas de recopie
- La mémoire occupée est directement proportionnelle au nombre d'éléments stockés.

Liste chaînée

```
class Cellule (object):
    def __init__(self, entier):
        self.entier = entier
        self.suiv = None
```

Liste chaînée

- Au début d'une liste chaînée on met une cellule en-tête qui ne contient aucune information
- Elle sert juste à conserver le début de la liste même lorsqu'elle est vide.

Liste chaînée

```
entete = Cellule (0)
courant = entete
for i in range (1, 11):
    courant.suiv = Cellule (i)
    courant = courant.suiv
```

Représentation de la liste :

[0,-]->[1,-]->[2,-]->…->[10,None]

Exercice : Afficher tous les éléments d’une liste

Liste chaînée

```
courant = entete.suiv
while (courant != None):
    print (courant.entier)
    courant = courant.suiv
```

Liste chaînée pour l'annuaire

```
class Cellule (object):
    def __init__(self, entier):
        self.numero = entier
        self.nom = None
        self.prenom = None
        self.suiv = None

entete = Cellule (-1)
```

Exercice : chercher un élément d’une liste

Liste chaînée pour l'annuaire

```
def chercher (id):
    p = entete.suiv
    while p != None:
        if p.numero == id:
            return p
        p = p.suiv
    return None
```

Exercice : ajouter un élément à une liste

Liste chaînée pour l'annuaire

```
def ajouter (p):
    suiv = entete.suiv
    entete.suiv = p
    entete.suiv.suiv = suiv

ajouter (Cellule(1024))
```

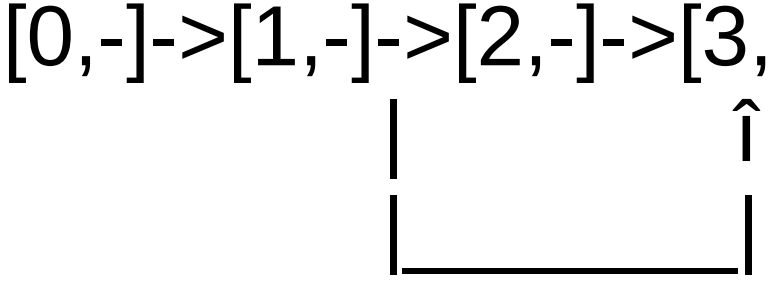
[0,-]->[1,-]->[2,-]->[3,-]->None
[0,-] → [1024] → [1,-] → [2,-] → [3,-] → None

Exercice : supprimer un élément d’une liste

Liste chaînée pour l'annuaire

```
def supprimer (id):
    p = entete
    while p.suiv != None:
        if p.suiv.numero == id:
            p.suiv = p.suiv.suiv
            return
        p = p.suiv
```

supprimer (2) :



Liste chaînée pour l'annuaire

```
p = Cellule(10)
p.nom = "Turing"
p.prenom = "Alan"
ajouter (p)
p1 = chercher(10)
print (p1.nom)
supprimer (10)
p1 = chercher (10)
if (p1 == None):
    print ("None")
```

Structures de données
listes, piles, files

Tri Fusion Tableaux
Tri Fusion Listes
Fractale
Courbe du dragon
Exercice

Tri Fusion

Principe :

- On sépare la liste en deux
- On trie chaque partie séparément
- On fusionne les deux listes triées pour reconstituer une liste triée.

Tri Fusion

Exemple :

[5, 3, 10, 4]

Séparation : [5, 3] et [10, 4]

Tri : [3, 5] et [4, 10]

Fusion : [3, 4, 5, 10]

Tri Fusion

Séparation en deux listes :

On prend dans une première liste les éléments jusqu'à longueur / 2 et dans une deuxième liste le reste des éléments.

Tri Fusion

Exemple pour un tableau :

```
def separation (t):
    t1 = t [0:len(t)//2]
    t2 = t [len(t)//2:len(t)]
    return t1,t2
```

Tri Fusion

Fusion de deux listes triées :

On crée un nouveau tableau en ajoutant au fur et à mesure l'élément le plus petit entre les deux tableaux à ajouter.

Fusion de [1, 3] et [2, 10] :

On ajoute 1 car il est plus petit que 2
Puis 2 car il est plus petit que 3
Puis 3 car il est plus petit que 10
Et enfin 10 => on retourne [1, 2, 3, 10]

Tri Fusion

```
def fusion (t1, t2):
    if t1 == []:
        return t2
    if t2 == []:
        return t1
    t = []
    i1 = 0
    i2 = 0
```

```
while (i1 < len (t1) or i2 < len (t2)):
    if (i1 == len (t1)):
        t.append (t2 [i2])
        i2 = i2 + 1
    elif (i2 == len (t2)):
        t.append (t1 [i1])
        i1 = i1 + 1
    else:
        if (t1 [i1] < t2 [i2]):
            t.append (t1 [i1])
            i1 = i1 + 1
        else:
            t.append (t2 [i2])
            i2 = i2 + 1
return t
```

Tri Fusion

Algorithme de tri :

Si le tableau a zéro ou un élément on le renvoie

Sinon :
- On sépare en deux tableaux,
- On trie chaque tableau
- On fusionne les tableaux triés

Ecrire l'algorithme de tri fusion à l'aide des fonctions separation et fusion.

Tri Fusion

```
def tri (t):
    n = len (t)
    if n < 2:
        return t
    else:
        t1,t2 = separation (t)
        t1 = tri (t1)
        t2 = tri (t2)
        return fusion (t1, t2)
```

Tri Fusion

On utilise la classe suivante pour les listes :

```
class Cellule (object):
    def __init__(self, entier):
        self.entier = entier
        self.suiv = None
```

Tri Fusion

La fusion de deux listes triées consiste à créer une nouvelle liste à partir des cellules des deux listes.

Principe : on parcourt les deux listes et on ajoute à chaque étape dans la liste principale le plus petit élément entre les deux listes.

Tri Fusion

```
def fusion (l1, l2):
    if l1.suiv == None:
        return l2
    if l2.suiv == None:
        return l1
    courant = Cellule (0)
    l = courant
    l1 = l1.suiv
    l2 = l2.suiv
```

```
while l1 != None and l2 != None:
    if l1.entier < l2.entier:
        courant.suiv = l1
        courant = courant.suiv
        l1 = l1.suiv
    else:
        courant.suiv = l2
        courant = courant.suiv
        l2 = l2.suiv
if l1 == None:
    courant.suiv = l2
elif l2 == None:
    courant.suiv = l1
return l
```

Tri Fusion

Pour séparer une liste en deux listes égales à un près on va avoir besoin de la taille d'une liste :

```
def taille (l):
    taille = 0
    while l.suiv != None:
        taille = taille + 1
        l = l.suiv
    return taille
```

Tri Fusion

La séparation prend une liste en argument et renvoie deux listes.

On va parcourir la liste jusqu'à la moitié.

Mémoriser la suite de la liste dans la deuxième liste.

Mettre un suivant à None pour arrêter la première liste à la moitié.

Tri Fusion

```
def separation (l):
    l1 = l
    l2 = Cellule (0)
    t = taille (l)
    for i in range (t // 2):
        l = l.suiv
    l2.suiv = l.suiv
    l.suiv = None
    return l1,l2
```

Tri Fusion

Le tri fusion est récursif.

Si la taille de la liste est < 2 la récursivité s'arrête.

Sinon on sépare en deux listes.

On trie chaque liste.

On fusionne les deux listes triées.

Tri Fusion

```
def tri (l):
    if (taille (l) < 2):
        return l
    l1,l2 = separation (l)
    l1 = tri (l1)
    l2 = tri (l2)
    l = fusion (l1, l2)
    return l
```

Tri Fusion

Pour le test du programme de tri fusion on va avoir besoin d'afficher des listes :

```
def affiche (l):
    while l.suiv != None:
        print (l.suiv.entier)
        l = l.suiv
```

Tri Fusion

```
entete = Cellule (0)
courant = entete
for i in range (1, 11):
    courant.suiv = Cellule (10 - i)
    courant = courant.suiv
print ("l avant : ")
affiche (entete)
l = tri (entete)
print ("l apres : ")
affiche (l)
```

Fractale

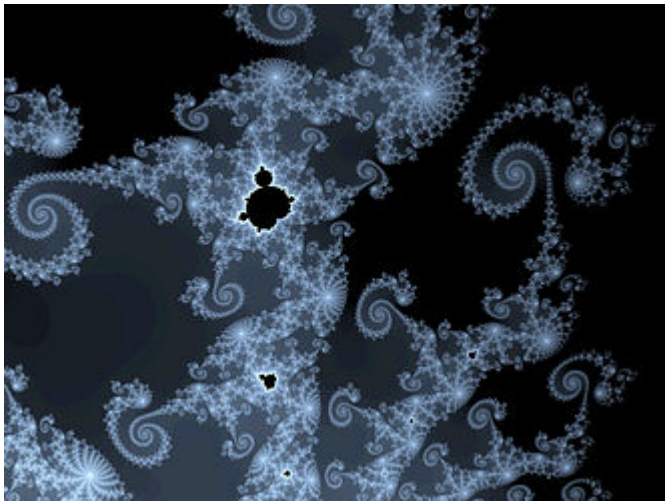
Une fractale est une courbe auto-similaire.

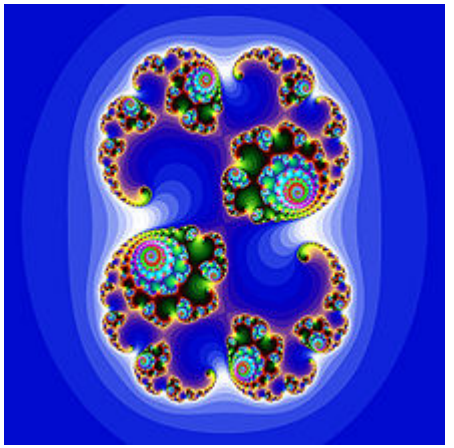
Si on agrandit une partie de la courbe on retombe sur la même courbe.

Fractale est un terme qui a été crée par le mathématicien Benoît Mandelbrot.

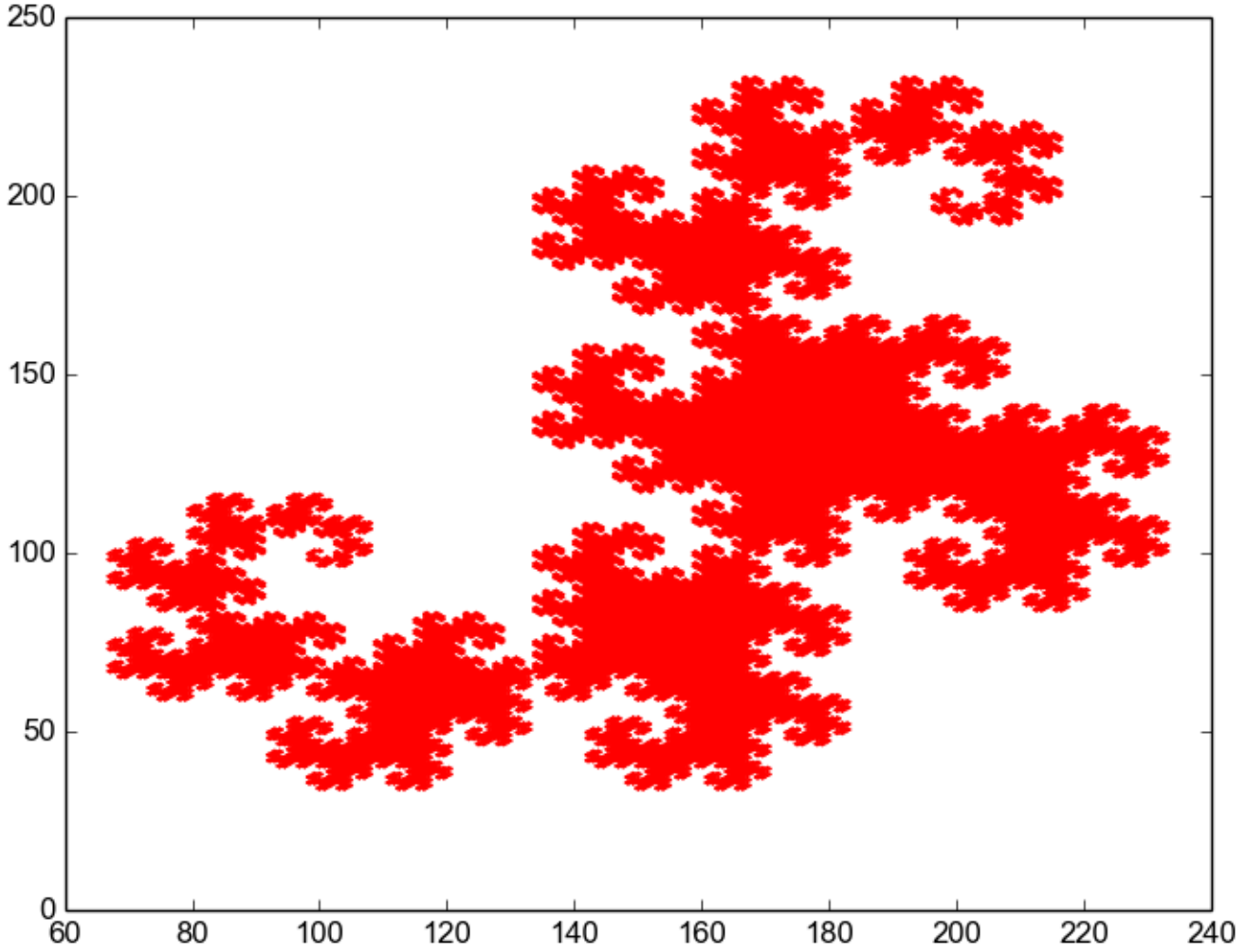
On trouve des objets fractals dans la nature : arbres, flocons, brocolis, vaisseaux sanguins, rivières, univers.

Fractale





La courbe du dragon



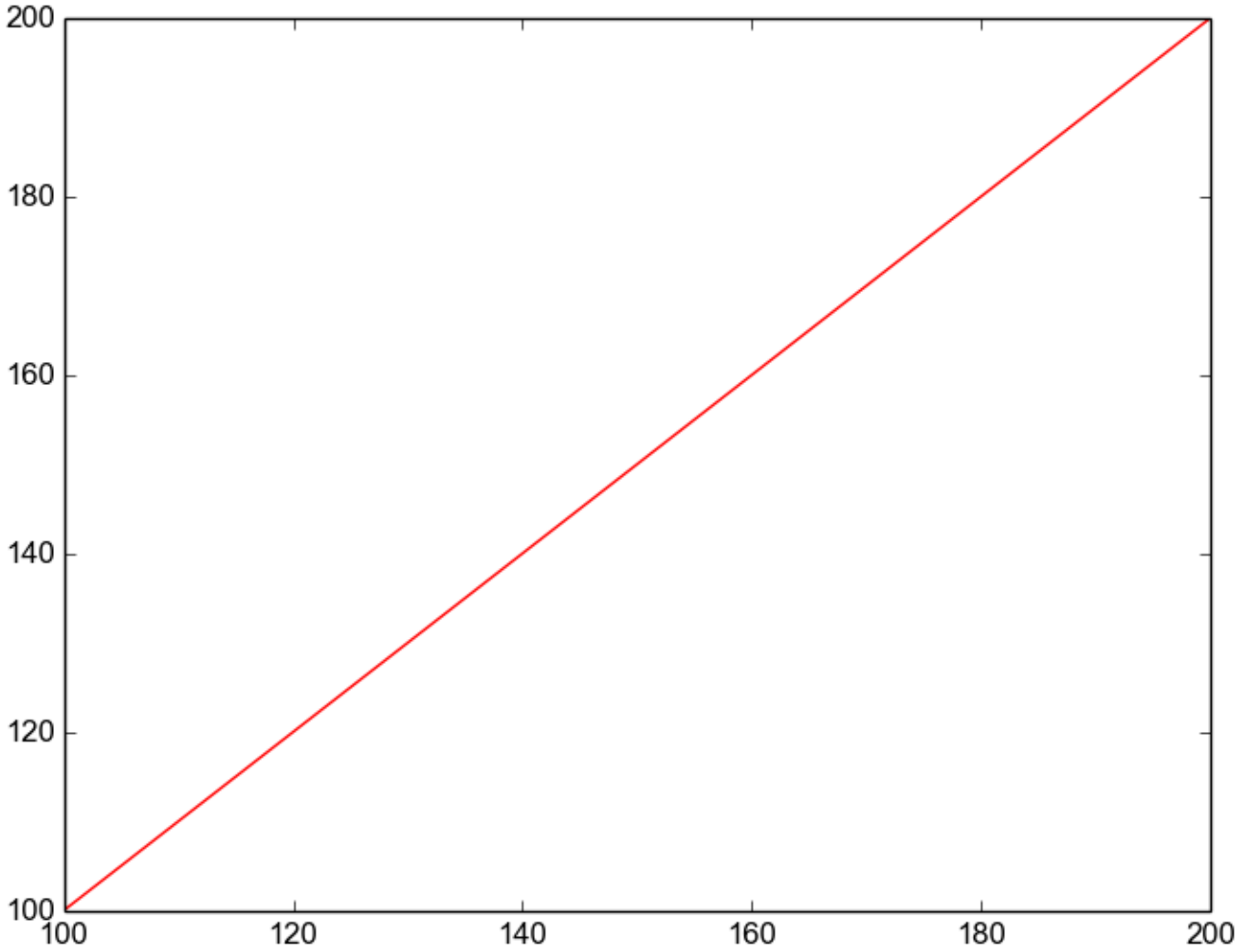
La courbe du dragon

La courbe d'ordre 1 est un segment de droite.

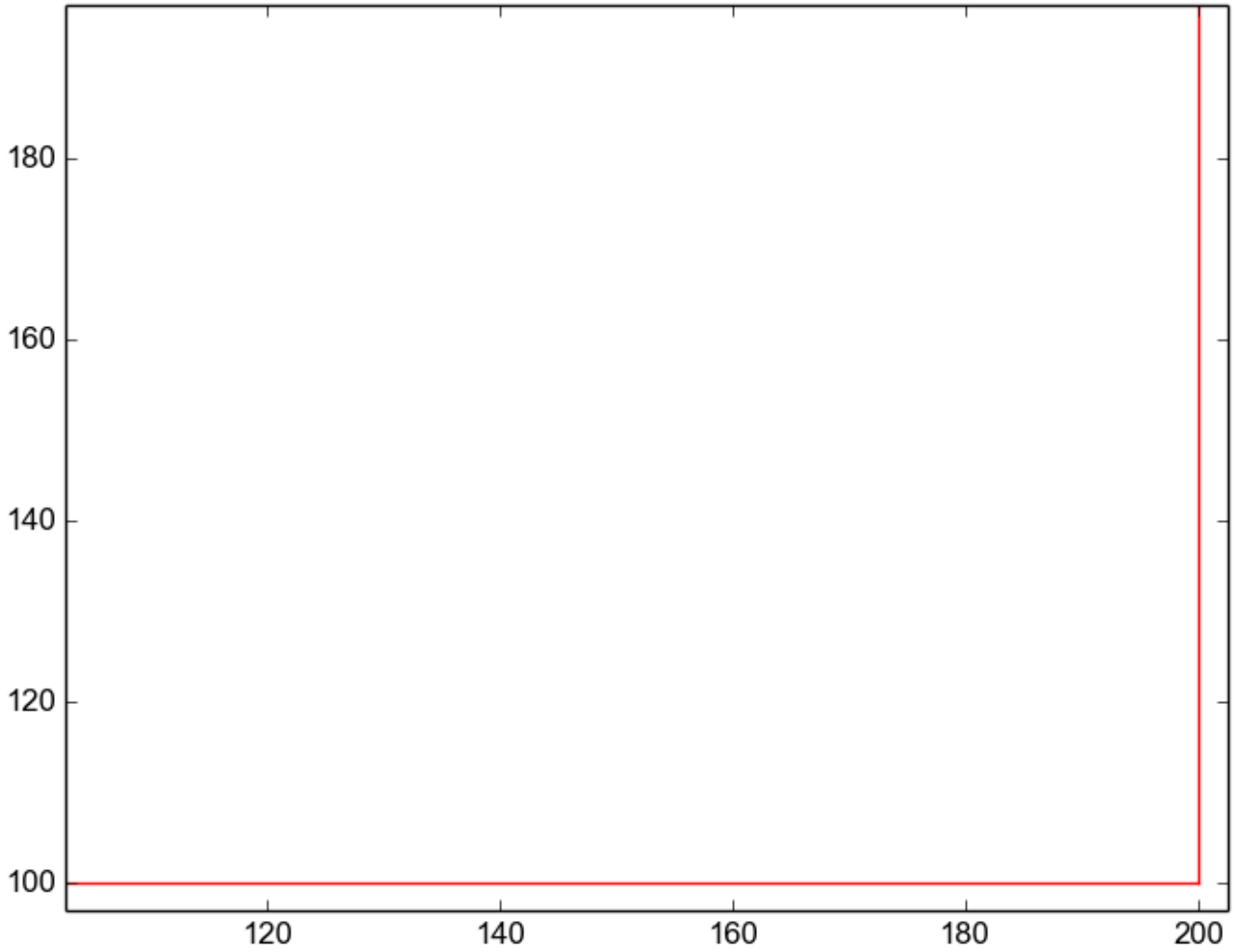
Le courbe d'ordre 2 remplace la courbe d'ordre 1 par deux segments qui ont les extrémités du segment d'ordre 1 et qui forment un angle droit.

La courbe d'ordre n est composée de deux courbes d'ordre n-1 qui forment un angle droit.

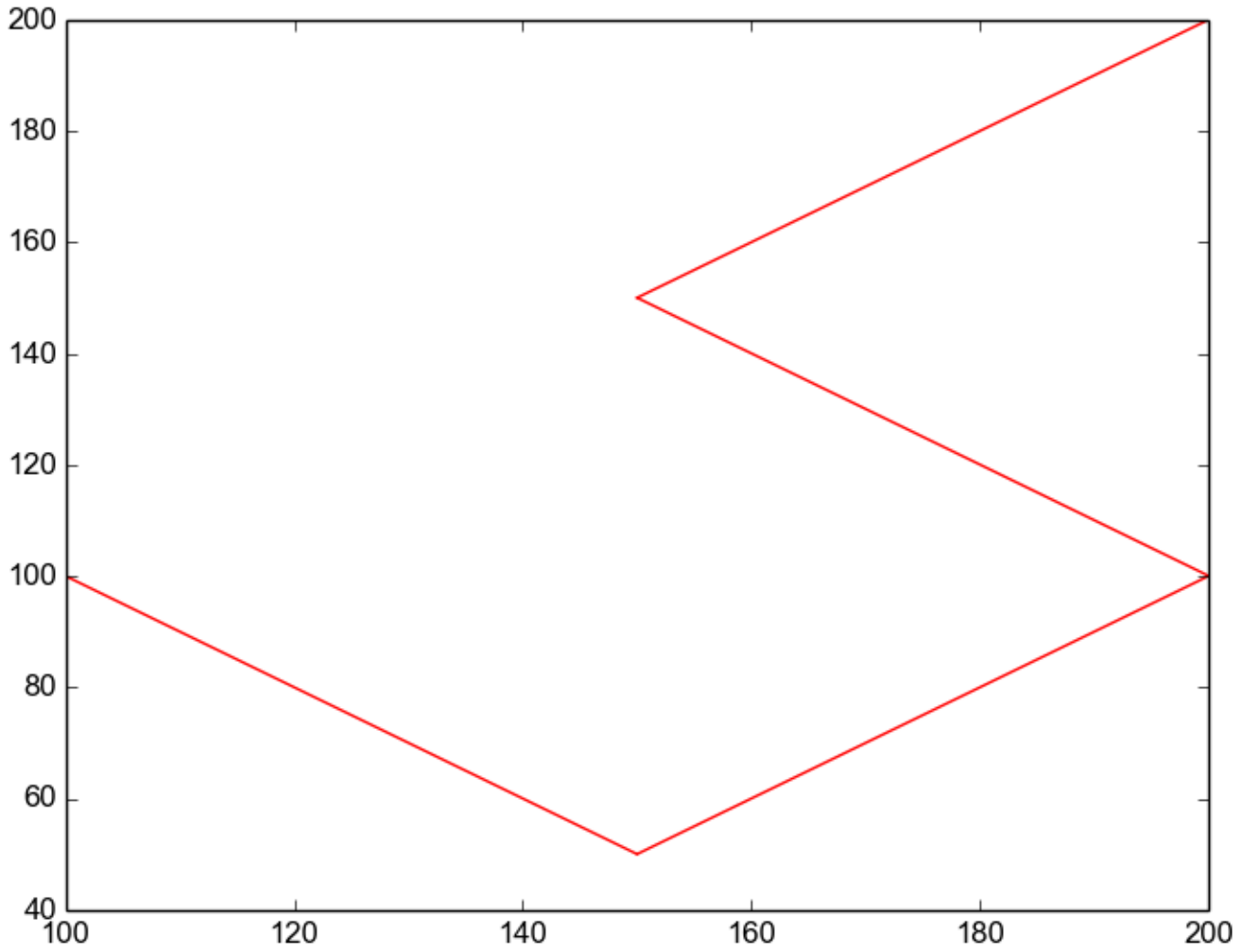
La courbe du dragon



La courbe du dragon



La courbe du dragon



La courbe du dragon

```
import matplotlib.pylab as plt

def dragon (n, x, y, z, t):
    if (n == 1):
        plt.plot ([x, z], [y, t], color='r');
    else:
        u = (x + z + t - y) / 2
        v = (y + t - z + x) / 2
        dragon (n - 1, x, y, u, v)
        dragon (n - 1, z, t, u, v)

dragon (17, 100.0, 100.0, 200.0, 200.0)
plt.show ()
```

Exercice

Ecrire une fonction qui vérifie si une liste chaînée est incluse dans une deuxième liste chaînée.

Exercice

```
def inclus (l1, l2):
    p = l1.suiv
    while p != None:
        dansl2 = False
        p1 = l2.suiv
        while p1 != None:
            if p.entier == p1.entier:
                dansl2 = True
            p1 = p1.suiv
        if dansl2 == False:
            return False
        p = p.suiv
    return True
```

Exercice

```
entete = Cellule (0)
courant = entete
for i in range (1, 11):
    courant.suiv = Cellule (i)
    courant = courant.suiv

entete2 = Cellule (0)
courant2 = entete2
entete2.suiv = Cellule (3)
entete2.suiv.suiv = Cellule (5)
print (inclus (entete2, entete))
entete2.suiv.suiv.suiv = Cellule (20)
print (inclus (entete2, entete))
```

Pile et File

- Pile avec un tableau
- Pile avec une liste chaînée
- File avec un tableau
- File avec une liste chaînée
- Exercice
- Histogramme et Camembert

Pile

Un pile est une structure de données qui permet :

- d'ajouter un élément à son sommet (empiler).
- d'ôter l'élément à son sommet (dépiler)

Le principe d'une pile est : dernier entré, premier sorti (LIFO : Last In First Out).

Pile

Les opérations sur une pile sont :

- empiler (e, p) ajoute l'élément e au sommet de p.
- depiler (p) qui ôte l'élément au sommet de p.
- sommet (p) qui renvoie l'élément au sommet de la pile p.

p=[1,3,9,5,4].
sommet(p) renvoie 4.
dépiler (p) donne p=[1,3,9,5].
empiler (6, p) donne p=[1,3,9,5,6].

Pile

Exercice : Ecrire les fonctions empiler, depiler et sommet avec un tableau.

- empiler (e, p) ajoute l'élément e au sommet de p.
- depiler (p) qui ôte l'élément au sommet de p.
- sommet (p) qui renvoie l'élément au sommet de la pile p.

Pile avec un tableau

```
def empiler (e, p):
    p.append (e)

def depiler (p):
    p.pop ()

def sommet (p):
    return p [len (p) - 1]
```

Pile avec un tableau

```
p = []
empiler (1, p)
empiler (2, p)
depiler (p)
print (sommet (p))
```

Pile

Exercice : Ecrire les fonctions empiler, depiler et sommet avec une liste chaînée.

- empiler (e, p) ajoute l'élément e au sommet de p.
- depiler (p) qui ôte l'élément au sommet de p.
- sommet (p) qui renvoie l'élément au sommet de la pile p.

Pile avec une liste chaînée

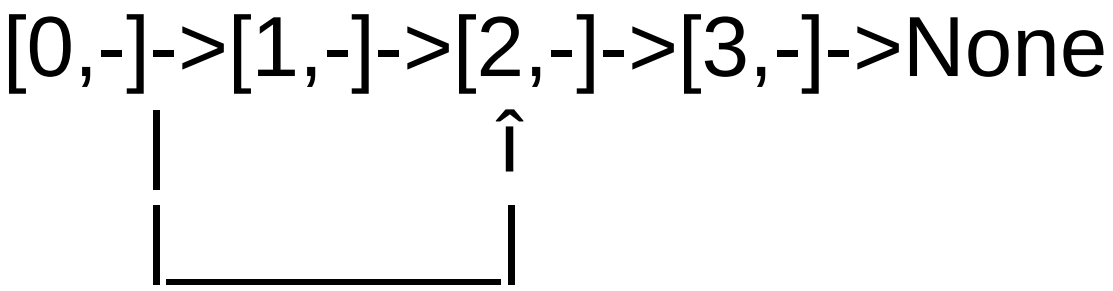
```
def empiler (e, p):
    suite = p.suiv
    p.suiv = Cellule (e)
    p.suiv.suiv = suite
```

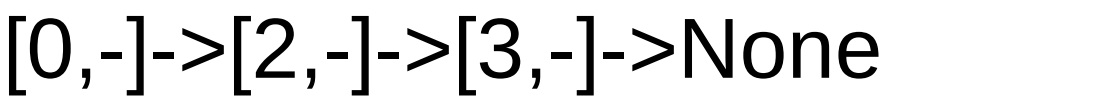
[-,-]->[1,-]->[2,-]->[3,-]->None

[-,-]->[e,-]->[1,-]->[2,-]->[3,-]->None

Pile avec une liste chaînée

```
def depiler (p):
    if p.suiv == None:
        return False
    else:
        suite = p.suiv.suiv
        p.suiv = suite
    return True
```





Pile avec une liste chaînée

```
def sommet (p):
    return p.suiv.entier

p = Cellule (0)
empiler (1, p)
empiler (2, p)
depiler (p)
print (sommet (p))
```

Pile pour calculer une expression

On peut représenter les expressions en notation polonaise inversée :

(4 + 3) * 5

Devient alors :

4 3 + 5 *

Pile pour calculer une expression

Il est plus simple de calculer une expression en notation polonaise inversée qu'en notation normale.

Quand on rencontre une valeur on l'empile.

Quand on rencontre un opérateur on dépile deux éléments, on fait l'opération et on empile le résultat.

Pile pour calculer une expression

Exemple de calcul de 4 3 + 5 * :

4	empiler (4, p)	p = [4]
3	empiler (3, p)	p = [4, 3]
+	dépiler deux fois	
	puis additionner	p = [7]
5	empiler (5, p)	p = [7, 5]
*	dépiler deux fois	
	puis multiplier	p = [35]

Pile pour calculer une expression

```
exp = [4, 3, '+', 5, '*']
p = []
for i in range (len (exp)):
    if (exp [i] == '+'):
        a = sommet (p)
        depiler (p)
        b = sommet (p)
        depiler (p)
        empiler (a + b, p)
```

Pile pour calculer une expression

```
    elif (exp [i] == '*'):
        a = sommet (p)
        depiler (p)
        b = sommet (p)
        depiler (p)
        empiler (a * b, p)
    else:
        empiler (exp [i], p)
print (sommet (p))
```

File

Une file est une structure de données dans laquelle l'insertion d'un élément ne peut se faire qu'en queue et la suppression ne peut se faire qu'en tête.

Le principe d'une file est : premier entré, premier sorti (FIFO : First In First Out).

File

Les opérations sur une file sont :

- enfiler (e, f) qui ajoute l'élément e en queue de la file f.
- défiler (f) qui ôte l'élément de la tête de la file f.
- tete (f) qui renvoie l'élément en tête de la file f.

File

Par exemple si on a la file f = [1,2,3,4,5].

enfiler (6, f) donne f = [1,2,3,4,5,6].

défiler (f) donne f = [2,3,4,5,6].

tete (f) donne alors 2.

File

Ecrire les opérations suivantes sur les files avec un tableau puis avec une liste chaînée :

- enfiler (e, f) qui ajoute l'élément e en queue de la file f.
- défiler (f) qui ôte l'élément de la tête de la file f.
- tete (f) qui renvoie l'élément en tête de la file f.

File avec une liste python

```
def enfiler (e, f):
    f.append (e)

def defiler (f):
    for i in range (len (f) - 1):
        f [i] = f [i + 1]
    f.pop ()

def defiler (f):
    f = f [1:]

def tete (f):
    return f [0]
```

File avec une liste chaînée

```
class Cellule (object):
    def __init__(self, entier):
        self.entier = entier
        self.suiv = None

class File (object):
    def __init__(self):
        self.tete = None
        self.queue = None
```

Ecrire enfiler (e,f)

File avec une liste chaînée

```
def enfiler (e, f):
    p = Cellule (e)
    if (f.queue == None):
        f.queue = p
        f.tete = p
    else:
        f.queue.suiv = p
        f.queue = p
```

File avec une liste chaînée

```
def defiler (f):
    if f.tete != None:
        f.tete = f.tete.suiv
        if (f.tete == None):
            f.queue = None
        return True
    return False
```

File avec une liste chaînée

```
def tete (f):
    if f.tete != None:
        return f.tete.entier
    return -1

def affiche (f):
    p = f.tete
    while (p != None):
        print (p.entier)
        p = p.suiv
```

File avec une liste chaînée

```
f = File ()
enfiler (1, f)
enfiler (2, f)
enfiler (3, f)
defiler (f)
affiche (f)
```

Éxecution :

```
2
3
```

Pile et liste chaînée

On cherche une fonction qui prend en entrée une pile d'entiers et qui fournit en sortie une liste chaînée qui ne contient que les nombres multiples de 3 contenus dans la pile.

Écrire une fonction qui donne en sortie les éléments en ordre inverse d'apparition de la pile.

Par exemple, [1,3,5,9,4,2,6,8] qui a 8 pour sommet de pile donne la liste [6,-]->[9,-]->[3,-] ->None.

Pile et liste chaînée

```
p = [1,3,5,9,4,2,6,8]
entete = Cellule (0)
courant = entete
while (len (p) > 0):
    e = sommet (p)
    depiler (p)
    if (e % 3 == 0):
        courant.suiv = Cellule (e)
        courant = courant.suiv
courant = entete.suiv
while courant != None:
    print (courant.entier)
    courant = courant.suiv
```

Pile et liste chaînée

Écrire une autre fonction qui donne cette fois les éléments dans le même ordre que la pile.

Par exemple, [1,3,5,9,4,2,6,8] qui a 8 pour sommet de pile donne la liste [3,-]->[9,-]->[6,-]->None.

Pile et liste chaînée

```
p = [1,3,5,9,4,2,6,8]
entete = Cellule (0)
while (len (p) > 0):
    e = sommet (p)
    depiler (p)
    if (e % 3 == 0):
        s = entete.suiv
        entete.suiv = Cellule (e)
        entete.suiv.suiv = s
courant = entete.suiv
while courant != None:
    print (courant.entier)
    courant = courant.suiv
```

Types de données abstrait

- TDA Ensemble
- TDA Liste
- Utilisation du TDA Liste
- Mise en oeuvre du TDA Liste par un tableau
- Mise en oeuvre du TDA Liste par une liste chaînée
- Comparaison des mises en œuvre
- Le TDA Pile
- Le TDA File

TDA Ensemble

- Définition : Un type de données abstrait (TDA) se dénit par un type de données et par les opérations qui lui sont associées.
- Exemple : Pour le TDA ensemble d'entiers on note les opérations :

  - UNION (A,B)

  - INTERSECTION(A,B)

  - DIFFERENCE (A,B)

TDA Ensemble

- Les types élémentaires sont ceux définis dans le langage de programmation :

  entiers, réels, caractères, chaînes de caractères.
- Les TDA sont une généralisation des types élémentaires de même que les procédures et fonctions sont des généralisations des opérations élémentaires +, -, /, *.
- Lorsqu'on écrit un algorithme à l'aide d'un TDA, l'algorithme ne change pas si on modifie la façon dont les procédures du TDA sont programmées.
- On peut traiter un TDA comme un type élémentaire associé à des primitives.

TDA Liste

- Si une liste est représentée par [a1,a2,a3,...,an], la longueur de la liste est n et l'élément en position i est ai.
- On utilise un type position qui peut être un entier ou un pointeur suivant que le TDA liste est mis en oeuvre par un tableau ou une liste chaînée.
- On suppose l'existence d'une position qui suit immédiatement le dernier élément de la liste.
- La fonction FIN(L) retourne la position suivant la nième position de la liste L ayant n éléments.
- La position FIN(L) varie avec la liste L.

TDA Liste

- Pour construire le TDA, on définit des opérations sur les objets de type liste appelées primitives.
- Pour toutes les primitives on notera :

  - L une liste d'éléments

  - x un élément

  - p une position
- La position peut être un entier ou un pointeur.

TDA Liste

- INSERER (x, p, L)

  - insère l'élément x à la position p dans la liste L.

  - si L= [a1,a2,...an] elle devient

    [a1,a2,...,ap-1,x,ap,...,an]

  - si p = FIN (L) alors la liste devient [a1,a2,…,an,x],
- FIN (L) est la position qui permet d'insérer en fin de liste.

TDA Liste

- LOCALISER (x, L) : position

  - retourne la position de x dans L.

  - si x apparaît plusieurs fois dans L, c'est la première position de x qui est renvoyée.

  - si x n'est pas dans L, c'est la position FIN (L) qui est renvoyée.

TDA Liste

- ACCEDER (p, L) : élément

  - retourne l'élément en position p dans L (sans le retirer).

  - l'opération est impossible si p = FIN (L) ou si la liste n'a pas de position p.

TDA Liste

- SUPPRIMER (p, L)

  - supprime l'élément en position p dans la liste L.

  - si la liste était L= [a1,a2,...,ap-1,ap,ap+1,...,an] elle devient [a1,a2,...,ap-1,ap+1,...,an].

  - l'opération est impossible si L n'a pas de position p ou si p = FIN (L). Dans ce cas la liste reste inchangée.

TDA Liste

- SUIVANT (p, L) : position

  - retourne la position suivant la position p dans la liste L.

  - si p est la dernière position de L, SUIVANT (p, L) retourne FIN (L).

  - l'opération est impossible si L n'a pas de position p ou si p = FIN (L).

TDA Liste

- PRECEDENT (p, L) : position

  - retourne la position précédent la position p dans la liste L.

  - l'opération est impossible si L n'a pas de position p ou si p est la première position de L.

TDA Liste

- PREMIER (L) : position

  - retourne la première position de L.

  - si L est vide, retourne FIN (L).

Utilisation du TDA Liste

- Sans connaître la structure de données utilisée pour représenter une liste, on peut écrire l'algorithme qui prend en entrée une liste L et qui élimine de L toutes les répétitions.
- L'algorithme Simplifier fonctionnera indépendamment de la manière dont les listes sont représentées.
- La complexité de Simplifier dépendra de la complexité des primitives qui elles mêmes dépendent de la mise en oeuvre retenue.

Utilisation du TDA Liste

```
def Simplifier (L):
    p = PREMIER (L)
    while (p != FIN (L)):
        q = SUIVANT (p, L)
        while (q != FIN (L)):
            if (ACCEDER (p, L) == ACCEDER (q, L)):
                SUPPRIMER (q, L)
            else:
                q = SUIVANT (q, L)
        p = SUIVANT (p, L)
```

Mise en oeuvre du TDA Liste par un tableau

- La classe Liste contient un enregistrement à deux champs :
  - un tableau de taille suffisante pour contenir les plus grandes listes à traiter.
  - un entier contenant la position du dernier élément de la liste.
- L'élément ai est placé dans la ième cellule du tableau.
  - Le type d'une position est un entier (l'indice dans le tableau).
  - la structure de données utilisée comprend un tableau et un entier :

```
class Liste (object):
    def __init__(self, entier):
        self.tableau = [0] * entier
        self.taille = entier
        self.sommet = 0
```

Mise en oeuvre du TDA Liste par un tableau

INSERER (x, p, L)

Pour insérer un élément à la position p :

1) Il faut déplacer les éléments situés aux positions p, p+1,..., L.sommet vers les positions p+1, p+2,...,L.sommet+1. Il faut faire attention à partir du dernier élément et à aller vers la position p, sinon on écrase les valeurs qui n'ont pas encore été déplacées.

2) Il faut alors insérer x en position p.

Mise en oeuvre du TDA Liste par un tableau

```
def INSERER (x, p, L):
    if L.sommet == L.taille + 1:
        print ("insertion impossible")
    else:
        for q in range (L.sommet - p):
            L.tableau [L.sommet - q] = L.tableau [L.sommet - q - 1]
        L.tableau [p] = x;
        L.sommet = L.sommet + 1
```

- Complexité en O(n) si on insère en tête de liste.

Mise en oeuvre du TDA Liste par un tableau

LOCALISER (x, L)

Consiste à parcourir séquentiellement le tableau à la recherche de l'élément x.

```
def LOCALISER (x, L):
    for p in range(L.sommet):
        if L.tableau [p] == x:
            return p
    return L.sommet
```

- Complexité en O (n) quand x n'est pas dans L.

Mise en oeuvre du TDA Liste par un tableau

SUPPRIMER (p, L)

- on recopie les éléments qui suivent p vers leur position précédente et on décrémente le sommet.

```
def SUPPRIMER (p, L):
    if (p < 0 or p >= L.sommet):
        print ("suppression impossible")
    else:
        for q in range (p, L.sommet - 1):
            L.tableau [q] = L.tableau [q + 1]
        L.sommet = L.sommet - 1
```

- Complexité en O (n) si suppression du premier élément.

Mise en oeuvre du TDA Liste par un tableau

```
def ACCEDER (p, L):
    return L.tableau [p]
def PREMIER (L):
    return 0
def FIN (L):
    return L.sommet
def SUIVANT (p, L):
    return p + 1
def AFFICHER (L):
    for q in range (L.sommet):
        print (L.tableau [q])
```

Mise en oeuvre du TDA Liste par une liste chaînée

- La position est de type pointeur. Il est plus commode de définir la position i comme le pointeur sur la cellule contenant le pointeur sur ai.
- La position 1 est un pointeur sur l'entête.
- La position FIN(L) est un pointeur sur la cellule contenant an.
- entête -> [0,-] -> [a1,-] -> [a2,-] -> ... -> [an-1,-] -> [an,-] -> NULL
- Position: 1 2 3 n FIN(L)
- Pour éviter de recalculer FIN (L) à chaque fois qu'on l'utilise on utilise un pointeur sur la dernière cellule dans la structure qui représente la liste.

```
class Liste (object):
    def __init__(self):
        self.entete = Cellule (0)
        self.dernier = self.entete
```

Mise en oeuvre du TDA Liste par une liste chaînée

INSERER (x, p, L)

- Pour insérer un élément à la position p :

  1) Il faut mémoriser la cellule suivante dans un pointeur temp.

  2) Il faut allouer une nouvelle cellule et lui affecter x.

  3) Il faut faire pointer la cellule suivante vers la nouvelle cellule

  4) Il faut faire pointer la nouvelle cellule vers la cellule mémorisée

- L -> 0[,-] -> [a1,-] -> [a2,-] -> [a3,-] -> None
  p temp
- L -> [0,-] -> [a1,-] -> [x,-] [a2,-] -> [a3,-] -> None
  p temp
- L -> [0,-] -> [a1,-] -> [x,-] -> [a2,-] -> [a3,-] -> None
  p temp

Mise en oeuvre du TDA Liste par une liste chaînée

```
def INSERER (x, p, L):
    if (p != None):
        temp = p.suiv
        p.suiv = Cellule (x)
        p.suiv.suiv = temp
        if temp == None:
            L.dernier = p.suiv
```

Mise en oeuvre du TDA Liste par une liste chaînée

```
def LOCALISER (x, L):
    p = L.entete
    while p.suiv != None:
        if p.suiv.entier == x:
            return p
        p = p.suiv
    return L.dernier
```

Mise en oeuvre du TDA Liste par une liste chaînée

- Exercice :

  Ecrire la primitive SUPPRIMER (p, L)

Mise en oeuvre du TDA Liste par une liste chaînée

```
def SUPPRIMER (p, L):
    if p != None and p.suiv != None:
        p.suiv = p.suiv.suiv
        if p.suiv == None:
            L.dernier = p
```

Mise en oeuvre du TDA Liste par une liste chaînée

```
def ACCEDER (p, L):
    return p.suiv.entier

def PREMIER (L):
    return L.entete

def FIN (L):
    return L.dernier

def SUIVANT (p, L):
    return p.suiv
```

Mise en oeuvre du TDA Liste par une liste chaînée

- Exercice :

  Ecrire la primitive AFFICHER (L) à l'aide des primitives déjà définies

Mise en oeuvre du TDA Liste par une liste chaînée

```
def AFFICHER (L):
    q = PREMIER (L)
    while q != FIN (L):
        print (ACCEDER (q,L))
        q = SUIVANT (q, L)
```

Comparaison des mises en oeuvre

Primitive	tableau	liste chaînée
INSERER(x,p,L)	O(n)	O(1)
LOCALISER(X,L)	O(n)	O(n)
ACCEDER(p,L)	O(1)	O(1)
SUPPRIMER(p,L)	O(n)	O(1)
SUIVANT(p,L)	O(1)	O(1)
PRECEDENT(p,L)	O(1)	O(n)
PREMIER(p,L)	O(1)	O(1)

Le TDA Pile

- On peut définir le TDA Pile à l'aide du TDA Liste :

  SOMMET(P) = ACCEDER(PREMIER(P))

  DEPILER(P) = SUPPRIMER(PREMIER(P), P)

  EMPILER(x,P) = INSERER(x, PREMIER(P), P)

- En utilisant l'implémentation sous forme de listes chaînées, toutes les opérations sont en O(1).

Le TDA File

- Exercice :

  Définir le TDA File à l'aide du TDA Liste

Le TDA File

- TDA File à l'aide du TDA Liste :

  ENFILER(x, F) = INSERER (x, FIN (F), F)

  DEFILER (F) = SUPPRIMER (PREMIER (F), F)

  TETE (F) = ACCEDER (PREMIER (F))
- ENFILER est en O(n) si on n'a pas de pointeur sur le dernier élément, sinon il est en O(1)

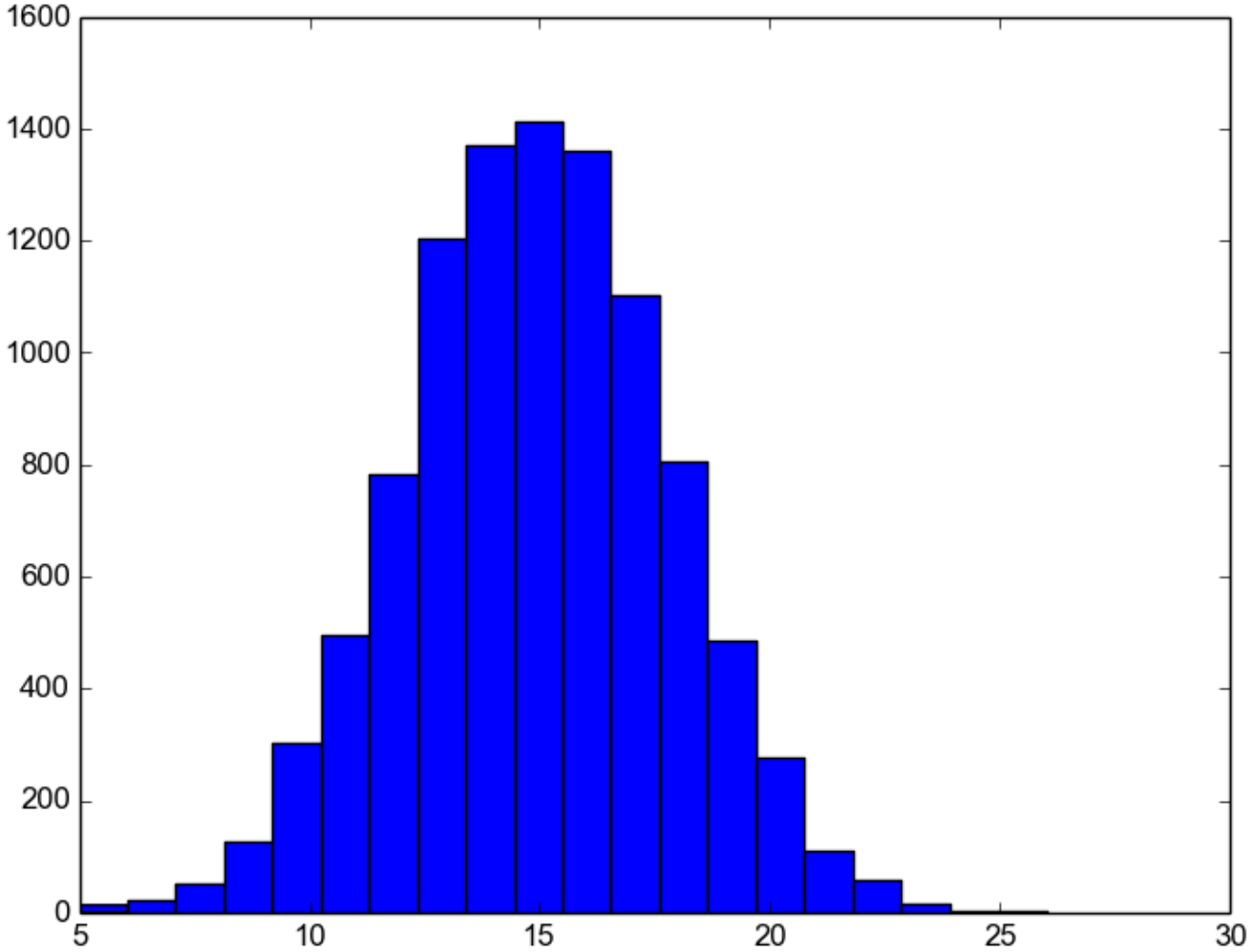
Histogramme

```
import matplotlib.pyplot as plt
import random

l = []
for x in range (100000):
    e = 0
    for i in range (30):
        e = e + random.randint (0, 1)
    l.append (e)

plt.hist(l, 20)
plt.show ()
```

Histogramme



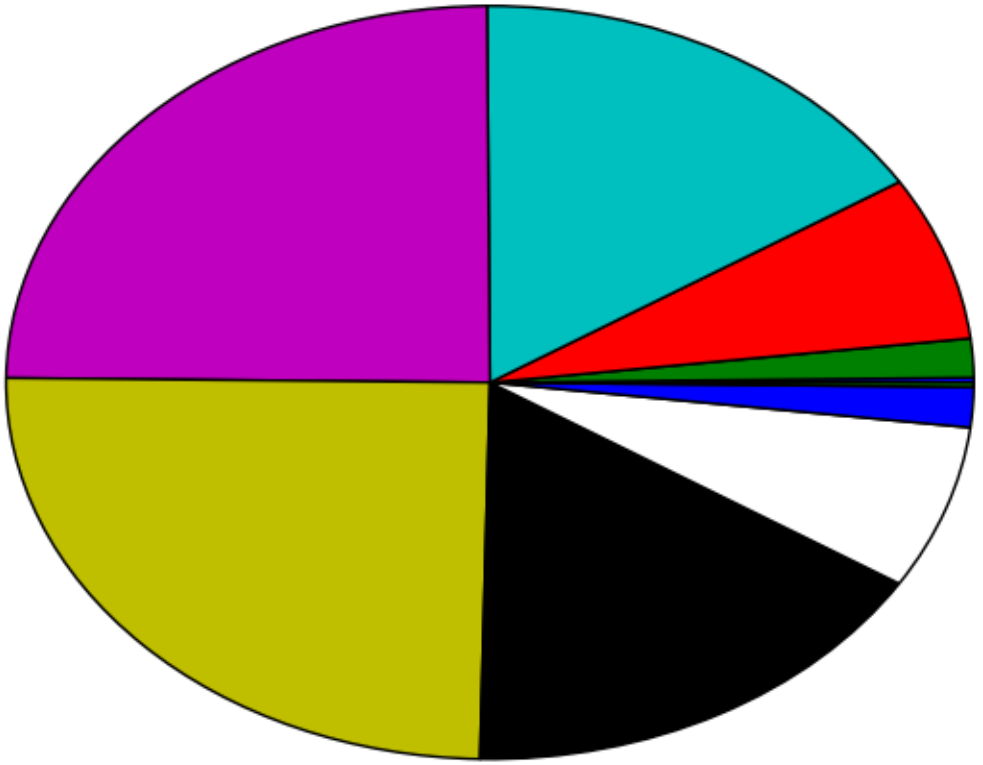
Camembert

```
import matplotlib.pyplot as plt
import random

l = [0] * 10
for x in range (100000):
    e = 0
    for i in range (9):
        e = e + random.randint (0, 1)
    l [e] = l [e] + 1

plt.pie (l)
plt.show ()
```

Camembert

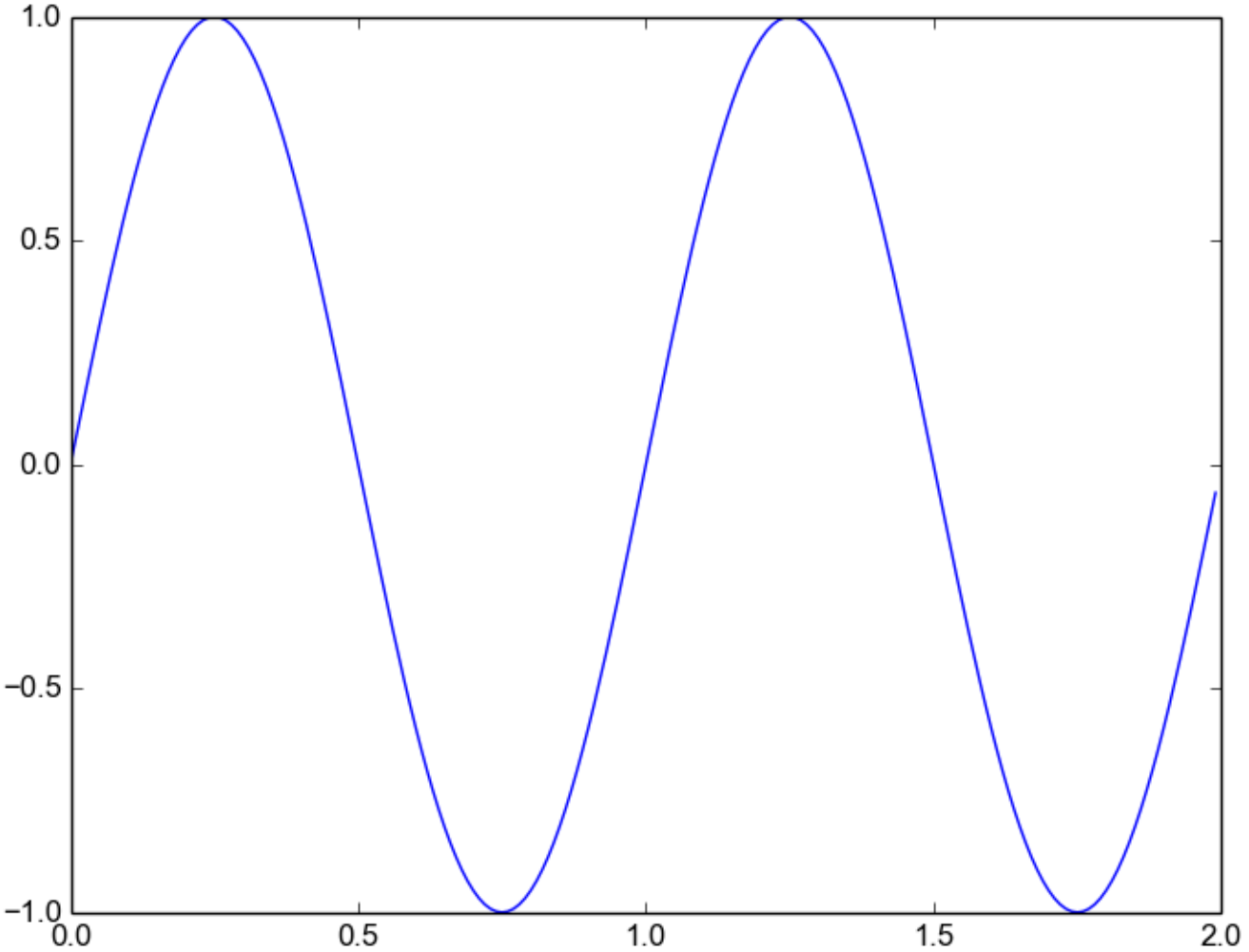


Sinus

```
import matplotlib.pyplot as plt
import math

t = []
s = []
for i in range (200):
    t.append (i / 100.0)
    s.append (math.sin (2 * math.pi * i / 100.0))
plt.plot (t, s)
plt.show ()
```

Sinus



Partiel 2013

Somme de deux éléments d'un tableau
Comptage d'éléments d'un tableau
Comptage d'éléments d'une liste chaînée
Indice d'un élément dans une liste chaînée
Produit des éléments d'une liste chaînée
Transformer un tableau en liste chaînée
Tri à bulle d'une liste chaînée
Opérations sur les produits
Plus grand budget
Liste triée de produits

Somme de deux éléments d'un tableau

Ecrire un algorithme qui trouve si un nombre donné n est la somme de deux nombres éléments d'un tableau T.

Par exemple pour n = 10 et T = [1, 3, 5, 7] l'algorithme renvoie True, alors que si T=[1, 5, 8] l'algorithme renvoie False.

Somme de deux éléments d'un tableau

```
T = [1, 3, 5, 7]
somme = False
n = 10
for i in range (len (T)):
    for j in range (i + 1, len (T)):
        if T [i] + T [j] == n:
            somme = True
```

Comptage d'éléments d'un tableau

On souhaite compter le nombre d'éléments d'un tableau T1 qui ne sont pas dans un tableau T2.

Si par exemple T1 = [1, 2, 3, 4] et T2 = [2, 4], la réponse est 2 puisque 1 et 3 ne sont pas dans T2.

Comptage d'éléments d'un tableau

```
T1 = [1, 2, 3, 4]
T2 = [2, 4]
n = 0
for i in range (len (T1)):
    element = False
    for j in range (len (T2)):
        if T1 [i] == T2 [j]:
            element = True
    if element == False:
        n = n + 1
```

Comptage d'éléments d'une liste chaînée

On utilise la classe suivante pour les listes chaînées :

```
class Cellule (object):
    def __init__(self, entier):
        self.entier = entier
        self.suiv = None
```

On souhaite compter le nombre d'éléments d'une liste chaînée qui ne sont pas dans une deuxième liste chaînée.

Comptage d'éléments d'une liste chaînée

```
n = 0
p1 = l1.suiv
while p1 != None:
    element = False
    p2 = l2.suiv
    while p2 != None:
        if p1.entier == p2.entier:
            element = True
        p2 = p2.suiv
    if element == False:
        n = n + 1
    p1 = p1.suiv
```

Indice d'un élément dans une liste chaînée

Ecrire un algorithme qui calcule l'indice d'un élément e dans une liste chaînée l.

Par exemple pour e = 4 et l = [1, 7, 4, 10, 3] l'algorithme renvoie 3.

Si l'élément ne fait pas partie de la liste l'indice vaut -1.

Indice d'un élément dans une liste chaînée

```
e = 2
indice = -1
i = 1
p = l.suiv
while p != None:
    if p.entier == e:
        indice = i
    p = p.suiv
    i = i + 1
```

Produit des éléments d'une liste chaînée

Ecrire un algorithme qui calcule le produit des éléments d'une liste chaînée.

Par exemple pour l = [2, 5, 10], l'algorithme calcule 100.

Produit des éléments d'une liste chaînée

```
produit = 1
p = l.suiv
while p != None:
    produit = produit * p.entier
    p = p.suiv
```

Transformer un tableau en liste chaînée

On souhaite construire une liste chaînée à partir d'un tableau.

Ecrire l'algorithme correspondant.

Transformer un tableau en liste chaînée

```
l = Cellule (0)
p = l
for i in range (len (T)):
    p.suiv = Cellule (T [i])
    p = p.suiv
```

Tri à bulle d'une liste chaînée

Le tri à bulle consiste à parcourir une liste et à échanger deux valeurs qui se suivent si elles ne sont pas bien ordonnées.

On fait ce parcours tant qu'au moins deux valeurs ont été échangées.

Ecrire l'algorithme qui effectue le tri à bulle d'une liste chaînée.

Tri à bulle d'une liste chaînée

```
echange = True
while echange == True:
    echange = False
    p = l.suiv
    if p != None:
        while p.suiv != None:
            if p.entier > p.suiv.entier:
                tmp = p.entier
                p.entier = p.suiv.entier
                p.suiv.entier = tmp
                echange = True
            p = p.suiv
```

Opérations sur les produits

On représente une liste de produits vendus en magasin.

Chaque produit a un identifiant entier.

On stocke de plus dans la structure de données son prix comme un entier, son nombre d'exemplaires comme un entier et sa date de péremption comme un entier.

La structure de données utilisée est la suivante :

Opérations sur les produits

La structure de données utilisée est la suivante :

```
class Produit (object):
    def __init__(self, identifiant, prix, nombre, date):
        self.identifiant = identifiant
        self.prix = prix
        self.nombre = nombre
        self.date = date
        self.suiv = None
```

Plus grand budget

Ecrire un algorithme qui donne l'identifiant du produit qui a le budget le plus grand.

Le budget d'un produit est le produit du nombre d'exemplaires par le prix.

Par exemple pour l = [1, 50, 2, 20, -] → [2, 100, 10, 30, -] → [3, 20, 1000, 15, -] → None, l'algorithme renverra 3.

Plus grand budget

```
p = l.suiv
id = p.identifiant
b = p.prix * p.nombre
while p != None:
    if p.prix * p.nombre > b:
        id = p.identifiant
        b = p.prix * p.nombre
    p = p.suiv
```

Liste triée de produits

En supposant que la liste l est triée par budget croissant, ajouter un produit p à la liste de façon à ce qu'elle reste triée.

Liste triée de produits

```
b = p.prix * p.nombre
p1 = l
nonInsere = True
if p1.suiv == None:
    p1.suiv = p
else:
```

Liste triée de produits

```
while p1.suiv != None and nonInsere:
    if p1.suiv.prix * p1.suiv.nombre > b:
        p.suiv = p1.suiv
        p1.suiv = p
        nonInsere = False
    p1 = p1.suiv
if nonInsere == True:
    p1.suiv = p
    p.suiv = None
```

Partiel 2014

Nombre d'éléments pairs et impairs d'un tableau
Décalage d'un tableau
Echange dans une liste chaînée
Fusion de deux listes chaînées
Doublons
Insertion tous les trois éléments
Insertion dans une liste triée
Séparation en deux listes

Nombre d'éléments pairs et impairs d'un tableau

- Vérifier qu'un tableau contient autant de nombres pairs que de nombres impairs.

- Par exemple la propriété est vérifiée pour

  T = [1, 7, 2, 4] mais pas pour T1 = [2, 5, 6, 8].

Nombre d'éléments pairs et impairs d'un tableau

```
T = [1, 7, 2, 4]
pairs = 0
impairs = 0
for i in range (len(T)):
    if T [i] % 2 == 0:
        pairs = pairs + 1
    else:
        impairs = impairs + 1
if pairs == impairs:
    p = True
else:
    p = False
print (p)
```

Décalage d'un tableau

- Effectuer un décalage d'un tableau de façon à ce que chaque élément se retrouve à deux indices après son indice initial.
- Les deux derniers éléments se retrouvant aux deux premiers indices.
- Par exemple le tableau T = [1, 7, 2, 4] devient T = [2, 4, 1, 7].

Décalage d'un tableau

```
T = [1, 7, 2, 4]
tmp1 = T [len(T) - 1]
tmp2 = T [len(T) - 2]
for j in range (1, len(T) - 1):
    T [len(T) - j] = T [len(T) - j - 2]
T [0] = tmp2
T [1] = tmp1
print (T)
```

Echange dans une liste chaînée

- Echanger deux à deux les éléments qui se suivent d'une liste chaînée pour tous les indices pairs.

- Par exemple la liste [1, 2, 3, 4, 5, 6] devient la liste [2, 1, 4, 3, 6, 5].

Echange dans une liste chaînée

```
p = l.suiv
while p != None:
    tmp = p.entier
    p.entier = p.suiv.entier
    p.suiv.entier = tmp
    p = p.suiv.suiv
```

Fusion de deux listes chaînées

- Fusionner deux listes chaînées de façon à ce que la liste résultat contienne deux éléments de la première liste suivis par deux éléments de la seconde liste et ainsi de suite.
- On suppose que les nombres d'éléments des deux listes sont pairs.
- Par exemple la fusion de la liste L1 = [1, 2, 3, 4] et de L2 = [5, 6, 7, 8] devient L1 = [1, 2, 5, 6, 3, 4, 7, 8].

Fusion de deux listes chaînées

```
p = l1.suiv
p1 = l2.suiv
while p1 != None:
    p = p.suiv
    tmp = p.suiv
    p.suiv = p1
    p1 = p1.suiv
    tmp1 = p1.suiv
    p1.suiv = tmp
    p = tmp
    p1 = tmp1
```

Doublons

- Oter tous les doublons d'une liste chaînée.

- Par exemple la liste L1 = [1, 2, 1, 3, 2] devient L1 = [1, 2, 3].

Doublons

```
p = l.suiv
while p != None:
    p1 = p
    while p1.suiv != None:
        if p1.suiv.entier == p.entier:
            p1.suiv = p1.suiv.suiv
        else if p1.suiv != None:
            p1 = p1.suiv
    p = p.suiv
```

Insertion tous les trois éléments

- Insérer tous les trois éléments d'une liste chaînée la somme de ces trois éléments.

- On suppose que le nombre d'éléments de la liste est un multiple de trois.

- Par exemple la liste L1 = [1, 2, 3, 4, 5, 6]

  devient L1 = [1, 2, 3, 6, 4, 5, 6, 15].

Insertion tous les trois éléments

```
p = l.suiv
while p != None:
    s = p.entier + p.suiv.entier + p.suiv.suiv.entier
    tmp = p.suiv.suiv.suiv
    p.suiv.suiv.suiv = Cellule (s)
    p = p.suiv.suiv.suiv
    p.suiv = tmp
    p = p.suiv
```

Opérations sur les articles

- On représente une liste d'articles d'un catalogue.
- Chaque article a un identifiant entier et un prix réél.
- La classe utilisée est la suivante :

```
class Article (object):
    def __init__(self, identifiant, prix):
        self.identifiant = identifiant
        self.prix = prix
        self.suiv = None
```

Insertion dans une liste triée

- Insérer un nouvel article dans une liste d'articles triée par identifiant, du plus petit identifiant au plus grand identifiant, de façon à ce que la liste reste triée.

Insertion dans une liste triée

```
a = Article (3, 30.0)
p = l.suiv
while p.suiv != None and p.suiv.identifiant < a.identifiant:
    p = p.suiv
tmp = p.suiv
p.suiv = a
a.suiv = tmp
```

Séparation en deux listes

- Séparer une liste d'articles en deux listes de tailles égales à un près de façon à ce que tous les prix de la première liste soient plus petits que tous les prix de la deuxième liste.

Séparation en deux listes

```
b = True
while b == True:
    b = False
    p = l.suiv
    nb = 1
    while p.suiv != None:
        nb = nb + 1
        if p.prix > p.suiv.prix:
            b = True
            tmp = p.prix
            p.prix = p.suiv.prix
            p.suiv.prix = tmp
            tmp = p.identifiant
            p.identifiant = p.suiv.identifiant
            p.suiv.identifiant = tmp
        p = p.suiv
```

Séparation en deux listes

```
p = l
n = 0
while n < nb / 2:
    n = n + 1
    p = p.suiv
l1 = Article (0, 0.0)
l1.suiv = p.suiv
p.suiv = None
```

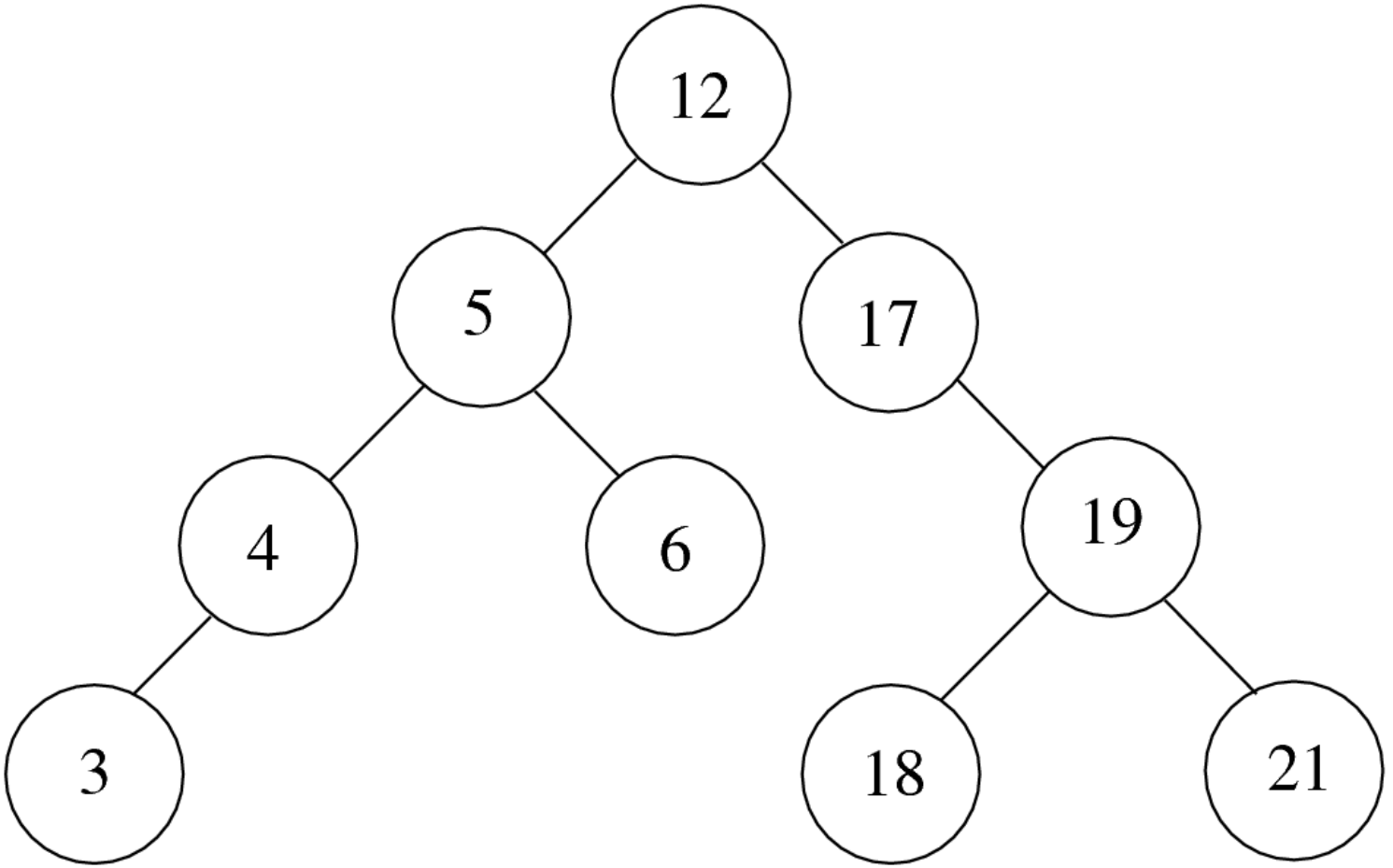
Les arbres

Définitions
Ordre sur les noeuds d'un arbre
Les arbres binaires
Parcours des arbres
Mise en œuvre des arbres

Définitions

- Un arbre est un ensemble d'éléments appelés noeuds reliés par des arcs.
- Il existe une relation de parenté entre les noeuds.
- Un noeud père est situé au dessus de ses noeuds fils.
- Un père est relié à ses fils par des arcs.
- Le noeud le plus haut placé dont dérivent tous les autres noeuds est appelé la racine.

Définitions



Définitions

- La racine est le noeud 12.
- Le père est placé au dessus des fils.
- Le segment reliant un fils à son père est un arc.

Définitions

- Définition récursive d'un arbre :
- Un noeud unique est un arbre. Dans ce cas il est aussi la racine de l'arbre.
- Si n est un noeud et A1, A2,..., Ak sont des arbres de racines respectives n1, n2, ..., nk alors on peut construire un arbre en associant comme père unique aux noeuds n1, n2, ..., nk le noeud n. Dans ce nouvel arbre, n est la racine et A1, A2, ..., Ak sont les sous arbres de cette racine.
- Les noeuds n1, n2, ..., nk sont appelés les fils du noeud n.

Définitions

- Définition : si n1, n2, ..., nj est une suite de noeuds telle que ni est le père de ni+1 pour i allant de 1 à j-1, alors cette suite est appelée chemin entre le nœud n1 et le noeud nj.
- La longueur du chemin est le nombre de noeuds - 1 (= nombre d'arcs).

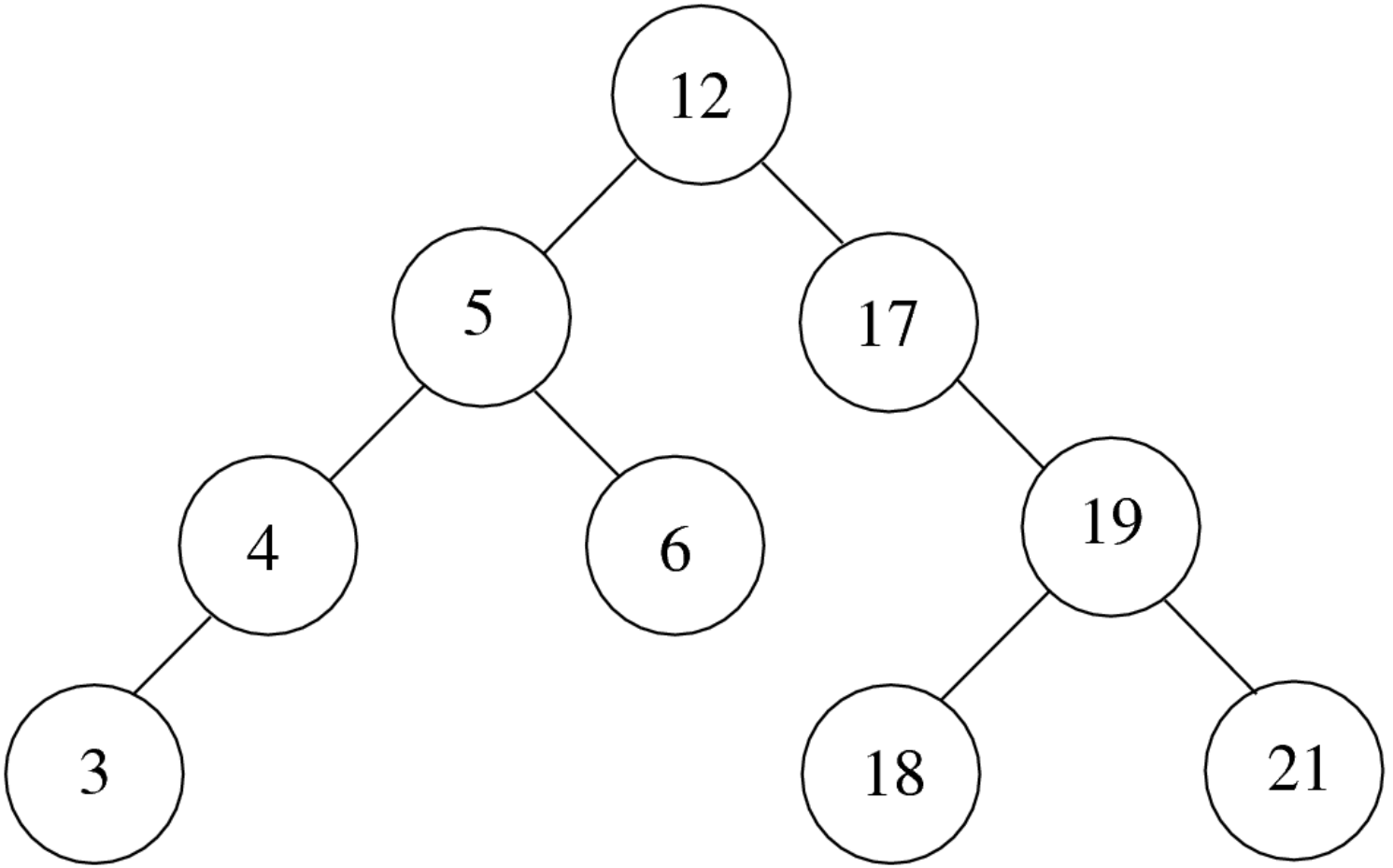
Définitions

- S'il existe un chemin entre les noeuds a et b alors a est dit ascendant de b (ou ancêtre), b est dit descendant de a.
- Tout noeud est un de ses descendants et un de ses ascendants.
- Tout ascendant ou descendant d'un noeud différent de lui même est un ascendant ou descendant propre.
- Dans un arbre seule la racine n'a pas d'ascendant propre.

Définitions

- Un noeud sans descendant propre est appelé une feuille.
- Des noeuds ayant le même père sont appelés frères.
- Un noeud qui n'est pas une feuille est un noeud interne.
- Il est parfois utile de nommer l'arbre sans noeuds noté A0.

Définitions



Définitions

- 12, 17, 19 est un chemin de longueur 2.
- 12, 17, 6 n'est pas un chemin.
- Les descendants propres de 5 sont 4, 3, 6.
- Les fils de 5 sont 4 et 6.
- Le père de 5 est 12.
- Les ascendants propres de 4 sont 5 et 12.
- Les feuilles de l'arbre sont 3, 6, 18, 21.
- 18 et 21 sont frères.

Définitions

- Définition : la hauteur d'un noeud dans un arbre est la longueur du plus long chemin que l'on peut mener entre ce noeud et une feuille de l'arbre.
- La hauteur d'un arbre est la hauteur de sa racine.
- La profondeur d'un noeud est la longueur du chemin entre la racine et ce noeud.
- Dans l'exemple :
- La hauteur de 5 est 2.
- La profondeur de 5 est 1.
- La hauteur de l'arbre est 3.

Ordre sur les noeuds d'un arbre

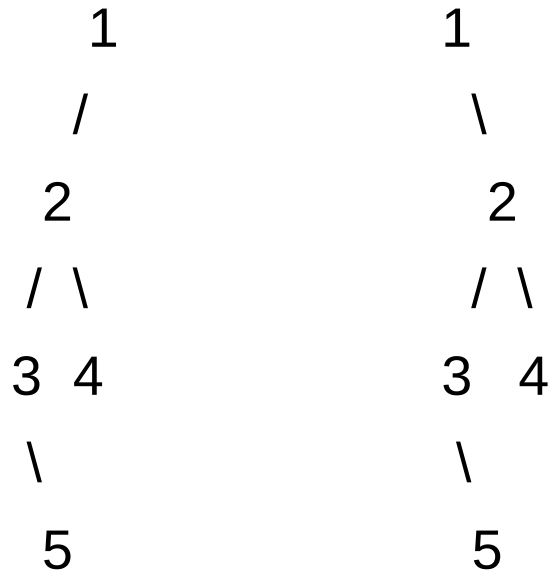
- Les fils d'un noeud sont habituellement ordonnés de gauche à droite.
- Les deux arbres ci dessous sont différents :

```
   a        a
  / \      / \
 b c      c  b
```

- L'ordre gauche-droite peut être prolongé pour comparer deux noeuds qui ne sont ni ascendants ni descendants.
- Si b et c sont deux frères, que b est à gauche de c, tout descendant de b est à gauche de tout descendant de c.
- Dans l'exemple :
- 5 est à gauche de 17.
- 4 est à gauche de 19.
- 6 est à gauche de 18.
- 17 n'est ni à gauche ni à droite de 18.

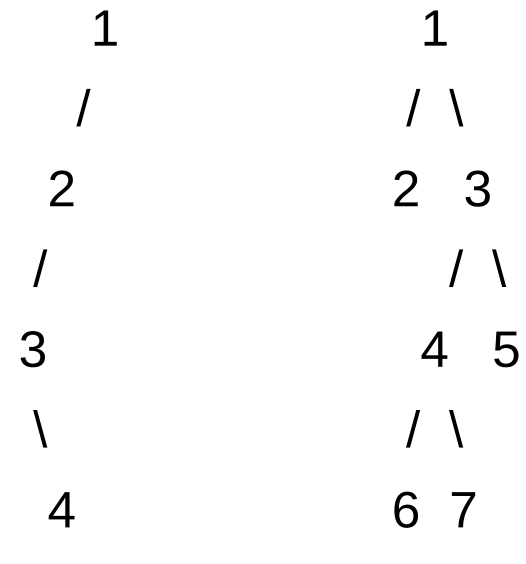
Les arbres binaires

- Définition : Un arbre binaire est un arbre dont chaque noeud a au maximum deux fils.
- Les arbres binaires distinguent le fils gauche du fils droit.
- Ces deux arbres binaires sont différents :

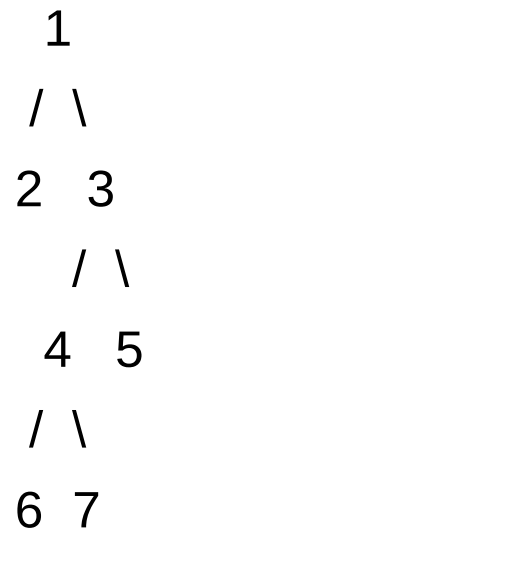


Les arbres binaires

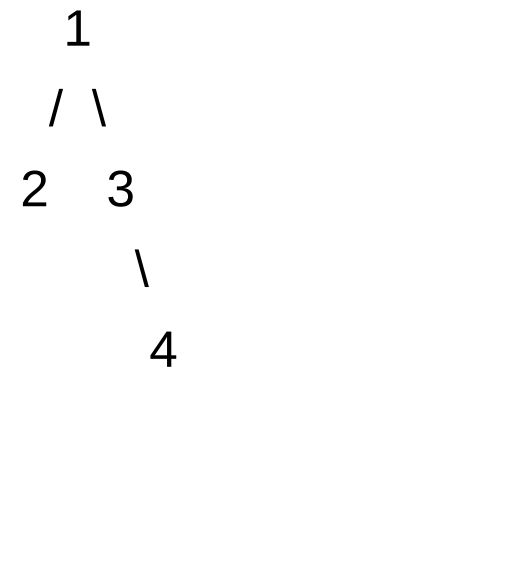
- Dans un arbre binaire dégénéré ou filiforme chaque noeud a un seul fils
- Un arbre binaire complet est un arbre binaire dont chaque noeud a deux fils ou est une feuille.
- Exemples :



filiforme



complet



pas complet

Les arbres binaires

- Théorème 1 : pour un arbre binaire possédant n noeuds et de hauteur h on a :

  $\log_2 (n) \leq h \leq n - 1$

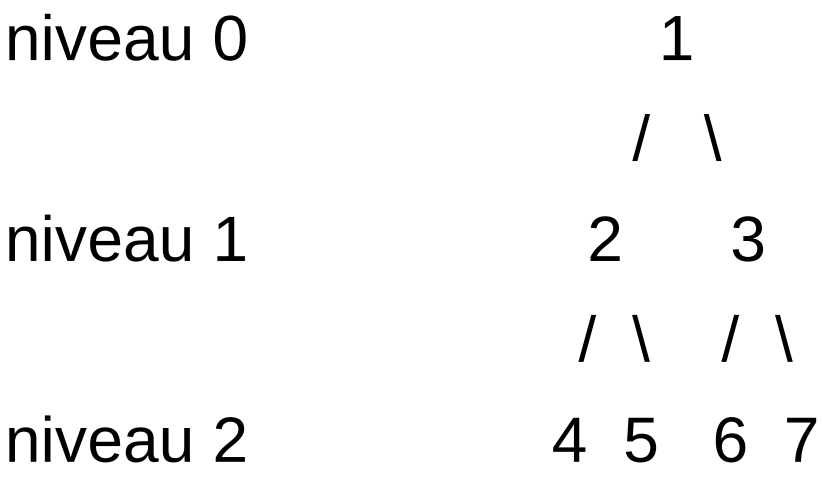
- Principe de la preuve : on étudie à hauteur fixée le nombre maximal et minimal de noeuds d'un arbre binaire.

Les arbres binaires

- 1) borne max
- Les arbres binaires qui ont le moins de noeuds sont les arbres dégénérés.
- Soit nd le nombre de noeuds d'un arbre dégénéré de hauteur h.
- nd = nombre d'arcs + 1 = h + 1 => h = nd - 1
- Tout arbre binaire de hauteur h a plus de noeuds qu'un arbre dégénéré de hauteur h.
- Donc tous les arbres binaires de hauteur h ont une hauteur plus petite que leur nombre de noeuds - 1.

Les arbres binaires

- 2) borne min
- Les arbres binaires de hauteur h ayant le plus de noeuds sont ceux pour lesquels tous les noeuds ont deux fils sauf ceux du dernier niveau qui sont des feuilles.
- Exemple :



Les arbres binaires

- $nmax = 1 + 2 + 2*2 + 2*2*2 + ... = \Sigma_{i <= h}\ 2^i$
- On a une suite géométrique de raison q=2 et de premier terme U0 = 1 .
- Rappel : si $Ui = U0\ q^i$ alors :

  $\Sigma_{i <= h}\ Ui = U0 \times (q^{n+1} - 1) / (q - 1)$
- donc $nmax = 1 \times (2^{h+1} - 1) / (2-1) = 2^{h+1} - 1$

Les arbres binaires

- Pour un arbre ayant n noeuds on a :

  $n \leq nmax = 2^{h+1} - 1$.

  $\Rightarrow n < 2^{h+1}$

  $\Rightarrow 2^{\log_2 (n)} < 2^{h+1}$

  $\Rightarrow \log_2 (n) < h + 1$

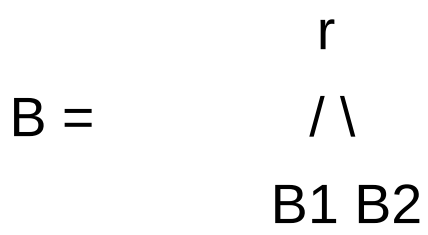
  $\Rightarrow h \geq$ Partie entiere de $\log_2 (n)$

Les arbres binaires

- Théorème 2 : un arbre binaire complet ayant n noeuds internes a n+1 feuilles.
- La propriété est vraie pour l'arbre ayant 0 noeuds internes et 1 feuille.
- Supposons la propriété vraie pour tous les arbres binaires complets ayant moins de n noeuds internes.

Les arbres binaires

- Soit B un arbre binaire complet :



- Les deux sous arbres B1 et B2 sont des arbres binaires complets.
- Soit n1 le nombre de noeuds internes de B1.
- Soit n2 le nombre de noeuds internes de B2.
- Soit n le nombre de noeuds internes de B.
- On a n = n1 + n2 + 1
- le nombre de feuilles de B = nombre de feuilles de B1 + nombre de feuilles de B2 = n1 + 1 + n2 + 1 = n + 1

Les arbres binaires

- Théorème 3 : Il existe une bijection entre l'ensemble Bn des arbres binaires ayant n noeuds et l'ensemble BCn des arbres binaires complets ayant 2n+1 noeuds.
- Si B est un arbre binaire complet ayant 2n+1 noeuds, alors l'arbre B' obtenu en enlevant toutes les feuilles de B a n noeuds.
- C'est une conséquence du théorème 2 puisque n noeuds internes correspondent à n+1 feuilles.

Les arbres binaires

- L'application inverse consiste à compléter l'arbre binaire B' de sorte que chaque noeud de B' ait deux fils.
- Tous les noeuds de B' deviennent alors internes, et les feuilles ajoutées ne sont pas des noeuds internes, l'arbre résultant a donc n nœuds internes, il est complet, il a donc n+1 feuilles, d'où 2n+1 noeuds.
- Illustration sur un arbre et son transformé.
- Comptage du nombre de noeuds.

Les arbres binaires

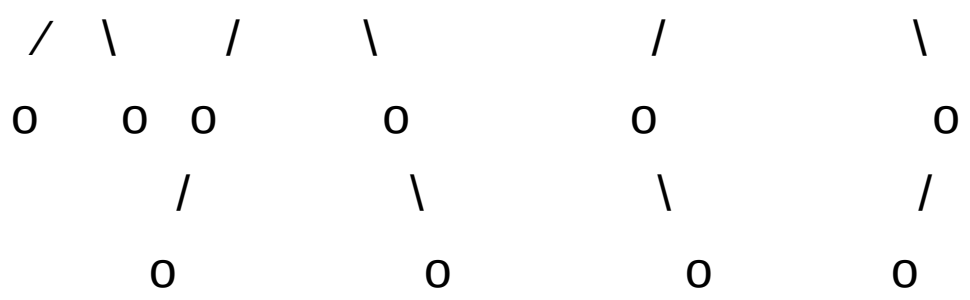
- Pour étudier la complexité en moyenne d'algorithmes sur les arbres binaires on doit connaître le nombre bn d'arbres binaires de taille n.

n=0 A0 b0 = 1

n=1 o b1 = 1

n=2 o et o b2 = 2

n=3 b3 = 5



Les arbres binaires

- Théorème 4 : le nombre d'arbres binaires de taille n est :

  bn = C (n, 2n) / (n+1)

  appelé le nième nombre catalan.
- Théorème 5 : le nombre d'arbre binaires complets ayant 2n+1 noeuds est :

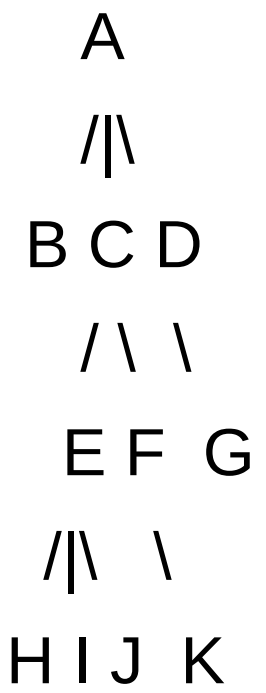
  bcn = C (n, 2n) / (n+1)
- La preuve est directe à partir des théorèmes 3 et 4.

Parcours des arbres

- On peut parcourir un arbre de trois façons :
- parcours préfixe : on commence par parcourir la racine puis les fils avec un parcours lui aussi préfixe.
- parcours infixe : on parcourt le premier sous arbre, puis la racine, puis les autres sous arbres. Les sous arbres sont aussi parcourus de façon infixe.
- parcours postfixe : on parcourt les sous arbres de façon postfixe, puis la racine.

Parcours des arbres

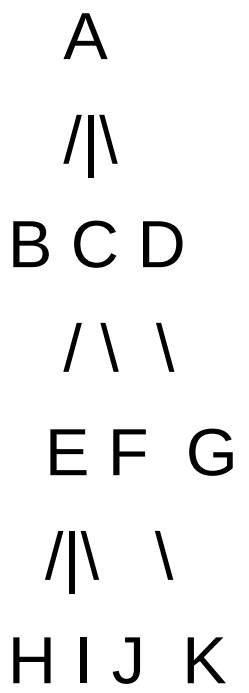
- Exemple :



- Parcours préfixe : A B C E H I J F K D G

Parcours des arbres

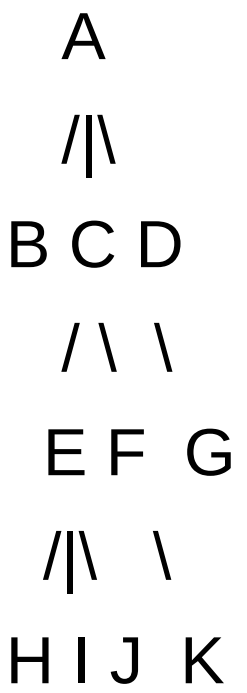
- Exemple :



- Parcours infixe : B A H E I J C K F G D

Parcours des arbres

- Exemple :



- Parcours postfixe : B H I J E K F C G D A

Mise en oeuvre des arbres

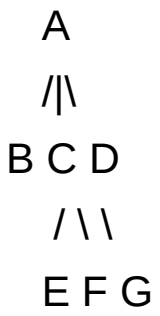
- La représentation la plus couramment utilisée est d'utiliser une structure pour chaque noeud de l'arbre.
- Cette structure contient un pointeur vers le fils le plus à gauche, et un pointeur vers le frère droit.
- On utilise en général aussi un étiquette par noeud.

Mise en oeuvre des arbres

```
class Arbre (object):
    def __init__(self, entier):
        self.entier = entier
        self.fils = None
        self.frere = None
```

Mise en oeuvre des arbres

- Pour représenter un arbre on a juste besoin d'un pointeur sur sa racine.
- Sur l'arbre suivant :



- La représentation donne :

racine -> [A,-,None]

|

[B,None,-] -> [C,-,-] -> [D,-,None]

| |

| [G,None,None]

[E,None,-] -> [F,None,None]

Mise en oeuvre des arbres

```
def prefixe (n):
    print (n.entier)
    tmp = n.fils
    while tmp != None:
        prefixe (tmp)
        tmp = tmp.frere
```

Mise en oeuvre des arbres

Exercice : ecrire infixe et postfixe

Mise en oeuvre des arbres

```
def infixe (n):
    tmp = n.fils
    if tmp != None:
        infixe (tmp)
        tmp = tmp.frere
    print (n.entier)
    while tmp != None:
        infixe (tmp)
        tmp = tmp.frere
```

Mise en oeuvre des arbres

```
def postfixe (n):
    tmp = n.fils
    while tmp != None:
        postfixe (tmp)
        tmp = tmp.frere
    print (n.entier)
```

Arbres binaires de recherche

- Motivation et définition
- Recherche dans un ABR
- Recherche d'un élément
- Recherche du plus grand élément
- Recherche de l'élément immédiatement inférieur
- Insertion dans un ABR

Motivation et définition

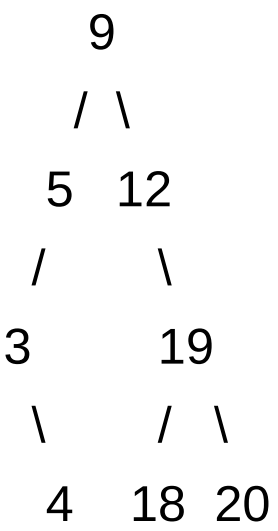
- La recherche séquentielle dans une liste chaînée non triée est en O(n) dans le pire des cas.
- La recherche dichotomique est en O(log(n)) lorsque les éléments sont triés dans un tableau.
- La représentation par tableau trié ne convient pas bien si la liste évolue dynamiquement.
- En utilisant des arbres binaires de recherche on est efficace pour les trois opérations : Recherche, Insertion, Suppression.
- On représente un ensemble ordonné de n éléments par un arbre étiqueté de n noeuds, chaque étiquette est un élément de l'ensemble.

Arbre Binaire de Recherche

- Un arbre binaire de recherche (ABR) est un arbre étiqueté tel que pour tout noeud n :

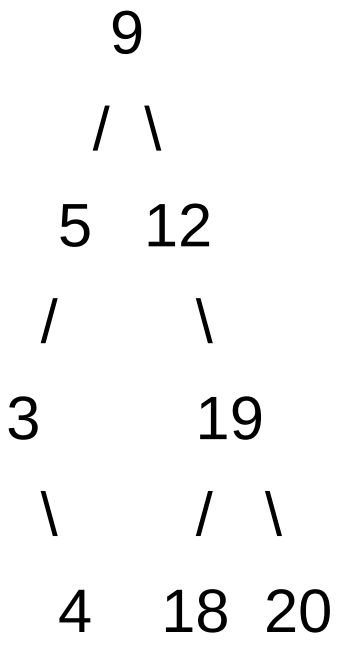
  tout noeud du sous arbre gauche a une valeur inférieure ou égale à la valeur de n

  tout noeud du sous arbre droit a une valeur supérieure à la valeur de n
- Exemple :



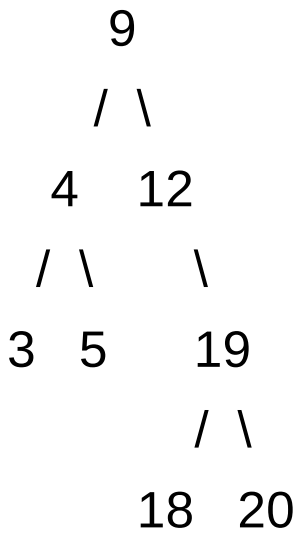
Arbre Binaire de Recherche

- Si on fait un parcours infixe de l'arbre, on obtient la liste des valeurs des nœuds triée en ordre croissant.
- Dans l'exemple : 3,4,5,9,12,18,19,20



Arbre Binaire de Recherche

- Il existe plusieurs représentations du même ensemble d'éléments par des arbres binaires.
- Par exemple, l'arbre binaire suivant représente aussi le même ensemble de chiffres :

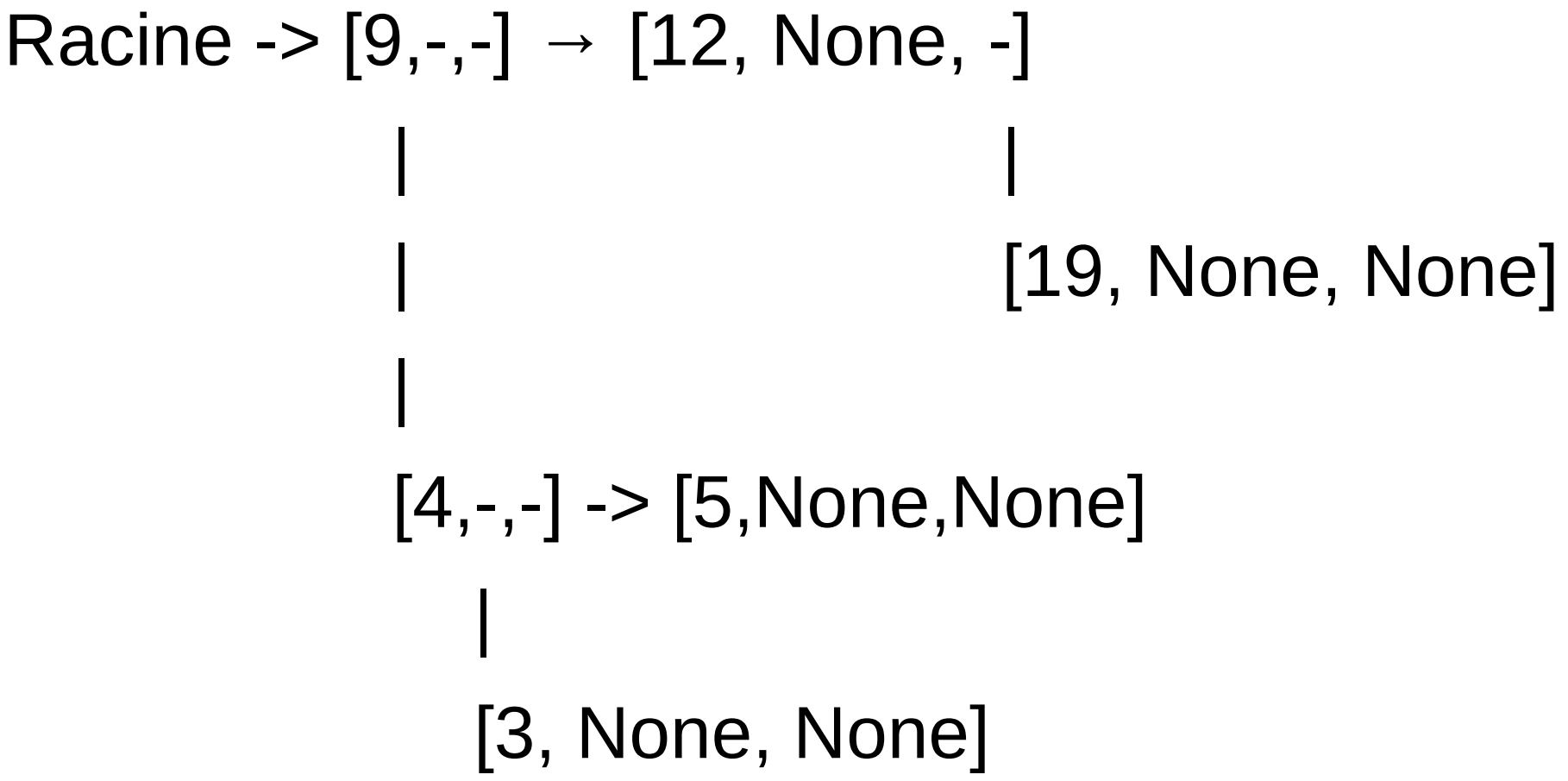


- Un parcours inxe de l'arbre donne la même liste triée.

Arbre Binaire de Recherche

```
class ABR (object):
    def __init__(self, entier):
        self.entier = entier
        self.gauche = None
        self.droite = None
```

Arbre Binaire de Recherche



Recherche dans un ABR

- Plusieurs recherches sont possibles dans un ABR :

  - rechercher un élément.

  - rechercher le plus grand ou le plus petit élément.

  - recherche pour un noeud donné celui qui lui est immédiatement supérieur (ou inférieur).

Recherche d'un élément

- Pour rechercher un élément x :

  - on compare l'élément x à la valeur de la racine

  - s'il est égal la recherche est terminée.

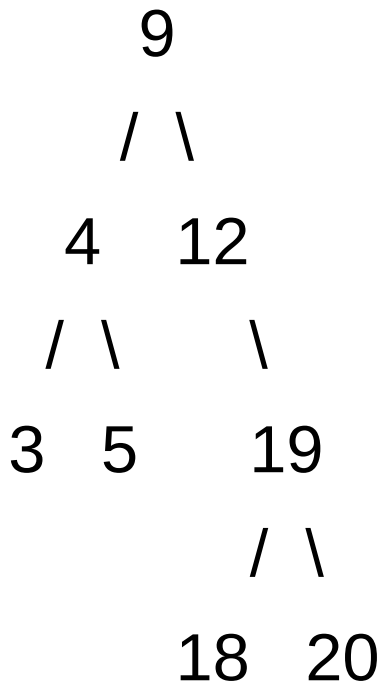
  - s'il est plus petit on poursuit la recherche dans le sous arbre gauche

  - s'il est plus grand on poursuit dans le sous arbre droit

  - si le sous arbre dans lequel on cherche est vide c'est que l'élément n'appartient pas à l'ensemble.

Recherche d'un élément

- Exemple : sur l'arbre suivant rechercher les éléments 18 et 7 :



Recherche d'un élément

- Ecrire une fonction python récursive qui recherche un élément dans un ABR.
- Le prototype est recherche (x, nœud).
- Un nœud est représenté par :

```
class ABR (object):
    def __init__(self, entier):
        self.entier = entier
        self.gauche = None
        self.droite = None
```

Recherche d'un élément

```
def recherche (x, nœud):
    if nœud == None:
        return None
    if nœud.entier == x:
        return nœud
    if x < nœud.entier:
        return recherche (x, nœud.gauche)
    else:
        return recherche (x, nœud.droit)
```

Recherche d'un élément

- Ecrire la fonction recherche (x, nœud) sans utiliser la récursivité.
- Utiliser une boucle while.

Recherche d'un élément

On peut aussi l'écrire de façon itérative :

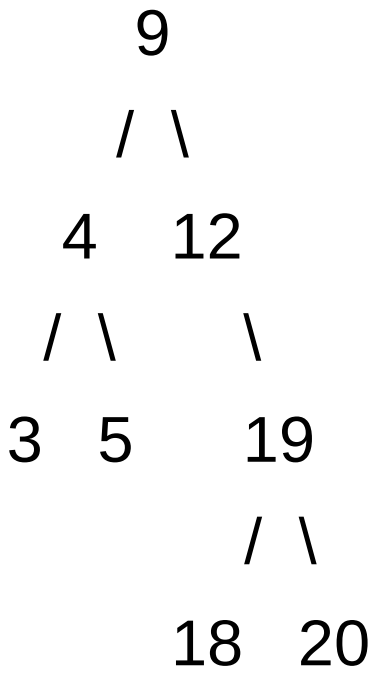
```
def recherche (x, n) :
    while n != None:
        if n.entier == x:
            return n
        if x < n.entier:
            n = n.gauche
        else:
            n = n.droit
    return None
```

Recherche du plus grand élément

- Le plus grand élément est toujours dans le fils droit puisque le fils droit contient tous les éléments plus grand que le noeud.
- S'il n'y a pas de fils droit, le plus grand élément est l'entier du noeud racine.
- Pour chercher le plus grand élément on descend donc toujours dans le fils droit tant que c'est possible, lorsque ce n'est plus possible on renvoie l'entier du noeud.

Recherche du plus grand élément

- Exemple : sur l'arbre suivant rechercher le plus grand élément :



Recherche du plus grand élément

Exercice :

Ecrire la fonction maximum(n) qui recherche le plus grand élément à partir du nœud n. Elle renvoie le nœud contenant l'entier le plus grand du sous arbre de n.

Recherche du plus grand élément

```
def maximum (n):
    if n != None:
        while n.droit != None:
            n = n.droit
    return n
```

Recherche de l'élément immédiatement inférieur

- Soit n le noeud dont on recherche l'élément immédiatement inférieur.

  1) Si n a un fils gauche, tous les éléments qui sont dans le sous arbre gauche sont inférieurs à n et il faut rechercher le plus grand élément de ce sous arbre noté y

- Preuve : il faut montrer qu'il n'existe pas ailleurs dans l'arbre un élément z tel que y < z < n

Recherche de l'élément immédiatement inférieur

- z ne peut être dans le sous arbre droit de n car tous les éléments de ce sous arbre sont supérieurs à n.
- si z est un ancêtre de n, comme z < n , n est dans le sous arbre droit de z. Or y est également dans la descendance droite de z car c'est un descendant de n, donc z < y .

Recherche de l'élément immédiatement inférieur

- si z est à gauche de n, soit r le premier sommet commun à leurs ancêtres.

  z est dans le sous arbre gauche de r => z < r

  n est dans le sous arbre droit de r => n > r

  y est dans le sous arbre droit de r => y > r

  d'où y > z et contradiction avec y < z < n
- si z est à droite de n, on aura z > n => contradiction.

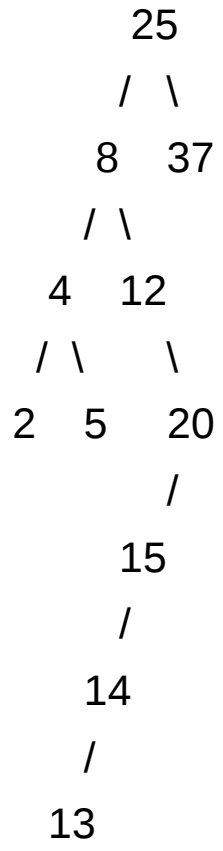
Recherche de l'élément immédiatement inférieur

2) Si n n'a pas de fils gauche

- si n est un fils droit alors l'élément immédiatement inférieur est son père.
- si n est un fils gauche alors l'élément immédiatement inférieur est le père du premier ancêtre rencontré lors de la remontée qui soit un fils droit.

Recherche de l'élément immédiatement inférieur

Exemple : sur l'arbre suivant rechercher les prédécesseurs de 12 (->8) et 13 (->12) :



Insertion dans un ABR

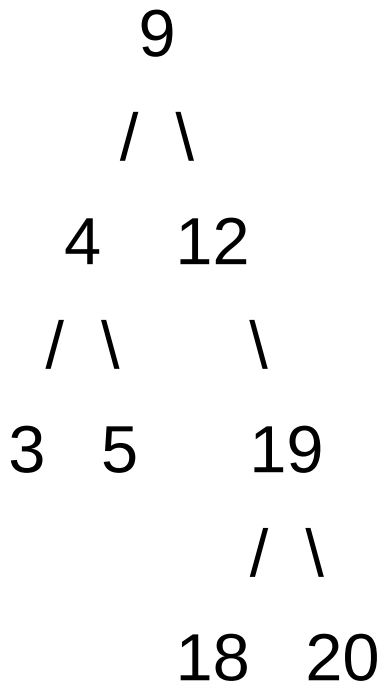
On peut ajouter un élément au niveau de la racine ou au niveau des feuilles

Ajout aux feuilles

- On cherche l'endroit où insérer le nouvel élément en recherchant ce nouvel élément.
- S'il est présent, il n'y a rien à faire.
- Si la recherche échoue on arrive sur un sous arbre vide et il faut ajouter une feuille contenant l'élément à cet endroit.
- Lorsqu'on descend dans l'arbre il faut conserver le père du noeud courant pour pouvoir y ajouter la nouvelle feuille lorsqu'on arrive sur le sous arbre vide.
- On notera nv le nouveau noeud à insérer, parent le pointeur sur le père et tmp le pointeur sur le noeud courant.

Ajout aux feuilles

- Exemple : ajouter 17 à l'arbre suivant :



Ajout aux feuilles

Exercice :

Ecrire la fonction ajouteFeuille (racine, x) qui insère x aux feuilles de l’arbre de racine x.

Ajout aux feuilles

```
def ajouteFeuille (racine, x):
    nv = ABR (x)
    tmp = racine
    parent = None
    while tmp != None:
        parent = tmp
        if x < tmp.entier:
            tmp = tmp.gauche
        elif x > tmp.entier:
            tmp = tmp.droit
        else:
            return racine
```

Ajout aux feuilles

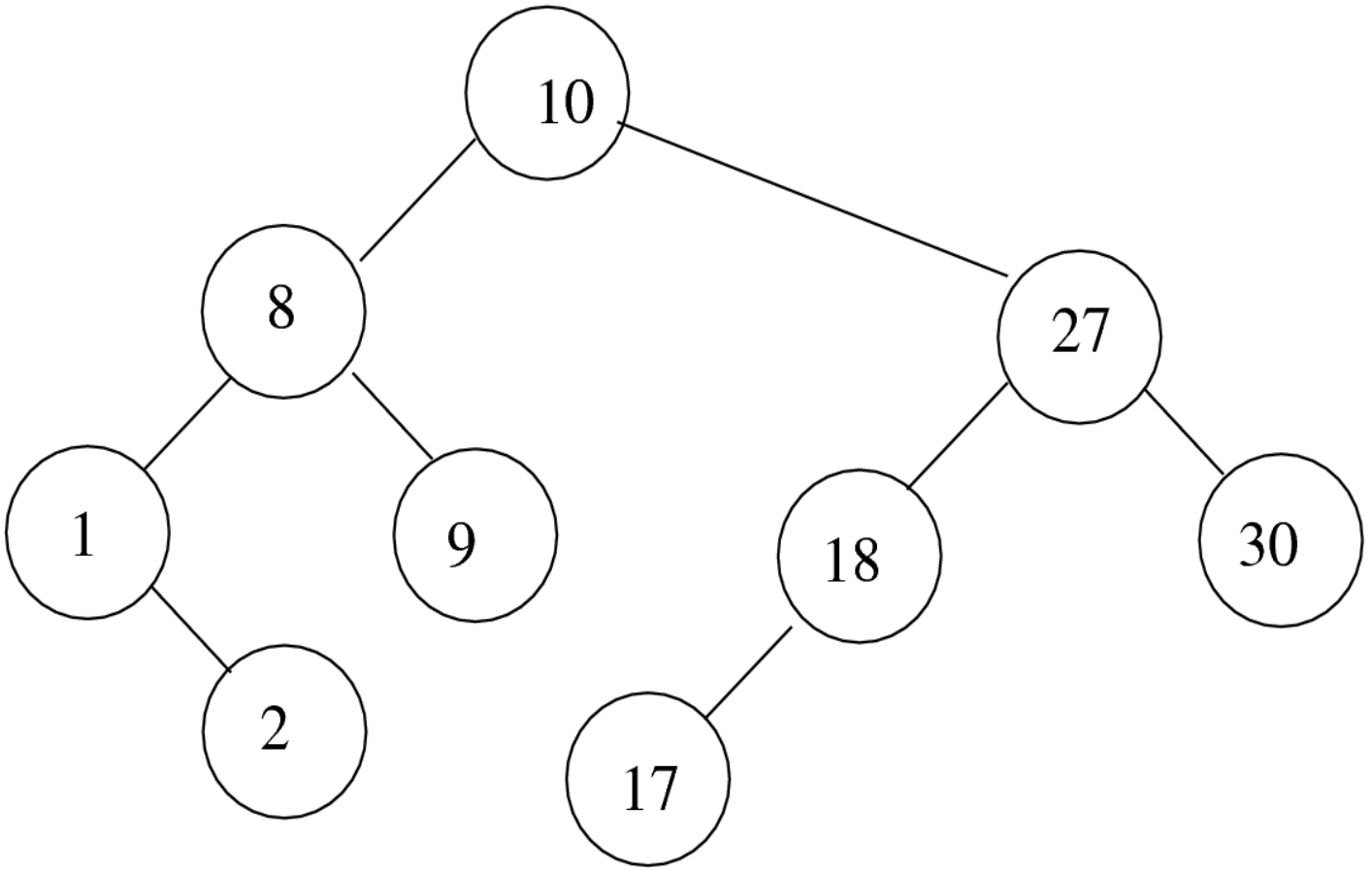
```
if parent == None:
    racine = nv
else:
    if x < parent.entier:
        parent.gauche = nv
    else:
        parent.droit = nv
return racine
```

Ajout aux feuilles

Exercice :

Représenter l’arbre binaire de recherche créé en insérant dans cet ordre les nombres suivants : 10, 8, 9, 27, 18, 30, 17, 1, 2. On ne représentera que les valeurs des noeuds et les flèches entre les noeuds.

Ajout aux feuilles



Liste triée

Exercice :

Ecrire une fonction decroissant (n, l) qui remplit la liste chaînée l de façon triée dans l'ordre décroissant à partir de l'arbre binaire de recherche de racine n.

Liste triée

```
def decroissant (n, l):
    if n != None:
        decroissant (n.gauche, l)
        s = l.suiv
        l.suiv = Cellule (n.entier)
        l.suiv.suiv = s
        decroissant (n.droit, l)
```

Arbre binaire complet

Exercice :

- Un arbre binaire complet est un arbre binaire dont chaque noeud a deux fils ou est une feuille.
- Ecrire une fonction complet (n) qui renvoie True pour une arbre binaire complet et False sinon.

Arbre binaire complet

```
def complet (n):
    if n.gauche == None and n.droit == None:
        return True
    if n.gauche != None and n.droit == None:
        return False
    if n.gauche == None and n.droit != None:
        return False
    if complet (n.gauche) and complet (n.droit):
        return True
    return False
```

Arbres Binaires de Recherche

- Suppression dans un ABR
- Complexité des opérations
- Algorithme de recherche
- Algorithme d'ajout
- Algorithme de suppression
- Les arbres AVL

Suppression dans un ABR

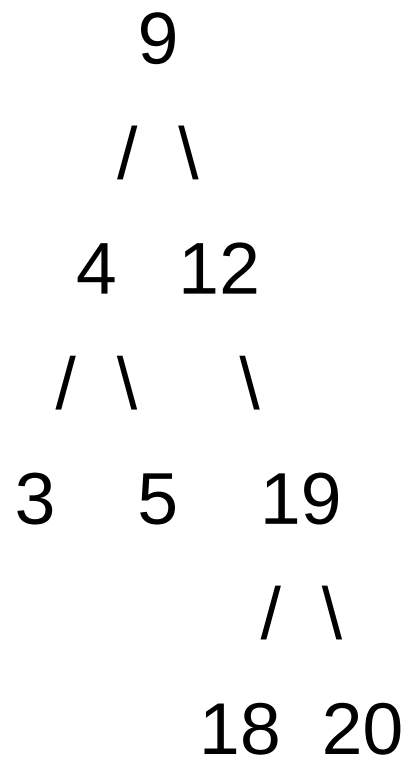
- Pour supprimer un élément dans un ABR, il faut déterminer sa place, puis effectuer la suppression.
- La suppression peut entraîner une réorganisation de l'arbre.
- si le noeud à supprimer n'a pas de fils, la suppression est immédiate.
- si le noeud à supprimer n'a qu'un seul fils, il suffit de le remplacer par ce fils.
- si le noeud à supprimer à deux fils, il existe deux possibilités :
- a) remplacer le noeud par son prédécesseur (élément immédiatement inférieur).
- b) remplacer le noeud par son successeur (élément immédiatement supérieur).
- on traite le cas a) et le cas b) est analogue.

Suppression dans un ABR

- Si on remplace x par son prédécesseur, il faut supprimer le prédécesseur.
- Le prédécesseur est le plus grand élément du sous arbre gauche, c'est donc le noeud le plus a droite du sous arbre gauche.

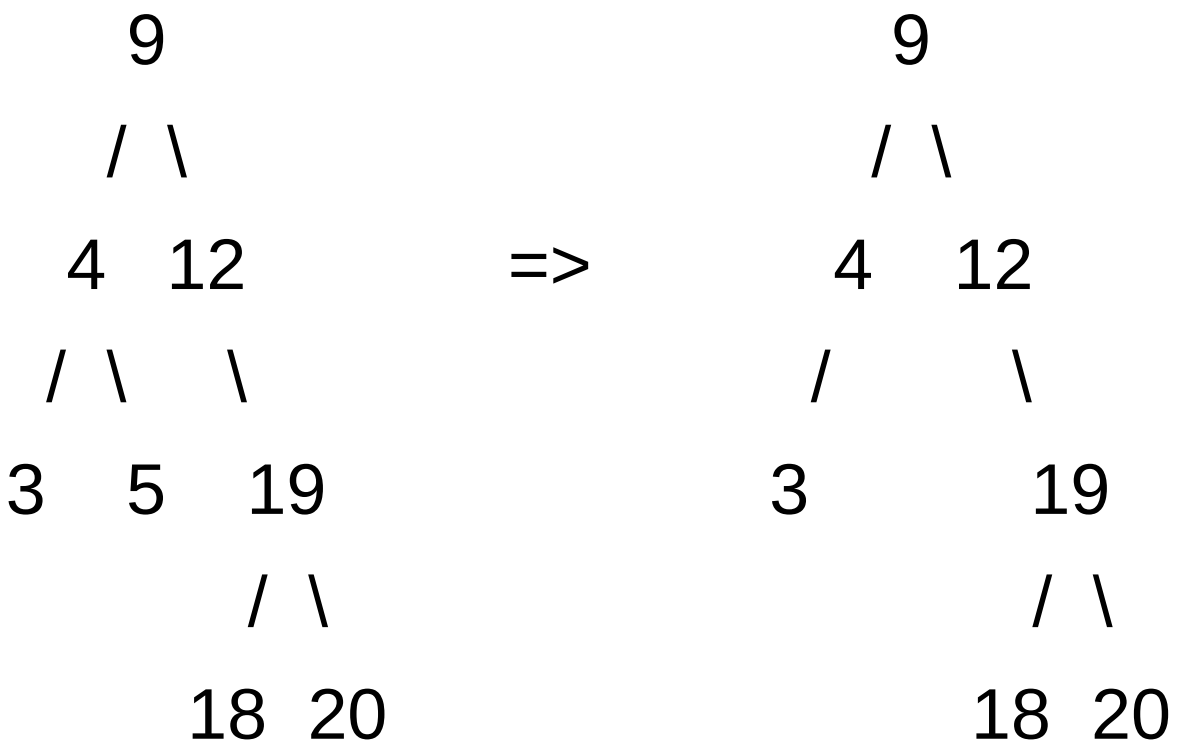
Suppression dans un ABR

- Exemples : supprimer 5, 12 et 9 de l'ABR suivant :



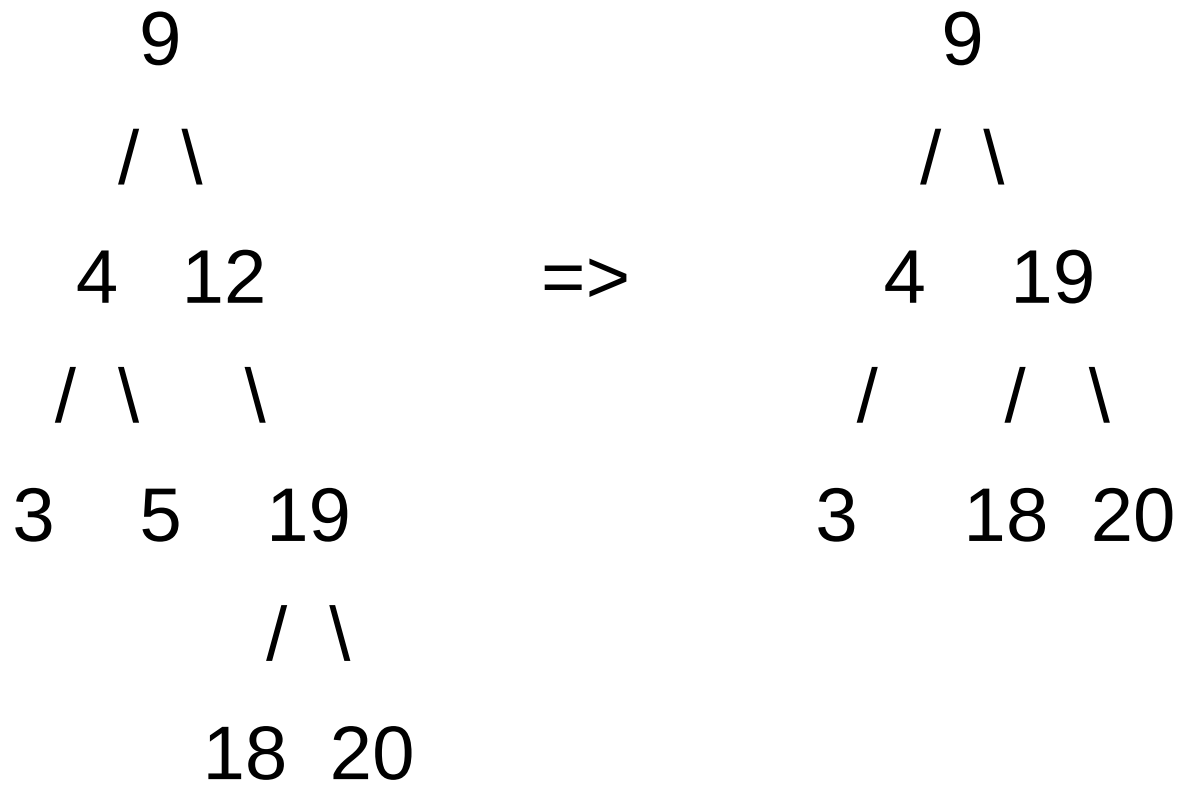
Suppression dans un ABR

- La suppression de 5 qui est une feuille donne :



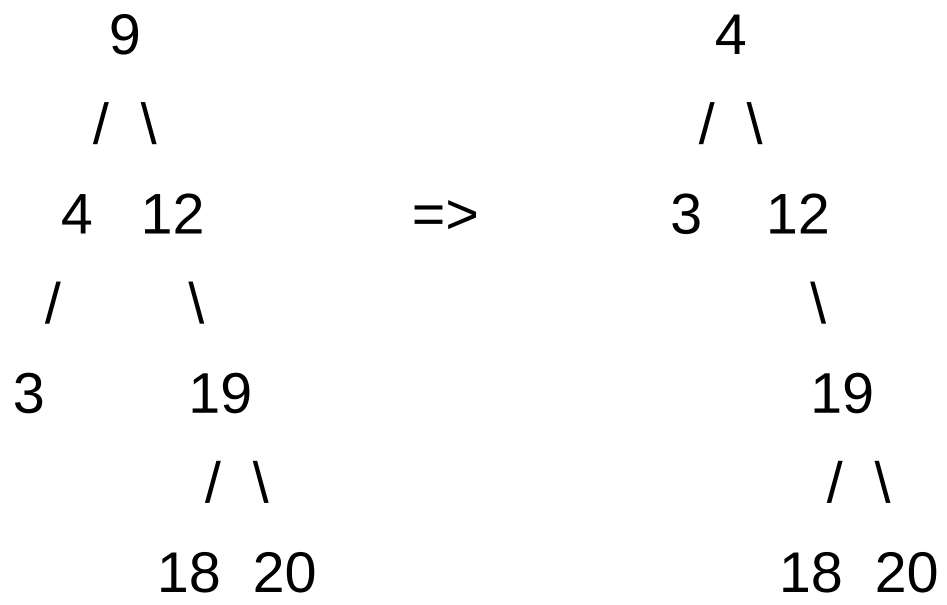
Suppression dans un ABR

- La suppression de 12 qui a un fils donne :



Suppression dans un ABR

- La suppression de 9 consiste à remplacer 9 par son prédécesseur 4. On doit donc supprimer 4 qui n'a qu'un fils et donc le remplacer par son fils. On obtient donc :



- Exercice : Ecrire l'algorithme supprimer

Suppression dans un ABR

```
Supprimer (racine, x, pere):
    if x < racine.entier:
        Supprimer (racine.gauche, x, racine)
    elif x > racine.entier:
        Supprimer (racine.droit, x, racine)
    elif racine.droit == None and racine.gauche == None:
        if pere != None:
            if pere.gauche == racine:
                pere.gauche = None
            else:
                pere.droit = None
```

Suppression dans un ABR

```
elif racine.droit == None:
    if pere != None:
        if pere.gauche == racine:
            pere.gauche = racine.gauche
        else:
            pere.droit = racine.gauche
elif racine.gauche == None:
    if pere != None:
        if pere.gauche == racine:
            pere.gauche = racine.droit
        else:
            pere.droit = racine.droit
```

Suppression dans un ABR

```
else:
    pred = Predecesseur (racine)
    racine.entier = pred.entier
    Supprimer (racine.gauche, pred.entier,
                        racine)
```

Complexité des opérations

- L'opération élémentaire est la comparaison entre deux éléments. On exprime la complexité en fonction du nombre de comparaisons.

Algorithme de recherche

- Dans le cas d'une recherche avec succès terminée sur le noeud x, le nombre de comparaisons est profondeur (x) + 1.
- On compare x à tous les noeuds depuis la racine.
- Une recherche infructueuse se termine à une feuille f et le nombre de comparaisons est profondeur (f) + 1.

Algorithme d'ajout

- Le nombre de comparaisons est le même que pour une recherche qui échoue, soit profondeur (f) + 1.

Algorithme de suppression

- Le nombre de comparaisons entre éléments est le même que pour la recherche de l'élément à supprimer dans l'arbre soit profondeur (x) + 1.

Conclusion

- L'analyse de complexité des algorithmes revient à l'étude de la profondeur des noeuds.
- Dans le pire des cas, la complexité est de l'ordre du noeud le plus profond.
- La plus grande profondeur est la hauteur de l'ABR.
- On a vu que la hauteur d'un ABR de taille n est comprise entre log(n) et n

Conclusion

- Dans le pire des cas, on a un arbre dégénéré et on recherche l'élément le plus profond de l'arbre. On a donc n comparaisons -> O(n).
- Pour les arbres binaires bien équilibrés, tous les noeuds ont deux fils et toutes les feuilles sont situées sur les deux derniers niveaux.
- La hauteur est alors en O(log(n)).
- Un ABR engendré aléatoirement est plutôt bien équilibré

Les arbres AVL

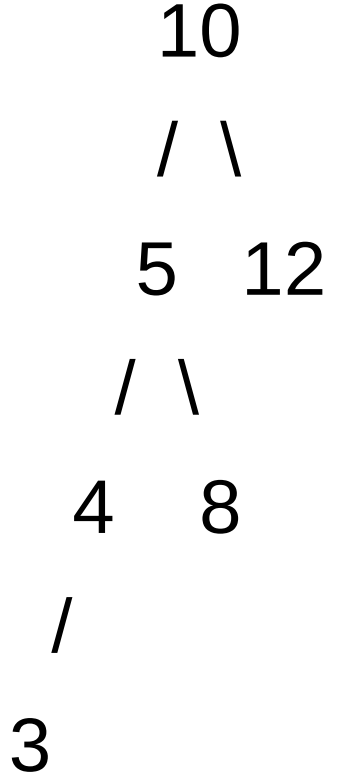
- Dans un arbre AVL, le sous arbre droit et le sous arbre gauche ont des hauteurs égales à un près.
- La recherche, l'insertion et la suppression sont en O(log(n)).

Insertion

- On commence à insérer un élément comme pour un arbre binaire normal
- Un noeud est déséquilibré si les hauteurs des sous arbres gauche et droit ont une différence strictement plus grande que un.
- En remontant du noeud inséré vers la racine, on fait des rotations sur les nœuds déséquilibrés.
- La rotation permet de faire remonter un noeud et descendre un autre en gardant l'ordre des éléments.
- La rotation permet de réduire la hauteur d'un arbre en faisant descendre les petits sous-arbres et remonter les grands.

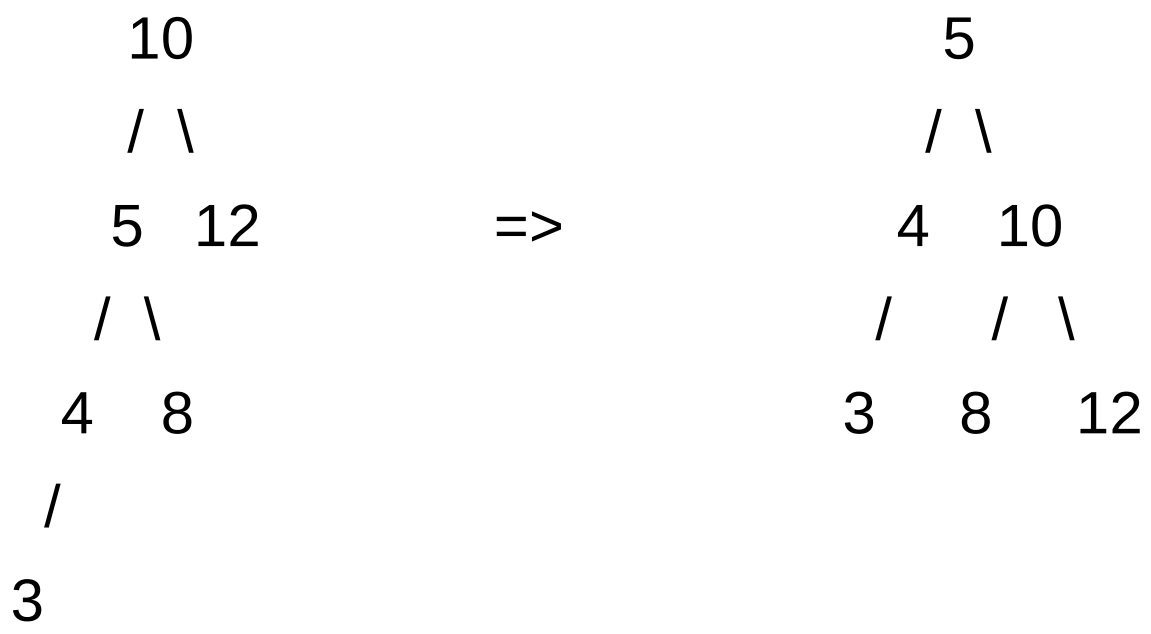
Insertion

- Par exemple pour l'arbre :



Insertion

- On effectue une rotation en faisant remonter 5 :



- C'est une rotation droite, on peut aussi faire des rotations gauches
- Ecrire une fonction RotationDroite (racine)

Algorithme de rotation

```
RotationDroite (racine):
    nouveau = racine.gauche
    racine.gauche = nouveau.droit
    nouveau.droit = racine
    nouveau.pere = racine.pere
    racine.pere = nouveau
    temp = racine.gauche
    if temp != None:
        temp.pere = racine
    return nouveau
```

Algorithme de rotation

```
def RotationGauche (racine):
    nouveau = racine.droit
    racine.droit = nouveau.gauche
    nouveau.gauche = racine
    nouveau.pere = racine.pere
    racine.pere = nouveau
    temp = racine.droit
    if temp != None:
        temp.pere = racine
    return nouveau
```

Somme des éléments

- Ecrire la fonction

  Somme (racine)

  qui effectue la somme des entiers des noeuds d’un arbre binaire.

Somme des éléments

```
def Somme (racine)
    if racine == None:
        return 0
    return Somme (racine.gauche) +
                Somme (racine.droit) +
                racine.entier
```

Nombre d'éléments supérieurs à une valeur

- Ecrire la fonction

  NombresSuperieurs (racine, x)

  qui compte le nombre d’éléments strictement supérieurs à x d’un arbre binaire de recherche.

Nombre d'éléments supérieurs à une valeur

```
def NombresSuperieurs (racine, x)
    if racine == None:
        return 0
    if racine.entier <= x:
        return NombresSuperieurs (racine.droit)
    else
         return NombresSuperieurs (racine.droit) +
                    NombresSuperieurs (racine.gauche) + 1
```

Hauteur d'un arbre binaire

- Ecrire la fonction

  Hauteur (racine)

  qui calcule la hauteur d’un arbre binaire

Hauteur d'un arbre binaire

```
def Hauteur (racine)
    if racine == None:
        return -1
    d = Hauteur (racine.droit)
    g = Hauteur (racine.gauche)
    if d > g:
        return 1 + d
    return 1 + g
```

Nombre de fils uniques d’un arbre binaire

- Ecrire une fonction qui compte le nombre de noeuds d’un arbre binaire qui n’ont qu’un seul fils.

Nombre de fils uniques d’un arbre binaire

```
def filsUniques (racine):
    if racine == None:
        return 0
    if racine.gauche == None and racine.droit != None:
        return 1 + filsUniques (racine.droit)
    elif racine.gauche != None and racine.droit == None:
        return 1 + filsUniques (racine.gauche)
    else :
        return filsUniques (racine.gauche) + filsUniques (racine.droit)
```

Nombre de noeuds à une profondeur donnée

- Ecrire une fonction qui renvoie le nombre de noeuds à profondeur p d’un arbre binaire.
- La profondeur de la racine est 0.

Nombre de noeuds à une profondeur donnée

```
def nombreNoeuds (racine, p, p1):
    if racine == None:
        return 0
    if p == p1:
        return 1
    return nombreNoeuds (racine.gauche, p, p1 + 1) +
                nombreNoeuds (racine.droit, p, p1 + 1)

def nombreNoeuds (racine, p):
        return nombreNoeuds (racine, p, 0)
```

Arbres binaires contenant les mêmes éléments

- Ecrire une fonction meme (n1, n2) qui teste si un arbre binaire n1 contient les mêmes éléments qu'un autre arbre binaire n2.

Arbres binaires contenant les mêmes éléments

```
def element (racine, e):
    if racine == None:
        return False
    if racine.entier == e:
        return True
    return element (racine.gauche, e) or element (racine.droit, e)

def meme (n1, n2):
     if n1 == None:
        return True
     if element (n1.entier, n2) == False:
           return False
     return meme (n1.gauche, n2) and meme (n1.droit, n2)
```

Arbre binaire équilibré

- Un arbre binaire de recherche est équilibré si pour chaque nœud la hauteur du sous arbre gauche et la hauteur du sous arbre droit ont une différence de -1, 0 ou 1.
- Ecrire l'algorithme Equilibre (racine) qui renvoie True si l'arbre binaire est équilibré et False sinon.

Arbre binaire équilibré

```
def equilibre (racine):
    if racine == None:
        return True
    g = hauteur (racine.gauche)
    d = hauteur (racine.droit)
    if g == d – 1 or g == d or g == d + 1:
         return equilibre (racine.gauche) and
                    equilibre (racine.droit)
    return False
```

Examen 2013

Indices de Fibonacci d’un tableau
Indices de Fibonacci d’une liste chaînée
Inversion de deux listes
Séparation en deux listes
Création d’un arbre binaire de recherche
Sommes égales
Vérification d’un tas
Elément le plus profond
Maisons les moins chères
Tri par surface

Indices de Fibonacci d’un tableau

- La suite de Fibonacci est telle que

  Un+2 = Un+1 + Un

  avec U0 = U1 = 1.
- On souhaite transférer dans un tableau T2 les éléments d’un tableau T1 qui sont aux indices correspondants aux nombres de Fibonacci.
- Par exemple pour T1 = [1, 2, 5, 7, 3, 9, 6, 4, 8] le tableau T2 contiendra les éléments d’indices 1, 2, 3, 5, 8 soit T2 = [2, 5, 7, 9, 8].

Indices de Fibonacci d’un tableau

```
U0 = 1
U1 = 1
T2 = []
while U1 < len (T1):
    T2.append (T1 [U1])
    U2 = U0 + U1
    U0 = U1
    U1 = U2
```

Indices de Fibonacci d’une liste chaînée

- Transférer dans une liste l2 les éléments d’une liste l1 qui sont aux indices correspondants aux nombres de Fibonacci.

Indices de Fibonacci d’une liste chaînée

U0 = 1

U1 = 1

l2 = Cellule (0)

p2 = l2

p1 = l1.suiv

indice = 0

Indices de Fibonacci d’une liste chaînée

```
while p1 != None:
    if indice == U1:
        p2.suiv = Cellule (p1.entier)
        p2 = p2.suiv
        U2 = U0 + U1
        U0 = U1
        U1 = U2
    p1 = p1.suiv
    indice = indice + 1
p2.suiv = None
```

Inversion de deux listes chaînées

- On souhaite qu'une liste chaînée l3 contienne les éléments de deux listes chaînées l1 et l2 dans l'ordre inverse.
- On suppose que l1 et l2 ont la même taille. Le premier élément de l3 est le dernier élément de l2, suivi du dernier élément de l1, suivi de l'avant dernier élément de l2, suivi de l'avant dernier élément de l1, etc.
- Par exemple pour l1 = [1, 2, 3, 4] et l2 = [5, 6, 7, 8] on aura l3 = [8, 4, 7, 3, 6, 2, 5, 1].

Inversion de deux listes

```
l3 = Cellule (0)
p1 = l1.suiv
p2 = l2.suiv
while p1 != None:
    p = l3.suiv
    l3.suiv = Cellule (p1.entier)
    l3.suiv.suiv = p
    p = l3.suiv
    l3.suiv = Cellule (p2.entier)
    l3.suiv.suiv = p
    p1 = p1.suiv
    p2 = p2.suiv
```

Séparation en deux listes chaînées

- Ecrire un algorithme qui permet de trouver l'élément e d'une liste chaînée tel qu'il y ait autant de valeurs supérieures à cet élément que de valeurs inférieures à cet élément dans la liste.
- Une fois cet élément trouvé séparer la liste en deux listes. La liste des éléments plus grands que e et la liste des éléments plus petits que e.

Séparation en deux listes

```
p = l.suiv
while p != None:
    plusGrand = 0
    plusPetit = 0
    p1 = l.suiv
    while p1 != None:
        if p1.entier > p.entier:
            plusGrand = plusGrand + 1
        if p1.entier < p.entier:
            plusPetit = plusPetit + 1
        p1 = p1.suiv
    if plusGrand == plusPetit:
        e = p.entier
    p = p.suiv
```

Séparation en deux listes

```
plusGrands = Cellule (0)
plusPetits = Cellule (0)
pg = plusGrands
pp = plusPetits
p = l.suiv
while p != None:
    if p.entier > e:
        pg.suiv = Cellule (p.entier)
        pg = pg.suiv
    if p.entier < e:
        pp.suiv = Cellule (p.entier)
        pp = pp.suiv
    p = p.suiv
```

Création d’un arbre binaire de recherche

- Représenter l’arbre binaire de recherche créé en insérant dans cet ordre les nombres suivants:

  23, 12, 5, 36, 30, 7, 6, 27

- On ne représentera que les valeurs des noeuds et les flèches entre les noeuds.

Sommes égales

- Ecrire un algorithme qui renvoie vrai si la propriété suivante est vérifiée et faux sinon.
- Pour tous les noeuds d’un arbre binaire la somme des éléments du sous arbre gauche est égale à la somme des éléments du sous arbre droit

Sommes égales

```
def somme (racine):
    if racine == None:
        return 0
    s = racine.entier
    s = s + somme (racine.gauche)
    s = s + somme (racine.droit)
    return s
```

Sommes égales

```
def sommesEgales (r):
    if r == None:
        return true
    if somme (r.gauche) != somme (r.droit):
        return false
    return sommesEgales (r.gauche) and
                sommesEgales (r.droit)
```

Vérification d’un tas

- Dans un arbre binaire qui représente un tas chaque valeur d’un noeud est supérieure à toutes les valeurs de tous ses descendants.
- Ecrire un algorithme qui renvoie vrai si cette propriété est vérifiée et faux sinon.

Vérification d’un tas

```
def tas (racine):
    if racine == None:
        return true
    if racine.gauche != None:
        if racine.entier < racine.gauche.entier:
            return false
    if racine.droit != None:
        if racine.entier < racine.droit.entier:
            return false
    return tas (racine.gauche) and tas (racine.droit)
```

Elément le plus profond

- Ecrire un algorithme qui renvoie l'élément le plus profond d'un arbre binaire de recherche.

Elément le plus profond

```
def hauteur (racine):
    if racine == None:
        return -1
    hg = hauteur (racine.gauche)
    hd = hauteur (racine.droit)
    if hg > hd:
        return hg + 1
    else:
        return hd + 1
```

Elément le plus profond

```
def plusProfond (racine):
    hg = hauteur (racine.gauche)
    hd = hauteur (racine.droit)
    if hg == 0 and hd == 0:
        return racine.entier
    if hg > hd:
        return plusProfond (racine.gauche)
    else:
        return plusProfond (racine.droit)
```

Liste de maisons

On souhaite modéliser une liste de maisons à l'aide de la structure suivante :

```
class Maison (object):
    def __init__(self, n, p, s):
        self.numero = n
        self.prix = p
        self.surface = s
        self.suiv = None
```

Maisons les moins chères

- Ecrire une fonction moinsCheres (l) qui renvoie la liste chaînée des deux maisons les moins chères de la liste chaînée l.

Maisons les moins chères

```
def moinsCheres (l):
    l1 = Maison (0, 0, 0)
    p = l.suiv
    if p != None:
        l1.suiv = Maison (p.numero, p.prix, p.surface)
        p = p.suiv
    if p != None:
        l1.suiv.suiv = Maison (p.numero, p.prix, p.surface)
        p = p.suiv
```

Maisons les moins chères

```
while p != None:
    if l1.suiv.prix < l1.suiv.suiv.prix:
        if p.prix < l1.suiv.suiv.prix:
            l1.suiv.suiv = Maison (p.numero, p.prix, p.surface)
    else:
        if p.prix < l1.suiv.prix:
            p1 = l1.suiv.suiv
            l1.suiv = Maison (p.numero, p.prix, p.surface)
            l1.suiv.suiv = p1
    p = p.suiv
return l1
```

Tri par surface

- Ecrire une fonction tri (l) qui trie la liste l de la maison ayant la plus petite surface à la maison ayant la plus grande surface.

Tri par surface

```
def tri (l):
    trie = False
    while trie == False:
        trie = True
        prec = l
        p = l.suiv
        while p != None:
            if p.suiv != None:
                if p.surface > p.suiv.surface:
                    trie = false
                    tmp = p.suiv.suiv
                    prec.suiv = p.suiv
                    prec.suiv.suiv = p
                    prec.suiv.suiv.suiv = tmp
            prec = p
            p = p.suiv
```

Examen 2014

Inversion d’un tableau
Inversion d’une liste chaînée
Listes suivies
Sommes égales
Création d’un arbre binaire de recherche
Largeur d’un arbre binaire de recherche
Profondeur minimale d’une feuille
Valeur de la feuille la moins profonde
Liste d’étudiants
Amplitude des notes
Ordre alphabétique

Inversion d’un tableau

- Ecrire un algorithme qui inverse un tableau tout en dupliquant ses éléments. Par exemple pour T1 = [1, 2, 3, 4] on obtiendra le tableau T2 = [4, 4, 3, 3, 2, 2, 1, 1].

Inversion d’un tableau

```
T2 = []
for i in range (len(T1)):
    T2.append (T1 [len(T1) – 1 – i])
    T2.append (T1 [len(T1) – 1 – i])
```

Inversion d’une liste chaînée

- Ecrire un algorithme qui inverse une liste chaînée tout en dupliquant ses éléments.

Inversion d’une liste chaînée

```
p1 = l1.suiv
l2 = Cellule (0)
while p1 != None:
    s = l2.suiv
    l2.suiv = Cellule (p1.entier)
    l2.suiv.suiv = Cellule (p1.entier)
    l2.suiv.suiv.suiv = s
    p1 = p1.suiv
```

Listes suivies

- On souhaite qu'une liste l3 contienne les éléments d'une liste l2 dans l'ordre inverse suivis des éléments d'une liste l1 dans le même ordre.
- Par exemple pour l1 = [1, 2, 3, 4] et l2 = [5, 6, 7, 8] on aura l3 = [8, 7, 6, 5, 1, 2, 3, 4].

Listes suivies

```
l3 = Cellule (0)
p3 = l3
p1 = l1.suiv
while p1 != None:
    p3.suiv = Cellule (p1.entier)
    p3 = p3.suiv
    p1 = p1.suiv
p2 = l2.suiv
while p2 != None:
    s = l3.suiv
    l3.suiv = Cellule (p2.entier)
    l3.suiv.suiv = s
    p2 = p2.suiv
```

Sommes égales

- Ecrire un algorithme qui permet de séparer une liste en deux listes de façon à ce que la différence des sommes des éléments des deux listes soit minimale.
- Par exemple pour l = [1, 2, 3, 4] on obtient l1 = [1, 4] et l2 = [2, 3], la somme des éléments de chaque liste fait 5 et la différence entre les deux sommes est minimale et égale à 0.

Sommes égales

```
def difference (T, T1):
    s1 = 0
    s = 0
    for i in range (len (T)):
        if T1 [i] == true:
            s1 = s1 + T [i]
        else:
            s = s + T [i]
    if s1 > s:
        return s1 – s
    else:
        return s - s1
```

Sommes égales

```
meilleur = -1
T2 = []
def recherche (T1, i)
    if i == len (T):
        eval = difference (T, T1)
        if eval < meilleur or meilleur == -1:
            meilleur = eval
            T2 = T1
    else:
        T1 [i] = true
        recherche (T1, i + 1)
        T1 [i] = false
        recherche (T1, i + 1)
```

Sommes égales

```
p = l.suiv
T= []
T1 = []
while p != None:
    T.append (p.entier)
    T1.append (true)
    T2.append (true)
    p = p.suiv
recherche (T1, 0)
l1 = Cellule (0)
l2 = Cellule (0)
p1 = l1
p2 = l2
for i in range (len (T)):
    if T2 [i] == true:
        p1.suiv = Cellule (T [i])
        p1 = p1.suiv
    else:
        p2.suiv = Cellule (T [i])
        p2 = p2.suiv
```

Création d’un arbre binaire de recherche

- Représenter l’arbre binaire de recherche créé en insérant dans cet ordre les nombres suivants : 3, 5, 19, 4, 7, 33, 12.
- On ne représentera que les valeurs des noeuds et les flèches entre les noeuds.

Largeur d’un arbre binaire de recherche

- Ecrire un algorithme qui calcule la largeur d’un arbre binaire de recherche.
- La largeur est définie par la longueur du chemin vers le plus petit élément plus la longueur du chemin vers le plus grand élément.

Largeur d’un arbre binaire de recherche

```
def largeur (racine):
    l1 = 0
    p = racine
    while p.gauche!= None:
        l1 = l1 + 1
        p = p.gauche
    l2 = 0
    p = racine
    while p.droit != None:
        l2 = l2 + 1
        p = p.droit
    return l1 + l2
```

Profondeur minimale d’une feuille

- Ecrire un algorithme qui renvoie la profondeur de la feuille la moins profonde d’un arbre binaire.

Profondeur minimale d’une feuille

```
def profondeur (racine, p):
    if racine.gauche == None and racine.droit == None:
        return p
    p1 = -1
    if racine.gauche != None:
        p1 = profondeur (racine.gauche, p + 1)
    p2 = -1
    if racine.droit != None:
        p2 = profondeur (racine.droit, p + 1)
```

Profondeur minimale d’une feuille

```
if p1 == -1
    return p2
if p2 == -1
    return p1
if p2 < p1:
    return p2
else:
    return p1
```

Valeur de la feuille la moins profonde

- Ecrire un algorithme qui renvoie la valeur de la feuille la moins profonde d’un arbre binaire de recherche.

Valeur de la feuille la moins profonde

```
def valeur (racine):
    if racine.gauche == None and racine.droit == None:
        return racine.entier
    p1 = -1
    if racine.gauche != None:
        p1 = profondeur (racine.gauche, 0)
    p2 = -1
    if racine.droit != None:
        p2 = profondeur (racine.droit, 0)
```

Valeur de la feuille la moins profonde

```
if p2 == -1
    return valeur (racine.gauche)
if p1 == -1
    return valeur (racine.droit)
if p1 < p2
    return valeur (racine.gauche)
else:
    return valeur (racine.droit)
```

Liste d’étudiants

On souhaite modéliser une liste d'étudiants à l’aide de la structure suivante :

```
class Etudiant (object):
    def __init__(self, n, no):
        self.nom = n
        self.note = no
        self.suiv = None
```

Amplitude des notes

- Ecrire une fonction amplitude (l) qui renvoie la liste des deux étudiants ayant la meilleure et la moins bonne note.

Amplitude des notes

```
def amplitude (l)
    p = l.suiv
    petit = Etudiant (p.nom, p.note)
    grand = Etudiant (p.nom, p.note)
    while p != None:
        if p.note < petit.note:
            petit.nom = p.nom
            petit.note = p.note
```

Amplitude des notes

```
    if p.note > grand.note:
        grand.nom = p.nom
        grand.note = p.note
    p = p.suiv
l1 = Etudiant (0, 0)
l1.suiv = petit
petit.suiv = grand
return l1
```

Ordre alphabétique

- Ecrire une fonction tri (l) qui trie la liste l par ordre alphabétique.

Ordre alphabétique

```
def tri (l)
    change = true
    while change == true:
        change = false
        p = l.suiv
        while p.suiv != None:
            if p.nom > p.suiv.nom:
                change = true
                nom = p.nom
                note = p.note
                p.nom = p.suiv.nom
                p.note = p.suiv.note
                p.suiv.nom = nom
                p.suiv.note = note
            p = p.suiv
```