ALGORITHME

- Définition

  nom masculin (d'Al-Khârezmi, médecin arabe).

  Suite de raisonnements ou d'opérations qui fournit la solution de certains problèmes.

- Objectifs

  Un algorithme sert à transmettre un savoir faire.

  Il décrit les étapes à suivre pour réaliser un travail.

  Il permet d'expliciter clairement les idées de solution d'un problème indépendamment d'un langage de programmation.

  L'utilisateur d'un algorithme n'aura qu'à suivre toutes les instructions, dans l'ordre pour arriver au résultat que doit donner l'algorithme.

Algorithme

- Le "langage algorithmique" que nous utilisons est un compromis entre un langage naturel et un langage de programmation.

- Nous présentons les algorithmes comme une suite d'instructions dans l'ordre des traitements. Ils sont toujours accompagnés d'un lexique qui indique, pour chaque variable, son type et son rôle.

- Nous manipulerons les types couramment rencontrés dans les langages de programmation : entier, réel, booléen, caractère, chaîne, tableau et type composite.

Formalisme

- Un algorithme doit être lisible et compréhensible par plusieurs personnes.
- Il doit donc suivre des règles. Il est composé d'une entête et d'un corps.
  - L'entête comprend :

    - Nom : le nom de l'algorithme

    - Rôle : ce que fait l'algorithme

    - Données : les données fournies à l'algorithme

    - Résultat : ce que l'on obtient à la fin du traitement

    - Principe : le principe utilisé dans l'algorithme
  - Le corps :

    - il est délimité par les mots clés début et fin.

    - il se termine par un lexique, décrivant les variables utilisées

Formalisme

- Par convention, tous les identifiants de variables seront notés en minuscule et auront un nom mnémonique
- Il en va de même pour les fonctions, dont l'identifiant doit être le plus explicite sur son rôle. Ce dernier peut être une contraction de plusieurs mots, par conséquent pour rendre la lecture plus facile, la première lettre de chaque mot est mis en majuscule (exemple : CalculerAireRectangle).

Formalisme

- Exemple d'algorithme :

Nom : AddDeuxEntiers.

Rôle : additionner deux entier et mémoriser le résultat

Données : les valeurs à additionner.

Résultat : la somme des deux valeurs.

Principe : Additionner deux entiers a et b et mettre le résultat dans c.

début

c ← a + b

fin

Lexique :

a : entier

b : entier

c : entier

Les variables

- Une variable est une entité qui contient une information, elle possède :
  - un nom, on parle **d'identifiant**
  - une valeur
  - un type qui caractérise l'ensemble des valeurs que peut prendre la variable

- L'ensemble des variables est stocké dans la mémoire de l'ordinateur

Les variables

- Type de variable
  - entier pour manipuler des entiers
  - réel pour manipuler des nombres réels
  - booléen pour manipuler des valeurs booléennes
  - caractère pour manipuler des caractères alphabétiques et numériques
  - chaîne pour manipuler des chaînes de caractères permettant de représenter des mots ou des phrases.

Les variables

- A un type donné, correspond un ensemble d'opérations définies pour ce type.
- Une variable est l'association d'un nom avec un type, permettant de mémoriser une valeur de ce type.

Opérateur, opérande et expression

- Un **opérateur** est un symbole d'opération qui permet d'agir sur des variables ou de faire des "calculs"
- Une **opérande** est une entité (variable, constante ou expression) utilisée par un opérateur
- Une **expression** est une combinaison d'opérateur(s) et d'opérande(s), elle est évaluée durant l'exécution de l'algorithme, et possède une valeur (son interprétation) et un type

Opérateur, opérande et expression

Exemple dans a + b :

a est l'opérande gauche

+ est l'opérateur

b est l'opérande droite

a + b est appelé une expression

Si par exemple a vaut 2 et b 3, l'expression a + b vaut 5

Si par exemple a et b sont des entiers, l'expression a + b est un entier

Les opérateurs

- Un opérateur peut être unaire ou binaire :
  - **Unaire** s'il n'admet qu'une seule opérande, par exemple l'opérateur non
  - **Binaire** s'il admet deux opérandes, par exemple l'opérateur +
- Un opérateur est associé à *un* type de donnée et ne peut être utilisé qu'avec des variables, des constantes, ou des expressions de *ce* type
- Par exemple l'opérateur + ne peut être utilisé qu'avec les types arithmétiques (naturel, entier et réel) ou (exclusif) le type chaîne de caractères

Les opérateurs

- **On ne peut pas additionner un entier et un caractère**
- Toutefois *exceptionnellement* dans certains cas on accepte d'utiliser un opérateur avec deux opérandes de types différents, c'est par exemple le cas avec les types arithmétiques (2 + 3.5)

- La signification d'un opérateur peut changer en fonction du type des opérandes
  - l'opérateur + avec des entiers aura pour sens l'addition, 2+3 vaut 5
  - avec des chaînes de caractères il aura pour sens la **concaténation** "bonjour" + " tout le monde" vaut "bonjour tout le monde"

Les opérateurs booléens

- Pour les booléens nous avons les opérateurs non, et, ou, ouExclusif

  - Non

a	non a
Vrai	Faux
Faux	Vrai

  - Et

a	b	a et b
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

Les opérateurs booléens

- Ou

a	b	a ou b
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

- Ou exclusif

a	b	a ouExclusif b
Vrai	Vrai	Faux
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

Les opérateurs booléens

- Rappels sur la logique booléenne...

  Valeurs possibles : Vrai ou Faux

- Associativité des opérateurs et et ou

  a et (b et c) = (a et b) et c

- Commutativité des opérateurs et et ou

  a et b = b et a

  a ou b = b ou a

Les opérateurs booléens

- Distributivité des opérateurs et et ou

  a ou (b et c) = (a ou b) et (a ou c)

  a et (b ou c) = (a et b) ou (a et c)

- Involution (homographie réciproque)

  (homos semblable et graphein écrire)

  non non a = a

- Loi de Morgan

  non (a ou b) = non a et non b

  non (a et b) = non a ou non b

Les opérateurs sur les numériques

- On retrouve tout naturellemment +, -, *, /
  - Avec en plus pour les entiers div et mod, qui permettent respectivement de calculer une division entière et le reste de cette division,

    par exemple :

    11 div 2 vaut 5

    11 mod 2 vaut 1

Les opérateurs sur les numériques

- L’opérateur d’égalité :

  C’est l’opérateur que l’on retrouve chez tous les types simples qui permet de savoir si les deux opérandes sont égales

  Il est représenté par le caractère =

  Le résultat d'une expression contenant cet opérateur est un booléen

- On a aussi l’opérateur d’inégalité : ≠

- Et pour les types possédant un ordre les opérateurs de comparaison

  <, ≤, >, ≥

Priorités des opérateurs

- Tout comme en arithmétique les opérateurs ont des priorités

  Par exemple * et / sont prioritaires sur + et -

  Pour les booléens, la priorité des opérateurs est non, et, ouExclusif et ou

- Pour clarifier les choses (ou pour dans certains cas supprimer toutes ambiguïtés) on peut utiliser des parenthèses

Manipulation de variables

- On ne peut faire que deux choses avec une variable :

1. Obtenir son contenu

    Cela s’effectue simplement en nommant la variable

2. Affecter un (nouveau) contenu

    Cela s’effectue en utilisant l’opérateur d’affectation représenter par le symbole ←

    La syntaxe de cet opérateur est :

    identifiant de la variable ← expression

Manipulation de variables

- Par exemple l'expression c ← a + b se comprend de la façon suivante :

  On prend la valeur contenue dans la variable a

  On prend la valeur contenue dans la variable b

  On additionne ces deux valeurs

  On met ce résultat dans la variable c

- Si c avait auparavant une valeur, cette dernière est perdue !

Les entrées / sorties

- Un algorithme peut avoir des interactions avec l'utilisateur
  - Il peut afficher un résultat (du texte ou le contenu d'une variable)
  - demander à l'utilisateur de saisir une information afin de la stocker dans une variable
- En tant qu'informaticien on raisonne en se mettant **"à la place de la machine"**, donc :

Les entrées / sorties

- Instruction d'écriture

  L'instruction de restitution de résultats sur le périphérique de sortie (en général l'écran) est :

  écrire(liste d'expressions)

  Cette instruction réalise simplement l'affichage des valeurs des expressions décrites dans la liste.

  Ces instructions peuvent être simplement des variables ayant des valeurs ou même des nombres ou des commentaires écrits sous forme de chaînes de caractères.

- Exemple d'utilisation : écrire(x, y+2, "bonjour")

Les entrées / sorties

- Instructions de lecture

L'instruction de prise de données sur le périphérique d'entrée (en général le clavier) est :

variable <- lire()

L'exécution de cette instruction consiste à affecter une valeur à la variable en prenant cette valeur sur le périphérique d'entrée.

Avant l'exécution de cette instruction, la variable avait ou n'avait pas de valeur. Après, elle a la valeur prise sur le périphérique d'entrée.

Les entrées / sorties

- Exemple

On désire écrire un algorithme qui lit sur l'entrée standard une valeur représentant une somme d'argent et qui calcule et affiche le nombre de billets de 100 Euros, 50 Euros et 10 Euros, et de pièces de 2 Euros et 1 Euro qu'elle représente.

Nom : DécompSomme.

Rôle : Décomposition d'une somme

Données : la somme à décomposée.

Résultat : les nombres de billets et de pièces.

Principe : on commence par lire sur l'entrée standard l'entier qui représente la somme d'argent et affecte la valeur à

une variable somme.

Pour obtenir la décomposition en nombre de billets

et de pièces de la somme d'argent, on procède par

divisions successives en conservant chaque fois le reste.

Les entrées / sorties

```
début
    somme <- lire()
    billet100 <- somme div 100
    reste100 <- somme mod 100
    billet50 <- reste100 div 50;
    reste50 <- reste100 mod 50
    billet10 <- reste50 div 10
    reste10 <- reste50 mod 10
    piece2 <- reste10 div 2
    reste2 <- reste10 mod 2
    piece1 <- reste2
    écrire (billet100, billet50, billet10, piece2, piece1)
fin
```

Les entrées / sorties

- Lexique

- somme : entier, la somme d'argent à décomposer
- billet100 : entier, le nombre de billets de 100 Euros
- billet50 : entier, le nombre de billets de 50 Euros
- billet10 : entier, le nombre de billets de 10 Euros
- piece2 : entier, le nombre de pièces de 2 Euros
- piece1 : entier, le nombre de pièces de 1 Euro
- reste100 : entier, reste de la division entière de somme par 100
- reste50 : entier, reste de la division entière de reste100 par 50
- reste10 : entier, reste de la division entière de reste50 par 10
- reste2 : entier, reste de la division entière de reste10 par 2

Instruction conditionnelle

- L’instruction si alors sinon permet de conditionner l’exécution d’un algorithme à la valeur d’une expression booléenne. Syntaxe :

si *expression booléenne* **alors**

suite d’instructions exécutées si l’expression est vrai

sinon

suite d’instructions exécutées si l’expression est fausse

finsi

Instruction conditionnelle

- La deuxième partie de l'instruction est optionnelle, on peut avoir la syntaxe suivante :

si *expression booléenne* **alors**

suite d'instructions exécutées si l'expression est vrai

finsi

Instruction conditionnelle

- Exemple

Nom : ValeurAbs

Rôle : Calcule la valeur absolue d'un entier

Données : La valeur à calculer

Résultat : La valeur Absolue

Principe : Si valeur < 0, on la multiplie par -1

```
début
    si valeur ≥ 0 alors
        valeurabsolue ← valeur
    sinon
        valeurabsolue ← valeur * -1
    finsi
fin
```


Lexique

- valeur : entier, la valeur à tester
- valeurabsolue : la valeur absolue

L'instruction cas

- Lorsque l'on doit comparer une **même** variable avec plusieurs valeurs, comme par exemple :

  si a=1 **alors**
      faire une chose
  sinon
      si a=2 **alors**
          faire une autre chose
      sinon
          si a=4 **alors**
              faire une autre chose
          sinon
          . . .
          finsi
      finsi
  finsi

- On peut remplacer cette suite de si par l'instruction cas

L'instruction cas

- Sa syntaxe est :

cas où v **vaut**
v1 : action1
v2 : action2
. . .
vn : actionn
autre : action autre
fincas

v1,. . . ,vn sont des **constantes** de type **scalaire** (entier, naturel, enuméré, ou caractère)

action i est exécutée si v = vi (on quitte ensuite l'instruction cas)

action autre est exécutée si quelque soit i, v ≠ vi

Les itérations

- Il arrive souvent dans un algorithme qu'une même action soit répétée plusieurs fois, avec éventuellement quelques variantes.

  Il est alors fastidieux d'écrire un algorithme qui contient de nombreuses fois la même instruction. De plus, ce nombre peut dépendre du déroulement de l'algorithme.

  Il est alors impossible de savoir à l'avance combien de fois la même instruction doit être décrite.

  Pour gérer ces cas, on fait appel à des instructions en boucle qui ont pour effet de répéter plusieurs fois une même instruction.

  Deux formes existent : la première, si le nombre de répétitions est connu avant l'exécution de l'instruction de répétition, la seconde s'il n'est pas connu. L'exécution de la liste des instructions se nomme itération.

Répétitions inconditionnelles

- Il est fréquent que le nombre de répétitions soit connu à l'avance, et que l'on ait besoin d'utiliser le numéro de l'itération afin d'effectuer des calculs ou des tests. Le mécanisme permettant cela est la boucle Pour.

- Forme de la boucle Pour :

```
Pour variable de valeur initiale à valeur finale faire
        liste d'instructions
fpour
```

Répétitions inconditionnelles

- La variable dont on donne le nom va prendre successivement toutes les valeurs entières entre valeur initiale et valeur finale. Pour chaque valeur prise par la variable, la liste des instructions est exécutée.

- La valeur utilisée pour énumérer les itérations est appelée valeur d'itération, indice d'itération ou compteur. L'incrémentation par 1 de la variable est implicite.

Répétitions inconditionnelles

- Autre forme de la boucle Pour :

Pour variable décroissante de valeur initiale à valeur finale faire

liste d'instructions

fpour

La variable d'itération est décrémentée de 1 après chaque itération.

Les répétitions conditionnelles

- L'utilisation d'une "boucle pour" nécessite de connaître à l'avance le nombre d'itérations désiré, c'est-à-dire la valeur finale du compteur. Dans beaucoup de cas, on souhaite répéter une instruction tant qu'une certaine condition est remplie, alors qu'il est à priori impossible de savoir à l'avance au bout de combien d'itérations cette condition cessera d'être satisfaite. Dans ce cas, on a deux possibilités :

  - la boucle Tant que

  - la boucle Répéter jusqu'à

- Syntaxe de la boucle Tant que :

```
tant que condition faire
    liste d'instructions
ftant
```

Les répétitions conditionnelles

- Cette instruction a une condition de poursuite dont la valeur est de type booléen et une liste d'instructions qui est répétée si la valeur de la condition de poursuite est vraie : la liste d'instructions est répétée autant de fois que la condition de poursuite a la valeur vraie. Le déroulement pas à pas de cette instruction

  équivaut à :

  si condition
  alors liste d'instructions
      si condition
      alors liste d'instructions
          si condition
          alors liste d'instructions
              ...

Les répétitions conditionnelles

- Etant donné que la condition est évaluée avant l'exécution des instructions à répéter, il est possible que celles-ci ne soient jamais exécutées.
- Il faut que la liste des instructions ait une incidence sur la condition afin qu'elle puisse être évaluée à faux et que la boucle se termine.
- Il faut toujours s'assurer que la condition devient fausse au bout d'un temps fini.
- Exemple

  Un utilisateur peut construire des rectangles de taille quelconque, à condition que les largeurs qu'il saisit soient supérieures à 1 pixel.

Les répétitions conditionnelles

Nom : saisirLargeurRectangle

Rôle : Vérification validité largeur saisie

Données : La largeur

Résultat :

Principe : Tant que la largeur est < 1, on demande de resaisir la largeur

```
début
    écrire ("indiquez la largeur du rectangle :")
    largeur <- lire()
    tant que largeur < 1 faire
        écrire ("erreur : indiquez une valeur strictement positive")
        écrire ("indiquez la largeur du rectangle :")
        largeur <- lire()
    ftant
fin
```

Lexique

- largeur : entier, largeur courante saisie

Les répétitions conditionnelles

- Syntaxe de la boucle Répéter jusqu'à :

  Répéter

  liste d'instructions

  jusqu'à condition

La séquence d'instructions est exécutée au moins une fois et jusqu'à ce que l'expression soit vraie. Dès que la condition est vrai, la répétitivité s'arrête.

Les répétitions conditionnelles

Nom : saisirLargeurRectangle
Rôle : Vérification validité largeur saisie
Données : La largeur
Résultat :
Principe : Tant que la largeur est < 1, on demande de resaisir la largeur

```
début
	Répéter
		 écrire ("indiquez la largeur du rectangle :")
		largeur <- lire()
		si largeur < 1 alors
			écrire ("erreur : indiquez une valeur strictement positive")
		fin si
	jusqu'à largeur >= 1
fin
```

Lexique
- largeur : entier, largeur courante saisie

Les répétitions conditionnelles

- Différences entre les boucles Tant que et Répéter jusqu'à :
  - la séquence d'instructions est exécuter au moins une fois dans la boucle Répéter jusqu'à, alors qu'elle peut ne pas être exécuter dans le cas du Tant que.

  - la séquence d'instructions est exécuter si la condition est vrai pour le Tant que et si la condition est fausse pour le Répéter jusqu'à.

  - Dans les deux cas, la séquence d'instructions doit nécessairement faire évoluer la condition, faute de quoi on obtient une boucle infinie.

Les tableaux

- Lorsque les données sont nombreuses et de même type, afin d'éviter de multiplier le nombre des variables, on les regroupe dans un tableau
- Le type d'un tableau précise le type (commun) de tous les éléments.
- Syntaxe :

  tableau type_des_éléments[borne_inf ... borne_sup]
- En général, nous choisirons toujours la valeur 0 pour la borne inférieure dans le but de faciliter la traduction de l'algorithme vers les autres langages (C, Java, ...). Pour un tableau de 10 entiers, on aura :

  tab : tableau entier[0..9]

Les tableaux

- Les tableaux à une dimension ou vecteurs

0	1	2	3	4	5	6	7	8	9
45	54	1	-56	22	134	49	12	90	-26

- Ce tableau est de longueur 10, car il contient 10 emplacements.

- Chacun des dix nombres du tableau est repéré par son rang, appelé indice

- Pour accéder à un élément du tableau, il suffit de préciser entre crochets l'indice de la case contenant cet élément.
  Pour accéder au 5ème élément (22), on écrit : Tab[4]

Les tableaux

- Les instructions de lecture, écriture et affectation s'appliquent aux tableaux comme aux variables.

  x ← Tab[0]

  La variable x prend la valeur du premier élément du tableau, c'est à dire : 45

  Tab[6] ← 43

  Cette instruction a modifiée le contenu du tableau

Les tableaux

- Parcours complet d'un tableau

  La plupart des algorithmes basés sur les tableaux utilisent des itérations permettant de faire un parcours complet ou partiel des différents éléments du tableau. De tels algorithmes établissent le résultat recherché par récurrence en fonction des éléments successivement rencontrés.

  Les répétitions inconditionnelles sont le moyen le plus simple de parcourir complètement un tableau.

Les tableaux

Dans l'exemple suivant, le programme initialise un à un tous les éléments d'un tableau de n éléments :

```
InitTableau
 début
     pour i de 0 à n-1 faire
         tab[i] ← 0
     fpour
 fin
```

Lexique :

- i : entier, indice d'itération
- n : entier, taille du tableau
- tab : tableau entier[0..n-1]

Les tableaux

- Les tableaux à deux dimensions ou matrices

Lorsque les données sont nombreuses et de même nature, mais dépendent de deux critères différents, elles sont rangées dans un tableau à deux entrées.

	0	1	2	3	4	5	6
0	12	28	44	2	76	77	32
1	23	36	51	11	38	54	25
2	43	21	55	67	83	41	69

Ce tableau a 3 lignes et 7 colonnes. Les éléments du tableau sont repérés par leur numéro de ligne et leur numéro de colonne.

Tab[1, 4] = 38

Les tableaux

- La variable T[L, C] s'appelle l'élément d'indice L et C du tableau T.

T[0, 0]	T[0, 1]	T[0, 2]	T[0, 3]	T[0, 4]
T[1, 0]	T[1, 1]	T[1, 2]	T[1, 3]	T[1, 4]
T[2, 0]	T[2, 1]	T[2, 2]	T[2, 3]	T[2, 4]

Tableau T à 3 lignes et 5 colonnes.

- Les tableaux peuvent avoir n dimensions.

Les procédures et les fonctions

- La méthodologie de base de l'informatique est :

1. Abstraire

Retarder le plus longtemps possible l'instant du codage

2. Décomposer

"...diviser chacune des difficultés que j'examinerai en autant de parties qu'il se pourrait et qu'il serait requis pour les mieux résoudre." Descartes

3. Combiner

Résoudre le problème par combinaison d'abstractions

Les procédures et les fonctions

- Par exemple, résoudre le problème suivant :

  Ecrire un programme qui affiche en ordre croissant les notes d'une promotion suivies de la note la plus faible, de la note la plus élevée et de la moyenne, revient à résoudre les problèmes suivants :

  - Remplir un tableau de naturels avec des notes saisies par l'utilisateur
  - Afficher un tableau de naturels
  - Trier un tableau de naturel en ordre croissant
  - Trouver le plus petit naturel d'un tableau
  - Trouver le plus grand naturel d'un tableau
  - Calculer la moyenne d'un tableau de naturels

Les procédures et les fonctions

Chacun de ces sous-problèmes devient un nouveau problème à résoudre.

Si on considère que l'on sait résoudre ces sous-problèmes, alors on sait "quasiment" résoudre le problème initial.

Donc écrire un programme qui résout un problème revient toujours à écrire des sous-programmes qui résolvent des sous parties du problème initial.

En algorithmique il existe deux types de sous-programmes :

- Les fonctions
- Les procédures

Les fonctions récursives

- Une fonction récursive est une fonction qui pour fournir un résultat, s'appelle elle-même un certain nombre de fois.

- Exemple : la formule de calcul de la factorielle d'un nombre n s'écrit :

  n ! = 1 x 2 x 3 x … x n

  Ce qui peut se programmer avec une boucle Pour.

Les fonctions récursives

Une autre manière de voir les choses, serait de dire que :

n ! = n x (n-1) !

D'où, la factorielle d'un nombre, c'est ce nombre multiplié par la factorielle du nombre précédent.

- Pour programmer cela, il faut une fonction Facto, chargée de calculer la factorielle. Cette fonction effectue la multiplication du nombre passé en argument par la factorielle du nombre précédent.

Et cette factorielle du nombre précédent va bien entendu être elle-même calculée par la fonction Fact.

Les fonctions récursives

- Pour terminer, il nous manque la condition d'arrêt de ces auto-appels de la fonction Facto.

  Dans le cas d'un calcul de factorielle, on s'arrête quand on arrive au nombre 1, pour lequel la factorielle est par définition 1.

```
Fonction Facto (n : Entier)
Début
Si n = 1 alors
    Renvoyer 1
Sinon
    Renvoyer Facto(n - 1) * n
Finsi
Fin
```

Les fonctions récursives

- Le processus récursif remplace en quelque sorte la boucle, c’est-à-dire un processus itératif.
- Il est à noter que l'on traite le problème à l’envers : on part du nombre, et on remonte à rebours jusqu’à 1, pour pouvoir calculer la factorielle.
- Cet effet de rebours est caractéristique de la programmation récursive.

Les fonctions récursives

- Pour conclure sur la récursivité, trois remarques fondamentales.
  - la programmation récursive, pour traiter certains problèmes, peut être très économique, elle permet de faire les choses correctement, en très peu de lignes de programmation.

  - en revanche, elle est très dispendieuse de ressources machine. Car il faut créer autant de variable temporaires que de " tours " de fonction en attente.

  - toute fonction récursives peut également être formulée en termes itératifs ! Donc, si elles facilitent la vie du programmeur, elle ne sont pas indispensable.

Les algorithmes de tri

- Les algorithmes de tri, sont utilisés principalement pour les tableaux et les listes, ils peuvent aller du plus simple au plus compliquer.

- Le tri à bulle

  C’est un des algorithmes le plus connu. Bien qu’il soit rarement efficace en terme de temps de calcul, il est néanmoins correcte.

  Le principe consiste à balayer tout le tableau, en comparant les éléments adjacents et en les échangeant s’ils ne sont pas dans le bon ordre.

Les algorithmes de tri

Un seul passage ne déplacera un élément donné que d'une position, mais en répétant le processus jusqu'à ce que plus aucun échange ne soit nécessaire, le tri sera effectué.

Ce tri va nécessiter un grand nombre de déplacements d'éléments, il est donc inutilisable dans les cas où ces déplacements sont coûteux en temps.

Il peut par contre être intéressant quand le tableau initial est pré-trié, les éléments n'étant pas disposés trop loin de leur position finale, par exemple lors d'un classement alphabétique, où les éléments sont déjà triés par leur première lettre.

Les algorithmes de tri

- Le tri par insertion

Plutôt que de déplacer les éléments d'une position, on peut prendre un élément après l'autre dans l'ordre initial, et le placer correctement dans les éléments précédents déjà triés, comme on le fait lorsque l'on classe ses carte à jouer après la donne.

Le tri par insertion peut être intéressant pour des tableaux ayant déjà été triés, mais où l'on doit rajouter quelques nouveaux éléments.

Les algorithmes de tri

- Le tri par sélection

Le but est désormais de déplacer chaque élément à sa position définitive.

On recherche l'élément le plus petit. Il faut donc le placer en premier, or cette position est déjà occupée, on échange donc les deux éléments.

Il ne reste plus qu'à répéter l'opération N fois.

Chaque échange met un élément en position définitive, l'autre par contre est mal placé. Cependant, aucun échange n'est inutile, car un élément qui a été bien placé, ne sera plus testé par la suite.

Les algorithmes de tri

- Le tri shell

C’est une amélioration du tri par insertion, au lieu d’effectuer une rotation de tous les éléments entre la position initiale et finale d’un élément, on peut faire des rotation par pas de P, ce qui rendra le fichier presque trié.

Chaque élément sera à moins de P positions de sa position exacte.

On répète ce tri pour P diminuant jusqu’à 1.

Une suite possible pour P est de finir par 1, les pas précédents étant de 4, 13, 40, 121, 364, 1093.

Les algorithmes de tri

D’autres suites sont possibles, à condition de prendre des valeurs qui ne soient pas multiples entre elles, pour ne pas toujours traiter les mêmes éléments et laisser de côté les autres.

Les puissances successives de 2 ne traiteraient que les positions paires, sauf au dernier passage.

L’intérêt de ce tri, est qu’il créé rapidement un fichier presque trié, en appliquant un dernier par insertion, celui-ci sera beaucoup plus rapide.

Les algorithmes de tri

- Le tri rapide (Quick Sort)

Ce tri est récursif. On cherche à trier une partie du tableau, délimitée par les indices gauche et droite.

On choisit une valeur de ce sous tableau que l'on appelle pivot (une valeur médiane serait idéale, mais sa recherche ralentit plus le tri que de prendre une valeur aléatoire).

Puis on cherche la position définitive de ce pivot, c'est-à-dire que l'on effectue des déplacements de valeurs de telle sorte que tous les éléments avant le pivot soient plus petit que lui et que tous ceux placés après lui soient supérieurs, mais sans chercher à les classer, pour accélérer le processus.

Les algorithmes de tri

Puis on rappelle récursivement le tri de la partie avant le pivot et celle de la partie après le pivot.

On arrête la récursivité sur les parties à un seul élément, dans ce cas, le tri à obligatoirement été effectué.

- Le tri par création

Lorsqu'il est nécessaire de disposer simultanément du tableau initial et du tableau trié, on peut recopier le tableau initial puis effectuer un tri sur la copie ou adapter un des algorithmes précédents.

Les algorithmes de tri

- Les autres tris

Suivant les données à trier, il peut être plus efficace de construire un algorithme de tri spécifique.

Par exemple si un tableau contient un grand nombre de valeurs similaires, on peut utiliser un algorithme simple, consistant à rechercher l'élément le plus petit, compter le nombre de ces éléments, les mettre dans un tableau destination, et répéter l'opération jusqu'à la fin du fichier. C'est le tri par comptage.

Le tri par fusion utilise un algorithme de fusion de deux tableaux triés en un seul plus grand, appelé récursivement sur les deux moitiés du tableau, jusqu'à une taille de tableau de 1.

Les algorithmes de recherche

- La recherche séquentielle

  A partir d'un tableau trié, on parcours ce tableau élément par élément jusqu'à trouver le bon élément.

- La recherche dichotomique

  La recherche dichotomique recherche un élément dans un tableau trié et retourne l'indice d'une occurrence de cet élément.

  On compare l'élément cherché à celui qui se trouve au milieu du tableau. Si l'élément cherché est plus petit, on continu la recherche dans la première moitié du tableau, sinon dans la seconde moitié.

  On recommence ce processus sur la moitié. On s'arrête lorsque l'on a trouvé l'élément, ou lorsque l'intervalle de recherche est nul.

Les structures de données dynamiques

- Jusqu'ici, on a étudié des structures de données statiques, c'est-à-dire qui ont un nombre fixe d'informations, utilisées pour la représentation de données dont la taille est connue à l'avance et n'évolue pas dans le temps.
- Dans la pratique, il arrive souvent que l'on ait besoin de représenter des objets dont on ne connaît pas à priori la taille, ou dont la taille est variable selon les cas ou au cours du temps.
- On utilise dans ce cas des structures de données dynamiques qui peuvent évoluer pour s'adapter à la représentation des ces objets.

Les structures de données dynamiques

- Exemple : On doit lire dans un fichier une suite de nombres sur lesquels on doit effectuer un traitement ultérieur. On ne connaît pas la quantité de nombres à lire et on ne veut pas les compter avant.

- La première solution consiste à utiliser un tableau surdimensionner, au risque qu'il soit trop petit ou réellement trop grand, ce qui induit une occupation mémoire inutile.

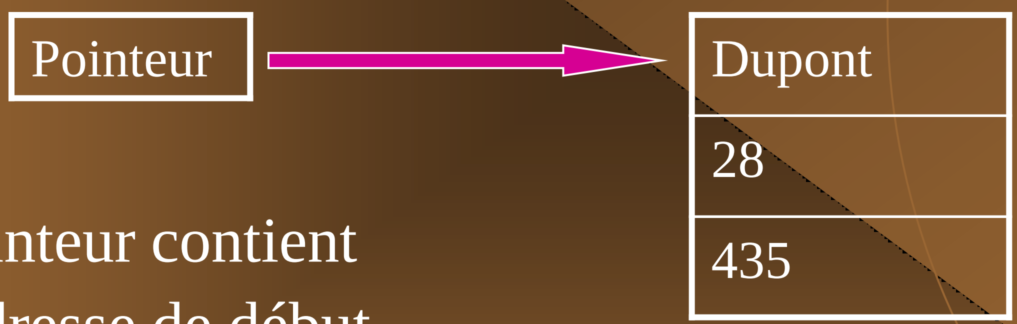
- L'autre solution consiste à utiliser une structure dynamique qui s'agrandit au fur et à mesure de la lecture des nombres.

Les structures de données dynamiques

- Le principe de base est de suivre les évolution de la structure en lui attribuant de la place en mémoire quand elle grandit et en la récupérant quand elle diminue
- Ceci est réalisé par un mécanisme d'allocation et de libération dynamique d'espace mémoire qui utilise une zone mémoire particulière appelée TAS (HEAP) dans laquelle on réserve des emplacements quand c'est nécessaire, Que l'on libère quand on en a plus besoin.
- On dispose de deux procédures standard d'acquisition et de libération de l'espace mémoire :RESERVE et LIBERE

Les pointeurs

- Les procédures RESERVE et LIBERE travaillent exclusivement avec des pointeurs.
- Une variable de type pointeur est une variable dont le contenu est une adresse qui peut indiquer l'emplacement en mémoire d'une autre variable, créée dynamiquement et ayant un type de base précis.



Pointeur contient l'adresse de début d'enregistrement

- Quand une variable pointeur ne pointe sur rien, elle doit contenir la valeur NULL.

Structure autoréférentielle

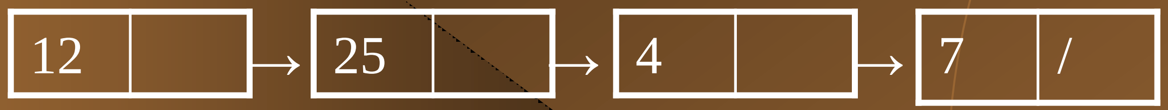
- Une structure autoréférentielle (ou structure récursive) correspond à une structure dont au moins un des champs contient un pointeur vers une structure de même type.
- De cette manière, on crée des éléments (appelés nœud ou lien) contenant des données, mais contrairement à un tableau, celle-ci peuvent être éparpillées en mémoire et reliées entre-elles par des liens logiques (des pointeurs).
- Un ou plusieurs champs dans chaque structure contient l'adresse d'une ou de plusieurs structures de même type.

Structure autoréférentielle

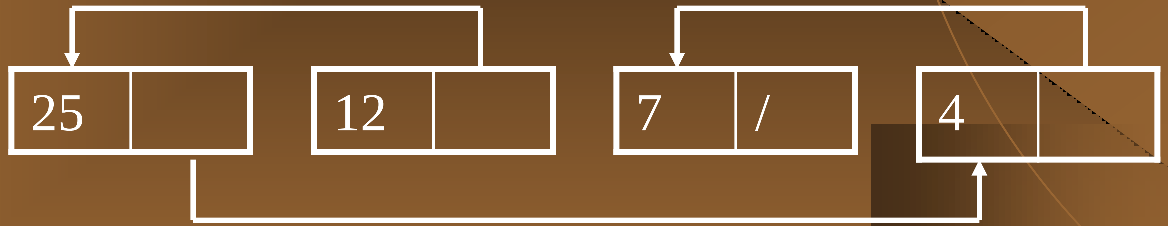
- Si une structure contient des données et un pointeur vers la structure suivante, on parle de **liste chaînée**.
- Lorsque la structure contient des données, un pointeur vers la structure suivante et un pointeur vers la structure précédente, on parle de **liste chaînée double**.
- Lorsque la structure contient des données, un pointeur vers une première structure suivante et un pointeur vers une seconde, on parle d'**arbre binaire**.

Les listes chaînées

- Une liste chaînée est une structure de données dans laquelle les éléments sont rangés linéairement. Chaque élément est lié à son successeur, il est donc impossible d'accéder directement à un élément quelconque de la liste



- Bien évidement, cette linéarité est purement virtuelle. A la différence du tableau, les éléments n'ont aucune raison d'être contigus ni ordonnés en mémoire.



Les listes chaînées

- La façon dont on met en œuvre ces structures dépend des langages, même si la façon dont cela est représenté ici ressemble fortement à celle du langage C.

- Une méthode simple pour se représenter une liste, consiste à se dire

  - qu'une liste *L* est soit vide, soit elle est constituée

  - d'une tête *T* (qui est donc la valeur du premier élément de la liste) et

  - d'une queue *Q* (qui est le reste de la liste).

Les listes chaînées

- En terme de pointeurs et de structures, une liste peut se représenter grâce au type suivant :

Structure Maillon

Entier valeur

Maillon suivant

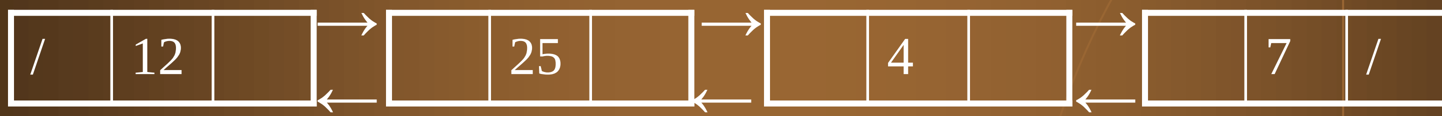
Les listes chaînées

- L'exemple précédent peut se définir par

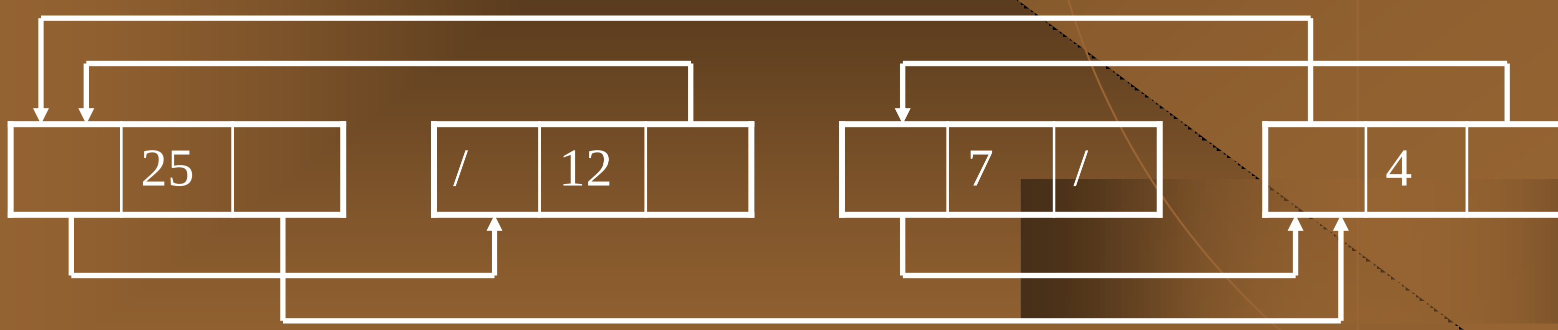
```
Maillon cell1, cell2, cell3, cell4
Début
    cell1.valeur ← 12
    cell1.suivant ← #cell2
    cell2.valeur ← 25
    cell2.suivant ← #cell3
    cell3.valeur ← 4
    cell3.suivant ← #cell4
    cell4.valeur ← 7
    cell4.suivant ← NULL
Fin
```

Les listes doublement chaînées

- Une liste doublement chaînée est une liste chaînée, à ceci près que l'on peut accéder à son prédécesseur.



- Bien sûr, cette linéarité est purement virtuelle. Tout comme pour les listes chaînées simples, les éléments n'ont aucune raison d'être contigus ni ordonnés en mémoire.



Les listes doublement chaînées

- En terme de pointeurs et de structures, on peut définir ce type de la façon suivante :

Structure Maillon

Entier valeur

Maillon suivant

Maillon precedent

Les listes doublement chaînées

Maillon cell1, cell2, cell3, cell4

Début

cell1.valeur ← 12

cell1.suivant ← #cell2

cell1.precedent ← NULL

cell2.valeur ← 25

cell2.suivant ← #cell3

cell2.precedent ← #cell1

cell3.valeur ← 4

cell3.suivant ← #cell4

cell3.precedent ← #cell2

cell4.valeur ← 7

cell4.suivant ← NULL

cell4.precedent ← #cell3

Fin

Les listes

- Elles ont l'avantage d'être réellement dynamiques, c'est dire que l'on peut à loisir les rallonger ou les raccourcir, avec pour seule limite la mémoire disponible.
- De plus, l'insertion d'un composant au milieu d'une liste ne nécessite que la modification des liens avec l'élément précédent et le suivant. Le temps nécessaire pour l'opération sera donc indépendant de la longueur de la liste.

Les piles et les files

- Ce sont des structures de données ordonnées, mais qui ne permettent l'accès qu'à une seule donnée.
- Les piles (LIFO : Last In First Out) correspondent à une pile d'assiettes : on prend toujours l'élément supérieur, le dernier emplilé.
- Les files (FIFO : First In First Out) correspondent aux files d'attente : on prend toujours le premier élément, donc le plus ancien.
- Elles sont très souvent utiles, elles servent à mémoriser des évènements en attente de traitement.

Les piles et les files

- Il n'y a pas de structures spécifiques prévues dans les langages de programmation, il faut donc les créer de toute pièce.

- Pour les piles, on utilisera :
  - un tableau unidimensionnel en cas de piles de hauteur maximale prévisible (la hauteur de la pile est mémorisée dans une variable entière).

  - une liste en cas de longueur très variable. Attention au surcoût de mémoire, dû aux liens (pointeurs), il faudra en créer autant que d'éléments empiler.

Les piles et les files

- Pour les files :

  - l'utilisation d'un tableau nécessite de mémoriser deux variables, la position du premier élément et celle du dernier. La suppression du premier élément ne se fait pas par décalage des suivants, mais en incrémentant la variable indiquant le premier. La gestion est alors un peu plus complexe que pour les piles.

  - l'utilisation d'une liste pour les files est aussi simple que pour une pile.

Les piles et les files

- Les fonctions de base
  - Pour les piles
    - l'empilage
    - le dépilage
  - Pour les files
    - l'enfilage
    - le défilage
  - Dans les deux cas on prévoira également une fonction d'initialisation et une fonction indiquant si la pile ou la file est vide.

    Seules ces deux fonctions dépendent de la méthode de mise en œuvre (tableau ou liste).

Les piles et les files

```
Structure Pile
        Entier sommet
        Réel tableau

Booléen Pile_Vide(Pile P)
Début
        Retourner(P.sommet = -1)            // test si pile vide
Fin

Empiler(Pile P, Réel x)
Début
        P.sommet ← P.sommet +1
        P.tableau[P.sommet] ← x
Fin
```

Les piles et les files

```
Réel Dépiler(Pile P)
Début
    Si Pile_Vide(P) Alors
        Ecrire("Pile vide!")
    Sinon
        P.sommet ← P.sommet -1
        Retourner P.tableau[P.sommet + 1]
Fin
```

Les arbres

- Les listes sont des structures dynamiques unidimensionnelles.
  Les arbres sont leur généralisation multidimensionnelle.

- Une liste est un arbre dont chaque élément a un et un seul fils.

- Chaque composante d'un arbre contient une valeur et des liens vers ses fils.

- Définition :
  Un arbre est formé d’un élément de type arbre appelé racine et d’un nombre fini d’arbres appelés sous-arbres.

  Un élément quelconque peut avoir plusieurs successeurs, mais un seul prédécesseur. Seul le premier élément n’a pas de prédécesseur.

Les arbres

- Terminologie
  - Les éléments
    - Un sommet ou nœud est un élément quelconque d'un arbre.
    - La racine est l'unique sommet n'ayant pas de prédécesseur.
    - Une feuille ou nœud terminal est un élément n'ayant pas de successeur.
    - Le prédécesseur unique d'un nœud est appelé le père.
    - Les successeurs d'un nœud sont appelés les fils.
    - Les nœuds qui ont le même père sont appelés frères.
    - Le frère placé le plus à gauche est l'aîné.
  - Les chemins
    - Un arc est un chemin reliant deux nœuds
    - Une branche est un chemin reliant la racine à une feuille.

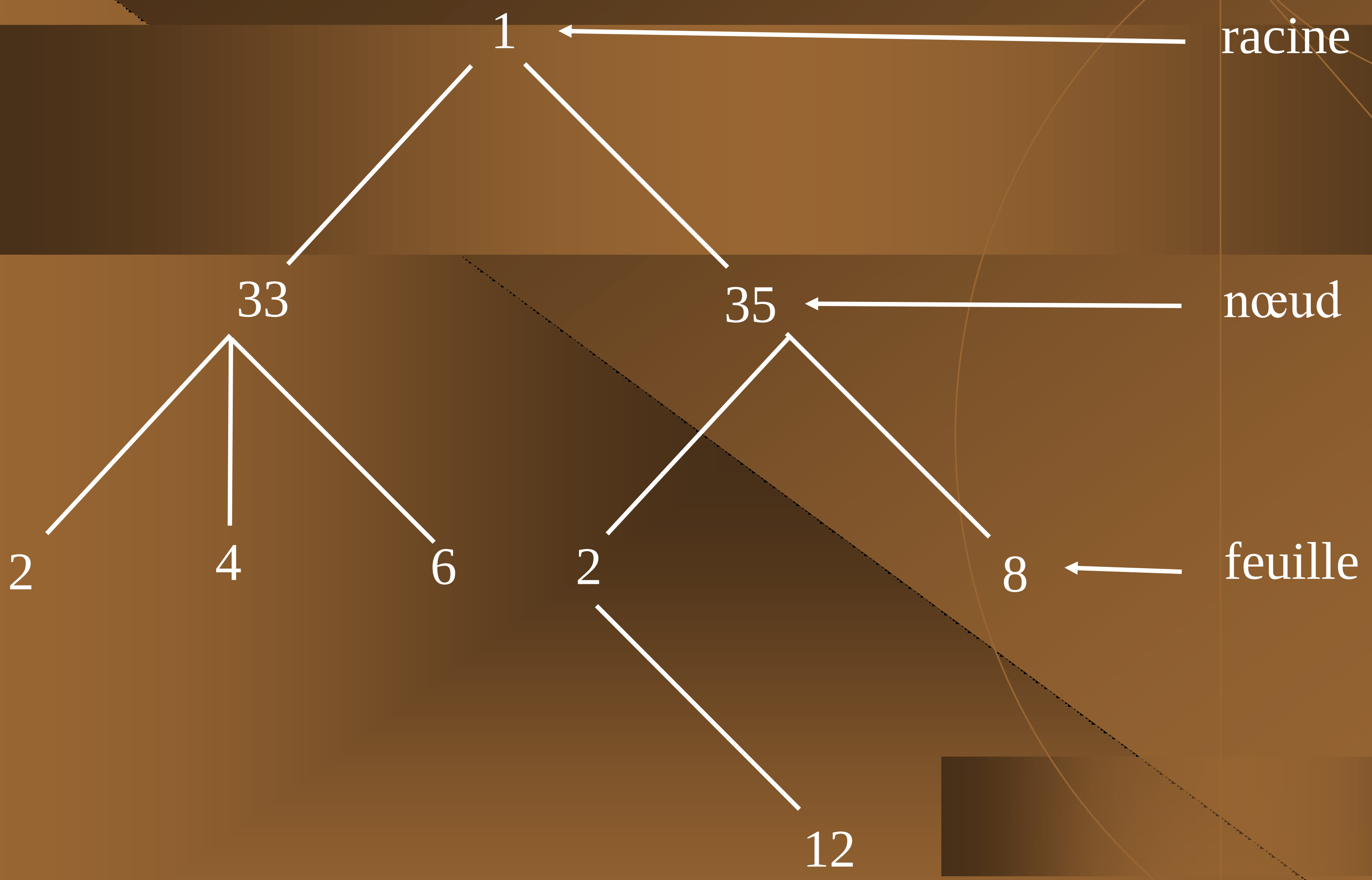
Les arbres

- Qualification des arbres
  - Horizontale
    - Un arbre est dit n-aires, lorsque le nombre maximum de fils d’un même nœud est n.
    - Un niveau est défini par un ensemble de nœuds disposés à égale distance de la racine. Le niveau d’un arbre n-aire est qualifié de saturé si tous ses nœuds ont exactement n fils.
    - Le prédécesseur unique d’un nœud est appelé le père.
    - Un arbre n-aire est complet au sens strict, si tout niveau commencé est saturé.

Les arbres

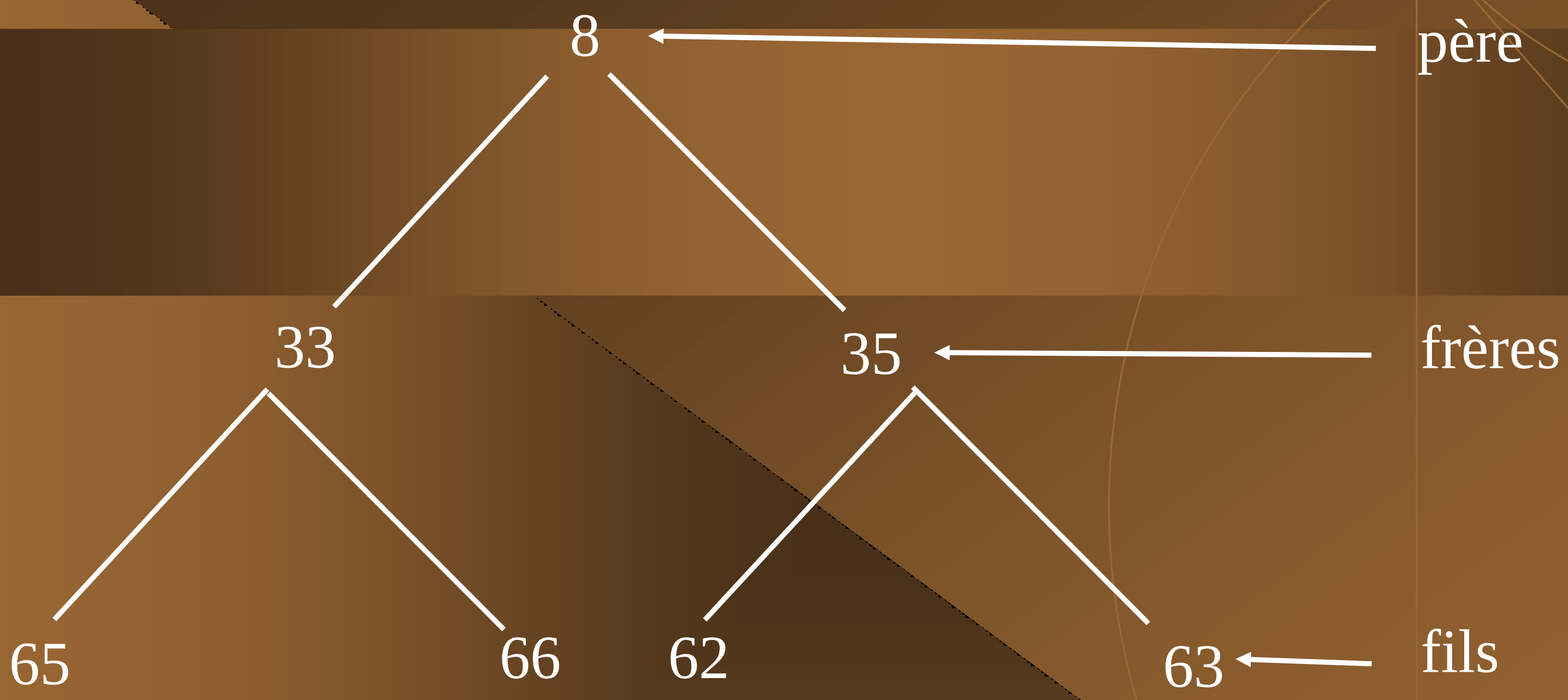
- Verticale
  - La hauteur d’un arbre binaire est le nombre de nœuds qui constituent la plus longue branche de la racine une feuille.
  - Un arbre constitué de seulement une racine a une hauteur de 1.
  - S’il est constitué d’une racine et d’une feuille sa hauteur est de 2.

Les arbres



Arbre ternaire incomplet de hauteur 4

Les arbres



Arbre binaire strictement complet de hauteur 3

Les arbres

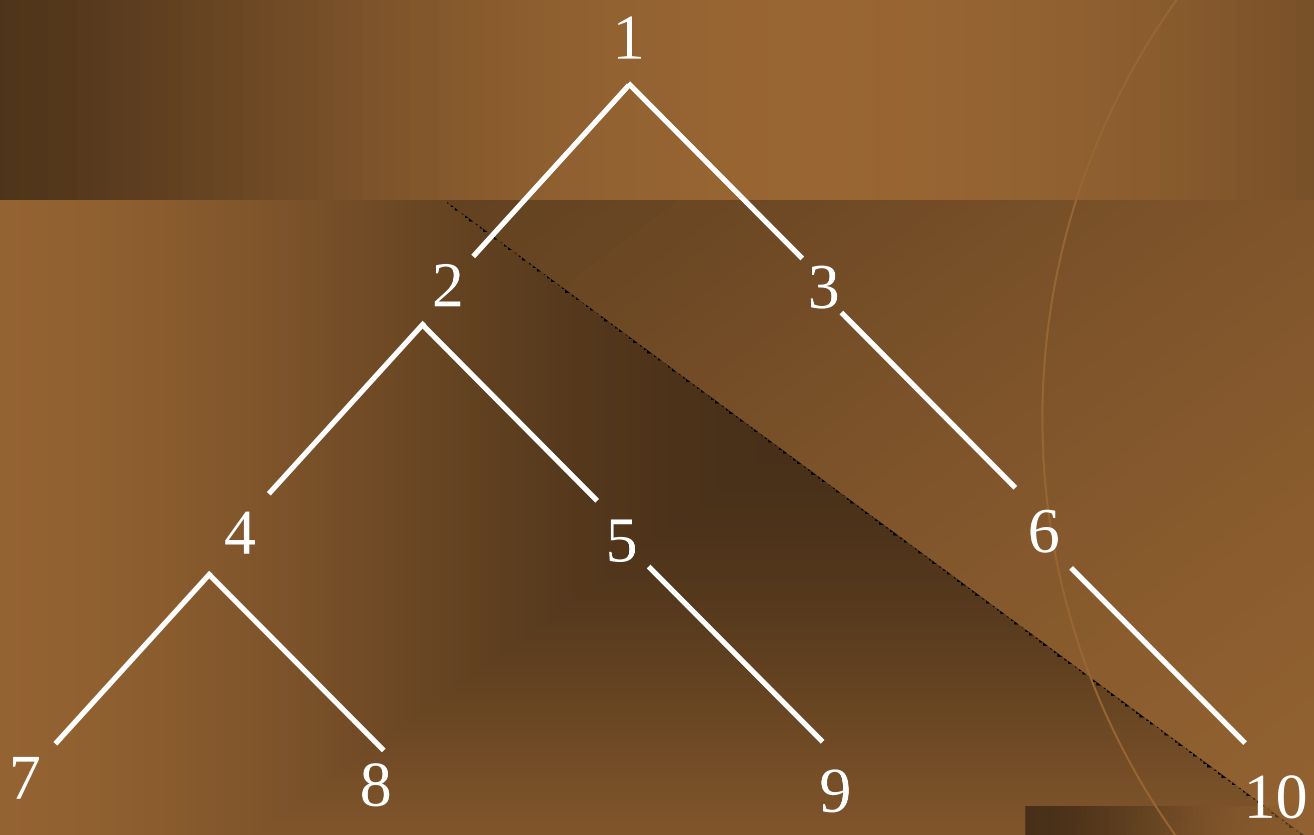
- Comme pour le premier d'une liste, l'adresse de la racine est nécessaire et suffisante pour accéder à l'intégralité d'un arbre.
- Une liste est un arbre unaire.
- Tout arbre peut être représenté par un arbre binaire

Les arbres

- De nombreux algorithmes ont été développés pour les arbres binaires.
- Comme les listes les arbres sont complètement dynamiques, mais les liens (pointeurs) sont également gourmands en mémoire.
- L'accès au données reste séquentiel, mais il est néanmoins rapide.
- Le parcours des arbres, c'est-à-dire le passage par tous les nœuds et feuilles se fait en général de manière récursive.

Les arbres

Parcours d'un arbre binaire



Les arbres

- Méthodologie

Plusieurs méthodes permettent d'accéder à tous les nœuds d'un arbre binaire.

Son parcours complet peut-être décrit récursivement par les actions :

- traiter la racine

- parcourir le sous-arbre gauche

- parcourir le sous-arbre droit

Il y a six séquencements possibles pour ces actions.

Les trois parcours suivant sont les plus classiques.

Les arbres

- Parcours préfixé

  Ce parcours est également appelé parcours en préordre ou descendant gauche droite.

  Il traite d'abord les nœuds de la branche la plus à gauche en descendant.

  - traiter la racine

  - parcourir le sous-arbre gauche

  - parcourir le sous-arbre droit

Les arbres

```
procedure prefixe(entree p : pointeur)
Debut
      si p <> NULL
            alors   afficher(p → val)
                    prefixe(p → g)
                    prefixe(p → d)
            sinon   afficher(f)
      fin si
Fin
```

Premier appel : prefixe(racine)

Résultat :

1 2 4 7 f f 8 f f 5 f 9 f f 3 f 6 f 10 f f

Les arbres

- Parcours postfixé

Ce parcours est également appelé parcours en postordre ou ascendant gauche droite ou en ordre terminal.

Il traite en dernier les nœuds de la branche la plus à droite en remontant.

- parcourir le sous-arbre gauche

- parcourir le sous-arbre droit

- traiter la racine

Les arbres

```
procedure postfixe(entree p : pointeur)
Debut
    si p <> NULL
        alors   postfixe(p → g)
                postfixe(p → d)
                afficher(p → val)
        sinon   afficher(f)
    fin si
Fin
```

Résultat :

f f 7 f f 8 4 f f f 9 5 2 f f f f 10 6 3 1

Les arbres

- Parcours infixé

  Ce parcours est également appelé parcours symétrique, projectif ou hiérarchique canonique.

  - parcourir le sous-arbre gauche

  - traiter la racine

  - parcourir le sous-arbre droit

Les arbres

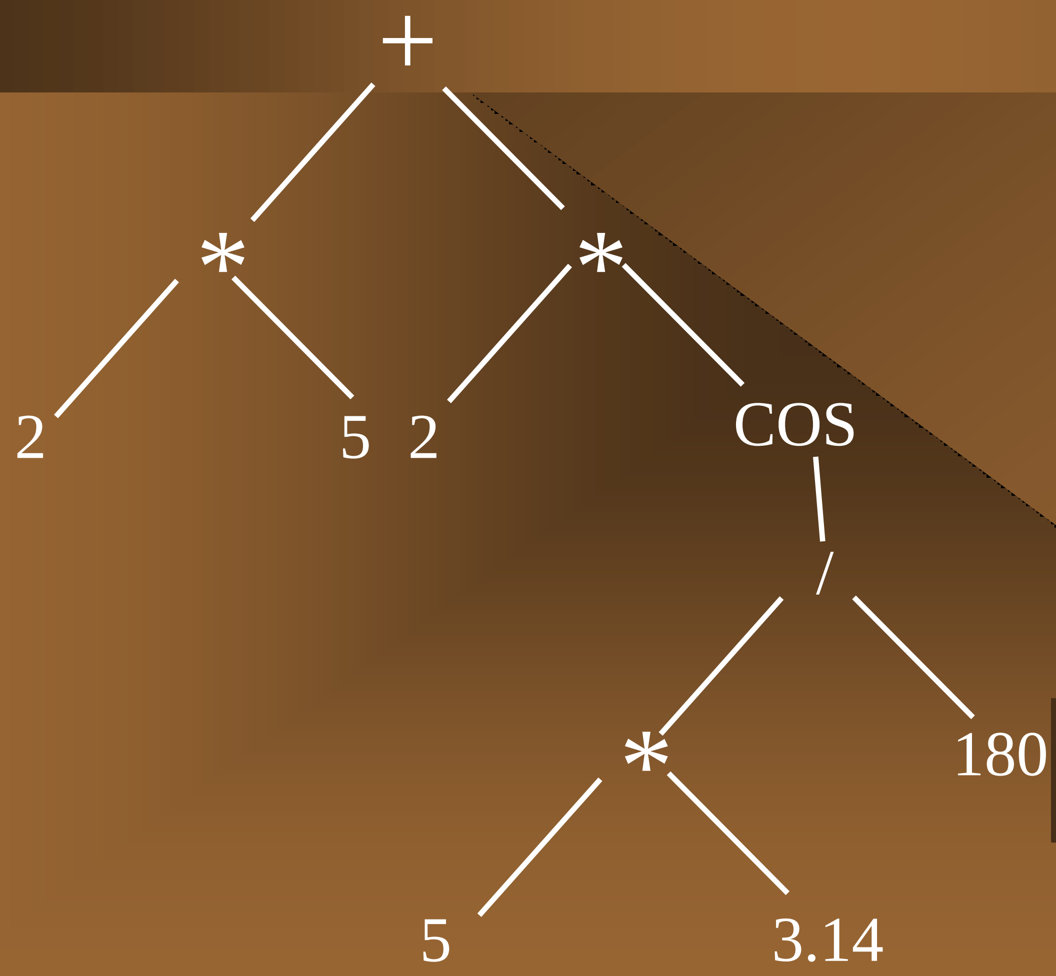
```
procedure infixe(entree p : pointeur)
Debut
      si p <> NULL
             alors    infixe(p → g)
                      afficher(p → val)
                      infixe(p → d)
             sinon    afficher(f)
      fin si
Fin
```

Résultat :

f 7 f 4 f 8 f 2 f 5 f 9 f 1 f 3 f 6 f 10 f

Les arbres

L'expression 2*5+2*(cos((5*3.14)/180)) peut se représenter de la façon suivante



Les graphes

- Un graphe est un ensemble de nœuds reliés par des liens.
- Ce n'est plus un arbre dès qu'il existe deux parcours différents pour aller d'au moins un nœud vers un autre.
- Un graphe est connexe, lorsqu'il est possible de trouver au moins un parcours permettant de relier les nœuds deux à deux (un arbre est un graphe connexe).
- Un graphe est dit pondéré lorsqu'à chaque lien est associé une valeur (appelée poids).

Les graphes

- On utilisera les graphes pondérés par exemple pour :
  - gérer des itinéraires routiers (quelle est la route la plus courte pour aller d'un nœud à un autre).
  - gérer des fluides (nœuds reliés par des tuyauteries de diamètre différents).
  - simuler un trafic routier.
  - simuler un circuit électrique.
  - prévoir un ordonnancement

- Un graphe est dit orienté lorsque les liens sont unidirectionnels.

Les graphes

- On peut représenter un graphe de manière dynamique, comme les arbres.

- Une autre solution consiste à numéroter les N sommets et d'utiliser une matrice carrée NxN, avec en ligne i et en colonne j un 0 si les nœuds i et j ne sont pas reliés, le poids de la liaison sinon.

- Un problème important est le parcours d'un graphe : il faut éviter les boucles infinies, c'est-à-dire retourner sur un nœud déjà visité et repartir dans la même direction.

  Pour cela, on utilise des indicateurs de passage (Booléen) ou une méthode à jeton.