Quand un problème se posera,
Calmement tu l'aborderas,
Proprement le définiras ;

Analyse tu t'imposeras
Avec rigueur la mèneras
Sans relâche persisteras
Et à coup sûr le résoudras.

Alors aisé coder sera,
En jouant tu le testeras.

Si malgré tout erreur il y a
Modularité tu loueras
Car à corriger t'aidera ;
Sur ces actions tu boucleras
Et satisfaction obtiendras.

Mes remerciements vont à Antoine Maillet et Véronique Elghour pour leur importante contribution à l'élaboration de ce document.

A L G O R I T H M I Q U E

1. Définition d'un projet informatique

Réaliser un **logiciel performant**, répondant aux **besoins exprimés** d'un client dans un **délai** donné.

Besoins exprimés	:	uniquement (cahier des charges).
Performant	:	dans la limite du nécessaire (adapté aux besoins).
Délai	:	impératif (pénalités de retard).

Un projet ne correspond en aucun cas à un travail facile à concevoir globalement, d'où la nécessité d'une **méthode de travail**. Elle est d'autant plus importante lorsque le volume de travail est grand et les délais courts. Dans ce cas, la mise en œuvre d'une équipe est nécessaire. Un chef de projet coordonne l'ensemble.

La communication : elle doit être omniprésente, qu'elle soit orale ou écrite.

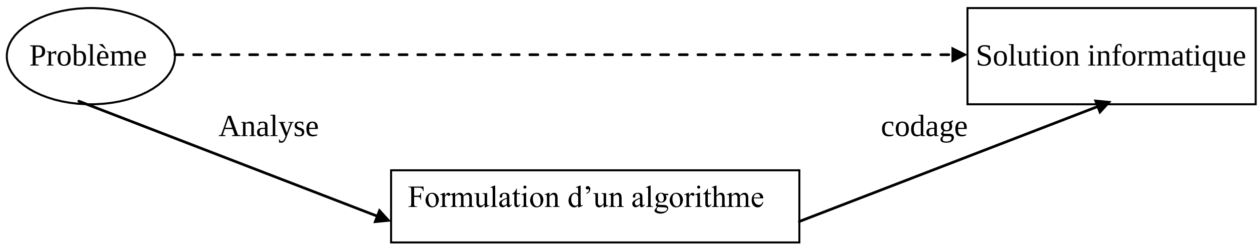
La communication écrite doit être consignée au niveau :

- du cahier des charges
- du dossier industriel (planning, tâches, ….)
- du dossier de fabrication (analyse, code, …)
- du manuel utilisateur.

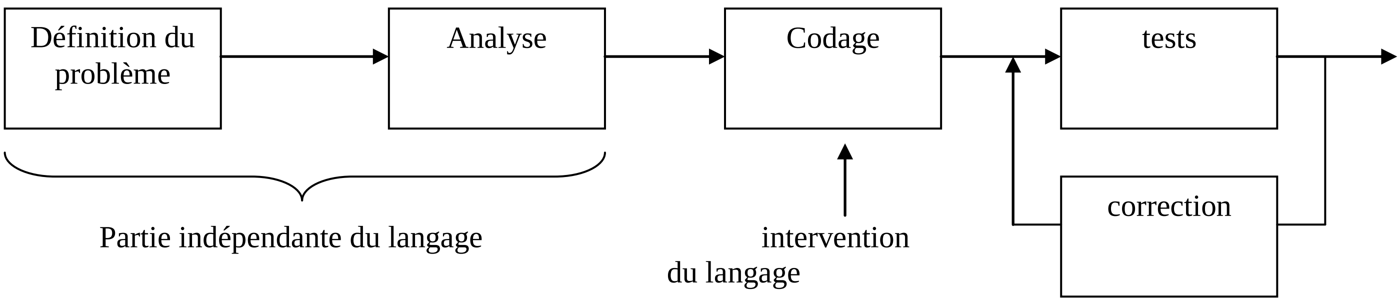
2. Description de la méthode.

Jusqu'en 1975, on pratiquait la « programmation sauvage », c'est à dire sans réflexion préalable.
Puis l'on s'est mis à réfléchir sur des méthodes de travail. Il en ressort plusieurs, mais toutes fonctionnent de la même façon dans les grandes lignes.

2.1. *Les étapes*



Les différentes étapes d'un travail méthodique conduisant à une solution informatique sont les suivantes.



Un problème correctement posé et analysé est quasiment résolu.

2.2. Définition du problème.

C'est la base d'une bonne analyse.
Avant de chercher une méthode de résolution, il faut définir :

- tout ce que l'on attend du programme (**résultats attendus**).
- Les éléments nécessaires à la résolution du problème (**les données**).
    - Où ira-ton les chercher ?
    - Sous quelle forme seront-ils ?
    - Est-on sûr de la validité de ces données ? (correspondent-elles à ce que l'on attendait ou faut-il les vérifier ?)
- la présentation des résultats (avoir un souci de convivialité, parfois rappeler les données).

2.3. Analyse

C'est une **étape fondamentale**. Elle doit aboutir à l'écriture d'un **algorithme structuré** en français.
Un algorithme est un processus qui, à partir de données, permet d'obtenir des résultats.
Ce processus est constitué d'une suite finie d'opérations à réaliser.
C'est un acte créatif, donc difficile, faisant appel à :

- la logique
- le raisonnement
- l'expérience
- l'intuition.

Un même problème peut conduire à des algorithmes différents, certains pouvant être plus rapides et/ou plus élégants que d'autres (multiplicité des opérations).

2.4. L'analyse descendante.

On part du niveau le plus élevé (définition du problème), puis on décompose le problème en plusieurs phases d'un niveau un peu moins complexe.
A ce niveau, on n'a aucune d'idée sur les **méthodes de résolution**, ni sur la **représentation physique** du problème. Seuls comptent l'identification et l'enchaînement des différentes opérations à effectuer.
A partir de chaque niveau défini, on réitère ce procédé jusqu'à obtenir des opérations de base du langage algorithmique.

2.4.1. Avantages

- C'est une démarche systématique. (on procède de la même façon à tous les niveaux).
- L'aspect hiérarchique se prête très bien au travail en équipe.

2.4.2. Inconvénients.

- On est amené à faire des choix de haut niveau dans les premières étapes, certains pouvant se révéler difficiles à réaliser quelques niveaux plus bas, d'où remise en cause d'un certain nombre de choix, ce qui oblige à reprendre l'application sur tous les niveaux intermédiaires.
- Génération d'algorithmes spécifiques semblables en différents points de l'application, pour résoudre des problèmes analogues.

2.5. L'analyse ascendante.

On dispose d'outils (algorithmes) réalisant des travaux élémentaires. On regroupe plusieurs d'entre eux pour en faire un outil un peu plus sophistiqué. Pour ce genre d'analyse, il est indispensable d'avoir à disposition une caisse à outils bien remplie et des informaticiens qui en connaissent parfaitement le contenu afin d'optimiser l'utilisation de ceux-ci. On réitère autant de fois que nécessaire.
Dans la pratique, on arrive rarement à mener cette méthode jusqu'aux niveaux les plus élevés, d'où une démarche **bi-directionnelle**.
Dans ce cas on décompose le problème en tâches moins complexes, et l'on procède de façon ascendante pour résoudre chacun des sous problèmes.

2.5.1. Avantages.

- on réutilise au maximum des outils déjà développés, ce qui augmente la vitesse de développement, réduit la taille de l'algorithme et simplifie les tests.
- obtention d'un prototype permettant d'évaluer le produit final, d'où possibilité :
  - d'une validation totale.
  - d'une remise en cause totale.
  - d'une remise en cause partielle (remplacement de séquences par d'autres plus performantes).

2.5.2. Inconvénients.

- Cette méthode ne procède pas d'une démarche systématique
- Risque de dérive vers la résolution d'un problème différent de celui qui était posé, ceci afin d'utiliser au maximum les outils existants.

➔ En conclusion, on en déduit qu'il est préférable d'utiliser une **démarche descendante**

2.6. Définition d'un pseudo-langage

Ce sont des structures de contrôle permettant d'orienter le déroulement de l'algorithme.
Il existe deux sortes de structures.

2.6.1. Les structures de choix

2.6.1.1. L'alternative

```
Si condition alors
        Action 1
Sinon
        Action 2
Finsi
```

2.6.1.2. Choix multiple (choix entre plus de 2 possibilités)

```
Décider selon expression entre
        Valeurs 1       :       action 1
        Valeurs 2       :       action 2
                .
        Valeurs n       :       action n
        Autres valeurs :        action 0
Fin-décider.
```

Remarque : les listes de valeurs doivent être disjointes.

2.6.2. Les structures de répétition

2.6.2.1. Tant que

Tant que condition faire
    Action 1 → éventuellement 0 itération
Fin tant que.

2.6.2.2. Répéter …. jusqu'à

Répéter
    Action → au moins 1 itération
Jusqu'à condition

2.6.2.3. Pour

Pour compteur allant de début à fin par pas de p, faire
    Action → nombre d'itérations défini à l'entrée dans la boucle
Fin pour

Remarques :

- si début < fin et pas <0

pas d'action.

- si début > fin et pas >0

pas d'action.

- si début = fin

1 action.

2.7. Utilisation des structures de données.

Au début, on mettait l'accent sur les traitements aux dépens des données.
Par la suite, les méthodes d'analyse et les langages se perfectionnant, ils ont donné à l'informaticien la possibilité de définir des données complexes (ou structurées), ce qui a rééquilibré les traitements par rapport aux données. Ceci améliore beaucoup la productivité des informaticiens.

2.7.1. Le langage procédural

Le stockage des données se fait au niveau de la case mémoire, d'où développement de séquences rendant celles-ci consistantes, c'est à dire, qui utilisent de manière cohérente plusieurs données élémentaires pour représenter une donnée complexe.

Par exemple, pour les langages d'assemblage, un tableau de caractères est à la charge du programmeur qui doit écrire les séquences simulant ce tableau, et des séquences permettant d'accéder à une donnée élémentaire du tableau.

Ceci présente les inconvénients suivants :

- charge de travail importante (problèmes à résoudre, consistance des données).
- Absence de portabilité des programmes.
- Grande vulnérabilité aux erreurs.
- Manque de lisibilité et difficultés pour maintenir les programmes.
- Impossibilité de paramétrer les programmes.

Tous ces inconvénients viennent du rôle mineur des données.

2.7.2. Les langages acceptant des données complexes.

Ils comportent des instructions permettant de définir des données structurées et des macro actions permettant de les manipuler.
Ainsi, l'informaticien est déchargé des problèmes de consistance des données. Il peut donc avoir une meilleure visibilité du problème posé.

Exemple :

Dans un texte, on veut repérer tous les séparateurs (ponctuation, espace).

Avec un langage purement procédural, on écrira :

```
Si caractere = espace alors
        Action
Sinon
        Si caractere = point alors
                Action
        Sinon
                Si caractere = virgule alors
                        Action
                Sinon
                …
```

Avec un langage autorisant le type ensemble (SET of en Pascal), on écrira :

```
Constante : séparateurs ← [ . , ; - ]
        Si caractere dans séparateur
                Action
        Finsi
```

L'informaticien a donc intérêt à prendre en compte les structures des données dès la phase d'analyse, car celles–ci peuvent avoir une forte incidence sur la formulation de l'algorithme .

Les structures des données ne doivent pas se limiter à celles prédéfinies dans les langages, il faut utiliser aussi les piles, files, graphes, …. ; d'où l'étude de ces différentes structures de données avec leurs primitives d'accès, c'est à dire les procédures standards de manipulation.

2.7.3. Structures de données statiques

C'est un ensemble d'informations élémentaires ayant un lien logique et dont le nombre est fixe. Si tous les éléments d'une telle structure sont de même type elle sera dite homogène, sinon elle sera dite hétérogène.

2.7.3.1. <u>Les homogènes</u>

C'est le tableau qui les représente.

Tableau à une dimension

Il porte le nom de vecteur ⇨ représentation informatique du vecteur.
Cette structure est composée d'un ensemble d'éléments accessibles directement en lecture et en écriture par l'intermédiaire d'un indice.

On peut le définir par :

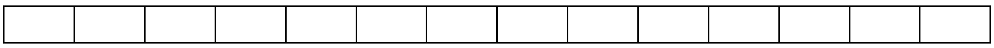
Type Nomdutype = Tableau [typeindice] de T

Type discret à bornes fixes

type de chacun
des éléments

Type dix = 1… 10
Type vecteur = Tableau [Dix] de réels
Type lettres = 'A'… 'Z'
Type vecteur bis = Tableau [Lettres] d'entiers

Représentation physique : zone mémoire contiguë



d

Un élément du tableau occupe p cases mémoire ⇨ c.p cases mémoires
L'indice peut prendre c valeurs différentes
Adresse de début du tableau : d
L'élément de rang i du tableau T, i.e. T (i) si l'indice démarre à 1, sera à l'adresse :

$$d + (i-l) * p$$

Tableau multidimensionnel

C'est l'extension des tableaux à une seule dimension utilisés pour représenter des données plus complexes telles que matrices, tenseurs…

Ils sont très utilisés :
- pour représenter une grandeur physique dans l'espace
- pour représenter des données lors de dépouillement multicritère en statistiques

T (i, j, k) → nombre d'individus ayant répondu à la 1ère question par la modalité i, à la 2ème par la modalité j et à la 3ème par la modalité k.

C'est le mode d'adressage des éléments qui diffère. Un élément est repéré par autant d'indices que le tableau a de dimensions.
Type Dix = 1… 10
Type lettres = 'A'… 'Z'
Type vecteur = Tableau [Dix] de réels
Type matrice = Tableau [Lettres] de vecteur

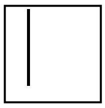
Cette définition du type matrice est équivalente à laquelle des deux définitions suivantes :

Type matrice = Tableau [Dix, Lettres] de réels → ordre descendant des indices
Type matrice = Tableau [Lettres, Dix] de réels → ordre ascendant des indices

Cela dépend des langages.

Tableau [Dix, Lettres] de réels

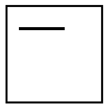
Lettres

Dix  Rangement en colonne

Comme le Fortran

Tableau [Lettres, Dix] de réels

Dix

Lettres  Rangement en ligne

Comme le Pascal ou le C

<u>Représentation physique : zone mémoire contiguë</u>

<u>*Tableau à k dimensions :*</u>

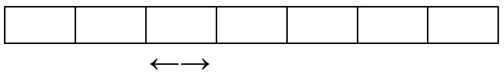


Tableau à k-l dimensions

Tableau à k dimensions

<u>*Tableau à 2 dimensions :*</u>

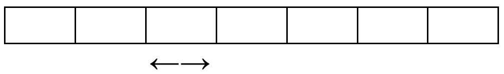


Tableau à l dimension = une ligne en C

Adresse d'un élément d'indice (i, j) : d + [(i-l) $*$ ncolonnes + (j-l)] $*$ p

<u>*Tableau à 3 dimensions :*</u>

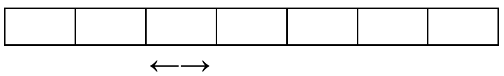


Tableau à 2 dimensions

Tableau à 3 dimensions

Adresse d'un élément d'indice (i, j, k) : d + [(i-l) $*$ ncolonnes $*$ ncotes + (j-l) $*$ ncotes + (k-l)] $*$ p

Tableau à k dimensions : $(r_1, r_2, r_3,\ldots r_k)$

$d + [(r_1 - l) \times c_2 \times c_3\ldots \times c_k + (r_2 - l) \times c_3 \times c_4\ldots c_k + \ldots + (r_{k-1} - l) \times c_r + (r_k - l)] \times p$
avec c_i = nombre de valeurs possibles pour l'indice i.

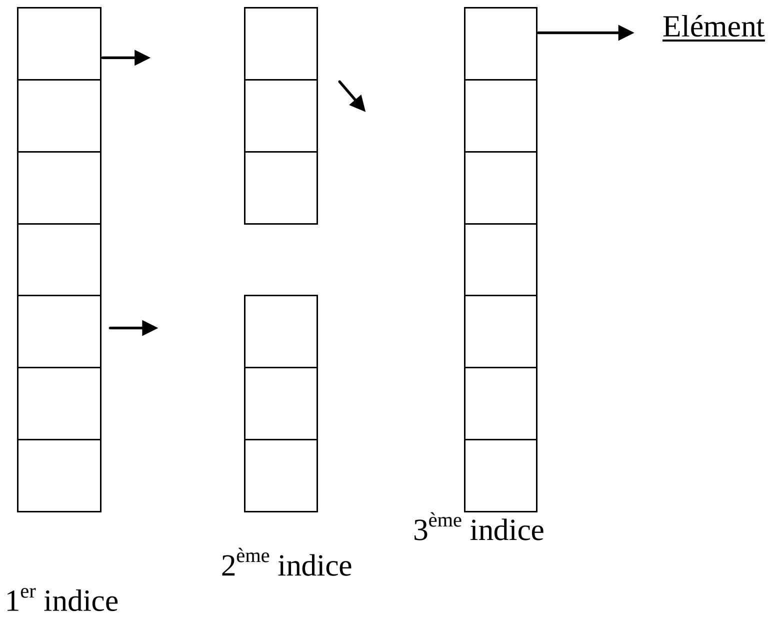
Tableau T à une dimension très grand

$T [f(r, r_2,\ldots r_k)]$

Représentation physique :

- zone mémoire contiguë
- chaînée

On utilise une hiérarchie de tableaux de pointeurs.



2.7.3.2. Les hétérogènes

Elle comporte un nombre fixe d'informations de types différents.
Structure très riche représentée par l'enregistrement.
Ces informations sont accessibles soit une à une, soit globalement, en lecture et en écriture.
Pour un certain nombre d'organismes, un individu est représenté par :

son nom : chaîne de 30 caractères
son prénom : chaîne de 20 caractères
son numéro de sécurité social : entier
son adresse : chaîne de 50 caractères
son numéro de téléphone : entier
son âge : entier

Si on veut accéder à la rubrique nomrubrequei d'une variable x de type nomdutype, on utilise la notation x.nomrubrequei

toto.numtel → n° de téléphone de toto

Type Nomdutype = enregistrement

Nomrubrique1 : type 1

Nomrubrique2 : type 2

Nomrubriquen : type n

Type individu = enregistrement

Nom : chaîne de 30 caractères
Prénom : chaîne de 30 caractères
Numéro de sécurité social : entier
Adresse : chaîne de 50 caractères
Numéro de téléphone : entier

<u>Exemple</u> :

Type client = enregistrement
Nom : tableau [l… 30] de caractères
Prénom : chaîne de 20 caractères
Adresse : chaîne de 50 caractères
Anniversaire : entier
Compte : comptebancaire
Fin

Représentation physique : zone mémoire contiguë

Type Comptebancaire = enregistrement
Compte chèque : cpt
PEL : cpt
CEL : cpt
PEA : cpt
Fin

Type Cpt = enregistrement
Numéro : entier
Solde : réel
Fin

Var toto : client
Toto.compte.PEA.solde

Représente alors le solde du PEA du client Toto

<u>Les enregistrements à champs variables</u>

Pour définir un même objet global, on peut être amené à considérer différentes versions de la structure en fonction de la valeur prise par une des rubriques, c'est à dire la présence de telles ou telles autres rubriques.

Par exemple, on définit un enregistrement prenant en compte la situation familiale d'un individu.

Type situation familiale = (célibataire, marié, divorcé, veuf)
Type personne = enregistrement
    Nom : chaîne de 30 caractères
    Enfants : entier
    Selon situation : situation familiale vaut :
        Célibataire : (vie maritale : booléen)
        Marié : (date mariage : entier, département mariage : entier)
        Divorcé : (date mariage : entier, date divorce : entier, garde enfants : booléen)
        Veuf : (date mariage : entier, date décès conjoint : entier, âge : entier)
    Fin selon
Fin Type

Enregistrement avec une partie fixe à 3 rubriques (Nom, nombre d'enfants, situation) et une partie variable qui est fonction de la rubrique situation.

Dans certains langages, la partie variable est obligatoirement à la fin de la structure.

Représentation physique :
La place mémoire d'une variable de ce type correspond à l'encombrement maximum, i.e. à la version la plus longue de la structure, pour respecter l'aspect statique de la structure.
Dans l'exemple, c'est le cas veuf qui est le plus long, car ce sont 3 entiers qui prennent plus de place que le cas divorcé (2 entiers + 1 booléen).

Les ensembles

Cette structure a des réalisations très différentes selon les langages. On utilisera la représentation faite par le langage Pascal : les caractéristiques, les limitations, les représentations physiques peuvent être très différentes dans d'autres langages.

Le type ensemble est la représentation informatique de l'objet mathématique ensemble. Un ensemble est une collection non ordonnée d'éléments sur lesquels on peut effectuer les opérations classiques, telles que intersection, réunion…

Un type ensemble particulier est défini à partir du type de base T de ses éléments.
→ Type ensemble = ensemble de T (avec T discret et pas continu).

Une variable de type ensemble peut prendre comme valeurs toutes les parties du type de base. Si le type de base a n éléments, il y a 2^n parties de ce type de base.
→ Type couleur = (bleu, jaune, vert)
    Type palette = ensemble de couleur
    Variable x : palette
    →8 valeurs : [], [bleu], [jaune], [vert], [bleu, jaune], [bleu, vert], [jaune, vert], [bleu, jaune, vert]

Les opérateurs

< = : inclusion > = : contenance = : égalité < > : inégalité	+ : réunion * : intersection - : différence dans : appartenance	Ils sont tous binaires ayant des opérandes de type ensemble, sauf "dans" dont l'opérande gauche est du type de base de l'ensemble et l'opérande droit du type ensemble.

→ Type qualité = (actif, intelligent, habile, astucieux, patient)
Type personnalité = ensemble de qualité
Var Alain, Jean : personnalité
Var X : qualité

Autorisé	Interdit
Alain ← [actif]	Alain ← actif
X ← actif	X ← [actif]
Jean ← Alain + [patient]	Jean ← Alain + X
Si Jean > = Alain alors	Si Jean > = X alors
Tant que Alain < = Jean faire	Tant que X < = Jean faire

→ Représentation physique

Le Pascal suppose que le nombre d'éléments du type de base est limité et que ce dernier est muni d'une relation d'ordre. La représentation d'une variable de type ensemble sera faite sous forme d'un tableau contigu, la présence de chaque élément est réalisée (codée) sur un seul bit (1 si l'élément est présent, 0 sinon) et le rang du bit dans le tableau permet de repérer l'élément grâce à la relation d'ordre existant sur le type de base.

Alain ← [actif]
Jean ← Alain +[patient]

Le codage de la variable Jean sera :

1 bit
←——→

1	0	0	0	1
actif	intelligent	habile	astucieux	patient

2.7.4. Structures de données dynamiques

2.7.4.1. Les besoins

Les structures de données statiques peuvent contenir un nombre fixe d'informations et sont utilisées pour représenter des structures ayant un nombre d'éléments connu à l'avance et ne pouvant pas évoluer.

Mais on a souvent à représenter des structures dont le nombre d'éléments n'est pas connu a priori et/ou le nombre d'éléments varie dans le temps. On utilise alors des structures de données dynamiques.

→ On doit lire des valeurs numériques dans un fichier pour faire un traitement sur ces données. On ne sait pas combien il y a de valeurs et il n'est pas question de les compter auparavant. Les structures de données utilisables sont :
  - un tableau "surdimensionné" avec le risque qu'il soit quand même trop petit si on a sous estimé le nombre d'éléments à lire, ou qu'il soit vraiment "trop grand", ce qui génère un gâchis de place mémoire (← pas bon).
  - une structure de données dynamiques, qui va grandir au fur et à mesure de la lecture des données (← bon).

→ On veut modéliser la file d'attente à un guichet d'une gare parisienne. Du fait des variations, les structures de données dynamiques s'imposent.

2.7.4.2. Représentation physique

Le principe de base est de suivre les évolutions de la structure en lui attribuant de la place mémoire quand elle en a besoin et en la récupérant quand le besoin disparaît.

On utilise des mécanismes d'attribution et de récupération de place mémoire. Ils utilisent une zone particulière de la mémoire, appelée Tas (Heap), dans laquelle ils réservent de la place quand la structure grandit et la récupèrent quand elle diminue. Dans les langages évolués, il existe 2 procédures standards de réservation/libération d'espace mémoire, appelées ici Réserve et Libère. L'informaticien doit donc prévoir les appels à ces procédures quand ses structures de données évoluent en taille, mais il est déchargé de la gestion effective de la place mémoire. Ces procédures utilisent exclusivement des pointeurs :

si P est un pointeur sur un objet de type T, sa définition se fera par :
var P: ↑T

Réserve (P) permet de réserver une place mémoire permettant de stocker la valeur d'un objet de type T dans le Tas et de faire pointer P sur cet emplacement. Pour accéder au contenu de la zone mémoire pointée par P, on utilise la notation P↑.
Libère (P) va libérer l'emplacement pointé par P, qui prendra la valeur rien.

→ On va représenter les données lues sur le fichier sous forme d'une liste chaînée en mémoire.

On peut utiliser la définition suivante du type Poste :

```
Type poste = enregistrement
        Info : réel
        Suivant : ↑Poste
    fintype

ou encore mieux :
Type lien = ↑Poste
    Type poste = enregistrement
        Info : réel
        Suivant : lien
    fintype
```

Les arguments peuvent être de 3 natures différentes :

- en entrée : donnée pour algorithme, sa valeur ne sera pas modifiée
- en sortie : pas de valeur en entrée, mais valeur en sortie
- en entrée/sortie : à la fois donnée et résultat, donc sa valeur peut être modifiée par l'algorithme

Notation : les arguments en entrée figureront toujours en tête, séparés des autres par un point-virgule.

```
lecture (; premier)
début
    premier ← rien
    si fichier pas vide alors
    réserve (premier)
        lire X
        premier ↑. info ← X
        premier ↑. suivant ← rien
        P ← premier
        tant qu'il y a encore des données faire
            lire X
            réserve (P↑. suivant)
            P ← P↑. suivant
            P↑. info ← X
            P↑. suivant ← rien
        fintantque
    finsi
fin
```

2.7.4.3. <u>Piles et files</u>

On a affaire à des collections d'objets ou d'individus dont le nombre varie dans le temps à la suite d'ajouts ou de retraits successifs et organisés de la façon suivante :

- On sait repérer soit l'une, soit l'autre, soit les deux extrémités de la structure.
- Les éléments de celle-ci sont ordonnés et on peut définir le suivant et le précédent de tout élément qui n'est pas une extrémité.
- Pour accéder à un élément particulier, on doit obligatoirement parcourir séquentiellement tous les éléments de la structure à partir d'une extrémité jusqu'à ce qu'on le trouve.

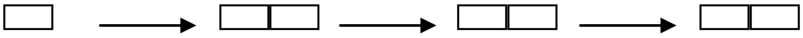
Selon le mode d'exploitation d'une telle suite d'éléments, on distingue différentes structures de données :

- les piles
- les files d'attente
- les files séquentielles

a) Les piles

Cela correspond à la notion de pile d'assiettes. Sa caractéristique essentielle réside dans le fait qu'on retire toujours en premier le dernier élément ajouté. L'ordre de sortie est l'inverse de l'ordre d'entrée (gestion LIFO → Last In First Out).
En informatique, lors d'appels en cascade de sous-programmes, c'est toujours le dernier appelé qui est achevé d'exécuter en premier.
Une pile est une structure de données dynamique homogène à un seul point d'accès en entrée et en sortie (le même) → c'est le sommet.

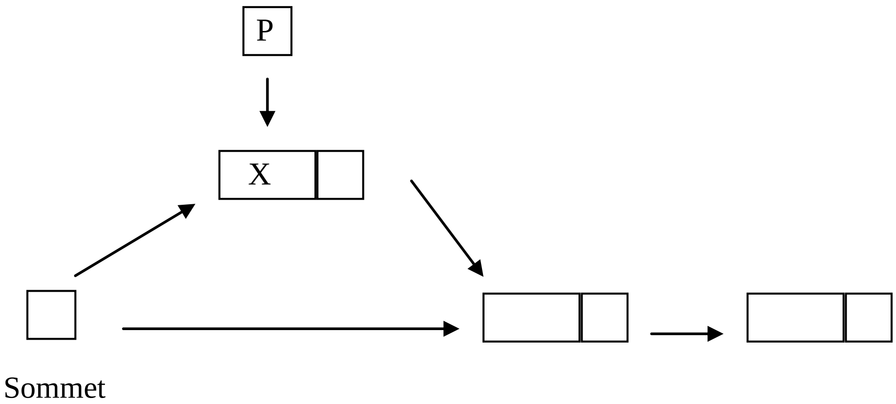
Représentation physique classique : sous forme d'une liste chaînée.



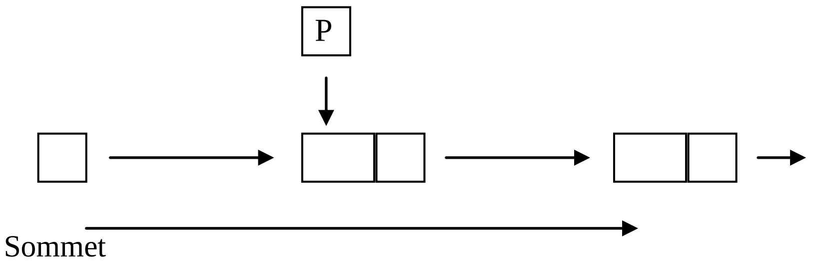
Type lien = ↑ Poste

Type Poste = enregistrement
| Info : T
| Suivant : Lien
fin
Valeur initiale de sommet : rien
Vide (sommet;)
début
| Vide ← sommet = rien (si vérifié → vrai/sinon → faux)
fin

Empiler (X ; sommet)
début
| Réserve (P)
| P↑. suivant ← sommet
| P↑. info ← X
| Sommet ← P
fin



```
Dépiler (; X , sommet)
début
  Si NonVide (sommet) alors
    X ← Sommet ↑. Info
    P ← Sommet
    Sommet ←  Sommet ↑. suivant
    Libère (P)
  finsi
fin
```



Autre représentation : tableau avec sommet indice du tableau (→ entier).

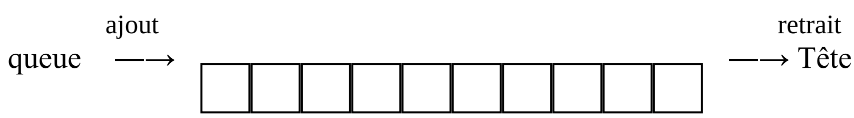
Moins performant car la taille du tableau est fixe et il faut introduire un test de débordement dans Empiler.

On peut avoir à gérer 2 piles d'éléments de même type non susceptibles d'être maximales en même temps : tableau dans lequel les piles sont représentées "tête bêche".

Le défaut est que le test de saturation se complique ; de plus on est dans l'impossibilité de généraliser cette représentation à plus de 2 piles.

b) <u>Les files d'attente</u>

C'est comparable aux files d'attente de clients à un guichet. En informatique, c'est le traitement de programmes dans un système de traitement par lots. A partir d'exemples, on déduit les caractéristiques d'une telle structure, à savoir une structure de données dynamique homogène à 2 points d'accès, l'un en lecture où on enlèvera des éléments, appelé tête, et l'autre en écriture où on ajoutera les éléments, appelé queue.

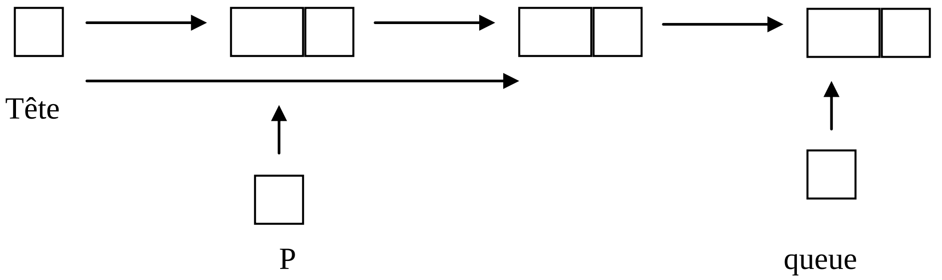


Le mode de gestion d'une telle structure est de type FIFO : First In First Out

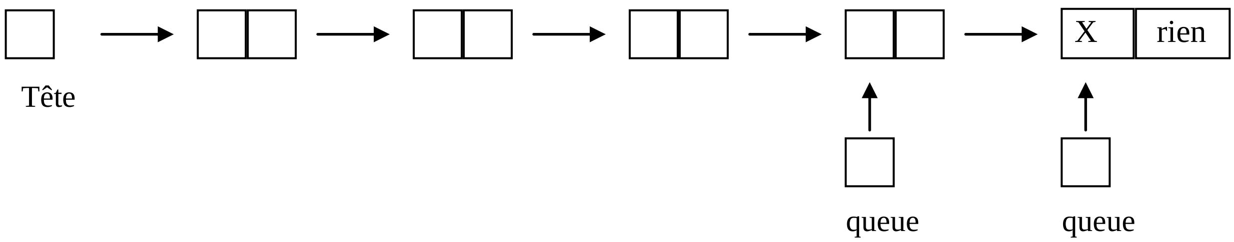
Représentation physique

- Liste chaînée par pointeur

```
Type Lien = ↑Poste

Type Poste = enregistrement
  Info : T
  Suivant : lien
fin


Type File d'attente = enregistrement
  Tête : lien
  Queue : lien
fin

var F : file d'attente


Vide (F)
début
  Vide ← F. Tête = rien
fin


Prendrefile (; X, F)
début
  Si NonVide (F) alors
    X ← F. Tête ↑. Info
    P ← F. Tête
    F. Tête ← F. Tête ↑. suivant
    Libère (P)
    Si F. Tête = rien alors
      F. Queue ← rien
    finsi
  finsi
fin
```



```
Mettre (X ; F)
début
  Si NonVide (F) alors
    Réserve (F. queue ↑. suivant)
    F. queue ← F. queue ↑. suivant
    F. queue ↑. Info ← X
    F. queue ↑. suivant ← rien
  Sinon
    Réserve (F. queue)
    F. Tête ← F. queue
    F. queue ↑. Info ← X
    F. queue ↑. suivant ← rien
finsi
fin
```



Représentation sous forme d'un tableau : Tête et Queue sont des indices du tableau.

La taille du tableau est limitée et donc il faut se prémunir d'un débordement éventuel. Suite à des ajouts, il pourrait y avoir plus d'éléments dans la file d'attente qu'il n'y a de cases dans le tableau. A la suite d'ajouts et de retraits successifs, les éléments dans la file d'attente montent inexorablement dans le tableau et il arrive un moment où il n'y a plus de place en haut du tableau, alors qu'il y en a en bas. Il y a plusieurs solutions à ces problèmes.

1) A chaque ajout d'un élément, on décale vers le bas tous les éléments situés dans la file d'attente afin de récupérer les emplacements libres en mettant à jour les indices Tête et Queue, mais cela est fréquent et coûteux en temps.
2) Lorsque, suite à un ajout, l'indice de Queue se trouve en haut du tableau, on procède au même décalage, mais seulement dans ce cas et du coup, le temps d'ajout d'un élément n'est plus constant.
3) On considère le tableau comme circulaire, c'est à dire quand l'indice de Queue atteint le haut du tableau, on ajoute les éléments en bas du tableau. Dans ce cas, le test de saturation correspond à une file vide ou saturée, d'où l'ajout d'un compteur des éléments dans la file.

```
Type Filedattente = enregistrement
  Tab : tableau [0…n-1] de T
  Tête : 0… n-1
  Queue : 0… n-1
  Nbéléments : 0… n
fin
```

Var F : Filedattente

<u>Remarque</u> : valeurs initiales : nbéléments à 0

Tête et Queue doivent avoir la même valeur, quelconque entre 0 et (n-1).

```
Vide (F)
début
|   Vide ← F. nbéléments = 0
fin
```

```
Plein (F)
début
|   Plein ← F. nbéléments = n
fin
```

```
Mettre File (X ; F)
début
|  Si NonPlein (F) alors
|  |  F. Tab [F. Queue] ← X
|  |  F. Queue ← F. Queue + 1 mod n
|  |  F. nbéléments ← F. nbéléments + 1
|  finsi
fin
```

```
Prendre File (; X, F)
début
|  Si NonVide (F) alors
|  |  X ← F. Tab [F. Tête]
|  |  F. Tête ← F. Tête + 1 mod n
|  |  F. nbéléments ← F. nbéléments – 1
|  finsi
fin
```

c) <u>Les files séquentielles</u>

Les exemples les plus courants sont les bandes magnétiques, les convoyeurs de carrosserie automobile dans un atelier de peinture. Une file séquentielle est une suite d'éléments dont on sait repérer le début et pour laquelle on dispose d'une station de traitement (ou tête d'accès) se déplaçant le long de la structure.

Les opérations habituellement faites sont :
- mettre la tête d'accès au début de la file séquentielle,
- lire ou écrire un élément dans la position (case) se trouvant devant la tête d'accès puis déplacer celle-ci devant la case suivante.

Type FileSéquentielle = File de T

→ Primitive d'accès :

Premier (F) : positionner la tête d'accès au début de la structure

Fin (F) : fonction qui vaudra "vrai" si la tête d'accès est située à la fin de la structure, sinon faux

PrendreFile (X, F) : recopier le contenu de l'élément situé devant la tête d'accès dans X, puis déplacer celle-ci d'une case en avant

MettreFile (X, F) : copier la valeur de X dans l'élément situé devant la tête d'accès, puis déplacer celle-ci d'une case en avant et y mettre la marque de fin de structure (Findefile)

On peut lire autant d'éléments qu'on veut dans une file séquentielle, mais l'écriture d'un élément dans la file provoque l'écriture de "Findefile" immédiatement après et ainsi tous les éléments de la file situés derrière deviennent inaccessibles. Si on veut modifier un élément sans perdre les autres, il faut dupliquer la file.

Dans une file F, on veut remplacer tous les éléments de valeur A par la valeur B :

```
Modifier (F, A, B ; G)
début
  | Premier (F)
  | Premier (G)
  | Tant que NonFin (F) faire
  |   | PrendreFile (X ; F)
  |   | Si X = A
  |   |   | X ← B
  |   |
  |   | finsi
  |   | Mettre (X, G)
  | fintantque
fin
```

Parfois, on fait une représentation en mémoire d'une file séquentielle, soit par tableau, soit par pointeur.

<u>Par tableau</u> :

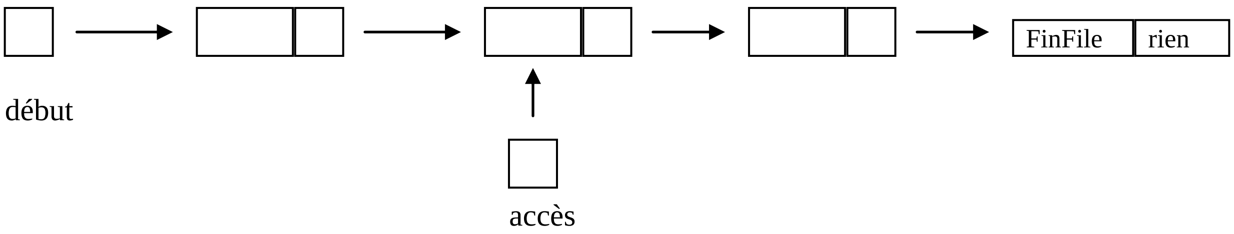
```
Type Fileséquentielle = enregistrement
  | Tab : Tableau [1… n] de T
  | Accés : 1… n
fin
```

Var F : File séquentielle

```
Premier (;F)
début
  | F. Accès ← 1
fin
```

```
Fin (F)
début
    Fin ← F. Tab [F. Accès] = "FindeFile"
    fin


PrendreFile (; X, F)
    début
    Si NonFin (F) alors
        X ← F. Tab [F. accès]
        F. accès ← F. accès + 1
    finsi
fin


Mettre (X ; F)
début
    Si F. accès < n alors
        F. Tab [F. accès] ← X
        F. accès ← F. accès + 1
        F. Tab [F. accès] ←"Findefile"
    finsi
fin
```

<u>Par pointeur</u> :




```
Type lien = ↑ Poste

Type Poste = enregistrement
    Info : T
    Suivant : lien
fin


Type Filséquentielle = enregistrement
    début : lien
    accès : lien
fin
```

```
var F : Fileséquentielle

Premier ( ; F)
début
  | F. accès ← F. début
fin

Fin (F)
début
  | Fin ← F. accès ↑. Info = "FindeFile"
fin

Prendre File ( ; X, F)
début
  | Si NonFin (F) alors
  |   | X ← F. accès ↑. Info
  |   | F. accès ← F. accès ↑. suivant
  | finsi
fin

Mettre File (X ; F)
début
  | Si Fin (F) alors
  |   | Réserve (F. accès ↑. suivant)
  |   | F. accès ↑. Info ← X
  |   | F. accès ← F. accès ↑. suivant
  |   | F. accès ↑. Info ← "FindeFile"
  |   | F. accès ↑. suivant ← rien
  | Sinon
  |   | F. accès ↑. Info ← X
  |   | P ← F. accès ↑. suivant
  |   | tant que P↑. Info ≠ "FindeFile" faire
  |   |   | Q ← P↑. suivant
  |   |   | Libère (P)
  |   |   | P ← Q
  |   | fintantque
  |   | F. accès ↑. suivant ← P
  |   | F. accès ← P
  | finsi
fin
```

3. La mesure de l'efficacité d'un algorithme.

Deux algorithmes résolvant le même problème peuvent différer par :
- la méthode de résolution (directement liée au problème à résoudre)
- la complexité apparente
- l'efficacité.

3.1. La complexité apparente

C'est la difficulté de lecture d'un algorithme. Elle est liée au nombre de structures de contrôle et à leur profondeur d'imbrication.

3.2. L'efficacité

Elle correspond à la plus ou moins bonne utilisation des ressources de la machine lors de l'exécution d'un algorithme :
- temps : complexité temporelle(liée au nombre d'opérations faites par l'algorithme).
- Place mémoire : complexité spatiale (liée à l'encombrement mémoire nécessaire pour exécuter l'algorithme).

3.3. La complexité pratique

C'est la mesure précise en heure/minutes/secondes du temps d'exécution de l'algorithme et la mesure précise en méga-octets de la place mémoire nécessaire.
Cette mesure présente l'inconvénient d'être subjective, car elle dépend des conditions pratiques d'exécution de l'algorithme sur la machine.

3.4. La complexité théorique

C'est un ordre de grandeur des coûts indépendant des conditions pratiques d'exécution.

Exemple :

trouver le plus grand diviseur d'un entier N > 1

3.4.1. Méthode 1

essayer tous les entiers successifs à partir de N-1 dans l'ordre décroissant.

```
PGD 1 (N)
        Début

        diviseur ← N-1

        Tant que n modulo diviseur <> 0, faire
                diviseur ← diviseur – 1
        Fin tant que

        PGD1 ← diviseur

        Fin
```

Nombre maximum d'opération : N-2
La complexité de l'algorithme est de l'ordre de N

3.4.2. Méthode 2.

Essayer tous les entiers successifs à partir de 2 dans l'ordre croissant jusqu'à **√ N**

PGD 2 (N)
Début

diviseur ← 2

Tant que **n modulo diviseur** <> **0** et **(diviseur < √ N)** , faire
diviseur ← diviseur + 1
Fin tant que

Si n modulo diviseur = 0
PGD2 ← n /diviseur
Sinon
PGD2 ← 1
Finsi

Fin

Nombre maximum d'opération : √ N-2
La complexité de l'algorithme est de l'ordre de √ N

Définition : Si dans un algorithme A, il existe un paramètre n caractérisant la taille des données, si le temps d'exécution est T(n) et s'il existe une fonction f(n) telle que T(n) / f(n) < constante (c'est à dire est bornée), alors on dit que A est de complexité O(f(n)).

La **complexité minimale** : c'est celle qui correspond au cas le plus favorable (par exemple tri de données déjà ordonnées).

La **complexité maximale** : c'est celle qui correspond au cas le plus défavorable.

La **complexité moyenne** : c'est celle qui correspond au cas moyen (données quelconques équiprobables dans un tri).

4. Méthode de recherche d'algorithme efficace

Il existe 2 principes généraux.

4.1. Le principe de division

On ramène un problème de taille N à un problème de taille M < N avec un certain coût et plus généralement, on ramène un problème de taille N à K problèmes de tailles respectives M_1, M_2, M_3, …, M_k telles que $M_1 + M_2 + M_3 + \ldots + M_k < N$, avec un certain coût. On recommence la même opération sur chacun des sous-problèmes résultants.

4.2. Le principe d'équilibrage

Toute les méthodes de division ne donnant pas le même résultat, on a intérêt à ce que les tailles des K problèmes résultants soient les plus proches possible.

5. La récursivité

En informatique, la récursivité est une procédure qui s'appelle elle-même.
Une image récursive : image comportant plusieurs éléments dont l'un est l'image elle même.

Définition récursive : définition comportant une référence à elle-même.
Exemple : un arbre est constitué d'une racine et de sous-arbres qui sont eux-même des arbres.

Un algorithme récursif est un algorithme qui s'appelle lui-même. → **récursivité directe**.
Un algorithme qui en appelle un second qui rappelle le premier → **récursivité indirecte ou croisée**.

Ce type d'algorithme impose une clause d'arrêt. C'est un cas trivial ou élémentaire où l'on fera une évaluation directe du résultat c'est à dire sans appel récursif.

5.1. Réalisation de récursivité

Procédure P(a,b)

```
début
    ⋮
    P(x,y)
    ⋮
fin
```

on recommence l'exécution de la procédure avec des données nouvelles avant d'avoir fini l'exécution avec les données précédentes
→ à chaque appel, empilement des différentes valeurs
→ dépilement des éléments au retour de chaque appel

5.2. Récursivité et efficacité

L'utilisation de la récursivité conduit à des méthodes puissantes et élégantes (macro efficacité), par contre cela génère une grosse perte de temps pour l'empilement et le désempilement. On utilise donc peu la programmation récursive.
Un raisonnement récursif se traduit en programmation par une dérécursivation de l'algorithme, c'est à dire que l'on en fait une traduction itérative.

6. Les méthodes de tri

6.1. Généralités

Trier, c'est ordonner un ensemble d'éléments en fonction de clés sur lesquelles est définie une relation d'ordre.

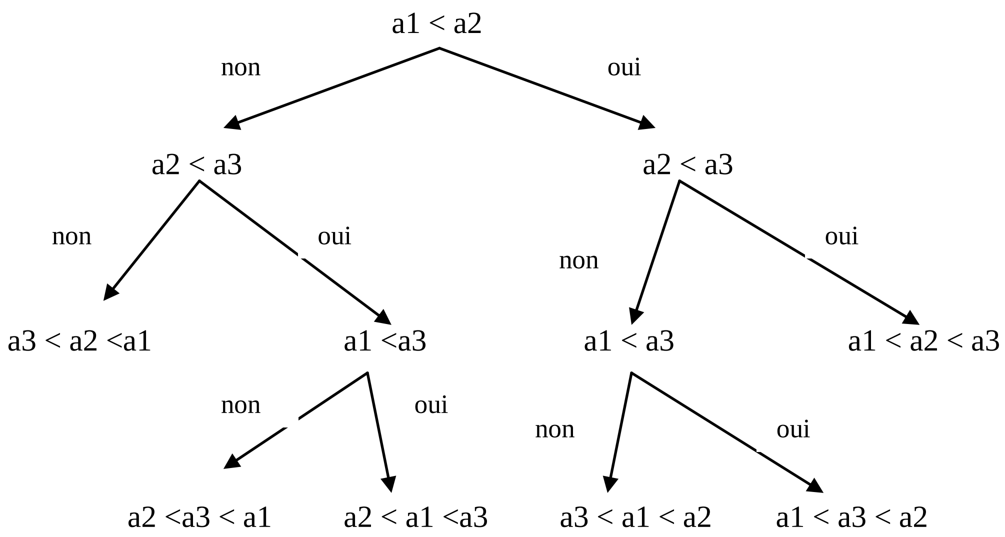
Il existe deux types de tri :

- **le tri interne :** tri d'éléments en mémoire centrale (tableau). Le temps d'accès aux éléments est très faible ; on s'attachera à minimiser le nombre de comparaisons et le nombre de permutations.
- **le tri externe :** tri d'éléments se trouvant sur une mémoire secondaire (fichiers). Là, le temps d'accès aux éléments est prépondérant, on s'attachera donc à minimiser le nombre d'accès à la mémoire secondaire.

Les 3 niveaux de complexité d'un algorithme de tri :

- **complexité minimale** : correspond au cas le plus favorable. Il s'agit d'un tri d'éléments déjà ordonnés. On aura donc au moins n-1 comparaisons, donc une complexité minimale au moins o (n)
- **Complexité maximale** : c'est le cas le plus défavorable

pour n = 3 on a les éléments a1,a2, a3

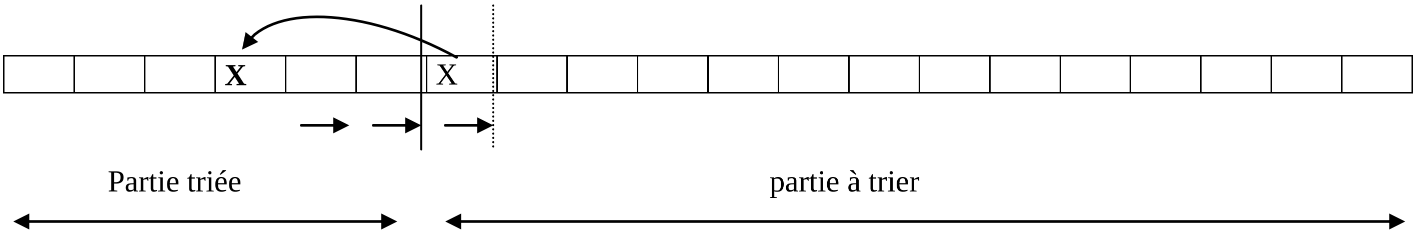


On obtient un arbre à 6 feuilles pour 3 éléments. Toutes les permutations des n éléments sont possibles, on a donc (n!) feuilles.
Le cas le plus défavorable est lorsque le nombre de comparaisons = hauteur de l'arbre (comparaisons).
Hauteur de l'arbre >= log(n !) c'est à dire de l'ordre de n.logn, donc une complexité maximale au moins en o (n.logn).

- **Complexité moyenne** : au moins O(n.logn) comparaisons.

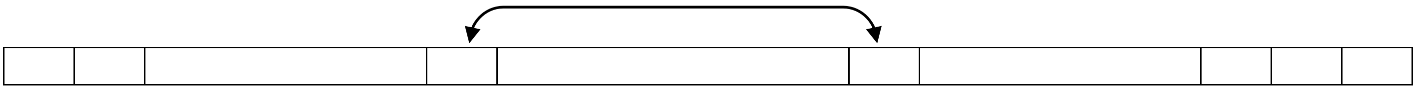
6.2. Les familles d'algorithmes de tri (pour les tris internes)

6.2.1. Tri par insertion



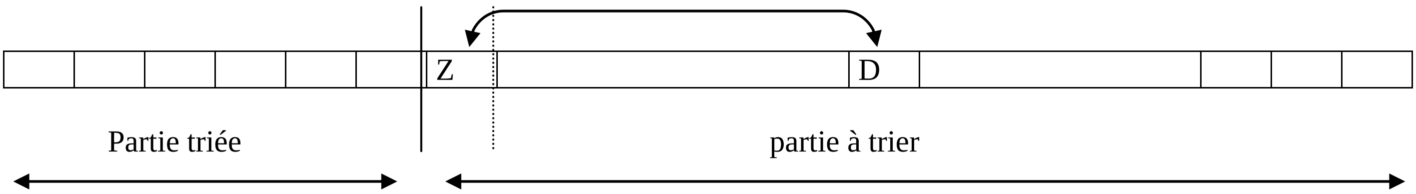
L'opération de base consiste à insérer l'élément frontière à sa place dans la partie triée, puis à déplacer la frontière d'une case vers la droite.
Cette méthode « opère sur place » (pas de tableau auxiliaire).

6.2.2. Tri par échange



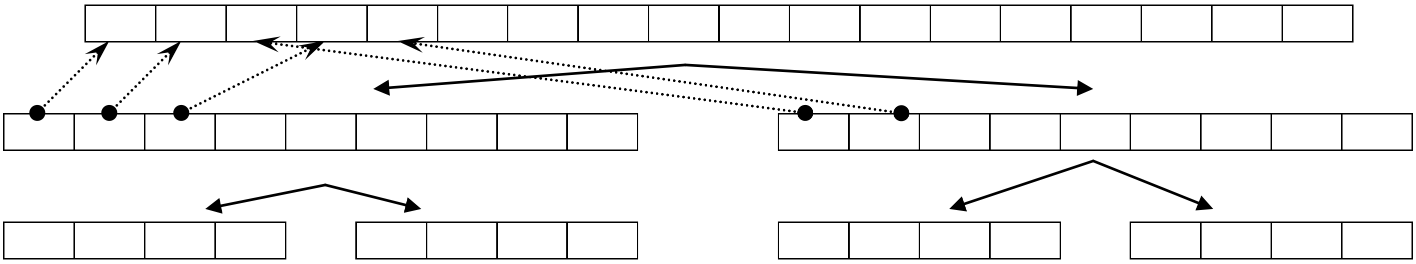
L'opération de base consiste à échanger des couples d'éléments mal classés tant qu'il y en a.
Cette méthode « opère sur place » (pas de tableau auxiliaire).

6.2.3. Tri par extraction



L'opération de base consiste à chercher l'élément extrêmal de la partie non triée, puis à le permuter avec l'élément frontière. On déplace ensuite la frontière d'une case vers la droite.
Cette méthode « opère sur place » (pas de tableau auxiliaire).

6.2.4. Tri par fusion



Ce tri consiste à diviser l'ensemble des éléments à trier en 2 sous-ensembles d'importance à peu près égale que l'on trie par fusion (récursivité) avant de les fusionner en un seul ensemble. Cette méthode « n'opère pas sur place »

6.2.5. Tri par ventilation

Il est utilisable à chaque fois que toute clé peut s'écrire sous la forme d'une suite de composantes.
Ex : clé = c_1, c_2, c_3, ……., c_k
Chacune de ces composantes peut prendre un nombre fini de valeurs. On effectue des tris successifs sur chacune de ces composantes en allant de droite à gauche. (ex : clé numérique ou alphabétique)

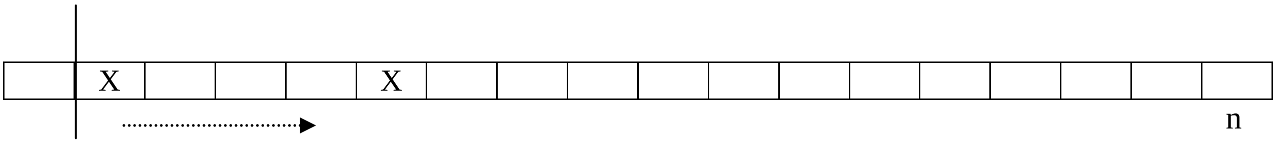
6.3. Les méthodes du tri par insertion

Tout algorithme de tri utilise des arguments qu'il reçoit en entrée et/ou restitue en sortie. On peut donc définir 3 types d'argument :

- les arguments en entrée : ce sont les données pour l'algorithme. Ils sont passés par valeur.
- les arguments en sortie : ce sont les résultats de l'algorithme. Ils sont passés par adresse.
- les arguments en entrée/sortie : ce sont à la fois des données et des résultats. Ils sont passés par adresse.

Dans la majorité des langages les 2 derniers types d'arguments sont regroupés.
Par convention, on fera figurer les arguments formels en entrée avant les autres, les deux sous-ensembles étant séparés par un point-virgule.

6.3.1. Méthode élémentaire



Le 1^{er} élément constitue la partie triée à lui tout seul (frontière = 2). Le dernier élément amènera la frontière à n .

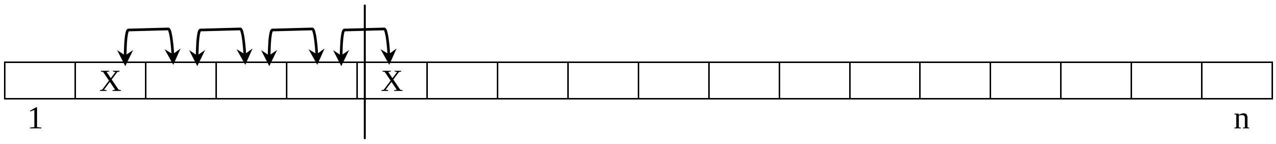
Triinsertion (n ; T)

argument en entrée

tableau contenant les élément non triés puis trié (argument d'entrée/sortie)

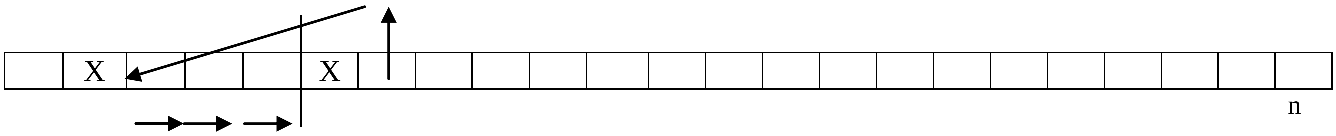
<u>triinsertion</u> (n;T)
début
    pour frontière allant de 2 à n par pas de +1, faire
        insérer (frontière,T)
    finpour
fin

6.3.1.1. insertion par permutations successives



<u>insérer (ifront ;T)</u>
début
    icourant ← ifront – 1
    tant que icourant > 0 et T(icourant) > T(icourant + 1) faire
        permuter T(icourant) et T(icourant + 1)
        décrémenter icourant de 1
    fin tant que
fin

6.3.1.2. insertion par décalage



<u>insérer (ifront ;T)</u>
début
    mémo ← T (ifront)
    icourant ← ifront – 1
    tant que icourant > 0 et T(icourant) > mémo, faire
        T(icourant + 1) ← T(icourant)
        icourant ← icourant – 1
    fin tant que
    T(icourant + 1) ← mémo
fin

6.3.1.3. version récursive

```
Triinsérer (n ;T)
début
        si n > 1 alors
                Triinsertion (n-1, T)
                Insérer (n,T)
        Finsi
Fin
```

Exemple pour n = 4 :

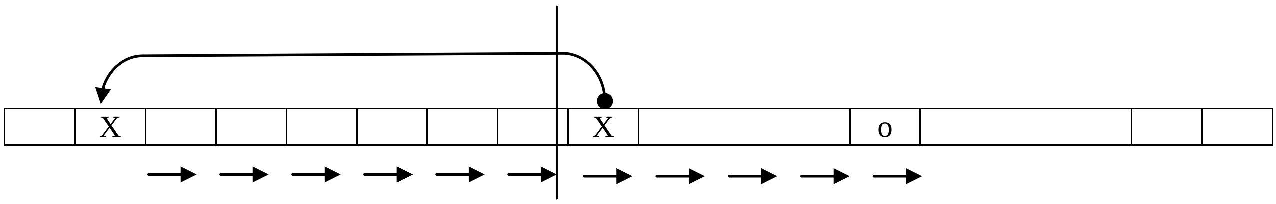
7	5	1	3

```
Triinsertion (4, T)
        N = 4
        Triinsertion (3,T)
                N = 3
        Triinsertion (2,T)
                N = 2
                        Triinsertion (1,T)
                        N = 1
                        Insérer (2,T)
                Insérer (3,T)
        Insérer (4,T)
```

Là le tri se fait à la remontée.

Complexité moyenne et maximale de $O(n^2)$. Son efficacité s'améliore considérablement si les éléménts à trier possèdent déjà une certaine notion d'ordre.

6.3.2. Amélioration : la méthode Shell



Dans la version élémentaire, on traite une structure de données à accès direct (par l'indice de tableau) avec un mécanisme séquentiel, il n'est donc pas surprenant d'obtenir un algorithme peu efficace.

La méthode SHELL utilise :

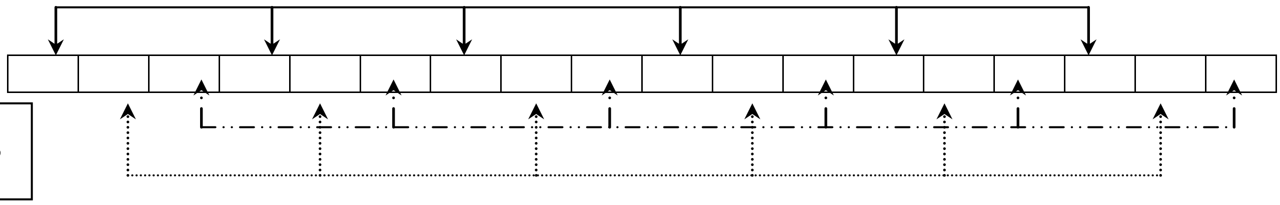
- la division
- l'équilibrage
- la remarque finale précédente (efficacité accrue si notion d'ordre).

* On choisit un entier h > 0.

On trie par insertion élémentaire les sous-tableaux suivants.

a_1, a_{1+h} , a_{1+2h} , a_{1+3h} , a_{1+4h} , ………………
a_2, a_{2+h} , a_{2+2h} , a_{2+3h} , a_{2+4h} , ………………
a_3, a_{3+h} , a_{3+2h} , a_{3+3h} , a_{3+4h} , ………………

a_h, a_{2h} , a_{3h} , a_{4h} , a_{5h} , ………………



h = 3

* On choisit un entier h' tel que h' > 0 et h' < h
on trie par insertion élémentaire les sous-tableaux suivants :

a_1, $a_{1+h'}$, $a_{1+2h'}$, $a_{1+3h'}$, $a_{1+4h'}$, ………………
a_2, $a_{2+h'}$, $a_{2+2h'}$, $a_{2+3h'}$, $a_{2+4h'}$, ………………
a_3, $a_{3+h'}$, $a_{3+2h'}$, $a_{3+3h'}$, $a_{3+4h'}$, ………………

$a_{h'}$, $a_{2h'}$, $a_{3h'}$, $a_{4h'}$, $a_{5h'}$, ………………

* On choisit un entier h''tel que h'' > 0 et h'' < h' …………

```
Shell (n ;T)
Début
        h← hinitial
        Répéter
                Pour premier  allant de 1 à h par pas de +1, faire
                        Triinsertion (premier, h, n ,T)
                Finpour
                h ← hsuivant
        jusqu'à h < 1
Fin
```

```
Triinsertion (ipremier ,pas ,dernier ;T)
Début
        Pour frontière allant de (ipremier + pas) à dernier par pas, faire
                Insérer (frontière,pas,T)
        Finpour
Fin
```

Insérer (ifront, p;T)
Début
    icourant ← ifront – p
    Tant que (icourant > 0) et T(icourant + p) > T(icourant) faire
        Permuter T(icourant) et T(icourant + p)
        icourant ← icourant – p
    Fin tant que
Fin

Pour le choix de h,  il n'existe pas une suite optimale de valeurs de h.

➔ la méthode est empirique, par essais : on a déterminé une suite en moyenne optimale de h.

$h_1 = 1$

$h_{i+1} = 3\ h_i + 1$

➔ $h_1=1$, $h_2=4$, $h_3=13$, $h_4=40$, $h_5=121$, $h_6=364$, $h_7=1093$, $h_8=3280$, …..

h initial > n/9

si n = 1000 éléments à trier → hinitial = h_5 = 121.
C'est à dire que l'on va trier 121 tableaux de 8 ou 9 éléments
Puis, 40 tableaux de 25 éléments
Puis, 13 tableaux de 76 ou 77 éléments
Puis, 4 tableaux de 250 éléments
Puis, 1 tableau de 1000 éléments

plus on descend, plus les tableaux sont pré-triés.

La méthode est de complexité moyenne et maximale O(n.logn), mais elle n'est pas stable.

6.4. Les méthodes du tri par échange

6.4.1. Méthode élémentaire : tri bulle

On fait le parcours des éléments à trier en les comparant 2 à 2 , consécutifs. On les échange si nécéssaire.

Ex :	12	8	7	15	9	10
	8	7	12	9	10	**15**

le dernier est à sa place finale, il ne reste plus que n – 1 parcours au plus à effectuer.

```
Parcours (n ;T)
Début
    Pour icourant allant de 1 à (n –1) par pas = +1, faire
        Si T(icourant) > T(icourant + 1) alors
            Echanger T(icourant) et T(icourant + 1)
        Finsi
    Finpour
Fin
```

On rajoute un indicateur d'échange pour limiter le nombre de parcours.

```
Parcours (n ; échange, T)
Début
    Pour icourant allant de 1 à (n –1) par pas = +1, faire
        Si T(icourant) > T(icourant + 1) alors
            Echange ← vrai
            Echanger T(icourant) et T(icourant + 1)
        Finsi
    Finpour
Fin
```

```
Tri bulle (n ;T)
Début
    Répéter
        Echange ← faux
        Parcours (n, échange, T)
    Jusqu'à non-échange
Fin
```

Complexité $O(n^2)$ → mauvais.

6.4.2. amélioration

Limiter l'étendue des parcours successifs, on sait que le dernier élément est toujours à sa place.

```
Tri bulle (n ;T)
Début
    dernier ← n
    Répéter
        Echange ← faux
        Parcours (dernier, échange, T)
        dernier ← dernier - 1
    Jusqu'à non-échange
Fin
```

Remarque : la procédure parcours n’a pas été touchée.

Cas particulier

2	3	4	5	6	1	
2	3	4	5	1	6	
2	3	4	1	5	6	
2	3	1	4	5	6	6 parcours
2	1	3	4	5	6	
1	2	3	4	5	6	

6.4.3. amélioration : alterner les sens de parcours

```
Tribulle2 (n ; T)
Début
    gauche ← 1
    droite ← n-1
    Répéter
        Echange ← faux
        Pour icourant allant de gauche à droite par pas = +1 faire
            Si T(icourant) > T(icourant + 1)
                Echange ← vrai
                Echanger T(icourant) et T(icourant + 1)
            Finsi
        finpour
        Décrémenter droite
        Si échange = vrai
            Echange ← faux
            Pour icourant allant de droite à gauche par pas de –1, faire
                Si T(icourant) > T(icourant +1) alors
                    Echange ← vrai
                    Echanger T(icourant) et T(icourant + 1)
                Finsi
            Finpour
            Gauche ← gauche + 1
        Finsi

    Jusqu’à plus d’échange
    Fin
```

6.4.4. amélioration : repérage de l'endroit du dernier échange

```
Tribulle3 (n ; T)
Début
        gauche ← 1
        droite ← n-1
        iéchange ← 1
        Répéter
                Pour icourant allant de gauche à droite par pas = +1 faire
                        Si T(icourant) > T(icourant + 1)
                                iéchange ← icourant
                                Echanger T(icourant) et T(icourant + 1)
                        Finsi
                finpour
                droite ← iéchange – 1
                Pour icourant allant de droite à gauche par pas = –1, faire
                        Si T(icourant) > T(icourant +1) alors
                                iéchange ← icourant
                                Echanger T(icourant) et T(icourant + 1)
                        Finsi
                Finpour
                gauche ← iéchange + 1
        Jusqu'à gauche > droite
Fin
```

6.4.5. amélioration décisive : le tri rapide ou Quicksort (Hoare 1962)

Principe :

$a, a_2, a_3, a_4, \ldots.., a_n$

- on choisit un élément particulier : le pivot
- on réorganise les éléments de la façon suivante :
    - Le pivot est à sa place finale 's'
    - Les éléments à gauche sont tous <= au pivot
    - Les éléments à droite sont tous > au pivot

a_i / a_i =< pivot	PIVOT	a_i / a_i > pivot

- on trie par la même méthode ($a_1 \ldots\ldots\ldots\ldots a_{s-1}$)
- on trie par la même méthode ($a_{s+1}, \ldots\ldots\ldots a_n$)

	Pivot	

s-1 s +1

	P				P'	

Piv	Piv

```
Trirapide (premier, dernier ; T)
Début
        Si dernier > premier alors
                Partager (premier, dernier, s, T)
                Trirapide (premier, s – 1, T)
                Trirapide (s + 1, dernier, T)
        Finsi
Fin
```

En cas de mauvais choix des pivots successifs, on aurait une grande profondeur d'appels récursifs d'où une place mémoire importante et dégénérescence de la méthode → complexité maxi $O(n^2)$.

Choix du pivot : dans la pratique, on choisit quelques éléments au hasard, on les trie et on en choisit l'élément médian.
On prend un nombre impair (soit 3, soit 5) d'éléments.

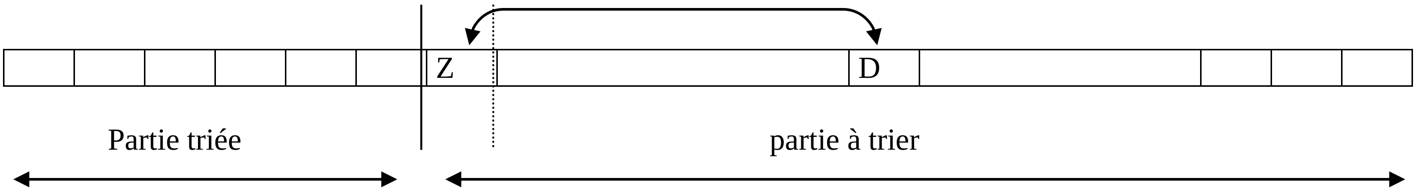
Cette méthode est utilisée seulement si le tableau à au moins un certain nombre d'éléments.

```
Trirapide (premier, dernier ; T)
Début
        Si dernier - premier > limite alors
                Partager (premier, dernier, s, T)
                Trirapide (premier, s – 1, T)
                Trirapide (premier, s + 1, T)
        Sinon
                Triinsertion (premier, dernier, T)
        Finsi
Fin
```

```
Partager(premier, dernier ; s, T)
Début
        Choix pivot(premier, dernier, T, s)          {s est la position initiale du PIVOT}
        Pivot ← T(s)
        Permuter T(premier) et T(s)
        gauche ← premier
        droite ← dernier
        Répéter
                Tant que gauche < droite et T(gauche) <= pivot, faire
                        gauche ← gauche + 1
                Fin tantque
                Tantque droite > gauche et T(droite) > pivot, faire
                        droite ← droite – 1
                Fin tantque
                Echanger T(droite) et T(gauche)
        Jusqu'à gauche = droite
        s = gauche – 1
        Permuter T(premier) et T(s)
Fin
```

Complexité moyenne : $O(n.\log n)$
Complexité maxi : $O(n^2)$

6.5. Les méthodes de tri par extraction.



1er élément frontière à traiter → indice 1
dernier élement frontière à traiter → indice n –1

6.5.1. Méthode élémentaire

Tri extraction(n ;T)
Début
    Pour frontière allant de 1 à (n-1) par pas = +1, faire
        imini ← frontière
        Pour icourant allant de frontière + 1 à n par pas = + 1
            Si T(icourant) < T(imini)
                imini ← icourant
            Finsi
        Finpour
        Permuter T(imini) et T(frontière)
    Finpour
Fin

Complexité : $O(n^2)$ → peu performant

6.5.2. Amélioration

Principe : on mémorise les résultats des différentes comparaisons.
C'est une méthode de type « TOURNOI »

Première étape :

n	57	66	11	31	92	18	4	75	n / 2 comparaisons
1er tour	57		11		18		4		n / 4 comparaisons
2ème tour	11				4				n / 8 comparaisons
finale	4								total : n – 1 comparaisons

On a construit un arbre ordonné verticalement. C'est un ordre partiel. L'arbre a n feuilles.
→ coût : $O(n)$

Deuxième étape (1ère phase) : on remplace la feuille égale à la racine par son adversaire du 1er tour

n 57 66 11 31 92 18 **75** 75

1er tour 57 11 18 4

2ème tour 11 4

finale 4

Deuxième étape (2ème phase) : on remet un ordre vertical sur l'arbre en parcourant la branche allant de la feuille modifiée à la racine

n 57 66 **11** 31 92 18 75 75

1er tour 57 **11** 18 **75**

2ème tour **11** **18**

finale **11**

$\log_2 n$ comparaisons

Le coût passe alors à **$\log_2$ n** pour cette phase qu'on répète (n-1) fois, d'où un coût **n.logn**, le coût global est donc **O(n.logn).**
Pour trier n éléments, on a 2n-1 cases mémoire → très volumineux.

6.5.3. Amélioration décisive : le tri arbre (ou HEAPSORT)

On considère l'ensemble des éléments à trier comme la représentation d'un arbre binaire stocké de manière compacte (1), on l'ordonne verticalement(2), puis on en extrait les extrema successifs.

(1) : Pour tout élément a_i, son fils gauche s'il existe est a_{2i}, son fils droit s'il existe est a_{2i+1}

A1	A2	A3	A4	A5	A6	A7	A8

A1

A2 A3

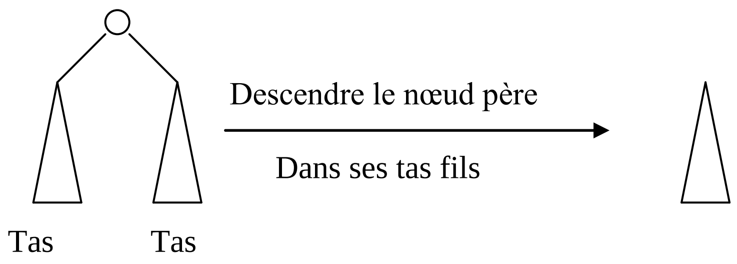
A4 A5 A6 A7

A8

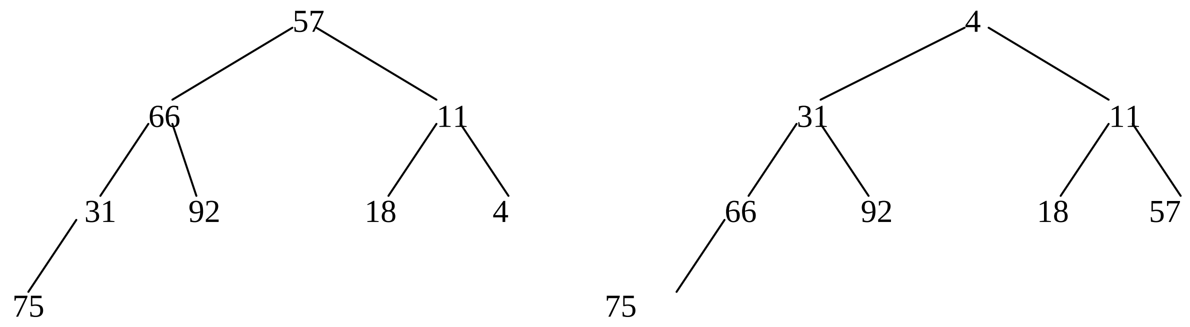
(2) : un arbre sera ordonné verticalement si en tout nœud sa clé est inférieure (respectivement supérieure) à celles de ses fils. C'est la structure de tas.

Heapsort (n;T)
Début
Construire le tas(n,T)
Trier le tas (n,T)
Fin

Construire le tas :
Les feuilles, c’est à dire a_i tel que i > n/2, sont des arbres possédant une structure de tas.
On fait remonter la structure de tas du niveau terminal au niveau global.



Pour cela, on considère les nœuds non feuilles, c’est à dire a_i tel que i <= n/ 2 dans l’ordre décroissant des indices pour effectuer sur chacun l’opération de descente.



Construire le tas (n ;T)
Début
Pour nœud allant de n/2 à 1 par pas = -1, faire
Descendre (nœud, n, T)
Fin pour
Fin

Descendre (version récursive)

```
Descendre(père, n ; T)
Début
            Si père =< n/ 2 alors                    (père n'est pas une feuille)
               Fils ← 2 * père                       (fils gauche)
               Si fils < n alors                     (existe-t-il un fils droit ?)
                       Si T(fils) > T(fils + 1) alors
                               Fils ← fils + 1
                       Finsi
               Finsi
               Si T(fils) < T(père), alors
                       Permuter T(père) et T(fils)
                       Descendre(fils, n, T)
               Finsi
            Finsi
Fin
```

Descendre (version itérative)

```
Descendre(père, n ; T)
Début
         mémo ← T(père)
         père courant ← père
         fils ← 2 * père
         descente ← vrai
         Tant que descente vrai et fils <= n , faire
               Si fils < n
                       Si T(fils) > T(fils+1), alors
                               fils ← fils + 1
                       Finsi
               Finsi
               Si T(fils) < mémo, alors
                       T(père courant) ← T(fils)
                       père courant ← fils
                       fils ← 2 * fils
               Sinon
                       descente ← faux
               Finsi
         Fin tantque
         T(père courant) ← mémo
Fin
```

Trier le tas :

- on permute T(1) et T(n)
- on remet une structure de tas sur l'arbre, par descente de la racine dans {T(1) …. T(n – 1)}
- on permute T(1) et T(n – 1)
- on remet une structure de tas sur l'arbre par descente de la racine dans {T(1) ….. T(n – 2)}
- on permute T(1) et T(n – 2)
- on remet une structure de tas sur l'arbre par descente de la racine dans {T(1) ….. T(n – 3)}
- on permute T(1) et T(3)
- on remet une structure de tas sur l'arbre par descente de la racine dans {T(1),T(2)}
- on permute T(1) et T(2)

```
trier le tas (n ; T)
début
        pour dernier allant de n à 3 par pas = -1, faire
                permuter T(1) et T(dernier)
                descendre (1,dernier –1, T)
        fin pour
        permuter T(1) et T(2)
fin
```

→ complexité moyenne et maximale : O(n.logn)

7. Comparaison des temps de traitement en fonction des volumes

N	10	100	1000	10000	50000	En ordre	Aléatoire	inverse
Bulle	0.16	20	2400			54	100	150
Extraction	0.12	7.3	680			49	51	70
Insertion	0.12	6.7	610			2	37	70
Shell	0.07	2	37	600	4200	6	13	16
Arbre	0.2	3.5	50	660	3960	11	11	10
Rapide		2	28	365	2140	3	6	4

- **Tri bulle** : à partir de 1000 enregistrements, peu performant, voir catastrophique lorsque le fichier est en ordre inverse.
- **tris extraction et insertion** : performances moyennes et équivalentes. A noter le bonne performance du tri insertion lorsque le fichier est en ordre.
- **Tris arbre, shell, et rapide**: très bonne performance, à noter la constance du tri arbre quelque soit l'état (l'ordre) du fichier à trier.

8. La gestion des tables

Définition : c'est un ensemble de couples(clé, information) où une information n'est accessible que par l'intermédiaire de sa clé. Par exemple, le fichier des salariés d'une entreprise constitue une table où la **clé** est le nom du salarié et l'**information** son adresse, numéro de téléphone, …

Les opérations sur les tables :
- recherche de l'information associée à une clé (recherche de l'adresse d'un salarié)
- ajout d'une clé et de son information associée (embauche d'un salarié)
- modification de l'information associée à une clé (salarié qui déménage)
- suppression d'une clé et de son information associée (salarié qui démissionne)

→ On essaye de réaliser ces opérations le plus rapidement possible en choisissant une organisation des données appropriée. Malheureusement, les différentes opérations conduisent à des contraintes d'organisation contradictoires. On est donc obligé de faire des compromis.

8.1. Les modes d'adressage.

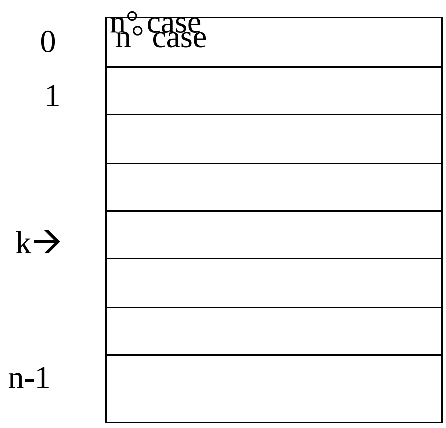
8.1.1. L'adressage fonctionnel simple

On considère que toutes les clés possibles peuvent figurer simultanément dans la table, et donc on réserve la place nécessaire pour stocker autant d'informations qu'il peut exister de clés.

F:clé ⟶
F(clé) = k
F est bijective

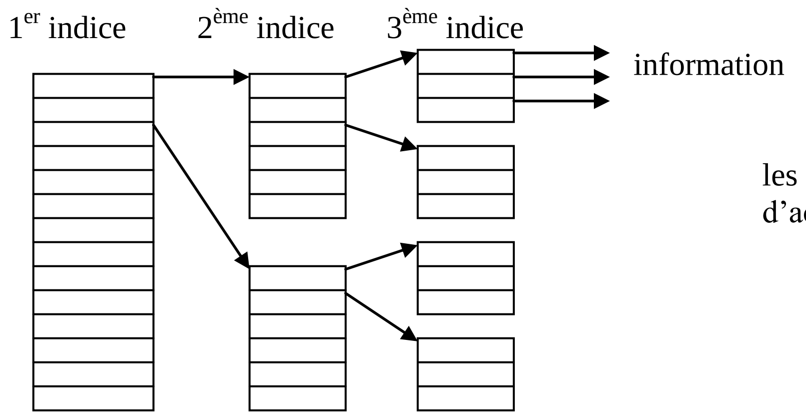
Avantages :
- on ne stocke pas les clés
- on a un accès direct à l'élément
- on n'a jamais à réorganiser la table

inconvénients :
- soit n le nombre de clés possibles et p le nombre de clés gérées, on a en général **p << n** d'où gâchis de place.



8.1.2. L'adressage fonctionnel hiérarchisé

C'est une amélioration de la technique précédente. On divise la table en couches hiérarchiques obtenues par la décomposition de la clé en plusieurs parties. La détermination de l'information associée à une clé se fera en plusieurs étapes (autant qu'il y a de parties résultant de la décomposition de la clé).
Application : stockage de tableaux très grands à 3 indices.



les 3 indices constituent la clé qui permet d'accéder à l'information .

8.1.3. L’adressage associatif simple

Stockage des clés et de l’information associée.
Recherche par comparaisons successives.

Si les clés sont non-triées :

- recherche = P / 2 (P est le nombre de clés).
- ajout = 1
- suppression = (P / 2) + 2
- modification = (P / 2) + 1

Si les clés sont triées :

- recherche = $\log_2 P$
- ajout = logp + (P / 2)
- suppression = logp + (P / 2)
- modification = logp + 1

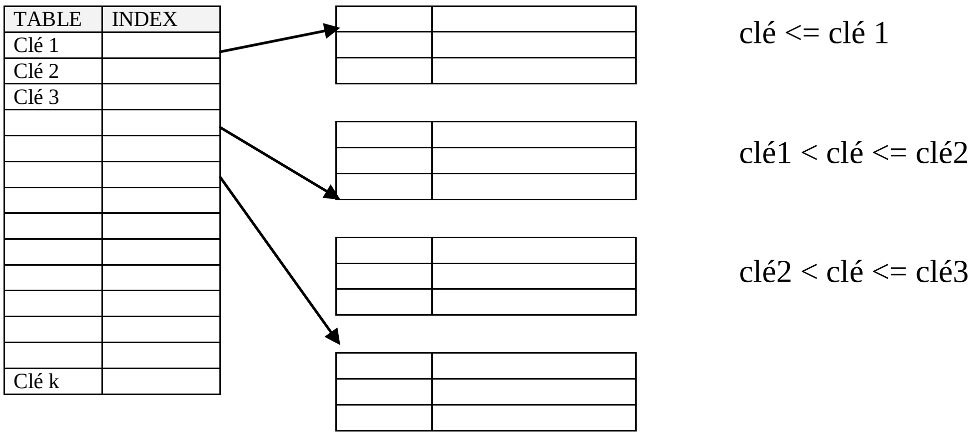
$\log_2 P$

→ pas utilisé car on a dans les deux cas des opérations en 0 (p).

8.1.4. L’adressage séquentiel indexé

On améliore en divisant la table en plusieurs sous-tables accessibles par une table d’index.

Table d’index



Les clés de la table d’index sont des clés effectivement gérées et telles que les effectifs des sous tables soient les plus voisins possible (c’est à dire que la taille des sous tables soit la plus constante possible).

- recherche = (k / 2) + (P / 2k) (P est le nombre de clés).
- ajout = (k / 2) + 1
- suppression = (k / 2) + (P / 2k) + 2
- modification = (k / 2) + (P / 2k) + 1

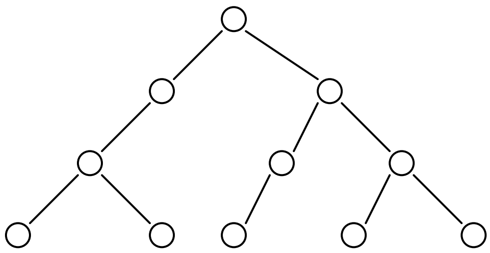
ce qui représente un gain de 10 fois supérieur à l’adressage associatif simple, si le nombre de sous-tables k est égal à 10.

8.1.5. L’adressage dispersé ou Hash-coding

C’est l’adressage fonctionnel sur une table de taille limitée au besoin. Le nom vient du fait que 2 clés à priori très proches, peuvent être très éloignées dans la table.

8.1.6. L’adressage arborescent

On utilise un arbre binaire



La recherche d’un élément est équivalente à la recherche dichotomique, dans le pire des cas on parcourt toute la hauteur de l’arbre ; l'ajout et la suppression d'élément voient leur performance améliorée.

8.2. L’adressage dispersé (HASH CODING)

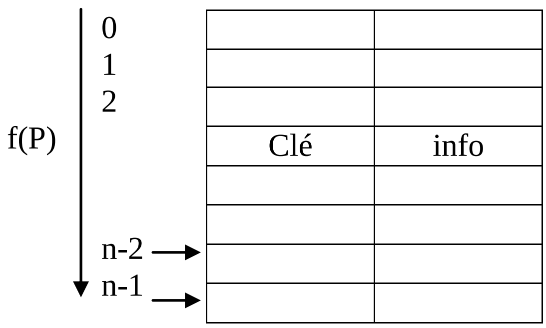
8.2.1. principe

c’est un adressage fonctionnel sur une table de taille limitée aux besoins

- n cases dans la table (de 0 à n-1).
- P clés présentes : P est de l’ordre de n.

Avantages : ceux de l’adressage fonctionnel (temps d’accès très faible, pas de réorganisation).

Inconvénients : on est incapable de définir à priori une fonction f qui soit bijective → 2 clés peuvent conduire à la même case : il y a collision.



8.2.2. La fonction de dispersion

Elle doit répartir le plus uniformément possible les clés dans la table, ceci afin de minimiser le nombre de collisions

Clé = $c_1\ c_2 \ldots\ldots\ldots\ldots\ldots c_k$

$$F(\text{clé}) = \sum_{i=1}^{i=k} (k-i+1) * \text{code}(c_i)$$

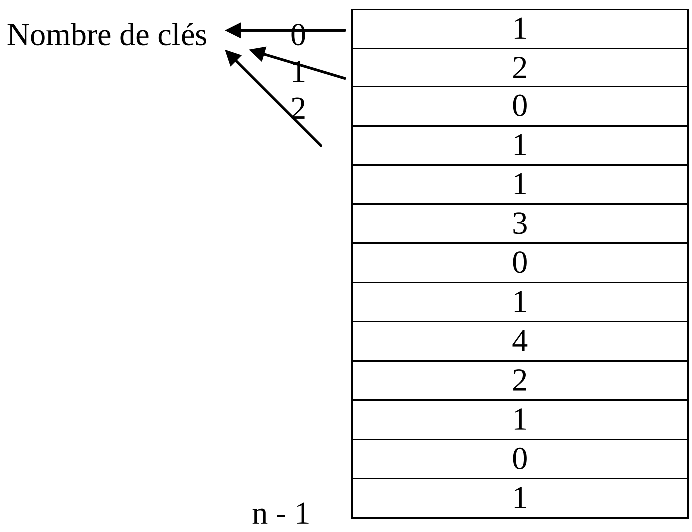
→ pour résoudre les problèmes de multiplicité des clés, on peut utiliser des fonctions à base de nombres premiers.

Il se peut que des cases ne soient jamais atteintes et que d'autres le soient plusieurs fois.
→ on parle de collisions primaire : f(clé1) = f(clé2), avant le modulo
→ on parle de collision secondaire f(clé1) = f(clé2), seulement après le modulo

<u>détermination du nombre moyen d'accès à la table pour la recherche d'une clé</u>

a :Probabilité de tomber dans une case à une clé quand on cherche une clé qui figure dans la table

$$a = \frac{\text{nombre de cases à une clé}}{\text{nombre total de clés}}$$



k : nombre moyen de clés dans les cases où il y a collision

$$k = \frac{\text{nombre total de clés en collision}}{\text{nombre de cases où il y a collision}}$$

b : probabilité de tomber dans une case vide quand on cherche une clé qui ne figure pas dans la table

$$b = \frac{\text{nombre de cases vides}}{\text{nombre de cases total}}$$

c : probabilité de tomber dans une case à une clé quand celle-ci ne figure pas dans la table

$$c = \frac{\text{nombre de cases à une clé}}{\text{nombre de cases total}}$$

- on cherche une clé qui figure dans la table.

  $Nma1 = 1 * a + \frac{k + 1}{2} * (1 - a)$

nb moyen

- on cherche une clé qui ne figure pas dans la table.

  $Nma2 = 1 * b + 1 * c + k * (1 - b - c)$

D'une façon générale, on considère que 9 fois sur 10 on cherche une clé qui se trouve dans la table, d'où,

$$\boxed{Nma = 0{,}9 * nma1 + 0{,}1 * nma2}$$

Nma < 1,5 correspond à une bonne fonction de dispersion.

8.2.3. Traitement des collisions.

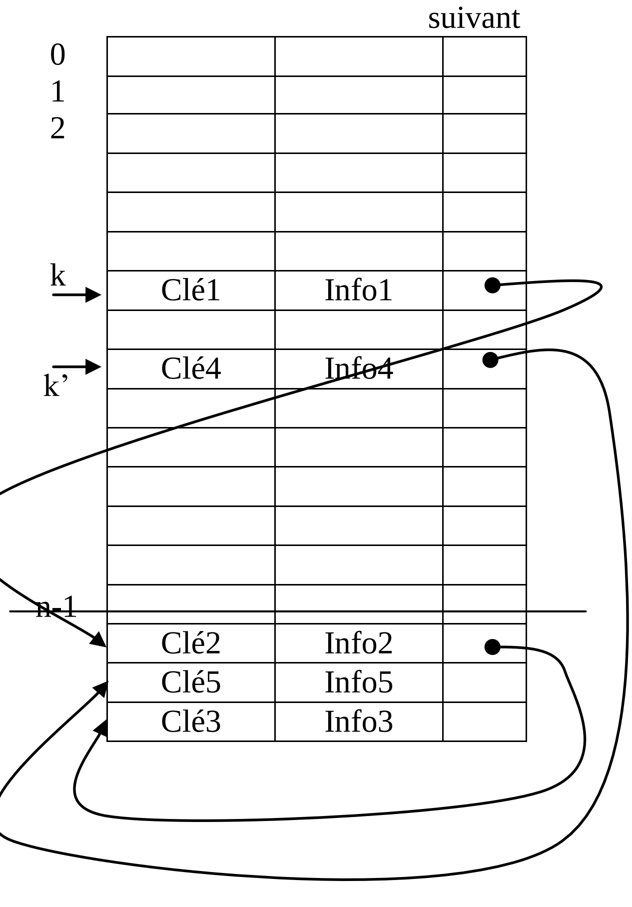
8.2.3.1. <u>Débordement en liste linéaire</u>

f(clé1) = f(clé2) = f(clé3) = k
f(clé4) = f(clé5) = k'

stocker les clés en collision sous forme d'une liste linéaire chaînée à partir de la case où il y a collision

en pratique, ces listes chaînées sont mises à la fin de la table

la table est constituée de 2 parties :
- une partie fixe à n cases qui peut contenir des trous
- une partie dynamique qui ne doit pas contenir de trous contenant les listes de collisions



Type élément table = enregistrement
    Clé : typeclé
    Info : typeinfo
    Suivant : entier
Fin

Recherche d'un élément

Recherche(X ; emplacement, trouvé)
Début
    Trouvé ← faux
    Emplacement ← rien
    Position ← HC(X.clé) (HC = hash coding).
    Répéter
        Lire un élément en position
        Si élément.clé = X.clé alors
            Trouvé ← vrai
            Emplacement ← position
        Sinon
            Position = élément.suivant
        Finsi
    Jusqu'à trouvé ou position = rien
Fin

Ajout d'un élément

Ajout(X)
Début
    Position ← HC(X.clé)
    Lire élément en position
    Si élément = élément vide alors
        X.suivant ← rien
        Ecrire X en position
    Sinon
        X.suivant ← élément.suivant
        Position X ← fin de table
        Ecrire X en position X
        Elément.suivant← position X
        Ecrire élément en position
    Finsi
Fin

Suppression d'un élément

3 cas :
- en partie fixe, sans liste de collision → RAZ, pas de suppression physique
- en partie variable donc dans la liste de collision : refaire le chaînage de collision et boucher le trou avec le dernier élément de la table, donc mettre à jour le lien avec son précédent

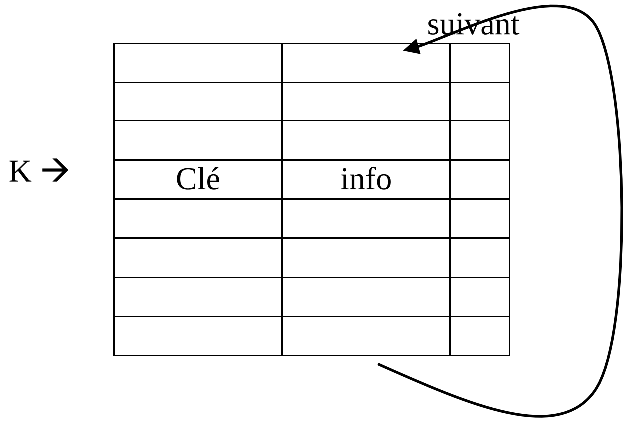
- la clé est dans la table et il y a une liste de collision attachée : on remonte le premier élément de la liste de collision en partie fixe et on bouche le trou généré avec le dernier élément de la table en mettant à jour le lien avec son précédent

Performant mais limité
P est de l'ordre de n → OK

- augmentation de (10%) → les listes de collision s'allongent, les performances se dégradent
- changement de structure des clés → revoir la fonction de dispersion.

8.2.3.2. Débordement à l'intérieur de la table

Si p =< n et s'il y a collision entre 2 clés, alors il y a au moins une case libre dans la table, on va donc la chercher en parcourant la table toujours selon la même séquence
On considère la table comme circulaire



Recherche linéaire
On essaye successivement toutes les cases qui suivent celle où a lieu la collision.

G(clé,i) = k + i – 1 modulo n
avec : k = f(clé)
i =numéro de la tentative de rangement

Ceci provoque une accumulation des clés dans les cases suivant celle où a lieu la collision engendrant en plus des collisions dans ces cases là.

Recherche quadratique

H(clé,i) = k + $(i- 1)^2$ mod n.

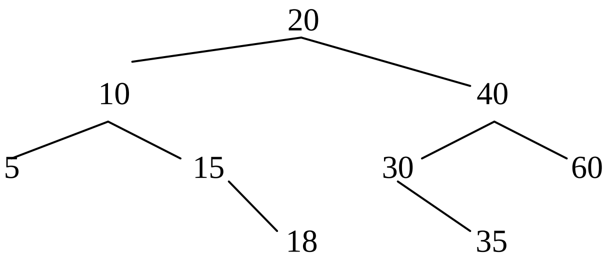
On n'est pas sûr de considérer toutes les cases de la table. On peut conclure que la table est saturée alors qu'il reste des places libres.
Si la taille n de la table est un nombre premier, alors on est sûr de considérer au moins n / 2 cases différentes avant de repasser dans une case déjà parcourue.
→ en pratique, on prend n premier et on considère que la table est saturée, si les n / 2 premières tentatives échouent.

8.3. Les tables arborescentes.

8.3.1. Les arbres binaires ordonnés horizontalement

Un arbre binaire est dit ordonné horizontalement si en tout nœud, sa clé est supérieure à toutes celles de son sous-arbre gauche (SAG) et inférieure à toutes celles de son sous-arbre droit (SAD).

Exemple



Structure de données utilisée :

```
type arbre binaire = enregistrement
        clé = typclé
        info = typinfo
        SAG = arbre binaire
        SAD = arbre binaire
Fin
```

<u>recherche d'un élément</u>

```
Recherche (X ; A , trouvé)          (algorithme récursif)
Début
        Si vide(A) alors
                Trouvé ← faux
        Sinon
                Si X < A.clé alors
                        Recherche (X,A.SAG, trouvé)
                Sinon
                        Si X > A.clé alors
                                Recherche(X,A.SAD, trouvé)
                        Sinon
                        Trouvé ← vrai
                                Afficher A.info
                        Finsi
                Finsi
        Finsi
Fin
```

Ajout d'un élément

On l'accroche à un élément qui a au plus un fils (un sous-arbre)

Ajout(X ; A, existedéjà) (X = élément , A = Arbre)

```
Début
        Si vide(A) alors
                Existedéjà ← faux
                A.clé ← X.clé
                A.info ← X.info
                A.SAG ← rien
                A.SAD ← rien
        Sinon
                Si X.clé < A.clé alors
                        Ajout (X, A.SAG, existe déjà)
                Sinon
                        Si X.clé > A.clé alors
                                Ajout (X, A.SAD, existe déjà)
                        Sinon
                                Existedéjà ← vrai
                        Finsi
                Finsi
        Finsi
Fin
```

Suppression d'un élément

Elément à supprimer :

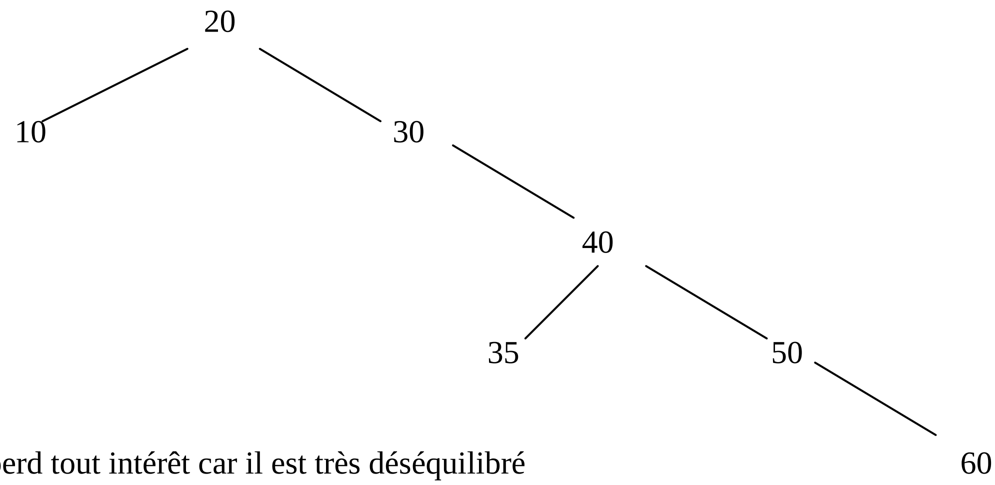
- feuille	:	on coupe la feuille
- il a un fils	:	on remonte le sous-arbre(fils)
- il a deux fils	:	on remonte : - soit l'élément le plus à droite du SAG - soit l'élément le plus à gauche du SAD

```
Suppression (X ; A, trouvé)
Début
    Si vide(A) alors                    (arrêt récursivité)
        Trouvé ← faux
    Sinon
        Si X< A.clé alors
            Suppression (X, A.SAG, trouvé)
        Sinon
            Si X > A.clé alors
                Suppression (X, A.SAD, trouvé)
            Sinon
                Trouvé ← vrai
                Si vide (A.SAD) alors
                    A ← A.SAG                 → cas 1 ou 2
                Sinon
                    Si vide (A.SAG) alors
                        A ← A.SAD             → cas 2
                    Sinon
                        Remonter(A.SAG,A)     → cas 3
                    Finsi
                Finsi
            Finsi
        Finsi
    Finsi
Fin
```

```
Remonter (;B,C)
Début
    Si vide(B.SAD) alors
        C.clé ← B.clé
        C.info ← B.info
        B ← B.SAG
    Sinon
        Remonter (B.SAD, C)
    Finsi
Fin
```

8.3.2. Les tables

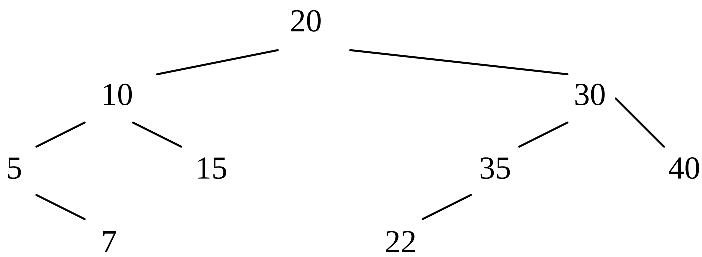
Exemple



Un tel arbre perd tout intérêt car il est très déséquilibré

8.3.2.1. <u>Les arbres parfaitement équilibrés</u>

Critère :
En tout nœud, le nombre d'éléments du SAG et le nombre d'éléments du SAD différent au plus de 1



A chaque ajout ou suppression, il est nécessaire de rééquilibrer l'arbre → trop coûteux → Abandon, d'où le paragraphe suivant

8.3.2.2. <u>Les arbres partiellement équilibrés</u>

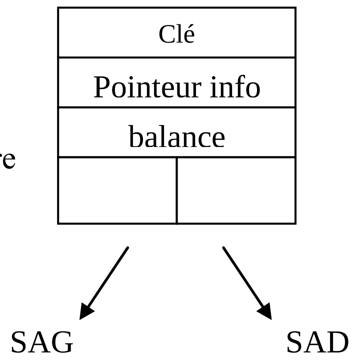
Le critère d'équilibre porte sur les hauteurs du SAD et du SAG

<u>Exemple</u>
Arbre AVL (Andelson Velskii, Landis)

On a un équilibre au sens AVL si en tout nœud la hauteur du SAG et la hauteur du SAD diffèrent au plus de 1

Représentation

la balance reflète l'état d'équilibre du nœud



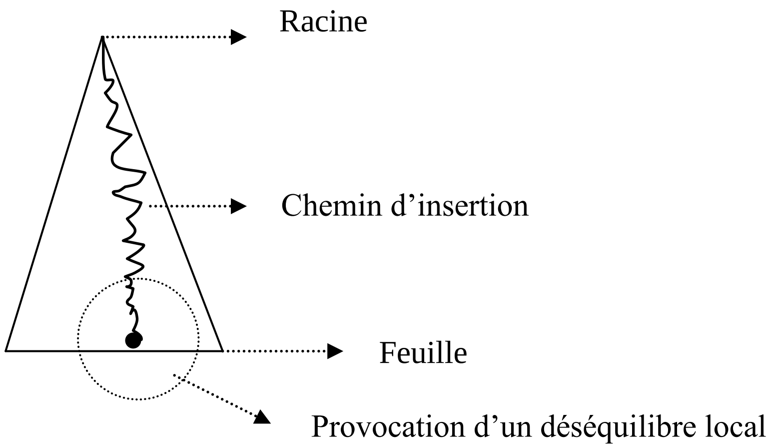
Elle peut prendre 3 valeurs :
- si hauteur du SAG < hauteur du SAD → balance = + 1
- si hauteur du SAG = hauteur du SAD → balance = 0
- si hauteur du SAG > hauteur du SAD → balance = - 1

Type Nœud AVL = enregistrement

Clé	: typclé
Pointeur info	: typepointeurdinfo
balance	: -1..+1
SAG	: Nœud AVL
SAD	: Nœud AVL

Fin

<u>Ajout d'un élément</u>

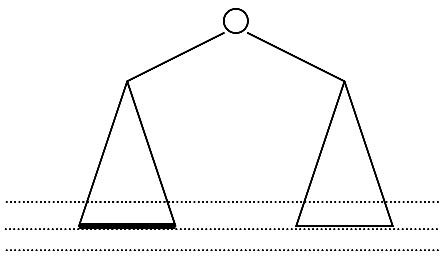


→ on est amené à envisager de rééquilibrer l'arbre à tous les niveaux intermédiaires.

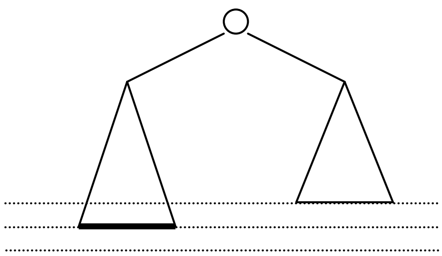
Cas d'ajout d'un élément dans le SAG

<u>1er cas</u> : le SAG n'a pas grandi → l'équilibre est conservé

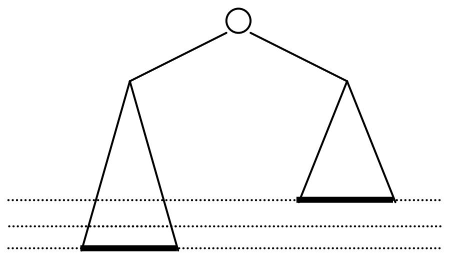
<u>2ème cas</u> : le SAG a grandi alors qu'il était minimal → l'équilibre s'est amélioré
l'arbre n'a pas grandi



<u>3ème cas</u> : le SAG a grandi alors que le SAG et le SAD avaient même hauteur → l'équilibre s'est dégradé
l'arbre a grandi.

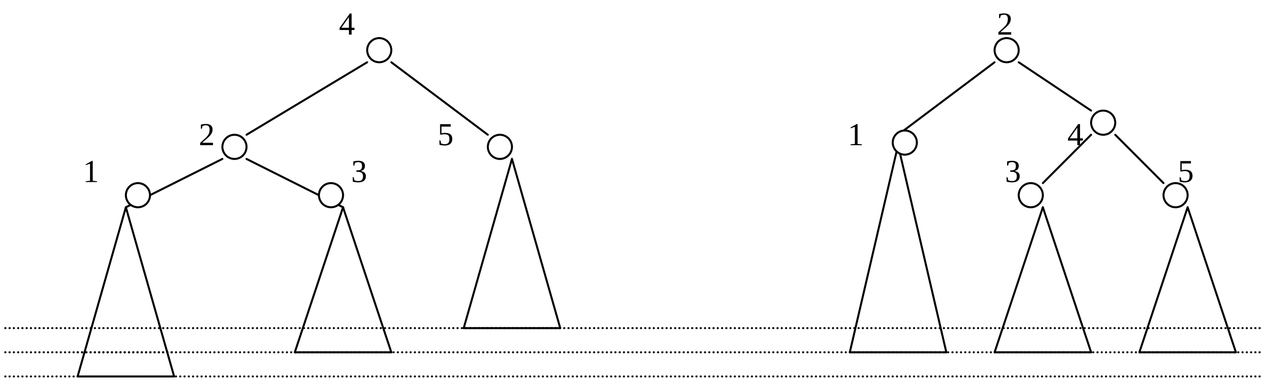


4ème cas : le SAG a grandi alors qu'il était maximal → il faut rééquilibrer



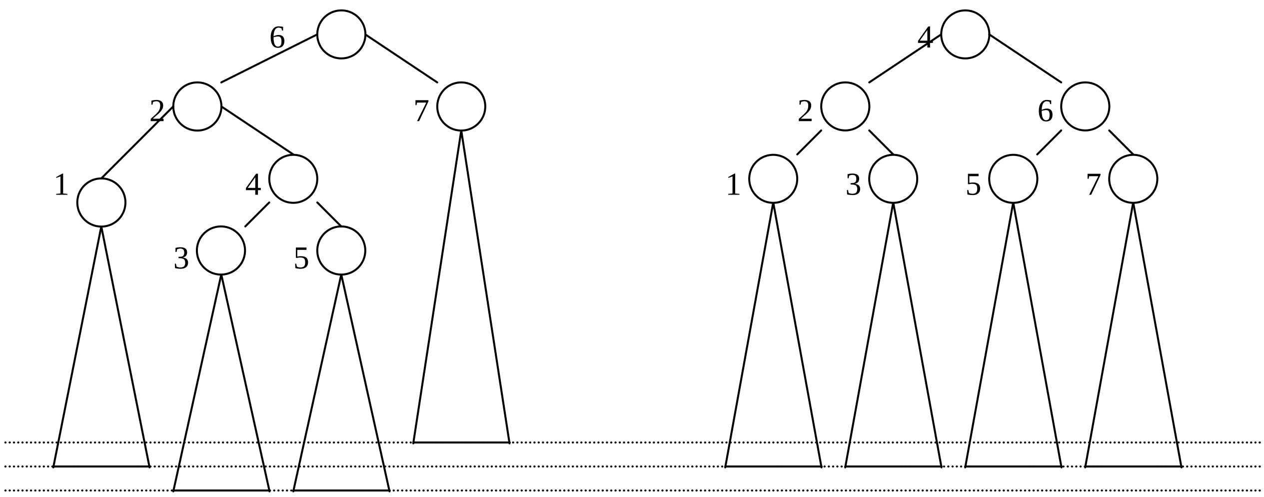
Rééquilibrage

- Cas gauche-gauche : déséquilibre dû au SAG du SAG



→ l'arbre n'a pas grandi, l'équilibre s'est amélioré

- Cas gauche-droite : déséquilibre dû au SAD du SAG



→ l'arbre n'a pas grandi, l'équilibre s'est amélioré

```
Ajout AVL (X ; A, hauteur)
Début
    Si vide(A) alors
        A.clé ← X.clé
        A.info ← X.info
        A.SAG ← rien
        A.SAD ← rien
        A.balance ← 0
        hauteur ← vrai                    (l'arbre a grandi)
    Sinon
        Si X.clé < A.clé alors
        Ajout AVL(X, A.SAG, hauteur)
        Si hauteur alors
            Si A.balance = -1
                Rééquilibrer soit gauche-gauche soit gauche-droite
            Sinon
                Si A.balance = 0 alors
                    A.balance ← -1
                Sinon
                    Hauteur ← faux
                    A.balance ← 0
                Finsi
            Finsi
        Finsi
    Sinon
        Ajout AVL(X, A.SAD, hauteur)
        Si hauteur alors
            Si A.balance = +1
                Rééquilibrer soit droite-droite soit droite-gauche
            Sinon
                Si A.balance = 0 alors
                    A.balance ← +1
                Sinon
                    Hauteur ← faux
                    A.balance ← 0
                Finsi
            Finsi
        Finsi
    Finsi
    Finsi
Fin
```

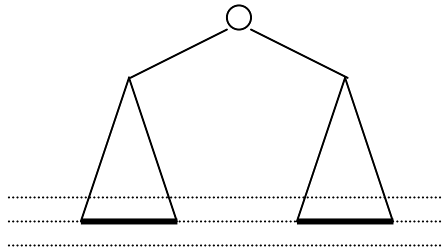
<u>Suppression d'un élément</u>

Comme pour l'ajout, on est amené à envisager de rééquilibrer l'arbre à tous les niveaux intermédiaires.

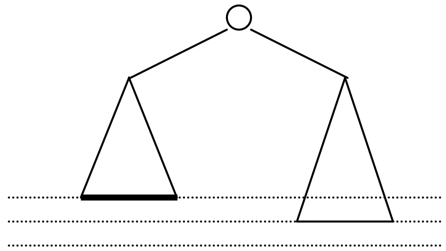
Cas de suppression d'un élément dans le SAG

1^{er} cas : le SAG n'a pas diminué de hauteur → l'équilibre est conservé

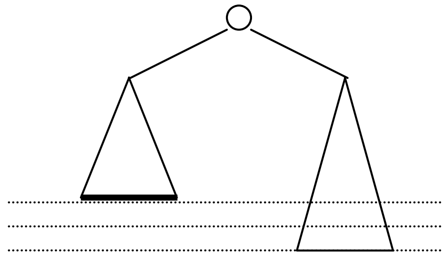
$2^{ème}$ cas : le SAG a diminué alors qu'il était maximal → l'équilibre s'est amélioré
l'arbre a diminué de hauteur



$3^{ème}$ cas : le SAG a diminué alors que le SAG et le SAD avaient même hauteur
→ l'équilibre s'est dégradé
l'arbre n'a pas diminué de hauteur

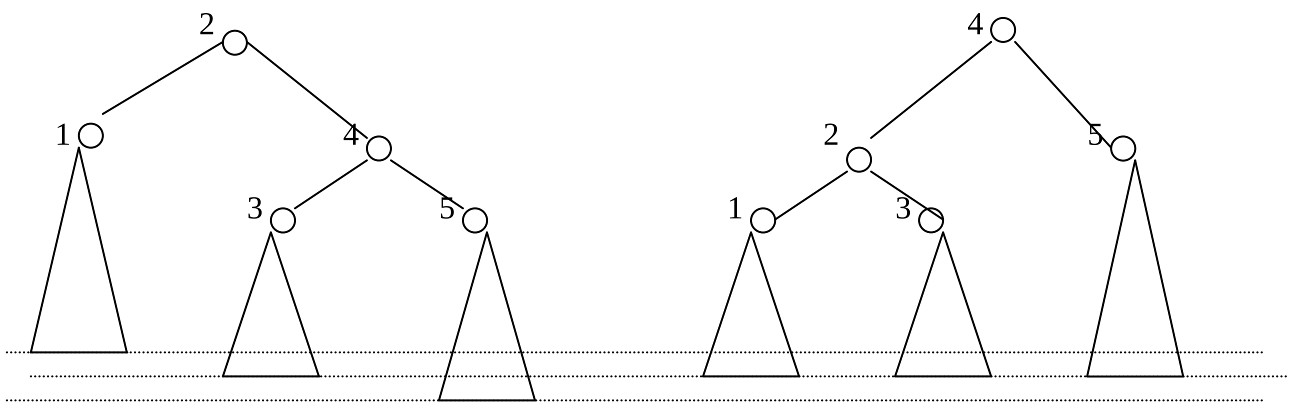


$4^{ème}$ cas : le SAG a diminué alors qu'il était minimal → il faut rééquilibrer



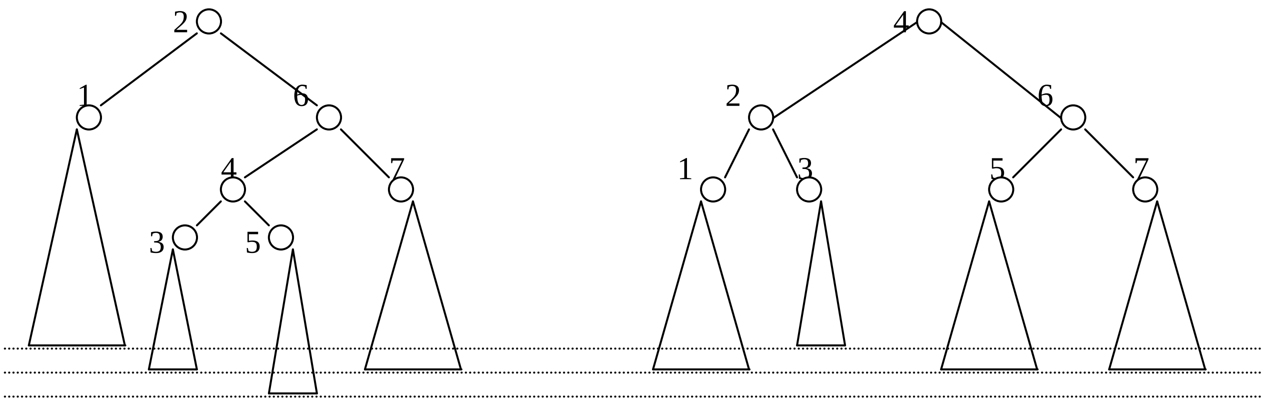
Rééquilibrage

- Cas droite-droite : déséquilibre dû au SAD du SAD



→ l'arbre a diminué de hauteur, l'équilibre s'est amélioré

- Cas droite-gauche : déséquilibre dû au SAG du SAD



→ l'arbre a diminué de hauteur, l'équilibre s'est amélioré

```
Supprimer AVL(X ;A,hauteur)
Début
    Si vide(A) alors
        Ecrire pas trouvé
    Sinon
        Si X < A.clé alors
            Supprimer AVL(X,A.SAG, hauteur)
            Si hauteur alors
                Si A.balance = 1 alors
                    Rééquilibrer soit droite-droite soit droite-gauche
                Sinon
                    Si A.balance = 0 alors
                        Hauteur ← faux
                        A.balance ← +1
                    Sinon
                        A.balance ← 0
                    Finsi
                Finsi
            Finsi
        Sinon
            Si X > A.clé alors
                Supprimer AVL(X,A.SAD, hauteur)
                Si hauteur alors
                    Si A.balance = -1  alors
                        Rééquilibrer soit gauche-gauche soit gauche-droite
                    Sinon
                        Si A.balance = 0 alors
                            Hauteur ← faux
                            A.balance ← -1
                        Sinon
                            A.balance ← 0
                        Finsi
                    Finsi
                Finsi
            Sinon
                Si vide(A.SAG) alors
                    A ← A.SAD
                    Hauteur ← vrai
                Sinon
                    Si vide (A.SAD) alors
                        A ← A.SAG
                        Hauteur ← vrai
                    Sinon
                        Remonter (A.SAG, A, hauteur)
                        Si hauteur alors
                            Si A.balance = 1 alors
                                Rééquilibrer soit droite-droite soit droite-gauche
                        Sinon
                            Si A.balance = 0 alors
                                Hauteur ← faux
                                A.balance ← 1
                            Sinon
                                A.balance ← 0
                            Finsi
                        Finsi
                    Finsi
                Finsi
            Finsi
        Finsi
    Finsi
Fin
```

```
Remonter ( ;B, C, haut)

Début
    Si non vide (B.SAD) alors
        Remonter(B.SAD, C, haut)
        Si haut alors
            Si B.balance = -1
                Rééquilibrer soit gauche-gauche soit gauche-droite
            Sinon
                Si B.balance = 0 alors
                    Haut ← faux
                    B.balance ← -1
                sinon
                    B.balance ← 0
                Finsi
            Finsi
        Finsi
    Sinon
        C.clé ← B.clé
        C.info ← B.info
        B ← B.SAG
        Haut ← vrai
    Finsi
Fin
```