Algorithmique (Support de cours)

N° 1

C.Ernst

H.Ettaleb

A.Fonkoua

Septembre 2005

Algorithmique : Plan

Séance I

- Performance des algorithmes
- Structures linéaires
  - piles, files
- Structure de données
  - enregistrement
  - tableau
- Implémentation : tableau

Séance II

- Contexte d'exécution d'un programme
- Pile & Tas
- Mémoire dynamique
  - gestion dynamique de la mémoire
  - allocation & libération
- Structure de données : listes chaînées
- Application : pile et file d'attente

Séance III

- Induction mathématique
- Récursivité
- Application : tri rapide

Séance IV

Exercice récapitulatif : tri par fusion

- Principe, Algorithmes
- Structures de données

Séance 1 : Tableaux

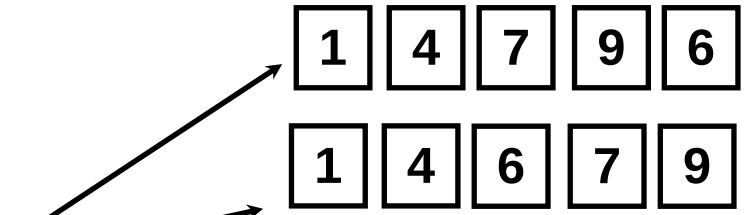
- **Performance des algorithmes**
- **Structures Linéaires**
  - **piles**
  - **files**
- **Structure de données :**
  - **enregistrement**
  - **tableau**
- **Implémentation des piles et files d'attente à l'aide de tableaux**

Performance des algorithmes

- **Performances d'un programme :**
  - **Temps d'exécution du programme**
  - **Encombrement mémoire nécessaire**
- **Intérêt :**
  - **critères de choix d'un algorithme en fonction de :**
    - » **son temps de réponse**
    - » **de l'espace mémoire nécessaire à son exécution**
- **Difficiles à évaluer en général, car dépendantes :**
  - **de la machine et du compilateur**
  - **des données**
- **Complexité d'un algorithme :**
  - **on essaie d'ignorer les spécificités de la machine et du compilateur ...**

Temps d'exécution des programmes : Exemple

- **Dépend des données :**
  - **taille des données : ex. nombre d'éléments à trier**
  - **nature des données : ex. nombre d'"inversions" (désordre)**



- **Dépend de la machine :**
  - **cycle machine**
  - **structure des instructions RISC/CISC**

- **Dépend du compilateur :**
  - **optimisations**
  - **facteur d'expansion en assembleur**

```
fonction Max ( valeur réel T[],
               valeur entier N)
               retourne réel
réel V
entier i
début
V ← T[0]
pour i de 1 à N - 1 faire
  si T[i] > V alors
     V ← T[i]
  finsi
finpour
retour V
fin
```

Complexité des algorithmes

- **On s'affranchit des performances :**
  - **de la machine**
  - **du compilateur**
- **En ne considérant que le nombre d'instructions «haut niveau» exécutées : on parle alors de la complexité de l'algorithme en temps**
- **Prise en compte des données, difficile en général :**
  - **temps de réponse maximal**
  - **temps de réponse moyen**
  - **temps de réponse minimal**

```
fonction Max ( valeur réel T[],
               valeur entier N)
               retourne réel
réel V
entier i
début
1        V ← T[0]
N - 1    pour i de 1 à N-1 faire
           si T[i] > V alors
A            V ← T[i]
           finsi
         finpour
         retour V
         fin
```

2 N - 1

N-1+ H_N

N

$H_N = \sum_{1<=k<=N} 1/K$

Complexité asymptotique : Notation grand O

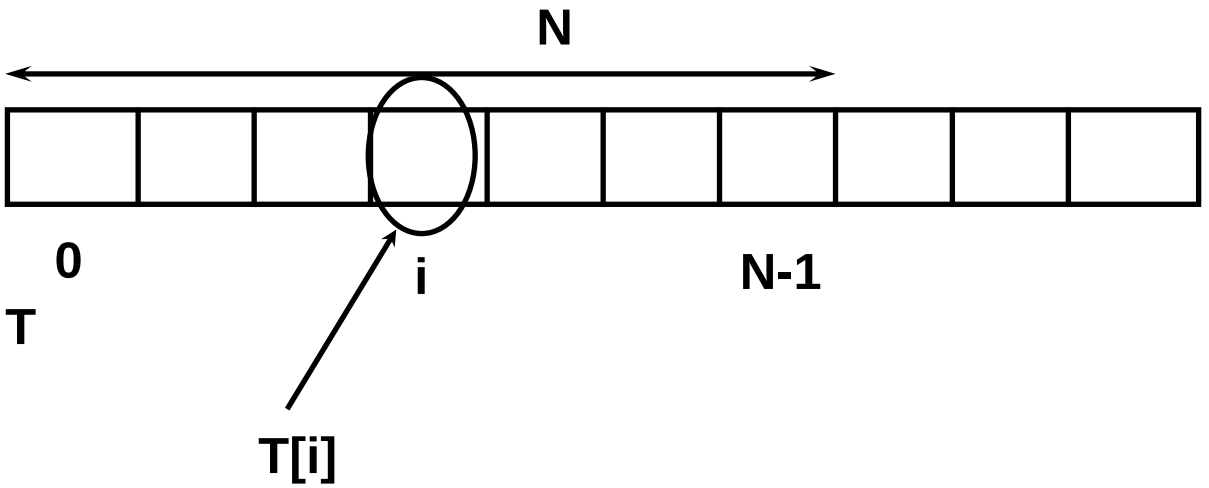
- La complexité est exprimée comme une fonction de la taille des données
- La complexité asymptotique s'intéresse à l'évolution du temps de réponse lorsque la taille des données tend vers l'infini
- Un programme est dit de complexité asymptotique f(n) et on note $O(f(n))$:
  - C(n)=complexité du programme. Si $\exists$ constante K telle que $C(n) <= K* f(n)$ quand $n \rightarrow \infty$
- On dit que le programme est de complexité $O(f(n))$
- Complexités remarquables :
  - constante : $O(1)$
  - linéaire : $O(n)$
  - quadratique : $O(n^2)$
  - logarithmique : $O(\log(n))$
  - exponentielle : $O(a^n)$ a>1
  - n log n : $O(n\log(n))$

La recherche du maximum est de complexité $O(n)$

Structure de données : Tableau

Un Tableau

- Ensemble d'éléments mémoire pouvant contenir des valeurs
- les valeurs appartiennent toutes à un même type : celui des éléments du tableau
- chaque élément est identifié par un indice



Déclaration de tableau

- **Nom du tableau**
- **Type des éléments**
- **Valeurs des indices :**
  - **Nombre d'éléments du tableau (C) : Nb**
    - » **indices de 0 à Nb-1**
  - **Plage des valeurs des indices (Pascal) :**
    - » **le nombre d'éléments est déterminé par le nombre de valeurs distinctes de la plage**
- **Tableaux non contraints :**
  - **le nombre d'éléments n'est pas indiqué**
  - **utilisés exclusivement comme paramètres de sous-programmes**
  - **intérêt : le paramètre effectif correspondant peut être de n'importe quelle taille**

```
entier T1[10]
```

```
entier T2[1..10]
```

```
procédure copie (
résultat caractère destination[ ],
valeur caractère source[ ])
```

Opérations sur un tableau

entier T[10], V

- **modification d'un élément** → **T[5] ← 50**
- **lecture d'un élément** → **V ← T[5]**
- **Certains langages permettent en outre (Ada) :**
  - **affectation globale**
  - **affectation de tranches**

Structure de données : Enregistrement

- **Un enregistrement est un ensemble d'éléments de mémorisation appelés des <u>champs</u>**

```
enreg Personne
  caractères nom[10]
  entier age
finenreg
```

- **Chaque champ est défini par :**
  - **un nom (identificateur)**
  - **un type**

Personne P
Personne Q

- **Opérations :**
  - **lecture/affectation globale**
  - **lecture/modification d'un champ**

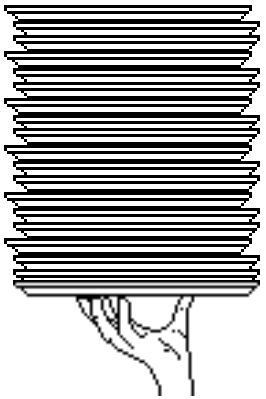
P ← Q

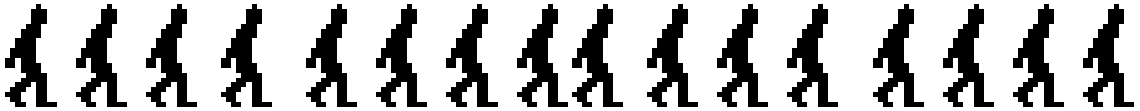
P.age ← Q.age

Modèle de données : Liste

- **Une liste est un ensemble fini d'éléments dont les propriétés sont déterminées par la position relative des éléments**

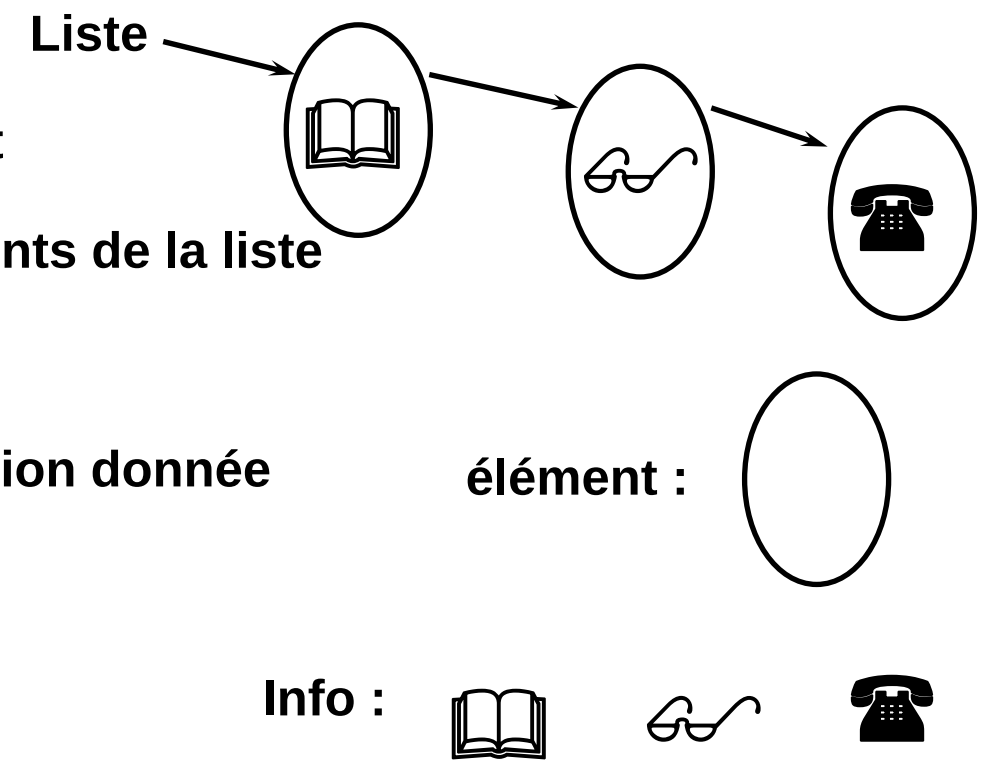
- **Une liste (non vide) est donc caractérisée par :**
  - **– son premier élément**
  - **– une relation de succession entre ses éléments**
  - **– son dernier élément**





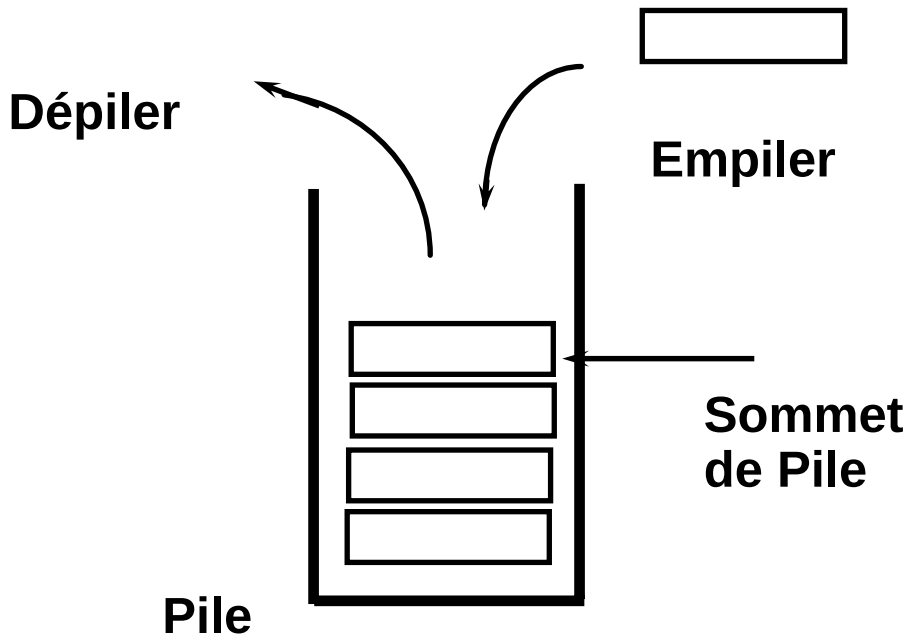
Opérations sur les listes

- **accéder à un élément à une position donnée (ex: premier élément, dernier élément, i-ème élément)**
- **accéder à l'élément suivant**
- **accéder à l'info portée par un élément**
- **parcourir séquentiellement des éléments de la liste**
- **rechercher un élément**
- **insérer un nouvel élément à une position donnée**
- **supprimer un élément**
- **fusionner plusieurs listes**
- **trier une liste, …**



Pile : définition

- **Modèle de rangement de données dans lequel :**
  - **la dernière donnée entrée est la première à sortir**
  - **pour obtenir une donnée, il faut enlever toutes celles qui sont «au dessus»**
- **Opérations :**
  - **Empiler : ajouter une donnée**
  - **Dépiler : retirer une donnée**
  - **SommetPile : lire l'information en sommet de pile**
  - **PileVide : teste si la pile est vide**
  - **PilePleine : teste si la pile est pleine**



File d'attente : définition

- **Modèle de rangement de données dans lequel :**
  - **la première donnée entrée est la première à sortir**
- **Opérations :**
  - **Ajouter : ajouter une donnée**
  - **Retirer : retirer une donnée**
  - **FileVide / FilePleine : teste si la file est vide / pleine**

Implémentation Pile : Tableau


```
enreg Tpile
  réel éléments[MAX]
  entier NbElts
finenreg
```

```
procédure initialiser (
        résultat Tpile pile)
```

```
procédure empiler (
        résultat Tpile pile,
        valeur réel E)
```

```
procédure dépiler (
        résultat Tpile pile,
        résultat réel E)
```

- **Réaliser les opérations :**
  - initialiser
  - empiler
  - dépiler
  - pilevide
  - pilepleine
  - sommetpile

```
fonction sommetpile (
        valeur Tpile pile
) retourne réel
```

```
fonction pilevide (
        valeur Tpile pile
) retourne entier
```

```
fonction pilepleine (
        valeur Tpile pile
) retourne entier
```

- **Complexité algorithmique**

File d'attente : Implémentation tableau


```
enreg Tfile
  réel éléments[MAX]
  entier id
  entier nb
finenreg
```

```
procédure initialiser (
        résultat Tfile file)
```

```
procédure ajouter (
        résultat Tfile file,
        valeur réel E)
```

```
procédure retirer (
        résultat Tfile file,
        résultat réel E)
```

- Réaliser les opérations :
  - initialiser
  - ajouter
  - retirer
  - filevide
  - filepleine

```
fonction filevide (
        valeur Tfile file
) retourne entier
```

```
fonction filepleine (
        valeur Tfile file
) retourne entier
```

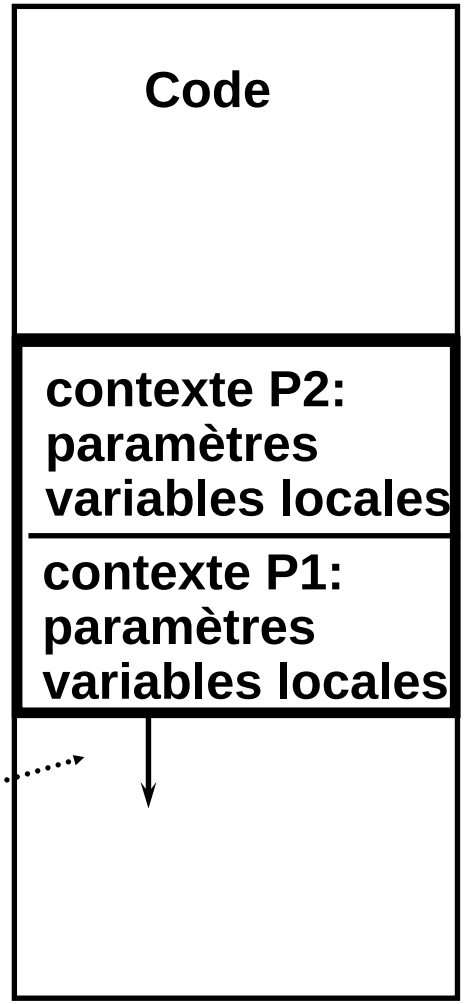
- En donner la complexité algorithmique

Séance 2 : Listes

- **Contexte d'exécution d'un programme**
- **Pile & Tas**
- **Mémoire dynamique**
  - **pointeurs**
  - **mémoire dynamique**
  - **allocation & libération**
- **Structure de données : listes chaînées**
- **Application : pile et file d'attente**

Evaluation des sous-programmes : Pile


```
procédure P1()
début
....
fin
procédure P2()
début
...
  P1()
...
fin
```



Pile

Sens de l'empilement des données

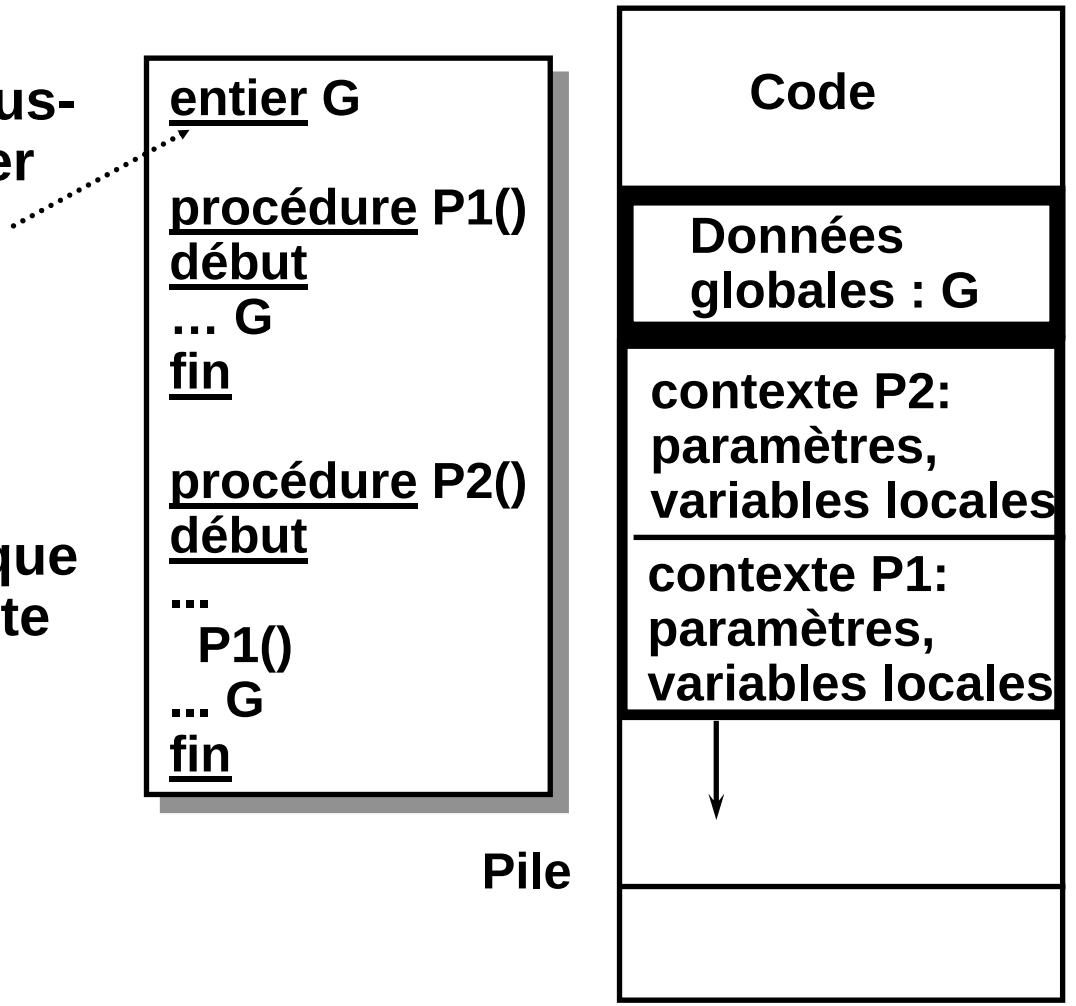
- L'évaluation des sous-programmes se fait sur une PILE
- A l'appel :
  - Empilement d'un contexte comprenant : paramètres et variables locales du sous-programme
  - Exécution du code du sous-programme
- Retour de sous-programme :
  - transmission des résultats éventuels à l'appelant
  - dépilement du contexte créé lors de l'appel
- Le sous-programme appelant reprend le cours de son exécution à l'instruction suivant l'appel

Variables Locales

- **Les paramètres et les variables locales sont alloués sur la pile lors de la création du contexte d'exécution**
- **Ils n'existent que pendant la durée d'exécution du sous-programme**
- **Les paramètres sont initialisés d'après le point d'appel**

Variables Globales

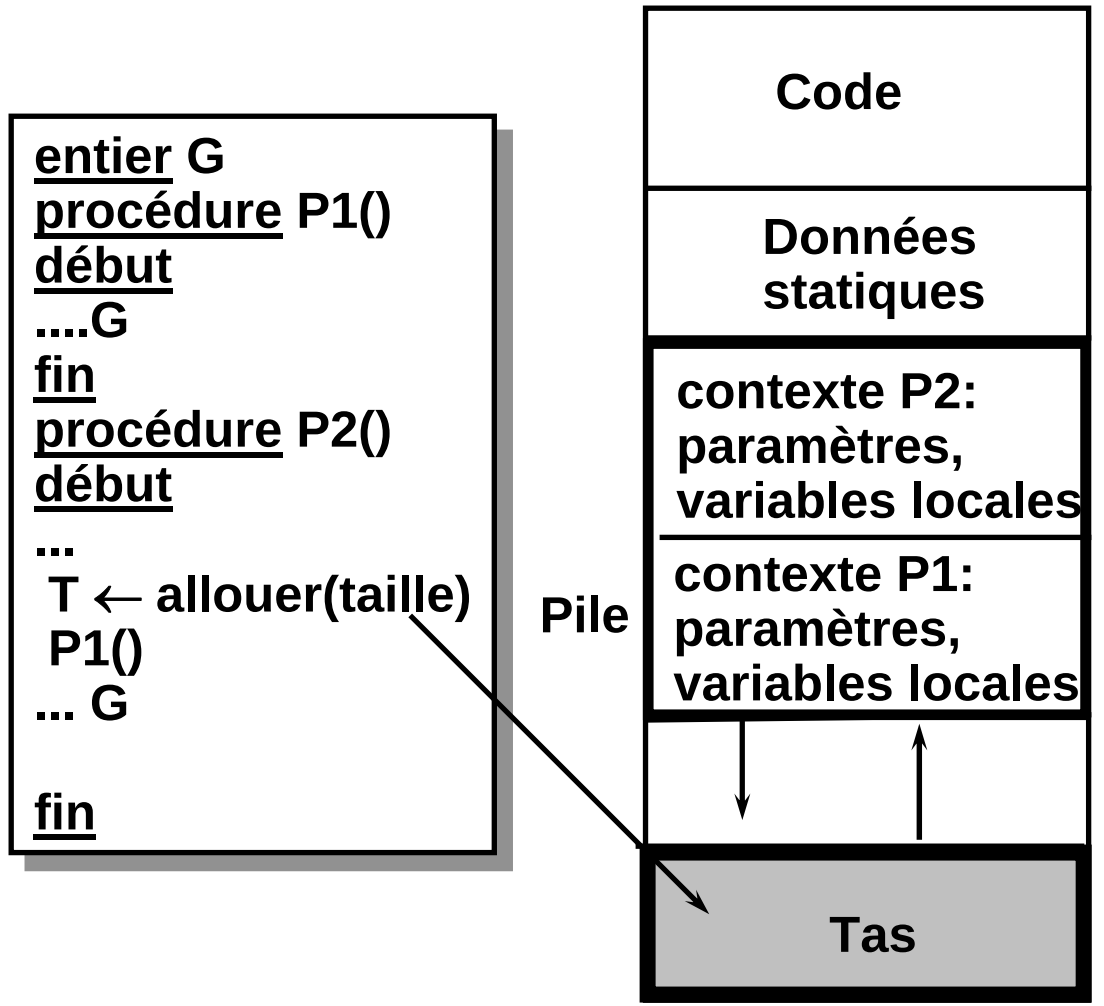
- **Lorsqu'on veut partager des informations entre plusieurs sous-programmes, on peut les stocker dans des variables globales**
- **Ces variables ne sont pas allouées sur la pile**
- **Une zone de mémoire dite statique est utilisée à cet effet qui persiste jusqu'à la fin de l'exécution du sous-programme**
- **La taille de cette zone est déterminée à la compilation du programme**



Le Tas

- **Lorsque la taille des données globales n'est pas connue à la compilation, elle doit être pourvue dynamiquement**
- **La mémoire ainsi allouée est prélevée dans une zone appelée le TAS**
- **Cette mémoire persiste jusqu'à ce qu'elle soit explicitement "libérée" ou jusqu'à la fin de l'exécution du programme**

```
entier G
procédure P1()
début
....G
fin
procédure P2()
début
...
 T ← allouer(taille)
 P1()
... G

fin
```



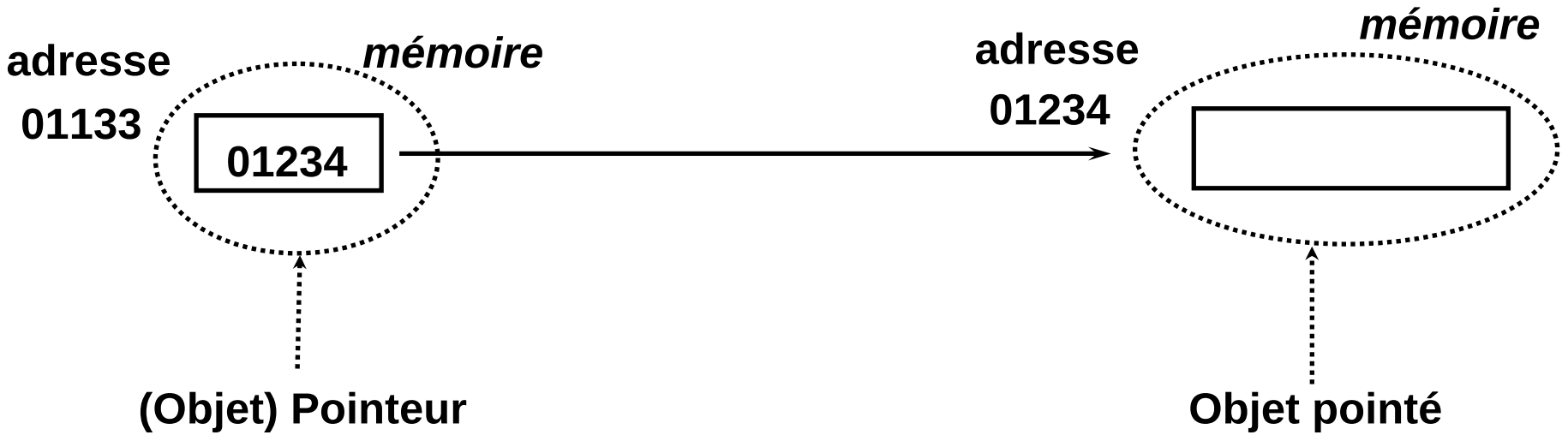
Mémoire dynamique

- **Variables Pointeurs**
- **Allocation dynamique de la mémoire**
- **Libération de la mémoire**

- **Application : Listes Chaînées**
  - **structure de données**
  - **opérations**
  - **pile & file**
  - **polynômes**

Variable Pointeur, Mémoire Pointée

- **Les objets (données variables ou constantes) manipulés par un programme sont généralement implantés en mémoire**
- **Chaque objet est caractérisé par une adresse mémoire**
- **Un pointeur est une variable qui peut contenir l'adresse mémoire d'un autre objet**
- **L'objet dont l'adresse est repérée par un pointeur s'appelle objet pointé**
- **La déclaration d'un pointeur implique d'indiquer le type des objets qu'il peut pointer**



Déclaration d'un Pointeur

- Une déclaration de pointeur doit :
  - indiquer le (ou les) noms des variables pointeurs
  - indiquer le type de l'objet pointé
  - préciser que la variable déclarée est un pointeur

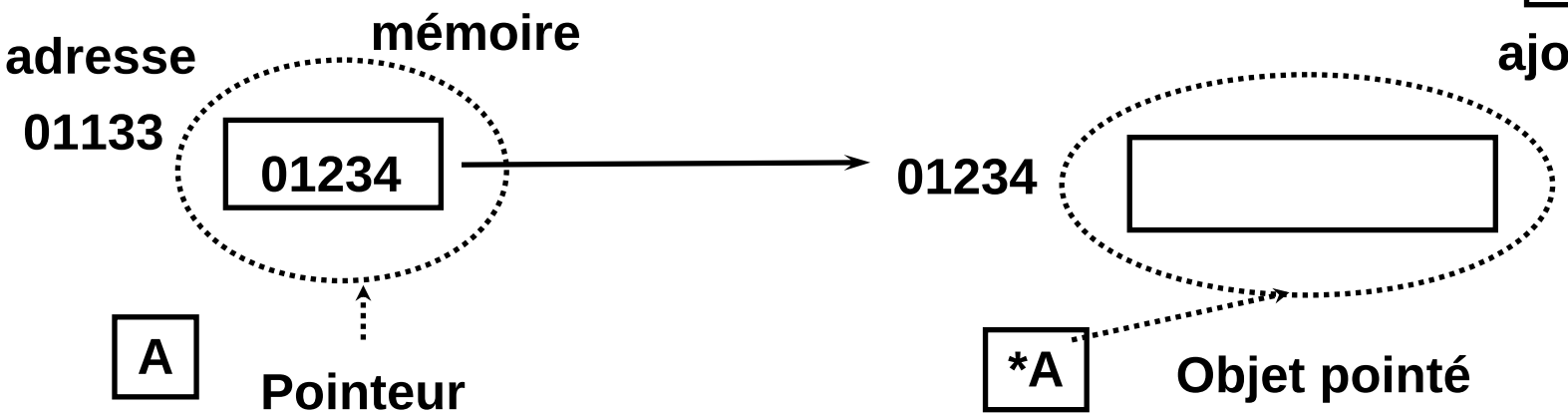
entier * A

- Plusieurs conventions suivant les langages; nous suivrons la convention "C"
- Une valeur particulière désigne une absence d'adresse : NUL

- Un pointeur doit être initialisé, et fournit une manière d'accéder à l'objet pointé

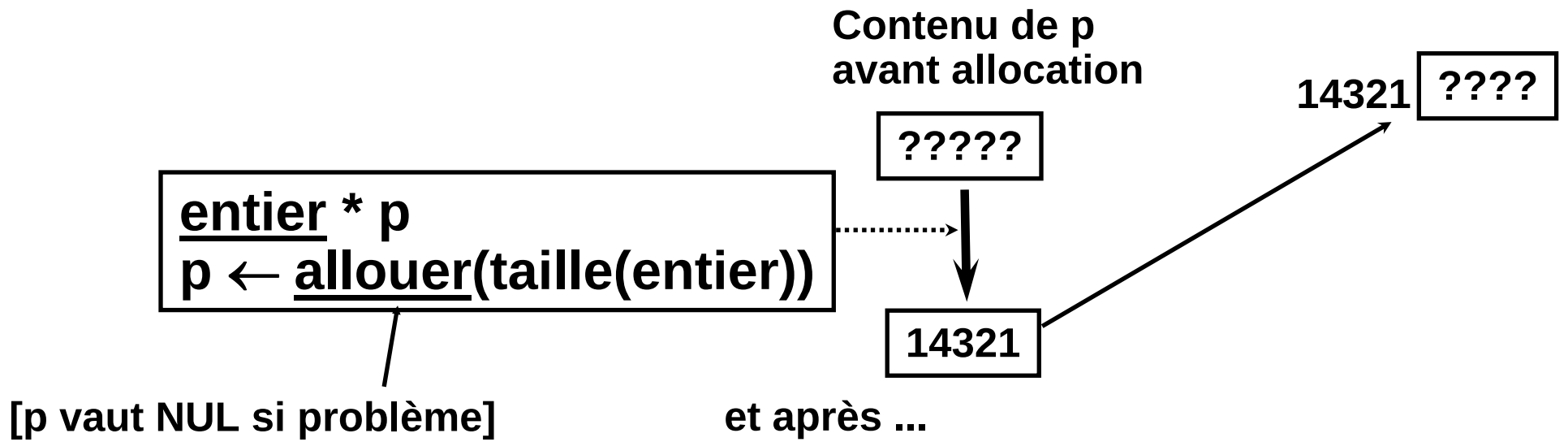
***A ← 12 + *A**

ajoute 12 à l'objet pointé



Allocation dynamique de la mémoire

- La plupart des langages informatiques permettent de réserver de la mémoire dynamiquement
- Une demande d'allocation doit préciser la taille de la zone mémoire à allouer
- Le système retourne l'adresse du début de la zone allouée
- Cette adresse est la seule clé d'accès à la zone allouée
- Elle doit être conservée précieusement dans un pointeur



Libération de la mémoire

- La mémoire allouée persiste jusqu'à ce que l'une des conditions suivantes soit remplie :
  - la zone allouée est explicitement libérée
  - le programme est fini
- La plupart des langages fournit un sous-programme pour libérer une zone précédemment allouée :
  - l'adresse retournée lors de l'allocation doit être fournie en paramètre

```
entier * p
p ← allouer(taille(entier))
.....
libérer(p)
```

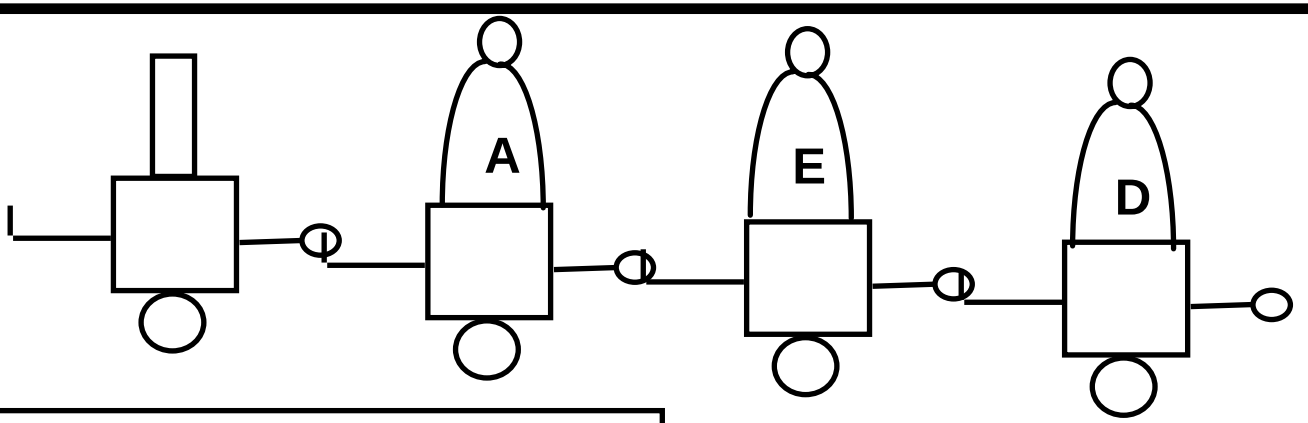
libération de la zone allouée

```
procédure T
entier * pt
début
   pt ← allouer(taille(entier))
   ... /* aucune libération de mémoire*/
fin
```

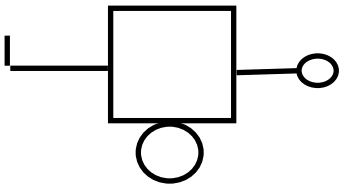
La mémoire allouée est perdue à tout jamais

Liste chaînée : définition, propriétés

Liste non vide

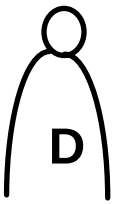


- **Dispositif de stockage d'un ensemble d'informations**
- **Chaque élément pointe vers le suivant sauf le dernier dont le champ suivant est NUL**
- **La liste est un pointeur :**
  - **sur le 1er élément si la liste n'est pas vide**
  - **NUL si la liste est vide**
- **Pour accéder à un élément, il faut parcourir la liste**

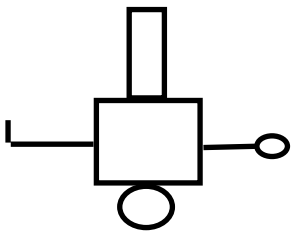


élément de la liste

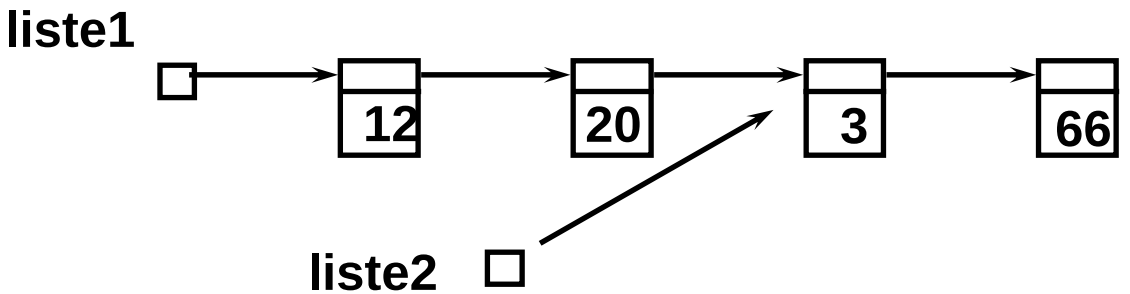
information



Liste vide



Structure de données : Liste Chaînée



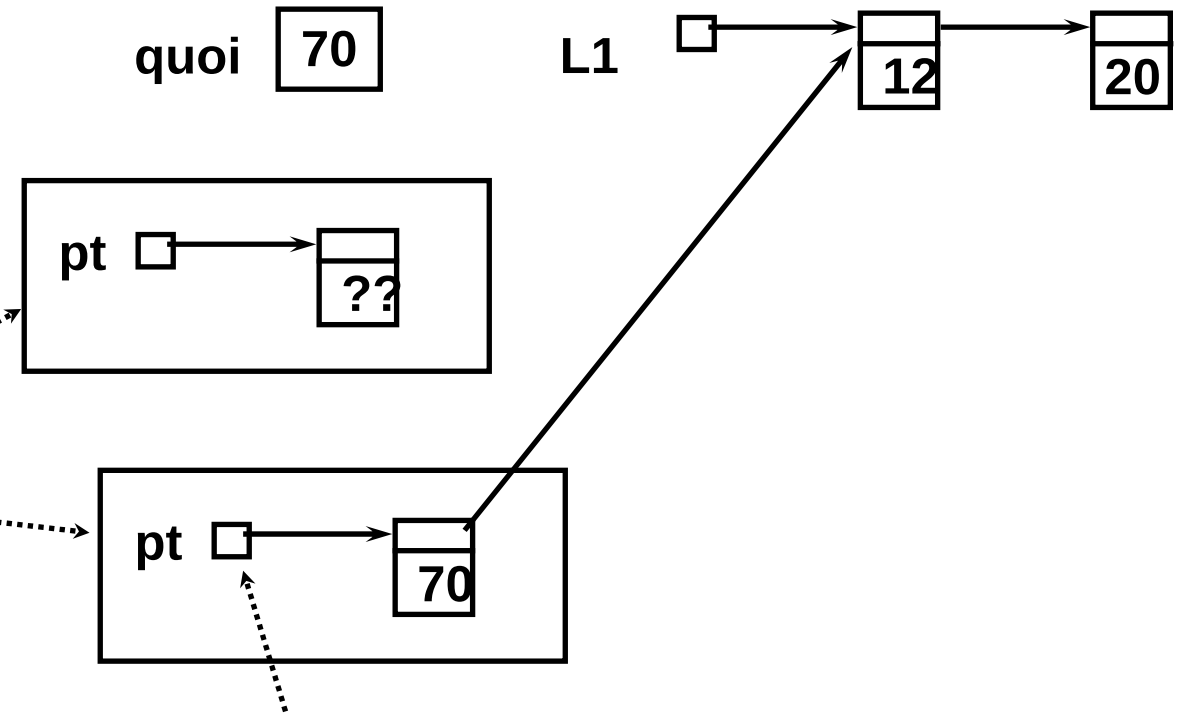

```
enreg TELEMENT
  TELEMENT *suivant
  TINFO info
finenreg
```

```
TELEMENT *liste1, *liste2
```

- Chaque élément est défini par
  - l'information qu'il contient
  - un pointeur sur l'élément suivant de la liste
- Une liste est définie par un pointeur sur son premier élément, appelé *tête*
- Les éléments d'une liste sont alloués dynamiquement lors de l'ajout d'un élément à la liste
- Le champ *suivant* du dernier élément vaut NUL
- La mémoire allouée est récupérée lors de la suppression d'un élément de la liste

Ajout d'un élément en tête de liste


```
fonction ajout_devant (
  valeur TELEMENT *L1,
  valeur entier quoi)
retourne TELEMENT *
TELEMENT *pt
début
  pt ← allouer(taille(TELEMENT))
  (*pt).info ← quoi
  (*pt).suivant ← L1
retour pt
fin
```



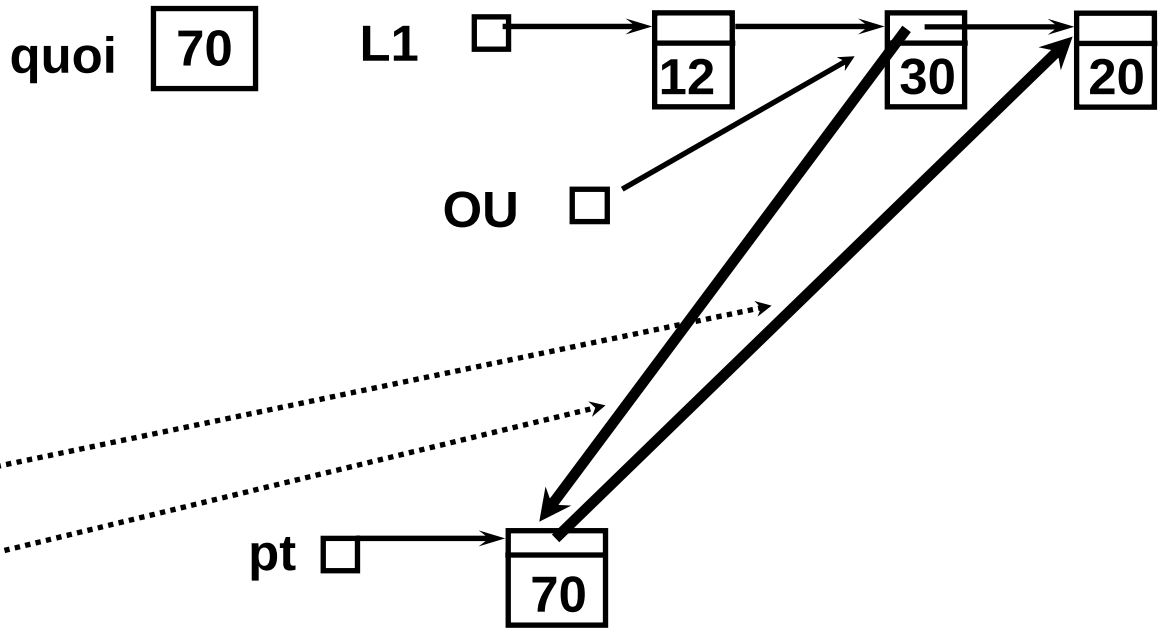
Valeur retournée : elle doit être affectée au pointeur désignant la liste

Exemple d'utilisation :

```
TELEMENT * Liste1 ← NUL
Liste1 ← ajout_devant (Liste1, 70)
```

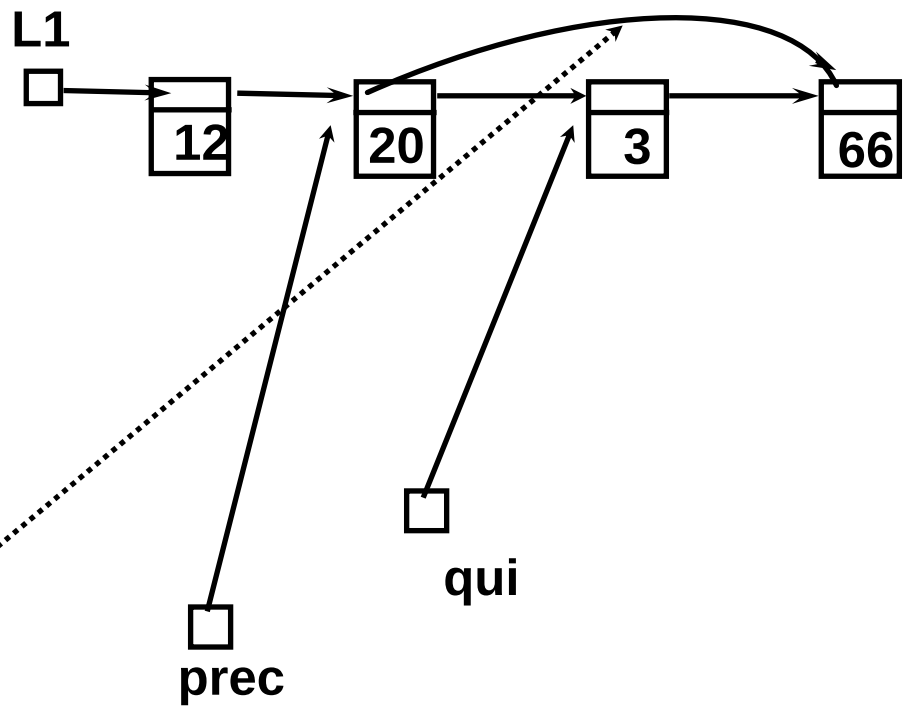
Ajout en une position donnée de la liste


```
procédure ajout_après (
  résultat TELEMENT *OU,
  valeur entier quoi)
  TELEMENT *pt
début
  pt ← allouer(taille(TELEMENT))
  (*pt).info ← quoi

  (*pt).suivant ← (*OU).suivant

  (*OU).suivant ← pt
fin
```



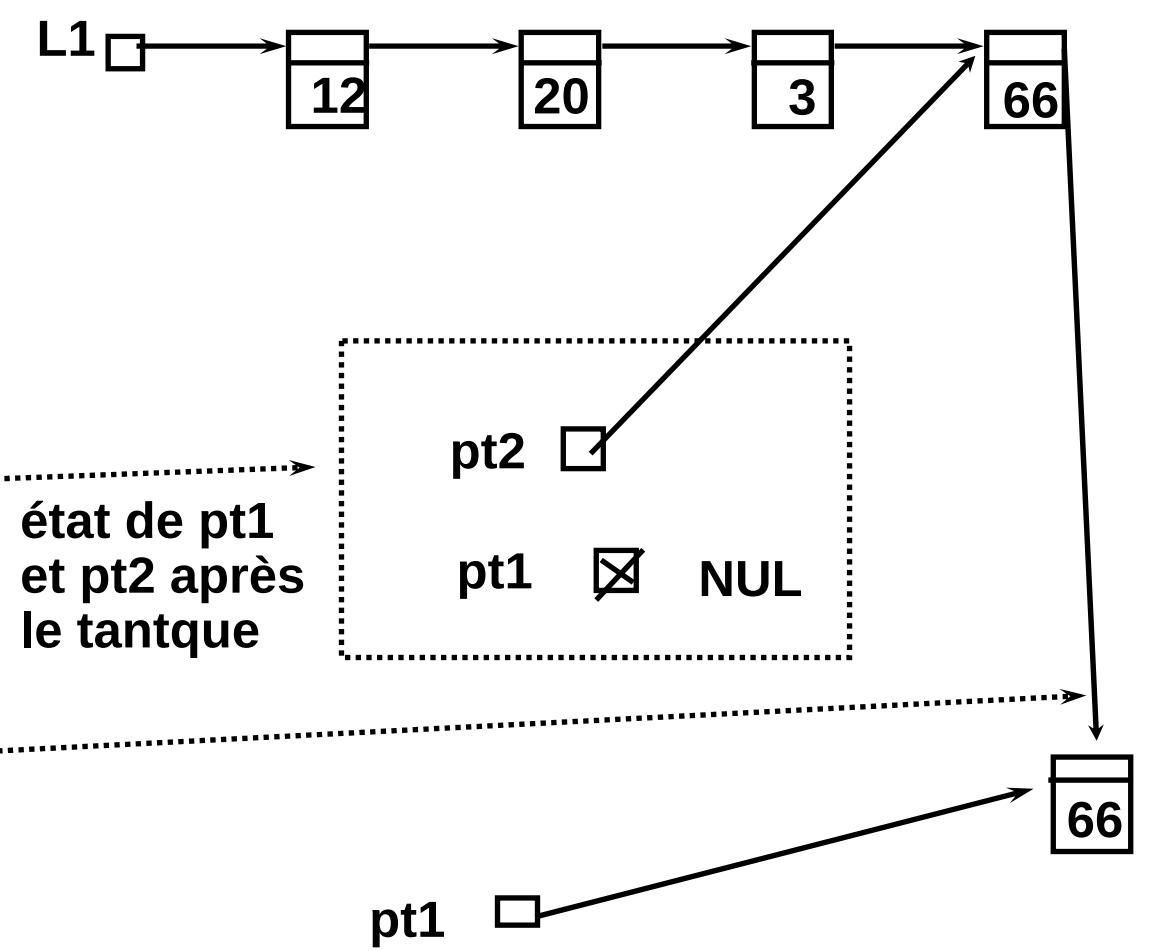
Supprimer un élément à une position


```
procédure supprimer (
 résultat TELEMENT *L1, valeur entier val)

 TELEMENT *prec, *qui
début
 qui ← L1        prec ← NUL
 tantque qui != NUL et (*qui).info != val faire
     prec ← qui
     qui ← (*qui ).suivant
 fintantque
 si qui = NUL alors écrire ("pas dans la liste")
 sinon
     si prec = NUL alors L1 ← (*qui ).suivant
     sinon (*prec ).suivant ← (*qui).suivant
     finsi
     libérer(qui)
 finsi
fin
```



Ajout en fin de liste


```
fonction ajout_fin (
  valeur TELEMENT *L1,
  entier quoi) retourne TELEMENT *
TELEMENT *pt1, *pt2
début
  pt1 ← L1    pt2 ← L1
  tantque pt1 != NUL faire
    pt2 ← pt1
    pt1← (*pt1).suivant
  fintantque
  pt1 ← allouer(taille(TELEMENT))
  (*pt1).info ← quoi
  (*pt1).suivant ← NUL
  si pt2 = NUL alors retour pt1
  sinon (*pt2).suivant ← pt1
        retour L1
  finsi
fin
```



Exo1 : Pile à l'aide de listes chaînées

- **Définir une structure de données Tpile permettant d'implanter les piles de réels sous forme de liste chaînée**

- **Donner l'algorithme des sous-programmes :**
  - **procédure empiler (résultat Tpile * pile, valeur réel val)**
  - **procédure dépiler (résultat Tpile * pile, résultat réel val)**
  - **fonction pilevide (valeur Tpile *pile) retourne entier**

- **Écrire un algorithme d'évaluation des expressions en notation polonaise inversée**

Exo2 : File d'attente & listes chaînées

Proposez une structure de donnée Tqueue permettant de réaliser efficacement une file d'attente d'entiers

Donner ensuite les algorithmes pour les opérations :

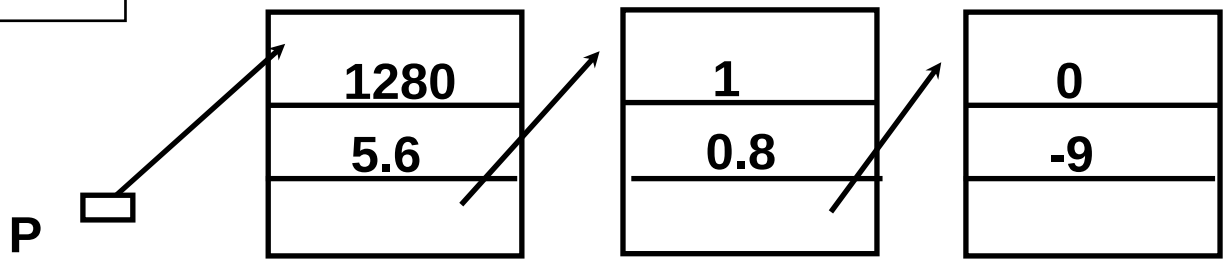
- **procédure Ajouter (résultat Tqueue *queue, valeur entier info)**
- **procédure Retirer (résultat Tqueue *queue, résultat entier val)**
- **fonction Vide (valeur Tqueue *queue) retourne entier**

Listes chaînées : polynômes creux

Exemple : $P(x) = 5.6x^{1280} + 0.8x - 9$

```
enreg TMonome
  int deg;
  float coef;
  TMonome *suiv;
finenreg
```

TMonome * P



Représentation en mémoire de P après allocation des différents monômes

Listes chaînées : gestion d'une promo

- On désire gérer les informations concernant les élèves d'une classe; chacun des élèves est caractérisé par :
  - son nom : 40 caractères maximum, son age, son numéro de sécu
  - et ses notes dans les matières : maths, physique, informatique (ces notes sont définies par les lettres : 'A', 'B', 'C', 'D', 'E')
- La base de données doit permettre :
  - de saisir un élève
  - d'afficher les caractéristiques d'un élève à partir de son numéro de sécu
  - d'afficher les notes des élèves dans une matière par groupe de mérite: ceux qui ont 'A', ensuite 'B',...
  - d'afficher les notes dans toutes les matières en privilégiant dans l'ordre : info, maths, physique : d'abord le groupe des meilleurs en info. Parmi ceux-ci d'abord les meilleurs en maths; parmi ceux-ci d'abord les meilleurs en physique,...
- On demande de proposer une solution à ce problème en utilisant des listes chaînées

Séance 3 : Récursivité

- **Induction mathématique**
- **Récursivité**
- **Application : Tri rapide**

Induction mathématique

Technique permettant de prouver qu'une assertion A(n) est vraie pour tous les entiers naturels ou pour tous les entiers à partir d'un rang n_0
Elle procède en 2 étapes :
- **Base : prouver que $A(n_0)$ est vraie**
- **Récurrence : prouver que :**
  - **si A(n) est vraie pour n naturel donné, alors A(n+1) est vraie**

Alternative

- **Base : prouver $A(n_0)$, $A(n_0+1)$, …, $A(n_0+K)$**
- **Récurrence : prouver que**
  - **si A(m) est vraie pour tout m < n, pour n donné, alors A(n) est vraie**

Applications :
- **Propriétés des programmes**
- **Conception d'algorithmes (définitions récursives)**
- **Performances des algorithmes**

Exercice 1

N° 41

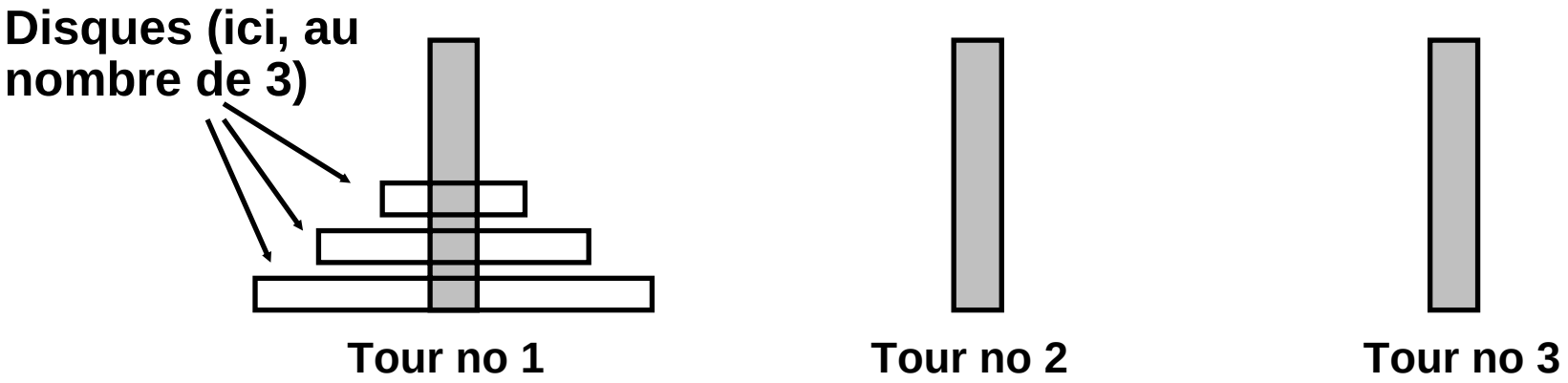
Prouver que, pour n >= 1, on a

$$P(n) = \sum_{k=1}^{n} (2k - 1) = n^2$$

Que peut-on en déduire?

Exercice 2 : Les tours de Hanoi (1)

N° 42

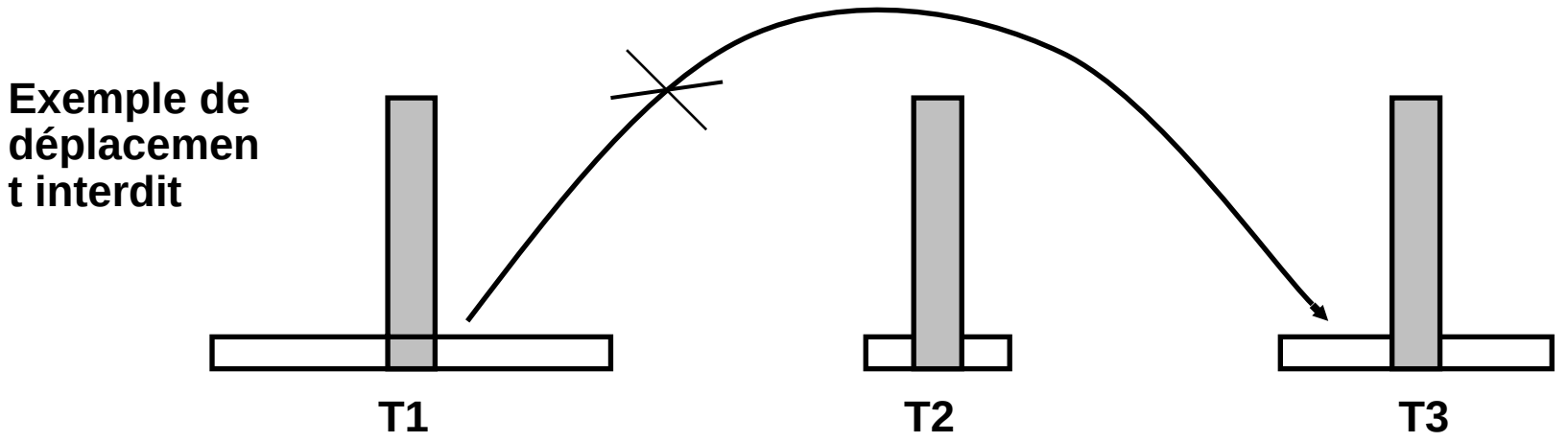


But du jeu : faire passer les disques de la tour no 1 à la tour no 3 en minimisant le nombre de coups nécessaires

Règle du jeu : on ne peut déplacer qu 'un seul disque à la fois (1 coup), en utilisant la tour no 2 de sorte qu'après chaque coup, aucun disque ne soit empilé sur un disque de plus petite dimension

Exercice 2 : Les tours de Hanoi (2)

N° 43



Déterminer:

- **C(N) en fonction de C(N-1), où C(I) désigne le nombre de coups (minimum) nécessaires à un jeu comportant I disques**
- **T(N) en fonction de N, où T(N) est le temps nécessaire pour mener à bien un jeu comportant N disques (on considère que le temps moyen pour réaliser un déplacement est de 1 seconde)**
- **T(12)**

Application : analyse des algorithmes

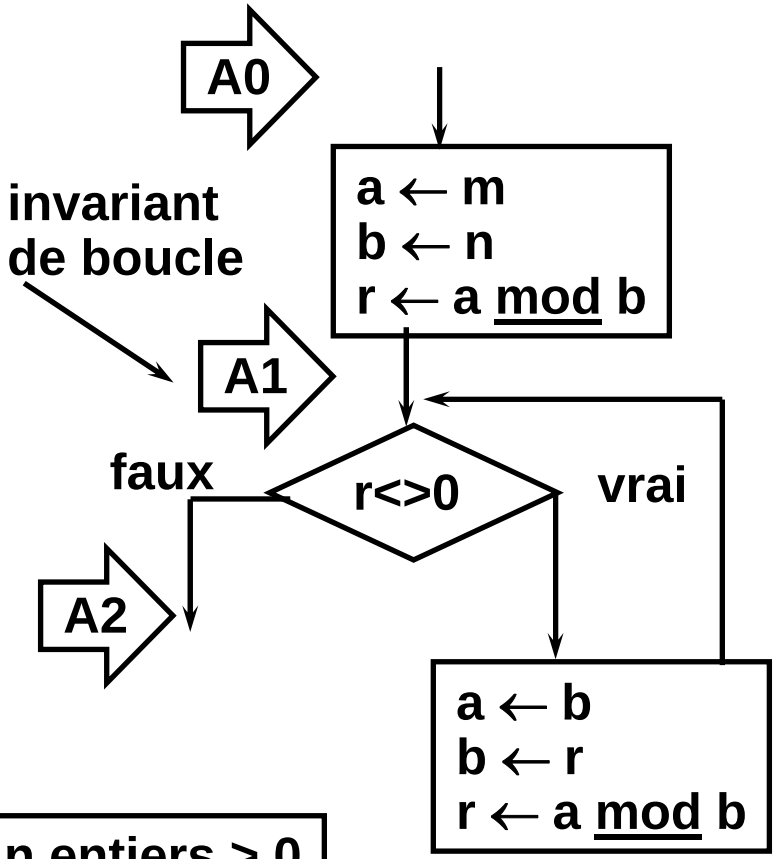
- **L'induction mathématique est utilisée pour prouver des propriétés sur la validité des algorithmes, notamment itératifs et récursifs**

- **Cela est d'autant plus cohérent que toute l'information manipulée par un ordinateur est discrète**

- **Trois propriétés essentielles sont généralement analysées :**
  - **la validité des algorithme : l'algorithme est il correct ?**
  - **preuve d'arrêt : l'algorithme s'arrête-il après un nombre fini d'étapes ?**
  - **performances des algorithmes :**
    - **complexité en nombre d'opérations**
    - **complexité en taille des données mémorisées**

Validité

- Le flot de l'algorithme est vu comme un ensemble de blocs séquentiels reliés par des flèches et des conditions traduisant le flot de contrôle
- Étiqueter le flot de l'algorithme avec des propriétés judicieusement choisies
- Prouver que ces propriétés sont vraies à chaque passage en ces points, c'est à dire :
  si les propriétés attachées aux flèches entrant dans le bloc sont vraies, alors les propriétés attachées aux flèches sortant du bloc seront vraies après exécution du bloc



A0: m,n entiers > 0

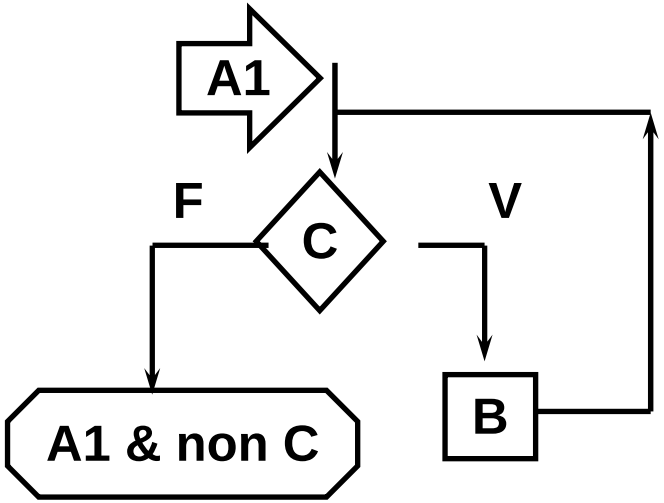
A1: pgcd(m,n) = pgcd(a,b) = pgcd(b,r)

A2: A1 et r=0 <=> pgcd(m,n)=b

Invariants de Boucle

- Les assertions associées à l'entrée d'une boucle sont appelées des invariants de boucle tantque
- On les démontre généralement par récurrence sur le nombre d'itérations
- En sortie de boucle :
  - l'invariant de boucle est toujours vrai
  - la condition de boucle est fausse

- Recommandation :
  - lors de la conception d'un algorithme, il est utile de se définir un invariant de boucle pour faciliter les preuves de validité de l'algorithme
  - cet invariant de boucle est une excellente façon de documenter l'algorithme



Définitions récursives

- **Les définitions récursives constituent une deuxième application de l'induction mathématique**
- **Le principe est de définir une série de traitements, ou d'objets, en deux étapes:**
  - **Base : on définit des objets ou des traitements de base**
  - **Induction : on définit des objets ou des traitements plus complexes en terme des objets ou des traitements déjà définis**
- **Ces définitions récursives se traduisent naturellement en algorithmes récursifs dans lesquels un sous-programme peut s'appeler lui-même avec un jeu de paramètres différents**

Exemples :

Base : 1! = 1, 0 != 1
Récurrence: n! = n* (n-1)!

Langage d'expressions arithmétiques E
Base : E = nombres entiers, nombres réels, variables
Récurrence :
(E1+E2), (E1-E2), (E1/E2), (E1*E2)

Application : sous-programmes récursifs

- **Un sous-programme est récursif lorsqu'il s'appelle lui-même**

- **La récursivité peut être :**
  - **directe**
  - **indirecte : si l'appel récursif n'est pas directement contenu dans le sous-programme, mais inséré dans un autre sous-programme appelé par le premier**
  - **croisée lorsque un ou plusieurs sous-programmes s'appellent mutuellement**

```
fonction Factorielle(valeur entier n)
retourne entier
début
   si n=0 ou n=1 alors retour 1
   sinon retour n * Factorielle(n-1)
   fsi
fin
```

Appel récursif

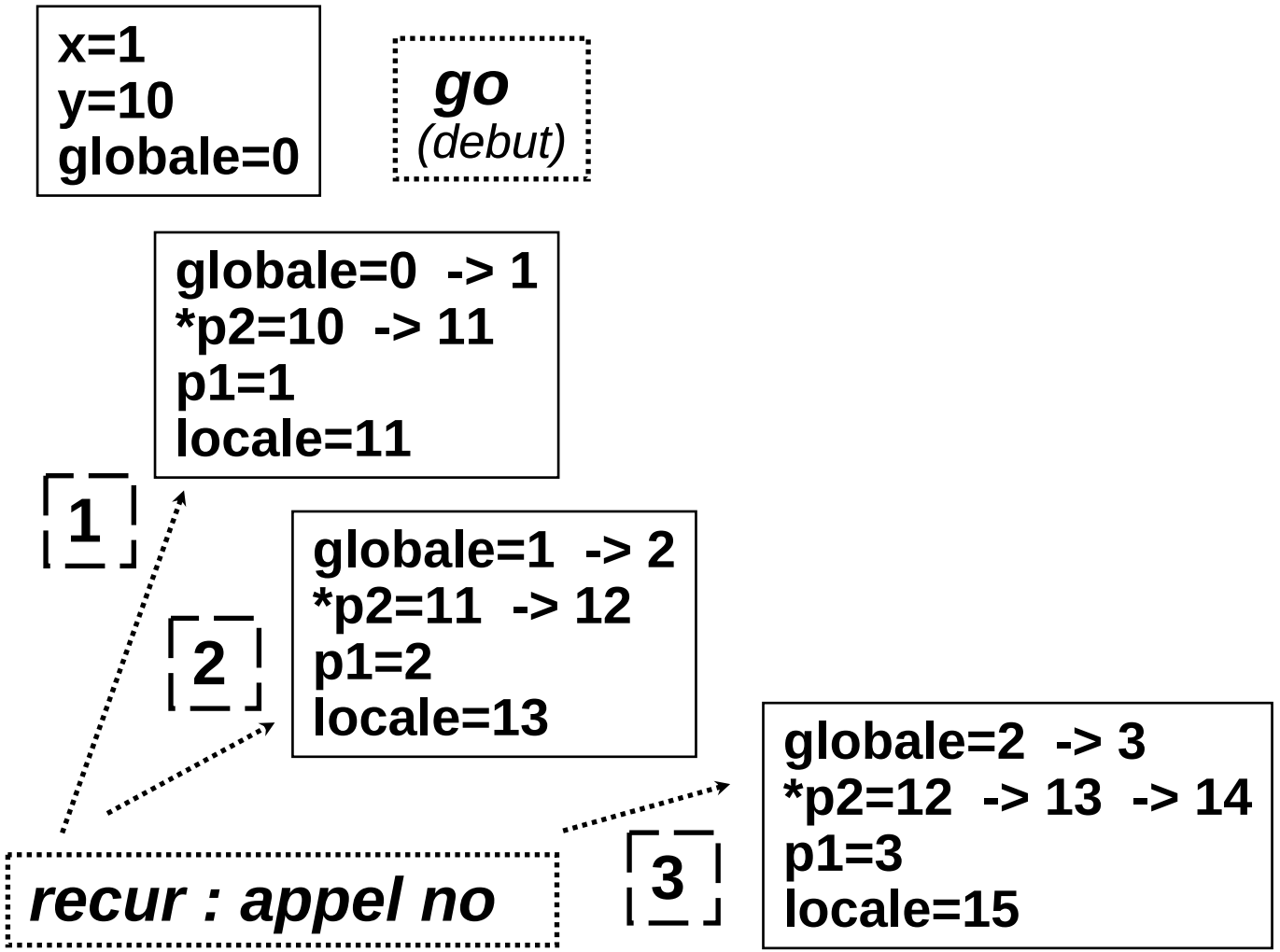
Récursivité directe : Exemple (1)


```
entier globale

procédure go ()
entier x, y
début
  x ← 1
  y ← 10
  globale ← 0
  recur (x, y)
  afficher (x, y, globale)
fin
```

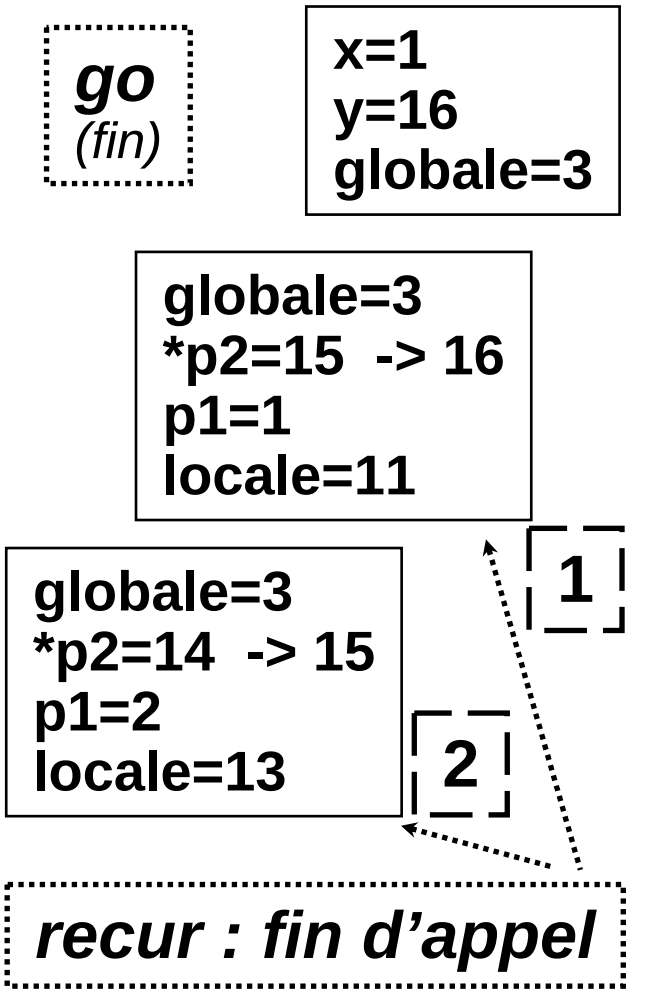
attention, p2 va "évoluer"

```
procédure recur (valeur entier p1,
        résultat entier p2)
entier locale
début
  locale ← p1 + p2
  p2 ← p2 + 1
  globale ← globale + 1
  si locale <= 13 alors recur (p1 + 1, p2)
  p2 ← p2 + 1
fin
```

Récursivité directe : Exemple (2)

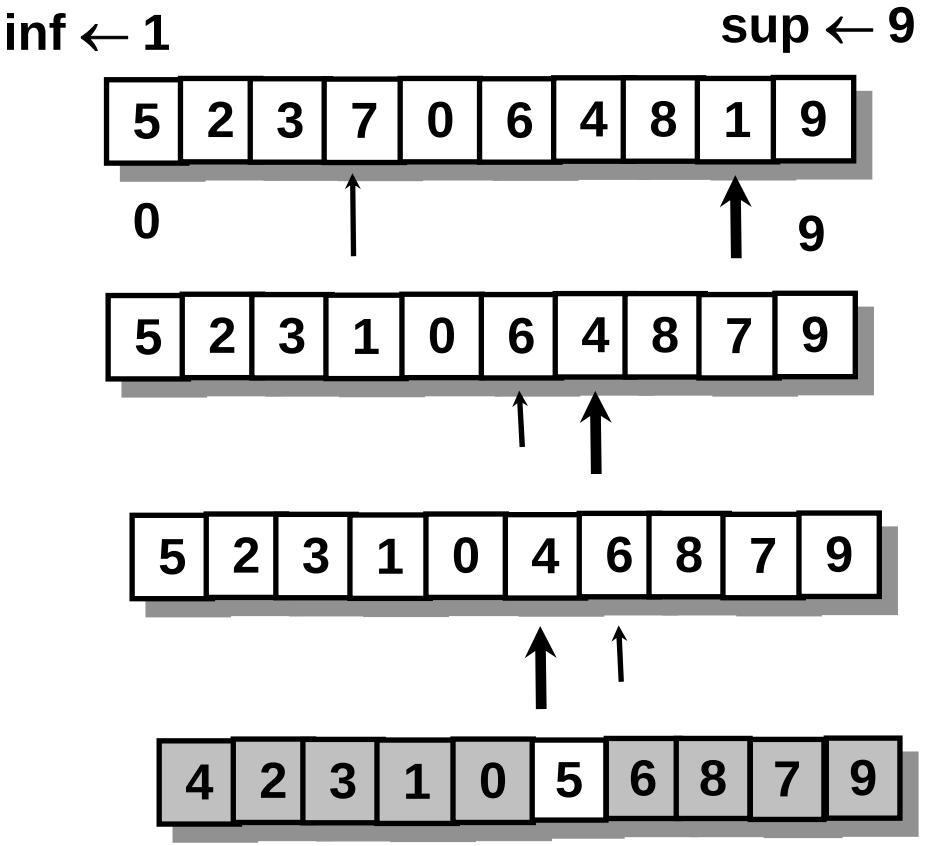


Récursivité directe : Exemple (3)



Application : Tri Quick-Sort

- **On recherche la position finale (après tri) du 1er élément du tableau :**
  - **partant de 9, on décrémente sup jusqu'à T[sup] < T[0]**
  - **partant de 1 on incrémente inf jusqu'à T[inf] > T[0]**
  - **on permute T[inf] et T[sup] et on recommence jusqu'à ce que inf >= sup**
- **On permute T[0] et T[sup]**
- **On applique les 2 points précédents à T[0 .. (sup-1)] et T[(sup+1) .. 9]**



procédure QuickSort (**résultat** entier T[], **valeur** entier BInf, **valeur** entier BSup)

Complexité des algorithmes récursifs

- **Elle est généralement définie par une équation de récurrence liant la complexité globale et celle des appels récursifs**
- **La détermination de cette complexité passe par la résolution des équations de récurrences. Exemples :**
  - **$T(n) = T(n-1) + bn^k \Rightarrow T(n)$ de complexité $O(n^{k+1})$**
  - **$T(n) = cT(n-1) + bn^k$ $c>1 \Rightarrow T(n)$ de complexité $O(c^n)$**
  - **$T(n) = cT(n/d) + bn^k$ $c>d^k \Rightarrow T(n)$ de complexité $O(n^{\log c/\log d})$**
  - **$T(n) = cT(n-1) + bn^k$ $c<d^k \Rightarrow T(n)$ de complexité $O(n^k)$**
  - **$T(n) = cT(n-1) + bn^k$ $c=d^k \Rightarrow T(n)$ de complexité $O(n^{k+1}\log n)$**

Exercice : Déterminez la complexité du tri par sélection et du tri rapide

Séance IV : Exercice récapitulatif

Tri par Fusion

- **Principe, Exemple**
- **Algorithmes**
- **Structures de données à utiliser**

Tri par fusion (1) : Principe

Soit S_0 la file 7 4 10 13 6 50 2 67 11

On décompose S_0 en M_i ensembles d'éléments ordonnés

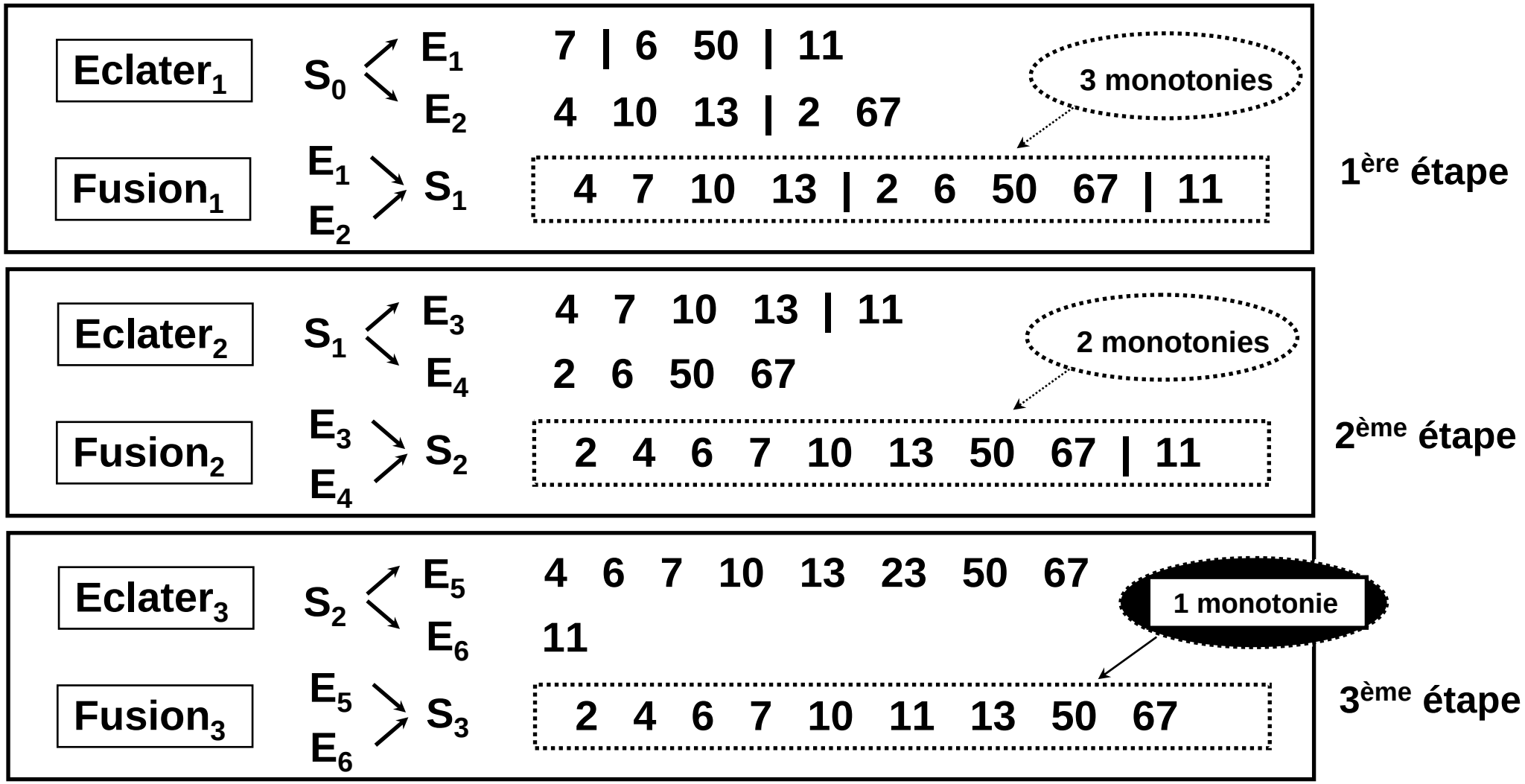
7 | 4 10 13 | 6 50 | 2 67 | 11 → 5 M_i

L'algorithme consiste, à chaque étape j

- à éclater les M_i monotonies en 2 sous-ensembles
  $\{ M_1, M_3, ..., M_{2p+1}\}$
  $\{ M_2, M_4, ..., M_{2p+2}\}$
- puis à fusionner dans S_j les M_{2p+1} et M_{2p+2} 2 à 2 en les ordonnant

jusqu'à ce que S_j soit ordonné

Tri par fusion (2) : Exemple



Tri par fusion (3) : Algorithmes

"Procédures" à écrire

- **Transfert_monotonie : transfère la monotonie tête de la file L1 en queue de la file L2**
- **Eclater : répartit les monotonies d'une file L entre 2 files L1 et L2**
- **Fusion_monotonie : retire les monotonies de tête des files L1 et L2 et les fusionne en ordonnant leurs éléments dans une file L3**
- **Fusionner : fusionne dans une file L les monotonies de 2 files L1 et L2**
- **Tri_par_fusion : réalise le ... tri par fusion d'une file L**

Tri par fusion (4) : Structures de données

Les algorithmes à développer doivent l'être indépendamment des structures physiques sur lesquelles ils s'appliquent

La structure de base à utiliser est donc un *TFile*

Lors du passage à la programmation, les algorithmes développés (sous forme d'un module à part) devront pouvoir accepter en paramètre et indifféremment des

- **tableaux**
- **listes chaînées**
- **fichiers**

d'éléments entiers

Éléments complémentaires d'Algorithmique

- **Grammaires**
- **Arbres**
- **Récursivité (bis)**

Définitions Récursives : Grammaire

- **Description d'un ensemble potentiellement infini des phrases d'un langage par un ensemble fini de règles**
- **Syntaxe : forme des phrases d'un langage abstraction faite de leur signification**
- **Sémantique : signification des phrases**

Une grammaire est définie par un quadruplet (V_T, V_N, S, R)**:**

- V_T **: un ensemble de symboles terminaux appelé alphabet**
- V_N **: un ensemble de symboles non terminaux**
- S **: symbole particulier de** V_N **appelé axiome ou symbole racine de la grammaire**
- R **: un ensemble de règles de production de la forme** $a \rightarrow b$
  où a et b sont des séquences de symboles appartenant à $V_T \cup V_N$

$$V = V_T \cup V_N$$

Les langages de programmation sont décrits par des grammaires
Des méthodes automatiques permettent de générer des analyseurs
à partir d'une grammaire donnée

Grammaires: exemple

$G = (V_T, V_N, S, R)$

axiome : S

$V_T = \{ a, b\}$

$V_N = \{S, T\}$

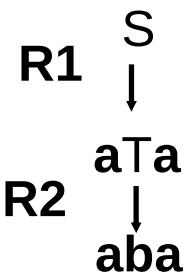
$R = \{ R1, R2, R3\}$

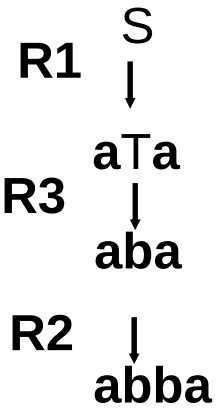
L(G) = ensemble de toutes les phrases du langage

R1 : $S \rightarrow$ **aTa**
R2 : $T \rightarrow$ **b**
R3 : $T \rightarrow$ **bT**

une phrase appartient à L(G) si on peut la dériver à partir de l'axiome en appliquant les règles de dérivation

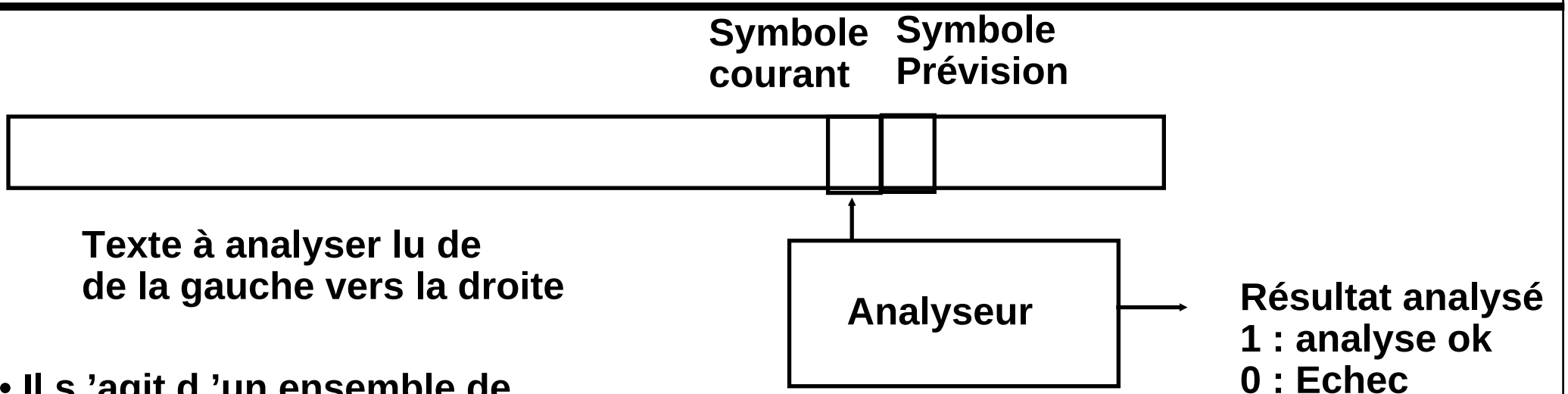
aba $\in$ **L(G) car on a**





Quelles sont les phrases de L(G) ?

Grammaire: Analyseur LL(k)



- Il s 'agit d 'un ensemble de sous-programmes récursifs
- Chaque symbole non terminal est associé à un sous-programme
- Un sous-programme vérifie si les règles relatives au symbole non terminal auquel il est lié s'appliquent

$R = \{S \rightarrow \mathbf{a}T\mathbf{a},\ T \rightarrow \mathbf{b},\ T \rightarrow \mathbf{b}T\}$

Grammaire LL(k) si la propriété est vraie

En anticipant sur la valeur des k symboles d'entrées à venir, le sous-programme "sait" quelle est la règle à appliquer

Exemple : Analyseur LL(1)

R1 : S → **a**T**a**
R2 : T→ **b**
R3 : T→ **b**T

```
fonction T retour entier
entier val
caractère carcour
début
   si Prévision() <> 'b' alors
      erreur(« 'b' attendu ») retour 0
   finsi
   carcour ← LireCaractère()
   si Prévision() = 'b' alors
      retour T()
   sinon
      retour 1
   finsi
fin
```

```
fonction S retour entier
entier val
caractère carcour
début
   si Prévision() <>'a' alors
      erreur(« 'a' attendu ») retour 0
   finsi
   carcour ← LireCaractère()
   val ← T()
   si val = 0 alors retour 0
   finsi
   si Prévision() <>'a' alors
      erreur(«  'a ' attendu ») retour 0;
   finsi
   carcour ← LireCaractère()
   retour 1
fin
```

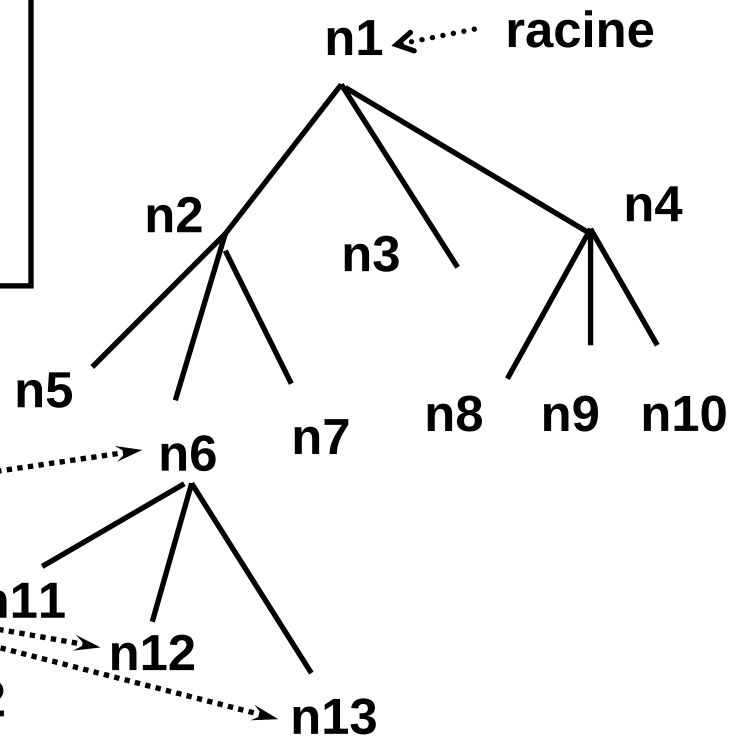
Expressions arithmétiques

E1 : Expression → Terme **+** Expression
E2 : Expression → Terme **-** Expression
E3 : Expression → Terme
T1 : Terme → Facteur ***** Terme
T2 : Terme → Facteur **/** Terme
T3 : Terme → Facteur
F1 : Facteur → **(**Expression**)**
F2 : Facteur → Nombre
N1 : Nombre → Digits
N2 : Nombre → Digits Nombre
D1 : Digits → **'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'**

Exercice :
1) écrire un analyseur pour cette grammaire
2) en déduire un programme qui calcule les expressions reconnues
3) en déduire un générateur de code pour une machine à pile

Modèle de données : Arbre

Un arbre est un ensemble de nœuds structurés ainsi :
- **Il existe un nœud appelé racine**
- **Le reste des nœuds, s'il y en a, est partitionné en un ensemble d'arbres appelés sous-arbres de la racine; on dit qu'il existe un arc entre la racine et les racines des sous-arbres**



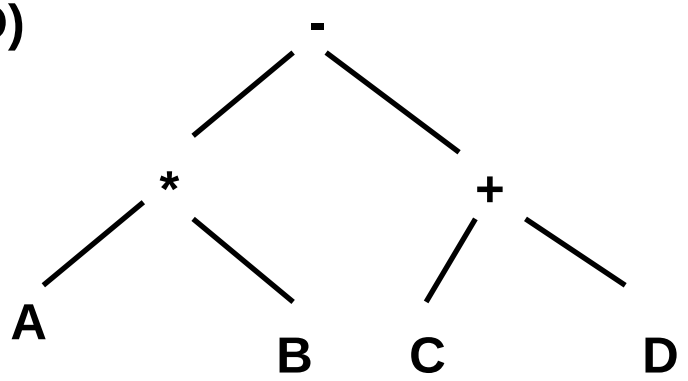
degré d'un nœud N : nombre de sous-arbres de N
feuille : arbre de degré 0
liste linéaire : arbre dont tous les nœuds ont un degré <=1

arbre binaire : arbre dont le degré max des nœuds est 2

profondeur :
profondeur(racine) = 0
profondeur(nœud) = 1 + profondeur(père)

Arbres (1) : un exemple

(A*B)-(C+D)



Représentation des expressions arithmétiques :
- **les nœuds intérieurs (degré>0) sont des opérations**
- **les feuilles (degré=0) sont des nombres ou des variables**

Transformation d'un arbre binaire d'expression en une liste :
<fils gauche> <opération> <fils droit>: expression en notation infixée e(A*B)-(C+D)
<opération> <fils gauche> <fils droit>: expression en notation préfixée -*AB+CD
<fils gauche> <fils droit> <opération>: expression en notation postfixée AB*CD+-

Arbres (2) : représentation


```
struct TNOEUD {
   TINFO Info;
   struct TNOEUD * fils[Max];
}
```

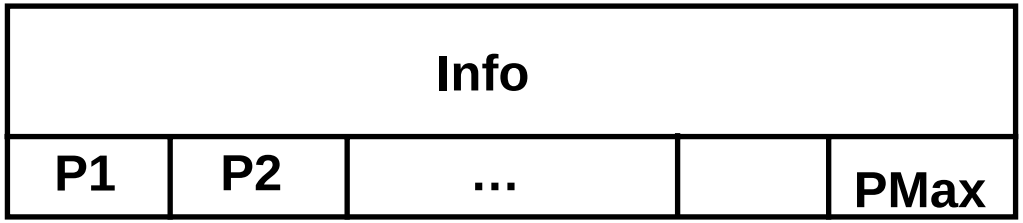
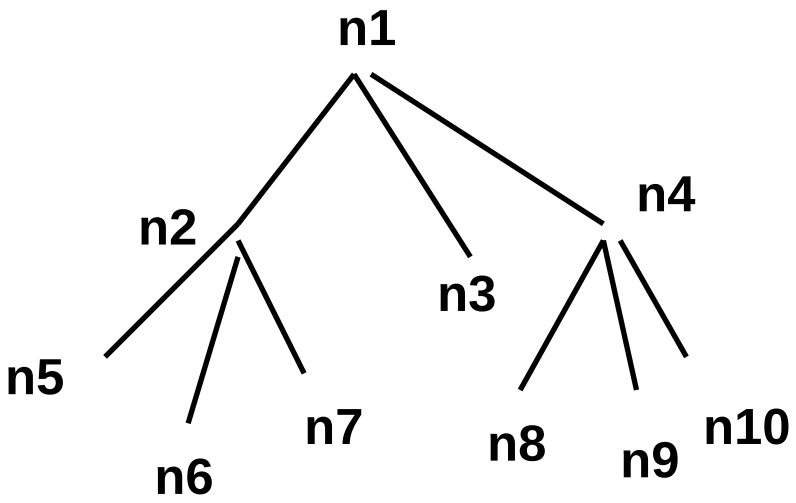
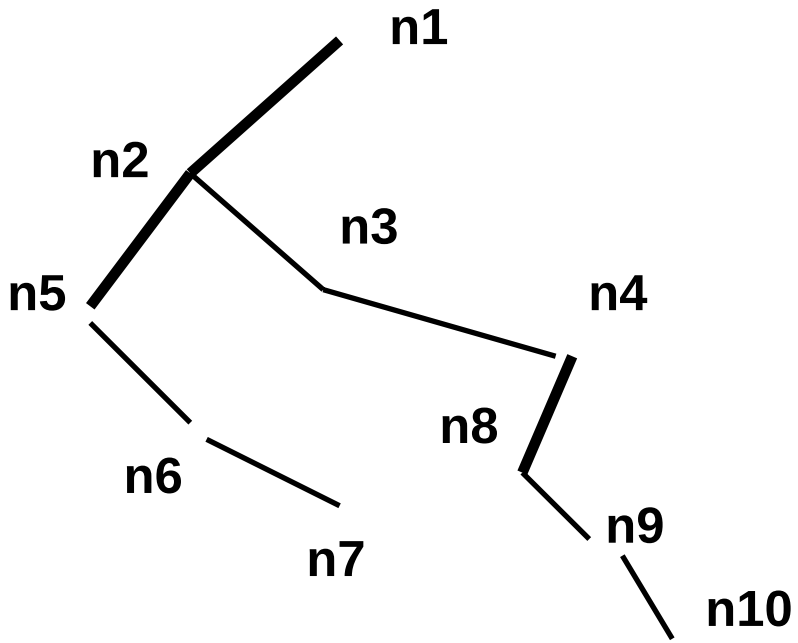


Tableau de pointeurs
- **Nécessité de connaître le nombre maximal de fils pour un nœud**
- **accès à un fils quelconque d'un nœud en une opération**

Alternative :
représenter les fils à l'aide d'une liste chaînée

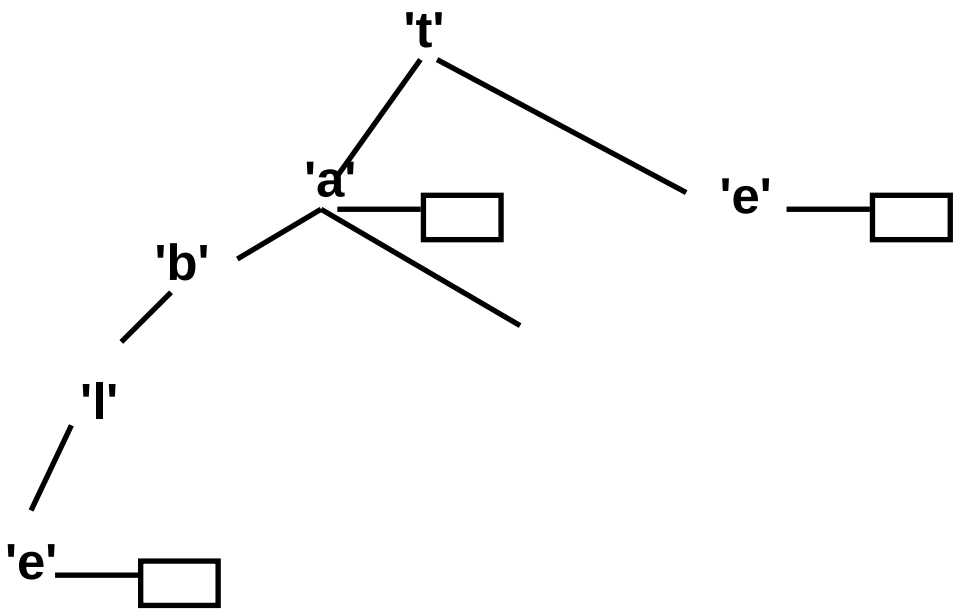
Arbres (3) : exemple d'arbre binaire





Représentation avec un arbre binaire

Arbres (4) : application (dictionnaire "trie")

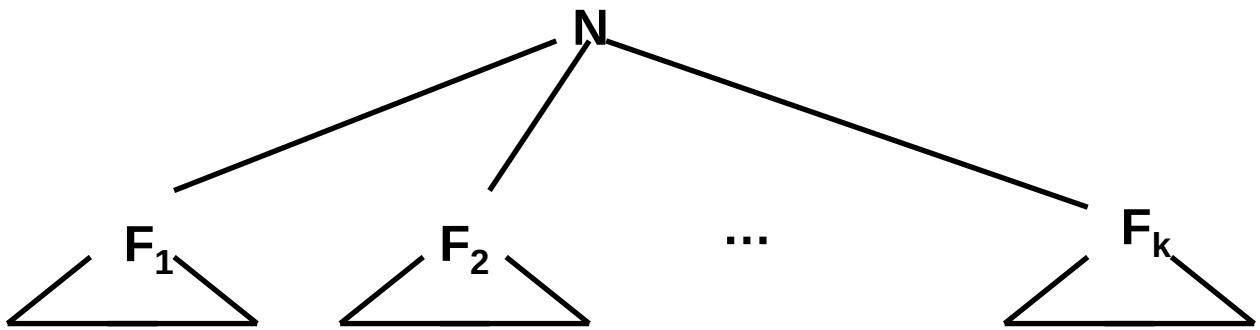



```
struct TMOT {
  char lettre;
  char *definition;
  struct TMOT *filsaine;
  struct TMOT *freredroit;
}
```

Exercice : représenter les mots
- **de**
- **debout**
- **dormir**
- **depart**

Arbres (5) : exercice sur la récursivité


```
P(TNOEUD N)
{
  action A0
  P(F1)
  Action A1
  P(F2)
  …
  P(Fk )
  Action Ak
}
```



Exercice :
1) afficher l'expression décrite par un arbre en notation préfixée
2) idem pour infixée et postfixée
3) insertion d'un mot dans un dictionnaire "trié"

Élimination de la récursivité (1) : simple terminale


```
Procédure P(Arg)
variable Local
début
  si C alors
    D;
    P(param)
  sinon
    T;
  finsi
fin
```

```
Procédure P(Arg)
variable Local, U
début
  U ← Arg
  tantque C faire
    D;
    U ← param
  fintantque
  T;
fin
```

Élimination de la récursivité (2) : simple non terminale


```
Procédure P(Arg)
variable Local
début
  si C alors
    D;
    P(param)
    E;
  sinon
    T;
  finsi
fin
```

```
Procédure P(Arg)
variable Local, U
pile Q
début
  Q ← pilevide
  U ← Arg
  tantque C faire
    D;
    empiler(Q, L, U)
    U ← param
  fintantque
  T;
  tantque !pilevide(Q) faire
    depiler(Q, <L,U>)
    E;
  fintantque
fin
```

Élimination de la récursivité (3) : boucle


```
Procédure P(Arg)
variable Local
début
  tantque C faire
    D;
    P(param)
    E;
  fintantque
  T;
fin
```

```
Procédure P(Arg)
variable Local, U
pile Q
début
  Q ← pilevide
  U ← Arg
répéter
   tantque C faire
     D;
     empiler (Q, L, U)
     U ← param
   fintantque
   T;
   tantque !pilevide(Q) faire
     depiler (Q, <L,U>)
     E;
   fintantque
 jusqu'à pilevide(Q)
fin
```


Exercice : trouver, pour chacun des 3 cas étudiés, un exemple concret; éliminer ensuite la récursivité