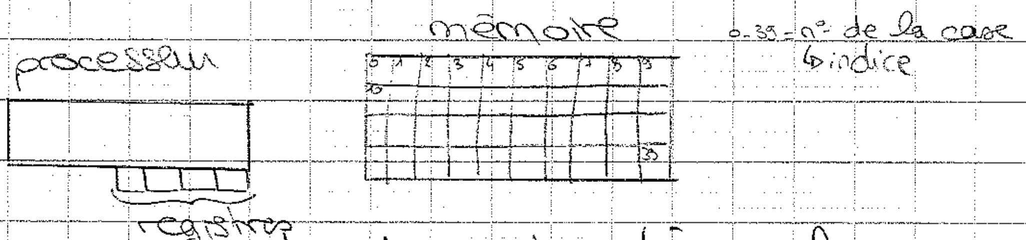
Algo
19/09/06

Alain.Griffault@labri.fr
Bureau 262

Algorithmique et structure de données

Chap 1 : Notions élémentaires

I) fctmt simplifié d'1 ordinateur

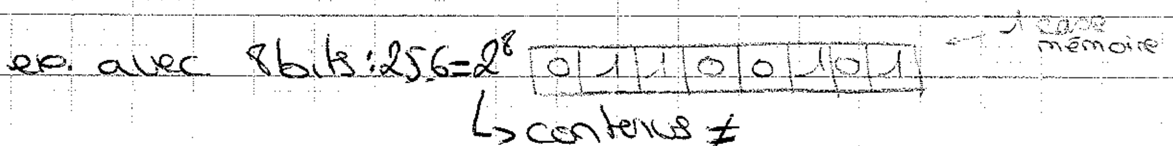


1 processeur exécute des instruct° machine
1 mémoire contient de l'informat°

ex. d'instruct° :
- additionner le contenu de 2 registres
- afficher à l'écran le contenu d'1 registre
- chger le contenu d'1 registre ...

ex. d'informat° ds la mémoire :
- chq "case" mémoire possède 1 indice
- chq case contient 1 info.
  aujourd'hui 32 bits ou 64 bits (1bit = 0 ou 1)

- ex. avec 8 bits : $256 = 2^8$



q signifie | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | ?

(64 32 16 8 4 2 1)

- si c'est l'entier : 101 = 64+32+4+1
- si c'est 1 caractère :
  ∃ des tables de caractères
  la + connue ASCII (codait 127 caractère)
  → 101ème caractère de la table
- si c'est 1 instruction
  ∃ 1 jeu d'instructions (table de correspondances : numéro = instruction)
- si c'est 1 indice d'1 case
  cela correspond à la case 101.

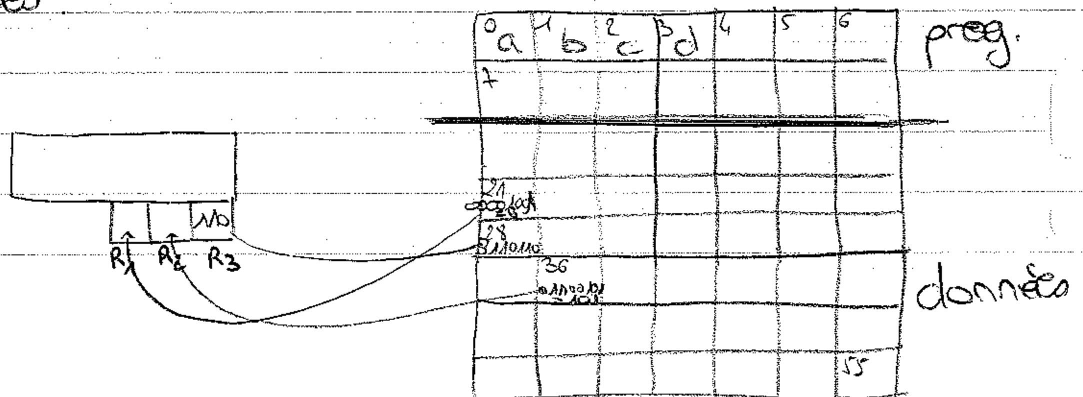
Le déroulement d'1 programme c'est exécuter séquentiellement des instructions et modifier le contenu de la mémoire.

ex:

programmer		algo
	a) chger R1 21 (charger)	"21" var X
	b) chger R2 36	"36" var Y
	c) add R1 R2 R3	X+Y
	d) sauvegarder R3 28	"28" var Z, Z=X+Y

} Z=X+Y

en g^{ral} mémoire % en 2 : programme et données



écrire 1 algo = c'est trouver 1 suite d'instruct° q modifie la mémoire pr obtenir le résultat souhaité / correct.

algo
"21" Vble X
"36" Vble Y
X+Y
X=X+Y
↳ nvelle valeur de X = ancienne valeur X + ancienne valeur Y

autres ex.
a) charger R1 21
b) charger R2 36
c) add R1 R2 R3
d) sauvegarde R3 21 ⟹ efface contenu de 21 pr le remplacer par 1 nvelle valeur

a) charger R1 (21)[(2)]
Supposons q case 21 contienne 37
dc a) ⟺ à charger R1 37

charger R1 ((21))[(3)]
21 contient 37
37 contient 43
⟺ charger R1 43

II) Variables et affectations

variable = 1 symbole x, y, z, ... q représente l'évolut° d'1 case mémoire au cours du déroulemt d'1 programme

affectation = opérat° de modificat° d'1 Vble
notations : = (langages C, JAVA)
:= (Pascal)
← ("algo") ← cours sans se soucier du langage utilisé

en fct° langages de prog notat° ≠.

ex:

X ← 3 "x reçoit 3" (= on met valeur 3 ds X)

X ← Y + Z

X ← 2 * X + 9

Notat° pour les algorithmes :

ex: 1 algo q ajoute 32 à X

nom programme → Ajouter 32 (X) { ← début du prog.
liste des paramètres

indentat° → X ← X + 32 ;
retourner X ; ← liste d'instruct° séparées par ; → séquence

fin prog → }

(retourner : retourne le résultat du prog)

Ajouter 32 (26) retourne 58

exercice :

1 algo q échange le contenu de X et Y

```
Echanger (X; Y) {
    X ← Y
    Y ← X
    retourner (X, Y)
}
```

incorrect.

```
Echanger (13; 27) {
    X ← 27
    Y ← 27
}
→ (27, 27)
```

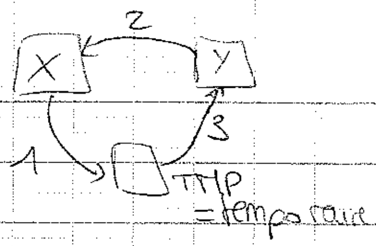
Z ← X

X ← Y

Y ← Z

Algo
19/09/06

```
Echanger (X, Y) {
    TMP <- X ;
    X <- Y ;
    Y <- TMP ;
    retourner (X, Y)
}
```



Rem :

affectation "parallèle" ← fait tjs réf aux anciennes valeurs

$(X_1, X_2, \dots, X_n) \leftarrow (\dots, \ ,\ ,)$

ex : $(X, Y) \leftarrow (3, 8) \equiv$ (équivalent) $\{ X \leftarrow 3 ; \ Y \leftarrow 8 ; \}$

$(X, Y) \leftarrow (X+7, Y+2) \equiv \{ X \leftarrow X+7 ; \ Y \leftarrow Y+2 ; \}$

$(X, Y) \leftarrow (X+Y, 2*X) \neq \{ X \leftarrow X+Y ; \ Y \leftarrow 2*X ; \}$ (ancienne valeur de X)

cela n'∃ q ds très peu de langage
∃ en Python

```
Echange_Python (X,Y) {
    (X,Y) <- (Y,X)
    retourner (X,Y)
}
```
} correcte

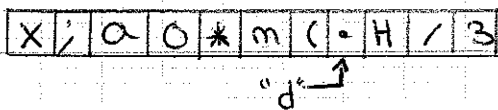
III) Tableaux et variables indicées

ex: age_1, age_2, age_3, ... n'est pas pratiq

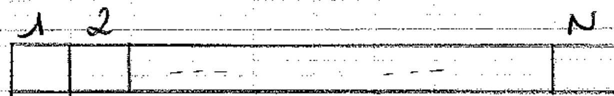
il est pratiq de pvr regrper des vbles de m̂ "nature".

tableau = struct. pr gérer 1 grd nb de vbles de m̂ type (entier, caractère, mot, ...)
on utilise 1 indice pr distinguer les ≠ vbles

ex:
un tableau T de 11 caractères



notat°: un tableau de taille N



} ≠ indices possibles

en Pascal T[13..21] <- -M ... -M+N-1

en C, en JAVA <- 0 ... N-1

indices ≠ selon les langages.

donner taille d'1 tableau ne suffit pas, il faut aussi donner cont varient les indices.

on utilisera indices entre 0 et N-1

ex: Ecrire T[3] → "O"
Ecrire T[0] → "X"
T[7] ← "d"

IV) Les structures de contrôle

1) la boucle "pour"

ex: un tableau T de taille 24 d'entiers
algo: ajouter 3 à toutes les Ddes

```
Ajouter3_version1(T) {
    T[0] ← T[0]+3 ;
    T[1] ← T[1]+3 ;
    T[2] ← T[2]+3 ;
    :
    T[23] ← T[23]+3 ;
    retourner T ;
}
```

peu pratiq

Syntaxe de la Ble "pour" :

```
pour indice de min à max faire {
    <liste d'instruction>
}
```

≡

```
indice ← min
<liste d'instruct°>
indice ← min+1
<liste d'instruct°>
:
indice ← max
<liste d'instruct°>
```

ex:

```
Ajouter_3(T) {
    pour indice de 0 à 23 faire {
        T[indice] ← T[indice] + 3 ;
    }
    retourner T ;
}
```

exercice =

un tableau d'entiers de taille N
algo pr calculer la somme des valeurs contenues ds le tableau

```
Somme_tableau(T) {
(Produit)
    TMP ← 0 ;
    (TMP ← 1)
    pour indice de 0 à N-1 faire {
        TMP ← TMP + T[indice] ;
        (TMP ← TMP * T[indice])
    } retourner TMP ;
}
```

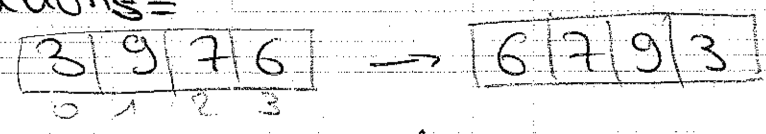
exercice =

1 tableau d'entiers de taille N
algo pr inverser les contenus

X Y ... T Z → Z T ... Y X

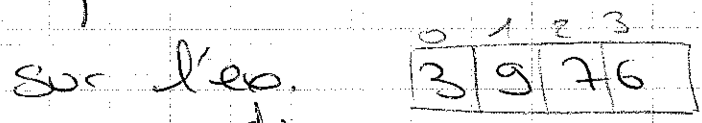
Algo
25/09/06

Solutions =



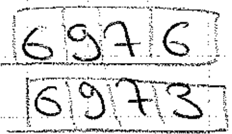

```
inverser_tableau(T) {
    pour indice de 0 à (N-1)/2 faire {
        tmp ← T[indice];
        T[indice] ← T[(N-1) - indice];
        T[(N-1) - indice] ← tmp;
    }
    retourner T;
}
```

Sur l'ex.

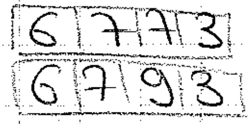


(N-1)/2 = 3/2 = 1   (N = 4)

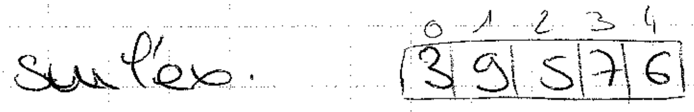
indice ← 0
tmp ← T[0]
T[0] ← T[3]
T[3] ← tmp



indice ← 1
tmp ← T[1]
T[1] ← T[2]
T[2] ← tmp

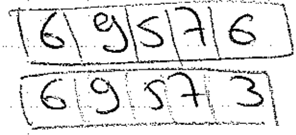


Sur l'ex.



(N-1)/2 = 2

indice ← 0
tmp ← T[0]
T[0] ← T[4]
T[4] ← tmp



indice ← 1
tmp ← T[1]   9
T[1] ← T[3] | 6 | 7 | 5 | 7 | 3 |
T[3] ← tmp | 6 | 7 | 5 | 9 | 3 |

indice ← 2
tmp ← T[2]   5
T[2] ← T[2] | 6 | 7 | 5 | 9 | 3 |
T[2] ← tmp | 6 | 7 | 5 | 9 | 3 |

⇒ algorithme juste, mais il y a du travail inutile pr 1 tableau impaire.

pour tableaux impairs:

Bonne solution

```
inverser tableau (T) {        → N/2 - 1
    pour indice 0 à (N-2)/2 faire {
        tmp ← T[indice];
        T[indice] ← T[(N-1)-indice];
        T[(N-1)-indice] ← tmp;
    } retourner T;
}
```

⇒ vérifie aussi pr tableaux pairs.

si 1 seule case $(N-2)/2 = -1$ ⇒ ne fait rien car pour i 0 à -1 = faux les indices doivent croître.

2 cases $(N-2)/2 = 0$ ⇒ pr i de 0 à 0 ⇒ ne fait q pr indice 0.

Algo
26/09/06

```
inverser_tableau(T){
    pour indice de 0 à (N-2)/2 faire{
            inverser (T[indice], T[(N-1)- indice]);
    }
    retourner T
}
```

2) les prédicats

déf = application d'1 ensemble de formules ds {vrai ; faux}

ex: X est positif est 1 prédicat
X est divisible par 2 "
X ≠ Y est 1 prédicat
X + Y > 2 "

∃ des opérateurs entre les prédicats (p)
ET, OU, NON

table de vérité

P	Non P
V	F
F	V

si P est Vrai, alors Non P est Faux

P	Q	P et Q	P ou Q
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

et => pr q qqc soit Vrai il faut q les 2 soient vrais, ds ts les autres cas c'est faux

ou => l'un des 2 est vrai

ex =

(X ≠ 2) ou (X + Y > 2) est 1 prédicat

algo = Divisible_par 2 (X) {
retourne X/2 * 2 = X // X/2 * 2 = X pair
// X/2 * 2 ≠ X impair
}

⇒ résultat vrai ou faux

3) Le branchement "si P alors ... sinon ..."

Syntaxe =
si P alors
<liste d'instruct° 1>
sinon
<liste d'instruct° 2>

déroulement
① évaluat° de P
② si P est vrai, seule liste 1 est exécutée
si P est faux, " 2 "

ex = Maximum (X, Y) {
si (X > Y) alors
max ← X ;
sinon
max ← Y ;
retourner max ;
}

Algo Exercice :

26/09/06

```
(1) Maximum_3_nombres (X, Y, Z) {
        si (X>Y) alors {
            si (X>Z) alors
                max ← X ;
            sinon
                max ← Z ;
        }
        sinon {
            si (Y>Z) alors
                max ← Y ;
            sinon
                max ← Z ;
        }
        retourner max ;
}
```

```
(2) Maximum_3_nombres (X, Y, Z) {
        si (X > Y) alors
            max ← Maximum(X, Z) ;
        sinon
            max ← Maximum (Y, Z) ;
        retourner max ;
}
```

```
③ Maximum_3_nombres (X, Y, Z) {
      max ← Maximum (X, Maximum (Y, Z));
      retourner max;
}

④ Maximum_3_nombres (X, Y, Z) {
      max ← X;
      si (Y > max) alors
            max ← Y;
      si (Z > max) alors
            max ← Z;
      retourner max;
}
```

exercice :

```
Maximum_tableau (T) {
      max ← T[0];
      pour les indices de 1 à N-1 faire {
            si (T[indice] > max) alors
                  max ← T[indice];
      }
      retourner max;
}
```

pas 0 car on considère q c'est le + grd donc pas la peine de le comparer

Algo
26/09/06

4) Boucle "tant que P faire ..."

syntaxe :
tant que P faire < liste d'instruct° >

déroulement :
① on évalue P
② si P est vrai on exécute la liste d'instruct° et on retourne en ①
si P est faux on sort de la boucle.

ex : vérifier q ts les éléments d'1 tableau st > 100.

103	1027	832	101

→ vrai

103	7	832	101

→ faux

1ère idée : V1
chercher le + petit → si + grd q 100 ⇒ vrai
→ si < 100 ⇒ faux

mais travail inutile de regarder ttes les cases dès qu'on trouve < 100 on peut s'arrêter

res = résultat

```
verifier_tableau_V1 (T){
    min ← T[0],
    pour indice de 1 à N-1 faire {
        si T[indice] < min alors
            min ← T[indice]
    }
    si(min < 100) alors
        res ← Faux;
    sinon res ← vrai
    retourner res;
}
```

0	1	2
121	142	83

N = 3

P	indice	résultat
V	0	
(121>100) V	1	
(142>100) V	2	
(83>100) F	3	
		F

N = 2

121	140

P	indice	résultat
V	0	
(121>100) V	1	
(140>100) V	2	
(T[3]>100) F		

```
vérifier_tableau(T)
    P ← vrai
    indice ← 0

    tant que P faire {   (*)
        P ← (T[indice] > 100);
        indice ← indice + 1;
    }
    retourner P
```

→ ne s'arrête pas ds le tableau

(*) correction
tant que P et (indice < N) faire {

exercice =
trouver si 1 nb X est dans 1 tableau

si x ds tableau → vrai
si x ∅ tableau → faux

```
Trouver_tableau (T, X) {
    P ← vrai  faux
    indice ← 0
    tant que non P et (indice < N) faire {
        P ← (T[indice] ≠ X);
        indice ← indice + 1;
    }
    retourner (P) (*)
}
```

= autre version

→ donc faux

mais on veut avoir vrai si x est ds tableau
dc retourner non P

(*) correction
retourner (non P)
}

Algo
03/10/06

5) Boucle "répéter ... jusqu'à"

syntaxe = repeter <liste d'instruct°> jusqu'à P

déroulemt :
1. on exécute la liste d'instruct°
2. on évalue le predicat P
3. Si P est faux on revient en 1
4. si P est vrai on passe à l'instruct° suite (on sort {de la boucle / du répéter)

ex : trouver si l'élément X est ds 1 tableau
on veut : retourne "vrai" ssi X est ds tableau T

```
Trouver_Tableau_V2 (T,X) {
    Si (N>0) alors {
    I <- 0
    repeter
        P <- (T[I] = X);
        I <- (I+1);
    jusqu'à P | (I=N);
    retourner P
    } sinon retourner faux
}
```

| = ou

3	10	17	2

trouver 17 ?

I	P	P|(I=N)	résultat
0			
	F	→ T[0] = 3 ≠ 17	
1			
		F	
	F	→ T[1] = 10 ≠ 17	
2			
		F	
	V	→ T[2] = 17 = 17	
3		V → sort de la boucle	V

3	10	15

trouver 17 ?

I	P	P / I = N	résultat (retourne P)
0			
	F		
1			
		F	
	F		
2			
		F	
	F		
3			
		V	
		↳ I = N	F

Rem = Liens entre "Tant que" et "répéter jusqu'à" :
2 propriétés =

Tant que P faire S ≡ si P alors
répéter S jusqu'à non P
sinon
"rien"

(S : suite d'instruct° ; ≡ : équivalent)
(évalue P, si P vrai on fait S, sinon sort de boucle)
(si P vrai on fait S, qd P devient faux on arrête de f^R S.)

Repeter S jusqu'à P ≡ S;
tant que non P faire S

$\frac{1}{=}$ rol : si obligat° f^R S au ⊖ 1 fois on utilise "repeter"
si on n'utilise pas S on fait "tant que"

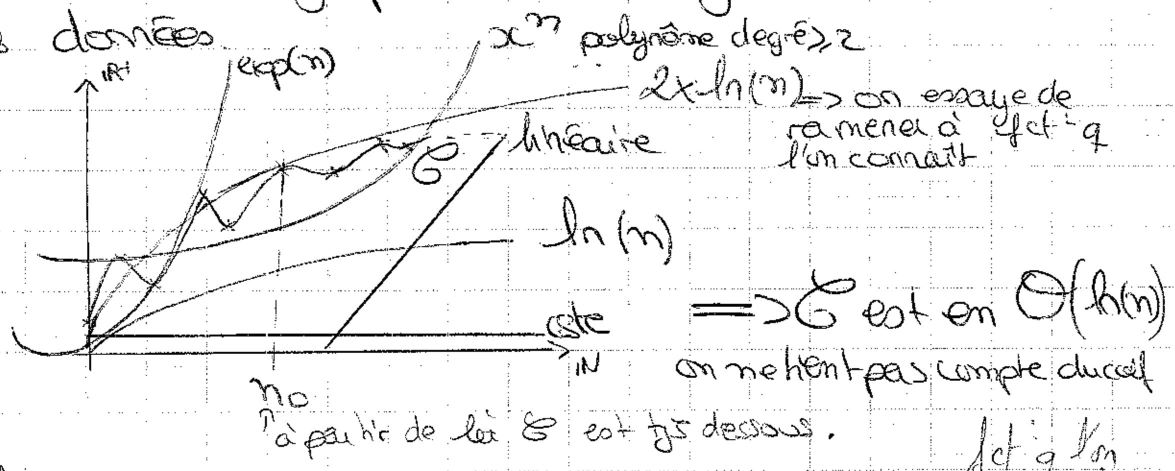
Algo
03/10/08

V) Complexité

Notat° O, Ω, Θ

fct° $\mathbb{N} \to \mathbb{R}^+$

<u>objectif</u> = avr estimat° assez précise de ce qu'va coûter 1 algo pr des très gdes valeurs des données



$\Rightarrow G$ est en $O(h(n))$
on ne tient pas compte du coef

O = complexité maximale

<u>Déf</u> = (1)

fct° T à laq° on s'intéresse — fct° q l'on connaît bien

on dit que $T(n)$ est en $O(f(n))$ ssi $\exists$ 2 cstes c et n_0 t.q qqs $n \geq n_0$

$T(n) \leq c.f(n)$

<u>ex</u> =

$$T(n) = (n+1)^2 = n^2+2n+1$$

$n=0$	1
$n=1$	4
$n=2$	9
$n3$	16

dc $T(n) = O(n^2)$

il suffit de prendre $c = 2$ et $n_0 = 3$

$O_1 = 1^2 \times 2 = 2$

$O_2 = 2^2 \times 2 = 8$

$O_3 = 3^2 \times 2 = 18 > 16$

$$T(n) = 3n^3 + 2n^2$$

$3n^3 + 2n^2$ tjs $\leq 5n^3$ dc $O(n^3)$

il suffit de prendre $c = 5$ et $n_0 = 1$

$T(n) = 3^n$ en $\Theta(2^n)$?

Supposons $\Theta(2^n) \Rightarrow \exists\ c, n_0\ \forall n \geq n_0\ 3^n \leq c.2^n$
$\Rightarrow \left(\frac{3}{2}\right)^n \leq c$ **FAUX**

Rem =

$T[N]$ \ N	1	10	1000	10^6
$\Theta(1)$	1sec	1s	1s	1s
$\Theta(\log_{10}(n))$	1s	2s	4s	7s
$\Theta(n)$	1s	10s	1000s ≃20min	10^6s ⇔ ≃ 10j.
$\Theta(n^2)$	1s	100s	10^6 s	10^{12} s
$\Theta(10^n)$	1s	10^{10} s	10^{1000} s	10^{10^6} s

→ ∀ taille N algo met tjs m̂ tps

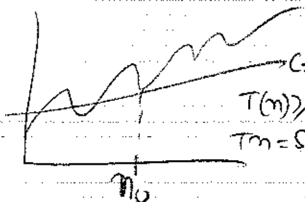
réaliste

non réaliste

1j = 10^5 s

log⊗ qu'on multiplie par 10 on ajoute 1

Def = ②
$T(n)$ est en $\Omega(g(n))$ ssi $\exists$ 2 cstes c et n_0 t.q $\forall n \geq n_0$ $T(n) \geq c \cdot g(n)$



Ω sert connaître le tps min. q prendra l'algo.

Algo
03/10/06

ex =

$T(n) = 3n^3 + 2n^2 \geq 2n^2 \rightarrow$ ~~$T(n) = \Omega(n^2)$~~

$\geq 3n^3 \rightarrow T(n) = \Omega(n^3)$

↑ + proche

Propriété =

$g(n) = O\, f(n) \overset{ssi}{\Longleftrightarrow} f(n) = \Omega\,(g(n))$

Rem =

$T(n) = O\, f(n)$ si $f(n) \neq g(n)$ il y a de
$T(n) = \Omega\, g(n)$ l'imprécision

ex $O(n^2)$ → $N = 1$ temps = 1s
$\Omega(n)$ → $N = 1000$ $1000s < tps < 10^6$

{est en =

Déf = ③

$T(n)$ est en $\Theta(f(n))$ ssi $T(n) = O\, f(n)$
et $T(n) = \Omega\, f(n)$

ex : $T(n) = 3n^3 + 2n^2$
$T(n) = \Theta(n^3)$

1) Complexité des algorithme

schématiquement 3 types d'instruct° =
- lecture d'1 Dble
- écriture d'1 Dble (= affectat°)
- évaluat° d'1 prédicat

complexité en temps d'1 algo ≈
nb lecture × tps d'1 lecture
+ nb écriture × tps d'1 écriture
+ nb test × tps d'évaluat° d'1 prédicat

<u>En pratique</u> =
tps d'1 lecture
tps 1 test << tps d'1 écriture

<u>Sut</u> =
complexité d'1 algo ≃ nb écriture × tps d'1 écriture

tps d'1 écriture = cste, dc ⇒ complexité d'1 algo ≃ nb écritures

<u>Ex</u> =

```
Echange (X, Y) {
    TMP ← X ;
    X ← Y ;
    Y ← TMP ;
}
```

3 écritures :
au max $\Theta(3) \equiv \Theta(1)$
au min $\Omega(3) \equiv \Omega(1)$
dc $\Theta(1)$

→ coût de cet algo est cst.

```
Somme_tableau(T) {
    S ← 0 ;
    pour indice 0 à N-1 faire {
        S ← S + T[indice] ;
    }
    retourner S ;
}
```

1+N écritures
$\Theta(N+1) \equiv \Theta(N)$
$\Omega(N+1) \equiv \Omega(N)$
dc $\Theta(N)$

Algo
03/10/06

1 boucle "pour" a 1 complexité linéaire

```
Trouve_tableau(T,x){
    I <- 0
    P <- vrai
    tant que P & I<N faire{
        P <- (P[I] ≠ x)
        I <- I+1
    }
    retourner NP
}
```

max : $2 + 2 \times N$
$\Theta(2N+2) \equiv \Theta(N)$

min : $2+2$ (si x est le 1er élément du tableau)
$\Omega(4) \equiv \Omega(1)$

dc il n'y a pas de Θ

1 matrice carrée T[N][N]

```
Somme_matrice_carrée(T){
    S <- 0;
    pour i de 0 à N-1 faire      // i indice de ligne
        pr j de 0 à N-1 faire    // j indice colonne
            S <- T[i][j] + S;
    retourner S
}
```

$1 + N*N$ = écritures $\Theta(N^2)$
$\Omega(N^2) \Rightarrow \Theta(N^2)$

Cas général :

```
Pour i1 de 0 à N1-1 faire{
  S1<-
  pour i2 de 0 à N2-1 faire{        tant que P & i2<N2
    S2<-
    pour i3 de 0 à N3-1 faire{
      S3<-
      :
      pour ik de 0 à Nk-1 faire{    tant que Pk & ik<Nk
retourner S1,S2,...Sk
```

Complexité =

écritures = S_1 N_1 écritures
+ S_2 $N_1 \times N_2$ "
+ S_3 $N_1 \times N_2 \times N_3$ "
⋮
+ S_k $N_1 \times N_2 \times N_3 \times \dots \times N_k$ "

$\Rightarrow N_1 \ll N_1 \times N_2 \ll N_1 \times N_2 \times N_3 \ll \dots \ll N_1 \times N_2 \times N_3 \dots N_k$

$\Theta(N_1 \times N_2 \times N_3 \times \dots \times N_k)$
$\Omega(N_1 \times N_2 \times N_3 \times \dots \times N_k)$ → la m̄ q Θ car ce n'est q des bcles "pour"

⇓

$\Theta(N_1 \times N_2 \times N_3 \times \dots \times N_k)$
N^k

si à la place du "pour" on a "tant que" (—)
max : $\Theta(N_1 \times N_2 \times \dots \times N_k)$
min : $\Omega(N_1 \times N_3 \times N_{k-1})$
↗ on enlève les N_i p les "tant que"

⇓

Il n'y a pas de Θ

Algo 18/10/05

f : Si P alors <instruct° 1> (S1)
sinon <instruct° 2> (S2)

supposons q l'on connait la complexité de S_1 et S_2.

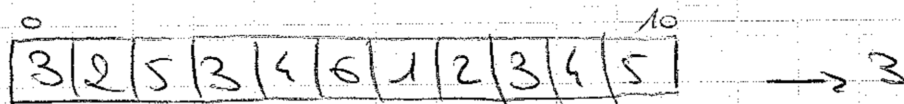
$$O(f) = \max(O(S1), O(S_2))$$

$$\Omega(f) = \min(\Omega(S1), \Omega(S2))$$

algo : suite d'instruct°
branchement
bcle pour
itérat° "tantque" ou "répéter jusqu'à"

→ on sait calculer O(algo) et Ω(algo)

ex : élément le + fréquent ds 1 tableau



élément_plus_fréquent (T) {

dernière →

version 1

```
Element_plus_frequent (T) {

    Si (N>0) alors {
cst          [  max ← 0;
                pour i de 0 à N-1 faire {
2boucles        cpt ← 0 / car on sait qu'il y en a au - 1 (celui de la case i) => le début du tableau a déjà été compté on a trouvé
imbriquées         pour j de i+1 à N-1 faire {
=> N^2                si (T[j] = T[i]) alors {
                          cpt ← cpt + 1;
                }
                si (cpt > max) alors {
négligeable        max ← cpt;
                   indice_max ← i;
                }
            }
cst          [  retourner T[indice_max]
    }
    Sinon retourner "erreur, le tableau est vide"
}
```

complexité :

si $N = 0$ $\Theta(1)$
si $N \neq$ $\Theta(N^2)$ => quadratique

modif
nb de fois q l'on passe ds la bcle

$i=0$ $N-1$
$i=1$ $+ N-2$
$i=2$ $+ N-3$
.
.
$i=N-2$ $+ 1$
$i=N-1$ 0 → $\sum_{i=1}^{n+1} = \frac{N(N+1)}{2} = \frac{1}{2}N^2 - \frac{1}{2}N$ => $\Theta(N^2)$

Algo 10/10/06 — → amélioration pr petites valeurs de N.

Supposons q ∀ $0 \leq T[i] \leq M$
on stocke les fréquences ds 1 tableau auxilliaire

version 2

```
Element_plus_fréquent (T) {
  si (N>0) alors {
  Aux [M+1] : tableau d'entiers // Aux[3] compte le nb de 3 ds T
M [ pour i de 0 à M faire {
      Aux[i] <- 0;
    }
N [ pour i de 0 à N-1 faire {
      Aux [T[i]] <- Aux [T[i]] + 1 ;
    }
    max <- 0
M [ pour i de 0 à M faire {
      si (Aux[i] > max) alors {
        max <- Aux[i]
        indice_max <- i ;
      }
    }
    retourner indice_max
  }
  sinon retourner "erreur..."
}
```

Ex :

	0										10	11	
T	3	2	5	3	4	6	1	2	3	4	5	3	=> 4

	0	1	2	3	4	5	6
Aux	0	0	0	0	0	0	0
i=0				1			
i=1			1				
i=2						1	
i=3				2			
⋮							
fin 1ère pour		1	2	4	2	2	1

	max	indice_max
	0	
i=1	1	1
i=2	2	2
i=3	4	3

[illegible] ←

complexité :

si $N = 0$ $\Theta(1)$
si $N \neq 0$ $M + N + M$
$\Theta(M+N)$

si $N >> M$ $\Theta(N)$
si $N \simeq M$ $\Theta(N)$ $(= \Theta(M))$
si $N \ll M$ $\Theta(M)$

$\Rightarrow$ linéaire ds ts les cas.

mais l'espace mémoire nécessaire et ⊕ grd
(on crée l'autre tableau)

ex : $N = 10^2$
$M = 4$ $\Rightarrow$ il faut utiliser version 2

$N = 10$
$M = 2^{16}$ // tableau d'entiers $\Rightarrow$ il faut utiliser version 1
$\simeq 65000$

$v1 \Rightarrow \Theta(N^2) \simeq 100$
$v2 \Rightarrow \Theta(M) \simeq 65000$

Algo
10/10/08

Chap. 2 : Algorithmes de recherche et de tri

I) Le tri par sélection

Pb = on veut trier par ordre croissant les entiers d'1 tableau.

3	18	10	25	9	4	11	13

Idée = trouver le ⊕ petit pr l'échanger avec le premier
trouver le 2e ⊕ petit pr l'échanger avec le 2nd

...

Étape								
①	3	18	10	25	9	4	11	13
②	3	4	10	25	9	18	11	13
③	3	4	9	25	10	18	11	13
④	3	4	9	10	25	18	11	13
⑤	3	4	9	10	11	18	25	13
⑥	3	4	9	10	11	18	25	18
⑦	3	4	9	10	11	13	18	25
⑧	3	4	9	10	11	13	18	25

Tri_selection (T) {

pour i de 0 à N-1 faire { // on place le ième + petit ds la ième place

indice min ← i ;

pour j de i+1 à N-1 faire {

si (T[j] < T[ind min]) alors

ind min ← j ;

}

si (i ≠ ind min) alors {

→ TMP ← T[i] ;

→ T[i] ← T[ind min] ;

→ T[ind min] ← TMP ;

}

}

}

complexité :

nb de fois q l'on passe ds la boucle pr

i = 0	N-1
i = 1	N-2
⋮	⋮
i = N-1	0

$$\sum_{j=0}^{N-1} j = \frac{N(N-1)}{2}$$

$\Rightarrow \Theta(N^2)$

si T déjà trié

T[j] < T[ind min] ⇒ tjs vrai

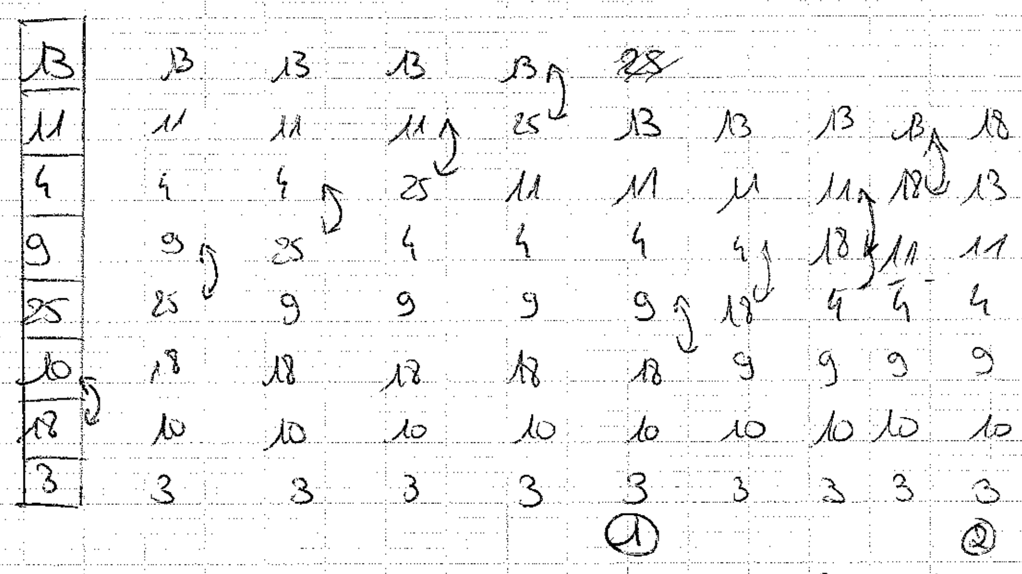
i ≠ ind min ⇒ tjs faux.

↳ 0 affectat°, on compte les comparaison

Algo
10/10/06

II) Le tri à bulle

Pb = on veut trier par ordre croissant les élém d'1 tableau



Idée = on permute # couple de cases successives mal ordonnées.

on remonte les ⊕ grds vers le haut

1er passage → le ⊕ grd est bien positionné
2e passage → les 2 ⊕ grds st " "

```
Tri_bulle (T) {
                     N-1 à 0                          // i' = N-1-i
   pour i de 0 à N-1 faire {
      pour j de 1 à N-1-i faire {
         si (T[j-1] > T[j]) alors {
            TMP ← T[j-1] ;
            T[j-1] ← T[j] ;
            T[j] ← TMP ;
         }
      }
}
```

Complexité :

si déjà trié on ne fait rien

on compte le nb de comparaisons

$i=0$	$N-1$
$i=1$	$N-2$
⋮	
$i=N-1$	0

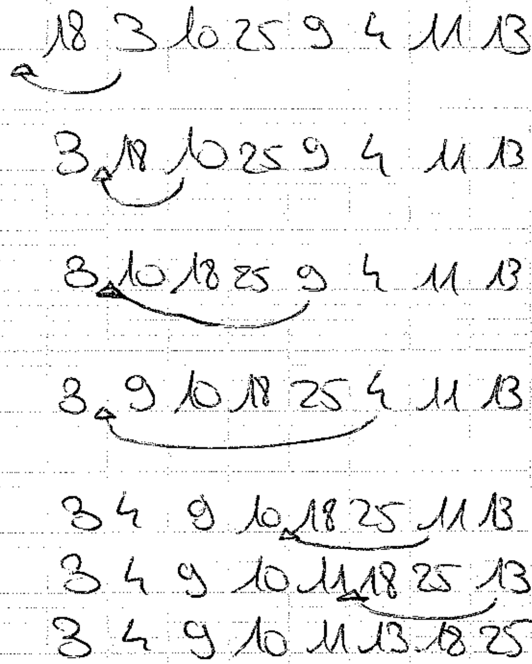
$$\Sigma = \Theta(N^2)$$

III) Le tri par insertion (le tri des joueurs de cartes)

18	3	10	25	9	4	11	13

Idée : on parcourt de gauche à droite en insérant à la bonne place 1 carte mal positionnée

Algo
10/10/06



11/10/06

```
Tri_insertion (T) {
    pour i de 1 à N-1 {
        TMP ← T[i];
        j ← i-1;
        tant que (TMP < T[j]) & j≥0 faire {
            T[j+1] ← T[j]
            j ← j-1;
        }
        T[j+1] ← TMP
    }
}
```

(pour ne pas sortir du tableau)

complexité = (nb test (TMP < T[j]))

la moitié > et l'autre inf à la case

	Min	Max		moy
i=1	1	1		1/2
i=2	1	2	$\Omega(N)$	2/2
i=3	1	3	$\Longrightarrow \Theta(N^2)$	3/2
⋮				
i=N-1	1	N-1		(N-1)/2
	N-1	$\sum_{i=1}^{N-1} = \frac{N(N-1)}{2}$		$\sum_{i=1}^{N-1} i/2 = \frac{N(N-1)}{4}$
	$\Omega(N)$	$\Theta(N^2)$		$\Theta(N^2)$

$\Omega(N)$ complexité linéaire → le tableau est déjà trié.

$\Theta(N^2)$ le tableau est trié à l'envers.

en moy $\Theta(N^2)$ → complexité quadratiq mais va 2 fois ⊕ vite

IV) La recherche séquentielle

<u>question</u> = X est-il ds le tableau T ?
si oui, sa place

```
Recherche_sequentielle (T,X){
    indice ← 0;
    trouve ← faux;
    tant que (indice < N) & (non trouvé) faire{
        trouve ← (T[indice] = X);
        indice = indice + 1;
    }

    si(trouve) alors
        retourner indice-1;
    sinon
        retourner "pas trouvé"
}
```

Algo
17/10/06

V) La recherche dichotomique

Hyp= le tableau est trié

question = x est-il ds T ?
si oui, sa posit°

idée = chq test permet de ↘ de moitié l'espace de recherche en comparant x avec l'élément "au milieu" du tableau

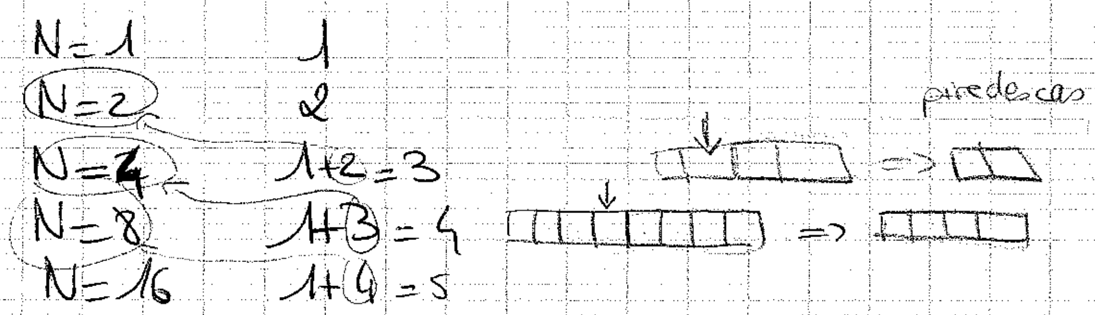
```
Recherche_dicothomique (T,x){
    gauche ← 0
    droite ← N-1;  //limites de l'espace de recherche
    trouve ← faux;
    tant que (gauche ≤ droite) & (non trouve) faire {
        milieu ← (gauche + droite)/2;
        trouve ← (T[milieu] = x);
        si (non trouve) alors {
            si (T[milieu] < x) alors :
                gauche ← milieu +1
            sinon :
                droite ← milieu -1
        }
    }
    si trouve alors
        retourner milieu
    sinon
        retourner "pas trouve"
```

Rem = Si on fait
gauche ← milieu
droite ← milieu
alors on a des "boucles infinies" ⇒ on ne sort jamais du tant que.

Complexité =

Min = 1 (x est au milieu du tableau)

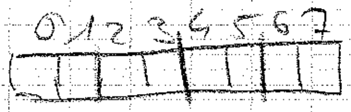
Max =

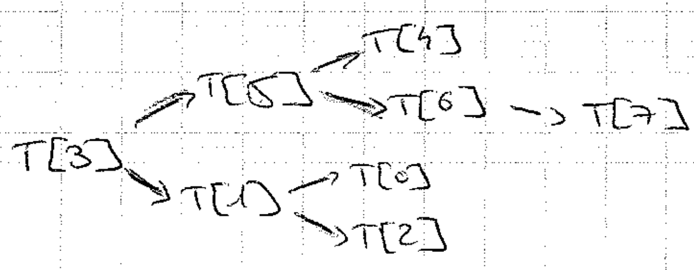


Rq fois q l'on dble N on augmente de 1 ⇒ complexité logarithmiq.
1 = le nb de passages ds le tant q

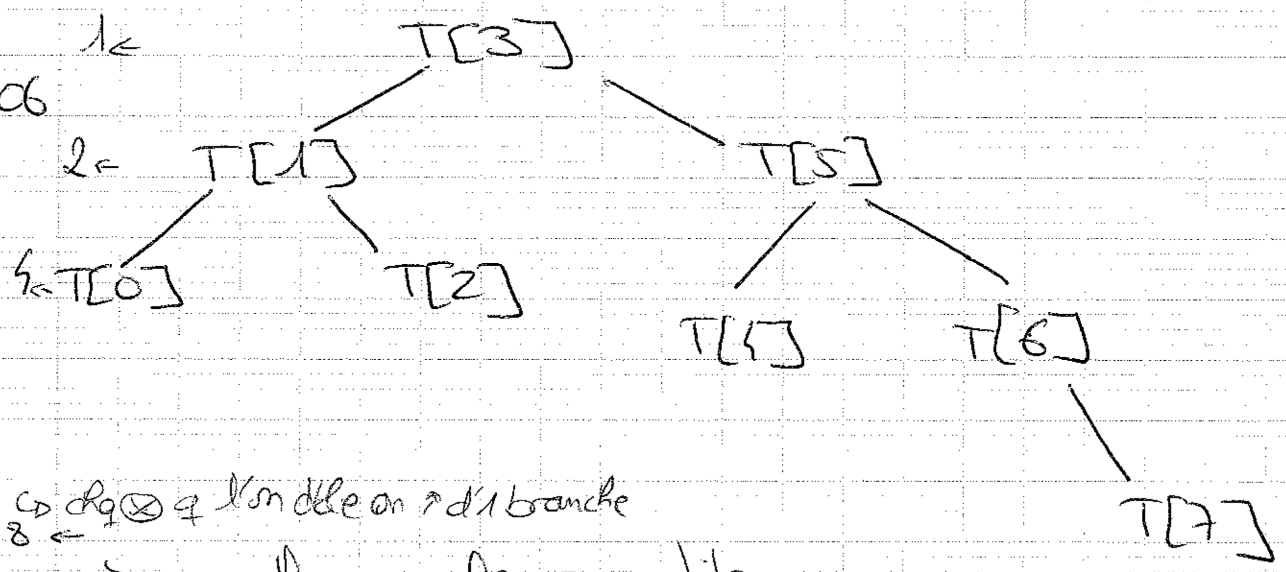
$\Theta(\log_2 N)$

Rem = autre représentat° d'1 tableau trié





Algo
17/10/06



⇔ chq ⊗ q l'on dble on ↗ d'1 branche
8 ←

à gauche → les ⊕ petits
à droite → les ⊕ gds

branche la ⊕ grande donne le nb ⊗ max q l'on peut passer ds la boucle tant que (ici = 4 ⊗)

VI] Recursivité

Déf = 1 algo est récursif s'il s'appelle lui-m̂

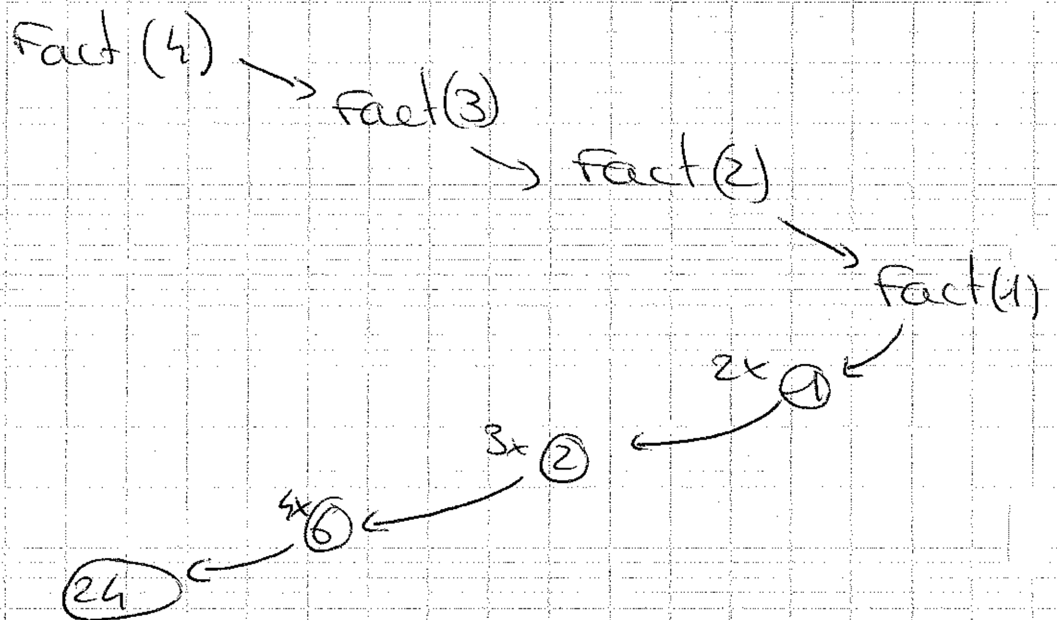
ex =

factorielle(n) : $n! = n \times (n-1)!$
$1! = 1$

$4! = 4 \times 3!$
$= 4 \times 3 \times 2!$
$= 4 \times 3 \times 2 \times 1!$
$= 4 \times 3 \times 2 \times 1$

```
Factorielle_recursive(n) {
    si (n = 1) alors
        retourner 1
    sinon
        retourner n x Factorielle_recursive (n-1)
}
```

Mécanisme de pile d'appels récursifs



Complexité =

N = 1	1
N = 2	1 + 1
N = 3	1 + 2
⋮	

(1 + 2 → multiplicat° par n + le tps pr faire Fact(2))

$t(n) = 1 + t(n-1)$ ⟹ complexité linéaire $\Theta(N)$

Algo
17/10/06

~~prop~~ $n! = \prod_{i=1}^{n} i$

__Rem__ =

```
factorielle_iteratif (n) {
    fact ← 1
    pour i de 1 à N faire
        fact ← fact x i
    retourner fact
}
```

⟹ complexité linéaire $\Theta(N)$
(boucle pour)

⊕ efficace q récursif.

VII) Tri par dénombrement

__idée__ = on compte le nombre d'occurences d'1 elt élément, puis on duplique le tableau ds le tableau résultat.

__ex__:

A	1	2	1	4	2	1	1

nb 1 → 4
nb 2 → 2
nb 4 → 1

si on sait que $\forall i\ 0 \leq T[i] < k$

	0	1	2	3	4
tableau C	0	4	2	0	1
C'	0	4	6	6	7

ts les 0 st avt case 0
les 1 st avt case 4
ts les 2 st avt case 6
ts les 3 st avt case 6
ts les 4 st avt case 7

B | 1 | 1 | 1 | 1 | 2 | 2 | 4 |

C' → B | | | | 1 | | |

on lit de droite à gauche
1 → on va ds C' ⇒ dit de mettre le 1
ds la case 4 puis C[1] = 3

0	1	2	3	4	5	6
		1	1			1

1 →
C'[1] = 2

2 → C' → ds case 6
puis C'[2] = 5

24/10/06

tri_par_denombrement (A) {
// A [0, N-1]
// C [0, K-1]   0 ≤ A[i] < K
// B [0, N-1]   le tableau trié

K [pour i de 0 à K-1 faire {
C[I] ← 0
N [pour i de 0 à N-1 faire {
C[A[i]] ← C[A[i]] + 1
K-1 [pour i de 1 à K-1 faire {
C[i] ← C[i] + C[i-1]
2N [pour i de 0 à N-1 faire {

Algo
24/10/06

donne l'indice

$B[C[A[i]]-1] \leftarrow A[i]$;
$C[A[i]] \leftarrow C[A[i]-1]$

retourner B.

Complexité = $3N + 2K - 1$

de $\Theta(N+K)$

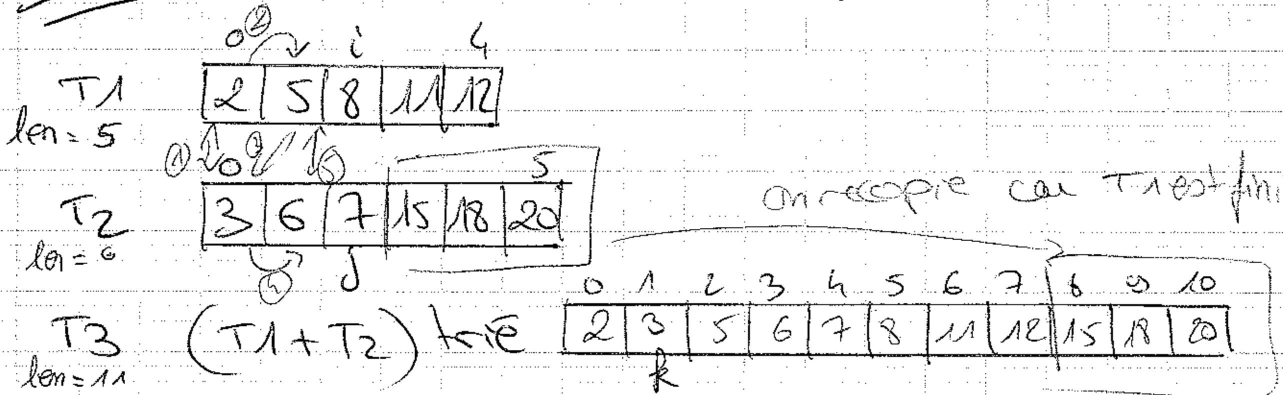
- si $N >> K$ (très gd tableau avec peu de valeurs ≠)

  $\Theta(N)$ ↳ très efficace

- si $N << K$ (pas bcp de valeurs à trier)

  $\Theta(K)$ ↳ pas efficace

VIII) Fusion de 2 tableaux triés



```
tri_fusion (T1,T2) {
  // T1 [0; N-1]
  // T2 [0; M-1]
  // T3 [0; N+M-1]

  i ← 0;
  j ← 0;
  k ← 0;

  tant que (i<N) & (j<M) faire {
     si T1[i] ≤ T2[j] alors {
          T3 [k] ← T1[i];
          i ← i+1;
          k ← k+1;   (barré)
     }
     sinon {
          T3[k] ← T2[j];
          j ← j+1;
          k ← k+1;   (barré)
     }
     k ← k+1
  }

  si (i = N) alors {
     tantq (j<M) faire {        ⇔ pr l de j à M-1 faire
          T3[k] ← T2[j];               T3[k] ← T2[j]
          j ← j+1;                     k ← k+1
          k ← k+1;
  } sinon tantq (i<M) faire {
          T3[k] ← T1[i];
          i ← i+1
          k ← k+1
     }
  retourner T3
```

début instruct° ⇔ haut boucle

au milieu condit° → on ne peut pas tjs en sortir

→ ne pas répéter ⇔ ⊕ lisible

en fin d'instruct° → fin bcle

Algo
24/10/06

ajouter cases au tableau = (très gd nb)
ajout des sentinelles

objectif = pr continuer 1 algo en connaissant à l'avance le résultat d'1 comparaison.

la ⊕ petite sentinelle est sup au max des comparaisons possibles.

```
trifusion avec sentinelle (T1, T2){
    // T1 [0 ; N-1]   + T1[N] sentinelle
    // T2 [0 ; M-1]   + T2[M] sentinelle
    // T3 [0 ; N+M-1]

    T1[N] ← T2[M-1]+1 ;

    T2[M] ← T1[N-1]+1 ;
    i ← 0 ;
    j ← 0 ;
    pour k de 0 à N+M-1 faire {
        si T1[i] ≤ T2[j] alors {
            T3[k] ← T1[i] ;
            i ← i+1 ;
        sinon {
            T3[k] ← T2[j] ;
            j ← j+1
    }
    retourner T3
}
```

Chap. 3 :

Piles et Files

I) Piles (en anglais : LIFO)

Déf : 1 struct. de données
1 Σ dynamiq ds laq[l] on peut :
- ajouter 1 élémt
- retirer le dernier élément ajouté
- savoir si la struct est vide

LIFO = Last In First Out

Primitives :

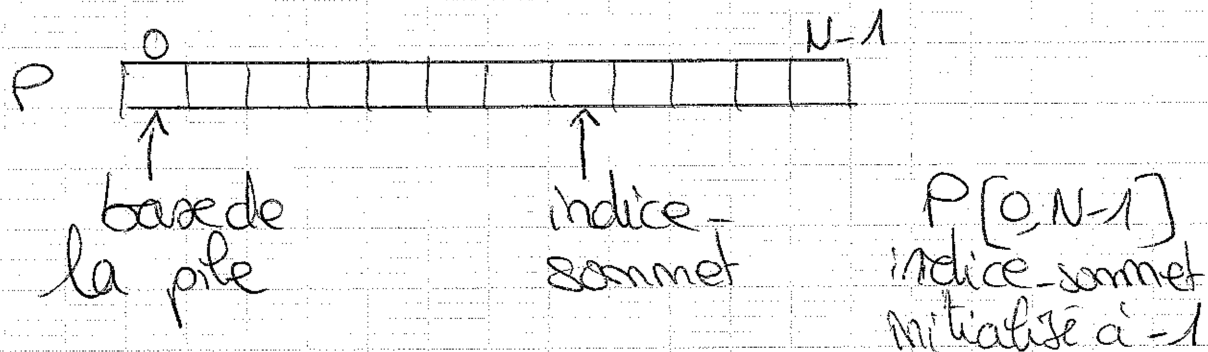
sommet_pile(P) retourne la dernière valeur ajoutée si la pile n'est pas vide

pile_vide(P) retourne vrai si la pile est vide, faux sinon

empiler (P, x) ajoute l'élémt x au sommet de la pile

Algo depiler(P) retire l'élmt situé au sommet
24/10/06 si la pile n'est pas vide

① implémentat° par 1 tableau




```
sommet_pile (P){
    si (non pile_vide(P)) alors
        retourner P[indice_sommet]
    sinon
        retourner "pile vide"
}
```

```
pile_vide(P){
    retourner (indice_sommet < 0)
}
```
↳ vrai si vide

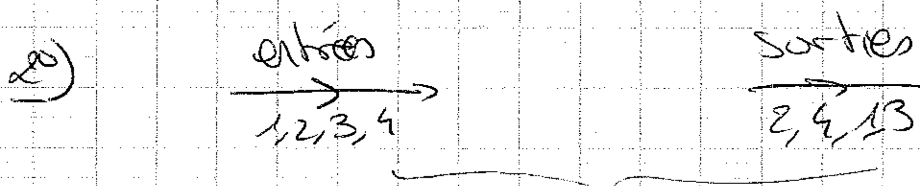
```
empiler (P,x) {
    si indice_sommet < N-1 alors {
        indice_sommet ← indice_sommet + 1
        P[indice_sommet] ← x
    sinon "Erreur, pile pleine"
```

* pr éviter de sortir du tableau

↳ on ne peut pas mettre ⊕ d'objets q la taille du tableau

```
depiler (P) {
    indice_sommet ← indice_sommet - 1
}
```

empiler (1)
empiler (2)
dépiler → 2
empiler (3)
empiler (4)
depiler → 4
depiler → 3
depiler → 1



empiler (1)
empiler (2)
dépiler (2)
empiler (3)
empiler (4)
depiler (4)
(3)
(1)

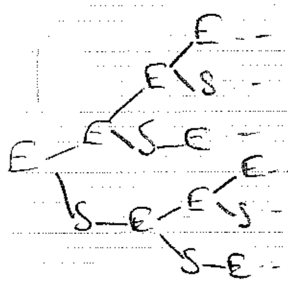
impossible

Algo
24/10/06

Question =

entrées 1, 2, 3, ..., N
quelles st les permutat° q st des sorties possibles ?

actions possibles (sans tenir des valeurs)
- E (entrée)
- S (sortie) à condit° qu'il y ait eu "assez d'entrées aut"



EESESEESSESSEESS

(( ) ( ) (( )) ( )) (( ))

(→ E
) → S

langage de Dick = expo bien parenthésée

② évaluat° des expo post-fixées

(2+3) × (4+5) { CASIO TEXAS

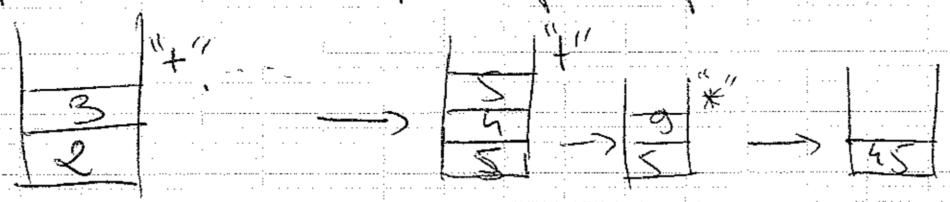
↳ 23+ 45+ * ← HP

les quiers trouve la seconde font le calcul

* + 23 + 45

pré fixée

pré fixe = notat° polonaise   post fixe = polonaise inversée



⊕ glmt =

E_1 E_2 2 expº post fixées

$P(E_1+E_2) \rightarrow P(E_1)\ P(E_2)\ +$

$P(E_1 \times E_2) \rightarrow P(E_1)\ P(E_2)\ *$

E : tableau contenant l'expº post-fixée

E

2	3	+	4	5	*

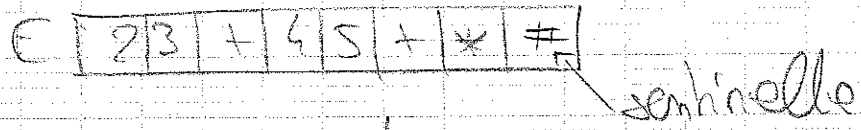
si c'est 1 nb x → empiler(x)
si c'est 1 opérateur "+", "*" →
 dépiler → A
 dépiler → B
 calculer A "op" B
 empiler le résultat
le résultat est à la base de la pile

```
Evaluation_expº_post_fixée
   P ← Init pile vide(); //E[0, N-1] contient l'expº
   i ← 0
tant q(E[i] ≠ sentinelle) faire pour i de 0 à N-1 {
      si(E[i] est 1 nb) alors {
         empiler(P, E[i]);
      sinon{      // c'est l'opérateur
         A ← sommet_pile(P);
         depiler(P);
         B ← sommet_pile(P);
         depiler(P);
```

Algo
24/10/06

C ← A "E[i]" B,
empiler(P,C),
}i ← i+1}
retourner sommet_pile(P)
}

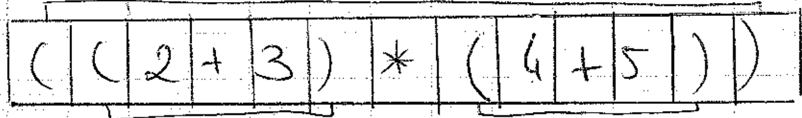
ajout sentinelle à exp°



⇒ vers° avec tant q

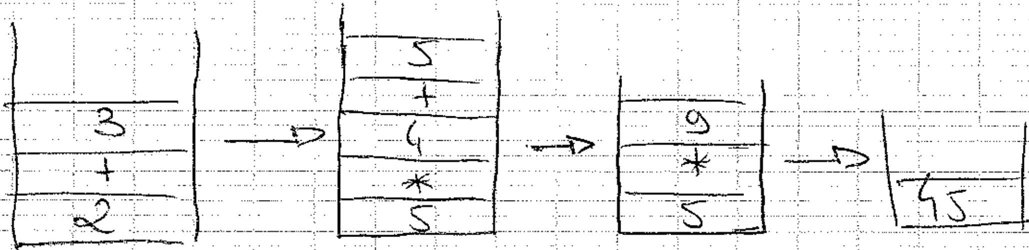
3/11/10/06 ③ évaluat° d'exp° bien parenthésées

((2+3) * (4+5))



dès qu'on tombe sur parenthèse) → on peut calculer l'exp°

idée = parenthèses ouvrantes inutiles
stocker ds la pile les nb et les opérateurs
parenthèse fermantes → effectuer l calcul




```
Evaluat_Expression_ParenthéséE(Exp) {
    P ← PileVide()
    pour i de 0 à n-1      // n = longueur de exp
        si (Exp[i] est nbou l'opérateur) alors
            empiler (P, Exp[i])
        sinon si (Exp[i] == ")") alors :
            v2 ← valeur_sommet(P),
            dépiler P
            op ← valeur_sommet(P),
            dépiler P
            v1 ← valeur_sommet (P),
            dépiler P,
            res ← v1 op v2
            empiler (P, res)
    retourner valeur_sommet
```

II) Les files

<u>Def</u> = Ensemble dynamiq ds leq^l on insère en queue et on retire en tête.

(analogie avec file d'attente)

en anglais = FIFO = First in First Out

3/11/08 Algo

Les primitives :

file_vide (F) → oui si file vide / → non sinon

enfiler (F, x)

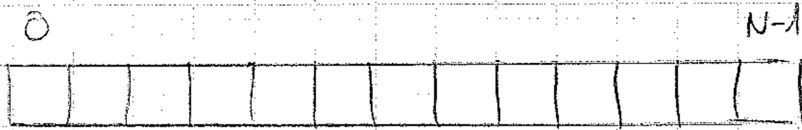
valeur_tete (F)

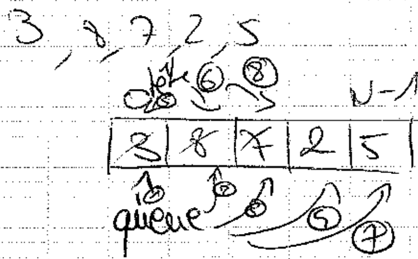
defiler (F)

file_pleine (F)

créer_fileVide (N) ← taille de la file

Implémentation par 1 tableau :

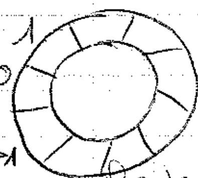




enfiler (3) ① — defiler (3) ⑤
enfiler (8) ② — defiler (8) ⑥
enfiler (7) ③ — enfiler (5) ⑦
enfiler (2) ④ — defiler (7) ⑧

⇒ on ne peut ⊕ enfiler alors q la file n'est pas pleine.
1 ⊗ arrivé à la fin du tableau on revient au début pr voir s'il est plein.

Implémentat° par 1 tableau circulaire :



file_vide → tête est queue au m̄ endroit (tête = queue)

file pleine → tête = queue ⇒ pb

queue = indice du prochain dépôt
tête = " " retrait

2 sol =

1) ajouter 2 Vbles booléennes q indiq si la file et si la file est pleine

2) laisser tjs au ⊖ 1 case vide

1)

```
créer_file Vide (N) {
    F [0, ..., N-1];
    queue ← 0;
    tête ← 0;
    vide ← vrai;
    pleine ← faux;
}

file vide (F) {
    retourner vide;
}

file pleine (F) {
    retourner pleine
}

valeur_tête (F) {
    si (non file vide (F))
        retourner F[tete]
    sinon "erreur file vide"
}

enfiler (F, x) {
    si non filepleine (F) alors :
        F[queue] ← x
        queue ← (queue + 1) % N
        (+) 
    sinon " erreur file pleine"
}
```

(+) vide ← faux
plein = (queue = = tête)

2)

```
créer_fileVide(N) {
    F[0, ..., N-1];
    queue ← 0;
    tête ← 0;
}

file vide(F) {
    retourner (tête == queue)
}

file pleine (F) {
    retourner ((queue+1)%N == tête)
}

valeur tête(F) {
    si (non file vide(F))
        retourner F[tete]
    sinon "erreur file vide"
}

enfiler (F, x) {
    si (non file pleine(F)) alors
        F[queue] ← x
        queue ← (queue+1)%N
    sinon "erreur file pleine"
```

Algo 31/10/06

```
defiler(F) {
    si (non file vide(F)) {
        tête ← (tête + 1) % N;
        plein ← faux
        vide = (tête == queue)
    sinon "erreur file vide"
}
```

```
defiler(F) {
    si (non file vide(F)) {
        tête ← (tête + 1) % N
    sinon
        "erreur, file vide"
}
```

Chap. 4 =

Les listes

I) Définition

Σ dynamiq ds laq^lle on peut ajouter et retirer un "élément" "n'importe où"

Les primitives :

liste_vide (L) → oui si la liste est vide
→ non sinon

ajouter en tête (L, x)

ajouter après (L, x, P) — P est une position ds la liste

retirer après (L, P)

retirer en tête

valeur (L, P)

position (L, x)

créer listevide ( )

Implémentation par 1 tableau ?

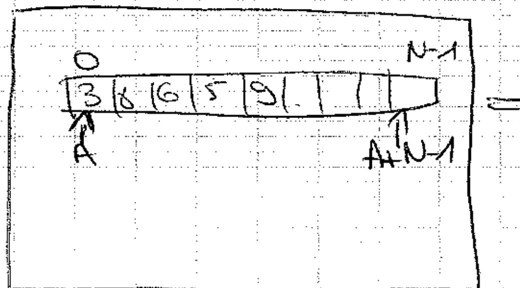
	0	1	2	3	4	5	6	7	8
L	3	8	6	5	9	2	7		

ajouter Après (L, 4, 2) // ajouter valeur 4 après position 2

0	1	2						
3	8	6	4	5	9	2	7	

décalages = trop coûteux
→ inefficace

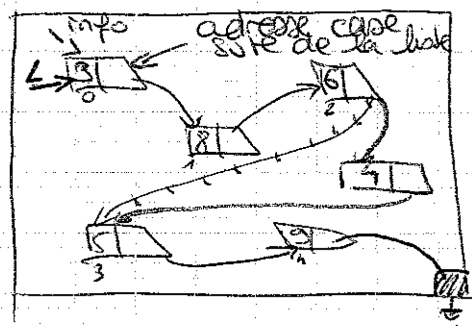
Idée d'implémentation :



⟹ pas bon

Algo
31/10/06

mémoire

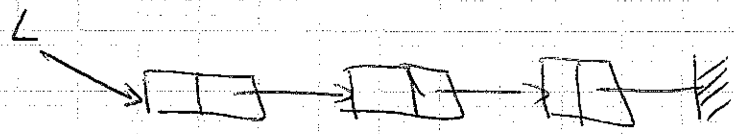


élément de la liste pas contigus en mémoire

II) les listes chainées

cellule = info
suit // adresse cellule suite ds liste ou "nil" si c'est la dernière cellule

une liste



Les primitives =

```
créer liste vide {
    L <= nil
}

liste vide (L) {
    retourne (L = nil)
}
```

```
créer cellule(x) {
    info <- x
    suivant <- nil
    retourner cellule
}

Insérer Après (L, x, P) {   //P pointe sur 1ø
    c <- créer_cellule(x);
    c.suivant <- P.suivant
    P.suivant <- c;                ↳ case suit P
}

Insérer en tête (L, x) {
    c <- créer_cellule(x)
    c.suivant <- L;
    L <- c;
}

Retirer Après (L, P) {
    si (P.suivant != nil) alors :
        P.suivant <- P.suivant.suivant
    sinon
        "erreur, P est la dernière é"
}

Retirer en tête (L) {
    si (L != nil) alors :
        L <- L.suivant
    sinon
        "erreur, liste vide"
}
```

(2P)

Algo
31/10/06

```
valeur (L, P){
    si (P != nil) alors
        retourner P.info,
}
```

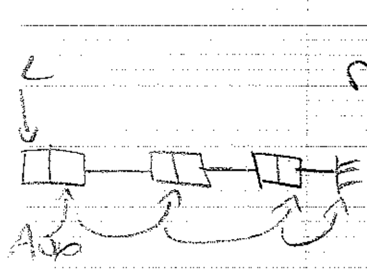
Exercice :

multiplier par 2 les nombres d'1 liste

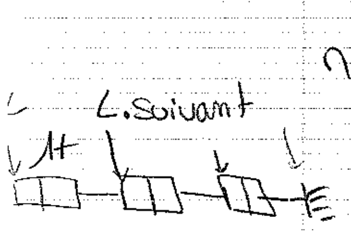
```
Traitement_Info(L){
    Aux <- L
    tant q (aux != nil) faire {
        Aux.info <- Traitement (Aux.info) //ici*2
        Aux <- Aux.suivant,
    }
}
```

```
Traitement Info recursif (L) {
    si (L != nil) alors
        L.info <- Traitement (L.info )
        Traitement Info.recursif (L.suivant),
}
```

07/11/08 II) Nombre d'éléments d'une liste

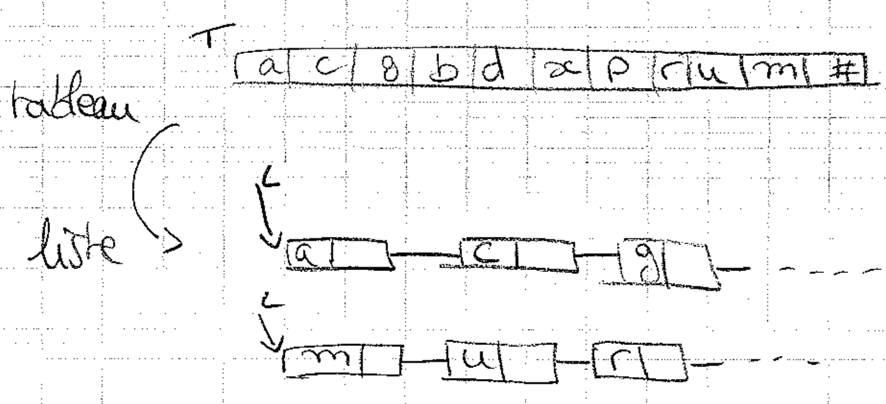



```
nb_élément_Itératif (L) {
    nb ← 0 ;
    Aux ← L ;
    tant q (Aux ≠ nil) faire {
        nb ← nb+1 ;
        Aux ← Aux.suivant ;
    }
    retourner nb ;
}
```

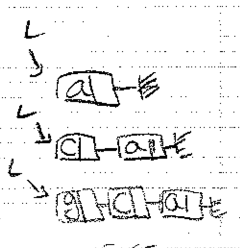



```
nb_élément_récursif (L) {
    si (L = nil) alors
        retourner 0
    sinon
        retourner 1+nb_élément_récursif(L.suivant)
}
```

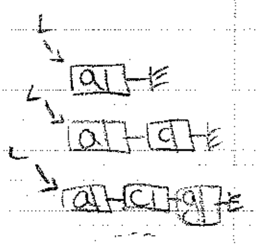
III) Créat° d'1 liste à partir d'un tableau



Algo
07/~~11~~/06

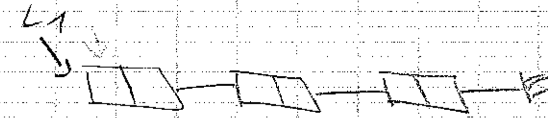


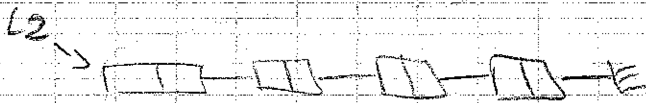
```
Créer liste_inverse(T){
    L <- nil;
    i <- 0;
    tant que T[i] ≠ '#' faire {
        insérer en tête (L, T[i]);
        i <- i+1;
    }
    retourner L;
}
```



```
Créer_liste(T){
    L <- nil;
  si (T[0] ≠ '#') alors {
    ~~i <- 0;~~
    ~~tant q (T[i] ≠ '#') faire {~~
    insérer en tête (L, T[0]);
    i <- i+1;
    P <- L;
   tant q (T[i] ≠ '#') faire {
      insérer après (L, P, T[i]);
      P <- P.suivant;
      i <- i+1;
   }
  } retourner L;
}
```

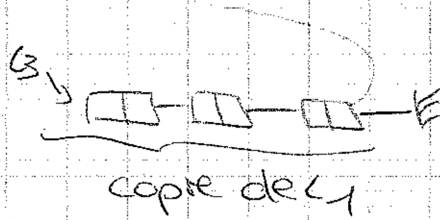
V) Concaténation de 2 listes





=> L_1 est le résultat (l'ancien L_1 n'∄ plus)

ou



=> résultat $L_3 = L_1 . L_2$

Concaténation (L_1, L_2) {
(*) Aux ← L_1
tant q (Aux.suivant ≠ nil) faire {
Aux ← Aux.suivant
}
Aux.suivant ← L_2;
retourner L_1;
} (⊙)

Si L_1 ≠ nil alors
Aux ← L_1
}
sinon retourner L_2
}

(*) Si L_1 ≠ nil alors {

}

(⊙) sinon
retourner L_2;

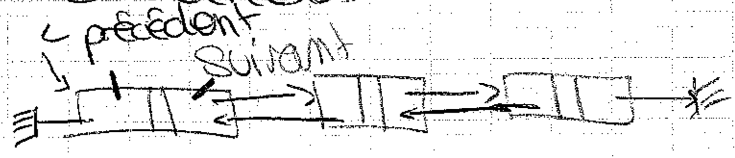
}

Algo
07/11/06

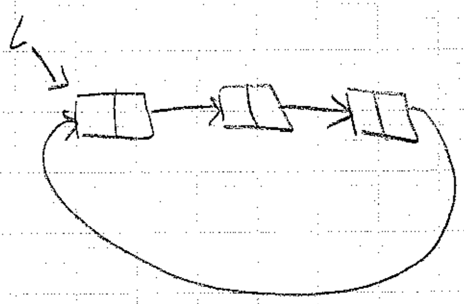
Concaténation_recursif (L_1, L_2) {
 si (L_1 = nil) alors
  retourner L_2
 sinon
  L_1.suivant ← concaténat_recursif(L_1.suivant, L_2)

 retourner L_1;
}

VI) Variantes et listes doublement chaînées



<u>liste doublement chaînée</u> ⇒ permet d'aller ds les 2 sens
 2 pointeurs : suiv
      précédent (au départ = nil)



<u>liste circulaire</u> ⇒ pas de pointeur à nil

<u>NB</u> : tous les algos peuvent ê adaptés pr ces variantes

Chap. 5 :

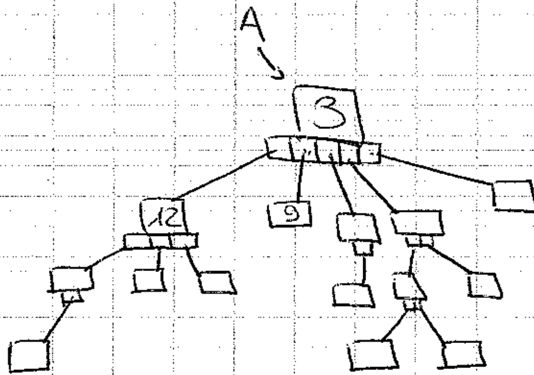
Les arbres

I) Définition

Σ dynamiq défini récursivement par :

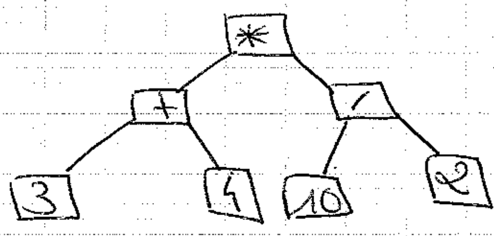
- 1 racine contenant de l'informat°
- 1 liste de pointeurs vers des arbres

II) exemples



N.B : ds représentat° on ne met ⊕ les cases pr représenter les pointeurs (⊕ lisible)

Algo
07/1/06

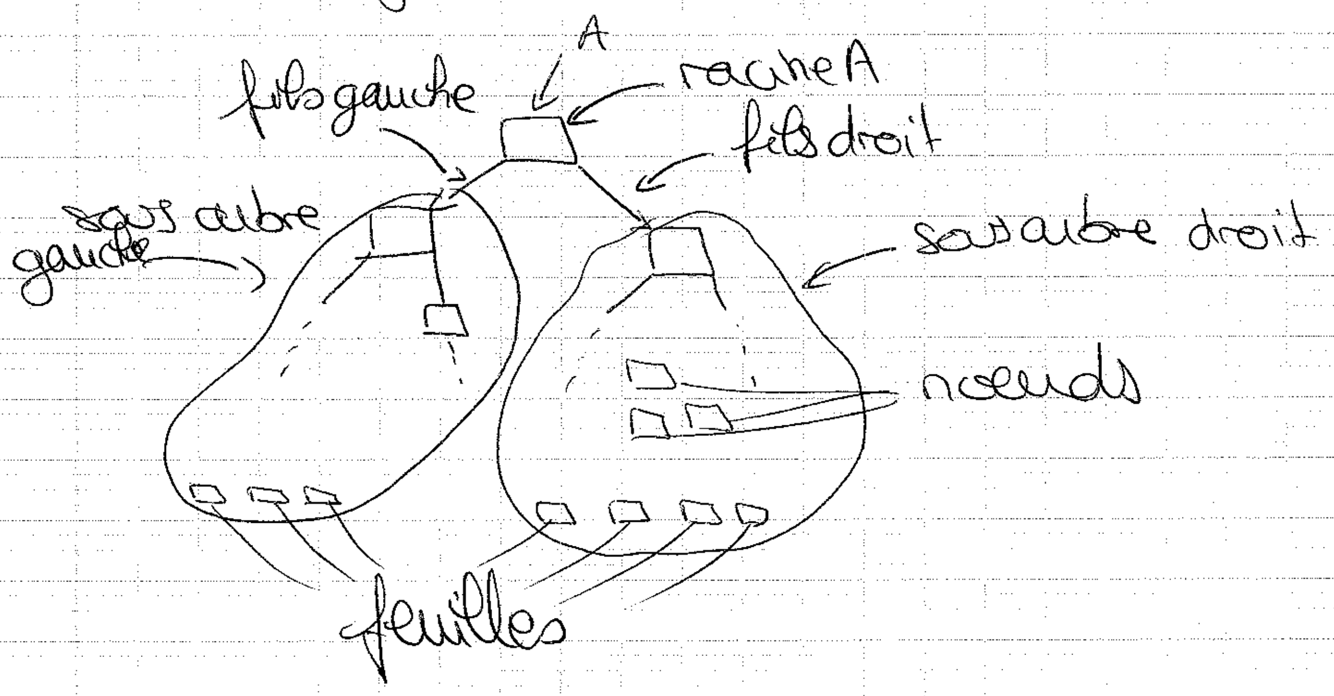


représente $(3+4)*(10/2)$

III) arbres binaires

Def = 1 arbre est ttes les listes possédant au ⊕ 2 éléments. ↳(pointeurs)

terminologie =

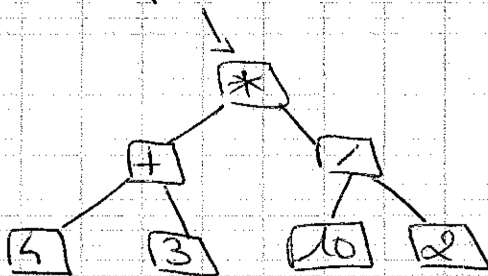


⟹ ts les noeuds ont au max 2 fils

Déf = arbres binaires complets
ttes les listes possèdent :
- soit 2 éléments
- soit 0 élément
(2 fils ou 0 fils mais jamais 1 fils)

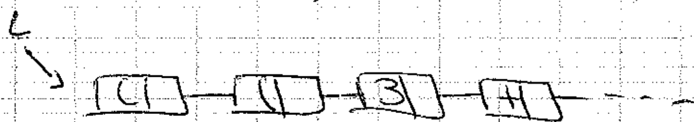
arbres binaires équilibrés
ttes les feuilles st au m̂ niveau
(distance à la racine)
⇔ nb de pointeur qu'il faut suivre à partir de la racine pr aller jusqu'à la feuille.

IV) représentation des arbres binaires



1) représentat° par 1 liste

((3 + 4) * (10 / 2))



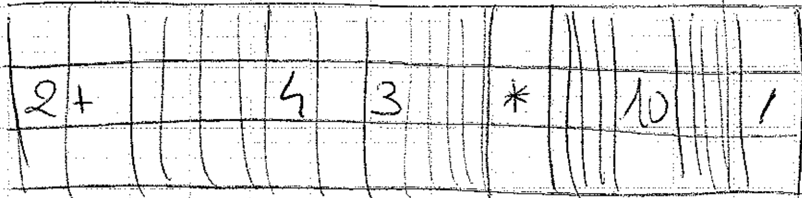
Algo
07/11/06

2) représentation par 1 tableau

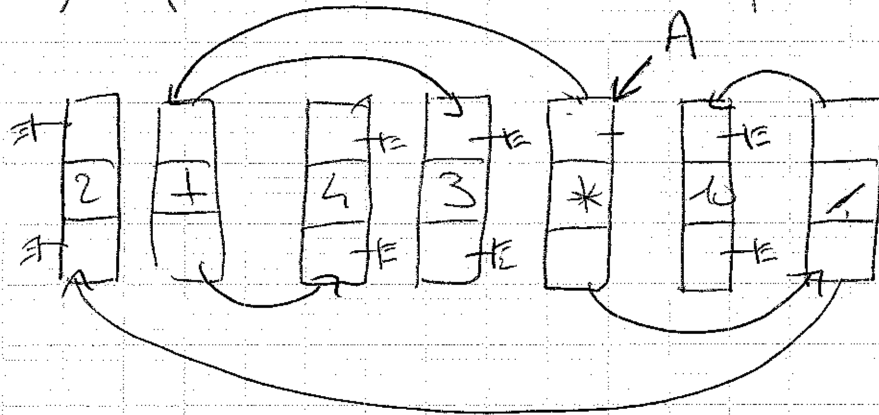
	0	1	2	3	4	5	6
fils gauche	-1	3	-1	-1	1	-1	5
info	2	+	4	3	*	10	/
fils droit	-1	2	-1	-1	6	-1	0

hors indices
<-> pr montrer ∄

=> tableau <=> mémoire



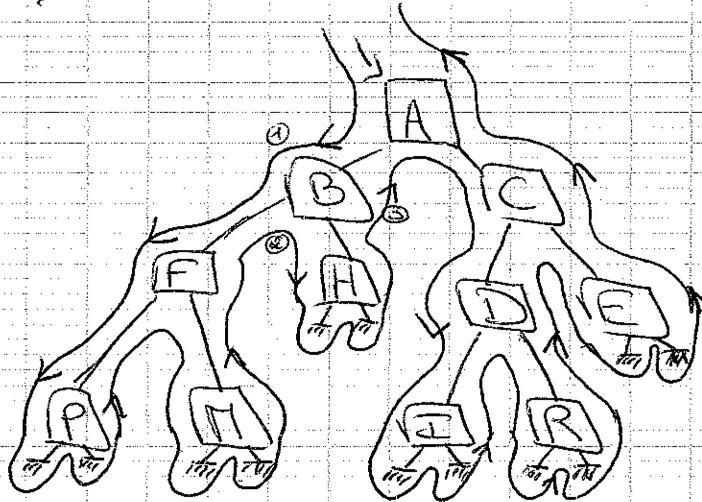
3) représentat° par des pointeurs



Arbre binaire

- info
- fils gauche = pointeur sur arbre binaire éventuel
- fils droit = " " " "

V) Parcours d'arbres binaires

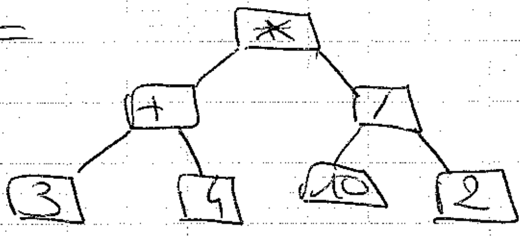


Parcours en "suivant les murs"
chq noeud ou feuille est vu 3 fois :
- 1 en descendant
- 1 entre ses 2 fils
- 1 en remontant

```
Parcours_recursif (A) {
    si (A ≠ nil) alors {
        traitement_~~descendant~~ [préfixe] (A.info);
        parcours_recursif (A.fils_gauche);
        traitement_~~intermédiaire~~ [infixe] (A.info);
        parcours_recursif (A.fils_droit);
        traitement_remontant [post_fixe] (A.info);
    }
}
```

Algo
07/11/06

ex =



descendant : afficher info * + 3 4 / 10 2
préfixe

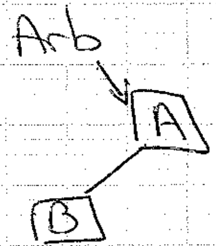
intermédiaire : afficher info 3 + 4 * 10 / 2
infixe

remontant : afficher info 3 4 + 10 2 / *
post fixe

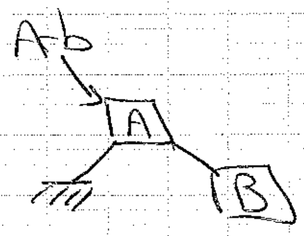
exercice =

un arbre Arb (binaire

- affichage préfixe A B D H M N E I J O C F G K P L
peut-on construire Arb?



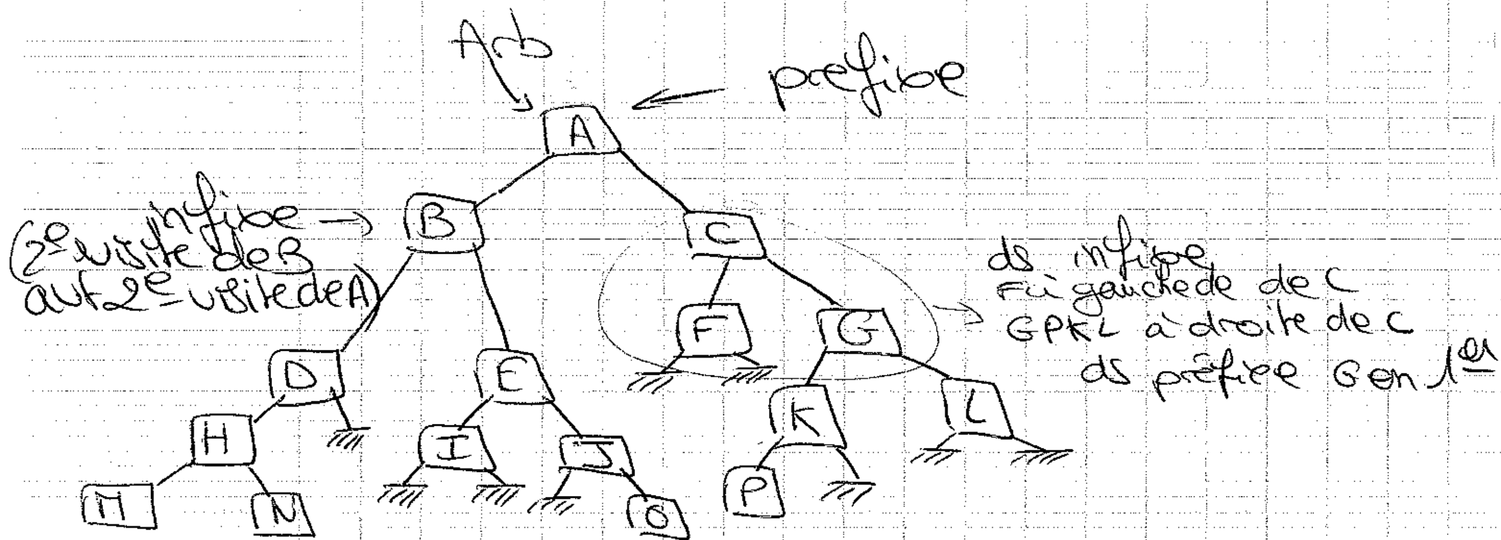
ou



=> NON

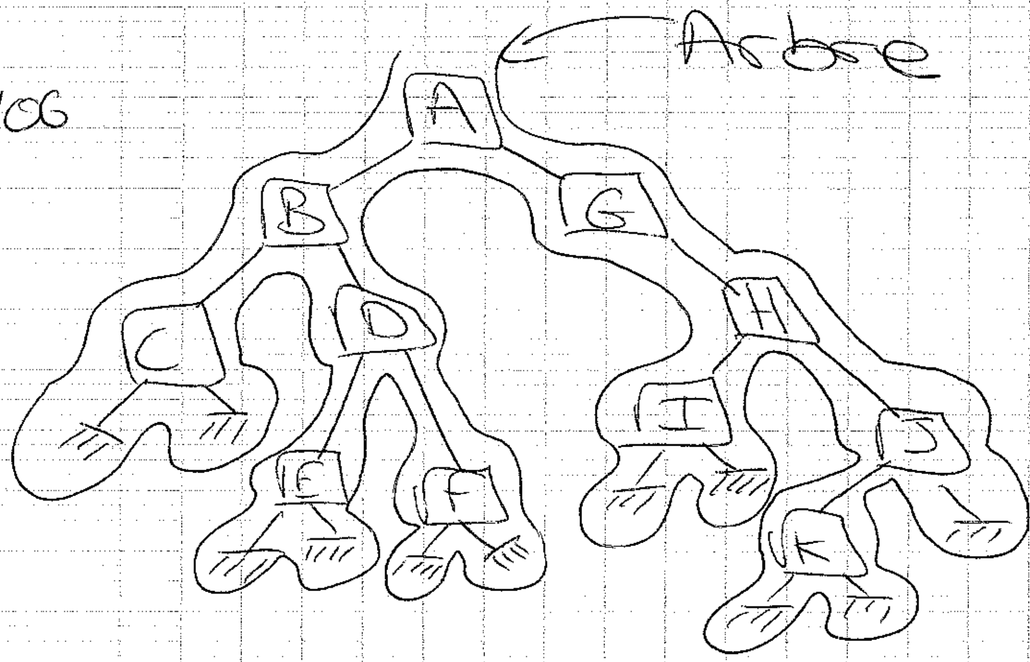
plus

- affichage infixe M H N D B I E J O A F C P K G L (sous arbre gauche : M H N D B I E J O ; sous arbre droit : F C P K G L)
peut-on construire Arb? (SAG, SAD)



écrire 1 parcours d'arbre itératif
⇒ utiliser 1 struct. de piles

Algo
14/11/06



Algo. parcours_iteratif (Arbre)

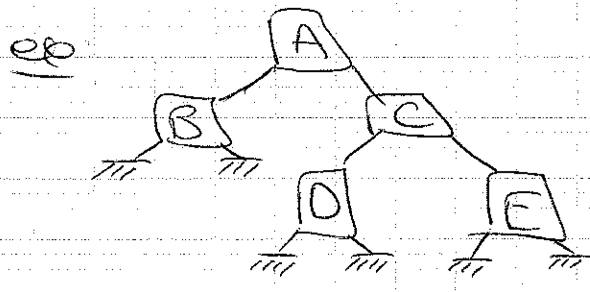
14/11/06

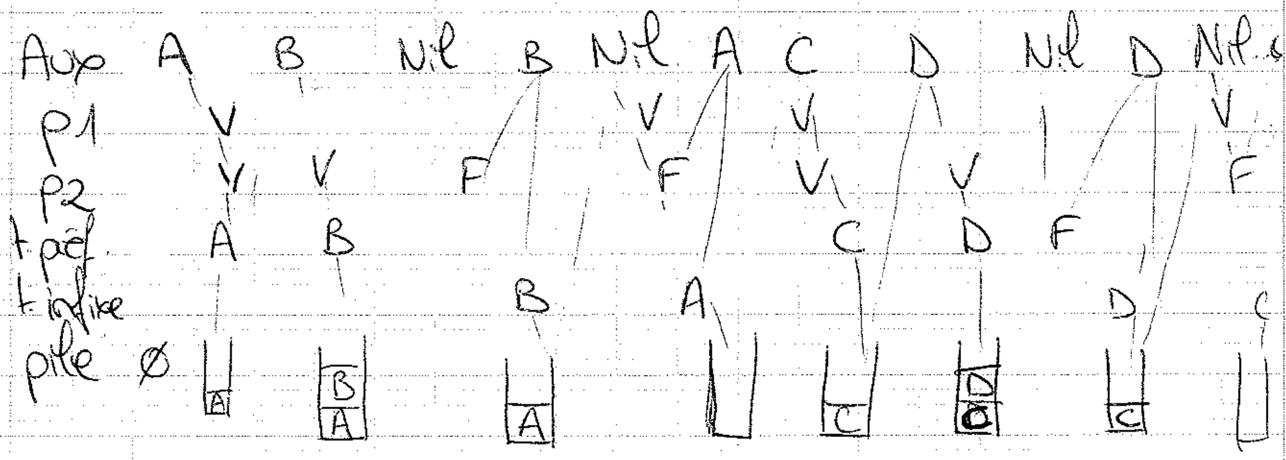
{
- Aux ← Arbre ;
- P ← créer_pile_vide() ;
- ~~tant que (Aux ≠ nil) ou (non pile_vide(P)) faire~~ tant que (non pile_vide(P)) faire
- {
  - tant que (Aux ≠ nil) faire
  - {
    - traiter_préfixé (Aux) ;
    - empiler (P, Aux) ;
    - Aux ← Aux.fils gauche ;
  - }
  - Aux ← sommet_pile(P) ;
  - traiter_infixe (Aux) ;
  - depiler(P) ;
  - Aux ← Aux.fils droit ;
- }

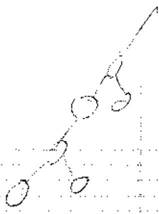
}

savoir le faire

ex

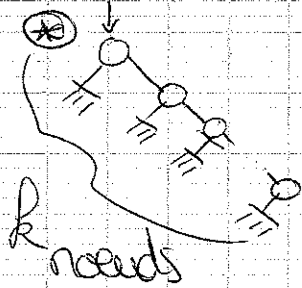




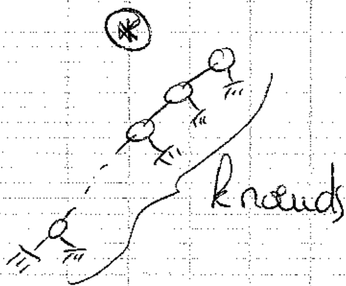


exercice =

Taille maximale de la pile

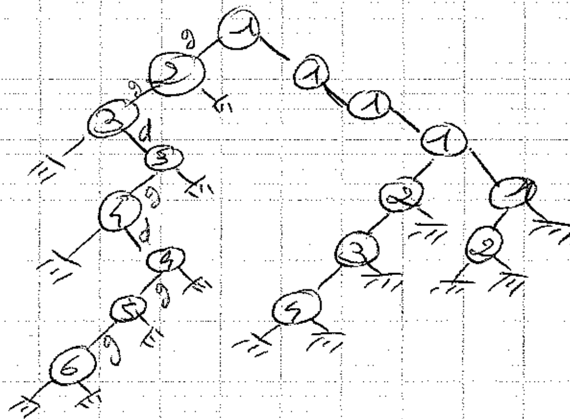


⇒ taille maximale pile = 1
ts les fils gauche st nils.
on ne passe qu'1 fois (à chq noeud) ds la bcle interne



⇒ ts les noeuds st ds la pile à 1 moment donné dc
taille maxi = k

pr 1 arbre binaire qlq



maxi = 6

1 chemin q va de la racine à 1 noeud
gdggd...
taille maxi = 1 + maxi(g) ds ts les chemins

Algo
14/11/06

VI) Algorithmes de recherche dans 1 arbre binaire

rechercher si x est 1 noeud de l'arbre :

```
recherche (Arbre x) {
    si (Arbre = nil) alors
        retourner Faux
    sinon si (Arbre.inf = x) alors
        retourner Vrai
    sinon
        trouve ← recherche (Arbre.filsgauche, x)
        si (trouve) alors          <- recherche = vrai
            retourner Vrai
        sinon
            retourner recherche(Arbre.filsdroit, x)
        }
}
```

```
recherche_iteratif (Arbre, x) {
    Aux ← Arbre
    P ← creer pile vide ()
    ~~trouve ← faux~~
    tant q (Aux ≠ nil) ou (non pile vide(P)) et (~~not trouve~~) faire
        {tant q (Aux ≠ nil) ~~ou~~ (not trouve) faire {
            si (Aux. info = x) alors
                trouve ← vrai
            sinon :
                empiler (Aux)
                Aux ← Aux. filsgauche
        }
        ~~Si not trouve alors~~
        Aux ← sommet pile (P)
        depiler (P)
        Aux ← Aux. fils droit
    }
    }
    retourner trouve
```

```
Si (Aux.info = x) alors
    retourner vrai
empiler (Aux)
Aux ← Aux. fils gauche

retourner faux ; }
}
```

VII / Nombre de sommets d'1 arbre binaire

Algo 14/11/06

```
nb_sommets (Arbre) {
    si (Arbre = nil) alors
        retourner 0
    sinon
        retourner 1 + nb_sommets(Arbre.fg) +
                      nb_sommets (Arbre.fd)
```

```
nb_sommets_itératif (Arbre) {
    Aux ← Arbre
    P ← creer pile vide()
    nb ← 0
    tant q (Aux ≠ nil) ou (non pile vide (P))
        tant q (Aux ≠ nil)
            ~~traiter préfixe (Aux)~~ nb ← nb+1
            empiler (P, Aux)
            Aux ← Aux.fg                      ou

        ~~Aux ← sommet pile(P)~~   nb ← nb+1
        ~~traiter_infixe (Aux)~~
        depiler(P)
        Aux ← Aux.fd
    }
    retourner nb
}
```

Chap.6 :

_ Les arbres binaires de recherche _

I) Définition

1 arbre binaire de recherche (ABR) est 1 arbre binaire t.q :

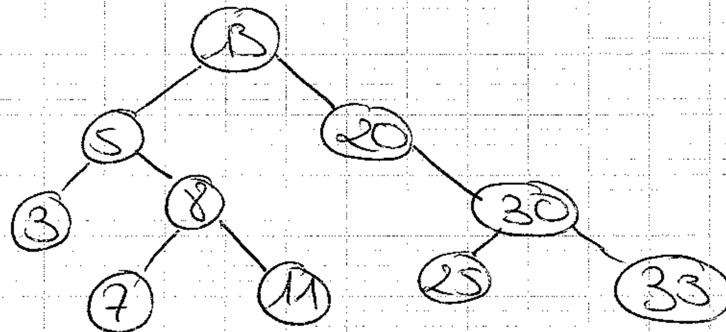
si X est 1 nœud et Y 1 nœud du ss-arbre gauche de X alors :

$Y.info \leq X.info$

si X est 1 nœud et Y 1 nœud du ss-arbre droit de X alors :

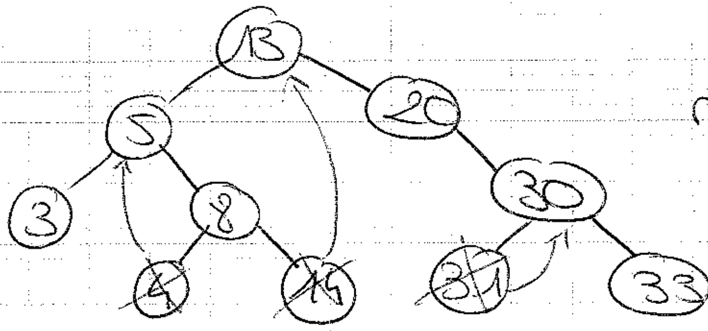
$X.info \leq Y.info$

ex :



est 1 ABR

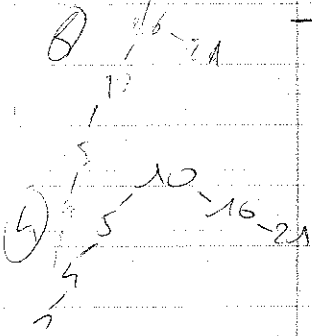
Algo
14/11/08



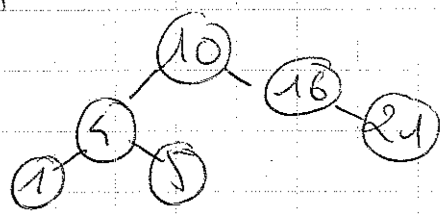
n'est pas 1 ABR

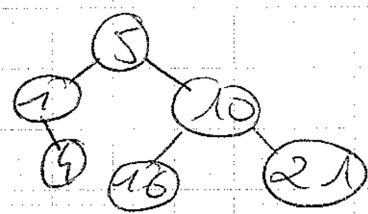
Les ABR st utilisés pr :
- les dictionnaires (ajouter/retirer 1 mot, mot suiv/précédent, trouver 1 mot)
- les files de priorité (min, max)
(struct. efficace pr coder de l'info ordonnée)

exercice :

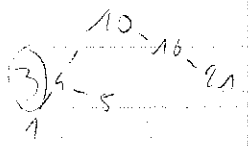


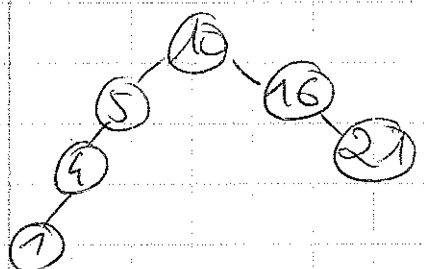
l'info 1, 4, 5, 10, 16, 21
ABR de hauteur 3, 4, 5, 6

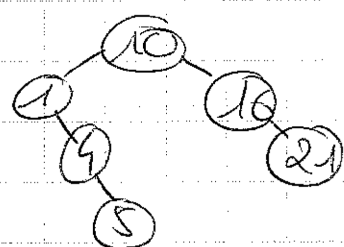




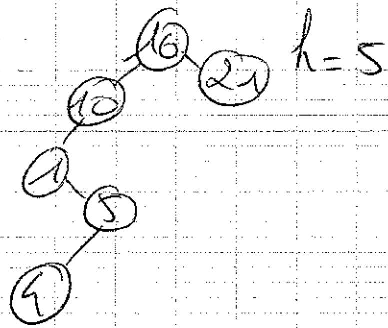
h = 3

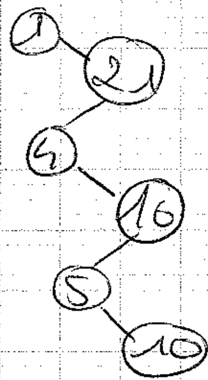




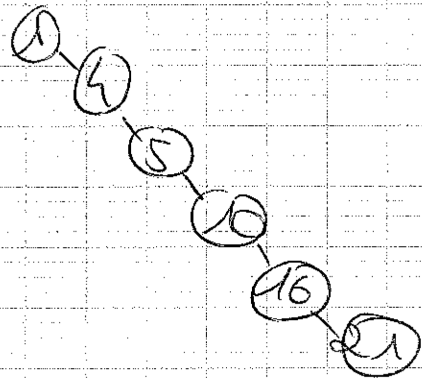


h = 4





h=6



exercice =

parcours infixe des arbres précédents
→ les nb st ordonnés du plus petit au plus gd.

propriété =

Affichage infixe d'1 ABR est la liste des valeurs triées de l'arbre

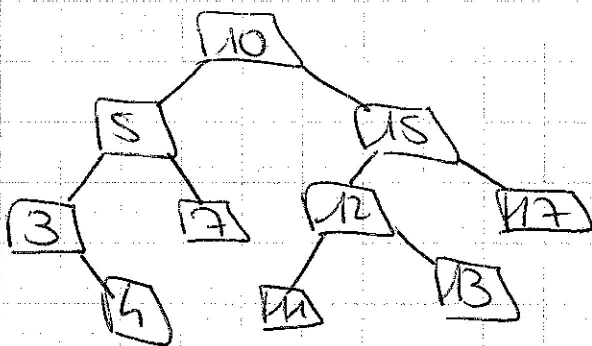
réciproque =

Soit A 1 arbre.
Si l'Affichage infixe de A est 1 liste croissante des valeurs de A, alors A est 1 ABR

Algo
14/11/06

II) Recherche d'1 noeud ds 1ABR

```
recherche(Abr, x) {
    si (Abr = nil) alors
        retourner faux
    sinon si (Abr.info = x) alors
        retourner vrai
    sinon si (x < Abr.info) alors
        retourner recherche(Abr.fg, x)
    sinon
        retourner recherche(Abr.fd, x)
}
```




```
recherche_iteratif(Abr, valeur) {
// retourne 1 pointeur sur une € q contient k
// retourne nil sinon.
    Aux ← Abr;
    tant q (Aux ≠ nil et Aux.info ≠ valeur) f {
        si (Aux.info > valeur) alors
            Aux ← Aux.fg
        sinon Aux ← Aux.fd
    }
```

retourner Aux
}

<u>complexité ABR_recherche =</u>

max $\Theta(h)$ h = hauteur de l'arbre

<u>très sur.</u>
h $\Theta(\log n)$ n = nb de noeuds

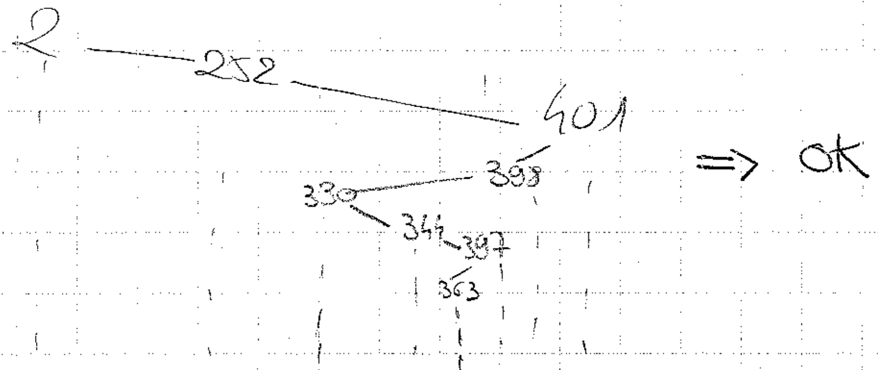
```
Arbre_minimum (Abr) {
 // retourne 1 pointeur sur le noeud
de valeur minimale
 // retourne nil si ∄ (Abr vide)
   Aux ← Abr
   si (Aux = nil) alors:
      retourner Aux
   sinon {
      tant q (Aux.fg ≠ nil) faire {
           Aux ← Aux.fg
      retourner Aux
      }
   }
}
```

Algo
21/11/06

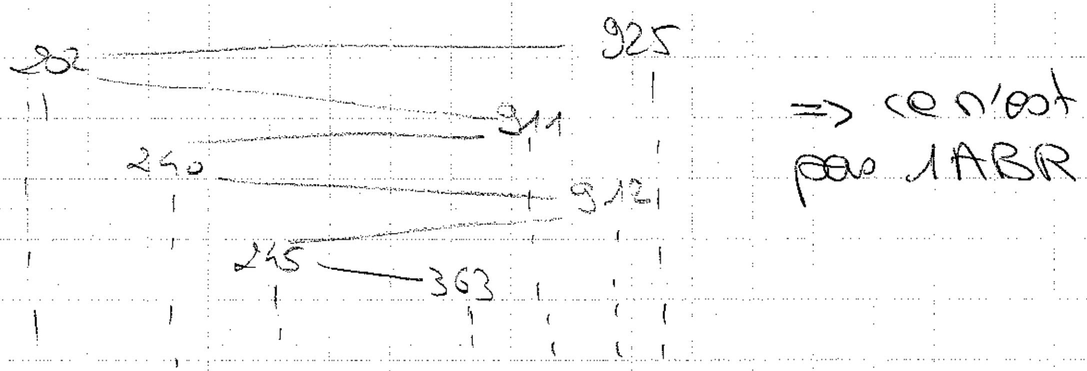
Exercice =
soit 1 arbre, la recherche de 363 donne les listes suites de noeuds visités

- 2, 252, 401, 398, 330, 344, 397, 363
- 925, 202, 911, 240, 912, 245, 363

Les arbres st-ils des ABR ?

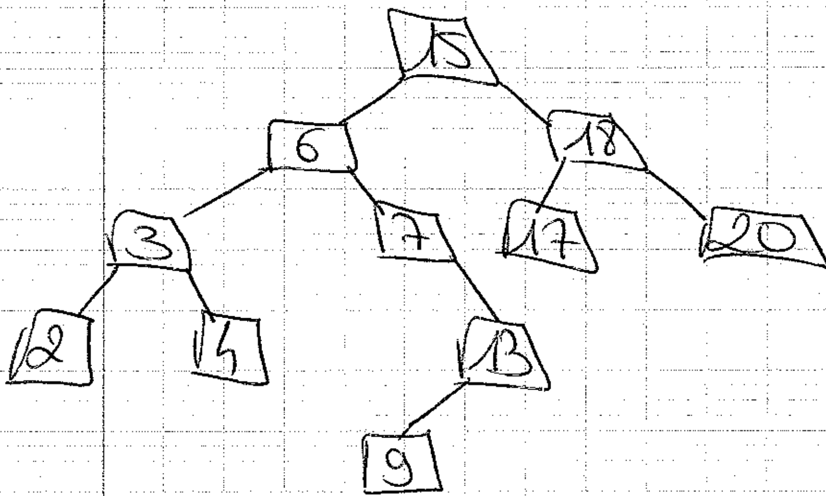


⇒ OK



⇒ ce n'est pas 1 ABR

III) Successeur d'1 nœud



successeur = ⊕ petite valeur suite

succ(3) = 4
Succ(13) = 15
succ(20) = nil

<u>Idée</u> =

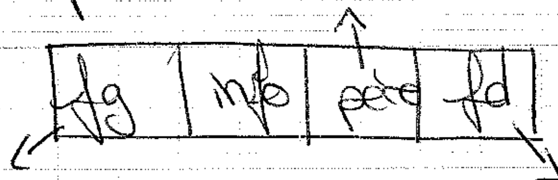
- Si x.fd ∃ succ(x) = min(x.fd)
- Si x.fd ∄ succ(x) est le nœud en sommet de pile lors d'1 parcours itératif avec pile

<u>Rem</u> =

- on peut écrire 1 algo avec 1 pile
- on préfère changer la structure de

Algo
21/11/06

codage par 1 ABR

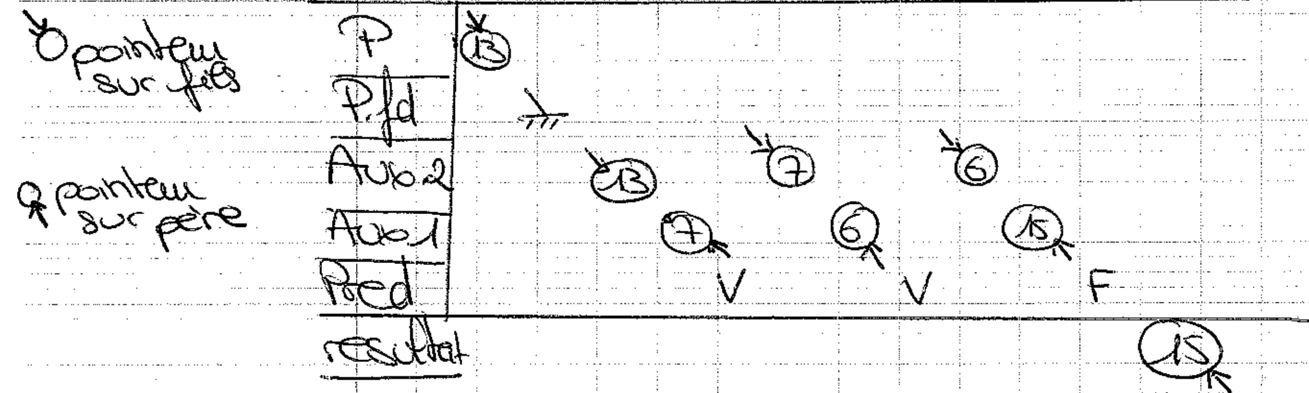


fg et fd ⇒ descendre ds l'arbre
père ⇒ remonter ds l'arbre.

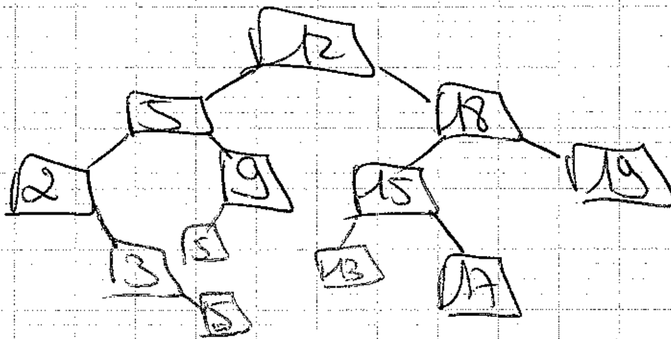
si x.fd ∄ succ(x) est l'ancêtre(x) dont le fg est également l'ancêtre de x.

```
Arbre_successeur(Abr, P) {
  // P est 1 noeud de Abr ≠ nil
  // retourne 1 pointeur Q t.q. succ(P.info) = Q.info
    si (P.fd ≠ nil) alors
        retourner Arbre_minimum(P.fd)
    sinon {
        Aux1 ← P.pere
        Aux2 ← P                    P = predisat = pred
        tant q (Aux1 ≠ nil et Aux1.fd = Aux2) {
            Aux2 ← Aux1;
            Aux1 ← Aux2.pere
        }
        retourner Aux1
}
```

Arbre_successeur (13)



IV) Insertion dans 1 ABR



insérer (3) → droite de 2
insérer (13) → gauche de 15
insérer (5) → 2 possibilités

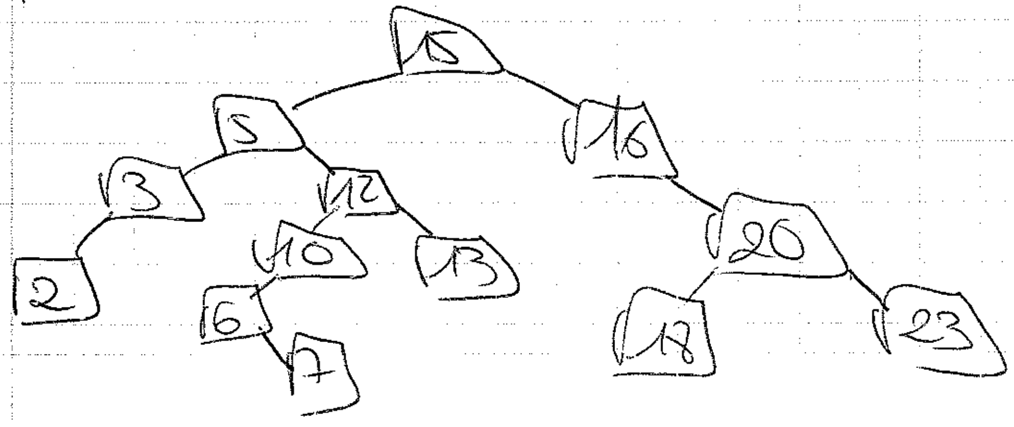
Idée :
- insérer des feuilles
- algo similaire à rechercher (valeur)

Algo 21/11/06

Arbre_inserer (Abr, P) {
(*) sinon, P contient 1 info,
    P.Aux ← Abr.pere
    Aux ← Abr
    tant q (Aux ≠ nil) faire {
        P.Aux ← Aux
        si (P.info < Aux.info) alors
            Aux ← Aux.fg
        sinon
            Aux ← Aux.fd
    }
    si (P.info < P_aux.info) alors {
        P_Aux.fg ← P
        P.pere ← P_aux
    sinon
        P_Aux.fd ← P
        P.pere ← P_aux
    }

(*) si (Abr = nil) alors
    Abr ← P

V) Supprimer dans 1 ABR



supprimer
feuille : (facile)
1 seul fils : on fait 1 pont
2 fils :
-> on remplace le successeur
-> on supprime le successeur
(il n'a q'1 fils)

Algo
01/12/06

Chap 7 :

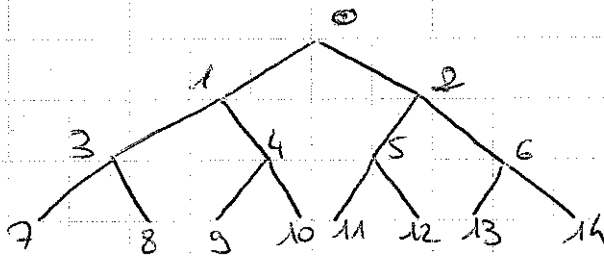
— Tri par tas (heapsort) —

I) Arbre binaire complet

- Ts les noeuds ont 2 feuilles ou 0.
- Ttes les feuilles st au m̂ niveau.

II) Representat° par 1 tableau

On numérote les noeuds par niveau de gauche à droite



ex: $H = 4$
$len(T) = 2^4 - 1 = 15$

- hauteur de l'arbre : H

→ tableau de taille : $2^H - 1$
tableau $[0, 2^H - 2]$

- racine : T[0]
- soit x 1 noeud : $X.fg = 2.X+1$

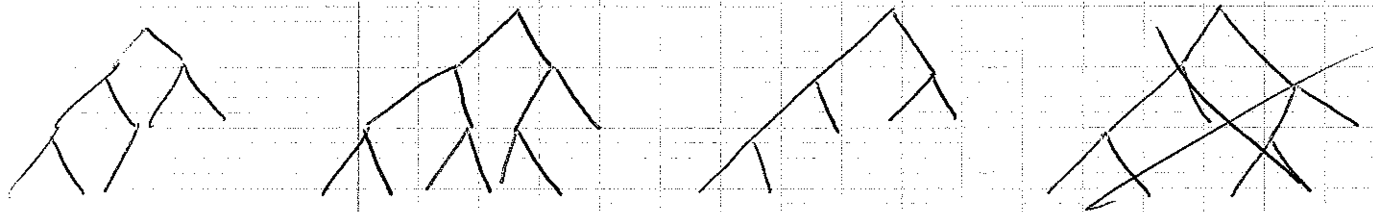
$X.fd = 2X+2$
$= 2(X+1)$

$X.pere = \frac{(X-1)}{2}$ (partie entière)

$\lfloor (X-1)/2 \rfloor$

III) Arbre binaire complet au sens large

- arbre binaire complet jusqu'à l'avant dernier niveau
- le dernier niveau est partiellement rempli à partir de la gauche



Propriété :
1 tableau de taille quelconque représente 1 arbre binaire complet au sens large

Algo
01/12/06

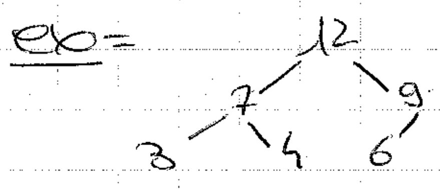
IV) Tas

Déf = 1 tas est 1 arbre binaire complet au sens large t.q :

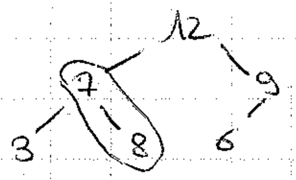
. $\forall X$ $\quad X.pere.info \geq X.info$ si $X.pere\ \exists$,

sinon $X.info \geq \begin{vmatrix} X.fd.info \\ X.fg.info \end{vmatrix}$

ex =



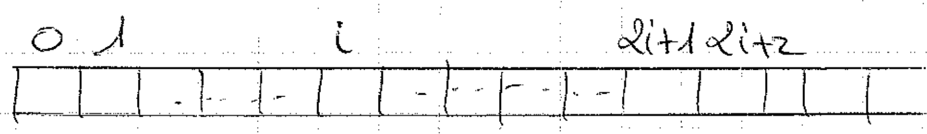
est 1 TAS
Rem = ce n'est pas 1 ABR



n'est pas 1 TAS

Propriété =

1 tableau trié en ordre décroissant est 1 tas



tri décroissant $\Rightarrow$ $T[i] \geq T[2i+1]$
$T[i] \geq T[2i+2]$

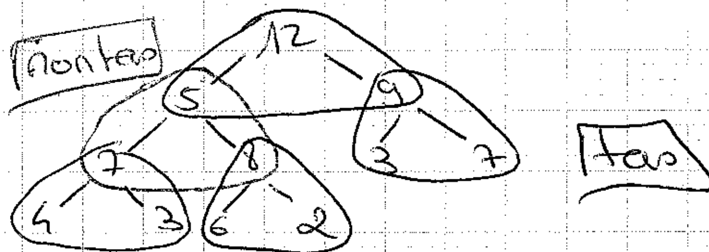
La réciproque est fausse.

Les TAS s't utilisés pr gérer des tâches avec priorités et permettent de trier.

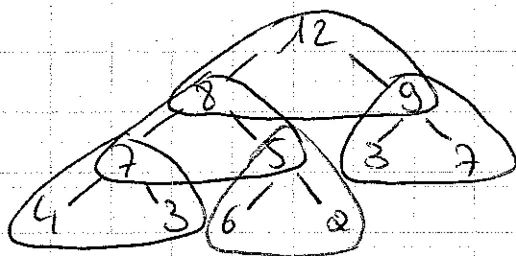
Primitives =

entasser (tas, x)
retirer_maximum (tas)

V) Algorithme pour entasser

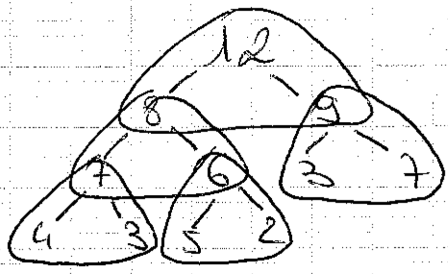


comment faire pr rendre cette arbre tas ?
5 échangé avec le ⊕ gd de ses fils (max(fils))



5 échgé avec le max(fils)

Algo
01/12/06



=> algo sur 1 tableau Tas de taille N

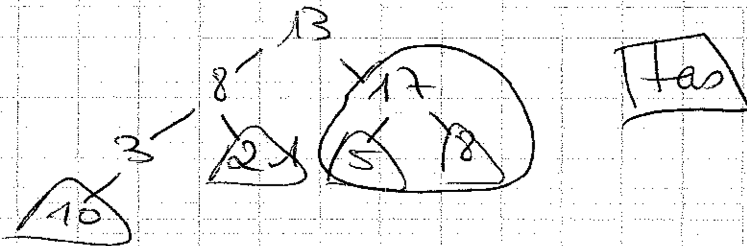
```
algo Entasser (Tas, i, N){      # Tas est "presq 1 tas"
    si(2i+1 >= N) alors         # i est 1 feuille, rien à faire
    |
    sinon: si(2i+2 >= N) alors {    # i a 1 seul fils
        si (Tas[i] < Tas[2i+1]) alors
            echange(Tas[i], Tas[2i+1]);
        sinon   # rien à faire
    }

        sinon { #i a 2 fils
            si Tas[i] < max (Tas[2i+1], Tas[2i+2]) alors
                si (Tas[2i+1] >= Tas[2i+2]) alors
                    max <- 2i+1
                sinon
                    max <- 2i+2
                echanger (T[i], T[max])
                entasser (Tas, max)
        }
    }
}
```

VI) Construction d'1 tas à partir d'1 tableau

Pb = on a 1 tableau
on veut q le tableau soit tas.

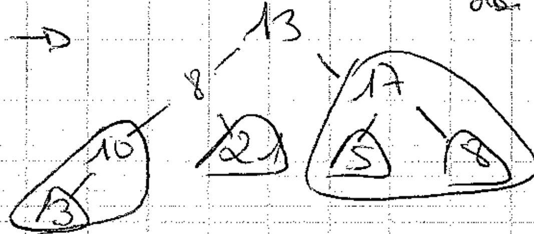
Idée =

13	8	17	3	21	5	8	10

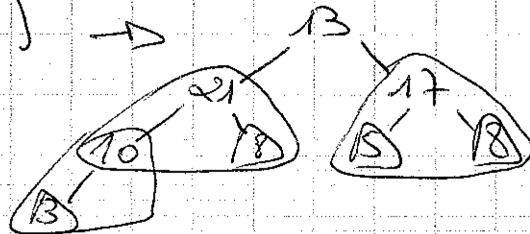


les feuilles sont des tas (parcours tableau de droite à gauche)

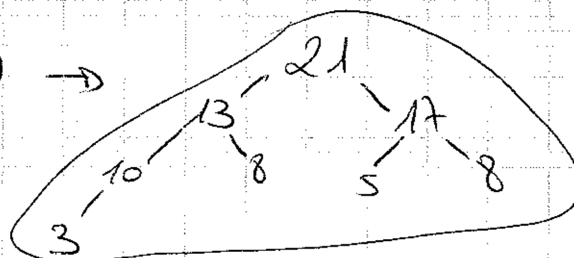
entasser ("3") →



entasser ("8") →



entasser ("13") →



Algo 01/12/06 donne le tableau : | 21 | 13 | 17 | 10 | 8 | 5 | 8 | 3 |

algo Construire_Tas (T) { # T [0, N-1]

pour i de père (N-1) à 0 faire { # père(N-1) = (N-2)/2
    entasser (T, i) ;
}
}

VII) Tri par tas

Pb = Trier 1 tableau en ordre croissant

Idée = Utiliser la struct. de Tas

13	8	17	3	21	5	8	10

—créer_TAS→

21	13	17	10	8	5	8	3

↓ échange racine, dernière feuille

représente 2 tas ("13") et ("17")

3	13	17	10	8	5	8	21

⟹ il suffit d'entasser ("3")

←créer TAS

17	13	8	10	8	5	3	21

↓ échange ("17", "3")

3	13	8	10	8	5	17	21

[bien trié

↳ itérer le processus

algo

```
Tri_par_Tas (T) {    # T[0, N-1]
    construire_Tas(T);
    pour i de N-1 à 1 faire {
        echanger (T[0], T[i]);
        entasser (T, 0, i)    # i limite
    }
}
```

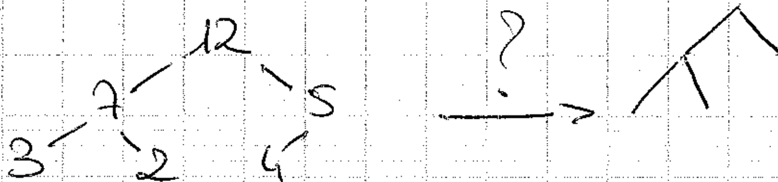
VIII Files de priorité

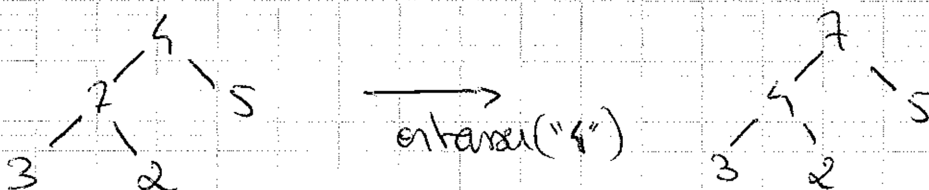
- Tâches à effectuer -> il y a des priorités
- Idée : gestion par 1 Tas

Primitives :

- ajouter_tache(Tas, p)   p = priorité
- extraire_max (Tas)

ex :



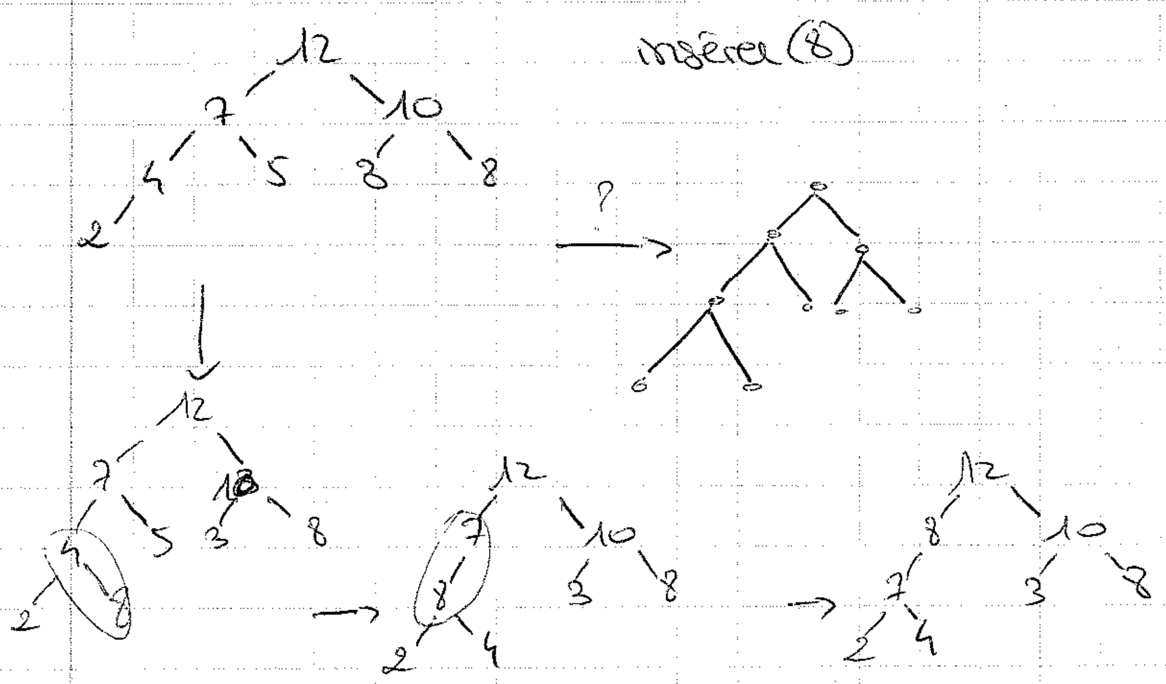


Algo
01/12/06

<u>algo</u> Extraire_max (Tas) {
N ← taille (Tas) ;
max ← Tas[0] ;
Tas[0] ← Tas [N-1] ;
taille (Tas) ← taille (Tas) - 1 ;
entasser (Tas, 0, taille (Tas)) ;
retourner max ;
}

abus : possible pr struct dynamiq mais pas pr tableau →

<u>Idée pr insérer :</u>



<u>algo</u> Insérer (Tas, x) { # T [0, N-1]
X priorité de la tâche

```
taille(Tas) ← taille(Tas) + 1;
i ← taille(Tas) - 1;
Tas[i] ← x;
tant q ((Tas[pere[i]]) < Tas[i]) faire)
      echanger Tas[pere[i]] , Tas[i];
      i ← pere[i];
}
}
```