Algorithmique et structures de données linéaires
HLIN301

P. Janssen : philippe.janssen@lirmm.fr

Université de Montpellier - Faculté des Sciences

Organisation de l'UE

Heures

- 15 heures de cours
- 22,5 heures de TD
- 9 heures de TP

Contrôle des connaissances

La note finale de l'UE est calculée à partir de 2 notes

- une note de contrôle continu (CC) composée de 2 notes :
  - une note de contrôle en cours (début novembre)
  - une note de contrôle en TD basée sur des interrogations, l'activité en TD et TP, la présence en TD/TP.
- une note d'examen (Exam) ($1^{ère}$ session en Janvier, $2^{ème}$ session en juin)

selon la formule :
note UE = Max(note_Exam, (2*note_CC + 3*note_Exam)/5))

Objectifs

Analyse et conception d'algorithmes et Étude de Structures de données

- Preuve d'algorithmes
- Complexité des algorithmes
- Structures de données simples : listes chaînées, piles, files.
- Structures de données arborescentes : arbres binaires, tas binaires.
- Algorithmes de tri

Livres

- *Introduction à l'algorithmique*,
  T. Cormen, C. Leiserson, R. Rivest ;
  Ed. Dunod

Schéma d'algorithme

Algorithme : nom(paramètres)
Données : description des paramètres-donnée de l'algorithme
Résultat : description du résultat : valeur renvoyée par l'algorithme
(ou paramètres modifiés par l'algorithme)
Déclaration des variables;
début
  Partie instructions
fin algorithme

Types

Type = Domaine de valeurs + opérations

- Réels : opérations : +, -, *, /
- Entiers : opérations : +, -, *,
  division euclidienne : quo, mod
- Booléens : opérations : non, et, ou
  opérateurs de comparaisons : <, >, =, $\neq$
- Tableaux : séquence de longueur fixe, d'éléments de même type
  opérateur d'accès à un élément : [ ]
  opération taille renvoie le nombre d'éléments d'un tableau.
  ex : `T` Tableau de 5 entiers,
  `T[2]` désigne le 3ème élément de `T`,
  `taille(T)` vaut 5

Variables

Variable = Nom + Type + éventuellement valeur
La déclaration d'une variable consiste à donner son nom et son type
Pas de valeur par défaut

Environnement

Ensemble d'associations *(Nom, Valeur)*
L'environnement est modifié par l'affection d'une valeur à un nom

Expressions

- *Constante*
- *Nom de Variable*
- *Nom de Variable Tableau*[*Expression*]
- *Expression opération Expression*
- *Opération(Expression)*

L'évaluation dans un environnement d'une expression composée de sous-expressions provoque l'évaluation de toutes les sous-expressions puis l'application de l'opération à ces valeurs.

Exception : l'évaluation des expressions booléennes est *paresseuse* :

Valeur(A et B)	=	si valeur(A)=Faux alors Faux sinon valeur(B)
Valeur(A ou B)	=	si valeur(A)=Vrai alors Vrai sinon valeur(B)

Conséquence : *B* n'est pas toujours évalué ;

ex : `(x` $\neq$ `0) et (50 quo x) > 5` est évaluable $\forall x \in \mathcal{Z}$.

Instructions, Structures de contrôle

- Affectation :$\leftarrow$
- L'instruction `Renvoyer(Expression)` correspond à la directive "`Le résultat est Expression`" utilisée en première année.
- Conditionnelles :

  si `Cond1` **alors**
  | Inst1
  sinon si `Cond2` **alors**
  | Inst2
  sinon
  | Inst3
  fin si

  où `Cond1` et `Cond2` sont des expressions à valeur booléenne
  Les parties `sinon` et `sinonSi` sont optionnelles.

Structures Répétitives

- **pour** `K` **de** `E1` **à** `E2` **faire**
  | Inst
  fin pour

  `K` est l'indice de boucle de type entier
  `E1, E2` 2 expressions à valeur entière.

  - `E1,E2` sont évaluées une fois pour toute avant l'itération.
  - le corps de l'itération ne peut pas modifier la valeur de la variable `K`
  - en sortie de l'itération `Pour` la variable de contrôle `K` n'a pas de valeur.

- **tant que** `Cond` **faire**
  | Inst
  fin tq

  où `Cond` est une expression à valeur booléenne

  Contrairement au `Tant que` , le nombre d'itérations de la structure `Pour` est indépendant de l'instruction itérée `Inst`.

Modes de passage des Paramètre(s) et Résultat(s)

Un algorithme décrit un calcul permettant d'obtenir un résultat à partir de données.
Les **paramètres formels** de l'algorithme correspondent aux données et résultats de l'algorithme.
Un paramètre formel peut être :

- une donnée de l'algorithme, précédé par la lettre d
- un résultat de l'algorithme, précédé par la lettre r
- une donnée et un résultat de l'algorithme, précédé par les lettres dr

Pour appliquer un algorithme, il faut indiquer son nom et ses **arguments** ("valeurs associées aux paramètres formels ").

Modes de passage des Paramètre(s) et Résultat(s)

Algorithme de style fonction

Tous les paramètres sont des données,
Le résultat est déterminé par l'instruction **renvoyer**

Algorithme : nom de l'algo(nom, nature et type des paramètres formels) : type du résultat
Données : description des paramètres données
Résultat : description du résultat
Déclaration des variables;
début
    Partie instructions ;
    renvoyer Expression ;
fin algorithme

Le corps de l'algorithme doit contenir l'instruction **renvoyer** `Expression`.
Le résultat calculé par l'algorithme est la valeur de l'expression `Expression`.
L'application d'un algorithme de style fonction correspond à une expression dont la valeur est le résultat calculé par l'algorithme.

Exemple d'algorithme de style fonction

Algorithme : noteUE(d NoteExam : Réel, d NotePartiel : Réel) : Réel
Données : NoteExam et NotePartiel sont 2 réels de l'intervalle [0 ;20]
Résultat : Renvoie la note finale du module sur 20 obtenue à partir des notes d'examen et de partiel

```
Variables Res : Réel;
début
    si NotePartiel<NoteExam alors
        Res ← NoteExam
    sinon
        Res ← (3*NoteExam+2*NotePartiel)/5
    fin si
    renvoyer Res
fin algorithme
```

Exemple d'application

```
...;
exam←11;
note ← noteUE(exam,8.5) ;
/* note=11, exam=11 */
...;
```

Modes de passage des Paramètre(s) et Résultat(s)

Algorithme de style procédure

Données et résultats sont des paramètres de l'algorithme.

Algorithme : nom de l'algo(nom, nature et type des paramètres formels)
Données : description des paramètres données
Résultat : description des paramètres résultat
Déclaration des variables;
début
   Partie instructions
fin algorithme

Le corps de l'algorithme ne doit pas contenir l'instruction **renvoyer**
L'application d'un algorithme de style procédure correspond à une instruction modifiant les valeurs des arguments associés aux paramètres résultats (et données–résultats).

Exemple d'algorithme de style procédure

Algorithme : échanger(dr a : Entier, dr b : Entier)
Données : a et b 2 entiers ; `(a=x, b=y)`
Résultat : Echange les valeurs de a et b ; `(a=y , b=x)`
Variables c : Entier;
début

$c \leftarrow a$;
$a \leftarrow b$;   PAS D'INSTRUCTION RENVOYER ! ! !
$b \leftarrow c$;

fin algorithme

Exemple d'application

...;
$P \leftarrow 11$;
$Q \leftarrow 3$;
échanger(P,Q);
`/* P=3; Q=11 */`

Exemple avec toutes les formes de paramètre formel

Algorithme : CalculFinal(dr NoteUE : Réel, d PtJury : Réel, r ECTS : Entier)
Données : `NoteUE` $\in$ [0 ;20] et `PtJury` $\geqslant$ 0 ; (`NoteUE=n,PtJury=p`)
Résultat : Ajoute les Points Jury à la note de l'UE ; Si la note obtenue est inférieure à 10 le nombre d'ECTS est 0, sinon il vaut 5 ; (`NoteUE=p+n; Si p+n <10 alors ECTS=0 sinon ECTS=5`)

```
début
    NoteUE ← NoteUE+PtJury ;        PAS D'INSTRUCTION RENVOYER !!!
    si NoteUE < 10 alors ECTS ← 0
    sinon
        ECTS ← 5
    fin si
fin algorithme
```

Exemple d'application

ue ← 9.75 ; pj ← 0.25;
CalculFinal(ue,pj,crédit) ;
`/* ue=10, pj=0.25, crédit=5 */`

Modes de passage des Paramètre(s) et Résultat(s)

Correspondance entre paramètre formel et argument lors de l'application d'un algorithme

- Avant l'exécution du corps de l'algorithme :
  Pour chaque paramètre donnée et donnée-résultat, la valeur de l'argument est affectée au paramètre formel correspondant.
- Exécution du Corps de l'algorithme
- Après l'exécution du corps de l'algorithme :
  Pour chaque paramètre résultat et donnée-résultat, la valeur du paramètre formel est affectée à l'argument correspondant.

Algorithme : monAlgo(d A : ..., dr B : ..., r C : ...)
Données : A et B
Résultat : B et C
début
| Corps de l'algorithme
fin algorithme

monAlgo(E1,E2,E3)

A←E1 ;B←E2;

début
| Corps de l'algorithme
fin algorithme

E2←B ;E3←C;

ue←9.75 ; pj←0.25;
CalculFinal(ue,pj,crédit) ;

ue←9.75 ; pj←0.25

NoteUE←ue ; PtJury ←pj

début

NoteUE ← NoteUE+PtJury
si `NoteUE < 10` **alors** ECTS←0
sinon ECTS←5

fin algorithme

ue←NoteUE ;credit←ECTS;

Modes de passage des Paramètre(s) et Résultat(s)

Contraintes liant paramètres et arguments

	Paramètre donnée	Paramètre résultat	paramètre donnée-résultat
Argument	Expression quelconque	Expression de variable	Expression de variable
Avant l'appel	doit avoir une valeur		doit avoir une valeur
Après l'appel	même valeur qu'avant		
Dans le corps de l'algorithme	ne doit pas être modifié	doit être modifié	

Expression de variable : expression pouvant être en partie gauche d'affection
Exemple : `A, T[i+1]`
Contre-exemple : `3, A+1`

Modes de passage des Paramètre(s) et Résultat(s)

Le principe de passage de paramètre décrit n'est qu'un modèle. Les mécanismes réels dépendent des langages et compilateurs.

En C, Scheme, Maple, Caml tous les paramètres sont des paramètres-données, donc non modifiables. Cependant en C, il est possible d'avoir une adresse comme paramètre. Dans ce cas, l'adresse-paramètre ne peut être modifiée, mais son contenu si. On peut ainsi par effet de bord simuler un paramètre résultat et/ou donnée-résultat.

exemple : traduction de l'algorithme `échanger` en langage C

```
void echanger(int *a,int *b)
{ int c; c=*a; *a=*b; *b=c; }
```

```
...
echanger(&X,&Y);
...
```

Traduction en C de la fonction CalculFinal

```
void CalculFinal(float *NoteUE, float PtJury, int *ECTS)
{
*NoteUE = *NoteUE + PtJury;
if(*NoteUE < 10) {*ECTS=0;}
else{*ECTS=5;}
}
```

```
int main()
{
int credit; float ue,pj;

ue=9.5; pj=0.5;
CalculFinal(&ue,pj,&credit);
... }
```

Mécanisme de passage de paramètres en C++

Par défaut, le passage de paramètre en C++ est comme en C, un passage par valeur. Il existe également un passage de paramètre **par référence**, qui correspond au mode de paramètre *donnée/résultat* de notre langage algorithmique.

Traduction en C++ de la fonction CalculFinal

```
void CalculFinal(float& NoteUE, float PtJury, int& ECTS)
{
NoteUE = NoteUE + PtJury;
if(NoteUE < 10) ECTS=0;}
else ECTS=5;}
}

int main() {
int credit; float ue,pj;
ue=9.5; pj=0.5;
CalculFinal(ue, pj, credit);
... }
```

Algorithmes de Recherche dans un tableau

Le problème

Algorithme : Recherche(d T : tableau de *type*, d e : *type*) : Booléen
Données : `T` un tableau, `e`
Résultat : Renvoie Vrai si `e` est élément de `T`, renvoie Faux sinon

0							taille(T)−1
5	22	13	5	2	67	13	20

Algorithmes de Recherche dans un tableau

Recherche Séquentielle de 10

5	22	13	5	2	67	13	20

Faux

Recherche Séquentielle

Algorithme : Recherche(d T : tableau, d e) : Booléen
Données : `T` un tableau, `e`
Résultat : Renvoie Vrai si `e` est élément de `T`, renvoie Faux sinon
variables : $i \in \mathbb{N}$;
début
  $i \leftarrow 0$;
  tant que i `< taille(`T`) et` $T[i] \neq e$ **faire**
    $i \leftarrow i + 1$
  fin tq
  /* $i \geqslant$ `taille(T) ou` $T[i] = e$ */
  si i `< taille(T)` **alors renvoyer** `Vrai`;
  sinon renvoyer `Faux`;
  Ou mieux
  renvoyer i `< taille(T)`
fin algorithme

Erreur Classique

```
début
    i ← 0;
    tant que  T[i] ≠ e et i < taille(T) faire
        i ← i + 1
    fin tq
    renvoyer (i < taille(T))
fin algorithme
```

Cas du tableau trié

Algorithme : Recherche(d T : tableau de *type*, d e : *type*) : Booléen
Données : `T` un tableau trié $\nearrow$, `e`
Résultat : Renvoie Vrai si `e` est élément de `T`, renvoie Faux sinon

0							taille(T)−1
2	5	5	13	13	20	22	67

Algorithmes de Recherche dans un tableau

Recherche Séquentielle de 10

2	5	5	13	13	20	22	67

Faux

Algorithmes de Recherche dans un tableau

Recherche Séquentielle

Algorithme : Recherche(d T : tableau, d e) : Booléen
Données : `T` un tableau trié $\nearrow$, `e`
Résultat : Renvoie Vrai si `e` est élément de `T`, renvoie Faux sinon
variables : $i \in \mathbb{N}$;
début

$i \leftarrow 0$;
tant que i `< taille(`T`) et` $T[i] < e$ **faire**
  $i \leftarrow i + 1$
fin tq
`/*` $i \geqslant$ `taille(`T`) ou` $T[i] > e$ `ou` $T[i] = e$ `*/`
renvoyer `(`i `< taille(`T`) et` $T[i] = e$`)`

fin algorithme

Recherche Dichotomique de 5

2	5	5	13	13	20	22	67

Vrai

Algorithmes de Recherche dichotomique

Recherche Dichotomique de 21

2	5	5	13	13	20	22	67

Faux

Algorithmes de Recherche dichotomique

Algorithme : RechercheDicho(d T : Tableau, d e) : Booléen
Données : T trié $\nearrow$ et e
Résultat : renvoie Vrai si e est élément de T, Faux sinon
Variables : $Deb,\ Fin,\ Mil \in \mathbb{N}$, $Trouve : Booleen$
début

```
    Deb ← 0 ; Fin ← taille(T) − 1 ; Trouve ← Faux ;
    tant que non(Trouve) et Deb ≤ Fin faire
        /*  ∀i ∈ [0 . . . taille(T)[,  Si  i ∉ [Deb, Fin]  alors  T[i] ≠ e    */
        Mil ← (Deb + Fin) quo 2 ;
        si T[Mil] = e alors Trouve ← Vrai
        sinon si e < T[Mil] alors
            Fin ← Mil − 1
        sinon
            Deb ← Mil + 1
        fin si
    fin tq
    renvoyer Trouve
```

fin algorithme

Algorithmes de Recherche dichotomique

Version récursive

Algorithme : RechDicho2(d T ,d e, d Deb : Entier, d Fin : Entier) : Booléen
Données : T trié $\nearrow$, e
Résultat : renvoie Faux si $\forall i \in [Deb \ldots Fin]$ $T[i] \neq e$ Vrai sinon
Variables : $Mil \in \mathbb{N}$
début

```
    si Deb > Fin alors renvoyer Faux
    sinon
        Mil ← (Deb + Fin) quo 2 ;
        si T[Mil] = e alors renvoyer Vrai
        sinon si e < T[Mil] alors
            renvoyer RechDicho2(T, e, Deb, Mil − 1)
        sinon
            renvoyer RechDicho2(T, e, Mil + 1, Fin)
        fin si
    fin si
```

fin algorithme

Pour savoir si e appartient au tableau T on applique
RechDicho2(T, e, 0, taille(T)-1)

Analyse d'un algorithme

- **Preuve de l'algorithme**
  Un algorithme est correct si pour toute valeur des paramètres-donnée vérifiant les spécifications des données :
  - l'exécution de la partie instructions s'arrête
  - le résultat calculé par l'algorithme correspond à la solution du problème, vérifie les spécifications du résultat.
- **Évaluation du coût de l'algorithme**

Preuve de l'arrêt d'un algorithme

Le problème se pose pour les itérations `Tant que`
(et pour la `récursivité`).

Montrer que le nombre d'itérations est fini : exhiber une expression

- dont la valeur décroît (ou croît) strictement à chaque itération
- montrer que cette valeur ne peut pas décroître (ou croître) indéfiniment.

Ex : expression strictement décroissante à valeur entière ($\in \mathbb{N}$)

Exemple

Algorithme : algo1(d A, d B, r Z)
Données : $A, B \in \mathbb{N}$
Résultat : $Z = ?$

Variables : X et $Y \in \mathbb{N}$
début
    $X \leftarrow A$; $Y \leftarrow B$; $Z \leftarrow 0$;
    tant que $X \neq 0$ **faire**
        si X `est impair` **alors**
            $Z \leftarrow Z + Y$; $X \leftarrow X - 1$;
        fin si
        $Y \leftarrow 2 * Y$; $X \leftarrow X/2$;
    fin tq
fin algorithme

Arrêt de l'algorithme

- $X \in \mathbb{N}$
- à chaque itération (si $X > 0$) X décroît strictement.

Notations

X_k ($k \geq 0$) : valeur de X après la k^{eme} itération
X_0 : la valeur initiale de X

- $X_0, X_1, X_2, \ldots, X_n$ est une suite entière positive strictement décroissante, et donc finie.

Exemple

Algorithme : rechDicho(d T : tableau ,d e,r present : Bool, r ind : entier)
Données : *T* tableau trié $\nearrow$, e
Résultat : Si $e \notin T$ alors $present = Faux$ sinon $present = Vrai$ et *ind* est l'indice maximum de *e* dans *T* ($ind = max\{i \in [1..taille(T)[\ : \ T[i] = e\}$)
Variables : $Deb, \ Fin, \ Mil \in \mathbb{N}$
début
  $Deb \leftarrow 0$; $Fin \leftarrow taille(T) - 1$;
  tant que $Deb \leq Fin$ **faire**
1    $Mil \leftarrow (Deb + Fin)$ *quo* 2 ;
2   **si** $T[Mil] \leq e$ **alors** $Deb \leftarrow Mil + 1$
    sinon
      $Fin \leftarrow Mil - 1$
    fin si
  fin tq
  si . . . **alors** $present \leftarrow Vrai$; $ind \leftarrow$ ...

  sinon
    $present \leftarrow Faux$
  fin si
fin algorithme

Exemple e=7

Deb			Mil			Fin
1	5	8	8	10	12	12

Deb	Mil	Fin				
1	5	8	8	10	12	12

		Mil Deb Fin				
1	5	8	8	10	12	12

	Fin	Deb				
1	5	8	8	10	12	12

num Itération	0	1	2	3
Fin-Deb	6	2	0	-1

Arrêt de l'algorithme RechercheDichotomique

$Fin - Deb \geq -1$ et décroît strictement à chaque itération.

- $Fin_0 - Deb_0 \geq 0$.
- On montre $\forall k$ si $Deb_k \leq Fin_k$ alors $-1 \leq Fin_{k+1} - Deb_{k+1} < Fin_k - Deb_k$.
  Après la ligne 1 on a $Deb_k \leq Mil_{k+1} \leq Fin_k$.
  A la ligne 2 on a 2 cas :
- Cas 1 : $T[Mil_{k+1}] > e$. $Deb_{k+1} = Deb_k, Fin_{k+1} = Mil_{k+1} - 1$.
  On a $Fin_{k+1} - Deb_{k+1} = Mil_{k+1} - Deb_k - 1$ et donc

$$\begin{aligned} Deb_k - Deb_k - 1 \leq Fin_{k+1} - Deb_{k+1} &\leq Fin_k - Deb_k - 1 \\ -1 \leq Fin_{k+1} - Deb_{k+1} &< Fin_k - Deb_k \end{aligned}$$

- Cas 2 : $T[Mil_{k+1}] \leq e$. $Deb_{k+1} = Mil_{k+1} + 1, Fin_{k+1} = Fin_k$.
  On a $Fin_{k+1} - Deb_{k+1} = Fin_k - (Mil_{k+1} + 1)$ et donc

$$\begin{aligned} Fin_k - (Fin_k + 1) \leq Fin_{k+1} - Deb_{k+1} &\leq Fin_k - (Deb_k + 1) \\ -1 \leq Fin_{k+1} - Deb_{k+1} &< Fin_k - Deb_k \end{aligned}$$

Arrêt des algorithmes récursifs

Trouver une expression qui décroît strictement à chaque appel récursif.

Algorithme : RechDicho2(d T ,d e, d Deb : Entier, d Fin : Entier) : Booléen
Données : T trié $\nearrow$, e
Résultat : renvoie Faux si $\forall i \in [Deb \ldots Fin]$ $T[i] \neq e$ Vrai sinon
Variables : $Mil \in \mathbb{N}$
début
  si $Deb > Fin$ **alors renvoyer** `Faux`
  sinon
    $Mil \leftarrow (Deb + Fin)$ *quo* 2 ; **si** $T[Mil] = e$ **alors renvoyer** `Vrai`
    sinon si $e < T[Mil]$ **alors**
      renvoyer `RechDicho2(`$T, e, Deb, Mil - 1$`)`
    sinon
      renvoyer `RechDicho2(`$T, e, Mil + 1, Fin$`)`
    fin si
  fin si
fin algorithme

La différence entre le quatrième et le troisième paramètre ($Fin - Deb$) décroît strictement à chaque appel récursif.

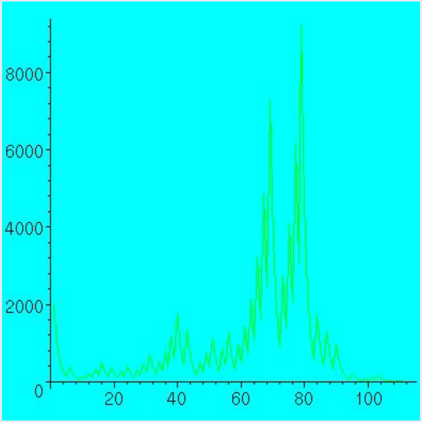
Arrêt de cet algorithme ?

Suite de Syracuse

Algorithme : f(d n :entier) : entier
Données : $n \in \mathbb{N}$
Résultat :
début

```
    si n = 1 alors renvoyer 1
    sinon si n est pair alors
        renvoyer f(n/2)
    sinon
        renvoyer f(3*n+1)
    fin si
```

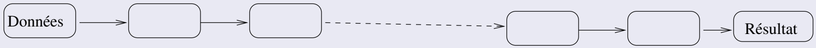
fin algorithme


8000
6000
4000
2000
0
20
40
60
80
100

Preuve du résultat d’un algorithme

Vérifier que le résultat correspond aux spécifications de l’algorithme

- L’algorithme décrit un enchaînement d’états à parcourir pour passer de l’état de départ correspondant aux données, à l’état d’arrivée correspondant au résultat.
- Etat : valeurs des variables (environnement)
- Pour montrer qu’un tel cheminement est correct, on décrit les états intermédiaires.
- Pour décrire ces états on donne des propriétés vérifiées par les variables (assertions).
- Problème avec les itérations



Invariants de boucle

Définition (Invariant)

Invariant d'une itération : propriété vérifiée à chaque itération par les valeurs de certaines variables.

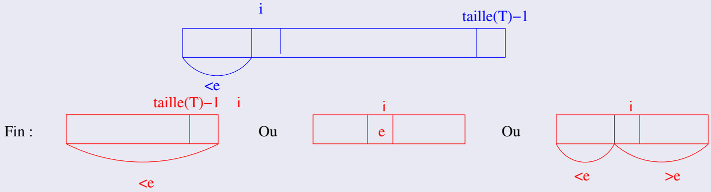
Les valeurs des variables peuvent changer à chaque itération, mais la propriété est « invariante ».
Intérêt : la propriété « invariante » est vérifée en fin d'itération

Exemple

Algorithme : Recherche(d T : tableau, d e) : Booléen
Données : `T` un tableau trié $\nearrow$, `e`
Résultat : Renvoie Vrai si `e` est élément de `T`, renvoie Faux sinon
variables : $i \in \mathbb{N}$;
début

$i \leftarrow 0$;
tant que $i < taille(T)$ `et` $T[i] < e$ **faire**
/* $\forall j < i, T[j] < e$ */
$i \leftarrow i + 1$
fin tq
/* $\forall j < i, T[j] < e$ `et` $(i \geqslant taille(T)$ `ou` $T[i] \geqslant e)$ */
renvoyer $(i < taille(T)$ `et` $T[i] = e)$

fin algorithme



Preuve d'Invariant

```
début
1     tant que Cond faire
2         Inst ;
3     fin tq
4
fin algorithme
```

Pour montrer que `Rel` est invariante pour une itération `Tant que` on utilise le schéma de preuve par récurrence sur le nombre d'itérations : il faut montrer que :

- `Rel` est vérifiée avant la première itération (ligne 1)
- si à la ligne 2 `Cond` et `Rel` sont vérifiées, alors `Rel` est vraie à la ligne 3.

Utilisation d'Invariant

```
début
  tant que Cond faire
    /* Rel                                    */
    Inst ;
  fin tq
4 /* non Cond et Rel                          */
fin algorithme
```

- Si `Rel` est une propriété invariante pour une itération
- alors en fin d'itération (ligne 4), `non Cond` et `Rel` sont vérifiées.

Que calcule algo1 ?

Exemple

Algorithme : algo1(d A,d B,r Z)
Données : $A, B \in \mathbb{N}$
Résultat : $Z = ?$

Variables : X et $Y \in \mathbb{N}$
début
    $X \leftarrow A$; $Y \leftarrow B$; $Z \leftarrow 0$;
    tant que $X \neq 0$ **faire**
        si X `est impair` **alors**
            $Z \leftarrow Z + Y$; $X \leftarrow X - 1$;
        fin si

        $Y \leftarrow 2 * Y$; $X \leftarrow X/2$
    fin tq
fin algorithme

Un exemple

$A = 6, B = 5.$

Num Itér	A	B	X	Y	Z
0	6	5	6	5	0
1	6	5	3	10	0
2	6	5	1	20	10
3	6	5	0	40	30

Résultat

Z=AxB

Invariant

$AxB = X \times Y + Z$

Preuve de l'invariant

Algorithme : Algo1(dA, dB, rZ)
Données : $A, B \in \mathbb{N}$
Résultat : $Z = A \times B$

Variables : X et $Y \in \mathbb{N}$
début
    $X \leftarrow A$; $Y \leftarrow B$; $Z \leftarrow 0$;
    tant que $X \neq 0$ **faire**
        si X `est impair`
         alors
            $Z \leftarrow Z + Y$
            $X \leftarrow X - 1$;
        fin si
        $Y \leftarrow 2 * Y$;
        $X \leftarrow X/2$;
    fin tq
fin algorithme

$\forall k \geq 0, A \times B = X_k \times Y_k + Z_k$

- k=0 ; $X_0 = A, Y_0 = B, Z_0 = 0$
- Hyp : $A \times B = X_k \times Y_k + Z_k$ et $X_k > 0$
  Cas 1 : X_k pair :
  $Z_{k+1} = Z_k, X_{k+1} = \frac{X_k}{2}, Y_{k+1} = 2.Y_k$

$$\begin{aligned} X_{k+1}.Y_{k+1} + Z_{k+1} &= \frac{X_k}{2}.2.Y_k + Z_k \\ &= X_k.Y_k + Z_k \\ &= A.B \end{aligned}$$

Preuve de l'invariant

Algorithme : Algo1(dA,dB,rZ)
Données : $A, B \in \mathbb{N}$
Résultat : $Z = A \times B$

Variables : X et $Y \in \mathbb{N}$
début
  $X \leftarrow A$; $Y \leftarrow B$; $Z \leftarrow 0$;
  tant que $X \neq 0$ **faire**
    si X `est impair`
     alors
      $Z \leftarrow Z + Y$
      $X \leftarrow X - 1$;
    fin si
    $Y \leftarrow 2 * Y$;
    $X \leftarrow X/2$;
  fin tq
fin algorithme

$\forall k \geq 0, A \times B = X_k \times Y_k + Z_k$

Hyp : $A \times B = X_k \times Y_k + Z_k$ et $X_k > 0$

Cas 2 : X_k impair

$Z_{k+1} = Z_k + Y_k, X_{k+1} = \frac{X_k - 1}{2}, Y_{k+1} = 2.Y_k$

$$
\begin{aligned}
X_{k+1}.Y_{k+1} + Z_{k+1} &= \frac{X_k - 1}{2}.2.Y_k + Z_k + Y_k \\
&= X_k.Y_k + Z_k \\
&= A.B
\end{aligned}
$$

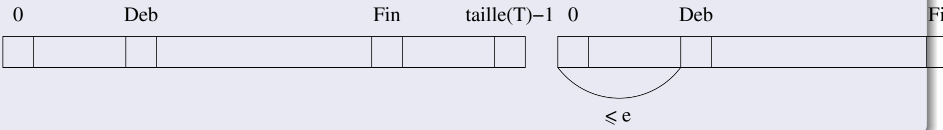
En fin d'itération

$A \times B = X \times Y + Z$ et $X = 0$

Donc $Z = A \times B$

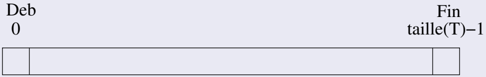
Invariant pour la recherche dichotomique

Algorithme : rechDicho(d T : tableau ,d e,r present : Bool, r ind : entier)
Données : T tableau trié $\nearrow$, e
Résultat : Si $e \notin T$ alors $present = Faux$ sinon $present = Vrai$ et ind est l'indice maximum de e dans T ($ind = max\{i \in [0..taille(T)[\ :\ T[i] = e\}$)
Variables : $Deb,\ Fin,\ Mil \in \mathbb{N}$
début
  $Deb \leftarrow 0$; $Fin \leftarrow taille(T) - 1$;
  tant que $Deb \leq Fin$ **faire**
    $Mil \leftarrow (Deb + Fin)\ quo\ 2$;
    si $T[Mil] \leq e$ **alors** $Deb \leftarrow Mil + 1$
    sinon $Fin \leftarrow Mil - 1$
  fin tq
  si . . . **alors** $present \leftarrow Vrai$; $ind \leftarrow ...$
  sinon $present \leftarrow Faux$
fin algorithme

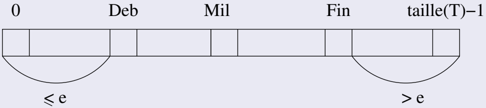


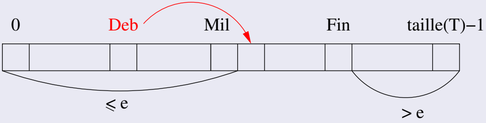
Preuve de l'invariant

- La propriété est vérifiée avant la première itération

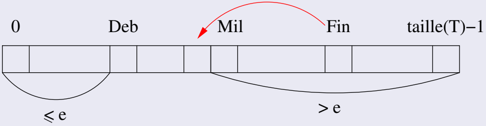


- Supposons la propriété vérifiée à l'itération k





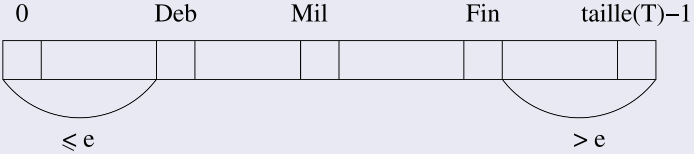
- Cas 1 $T[Mil] \leq e$



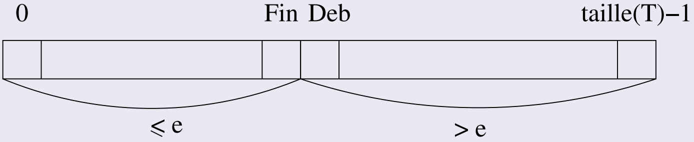
- Cas 2 $T[Mil] > e$

En fin d'itération

- Invariant



- Fin de l'itération
  $Deb > Fin$ ($Deb = Fin + 1$ d'après la preuve d'arrêt)
- Donc



Si e est dans T alors

  - $T[Fin] = e$
  - Fin **est le plus grand indice de** e **dans le tableau** T

Algorithme de recherche dichotomique complet

Algorithme : rechDicho(d T : tableau ,d e,r present : Bool, r ind : entier)
Données : T tableau trié $\nearrow$, e
Résultat : Si $e \notin T$ alors $present = Faux$, sinon $present = Vrai$ et ind est l'indice maximum de e dans T
Variables : $Deb,\ Fin,\ Mil \in \mathbb{N}$

début

- $Deb \leftarrow 0$; $Fin \leftarrow taille(T) - 1$;
- **tant que** $Deb \leq Fin$ **faire**
  - $Mil \leftarrow (Deb + Fin)\ quo\ 2$;
  - **si** $T[Mil] \leq e$ **alors** $Deb \leftarrow Mil + 1$
  - **sinon**
    - $Fin \leftarrow Mil - 1$
  - **fin si**
- **fin tq**
- **si** $Fin \geqslant 0$ `et` $T[Fin] = e$ **alors** $present \leftarrow Vrai$; $ind \leftarrow Fin$
- **sinon**
  - $present \leftarrow Faux$
- **fin si**

fin algorithme

Invariant pour les itérations `Pour`

Même principe. Pour régler le problème des *instructions cachées*, on traduit l'itération `Pour` en itération `Tantque`.

Exemple

Algorithme : rechercheSequentielle(d T : tableau , d e, r present : Bool)
Données : T tableau, e
Résultat : $present = Vrai$ si $e \in T$, $present = Faux$ sinon
Variable $i \in \mathbb{N}$
début
  $present \leftarrow Faux$;
  pour i `de 0 à` $taille(T) - 1$ **faire**
    si $T[i] = e$ **alors** $present \leftarrow Vrai$
  fin pour
fin algorithme

Version Pour

Données : T, e
Résultat : $present = (e \in T)$
Variable $i \in \mathbb{N}$
début
  $present \leftarrow Faux$;
  pour i `de 0 à taille(T)-1` **faire**
    /* $present = \exists j < i, T[j] = e$ */
    si $T[i] = e$ **alors**
      $present \leftarrow Vrai$
    fin si
  fin pour
fin algorithme

Version Tant que

Données : T, e
Résultat : $present = (e \in T)$
Variable $i \in \mathbb{N}$
début
  $present \leftarrow Faux$; $i \leftarrow 0$;
  tant que $i \leqslant taille(T) - 1$ **faire**
    /* $present = \exists j < i, T[j] = e$ */
    si $T[i] = e$ **alors**
      $present \leftarrow Vrai$
    fin si
    $i \leftarrow i + 1$;
  fin tq
fin algorithme

Invariant

$present = Vrai$ si et seulement si il existe $j \in [0..i[$ tel que $T[j] = e$

Complexité d’un algorithme

Objectif

- Définir des critères pour la comparaison d’algorithmes ou pour évaluer la qualité d’un algorithme.
- Critères : temps, mémoire utilisés pour le calcul décrit par l’algorithme.
- Évaluation du temps de calcul d’un algorithme : **nombre d’opérations élémentaires** exécutées par l’algorithme en fonction de la **taille de la donnée** et **dans le plus mauvais des cas**.

opération élémentaire

Définition

Opération élémentaire : opération dont le temps d'exécution est borné par une constante (indépendant de la donnée).

Exemples

- Affectation de variables simples
- Accés à un élément de tableau
- Opérations booléennes, comparaisons
- Opérations arithmétiques (lorsque opérandes et résultats sont bornés)

Contre-Exemples

- Initialisation, comparaison, affectation de tableau
- Algorithme
- Arithmétique sur grand nombre, exponentiation

taille de la donnée

Définition

Taille de la donnée : taille du codage (nombre de mots mémoire) de l'ensemble des paramètres-donnée.

Simplification

- pour un booléen, caractère, nombre borné : 1
- pour un tableau ou un ensemble : nombre d'éléments (si la valeur des éléments est bornée)
- pour un arbre : nombre de noeuds et hauteur
- (pour un graphe : nombre de sommets et nombre d'arêtes)
- (pour un problème de « nature numérique » (ex : test de primalité) : taille du codage des nombres)

Compter le nombre d'opérations élémentaires

Algorithme : SomTab(**d** T) : Entier
Données : *T* tableau d'entiers
Résultat : Renvoie la somme des éléments de *T*
Variables : $i, S \in \mathbb{N}$
début

```
    i ← 0 ; S ← 0 ;
    tant que i< taille(T) faire
        S ← S + T[i] ; i ← i + 1 ;
    fin tq
    renvoyer S
```

fin algorithme

- Taille du problème : taille(T) noté *N*
- Nombre d'itérations : *N*
-

renvoyer	1
accès tableau	N
additions	2.N
affectations	2+2.N
comparaisons	N+1
en tout	6.N+4

L'algorithme `SomTab` exécute 6.N+4 opérations élémentaires pour une donnée de taille N.

Types d'analyse

Pour une même taille de donnée le nombre d'opérations élémentaires exécutées par 1 algorithme peut varier selon les données ; on peut effectuer plusieurs types d'analyse :

- dans le plus mauvais des cas : on compte $t^{A}_{max}(n)$, le nombre maximum d'opérations élémentaires exécutées par l'algorithme `A` pour les données de taille n.
- **en moyenne** : $t^{A}_{moy}(n)$: moyenne du nombre d'opérations élémentaires exécutées par l'algorithme `A` pour toutes les données de taille n

Ensemble (incomplet) de règles (imprécises) de calcul

Calcul de $T^A_{max}(n)$:

- Si `A` est une affectation de type simple : $X \leftarrow E$
  $T^A_{max}(n) = 1 + T^E_{max}(n)$ où $T^E_{max}(n)$ est le nb max d'opérations élémentaires pour évaluer l'expression E pour un problème de taille n
- Si `A` est un appel à l'algoritme **B**
  $T^A_{max}(n) = T^B_{max}(m)$ où m est la taille de la donnée de l'algorithme `B`
- Si `A` est une séquence d'instuctions : `I ; J;`
  $T^A_{max}(n) = T^I_{max}(n) + T^J_{max}(n)$
- Si `A` est une conditionnelle : `Si C alors I sinon J fin Si`
  $T^A_{max}(n) = T^C_{max}(n) + Max(T^I_{max}(n) + T^J_{max}(n))$
- Si `A` est une itération : `Tant que C Faire I Fin Tq`
  $T^A_{max}(n) = T^C_{max}(n) + \sum_{k=1}^{maxIter} T^C_{max}(n) + T^{I_k}_{max}(n)$ où `maxIter` est le nombre maximum d'itérations et I_k est l'instruction à l'itération k.
- Si `A` est une itération : `pour v de E1 à E2 Faire I Fin Pour`
  $T^A_{max}(n) = T^{E_1}_{max}(n) +^{E_2}_{max}(n) + \sum_{k=1}^{maxIter} 2 + T^{I_k}_{max}(n)$ où `maxIter` est le nombre maximum d'itérations et I_k est l'instruction à l'itération k.

Exemple

Algorithme : rechSeq(**d** T, **d** e) : Bool
Données : T tableau trié $\nearrow$ et e
Résultat : Renvoie Vrai ssi $e \in T$
Variables : $i \in \mathbb{N}$

```
début
    i ← 0 ;
    tant que i < taille(T) et T[i] < e faire
        i ← i + 1 ;
    fin tq
    si i ⩾ taille(T) ou T[i] > e alors
        renvoyer Faux
    sinon
        renvoyer Vrai
    fin si
fin algorithme
```

- Taille du problème : $N + 1 (N = \mathit{taille}(T))$
- Meilleur cas : $T[0] \geqslant e$

affectations	1
additions	0
accès tableau	2
comparaisons	4
opérations bool.	2
renvoyer	1
en tout	10

- Pire cas : $T[N-1] < e$
  N itérations

affectations	$N+1$
additions	N
accès tableau	N
comparaisons	$2N+2$
opérations bool.	$N+2$
renvoyer	1
$t^{RS}_{max}(N+1) =$	$6N+6$
$t^{RS}_{max}(N) =$	$6N$

Ordre de grandeur

Rôle des constantes

Dans l'expression de $t_{max}(n)$ les constantes ont :

- peu de sens : par exemple l'exécution d'une opération booléenne est plus rapide que l'accès à un élément de tableau.
- peu d'importance : Si pour un même problème 2 algorithmes `A` et `B` ont les complexités $t^A_{max}(n) = n^2 + 1$ et $t^B_{max}(n) = 4.n + 3$, on préfèrera l'algorithme `B` même si $t^A_{max}(n) < t^B_{max}(n)$ pour $n < 5$.

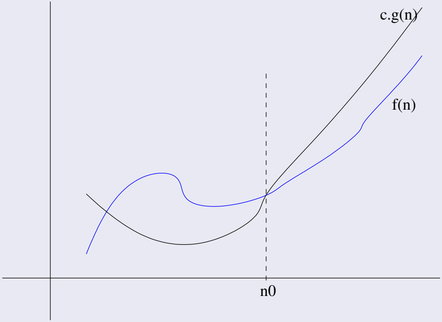
Ce qui nous intéresse

Comportement asymptotique de $t_{max}(n)$.

Ordre de grandeur

Définition

2 fonctions $f, g : \mathbb{R} \to \mathbb{R}$
$f(n) \in O(g(n))$ si $\exists c > 0, \exists n_0$ tels que $\forall n > n_0, f(n) \leqslant c.g(n)$
f est dominé asymptotiquement par g.
On note également $f = O(g)$ ou encore $f(n) = O(g(n))$.



Exemple

- $6n + 6 \in O(n)$. Noté aussi $6n + 6 = O(n)$. (par exemple $c = 12$ et $n_0 = 1$)
- $(n + 1)^2 = O(n^2)$ (par exemple $c = 3, n_0 = 1$)
- $4n^3 + 10n^2 + 8 = O(n^3)$.

$t_{max}^{RS}(N) = 6N = O(N)$

Propriétés

$O(f).O(g) = O(f.g)$
$O(f)+O(g) = O(f+g) = O(max(f,g))$

Dans le pire des cas :

ligne 1	O(1)
N itérations	
Une itération	O(1)
Conditionnelle	O(1)
	O(1+N.1+1)=O(N)

Exemple

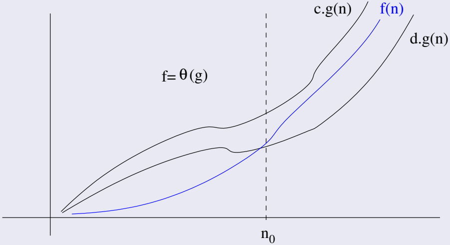
début
  $i \leftarrow 0$;
  tant que $i < N$ `et` $T[i] < e$ **faire**
    $i \leftarrow i + 1$;
  fin tq
  si $i \geqslant N$ `ou` $T[i] > e$ **alors**
    renvoyer Faux
  sinon
    renvoyer Vrai
  fin si
fin algorithme

Notation θ

On a $6n = O(n)$ mais aussi $6n = O(n^2)$ car $O(n) \subset O(n^2)$.

Définition

2 fonctions $f, g : \mathbb{R} \to \mathbb{R}$
$f(n) \in \theta(g(n))$ si $\exists c, d \in \mathbb{R}^*, \exists n_0$ tels que $\forall n > n_0, d.g(n) \leq f(n) \leq c.g(n)$
On note également $f = \theta(g)$.



$t_{max}^{RS}(N) = 6N = \theta(N)$

Principaux ordres de grandeurs

- $\theta(1)$: constant
- $\theta(log_2 n)$: logarithmique
- $\theta(n)$: linéaire
- $\theta(nlog_2 n)$
- $\theta(n^2)$: polynomial
- $\theta(n^3)$
- $\theta(2^n)$: exponentiel

Des chiffres

Exemple

On exécute des algorithmes de complexité différentes sur une machine 1 Ghz, exécutant une opération élémentaire en $1\eta s$ ($10^{-9}s$) :

	$log_2 n$	n	$n.log_2 n$	n^2	n^3	2^n
10^2	$6\eta s$	$100\eta s$	$700\eta s$	$10\mu s$	$1ms$	$40 \times 10^{12} a$
10^3	$10\eta s$	$1\mu s$	$10\mu s$	$1ms$	$1s$	
10^4	$13\eta s$	$10\mu s$	$0,1ms$	$0,1s$	$16mn$	
10^5	$16\eta s$	$100\mu s$	$1ms$	$10s$	$11j$	
10^6	$20\eta s$	$1ms$	$19ms$	$16mn$	$31a$	
10^7	$23\eta s$	$10ms$	$0,3s$	$27h$		

Rappel : l'age de l'univers est estimé à 14 milliards d'années (10^{10} a.).

Algorithme : rechDicho(**d** T,**d** e) : Bool
Données : T tableau trié $\nearrow$, e
Résultat : Renvoie $e \in T$
Variables : $Deb, \ Fin, \ Mil \in \mathbb{N}$
début

```
    Deb ← 0 ; Fin ← taille(T) − 1 ;
    tant que Deb ≤ Fin faire
        Mil ← (Deb + Fin) quo 2 ;
        si T[Mil] = e alors renvoyer Vrai
        sinon si T[Mil] ≤ e alors
            Deb ← Mil + 1
        sinon
            Fin ← Mil − 1
        fin si
    fin tq
    renvoyer Faux
```

fin algorithme

Preuve

- Arrêt :
  $Fin - Deb \geq -1$ et décroît strictement à chaque itération.
- Invariant :
  $\forall i \in [0 \ldots Deb[$
  $\forall j \in]Fin \ldots taille(T)[$
  $T[i] < e < T[j]$

```
début
    Deb ← 0 ; Fin ← taille(T) − 1 ;
    tant que Deb ≤ Fin faire
        Mil ← (Deb + Fin) quo 2 ;
        si T[Mil] = e alors
            renvoyer Vrai
        sinon si T[Mil] ≤ e alors
            Deb ← Mil + 1
        sinon
            Fin ← Mil − 1
        fin si
    fin tq
    renvoyer Faux
fin algorithme
```

Analyse de la complexité

- taille du problème : $N + 1$ ($N = \mathit{taille}(T)$)
- pire des cas : $e \notin T$
- complexité d'une itération : $O(1)$
- la complexité de l'algo est O(nombre d'itérations).

Calcul du nombre maximum d'itérations

- À chaque itération $Fin - Deb$ est divisé par 2. Soit Nbe le nombre d'éléments entre les indices Deb et Fin.
- $Nbe_0 = N$
- $Nbe_1 \leq \frac{N}{2}$
- $Nbe_i \leq \frac{Nbe_{i-1}}{2} \leq \frac{Nbe_{i-2}}{2^2} \leq \frac{N}{2^i}$

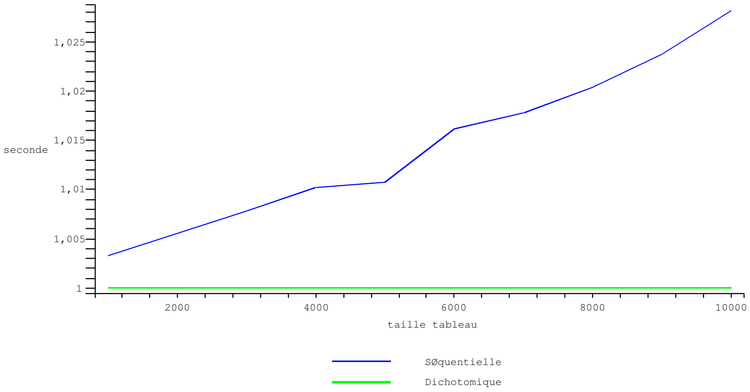
  Soit m le nombre d'itérations :
- $Nbe_{m-1} \geq 1$
- $\frac{N}{2^{m-1}} \geq 1$
- $m \leq log_2(N) + 1$

  Complexité dans le pire des cas $t_{max}^{RD}(N+1) \in O(log_2 N)$.
  $t_{max}^{RD}(N) \in O(log_2 N)$

Améliorer un algorithme : changer sa classe de complexité !

Ex : Recherche d'un élément dans un tableau trié

Moyenne pour 100 tirages aléatoires



Exemple : multiplication de deux entiers par additions

Mesure de la complexité

On compte le nombre d'additions en fonction de la valeur des opérandes

Algorithme par additions successives

Algorithme : Mult1(**d** A, **d** B, **r** Z)
Données : $A, B \in \mathbb{N}$
Résultat : $Z = A \times B$
Variables : $X \in \mathbb{N}$
début
  $Z \leftarrow 0$;
  pour X `de 1 à` A **faire**
    $Z \leftarrow Z + B$;
  fin pour
fin algorithme

Calcul du nombre d'additions

- A chaque itération une addition est exécutée
- La complexité est donnée par le nombre d'itérations.
- L'algorithme exécute A itérations
- Le nombre total d'additions est A.

Exemple : multiplication de deux entiers par additions

Multiplication « Russe »

Algorithme : mult2(**d** A, **d** B, **r** Z)
Données : $A, B \in \mathbb{N}$
Résultat : $Z = A \times B$
Variables : X et $Y \in \mathbb{N}$
début
  $X \leftarrow A$; $Y \leftarrow B$; $Z \leftarrow 0$;
  tant que $X \neq 0$ **faire**
    si X `est impair` **alors**
      $Z \leftarrow Z + Y$; $X \leftarrow X - 1$
    fin si
    $X \leftarrow X/2$; $Y \leftarrow Y + Y$;
  fin tq
fin algorithme

Calcul du nombre d'additions

- 2 additions au plus par itération
- à chaque itération k la valeur de X est divisée par 2 :
  $X_k \leqslant \frac{X_{k-1}}{2} \leqslant \frac{X_{k-2}}{4} \leqslant \frac{A}{2^k}$
- donc le nombre d'itérations i vérifie
  $X_i = 0, 1 = X_{i-1} \leqslant \frac{A}{2^{i-1}}$
- le nombre d'additions exécutées est au plus $2 \times (log_2(A) + 1)$
- pire des cas : X est toujours impair :
  $A = 2^n - 1$
  ex : $X_0 = 31, X_1 = 15, X_2 = 7,$
  $X_3 = 3, X_4 = 1, X_5 = 0$

Comparaison des temps pour le calcul de $(2^n - 1) \times 99$

Limites de l'analyse

- en prenant l'ordre de grandeur, on néglige les constantes qui peuvent être grandes !
  On peut faire une analyse plus fine.
- le plus mauvais des cas peut être très peu fréquent ; pour certains problèmes les algorithmes utilisés en pratique ne sont pas ceux de plus petite complexité dans le pire des cas.
  On peut faire une analyse en moyenne, mais il faut connaître la distribution des données

Tous ces cas sont rares.
L'analyse de la complexité asymptotique dans le pire des cas est une bonne mesure.

Type de Données Abstrait et Structures de Données

Représentation des données nécessaires à la résolution d'un problème

Définition d'un Type de Données Abstrait

Spécification des opérations permettant de manipuler les objets du type.
Exemple : opérations du TDA `Ens` :

appartient ? :	Objet x Ens	$\longrightarrow$	Booléen
ajouter :	Objet x Ens	$\longrightarrow$	Ens

...

$\forall o \in Objet, E \in Ens, appartient?(o, ajouter(o, E)) = Vrai$

...

Implantation d'un Type de Données Abstrait

Description de l'organisation des données et algorithmes réalisant les opérations du TDA (vérifiant ses spécifications) : **Structures de Données**
Exemple : plusieurs implantations possibles du TDA *Ens* : tableau, tableau de booléen, liste chaînée, arbre, table de Hachage, ...

Le type de Données Abstrait Pile

Opérations

pileVide ?(**d** P :Pile) :Bool ;
Données : P une Pile
Résultat : Renvoie un booléen indiquant si le Pile P est vide

créerPile() :Pile ;
Données :
Résultat : Renvoie une Pile vide

sommetPile(**d** P :pile) :X ;
Données : P une Pile non vide
Résultat : Renvoie l'élément sommet de pile

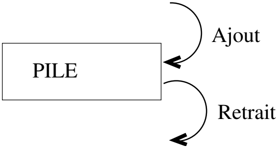
empiler(**dr** P :Pile, **d** e :X) ;
Données : une pile P, e
Résultat : ajoute e à la pile P

dépiler(**dr** P :Pile) ;
Données : une pile P non vide
Résultat : supprime le sommet de P

Le type de Données Abstrait Pile

Propriété

L'élément renvoyé par `sommetPile(P)` est le dernier élément empilé.
Gestion LIFO («Last In First Out»).



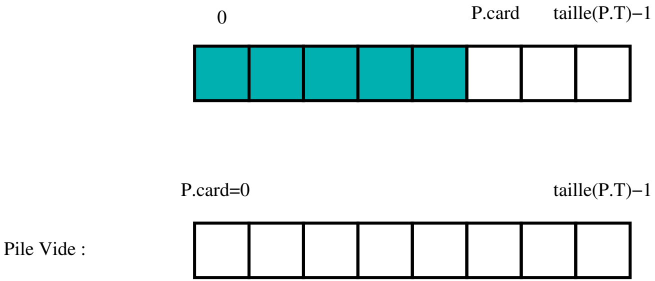
Exemple

P← créerPile() ; empiler(P,1) ; empiler(P,2) ; empiler(P,3) ;
A ← sommetPile(P) ; dépiler(P) ;
`/* A=3 */`
B ← sommetPile(P) ; dépiler(P) ;
`/* B=2 */`
C ← sommetPile(P) ; dépiler(P) ;
`/* C=1 */`

Implantation du type pile par un Tableau

Une Pile *P* est représentée par un couple :

- un tableau, noté `P.T`
- nombre d'éléments de la pile, noté `P.card`



Implantation du type pile par un Tableau

Algorithme : pileVide ?(**d** P) :Bool ;
début
  renvoyer `(P.card=0);`
fin algorithme

Algorithme : créerPile() :Pile ;
début
  P.card $\leftarrow$ 0 ; **renvoyer** *P* ;
fin algorithme

Algorithme : sommetPile(**d** P) :X ;
début
  renvoyer `P.T[P.card-1];`
fin algorithme

Opérations en $\theta(1)$

Algorithme :
empiler(**dr** P : Pile, **d** e : X)
début
  si `P.card=taille(P.T)` **alors**
    Débordement de Pile
  sinon
    P.T[P.card]$\leftarrow$ e ;
    P.card$\leftarrow$P.card+1 ;
fin algorithme

Algorithme : dépiler(**dr** P : Pile)
début
  P.card$\leftarrow$P.card-1 ;
fin algorithme

Le Type de Données Abstrait File

Opérations

fileVide ?(**d** F : File) : Bool ;
Données : F une File
Résultat : Renvoie un booléen indiquant si la File est vide

créerFile() :File ;
Données :
Résultat : Renvoie une File vide

têteFile(**d** *F* : File) :X ;
Données : *F* une File non vide
Résultat : Renvoie l'élément en tête de File

ajouterFile(**dr** *F* : File,**d** *e* : X) ;
Données : une File *F*, *e*
Résultat : ajoute *e* à la File *F*

retirerFile(**dr** *F* : File) ;
Données : une File *F* non vide
Résultat : supprime la tête de la File *F*

Le Type de Données Abstrait File

Propriété

L'élément renvoyé par `têteFile(F)` est le premier élément ajouté à la file F.
Gestion FIFO (« First In First Out»).

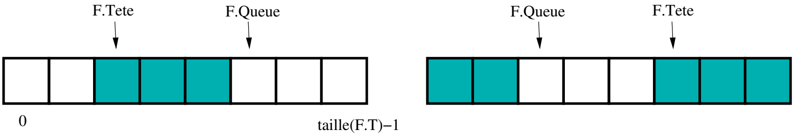
Exemple

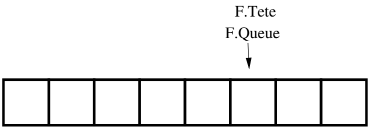
F← créerFile() ; ajouterFile(F,1) ; ajouterFile(F,2) ; ajouterFile(F,3) ;
A ← têteFile(F) ; retirerFile(F) ;
`/* A=1 */`
B ← têteFile(F) ; retirerFile(F) ;
`/* B=2 */`
C ← têteFile(F) ; retirerFile(F) ;
`/* C=3 */`

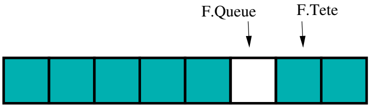
Implantation du type File par un Tableau

Une File *F* est représentée par un triplet :

- un Tableau, noté `F.T`
- l'indice de l'élément Tête de File, noté `F.Tête`
- l'indice suivant le dernier élément ajouté à la File, noté `F.Queue`



File Vide : 

File Pleine : 

Algorithme : fileVide ?(**d** F : File) :Bool ;
début
| **renvoyer** `(F.Tête=F.Queue);`
fin algorithme

Algorithme : créerFile() : File ;
début
| F.Tête←0 ;F.Queue←0 ;
| **renvoyer** `F` ;
fin algorithme

Algorithme : têteFile(**d** F : File) : X ;
début
| **renvoyer** `F.T[F.Tête]`
fin algorithme

Algorithme : retirerFile(**dr** F : File)
début
| F.Tête ← (F.Tête +1)mod taille(F.T) ;
fin algorithme

Algorithme : FilePleine ?(**d** F :File) :Bool ;
début
| **renvoyer** `((F.Queue+1) mod taille(F.T))=F.Tête`
fin algorithme

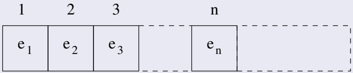
Algorithme : ajouterFile(**dr** F : File, **d** e : X)
début
| **si** `(FilePleine?(F)` **alors** Débordement de File
| **sinon** F.T[F.Queue]← e ; F.Queue←(F.Queue+1) mod taille(F.T)
fin algorithme

Opérations en $\theta(1)$

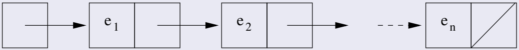
Listes

Une liste est une séquence d'éléments. $(e_1, e_2, \ldots, e_n)$.
Chaque élément de la liste a une place dans la liste. Les éléments de la liste peuvent être rangés de manière

- contigüe : l'élément suivant un élément de place p est à la place $p+1$
  Tableau, vecteur



- chaînée : la place de l'élément suivant est mémorisée avec l'élément
  Liste chaînée



Remarque

- Le type Liste vu en GLIN101 : représentation chaînée
- Liste en CAML et SCHEME : repésentation chaînée
- Liste en MAPLE, PYTHON : représentation contigüe

Définition récursive des Listes Chaînées

Une liste est :

- soit la liste vide
- soit un couple (`Premier élément,Suite de la liste`)
  (tête de liste, queue de liste)

« Le type Liste Chaînée »

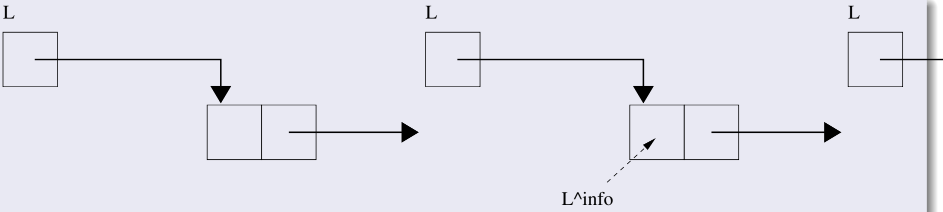
Pour manipuler une liste chaînée, nous avons besoin d'opérations pour :

- tester si une liste est vide
- accéder aux informations d'une liste non vide :
  - valeur du premier élément de la liste *(tête de liste)*
  - suite de la liste *(queue de liste)*
- construire une liste
- **modifier les informations d'une liste** *(nouveau)*

Une structure de Données pour les Listes Chaînées

- Chaque élément d'une liste sera représenté par une cellule double
- Une cellule est connue par son adresse.
- Une Liste est l'adresse de la cellule associée à son premier élément
- La liste vide est représentée par l'adresse `NULL`

Opérations sur les Listes Chaînées



L étant une variable de type liste non vide (pointeur dont la valeur est l'adresse de la cellule associée à son premier élément) :

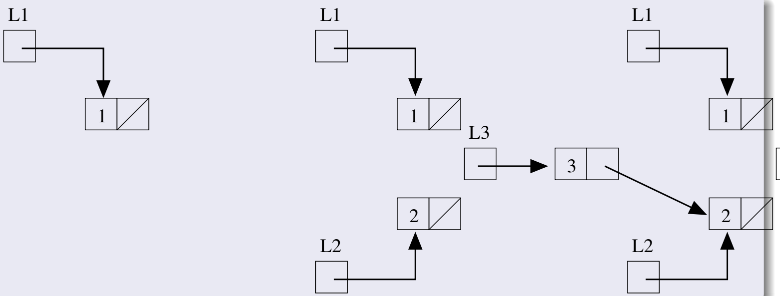
- `L↑info` désigne la valeur du premier élément de la liste (*tête de liste*)
- `L↑succ` désigne la suite de la liste (*queue de liste*)

La fonction `créerListe` permet de construire une nouvelle liste

Algorithme : créerListe(**d** e : élément, **d** SL : Liste) : Liste
Données : e, SL
Résultat : Renvoie la liste dont le premier élément est `e` et la suite de la liste est `SL`

Exemple de manipulation de liste chaînée

- L1← créerListe(1,NULL) ;
- L2← créerListe(2,NULL) ;
- L3← créerListe(3,L2) ;
- L3↑info ← L2↑info ;
- L3↑succ ← L1 ;
- L2↑succ ← L3↑succ ;



Exemple d'algorithme sur les listes chaînées

Algorithme : longListe(**d** L :Liste) : entier
Données : `L` une liste
Résultat : Renvoie le nombre d'éléments de `L`

Variables P :Liste, nbElem : entier
début
    P $\leftarrow$ L ; nbElem$\leftarrow$ 0 ; **tant que** `P` $\neq$ `NULL` **faire**
        nbElem$\leftarrow$nbElem +1 ; P$\leftarrow$P$\uparrow$succ ;
    fin tq
    renvoyer `nbElem`
fin algorithme

Version récursive

Algorithme : longListe(**d** L :Liste) : entier
Données : `L` une liste
Résultat : Renvoie le nombre d'éléments de `L`

début
  si `L=NULL` **alors**
    renvoyer `0`
  sinon
    renvoyer `1+longListe(L↑succ)`
  fin si
fin algorithme

Version affreusement fausse et trop souvent rencontrée !

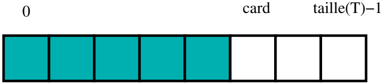
Variable cpt : entier
début
  cpt $\leftarrow$ 0
  si `L=NULL` **alors**
    renvoyer `cpt`
  sinon
    cpt $\leftarrow$ cpt+1
    longListe(L↑succ)
  fin si
fin algorithme

Opérations sur les Listes Chaînées

Opérations sur les listes

1. Recherche de la place d'un élément dans une liste
2. Insertion d'un élément dans une liste à une place déterminée
3. Suppression d'un élément d'une place donnée d'une liste

Si liste implantée par tableau,



Les complexités de ces opérations sont, en fonction du nombre n d'éléments :

1. $\theta(n)$
2. $\theta(n)$
3. $\theta(n)$

Recherche

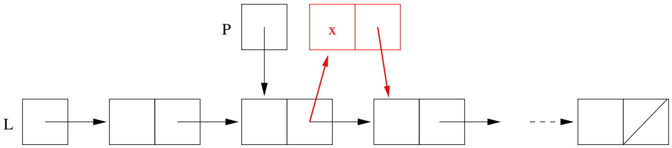
Algorithme : recherche(**d** L : Liste, **d** x : X) :Liste
Données : *L* est une Liste Chaînée ; *x*
Résultat : renvoie NULL si $x \notin L$; sinon renvoie l'adresse de la première cellule de *L* contenant *x*

Variables : P : Liste

début
    P←L
    tant que `P≠ NULL et (P↑info)≠ x` **faire**
        P← P↑ succ ;
    fin tq
    renvoyer `P`
fin algorithme

début
    si `L=NULL ou L↑info=x` **alors**
        renvoyer `L`
    sinon
        renvoyer `recherche(L↑succ,x)`
    fin si
fin algorithme

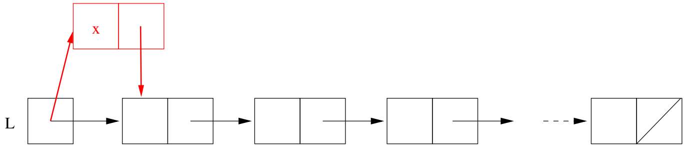
$\theta(n)$



Insertion

Algorithme : insérerAprès(**dr** L :Liste,**d** P :Liste, **d** x : X)
Données : *L* est une Liste non vide, *P* un pointeur vers une cellule de *L*, *x*
Résultat : Insère dans la liste *L* une cellule contenant *x* après celle pointée par *P*
Variable Q : Liste
début
| P↑succ ← creerListe(x,P↑succ) ;
fin algorithme

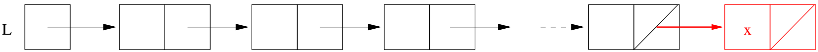
$\theta(1)$



Insertion

Algorithme : insérerDébut(**dr** L :Liste, **d** x : X)
Données : *L* est une Liste chaînée
Résultat : Insère au début de *L* une cellule contenant *x*
début
| L ← créerListe(x,L)
fin algorithme

$\theta(1)$



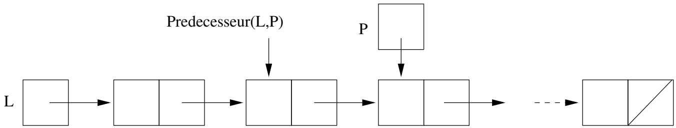
Insertion

Algorithme : insérerFin(**dr** L :Liste, **d** x : X)
Données : *L* est une Liste chaînée
Résultat : Insère en fin de la Liste *L* une cellule contenant *x*

Variable P : Liste
début
  si L=NULL **alors**
    L← créerListe(x,NULL)
  sinon
    P←L
    tant que P↑succ≠NULL **faire**
      P ← P↑succ ;
    fin tq
    P↑succ← créerListe(x,NULL)
  fin si
fin algorithme

début
  si L=NULL **alors**
    L← créerListe(x,NULL)
  sinon
    insérerFin(L↑succ,x)
  fin si
fin algorithme

$\theta(n)$



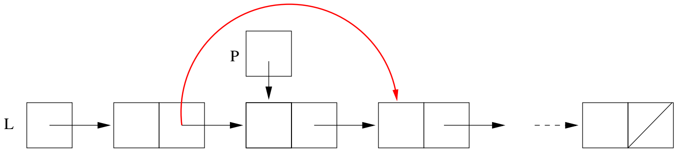
Prédécesseur

Algorithme : predecesseur(**d** L : Liste,**d** P : Liste) :Liste
Données : L est une Liste non vide ;P pointeur vers un élément de L, $L \neq P$
Résultat : renvoie l'adresse de la cellule précédant dans L celle repérée par P

Variables : Q :Liste
début
  $Q \leftarrow L$
  tant que $(Q \uparrow succ) \neq P$ **faire**
    $Q \leftarrow Q \uparrow succ$;
  fin tq
  renvoyer `Q`
fin algorithme

début
  si `(L↑succ)=P` **alors**
    renvoyer `L`
  sinon
    renvoyer `predecesseur(L↑succ,P)`
  fin si
fin algorithme

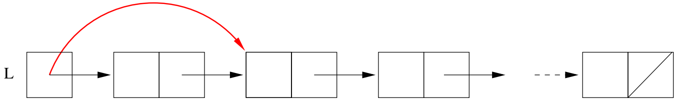
$\theta(n)$



Suppression

Algorithme : supprimer(**dr** L : Liste, **d** P : Liste)
Données : *L* est une Liste non vide ;*P* un pointeur vers une cellule de *L*
Résultat : Modifie L en supprimant de *L* la cellule pointée par *P*
début
  si `L=P` **alors**
    L← L↑succ
  sinon
    predecesseur(L,P)↑succ← P↑succ
  fin si
fin algorithme

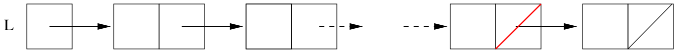
$\theta(n)$



Suppression

Algorithme : supprimerDébut(**dr** L : Liste)
Données : *L* est une Liste non vide
Résultat : Supprime le premier élément de *L*
début
| L← L↑succ
fin algorithme

$\theta(1)$



Suppression

Algorithme : supprimerFin(**dr** L : Liste)
Données : *L* est une Liste non vide
Résultat : Supprime le dernier élément de *L*
Variable P

```
début
  si L↑succ=NULL alors
  |  L←NULL
  sinon
  |  P←L
  |  tant que
  |   (P↑succ)↑succ≠NULL faire
  |   |  P← P↑succ ;
  |  fin tq
  |  P↑succ←NULL ;
  fin si
fin algorithme
```

```
début
  si L↑succ=NULL alors
  |  L←NULL
  sinon si (L↑succ)↑succ=NULL
   alors
  |  L↑succ←NULL
  sinon
  |  supprimerFin(L↑succ)
fin algorithme
```

$\theta(n)$

Exemple de représentation des listes en C

Définition du type liste et de la fonction `créerListe`

- typedef struct cellule {
  int info ;
  struct cellule *succ ;} CelluleSC ;
  typedef CelluleSC *ListeSC ;
- ListeSC creerLSC(int val, ListeSC succ){
  ListeSC li = (ListeSC) malloc(sizeof(CelluleSC)) ;
  li -> info = val ;
  li -> succ = succ ;
  return li ;}

version récursive de la fonction `insérerFin`

```
void insererFinLSC( ListeSC *p, int val ){
     if ((*p)==NULL) (*p)=creerLSC(val,NULL) ;
     else insererFinLSC(&((*p)->succ),val) ;
     return ; }
```

Implantation de Pile et File par Liste Chaînée

Pile

On obtient les propriétés d'une Pile :

- empiler : insérer en début de liste
- dépiler : supprimer en début de Liste

Toutes les opérations en $\theta(1)$

File

On obtient les propriétés d'une File :

- ajouterFile : insérer en fin de liste
- retirerFile : supprimer en début de Liste

Toutes les opérations en $\theta(1)$ sauf l'ajout qui est en $\theta(n)$.

Autres Listes

Défaut des Listes Simplement Chaînées

Les opérations coûteuses sur les listes chaînées :

- Calul de la cellule précédant une cellule (opération de suppression)
- Calcul de la dernière cellule (opérations d'insertion et de suppression en fin de liste)

Pour améliorer les complexités on peut mémoriser ces informations.

Listes Doublement Chaînées

Mémorisation du prédécesseur

Chaque élément d'une Liste Doublement Chaînée est représenté par une cellule triple. Chaque cellule d'adresse P contient contient 3 informations :

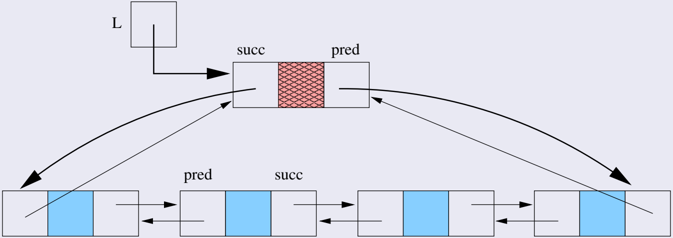
- la valeur de l'élément : $P\uparrow$info
- l'adresse de la cellule suivante : $P\uparrow$succ
- l'adresse de la cellule précédente : $P\uparrow$pred

Mémorisation du dernier

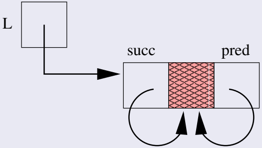
Une liste est représentée par 2 pointeurs contenant les adresses de la première et dernière cellule de la liste. Ces 2 pointeurs sont mémorisés dans une cellule de la liste. Une Liste Doublement Chaînée (LDC) L est un pointeur vers une cellule :

- $L\uparrow$info n'est pas utilisé
- $L\uparrow$succ est l'adresse de la cellule associée au premier élément
- $L\uparrow$pred est l'adresse de la cellule associée au dernier élément

Liste Doublement Chaînée de 4 éléments



Liste Doublement Chaînée Vide



Recherche

Algorithme : recherche(**d** L :LDC,**d** x : X) :adrCellule
Données : *L* est une Liste Doublement Chaînée *x*
Résultat : renvoie NULL si $x \notin L$; sinon renvoie l'adresse de la première cellule de *L* contenant *x*

Variables : P : adrCellule ; **début**

```
    P←L↑succ ; tant que P≠ L et (P↑ info)≠ x faire
    |   P← P↑ succ ;
    fin tq
    si P=L alors
    |   renvoyer NULL
    sinon
    |   renvoyer P
    fin si
```

fin algorithme

$\theta(n)$

Algorithme : créerTriplet(**d** LPred : adrCellule, **d** x : X, **d** LSucc : adrCellule) : adrCellule
Données : *LPred* et *LSucc* 2 adresses de cellule ; *x* une valeur
Résultat : Renvoie l'adresse d'une nouvelle cellule triple, dont l'information est *x*, l'adresse de la cellule précèdente est *LPred*, l'adresse de la cellule suivante est *LSucc*.

Insertion

Algorithme : insérerAprès(**d** L :LDC,**d** P :adrCellule, **d** x : X)
Données : *L* est une Liste Doublement Chaînée non vide, *P* un pointeur vers une cellule de *L*, *x*
Résultat : Insère dans la liste *L* une cellule contenant *x* après celle pointée par *P*
Variable Q : adrCellule ; **début**
| Q ← créerTriplet(P,x,P↑succ) ; (P↑succ)↑ pred ← Q ; P↑succ ← Q ;
fin algorithme

$\theta(1)$

Insertion en début de liste

Algorithme : insérerDébut(**d** L :LDC, **d** x : X)
Données : *L* est une Liste Doublement Chaînée
Résultat : Insère au début de la liste *L* une cellule contenant *x*
début
| insérerAprès(L,L,x)
fin algorithme

$\theta(1)$

Insertion en fin de liste

Algorithme : insérerFin(**d** L :LDC, **d** x : X)
Données : *L* est une Liste Doublement Chaînée
Résultat : Insère en fin de la liste *L* une cellule contenant *x*
début
| insérerAprès(L,L↑pred,x)
fin algorithme

$\theta(1)$

Suppression

Algorithme : supprimer(**d** L : LDC, **d** P : adrCellule)
Données : *L* est une Liste Doublement Chaînée non vide ;*P* un pointeur vers une cellule de *L* ($P \neq L$)
Résultat : Supprime de *L* la cellule pointée par *P*
début
  (P↑pred)↑succ←P↑succ ;
  (P↑succ)↑pred←P↑pred ;
fin algorithme

$\theta(1)$

Suppression en début de liste

Algorithme : supprimerDébut(**d** L :LDC)
Données : *L* est une Liste Doublement Chaînée non vide
Résultat : Supprime le premier élément de *L*
début
| supprimer(L,L↑succ)
fin algorithme

$\theta(1)$

Suppression en fin de liste

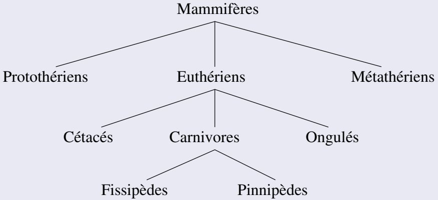
Algorithme : supprimerFin(**d** L :LDC)
Données : *L* est une Liste Doublement Chaînée non vide
Résultat : Supprime le dernier élément de la liste *L*
début
| supprimer(L,L↑pred)
fin algorithme

$\theta(1)$

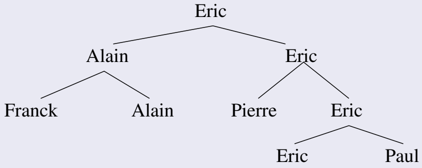
Les Arbres (Arborescences)

Structure permettant de représenter un ensemble d'objets muni d'un ordre.

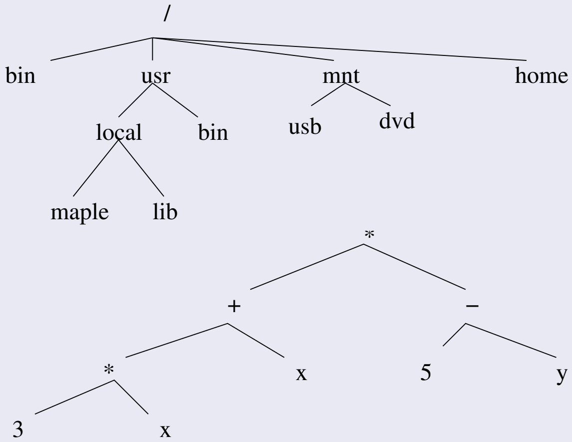
Classification des mammifères :



Arbre tournoi :



Arbres en informatique

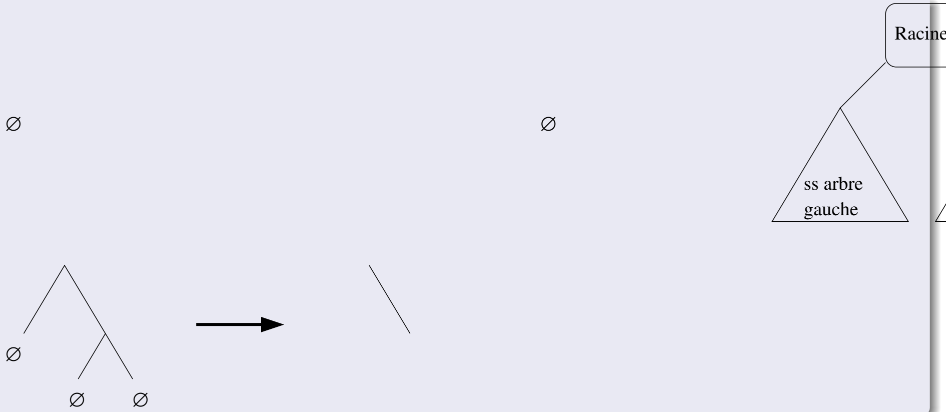


$$((3 * x) + x) * (5 - y)$$

Définition

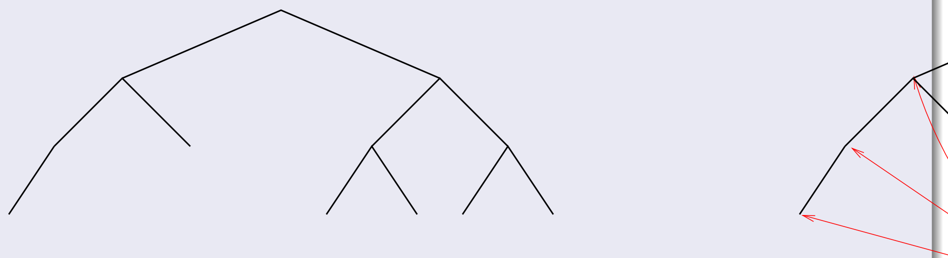
Un arbre binaire est

- soit l'arbre vide
- soit un triplet composé d'une racine et de 2 arbres binaires, appelés sous-arbre gauche et sous-arbre droit.



Définition

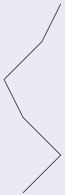
- Noeuds d'un arbre non vide = sa racine + les noeuds de ses 2 sous-arbres.
- Arbre étiqueté : à chaque noeud est associée une information (étiquette).
- Feuille : Arbre non vide dont les 2 sous-arbres sont vides.
- Profondeur d'un noeud d'un arbre :
  - profondeur de la racine = 0
  - profondeur d'un noeud = profondeur de son père + 1
- Hauteur d'un arbre : profondeur maximum de ses noeuds



Arbres particuliers

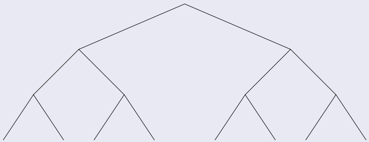
Arbres Filiformes

Arbres n'ayant qu'une feuille.



Arbres Complets

Toutes les feuilles ont même profondeur et seules les feuilles ont un sous-arbre vide.



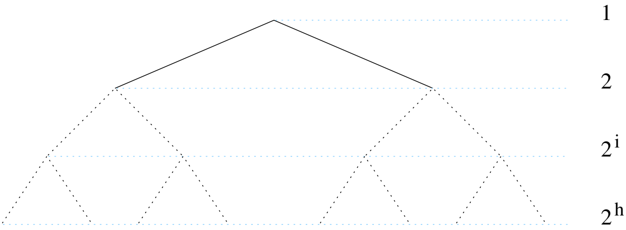
Relation entre hauteur et nombre de noeuds d'un arbre binaire

La hauteur h d'un arbre binaire possédant n noeuds vérifie :

$$\lceil log(n+1) \rceil - 1 \leq h \leq n - 1$$

- Arbre filiforme $h = n - 1$
- Arbre Complet

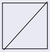
$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

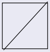


Structure de Données Arbre Binaire

Un arbre A est :

- soit l'arbre vide représenté par le pointeur `NULL`
- soit un arbre non vide (e, A_1, A_2) représenté par un pointeur vers une cellule contenant :
  - $A \uparrow$ *info* : étiquette de la racine de l'arbre (e)
  - $A \uparrow$ *sag* : sous-arbre gauche (A_1)
  - $A \uparrow$ *sad* : sous-arbre droit (A_2)

A 

A 

Constructeur d'arbre

Algorithme : créerArbre(**d** x : X, **d** A1 : ArbreBin, **d** A2 : ArbreBin) : ArbreBin
Résultat : Renvoie l'arbre dont la racine a pour étiquette x, le sous–arbre gauche est $A1$ et le sous–arbre droit est $A2$

Exemple d'algorithme sur les arbres

Exemple

Algorithme : nbNoeudsArbre(**d** A : ArbreBin) : Entier
Données : *A* est un arbre binaire
Résultat : Renvoie le nombre de noeuds de l'arbre *A*

```
début
    si A=Null alors
        renvoyer 0
    sinon
        renvoyer 1+nbNoeudsArbre(A↑sag)+nbNoeudsArbre(A↑sad)
    fin si
fin algorithme
```

Parcours d'un Arbre Binaire

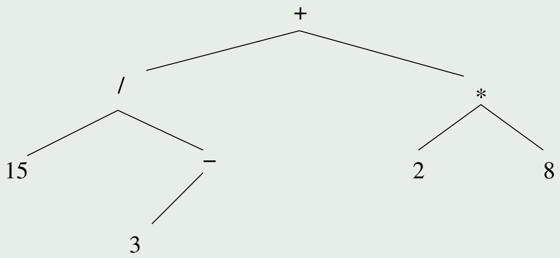
Ordres d'exploration des noeuds d'un Arbre Binaire

Algorithme : Prefixe(A)
début
 si $A \neq NULL$ **alors**
  Traiter(A↑info) ;
  Prefixe(A↑sag) ;
  Prefixe(A↑sad) ;
 fin si
fin algorithme

Algorithme : Infixe(A)
début
 si $A \neq NULL$ **alors**
  Infixe(A↑sag) ;
  Traiter(A↑info) ;
  Infixe(A↑sad) ;
 fin si
fin algorithme

Algorithme : Suffixe(A)
début
 si $A \neq NULL$ **alors**
  Suffixe(A↑sag) ;
  Suffixe(A↑sad) ;
  Traiter(A↑info) ;
 fin si
fin algorithme

Exemple

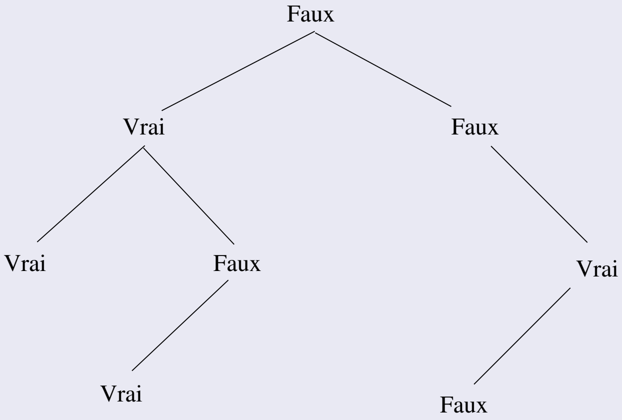


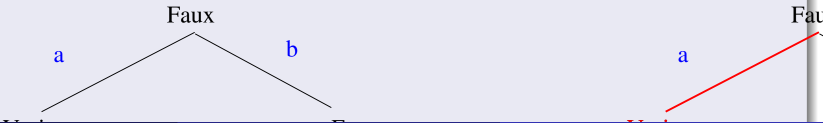
- Le parcours dans l'ordre préfixe est : + / 15 - 3 * 2 8
- Le parcours dans l'ordre infixe est : 15 / 3 - + 2 * 8
- Le parcours dans l'ordre suffixe est : 15 3 - / 2 8 * +
- Parcours en largeur (par profondeur croissante) : + / * 15 - 2 8 3

Ex d'arbres binaires :arbre préfixe, dictionnaire, tries

Soit un alphabet à 2 lettres a et b.
On peut représenter un ensemble fini de mots construits avec ces lettres par un arbre binaire étiqueté par des booléens.
L'ensemble de mots représenté par un arbre A, noté $Dico(A)$, est défini par :

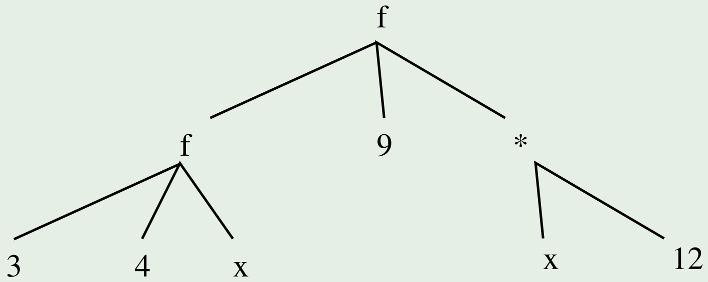
- $Dico(NULL) = \emptyset$
- si $A \neq NULL$ et $A \uparrow info = Faux$,
  $Dico(A) = \{a.m | m \in Dico(A \uparrow sag)\} \cup \{b.m | m \in Dico(A \uparrow sad)\}$
- si $A \neq NULL$ et $A \uparrow info = Vrai$,
  $Dico(A) = \{\epsilon\} \cup \{a.m | m \in Dico(A \uparrow sag)\} \cup \{b.m | m \in Dico(A \uparrow sad)\}$


Faux
Vrai
Faux
Vrai
Faux
Vrai
Vrai
Faux


Faux
a
b
a

Arbres Généraux

Exemple



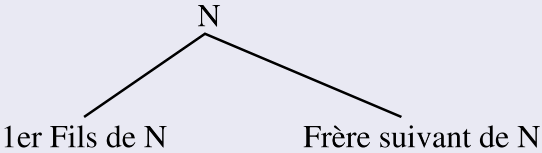
$f(f(3,4,x),9,X*12)$

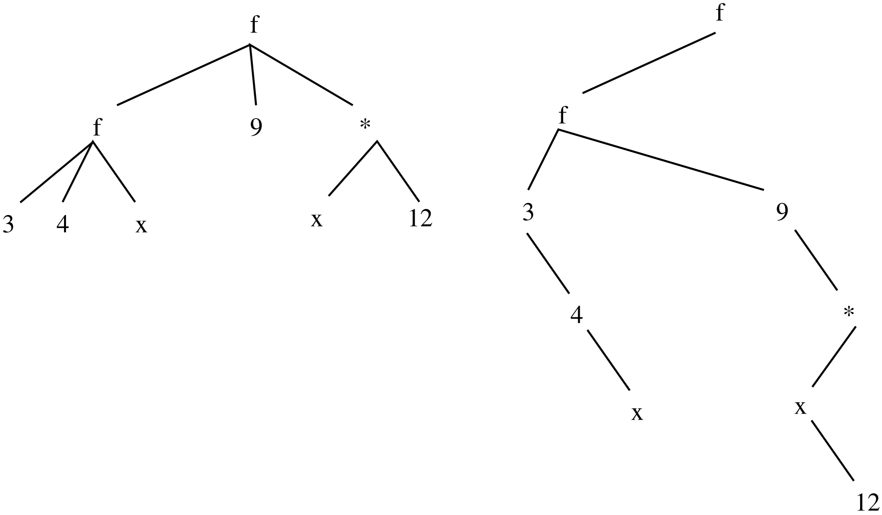
Arbres Généraux

Généralisation des arbres binaires : le nombre de sous-arbres peut être supérieur à 2.

Représentation des arbres N-aires

- Extension de la Structure de Données
- Utilisation des arbres binaires

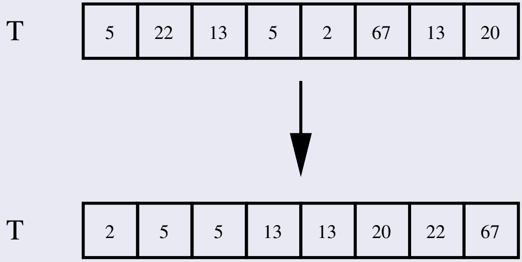



f
f
9
*
3
4
x
x
12
f
f
3
9
4
*
x
x
12

Problème de Tri

Algorithme : TriTableau(**dr** T : tableau)
Données : T tableau
Résultat : T est modifié :

- T contient les mêmes élements qu'en entrée,
- T est croissant ($\forall 0 \leq i < j < taille(T), T[i] \leq T[j]$)

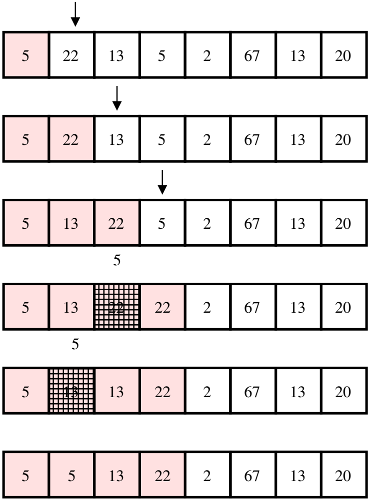


Beaucoup de problèmes sur les tableaux peuvent être résolus plus efficacement si les tableaux sont triés (Recherche d'un élément).

Spécifications du problème

- On étudie le problème du tri d'un tableau d'entiers mais les algorithmes sont applicables à d'autres types munis d'un ordre total
- Les valeurs à trier constituent une partie des *objets* à trier, leur clé
- Plusieurs éléments du tableau peuvent avoir la même valeur
- Pas d'hypothèse sur le domaine des valeurs à trier (domaine infini)
- Les algorithmes étudiés opèrent par comparaisons d'éléments (d'autres techniques existent si hypothèses sur le domaine (voir TD))
- Pour évaluer la complexité des algorithmes on compte le nombre de comparaisons entre éléments du tableau effectuées ; (on peut également compter le nombre d'échanges de place)
- La taille du problème est le nombre d'éléments du tableau

Tri par insertion



Algorithme

Algorithme : TriInsertion(**dr** T : tableau d'entiers)
Variables : $i, j, x \in \mathbb{N}$; **début**

```
pour i de 1 à taille(T)-1 faire
    x ← T[i] ; /* Insertion de x dans le sous-tableau trié
        T[0..i-1]                                          */
    j ← i − 1 ; tant que j ≥ 0 et T[j] > x faire
        T[j + 1] ← T[j] ; j ← j − 1 ;
    fin tq
    T[j + 1] ← x ;
fin pour
```

fin algorithme

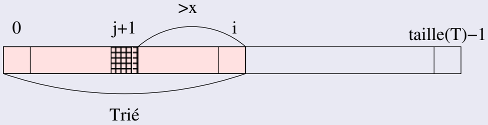
Preuve

Arrêt

Évident car pour le `Tant que` $j+1 \in \mathbb{N}$ et décroît strictement à chaque itération.

Invariants

- pour le `Pour` : $T[0..i-1]$ contient les i premiers éléments de T triés ;
- pour le `Tant que` :
  - $T[0], \ldots, T[j], T[j+2], \ldots, T[i]$ sont les i premiers éléments de T triés.
  - $T[j+2], \ldots; T[i]$ sont supérieurs strictement à x



Complexité.

```
début
  pour i de 1 à N-1 faire
    x ← T[i] ; j ← ... ; tant que ... et T[j] > x faire
      | ... ;
    fin tq
    ... ;
  fin pour
fin algorithme
```

- on note N= taille(T)
- la seule comparaison est dans la condition du `Tant que`.
- Meilleur des cas : tableau trié $\nearrow$; au total $N-1$ comparaisons
- Pire des cas : tableau trié $\searrow$;
  - Pour i fixé le nombre de comparaisons est i
  - Au total : $\sum_{i=1}^{N-1} i = \frac{N.(N-1)}{2} = O(N^2)$

Tri par Tas (HeapSort)

Principe

Utiliser une structure de données efficace.
Le **Tas** (tas binaire ou file de priorité) permet de représenter un ensemble muni d'une relation d'ordre optimisant les opérations suivantes :

- **CréerTas() :** renvoie un tas vide
- **InsérerTas(d e : entier, dr t : tas) :** insère l'élément `e` dans le tas `t`
- **ExtraireMax(dr t : tas, r e : entier) :** en résultat *e* est l'élément max du tas `t` ; de plus*e* est supprimé de `t`

Tri par Tas (HeapSort)

L'algorithme

Algorithme : TriParTas(**dr** $T[0..N-1]$: Tableau)
Variables : tas : Tas binaire ; **début**

```
    tas ←CreerTas() ;; /* Remplir le tas */
    pour i de 0 à N − 1 faire
      | InsérerTas(T[i],tas)
    fin pour
    ; /* Vider le tas */
    pour i de N − 1 bas 0 par pas de −1 faire
      | ExtraireMax(tas,T[i])
    fin pour
    ;
```

fin algorithme

Tri par Tas (HeapSort)

Complexité

- Nous allons voir que
  - L'opération `CréerTas` peut être réalisée en temps $\theta(1)$
  - Les opérations `InsérerTas(e,t)` et `ExtraireMax(t,e)` ont une complexité dans le pire des cas en $O(log_2 n)$ (n = nombre d'éléments du tas t)
- La complexité du tri par Tas est alors

$$\sum_{i=1}^{N} O(log_2(i)) + \sum_{i=1}^{N} O(log_2(i)) = O(N.log_2(N))$$

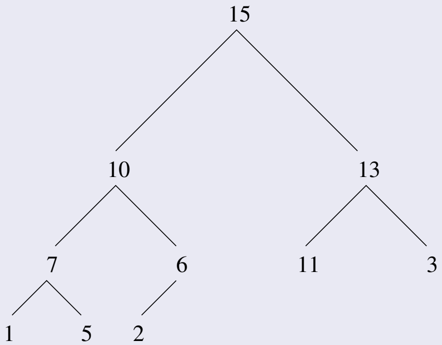
Réalisation d'un Tas

Définition

Un **Tas** est un arbre binaire vérifiant les 2 conditions suivantes :

- Condition sur les valeurs **CV** :
  tout noeud v de l'arbre, autre que la racine, vérifie $info(v) \leq info(pere(v))$
- Condition sur la forme de l'arbre **CF** :
  l'arbre est parfait
  - h étant la hauteur de l'arbre binaire, tous les niveaux de profondeur $p = 0, 1, \ldots, h - 1$ sont complets : nombre de noeuds de profondeur p = 2^p
  - les feuilles de profondeur h sont regroupées *à gauche*

Exemple



Propriété

La hauteur d'un tas contenant n éléments est $\lfloor log(n) \rfloor$

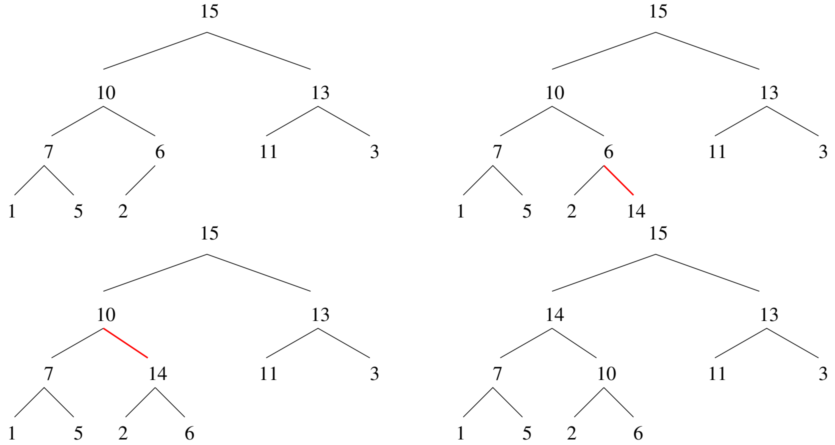
Opérations sur les arbres utilisées pour les fonctions du Tas

- *CréerArbre()* : renvoie l'arbre vide
- *Racine(**d** A)* : renvoie le noeud racine de l'arbre *A*.
- *Père(**d** N)* : renvoie le noeud père du noeud *N*.
- *Filsd(**d** N)* : renvoie le noeud fils droit du noeud *N*.
- *Filsg(**d** N)* : renvoie le noeud fils gauche du noeud *N*.
- *Info(**d** N)* : renvoie l'information contenue dans le noeud *N*.
- *Feuille ?(**d** N)* : renvoie vrai ssi *N* est une feuille.
- *DernièreFeuille(**d** A)* : renvoie la feuille la plus à droite du dernier niveau de l'arbre *A*.
- *CréerFeuille(**dr** A, **d** e, **r** N)* : modifie l'arbre *A* en créant après la dernière feuille une nouvelle feuille *N* de contenu *e*.
- *SupprimerFeuille(**dr** A)* : Supprime la dernière feuille de l'arbre *A*.
- *EchangerContenu(**dr** N1,**dr** N2)* : échange les contenus des noeuds *N*1 et *N*2.
- *FilsMax(**d** N)* : Renvoie parmi les 2 fils du noeud *N* celui dont le contenu est le plus grand.

InsérerTas :Ex : ajout de 14

Principe

Satisfaire la condition sur la forme d'un tas (**CF**) puis échanger les contenus des noeuds pour satisfaire la condition sur les valeurs (**CV**).



Algorithme : InsérerTas(**d** e : entier,**dr** t : tas)
Données : *t* un tas et *e* en entier
Résultat : *t* est un tas contenant les éléments du tas initial plus *e*
Variables : *q* un noeud ; **début**

$CreerFeuille(t, e, q)$; **tant que** $q \neq racine(t)$ `et` $info(pere(q)) < info(q)$ **faire**

$EchangerContenu(q, pere(q))$; $q \leftarrow pere(q)$;

fin tq

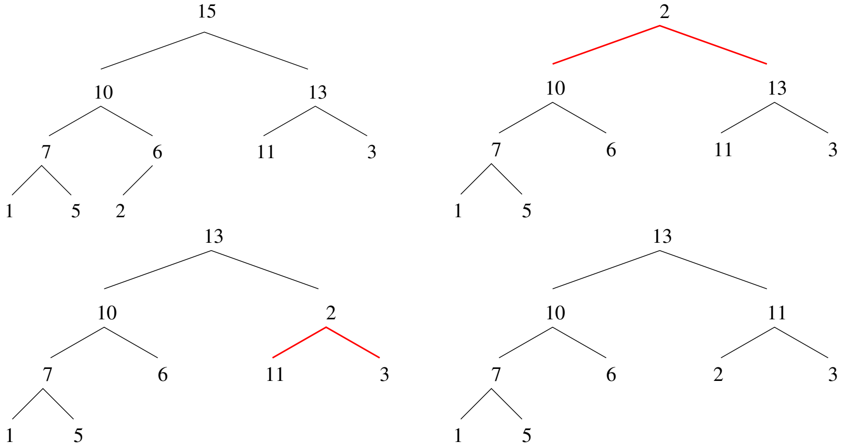
fin algorithme

Preuve

- **Arrêt :**
  la profondeur du noeud q décroît stritement à chaque itération.
- **Invariant :**
  la condition **CF** est satisfaite ; la condition **CV** est satisfaite sauf éventuellement au noeud q : pour tout noeud v autre que q et la racine $info(pere(v)) \geq info(v)$.
- **Complexité :**
  Le nombre de comparaisons est égal au nombre d'itérations qui, dans le pire des cas est la hauteur de l'arbre, d'où $O(log|t|)$.

Principe de l'algorithme

Comme pour l'insertion : satisfaire la condition **CF** puis échanger les contenus des noeuds pour satisfaire la condition **CV**.



Algorithme : ExtraireMax(**dr** t : tas, **r** max : entier)
Données : *t* un tas non vide
Résultat : *max* est le plus grand élément de *t* et *max* est supprimé de *t*.
Variables : q, f 2 noeuds ; **début**

```
max ← Info(racine(t)) ; EchangerContenu(racine(t), dernierefeuille(t)) ;
SupprimerFeuille(t) ; si nonarbrevide(t) alors
    q ← racine(t) ; tant que non(feuille(q)) et info(q) < info(filsMax(q)) faire
        f ← filsMax(q) ; EchangerContenu(q, f) ; q ← f ;
    fin tq
fin si
```

fin algorithme

Preuve et Invariant

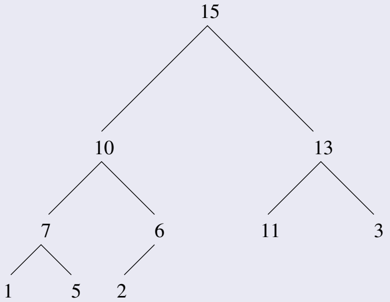
- **Arrêt :**
  la profondeur du noeud q croît strictement à chaque itération.
- **Invariant :**
  la condition **CF** est satisfaite ; la condition **CV** est satisfaite sauf éventuellement entre q et ses fils.
- **Complexité :**
  Le nombre de comparaisons est égal au nombre d'itérations qui, dans le pire des cas est la hauteur de l'arbre, d'où $O(log|t|)$.

Complexité en espace

Cet algorithme de Tri par Tas utilise une structure de Données supplémentaire (le Tas), sa complexité en place est $O(n)$.
En fait le Tas peut être représenté à l'aide du tableau que l'on trie (voir TD).
$\Rightarrow$ Pas besoin d'espace supplémentaire.

Représentation d’un Tas par un tableau

Parcours en largeur de l’arbre

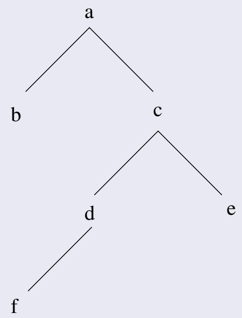


15	10	13	7	6	11	3	1	5	2

Cette représentation est compacte (taille du tableau = nombre de noeuds de l’arbre) et permet d’implanter chaque opération avec une complexité en O(1).

Codage pas toujours compact

Cette représentation par parcours en largeur est compacte pour les arbres parfaits . Ce n'est pas le cas pour tous les arbres :



a	b	c			d	e					f

« Diviser pour Résoudre »

Principe Général

Résoudre un problème :

- Décomposer le problème en sous-problèmes
- Résoudre indépendamment chaque sous-problème

Application au Tri

- Diviser le tableau en 2 sous–tableaux
- Trier chacun des 2 sous-tableaux

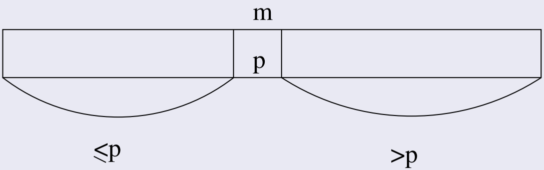
Ce principe général peut être appliqué de plusieurs façons :

- Tri Fusion
  1. Diviser le tableau en 2 sous–tableaux de taille identique
  2. Trier les 2 sous-tableaux
  3. Fusionner les 2 sous-tableaux triés

Tri Rapide (QuickSort)

Tri Rapide :

- Diviser le tableau en 2 sous–tableaux, de sorte que le tableau trié final s'obtienne directement à partir des 2 sous-tableaux triés.
  La division du tableau se fait par rapport à une valeur pivot :



Exemple

5	22	13	5	2	67	13	20

choix valeur Pivot = 13

5	22	13	5	2	67	13	20

division du tableau

5	13	2	5	13	67	22	20

tri du premier sous-tableau

2	5	5	13	13	67	22	20

tri du second sous-tableau

2	5	5	13	13	20	22	67

Algorithme : TriRapide(**dr**T : Tableau ,**d** g : indice,**d** d : indice)
Données : $T[g \dots d]$; $g \leq d+1$
Résultat : Permute les éléments du sous-tableau $T[g \dots d]$ pour que $T[g \dots d]$ soit trié
Variables : $m \in \mathbb{N}$; **début**

si $g < d$ **alors**

Pivot(T,g,d,m) ; TriRapide(T,g,m-1) ; TriRapide(T,m+1,d) ;

fin si

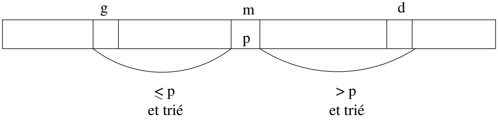
fin algorithme

Spécifications de l'algorithme `Pivot`

Algorithme : Pivot(**dr** T : tableau,**d** g : indice,**d** d : indice,**r** m : indice)
Données : $g < d$, $T[g \dots d]$
Résultat : en résultat $m \in [g \dots d]$ et $T[g \dots d]$ tels que et
$\forall i \in [g \dots m-1], T[i] \leq T[m], \forall i \in [m+1 \dots d], T[m] < T[i]$

Preuve du tri rapide

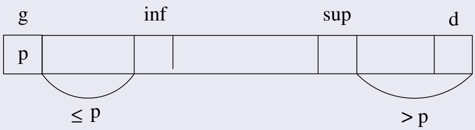
- Arrêt : la taille du sous-tableau à trier ($d - g + 1$)
  - est un entier naturel
  - décroît strictement à chaque appel récursif
- Preuve du résultat par induction sur la taille du problème .
  P(n) : TriRapide est correct pour tout tableau de taille n
  - $P(0), P(1)$ sont vérifiés car un tableau à 0 ou 1 élément est trié
  - soit $n > 1$ et supposons que $\forall n' < n, P(n')$
    soit $T[g \ldots d]$ un tableau de taille n. Après le calcul du pivot on a $g \leq m \leq d$, donc $T[g \ldots m-1]$ et $T[m+1 \ldots d]$ ont une taille inférieure strictement à n.
    Par hypothèse d'induction les 2 appels récursifs trient correctement
    $T[g \ldots m-1]$ et $T[m+1 \ldots d]$
    Donc après le deuxième appel récursif on a :



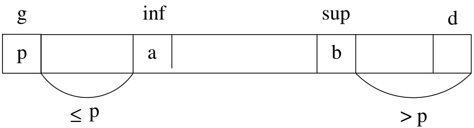
Donc $T[g \ldots d]$ est trié , donc $P(n)$ est vérifié

Calcul du Pivot

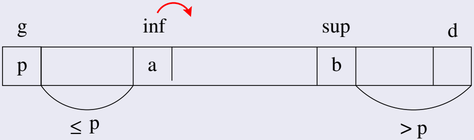
- Choix de la valeur pivot le premier élément : $p = T[g]$;
- On parcourt les éléments du tableau en permutant les éléments mal placés par rapport au pivot en maintenant l'invariant suivant :



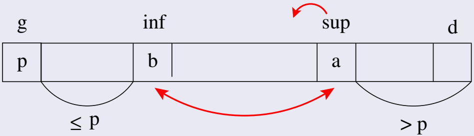
itération du calcul de pivot



- Si $T[inf] \leq p$ incrémenter *inf*

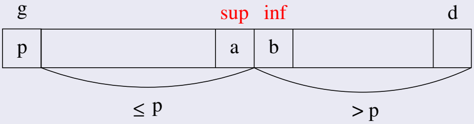


- Sinon échanger $T[inf]$ et $T[sup]$ et décrémenter *sup*

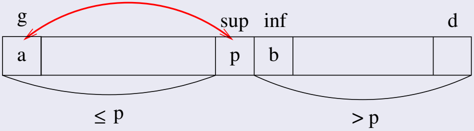


Arrêt

- On s'arrête quand $sup = inf - 1$



- On place la valeur pivot à l'indice `Sup`



L'algorithme

Algorithme : Pivot(**dr** T : tableau,**d** g : indice,**d** d : indice,**r** m : indice)
Données : $g < d$, $T[g \ldots d]$
Résultat : en résultat $m \in [g \ldots d]$ et $T[g \ldots d]$ tels que et
$\forall i \in [g \ldots m-1], T[i] \leq T[m], \forall i \in [m+1 \ldots d], T[m] < T[i]$

début
  $p \leftarrow T[g]$; $inf \leftarrow g+1$; $sup \leftarrow d$; **tant que** $inf \leq sup$ **faire**
    si $T[inf] \leq p$ **alors**
      $inf \leftarrow inf + 1$
    sinon
      $T[inf] \leftrightarrow T[sup]$;$sup \leftarrow sup - 1$
    fin si
  fin tq
  $T[g] \leftrightarrow T[sup]$;$m \leftarrow sup$;
fin algorithme

Complexité du Tri Rapide

Complexité de Pivot(T,g,d,m)

Le nombre de comparaisons effectuées est dans tous les cas exactement $d - g$, nombre d'éléments du sous–tableau `T[g..d]` - 1 .

Complexité de l'algorithme récursif Tri Rapide

Soit

- $t(n)$ le nombre de comparaisons effectuées par le tri rapide pour un tableau de n éléments
- n_1 la taille du premier sous-tableau $(m - g)$
- n_2 celle du second $(d - m)$.

  t vérifie la récurrence (avec $n_1 + n_2 = n - 1$) :

$$\begin{aligned} t(n) &= t(n_1) + t(n_2) + n - 1 \ si\ n > 1 \\ t(n) &= 0 \ si\ n \leq 1 \end{aligned}$$

Pire des cas

est atteint lorsque
l'un des sous–tableaux est vide : $n_1 = 0$ ou $n_2 = 0$.

$$\begin{aligned} t(n) &= t(n-1) + n - 1 \; si \; n > 1 \\ t(n) &= 0 \; si \; n \leq 1 \end{aligned}$$

$$t(n) = \sum_{i=1}^{n-1} i = n.(n-1)/2 = O(n^2)$$

Meilleur des cas

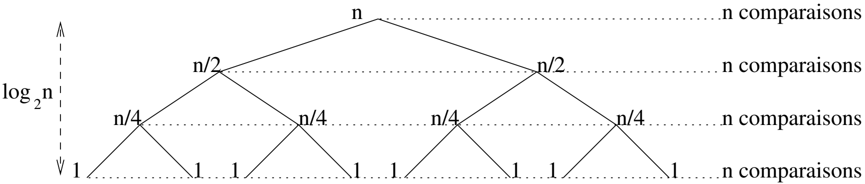
est atteint
lorsque les 2 sous-tableaux sont de même taille.

$$\begin{aligned} t(n) &= t(\lfloor n/2 \rfloor) + t(n - \lfloor n/2 \rfloor - 1) + n - 1 \; si \; n > 1 \\ t(n) &= 0 \; si \; n \leq 1 \end{aligned}$$

Pour simplifier les calculs on résout la récurrence suivante qui majore t

$$\begin{aligned} t'(n) &= 2.t'(n/2) + n \; si \; n > 1 \\ t'(n) &= 0 \; si \; n \leq 1 \end{aligned}$$

Intuition



$t'(n) = O(nlog_2(n))$

Preuve de $t'(n) \in O(n.log_2 n)$

$$
\begin{aligned}
t'(n) &= 2.t'(n/2) + n \; si \; n > 1 \\
t'(n) &= 0 \; si \; n \leq 1
\end{aligned}
$$

On montre que $\exists n_0, \exists c, \forall n > n_0, t'(n) \leq c.n.log_2(n)$ par récurrence sur n :

- $n = 1$ OK
- HR : $\forall n' < n, t'(n') \leq c.n'.log_2(n')$

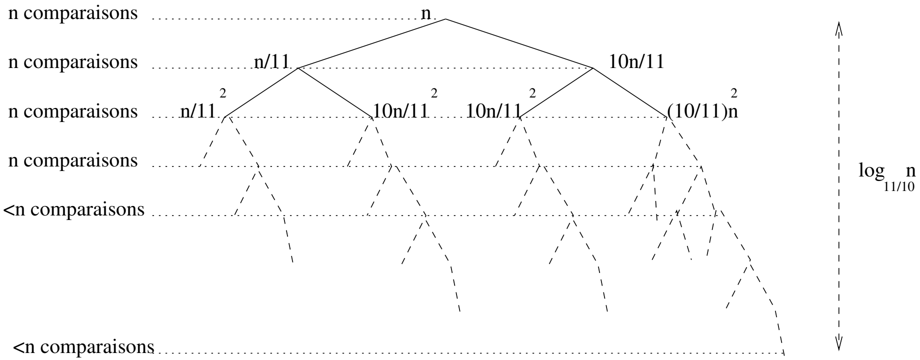
$$
\begin{aligned}
t'(n) &= 2.t'(n/2) + n \\
&\leq 2c.n/2.log_2(n/2) + n \\
&\leq c.n.(log_2(n) - 1) + n \\
&\leq c.n.log_2(n) + n(1 - c) \\
&\leq c.n.log_2(n)
\end{aligned}
$$

Vérifié si $c \geq 1$ (n_0 quelconque)

Tri Rapide ?

- La complexité dans le pire des cas du Tri rapide est pire que celle du Tri par Tas et identique à celle du Tri par insertion
- La complexité dans le meilleur des cas du tri rapide est pire que celle du Tri par insertion
- Mais le meilleur des cas est fréquent : même si à chaque étape le tableau n'est pas divisé en 2 sous-tableaux de même taille, la complexité peut être $O(n.ln(n))$.

Tri Rapide ?



- Par exemple si à chaque étape un sous-tableau est 10 fois plus grand que l'autre, la complexité reste $O(n.log_{\frac{11}{10}}(n)) = O(n.ln(n))$.
- on peut montrer que la complexité en moyenne est $t_{moy}(n) = O(n.ln(n))$

Conclusions Tri Rapide

Remarques

- limite de la complexité dans le pire des cas
- L'algorithme de calcul de Pivot ne minimise pas le nombre de déplacements d'éléments du tableau ; il peut être amélioré

Complexité minimum du problème de Tri par comparaisons

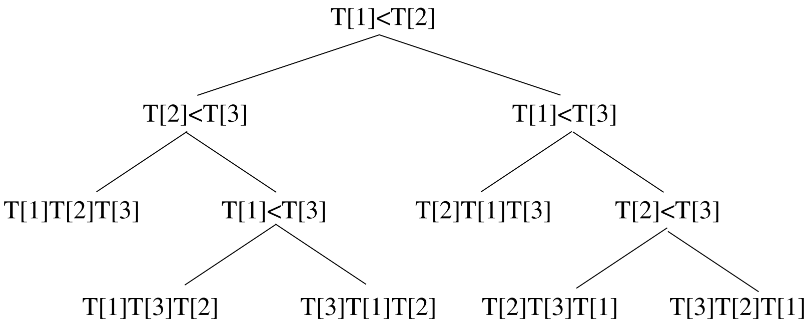
Peut-on trouver un algorithme de tri exécutant moins de O(n.ln(n)) comparaisons dans le pire des cas ?

- Un algorithme de tri par comparaisons exécute une séquence de comparaisons.
- La comparaison suivante dépend du résultat des comparaisons précédentes.
- On peut représenter l'ensemble des exécutions possibles d'un algorithme par un arbre binaire dont les étiquettes sont des comparaisons entre 2 éléments de tableau
- un noeud d'étiquette $T[i] < T[j]$ a pour sous–arbre gauche (respectivement droit) l'arbre représentant les comparaisons réalisées par l'algorithme lorsque $T[i] < T[j]$ (respectivement $T[i] \geqslant T[j]$)

Exemple

```
Algorithme : tri1(T[1..3])
début
    si T[1]<T[2] alors
        si T[2]<T[3] alors
            renvoyer T[1]T[2]T[3]
        sinon si T[1]<T[3] alors
            renvoyer T[1]T[3]T[2]
        sinon
            renvoyer T[3]T[1]T[2]
        fin si
    sinon si T[1]<T[3] alors
        renvoyer T[2]T[1]T[3]
    sinon si T[2]<T[3] alors
        renvoyer T[2]T[3]T[1]
    sinon
        renvoyer T[3]T[2]T[1]
    fin si
fin algorithme
```

Exemple


T[1]<T[2]
T[2]<T[3]
T[1]<T[3]
T[1]T[2]T[3]
T[1]<T[3]
T[2]T[1]T[3]
T[2]<T[3]
T[1]T[3]T[2]
T[3]T[1]T[2]
T[2]T[3]T[1]
T[3]T[2]T[1]

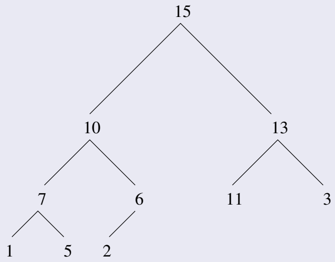
- Chaque exécution correspond à une branche de l'arbre. Le nombre de comparaisons qu'elle exécute est la longueur de la branche
- Le pire des cas correspond à la hauteur de l'arbre
- Tout algorithme de Tri doit différencier les $n!$ permutations. L'arbre a donc au moins $n!$ feuilles
- La hauteur minimum d'un arbre binaire possédant $n!$ feuilles est $log_2(n!) = O(n.log_2 n)$

Propriété

La complexité dans le pire des cas d'un algorithme triant par comparaison un tableau de n éléménts est $O(n.log_2 n)$

Arbre Binaire de Recherche

Retour sur le Tas



Opérations efficaces avec un Tas

- Ajouter un élément
- Supprimer l'élément de valeur maximum

Rechercher un élément dans un Tas de n éléments ? $O(n)$ -> Pas efficace.

Arbre Binaire de Recherche

Principe

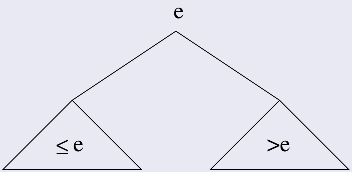
Comme le *Tas Binaire*, **l'Arbre Binaire de Recherche** est une représentation d'un ensemble muni d'un ordre total.
On prend comme exemple les entiers naturels.

Définition

Un **Arbre Binaire de Recherche (ABR)** est un arbre binaire dont tous les noeuds N vérifient :
Pour tout noeud N_1 du sous-arbre gauche de N
Pour tout noeud N_2 du sous-arbre droit de N on a :

$$\text{information de } N_1 \leq \text{information de } N < \text{information de } N_2.$$

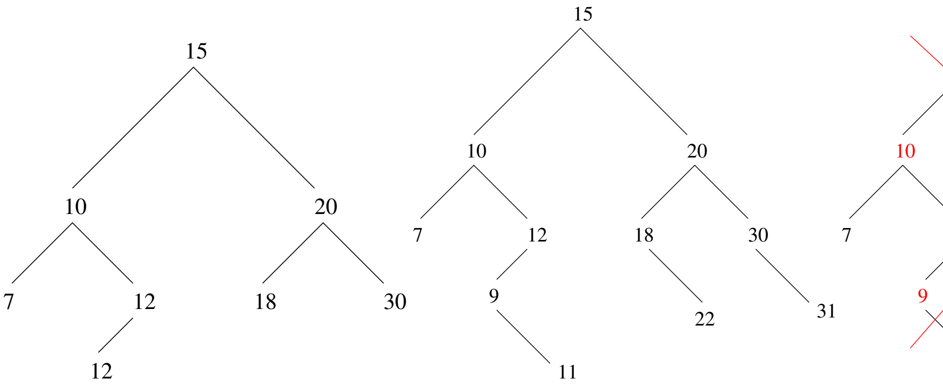
Arbre Binaire de Recherche



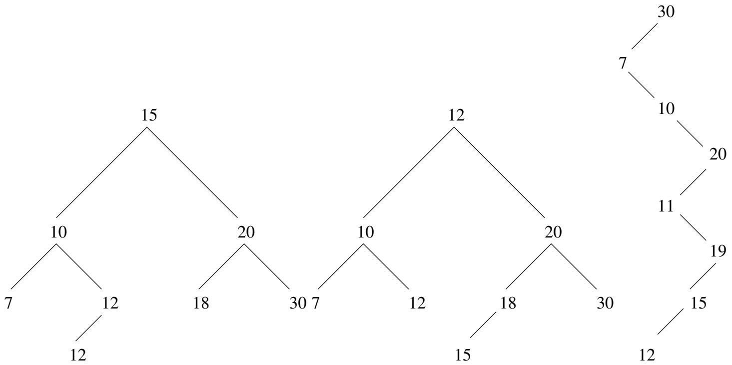
Remarque

Condition sur les étiquettes
Pas de condition sur la forme de l'arbre.

Exemple d'Arbre Binaire de Recherche



Plusieurs ABR peuvent représenter un même ensemble

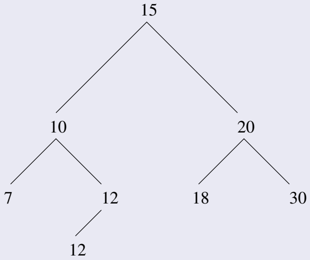


La hauteur d'un ABR représentant un ensemble à n éléments peut varier de $\lfloor log_2(n) \rfloor$ à $n-1$.

Tri et Arbre Binaire de Recherche

Pour obtenir la liste triée des étiquettes d'un ABR
on parcourt l'arbre en ordre infixe.

Exemple



7, 10, 12, 12, 15, 18, 20, 30

Opérations sur les ABR

Les opérations de base pour manipuler un ensemble :

- Rechercher un élément
- Ajouter un élément
- Supprimer un élément

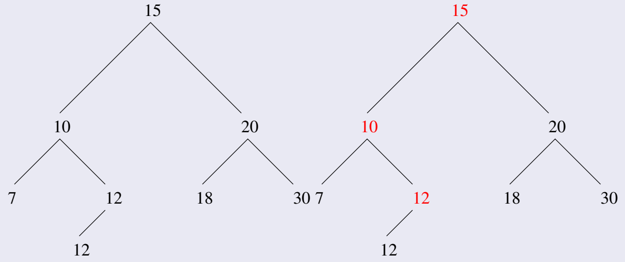
peuvent être réalisées sur un ABR dans un temps proportionnel à la hauteur de l'ABR.

Recherche d'une étiquette dans un ABR

Principe

Si l'élément recherché est différent de l'étiquette de la racine de l'ABR, en les comparant , on sait quel sous–arbre peut contenir l'élément recherché.

Exemple



Recherche de 13 : comparaisons avec 15, 10, 12

Algorithme : Recherche(**d** A : ABR, **d** x :entier)
Données : A 1 ABR, x un entier
Résultat : Renvoie `NULL` si A n'a pas de noeud d'étiquette x
Sinon renvoie un noeud de A ayant x pour étiquette

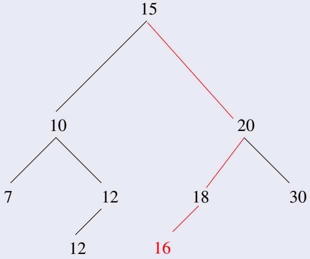
```
début
    si A = NULL ou A↑info = x alors
        renvoyer A
    sinon
        si x < A↑info alors
            renvoyer Rechercher(A↑sag, x)
        sinon
            renvoyer Rechercher(A↑sad, x)
        fin si
    fin si
fin algorithme
```

Complexité : $O(hauteur(A))$

Insertion dans un ABR

Exemple

Ajout du nouveau noeud comme feuille de l'arbre

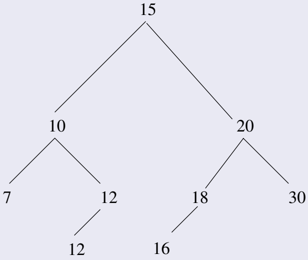


Ajout de 16

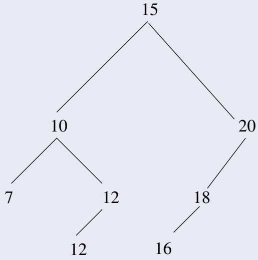
Algorithme : Insertion(**dr** A : ABR,**d** x : entier)
Données : A 1 ABR, x un entier
Résultat : Modifie A en y ajoutant un noeud d'étiquette x
début
  si $A = NULL$ **alors**
    $A \leftarrow$ créerArbre($x, NULL, NULL$)
  sinon si $x \leq A \uparrow info$ **alors**
    Insertion($A \uparrow sag, x$)
  sinon
    Insertion($A \uparrow sad, x$)
fin algorithme

Complexité : $O(hauteur(A))$

Cas 1 : Si le noeud à supprimer est une feuille

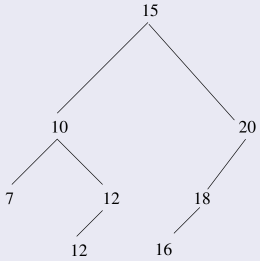


suppression de 30

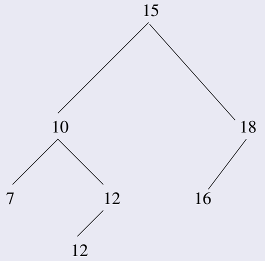


Suppression du noeud

Cas 2 : Si le noeud à supprimer a l'un de ses 2 sous-arbres vide



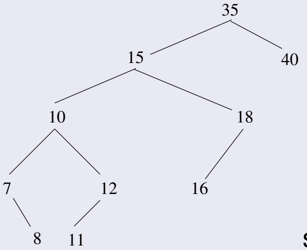
suppression de 20



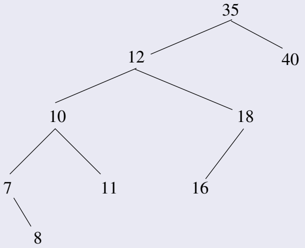
On remplace le noeud par le sous–arbre non vide

Suppression dans un ABR

Cas 3 : Si le noeud à supprimer n'a pas de sous-arbre vide

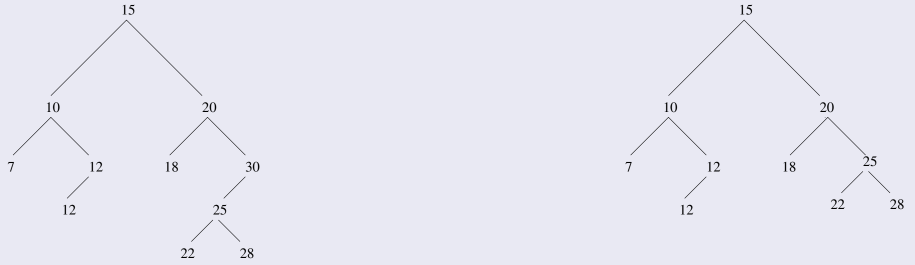


suppression de 15



On remplace l'étiquette par l'étiquette qui la précède dans l'arbre
On supprime cette dernière
Comment trouver l'étiquette qui précède l'étiquette d'un noeud ?
Étiquette max de son sous–arbre gauche

Suppression du noeud d'étiquette max dans un ABR



Suppression de l'étiquette max d'un ABR

Algorithme : SupprimerMax(**dr** A : ABR, **r** max : entier)
Données : A 1 ABR non vide
Résultat : max est la plus grande étiquette de A ; supprime l'étiquette max de l'arbre A
début
  si $A \uparrow sad = NULL$ **alors**
    $max \leftarrow A \uparrow info$; $A \leftarrow A \uparrow sag$;
  sinon
    SupprimerMax($A \uparrow sad$, max)
  fin si
fin algorithme

Complexité : $O(hauteur(A))$

Suppression dans un ABR

Algorithme : Suppression(**dr** A : ABR, **d** x : entier)
Données : A 1 ABR non vide contenant l'étiqtette x
Résultat : Supprime de A un noeud d'étiquette x
début
  si $x < A \uparrow info$ **alors**
    Suppression($A \uparrow sag, x$)
  sinon si $A \uparrow info > x$ **alors**
    Suppression($A \uparrow sad, x$)
  sinon
    $/* \quad x = A \uparrow info \quad */$
    si $A \uparrow sag = NULL$ **alors**
      $A \leftarrow A \uparrow sad$
    sinon
      SupprimerMax($A \uparrow sag, max$) ; $A \uparrow info \leftarrow max$
    fin si
fin algorithme

Complexité : $O(hauteur(A))$

Conclusion

Dans un ABR Rechercher, ajouter et supprimer un élément peuvent être réalisés en `O(hauteur)`. Mais la hauteur d'un ABR peut être égale au nombre d'éléments de l'ABR.

Si dans le pire des cas la hauteur d'un arbre est de l'ordre du nombre de ses noeuds, en moyenne elle est de l'ordre du logarithme du nombre de noeuds. Il existe des Structures de Données qui

- vérifient la même condition sur les valeurs que les ABR
- vérifient une condition sur la Forme de l'arbre, garantissant une hauteur en `O(log(nombre d'éléments))`
- permettent de réaliser les 3 opérations avec la même complexité (en `O(hauteur)`)

`AVL, Arbre Rouge et Noir`