

LIF5 Algorithmique et programmation procédurale

Table des matières

1. Chapitre : Rappels	4
1.1. Tableau des bases 10,16 et 2	4
1.2. Codage Décimal (base 10) à Binaire (base 2)	4
1.3. Codage Décimal (base 10) à Hexadécimal (base 16)	4
1.4. Codage des nombres négatifs	5
1.5. Dépassement de capacité (overflow)	5
1.6. Nombres signés (langage machine)	5
1.7. Soustraction en base 2	5
1.8. Multiplication en base 2	6
1.9. Division en base 2	6
2. Chapitre : Codage des réels : la représentation en virgule flottante	7
2.1. Définition	7
2.2. Codage des réels : le standard utilisé par les ordinateurs	7
2.3. Exemples et représentations	8
3. Chapitre : Vie et mort des variables en mémoire	9
3.1. L'espace d'adressage d'un programme sous Linux	9
3.2. Ce qu'il se passe en mémoire lorsqu'on appelle une fonction ou une procédure	10
3.3. Taille allouée dans la pile (modèle théorique)	11
3.4. Les variables de type pointeur : définition	12
3.5. A quoi servent les pointeurs	12
3.6. Utilité des pointeurs : l'allocation dynamique de mémoire	13
3.7. Norme algorithmique	13
3.8. Lors de l'appel à une fonction (important)	13
3.9. Utilité des pointeurs : bilan	13
4. Chapitre : Types agrégés : tableaux et structures	14
4.1. Problème	14
4.2. Algorithmes pouvant s'appuyer sur une structuration en tableau simple	15
4.2.1. Tri par sélection :	15
4.2.2. Tri par insertion : Le tri le plus efficace sur des listes de petite taille.	15
4.3. Fonctionnement	16
4.3.1. Tri par sélection	16
4.3.2. Tri par insertion	16
4.4. Calcul de complexité	17
4.5. Définition : Invariant de boucle	18
5. Chapitre : Les fichiers	19
5.1. Généralités sur les fichiers	19
5.2. Fichier texte	19
5.3. Fichier binaire	19
5.4. Manipuler des fichiers depuis un programme, lecture/écriture	20
5.5. Ecrire un fichier en langage algorithmique	20
5.6. Ecrire un fichier en C	20
5.7. Lire un fichier en langage algorithmique	21
5.8. Résumé	21
5.9. Lecture d'un fichier et piège	22
5.10. Tri fusion sur fichiers	22
5.10.1. Tri externe :	22
5.10.2. Notion de monotonie	22
5.10.3. Principe du tri par fusion	22
5.10.4. Avec 3 fichiers (à connaître)	23
6. Chapitre : Les types de données abstraits (TDA) et programmation séparée	24

6.1.	Génie logiciel	24
6.2.	Types de données abstraits : TDA	24
6.3.	Norme Algorithmique	24
6.4.	Exemple avec des nombres complexes (Algorithme)	25
6.5.	Le programme du coté utilisateur	26
6.6.	Intérêt de séparer interface et implantation	26
6.7.	Norme en C/C++	26
6.8.	Mot-clé <i>static</i> : (utile pour TP sur les arbres)	26
6.9.	Compiler un programme réparti sur plusieurs fichiers	27
7.	Chapitre : Algorithmique et structures de données	28
7.1.	Les Types de Données Abstraits	28
7.2.	Première solution : Structure du Tableau Dynamique en c, implémentation	29
7.3.	Deuxième solution : Liste d'éléments (type abstrait)	30
7.4.	Représentation graphique d'une liste chaînée (structure à accès séquentiel) :	30
7.5.	Implémentations	31
7.5.1.	Création d'une liste chaînée	31
7.5.2.	Chaînage en tête (suite de l'image précédente)	32
7.5.3.	Destruction d'une liste	33
7.5.4.	Chaînage en queue	34
7.5.5.	Exemple de codes	35
7.5.6.	Bilan du chaînage	38
7.5.7.	Liste chaînée circulaire	39
7.6.	Makefile (plus développé que sur la p.23)	40
8.	Chapitre : TDA Pile et File	41
8.1.	TDA Pile	41
8.2.	TDA File	42
8.2.1.	Rappel sur les Arbres	42
8.2.2.	Les arbres binaires	42
8.2.3.	Arbre binaire avec les listes chaînées	43
8.2.4.	Trois façons de parcourir un arbre	44
8.2.5.	Afficher un arbre par niveau (4 ^{ème} façon)	46
8.3.	Arbre binaire de recherche	47
8.4.	Arbres binaires et tableau (wikipedia)	47

1. Chapitre : Rappels

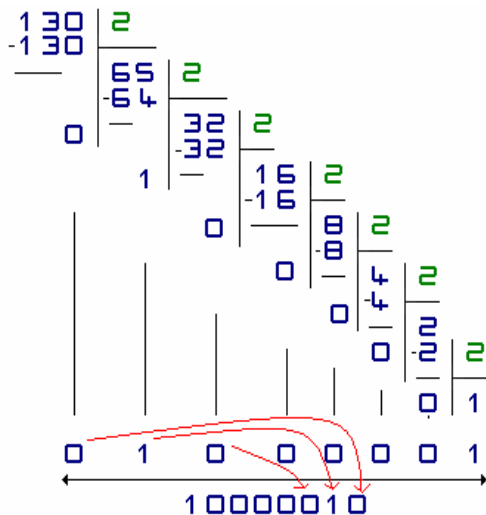
1.1. Tableau des bases 10,16 et 2

Base 10	Base 16	Base 2
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

1.2. Codage Décimal (base 10) à Binaire (base 2)

107 (est en base 10 par défaut et s'écrit 107_{10}) :

On soustrait à 107 la plus grande puissance de 2 sans dépasser le reste :



$$\begin{array}{rclclcl}
 107 & - & 2^6 & = & 43 & \Leftrightarrow & 107 & - & 64 & = & 43 \\
 43 & - & 2^5 & = & 11 & \Leftrightarrow & 43 & - & 32 & = & 11 \\
 11 & - & 2^3 & = & 3 & \Leftrightarrow & 11 & - & 8 & = & 3 \\
 3 & - & 2^1 & = & 1 & \Leftrightarrow & 3 & - & 2 & = & 1 \\
 1 & - & 2^0 & = & 0 & \Leftrightarrow & 1 & - & 1 & = & 0
 \end{array}$$

On a donc

$$0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Ce qui équivaut à 01101011.

On peut aussi procéder par divisions successives par 2 (surtout si le nombre est grand).

1.3. Codage Décimal (base 10) à Hexadécimal (base 16)

On fait déjà le passage décimal au binaire, donc $107_{10} \Rightarrow 01101011_2 \Rightarrow 0110 \ 1011_2$

On découpe le nombre binaire par groupe de 4 bits puis on obtient :

$$0110_2 \Leftrightarrow 6_{16} \qquad 1011_2 \Leftrightarrow B_{16}$$

On a alors $107_{10} = 6B_{16}$

1.4. Codage des nombres négatifs

codage de -45 (sur 8 bits)

- valeur absolue = $45_{10} = 00101101_2$
- complément à 1 = 11010010_2 → On inverse les 1 et 0
- complément à 2 = $11010010_2 + 1_2 = 11010011_2$ → On ajoute 1 si négatif

1.5. Dépassement de capacité (overflow)

Exemple avec l'addition :

$$\begin{array}{r} \text{x } 11001000 \\ + 01100100 \\ \hline \text{X } 00101100 \end{array}$$

La dernière retenue et le dernier 1 sont perdus !

1.6. Nombres signés (langage machine)

Soit $(11010010)_2$. Si le bit de poids fort (celui le plus à gauche) est à 1 → c'est un nombre négatif, sinon c'est un positif. Si on le traduit sur \mathbb{N} on a $(210)_{10}$ mais pour l'avoir sur \mathbb{Z} il faut faire enlever le complément à 2 et faire le complément à 1, ce qui donne :

On enlève le Complément à 2 : $(11010010)_2 - (00000001)_2 = (11010001)_2$
Complément à 1 (inversion des bits) : $(11010001)_2 \rightarrow (00101110)_2 = (-46)_{10}$

Cela revient à faire en base 10 (décimal) : $210 - 2^8 = 210 - 256 = -46$ sur \mathbb{Z} car c'est un négatif. Si le bit de poids fort était à 0 on aurait $(01010010)_2 = (82)_{10}$ sur \mathbb{Z} .

1.7. Soustraction en base 2

On va prendre l'exemple : $10 - 3 = 7$

Equivalent à :

$$\begin{array}{r} 1010 \rightarrow 10 \\ - 0011 \rightarrow 3 \\ \hline \end{array}$$

1101 → Faux, on trouve 13

On a alors :

$$\begin{array}{r} 1010 \rightarrow 10 \\ - 0011 \text{ Faux} \rightarrow \text{complément à 1} = 1100 \rightarrow \text{complément à 2} = 1100 + 0001 = 1101 \text{ Bon} \\ + 1101 \text{ Bon} \\ \hline \end{array}$$

$$\text{1011} \rightarrow (0111)_2 = 7_{10}$$

Si un bit se rajoute à la fin on l'ignore, il est là pour régler le problème avec la soustraction (si on a $1000 - 0110$ on fait le Ca1 et Ca2 sur $(0110)_2 \rightarrow 1000 + 1010 = 0010 = 2 \quad 8 - 6 = 2$).

1.8. Multiplication en base 2

```

  0 1 0 1 multiplicande
x 0 0 1 0 multiplicateur
-----
  0 0 0 0
 0 1 0 1
0 0 0 0
-----
0 1 0 1 0

```

1.9. Division en base 2


En binaire: $25 = (11001)_2$ $4 = (100)_2$

<p>1. On prend le nombre de chiffre nécessaire pour la première division (ici 3)</p> <pre> 11001 100 ----- </pre>	<p>5. On recommence tant que tous les chiffres n'ont pas été abaissés</p> <pre> 11001 100 0100 ----- 11 </pre>
<p>2. Naturellement on pose un 1 en quotient puis on fait la soustraction $110 - (1 * 100)$</p> <pre> 11001 100 ----- 1 </pre>	<p>6. On fait la soustraction</p> <pre> 11001 100 0100 ----- 11 </pre>
<p>3. Ce qui donne</p> <pre> 11001 100 010 ----- 1 </pre>	<p>7. On abaisse le chiffre suivant</p> <pre> 11001 100 0100 ----- 11 </pre>
<p>4. On abaisse un chiffre, ici un zéro</p> <pre> 11001 100 0100 ----- 1 </pre>	<p>8. Celui ci va zéro fois, on pose un zéro dans le quotient et on abaisse le chiffre suivant, mais il n'y en a plus donc c'est fini.</p> <pre> 11001 100 0100 ----- 110 </pre>

2. Chapitre : Codage des réels : la représentation en virgule flottante

2.1. Définition

Notation scientifique usuelle : $a = m \cdot 10^e$



- Nombre de chiffres alloués à l'exposant amplitude de l'intervalle
- Nombre de chiffres alloués à la mantisse précision du codage

- Multiplication :

- addition des exposants
- multiplication des mantisses

- Division :

- soustraction des exposants
- division des mantisses

- Addition (ou soustraction) :

- dénormalisation de la plus petite valeur
- addition (ou soustraction) des mantisses

– renormalisation si nécessaire pour les 3 cas précédents

Attention : Risque d'overflow ou d'underflow

+ Risque d' « absorption » lors de l'addition

+ Risque de « cancellation catastrophique » lors de la soustraction

Si les deux opérandes sont trop différents (en exposant), alors la plus petite valeur sera, au cours de la dénormalisation, confondue avec 0 et donc négligée dans le calcul...

⇒ $a \gg b$, alors $a + b = a$ (phénomène d'absorption)

Lors de la renormalisation, l'incertitude « conquiert » les deux derniers digits stockés : la soustraction révèle les approximations précédentes (dépend de la norme choisie).

2.2. Codage des réels : le standard utilisé par les ordinateurs

Principe mantisse - exposant, mais avec des puissances de 2 et non de 10

- Standard IEEE 754 simple précision (4 octets, type « float » en C) :

$(-1)^s \cdot (1 + f) \cdot 2^{(e-127)}$ avec $f = 0$, quelque chose

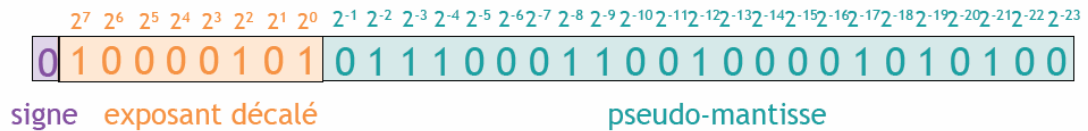
- Standard IEEE 754 double précision (8 octets, type « double » en C) :

$(-1)^s \cdot (1 + f) \cdot 2^{(e-1023)}$ avec $f = 0$, quelque chose

- e est codé sous forme de puissances positives de 2
- f est codé sous forme de puissances négatives de 2

2.3. Exemples et représentations

Exemple en simple précision :



$$s = 0$$

$$e = 2^0 + 2^2 + 2^7 = 1 + 4 + 128 = 133$$

$$\begin{aligned} f &= 2^{-2} + 2^{-3} + 2^{-4} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-17} + 2^{-19} + 2^{-21} \\ &= 0,25 + 0,125 + 0,0625 + 0,00390625 + \dots + 0,000000476837158203125 \\ &= 0,443613529205322 \end{aligned}$$

$$\begin{aligned} \text{Nombre} &= (-1)^0 \cdot (1 + 0,443613529205322) \cdot 2^{(133-127)} \\ &= 1 \cdot 1,443613529205322 \cdot 2^6 \\ &= 92,3912658691405 \end{aligned}$$

Quelle est la représentation IEEE 754 simple précision de -2,625 ?

- Nombre négatif, donc $s = 1$
- La valeur absolue peut s'exprimer comme somme de puissances positives et négatives de 2 :
 $2,625 = 2 + 0,5 + 0,125 = 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 10,101_2$
- Il faut normaliser pour que la mantisse s'écrive « 1,quelquechose »
 Ici on décale la virgule d'un cran à gauche : $1,0101 \cdot 2^1$
- Identification : $(1 + f) \cdot 2^{(e-127)} = 1,0101 \cdot 2^1$

$$\Rightarrow f = 0,010100000000000000000000_2$$

$$\Rightarrow e - 127 = 1 \quad \text{donc } e = 128_{10} = 01111111_2$$



Attention : L'ordinateur ne peut pas manipuler exactement 0,1 ! (ni 0,2 ni 0,3 ...).

	Signe	Exposant décalé	Pseudo-mantisse
Nombre normalisé	0 ou 1	toute configuration de bits, sauf « tous les bits à 0 » ou « tous les bits à 1 »	toute configuration de bits
Zéro	0 ou 1	tous les bits à 0	tous les bits à 0
Infini	0 ou 1	tous les bits à 1	tous les bits à 0
Not a Number (NaN)	0 ou 1	tous les bits à 1	toute configuration de bits non nulle

A retenir : L'arithmétique des ordinateurs n'est PAS celle (idéale) des mathématiques, car les ordinateurs travaillent avec un nombre fini de valeurs.

A retenir : Les algorithmes peuvent s'appuyer sur une structuration particulière des données pour être plus efficaces.

3. Chapitre : Vie et mort des variables en mémoire

Méthode de traitement des éléments d'une file : **FIFO : First In, first Out**

Méthode de traitement des éléments d'une Pile : **LIFO : Last in, first Out**

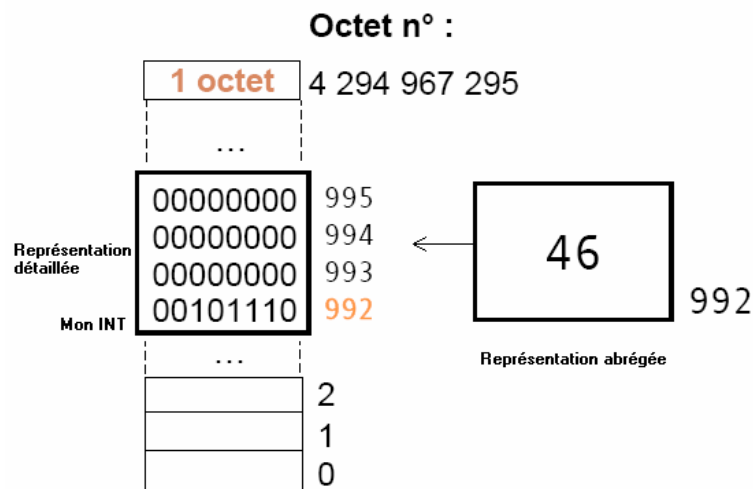
Variable : Accès au contenu d'un emplacement mémoire, seul ça valeur peut changer au cours d'une exécution. Dans une expression, écrire un nom de variable revient à récrire le contenu de l'emplacement mémoire n°....

- Les variable globale sont déclarées en dehors de tout bloc d'instructions { }.
- Les variables locales sont déclarées à l'intérieur d'un bloc d'instructions (dans une fonction ou une boucle par exemple), elles sont inutilisables dans un autre bloc.

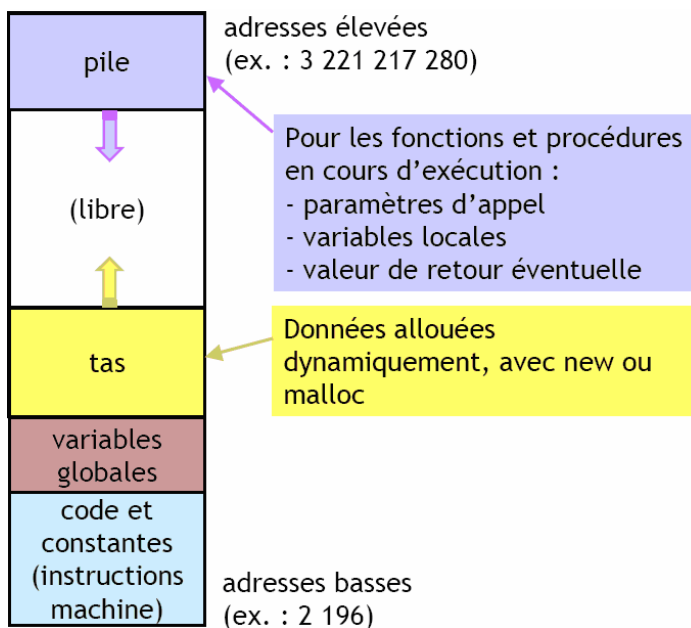
Mémoire : La mémoire est codée sur 32 bits $\rightarrow 2^{32} - 1$ soit environ 4 milliards d'adresses.

Adresse mémoire d'une variable = adresse du 1^{er} octet qu'elle occupe.

Une adresse mémoire = 4 octets sur 4 milliards d'octets (4 Go alloués au maximum par l'os).



3.1. L'espace d'adressage d'un programme sous Linux



Type	Longueur (octets, pour la pile)	Printf	Scanf
Char	1	%c	%s
Short	2	%d	%hd
Int	4	%d	%d
Float	4	%f	%f
Long	4	%ld	%ld
Double	8	%f	%lf
Long double	16		
Multiplier par 8 la « Longueur » pour avoir le nombre de bits nécessaire au codage.			

3.2. Ce qu'il se passe en mémoire lorsqu'on appelle une fonction ou une procédure

Pour une fonction

```
#include <stdio.h> /* pour printf */
```

```
double calculSolution (double a, double b)
{
    /* Préconditions : a n'est pas nul
    Résultat : Calcule et retourne la solution de
    l'équation  $ax + b = 0$  */
```

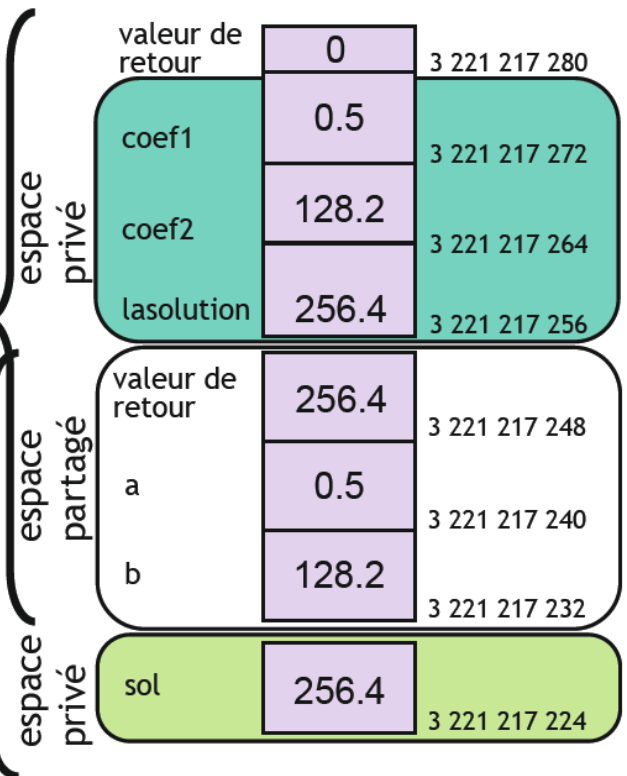
```
    double sol;
    sol = - b / a;
    return (sol);
}
```

```
int main()
```

```
{
    double coef1 = 0.5;
    double coef2 = 128.2;
    double lasolution;
    lasolution = calculSolution(coef1, coef2);
    printf("La solution est %f \n", lasolution);
    return 0;
}
```

Frame
de main

Frame de
calculSolution



Fonctionnement de la Pile (frame = cadre)

Les instructions d'une fonction ne peuvent manipuler que les données de sa frame.

- 1- Le main va attribuer de l'espace pour ses variables : coef1, coef2, lasolution dans l'espace privé
- 2- La fonction CalculSolution est lancée, une nouvelle frame est créée
- 3- L'espace partagé est créé avec les variables a, b + un espace privé réservé à la solution 'sol'
- 4- La solution est mise en mémoire dans 'sol'
- 5- La solution est mise en mémoire dans la valeur de retour de l'espace partagé (3 221 217 248)
- 6- Après le return, l'espace partagé est supprimé sauf la valeur de retour
- 7- la valeur de retour est inscrite à l'adresse 3 221 217 256 'lasolution'
- 8- La valeur de retour est supprimée de la pile
- 9- Une partie de la mémoire est allouée pour le printf (n'apparaît pas dans la pile),
- 10- Le champs « Valeur de retour » en haut de la pile est remplie par un 0,
- 11- Tout est effacé de la mémoire.

Fonctions :

Paramètre effectif : quand on fait un appel à une fonction, le a et b sont des paramètres effectifs.

Exemple : résultat = fonction (a,b);

Paramètre formel : pour une fonction, le a et b sont des paramètres formel.

Exemple : double calcul (int a, int b) { ... }

3.3. Taille allouée dans la pile (modèle théorique)

On part de l'adresse la plus haute et on descend (c'est l'inverse pour le tas). A chaque variable créée dans la pile, l'adresse de départ de la pile va diminuer.

Taille disponible : $2^{32} - 1 = 4\,294\,967\,295$ octets

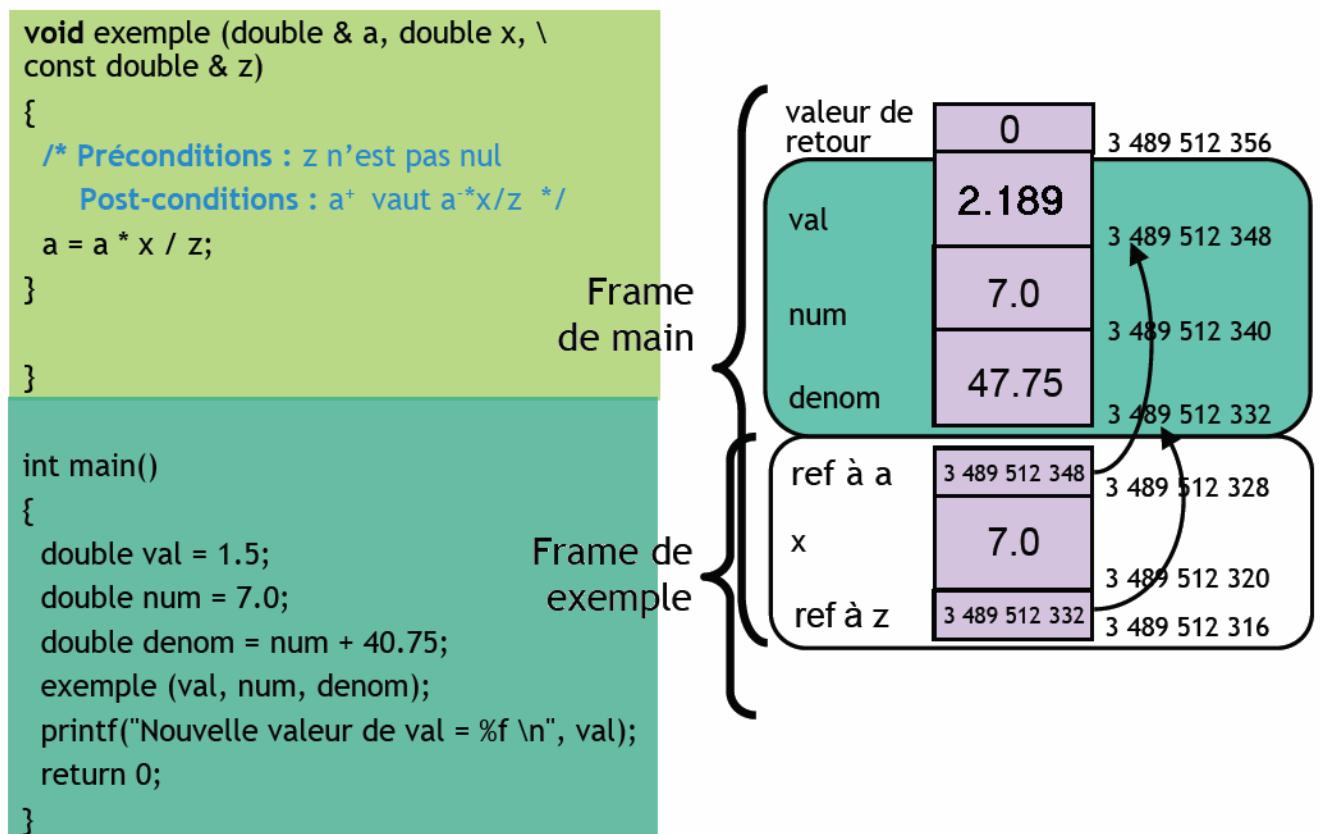
Taille allouée pour chaque déclaration :

Char : 1 octet
 Short : 2 octets
 Int : 4 octets
 Double : 8 octets
 Pointeur : 4 octet (on stock toujours l'adresse de la variable pointée)

Exemple : adresse de départ (image de la page précédente)

Adresse de départ : 3 221 217 180 → Valeur de retour
 Double coef1 ; 3 221 217 172 → -8 octets
 Short mot ; 3 221 217 170 → -2 octets
 Double lasolution ; 3 221 217 162 → -8 octets
 Double *monpointeur ; 3 221 217 158 → -4 octets

Pour une procédure (rappel : & signifie donné – résultat)



Variables passées en donné - résultat : on récupère l'adresse de la variable au lieu de copier la variable.
Vocabulaire : Boucle non fermée = stack overflow (dépassement de la pile), la pile s'agrandit jusqu'à dépasser la place allouée.

Passage de paramètres par référence : à retenir

- ✚ Permet à une procédure d'accéder aux données (normalement privées) de la frame appelante
- ✚ Entorse au principe de séparation des données
- ✚ Permet à la procédure d'avoir des effets de bord

3.4. Les variables de type pointeur : définition

Un pointeur : variable destinée à contenir une adresse mémoire (4 octets).

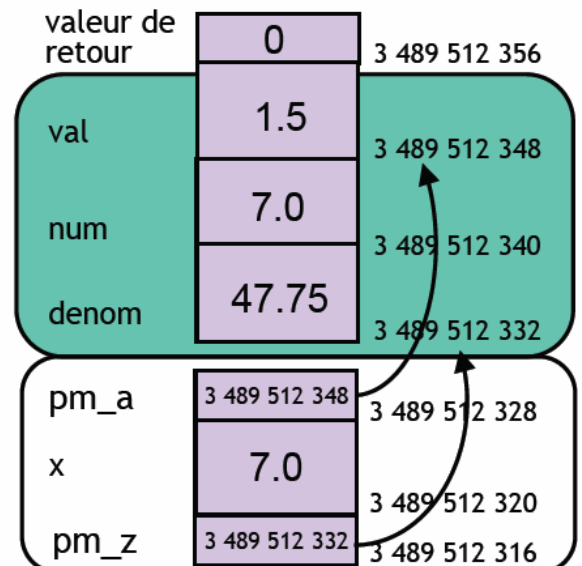
En algorithme	En C	
p : pointeur sur type	type *p ;	pointeur sur type
&	&	Accès à l'adresse d'une variable
Pointeur↑	*monpointeur tab []	Accès à la valeur d'une adresse pointée
		Mode donnée
	type a	Petits objets
	const type *a	Gros objets (structures, tab, etc)
	type *a	➔ Mode donnée- résultat ou résultat

Dans le tableau ci-dessus, **type** est égal à int ou double ou float ou etc ...

3.5. A quoi servent les pointeurs

```
void exemple (double * pm_a, double x, \
const double * pm_z)
{
    /* Préconditions : z n'est pas nul
    Post-conditions : a+ vaut a*x/z */
    *pm_a = (*pm_a) * x / (*pm_z);
}
```

```
int main()
{
    double val = 1.5;
    double num = 7.0;
    double denom = num + 40.75;
    exemple (&val, num, &denom);
    printf("Nouvelle valeur de val = %f \n", val);
    return 0;
}
```



- ✚ Attention lors de l'appel, fournir explicitement une adresse et non une valeur.

3.6. Utilité des pointeurs : l'allocation dynamique de mémoire

Tas (voir schéma p6) : Les données mises dans le tas ne sont pas détruites en sortie de fonction ou de procédure (à l'inverse de la pile).

Norme algorithmique	Langage C
p : pointeur sur réel	double *p;
p ← réserve réel	p = (double *) malloc(sizeof(double));
...	...
libère p	free(p);
p ← réserve tableau[1..4] de réels	p = (double *) malloc(4*sizeof(double));
...	...
libère p	free(p);

3.7. Norme algorithmique

Réservation

p ← réserve type

Ou : p ← réserve tableau [1..n] de type

Libérer

Libère p Rend l'emplacement mémoire disponible pour d'autres utilisations éventuelles (les données sont perdues).

malloc et free sont fournis par la librairie stdlib : #include <stdlib.h> nécessaire.

new et delete sont des opérateurs, pas besoin de lier une librairie.

3.8. Lors de l'appel à une fonction (important)

Void Fonction (double *var) { ... }

/* Le double sert simplement à dire comment on va utiliser la donnée *var, ce qui équivaut à dire 'double a' vu que l'on pointe vers la valeur de a (sauf que le pointeur est un gain pour la mémoire).

Important : La taille de 'var' dans l'appel sera toujours de 4 octets dans la pile et cela quelque soit le type placé avant : int/double/float/etc ... *var. */

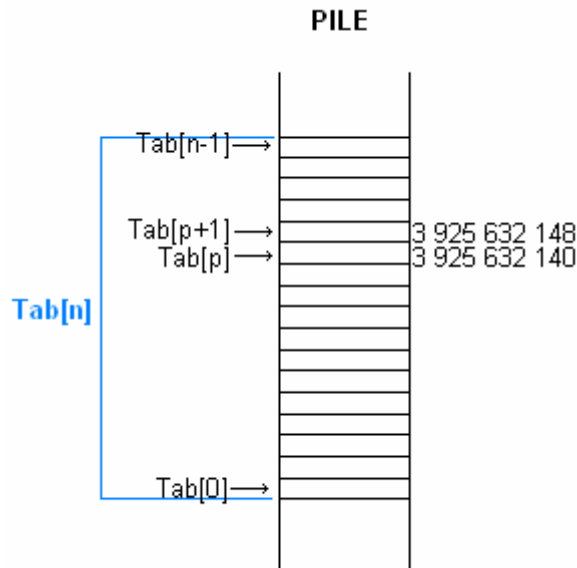
```
Int main () {  
    double a = 3 ;  
    double *p;    // on crée le pointeur  
    p = &a ;      //on l'initialise avec l'adresse de a (important)  
    Fonction (p) ; //on envoi l'adresse comme paramètre  
}
```

3.9. Utilité des pointeurs : bilan

- Passage de paramètres en mode donné/résultat
- Allocation dynamique dans le tas
- Gestion des tableaux en C

4. Chapitre : Types agrégés : tableaux et structures

Exemple : double Tab[n] : Soit $p > n$ alors si Tab[p] à pour adresse 3 925 632 140, l'élément suivant Tab[p+1] aura pour adresse 3 925 632 148 (car on à n-1 cases qui sont de type double).



A retenir : En C, la mise en oeuvre de l'opérateur [] repose sur l'arithmétique des pointeurs.

4.1. Problème

```
void afficher (const double * t, int taille) { ... }
```

```
double * doubler (const double * t)
{
  /* Précondition : t tableau de taille 3 */
  double res[3];
  int i;
  for(i = 0; i < 3; i++) res[i] = t[i] * 2.0;
  return res ;
}
int main()
{
  double lesNombres[] = { 6.1, 51.7, 8.4 };
  double * lesAutres ;
  lesAutres = doubler(lesNombres);
  afficher(lesAutres, 3);
  return 0;
}
```

Problème : 'return res ;' ➔ Retour de l'adresse d'un tableau détruit, il faut donc le stocker dans le tas pour pouvoir le conserver à la fin de la fonction : allocation dynamique par « malloc » et le libérer ensuite avec « free ».

4.2. Algorithmes pouvant s'appuyer sur une structuration en tableau simple

4.2.1. Tri par sélection :

Recherche le plus grand élément (ou le plus petit) que l'on va remplacer à sa position finale c.a.d en dernière position (ou 1ère), de même pour le second que l'on va remplacer également à sa position finale c.a.d en avant-dernière position (ou en seconde), etc., jusqu'à ce que le tableau soit entièrement trié.

```
typedef int tab_entiers[MAX];
void selection(tab_entiers t)
{
    int i, min, j , x;
    for(i = 0 ; i < MAX - 1 ; i++)
    {
        min = i;
        for(j = i+1 ; j < MAX ; j++)
            if(t[j] < t[min])
                min = j;
        if(min != i)
        {
            x = t[i];
            t[i] = t[min];
            t[min] = x;
        }
    }
}
```

4.2.2. Tri par insertion : Le tri le plus efficace sur des listes de petite taille.

Il suffit de parcourir une liste : on prend les éléments dans l'ordre. Ensuite, on les compare avec les éléments précédents jusqu'à trouver la place de l'élément qu'on considère. Il ne reste plus qu'à décaler les éléments du tableau pour insérer l'élément considéré à sa place dans la partie déjà triée.

```
#define MAX 100

void insertion(int t[MAX])
{
    /* Spécifications externes : Tri du tableau t par insertion séquentielle */
    int i,p,j;
    int x;

    for (i = 1; i < MAX; i++)
    {
        /* stockage de la valeur en i */
        x = t[i];

        /* recherche du plus petit indice p inférieur à i tel que t[p] < t[i] */
        for(p = 0; t[p] < x; p++);
        /* p pointe une valeur de t supérieure à celle en i */

        /* décalage avant des valeurs de t entre p et i */
        for (j = i-1; j >= p; j--) {
            t[j+1] = t[j];
        }

        t[p] = x; /* insertion de la valeur stockée à la place vacante */
    }
}
```

4.3. Fonctionnement

4.3.1. Tri par sélection

Procédure tri_selection(tab : tableau[1..n] de Element)

Précondition : tab[1], tab[2], .. , tab[n] initialisés

Postcondition : tab[1] ! tab[2] ! .. ! tab[n]

Paramètre en mode donnée-résultat : tab

Variables locales : i, j, indmin : entiers min : Element

Début	
Pour i de 1 à n-1 par pas de 1 Faire	
indmin <- i	
Pour j de i+1 à n Faire	On recherche le plus petit élément de la partie non triée. On sauvegarde son indice dans "indmin".
Si tab[j] < tab[indmin] alors	
indmin <- j	
FinSi	
FinPour	
min <- tab[indmin]	On permute le minimum (d'indice "indmin") avec l'élément courant (d'indice i).
tab[indmin] <- tab[i]	
tab[i] <- min	
FinPour	
Fin tri_selection	

4.3.2. Tri par insertion

```
Chaine a trier : 5 4 3 2 1 0
*****
valeur 4 a mettre en tab[0] a la place de la valeur 5
1 decalage(s) :
5 5 3 2 1 0
4 5 3 2 1 0 Insertion de la valeur
*****Fin boucle 1*****
valeur 3 a mettre en tab[0] a la place de la valeur 4
2 decalage(s) :
4 5 5 2 1 0
4 4 5 2 1 0
3 4 5 2 1 0 Insertion de la valeur
*****Fin boucle 2*****
```

1. On enregistre dans l'élément x la valeur de l'élément courant t[i].
2. On compare x avec les nombres de gauche qui ont déjà été triés (soit p=0 la variable de boucle).
3. Quand enfin x est supérieur à l'élément t[p] de la partie triée, on garde la variable p qui sera la position où x sera inséré.

4. On va ensuite décaler tous les nombres à droite, compris entre la position d'insertion p et l'élément courant x=t[i] pour pouvoir insérer x (le nombre sauvegardé) au bon endroit de la partie triée.
5. Donc le nombre à la position t[i] (notre valeur enregistrée dans x) va être remplacé par le nombre à la position t[i-1]. On aura donc 2 fois la valeur de t[i-1] dans notre tableau (à i-1 et i).


```

valeur 2 a mettre en tabl[0] a la place de la valeur 3
3 decalage(s) :
3 4 5 5 1 0
3 4 4 5 1 0
3 3 4 5 1 0
2 3 4 5 1 0 Insertion de la valeur
*****Fin boucle 3*****
valeur 1 a mettre en tabl[0] a la place de la valeur 2
4 decalage(s) :
2 3 4 5 5 0
2 3 4 4 5 0
2 3 3 4 5 0
2 2 3 4 5 0
1 2 3 4 5 0 Insertion de la valeur
*****Fin boucle 4*****
valeur 0 a mettre en tabl[0] a la place de la valeur 1
5 decalage(s) :
1 2 3 4 5 5
1 2 3 4 4 5
1 2 3 3 4 5
1 2 2 3 4 5
1 1 2 3 4 5
0 1 2 3 4 5 Insertion de la valeur
*****Fin boucle 5*****
*****
Chaine triee : 0 1 2 3 4 5

```

6. On répète l'opération $i-p$ fois pour avoir $i-p$ décalages.
7. Quand tout est décalé, le nombre le plus à gauche (pas forcément le dernier à gauche) **qui a été répété** est remplacé par la fameuse valeur x .
8. Ce sera le moment ou on 'insert' x .
9. On répète ces opérations pour la valeur suivante $i+1$ jusqu'à $i=MAX-1$.

L'élément à la position $i=0$ sera automatiquement trié.

4.4. Calcul de complexité

- ✚ On ne compte que les **affectations et les comparaisons d'éléments**,
- ✚ On néglige les affectations et les comparaisons d'indices.

Pour le tri par sélection :

- ✚ $n-1$ passages dans la boucle «"i"», avec à chaque passage :
 - 3 affectations
 - 0 comparaison
- ✚ $n-i$ passages dans la boucle «"j"», avec à chaque passage :
 - 0 affectation
 - 1 comparaison

Soit au final :

$(n-1)*3$ affectations

$(n-1)*n/2$ comparaisons, Détails :

1 ^{ère} boucle : de $i=1$ à $i=n-1$	2 ^{ème} boucle : de $j= i+1$ à $i=n$	Nombre de comparaisons max	
$i = 1$	$j=2$ à $j=n$	$(n-2) + 1$	$n-1$
$i = 2$	$j=3$ à $j=n$	$(n-3) + 1$	$n-2$
...
$i = n-2$	$j=n-1$ à $j=n$	$(n - (n-1)) + 1$	2
$i = n-1$	$j=n$ à $j=n$	$(n - n) + 1$	1

On additionne 2 sommes pour simplifier les calculs :

Somme des comparaisons = $1 + 2 + \dots + n-2 + n-1$

Somme des comparaisons = $n-1 + n-2 + \dots + 2 + 1$

2*Somme des comparaisons = $n + n + \dots + n + n = (n-1) * n$

Somme des comparaisons = $(n-1)*n/2$

4.5. Définition : Invariant de boucle

L'invariant de boucle respecte ces 3 conditions :

- ✚ Initialisation : La propriété est vraie avant la 1^{re} itération
- ✚ Conservation : Si la propriété est vraie avant l'itération i , alors elle reste vraie avant l'itération $i+1$
- ✚ Terminaison : Une fois la boucle terminée, la propriété est utile pour montrer la validité de l'algorithme.

5. Chapitre : Les fichiers

5.1. Généralités sur les fichiers

Format des informations	
« texte » (codes de caractères)	« binaire » (comme en mémoire vive)
carnet d'adresses personnelles	fichiers image
fichiers .C	fichiers mp3
fichiers de configuration Unix	fichiers doc
etc.	fichiers pdf
	etc.

Les fichiers :

- ✚ Insuffisant pour combler les besoins réels des informaticiens
- ✚ Ils sont là pour stocker des informations de manière permanente, entre deux exécutions d'un programme :
 - Stockage sur des périphériques « à mémoire de masse » (disque dur, clé USB, DVD...)

5.2. Fichier texte

- ✚ Les octets du fichier ne contiennent que des codes de caractères
- ✚ Les données numériques doivent donc être converties en caractères

Avantages :

- Portabilité
- Fichier lisible avec un simple éditeur de texte

Inconvénients :

- Coût en temps induit par les conversions nécessaires
- Penser à écrire suffisamment de décimales pour ne pas perdre en précision numérique

5.3. Fichier binaire

- ✚ On recopie l'information comme elle figure en mémoire vive

Avantages :

- Fichier plus compact
- Lecture / écriture sans conversion = plus rapide
- Pas de formatage de présentation = pas de perte de précision

Inconvénients :

- Fichier illisible par un simple éditeur de texte
- il faut gérer les problèmes de portabilité

5.4. Manipuler des fichiers depuis un programme, lecture/écriture

Doit-on se soucier du support physique des fichiers ?
Il existe des normes algorithmiques pour les supports.

Buffer (mémoire tampon) : savoir ce qui est en bleu/rouge

Les données envoyées vers un périphérique (externe) sont le plus souvent stockées dans des **mémoires tampon** en attente de leur envoi effectif, cela **pour épargner** à l'ordinateur le contretemps dû à **la différence de débits entre le microprocesseur interne et les différents périphériques souvent lents**. De même, les données reçues de l'extérieur sont le plus souvent stockées dans des tampons en attente de leur traitement par l'ordinateur (pour des raisons d'efficacité, et aussi pour éviter qu'une réception de données trop rapprochées fasse que certaines, non traitées, ne soient perdues).

Ecriture : **quand le buffer est plein**, appel système à write = écriture sur le disque,

Lecture : **quand le buffer est vide**, appel système à read pour le remplir = lecture depuis le disque.

5.5. Ecrire un fichier en langage algorithmique

```
monFic: fichier
monFic ← ouvrir « data.bin » en écriture
générer(monFic)           Créer un tampon d'écriture
tampon(monFic) ← 1356     Ecrire 1356 dans le tampon
mettre(monFic)            Ecrire le contenu du tampon sur le disque. Le tampon est alors vide
...
fermer(monFic)
```

5.6. Ecrire un fichier en C

Cas d'un fichier « **Binaire** »

```
FILE * monFic;
monFic = fopen(« data.bin », « wb »);
if (monFic == NULL) {
    printf(« Impossible d'ouvrir le fichier \n»);
    exit(EXIT_FAILURE);}

int e = 1356; int valret;
valret = fwrite(&e, sizeof(int), 1, monFic);

if(valret < 1) { printf(« Erreur d'écriture \n»);
exit(EXIT_FAILURE); }
...
fclose(monFic);
```

fwrite(adresse 1er bloc à copier, taille bloc, nb
blocs à copier, nom interne fichier)

Cas d'un fichier « **Texte** »

```
monFic = fopen(« data.txt », « w »);
```

```
valret = fprintf(monFic, « %d », e);
```

Mettre les codes ASCII de '1', '3', '5', et '6'
dans le buffer

fprintf(nom interne fichier, code(s) de format,
valeur1, valeur2, ...)

Erreurs d'écriture possibles : plus de place sur le volume, erreur matérielle, ...

Attention, la présence du buffer (= Tampon) peut influencer sur le moment où ces erreurs sont détectées.

5.7. Lire un fichier en langage algorithmique

monFic: fichier
e : entier
monFic ← ouvrir « data.txt » en lecture
inspecter(monFic) Réserver un espace en mémoire vive pour le tampon et l'initialiser avec les données se trouvant au début du fichier
e ← tampon(monFic)
prendre(monFic) Remplacer le contenu du tampon par les données suivantes du fichier
...
fermer(monFic)

Cas d'un fichier « Binaire »

```
FILE * monFic;  
int e, valret;  
monFic = fopen(« data.bin », « rb »);  
if(monFic == NULL) {  
    printf(« Impossible d'ouvrir le fichier\n »);  
    exit(EXIT_FAILURE);  
}  
valret = fread(&e, sizeof(int), 1, monFic);  
if(valret != 1) {  
    printf(« Fin de fichier ou erreur de lecture \n »);  
}  
/* ... */  
fclose(monFic);
```

fread(adresse 1er bloc à remplir, taille bloc, nb blocs à remplir, nom interne fichier)




Cas d'un fichier « Texte »

```
monFic = fopen(« data.txt », « r »);  
  
valret = fscanf(monFic, « %d », &e);
```

fscanf(nom interne fichier, code(s) de format, adresse variable 1, adresse variable 2, ...)

5.8. Résumé

Lecture d'un fichier binaire :

-  Lire les 4 premiers octets du tampon (il est automatiquement rempli d'après le fichier s'il est vide),
-  Les copier dans la variable valret,
-  Puis les enlever du tampon

Lecture d'un fichier texte :

-  Lire les premiers caractères du tampon jusqu'à tomber sur un espace ou une fin de ligne,
-  Convertir ces caractères vers l'entier correspondant mettre ce nombre dans la variable e,
-  Puis enlever les caractères traités du tampon

Binaire : rb Texte : r ➔ Lecture
Binaire : wb Texte : w ➔ Ecriture

5.9. Lecture d'un fichier et piège

Il ne faut pas utiliser « feof » comme condition de bouclage car on boucle une fois de trop et donc la lecture des derniers octets ne permet pas de conclure ! Il faut utiliser la valeur de retour de « fscanf » ou « fread ».

```
int nb_passages = 0;
```

```
while(fscanf(monFic, "%d", &monInt) == 1)
{ printf("Je viens de lire %d\n", monInt);
  nb_passages++; }
```

```
if(feof(monFic)) printf("Fin de fichier \n"); else printf("Autre probleme \n");
```

5.10. Tri fusion sur fichiers

5.10.1. Tri externe :

- ✚ Pendant le tri, seule une partie des données se trouve en mémoire vive, le reste est stocké sur disque ou sur bande magnétique
- ✚ Accéder aux données est donc beaucoup plus coûteux que de les comparer ou de faire des opérations arithmétiques dessus
- ✚ Il peut y avoir des restrictions fortes sur l'accès

Critère 1 : minimiser le nombre de fois où l'on accède à un élément

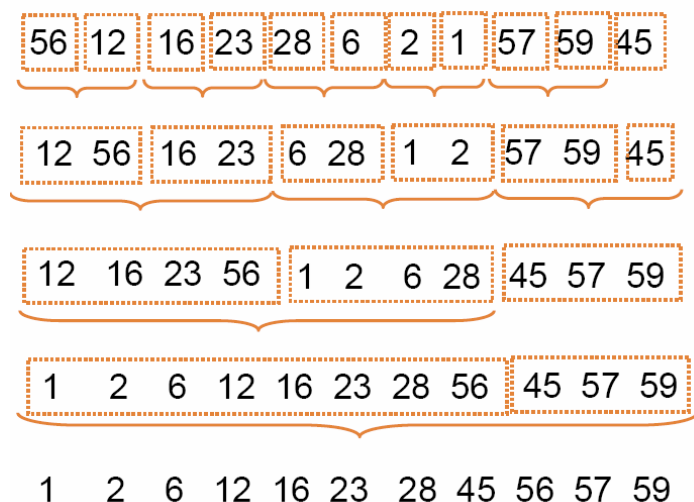
Critère 2 : accéder séquentiellement aux éléments

5.10.2. Notion de monotonie

- ✚ Une monotonie est une sous séquence d'éléments triés
- ✚ Toute sous séquence de taille 1 est trivialement une monotonie

Exemple : 56 12 | 16 23 | 28 6 | 2 1 | 57 59 | 45 ➔ 5 monotonies de longueur 2 et une de 1.

5.10.3. Principe du tri par fusion



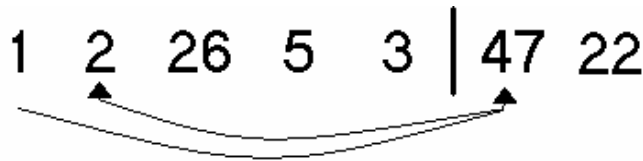
Approche **bottom-up** (utilisée en tri externe) :

Fusionner les paires d'éléments successifs pour obtenir une séquence de $n/2$ monotonies de longueur 2,

Fusionner les paires de monotonies successives pour obtenir une séquence de $n/4$ monotonies de longueur 4,

Continuer jusqu'à obtenir une seule monotonie de longueur N .

Il n'est pas conseillé de réaliser le tri fusion avec seulement 2 fichiers, pourquoi ?!



Avec l'exemple ci-contre, on voit que la tête de lecture va faire beaucoup de voyage, on est plus en séquentielle. On le fera plutôt avec 3 fichiers

5.10.4. Avec 3 fichiers (à connaître)


Phase d'éclatement :

On répartit les monotonies contenues dans le fichier X dans deux fichiers :

- 1 sur 2 va dans le fichier A
- 1 sur 2 va dans le fichier B

Phase de fusion :

- On fusionne la 1^{ère} monotonie du fichier A avec la 1^{ère} monotonie du fichier B, en écrivant la monotonie résultante dans le fichier X (on écrase l'ancien contenu du fichier X),
- Idem avec les 2^{ème} monotonies des fichiers A et B,
- Et ainsi de suite jusqu'à la fin des fichiers A et B.

-  On recommence la phase d'éclatement du fichier X, la nouvelle longueur d'une monotonie sera 2 fois l'ancienne...etc.

Idée = tenir compte des **monotonies naturelles**, c'est à dire des monotonies maximales du fichier initial.

En résumé : jusqu'à avoir une monotonie de 1 (toute la liste)

1- éclatement : les dernières monotonies ne sont pas forcément uniformes dans A et B

2- fusion : attention, l'un des fichiers peut être épuisé avant l'autre selon l'éclatement effectué !

3- longueur monotonie = 2^* longueur monotonie

4- refaire 1,2,3,4.

6. Chapitre : Les types de données abstraits (TDA) et programmation séparée

6.1. Génie logiciel

- Maîtriser la complexité : Diviser pour mieux régner en modules indépendants
 - Si le module est juste on pourra alors le modifier à tout moment
- Utiliser des bibliothèques déjà existantes.

6.2. Types de données abstraits : TDA

Un TDA est donc :

- un type de données
- doté d'un ensemble d'opérations
- dont les détails d'implémentation restent cachés (abstraction)

Les TDA sont les briques de base d'une conception modulaire (un .c + .h).

Un module regroupe des définitions :

- de constantes,
- de variables globales,
- de types,
- de procédures et de fonctions qui permettent de manipuler ces types.

On sépare en 2 ou plusieurs fichiers :

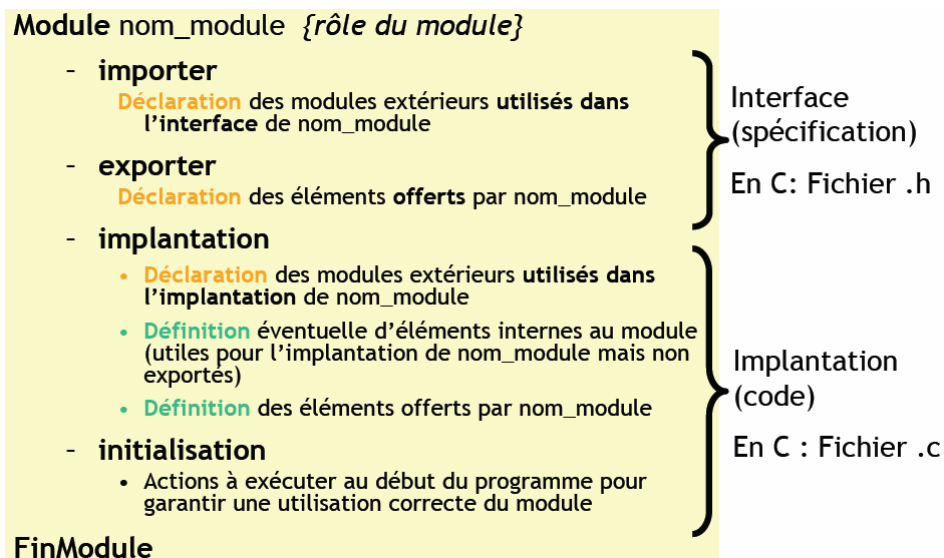
- l'interface : spécification, déclarations, entêtes
- Implémentation : définitions, code.

Par exemple, pour l'addition : il n'est pas nécessaire de connaître le programme pour l'utiliser mais il faut connaître les préconditions.

Quand on veut créer des TDA, il faut avoir :

- une vision interne : représentation, utilisation, implémentation !
- une vision externe (l'utilisateur) : service, besoin, performances !

6.3. Norme Algorithmique



6.4. Exemple avec des nombres complexes (Algorithme)

Module NombreComplexe

- exporter

Variable compteurComplexes : entier {nombre de complexes initialisés}

Type Complexe

Procédure initialiser(c : Complexe, a : réel, b : réel)

Préconditions : ...

Postconditions : $c = a + ib$, et compteurComplexes incrémenté

Paramètres en mode donnée : a, b

Paramètre en mode résultat : c

Procédure testament (c : Complexe)

Préconditions : ...

Postconditions : compteurComplexes décrémenté

Paramètre en mode donnée-résultat : c

Procédure affectation (c1 : Complexe, c2 : Complexe)

Préconditions : c1 initialisé

Postcondition : c1 prend la même valeur que c2, compteurComplexe inchangé

Paramètre en mode donnée : c2

Paramètre en mode donnée-résultat : c1

Fonction addition (c1 : Complexe, c2 : Complexe) : Complexe

Préconditions : aucune

Postcondition : aucune (compteurComplexe inchangé)

Résultat : retourne la somme de c1 et c2

Paramètre en mode donnée : c1, c2

- implantation

Variable compteurComplexes : entier

Type Complexe = structure

x, y : réels

Fin structure Complexe

Procédure initialiser(c : Complexe, a : réel, b : réel)

Début

c.x \leftarrow a

c.y \leftarrow b

compteurComplexes \leftarrow compteurComplexes + 1

Fin initialiser

...

Procédure testament (c : Complexe)

Début

compteurComplexes \leftarrow compteurComplexes - 1

Fin testament

Procédure affectation (c1 : Complexe, c2 : Complexe)

Début

c1.x \leftarrow c2.x

c1.y \leftarrow c2.y

Fin affectation

Fonction addition (c1 : Complexe, c2 : Complexe) : Complexe

Variables locales : res : Complexe

Début

res.x \leftarrow c1.x + c2.x

res.y \leftarrow c1.y + c2.y

Retourner res

Fin addition

- initialisation

compteurComplexe \leftarrow 0

FinModule NombreComplexe

6.5. Le programme du côté utilisateur

```
Importer
  module NombreComplexe

Début
  monComp1, monComp2, maSomme : Complexe

  initialiser(monComp1, 2.5, 8.0)
  initialiser(monComp2, 1.0, 0.0)
  initialiser(maSomme, 0.0, 0.0)
  afficher(« Nb de complexes = », compteurComplexes)
  affectation(maSomme, addition(monComp1, monComp2))
  testament(monComp1)
  testament(monComp2)
  testament(maSomme)
  afficher(« Nb de complexes = », compteurComplexes)
Fin
```

6.6. Intérêt de séparer interface et implantation

- ✚ On peut aussi choisir de représenter les complexes en coordonnées polaires (module, argument)
- ✚ On ne change que la partie « implantation »
- ✚ La partie « exporter » (de la partie interface) est inchangée, donc les utilisateurs du module n'ont pas besoin de changer leurs codes
- ✚ On peut faire 2 modules avec les mêmes interfaces mais des implantations différentes
 - partie réelle, partie imaginaire : plus efficace pour les sommes
 - module, argument : plus efficace pour les produits
 - l'utilisateur choisit le module qui lui convient le mieux en fonction des calculs qu'il doit faire le plus souvent

6.7. Norme en C/C++

Fichier .h (fichier d'entêtes) = fichier de promesses !

- contient l'équivalent des parties « importer » et « exporter »
- et contient aussi malheureusement les définitions de type

Fichier .c (fichier source) = mise en oeuvre :

- contient l'équivalent des parties « implantations » et « initialisation »
- ... sauf les définitions de type

L'utilisateur du module écrit son « main » dans un troisième fichier, il n'a besoin de regarder que le .h pour pouvoir utiliser le module.

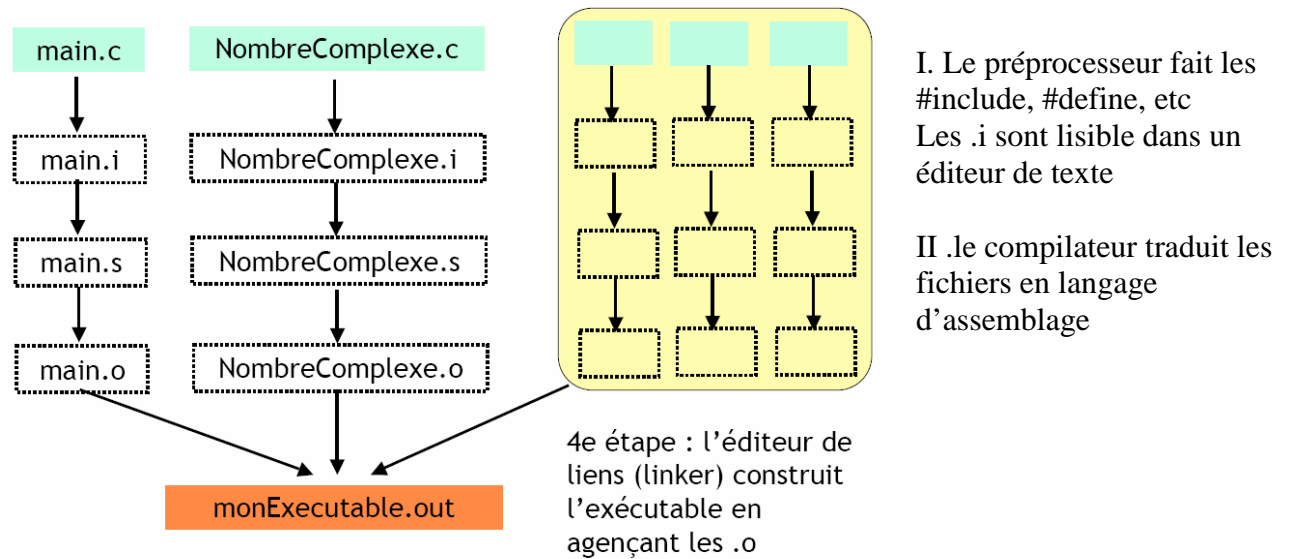
6.8. Mot-clé *static* : (utile pour TP sur les arbres)

- ✚ Permet de définir des éléments internes à un module
 - non exportés
 - non visibles depuis les programmes utilisateurs

Les éléments « static » peuvent être des variables, des fonctions ou des procédures.

6.9. Compiler un programme réparti sur plusieurs fichiers

Solution : `#include "NombreComplexes.h"` au début de `main.c`



`#` → Directive de compilation : instruction prise en compte par le compilateur

Exemple : `#ifndef _ TabDyn` : si l'identificateur `TabDyn` n'est pas définie on le définit

Quelle est la différence entre `#include<stdio.h>` et `#include "element.h"` ?

Avec les `<>` : inclusion d'un fichier system du compilateur

Avec les doubles guillemet : inclusion d'un fichier présent sur le disque dure.

La commande de compilation : (`-c` : ne pas faire l'exécutable, s'arrêter au `.o`)

`g++ -c -Wall -ansi -pedantic main.c`

Edition des liens :

`g++ main.o NombreComplexe.o -o monExecutable.out` → crée « `monExecutable.out` »

Pour simplifier tout ça, on va créer des **Makefile**, en voici un exemple :

Nom du fichier : **makefile** simplement et sans extension.

```
monExecutable.out: main.o NombreComplexe.o
    g++ main.o NombreComplexe.o -o monExecutable

main.o: main.c NombreComplexe.h
    g++ -c -Wall -ansi -pedantic main.c

NombreComplexe.o: NombreComplexe.c NombreComplexe.h
    g++ -c -Wall -ansi -pedantic NombreComplexe.c
```

Important :

On met comme première ligne celle qui va créer le `.out` sinon ça ne fonctionne pas. Puis dans le terminal on tape **make** (il faut bien sûr ce trouver dans le dossier qui contient les `.h`, `.c` et le `makefile`).

7. Chapitre : Algorithmique et structures de données

7.1. Les Types de Données Abstraits

- 1) Tableau Dynamique
- 2) Liste
- 3) File
- 4) Pile
- 5) Arbre

Type à accès direct : on ne s'occupe pas de la taille du tableau.

- Il faut pouvoir augmenter la taille à loisir
- la gestion de la taille ne doit pas être visible par l'utilisateur

Exemple 1 :

Recherche par dichotomie dans un tableau :

1. Séparer le tableau en deux
2. Si l'élément recherché est inférieur au dernier élément du 1er sous tableau alors rechercher dans le 1^{er} sous tableau sinon recherche dans le 2^{ème}.

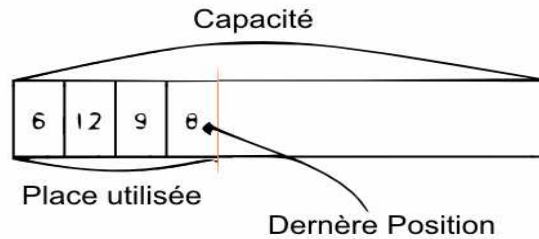
Exemple 2 :

Estimation du coût des fonctions suivantes pour un tableau dynamique de taille N :

AjouteElement :	Constant,
InsereElement :	Linéaire, $K*N$,
SupprimeElement :	Constant si on choisie un élément,
AccesElement :	Si on à l'indice : constant, sinon linéaire $K*N$,
ModifieElement :	Si on à l'indice : constant, sinon linéaire $K*N$.

7.2. Première solution : Structure du Tableau Dynamique en c, implémentation

Représentation imagée :



Dans le element.H (element.h) :

```
#ifndef _ELEMENT
#define _ELEMENT
typedef double Element;
#endif
```

/* **Element** sera de type double, **double mavariable = Element mavariable**. Comme ça, on peut modifier le type de Element sans avoir à le faire pour tous les sous-fichiers. */

Dans TableauDynamique.h :

```
#ifndef __TabDyn
#define __TabDyn
#include element.H // On inclus le header pour avoir le type de 'Element'

struct Tab_Dynamique
{
    Element *ad; // Équivaut à double *ad; (dans notre cas). On aura dans t.ad[i]= valeur.
    int capacite; // La capacité totale du tableau dynamique
    int derniere_position; // La position de la prochaine valeur
};

Void initialiser(Tab_Dynamique &t, int taille); /* initialisation du tableau */

#endif
```

Dans le .c qui regroupe les fonctions :

```
#include <TableauDynamique.h>
void initialiser(Tab_Dynamique &t, int taille)
{
    t.capacite = taille;
    t.ad = new Element[taille]; // on aura l'adresse du premier élément dans t.ad
    t.derniere_position = 1;
}
```

ATTENTION :

1- les structure ne sont pas des donné/résultat, il faut mettre un et commercial,
Exemple : void initialiser(Tab_Dynamique &t, int taille).

2- Si on a t.derniereposition = t.capacité alors on va créer un autre tableau dans le tas qui aura t.capacite= 2*t.capacite. On recopiera chaque valeur de l'ancien tableau dans le nouveau et on détruira l'ancien.

7.3. Deuxième solution : Liste d'éléments (type abstrait)

Qu'est-ce qu'une liste chaînée ?

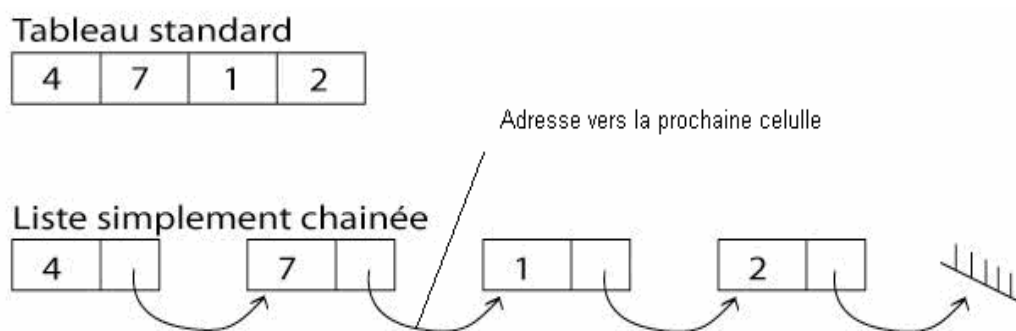
C'est un système informatique qui permet la **sauvegarde dynamique de données en mémoire** sans se préoccuper de leur nombre et en rendant leur allocation plus transparente.

On dit liste chaînée car les données sont chaînées les unes avec les autres. On **accède aux données à l'aide d'un ou deux points d'entrées** qui se situent la plus part du temps aux **extrémités de la liste**. Dans la pratique ces points d'entrées seront des **pointeurs soit sur le premier ou le dernier élément de la liste voir sur les deux ou même mobile**.

Les éléments de la liste sont chaînés entre eux à l'aide de pointeurs **sur leurs éléments suivants ou précédents voir sur les deux**. Ces pointeurs doivent donc faire partie de l'élément. Cette structure aura donc la particularité d'avoir au moins un pointeur sur des variables du même type qu'elle. Ce pointeur servira à **relier les éléments de la liste entre eux**. La structure contiendra bien sûr aussi les données que l'on veut mémoriser.

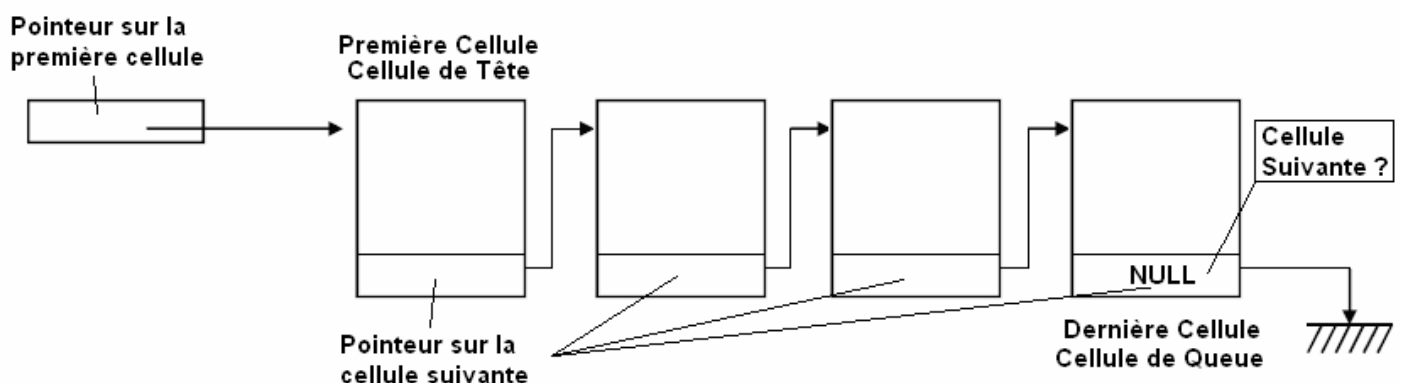
7.4. Représentation graphique d'une liste chaînée (structure à accès séquentiel) :

Exemple : représentation simplifiée



La liste est donc chaînée grâce à l'utilisation des adresses.

Exemple : Bonne représentation, plus complexe




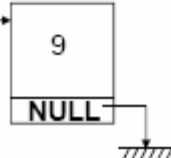
7.5. Implémentations

7.5.1. Création d'une liste chaînée




7.5.1.1. Introduction

On va prendre la liste {9, 15, 12, 5}

Dans ce qui va suivre, l.tete est l'adresse de la cellule en tête de liste.

Langage algorithmique		C/C++	
Cellule = Structure info : Entier suivant : pointeur sur Cellule Fin Cellule		<pre>struct Cellule { int info ; Cellule * suivant ; };</pre>	// La structure Cellule est créée
I : Liste	I.tete ?	Liste I;	// On crée la liste
I.tete ← reserve Cellule	tete 123451 → 	I.tete = new Cellule;	// On crée une Cellule dans le tas, I.tete (dans la pile) = adresse de cette nouvelle cellule
I.tete↑.info ← 9 I.tete↑.suivant ← nil	tete 123451 → 	<pre>(*I.tete).info = 9 ; (*I.tete).suivant = NULL;</pre>	// 9 dans la case // NULL dans l'adresse car il n'existe pas d'autres cellules avant 9

Pour ajouter une cellule, il existe 3 méthodes :

-  Chaînage en tête : on à déjà l'adresse de départ
-  Chaînage en queue : il nous faut la dernière adresse
-  Chaînage à l'intérieur

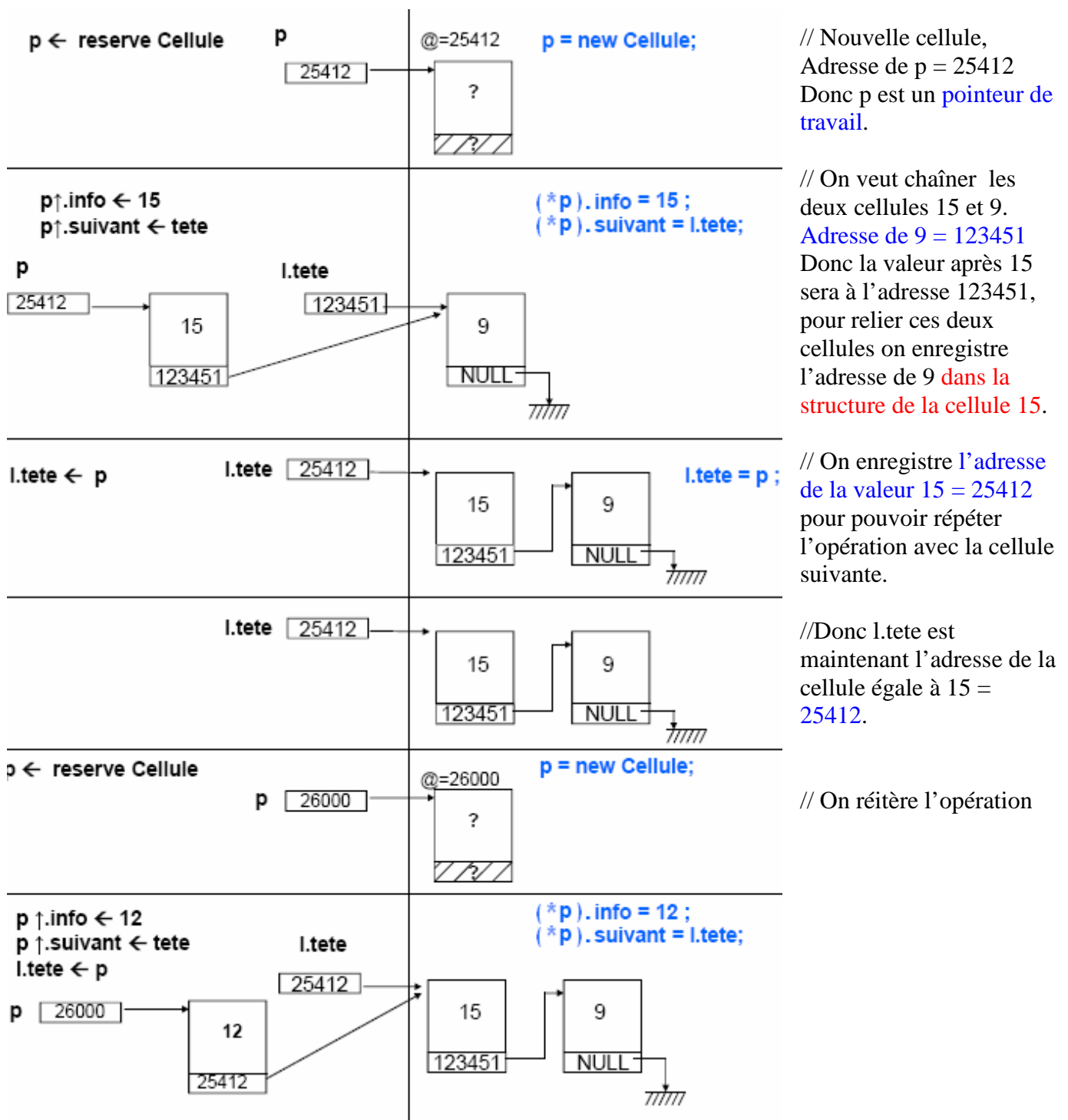
Lorsqu'on manipule les listes, on a besoin d'un pointeur de travail (nommé p dans le cours) qui va permettre de créer les cellules dans le tas, exemple : p= new Cellule donc p est une adresse.

Attention : Les structures ne sont pas par défaut des données/résultats ! Il faudra donc utiliser un Et Commercial "&" pour le préciser. De plus, pour éviter que les listes ne soient modifiées, on mettra "const" (pour constante) dans certaines fonctions.

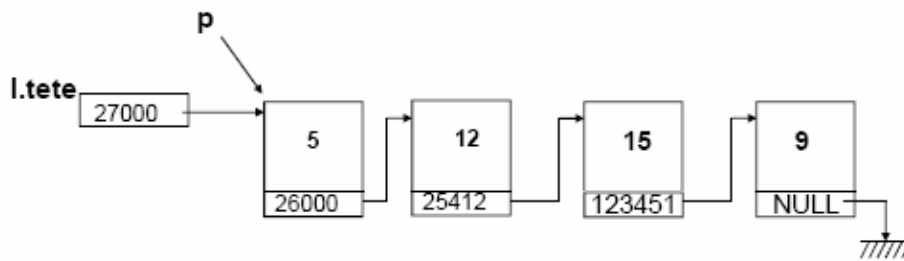
Forme Algorithmique : p↑ fg ou p→ fg (on met → car on a pas de ↑ sur l'ordinateur)
Forme en C/C++ : (*p).fg

7.5.2. Chaînage en tête (suite de l'image précédente)

7.5.2.1. Création



Après plusieurs ajouts, on aura :



Donc pour récupérer les valeurs on a :

Début	→	p = l.tete = 27000	→	(*p).info = 5, (*p).suivant = 26000
p = 26000	→	(*p).info = 12		
p = 25412	→	(*p).info = 15		
p = 123451	→	(*p).info = 9	→	(*p).suivant = NULL → fin

7.5.3. Destruction d'une liste

On va supprimer chaque cellule sans perdre l'adresse de la cellule suivante. Cellule *p = l.tete, p est de type Cellule, on a donc associé à p la structure cellule.

→ $P \leftarrow l.tete$ ①

1- On enregistre la dernière adresse dans p, pour pouvoir effacer le bloc :
Exemple avec la liste chaînée précédente: p = 27000 et ici (*p).info=5.

→ $l.tete \leftarrow (l.tete \uparrow .suivant)$ ②

2- Avant d'effacer la cellule on va sauvegarder l'adresse de la valeur suivante. Ce sera la nouvelle cellule de tête. Exemple avec la liste chaînée précédente : $l.tete \leftarrow (*p).suivant$ (= 26000)

→ libere p ③

3- On a bien l'adresse de la valeur suivante dans l.tete et p correspond à l'adresse de la cellule que l'on souhaite effacer. Donc on peut libérer p !
4- On réitère le processus tant que l.tete n'est pas NULL.

7.5.3.1. Implémentation

Algorithme :

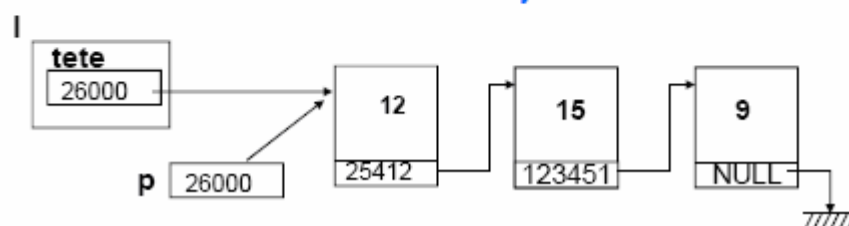
```

tantque l.tete <> null faire
  p ← l.tete
  l.tete ← l.tete↑.suivant
  libere p
fin tantque
  
```

Traduction en C/C++ :

```

while(l.tete != NULL)
{
  p = l.tete;
  l.tete = (*l.tete).suivant;
  delete p;
}
  
```

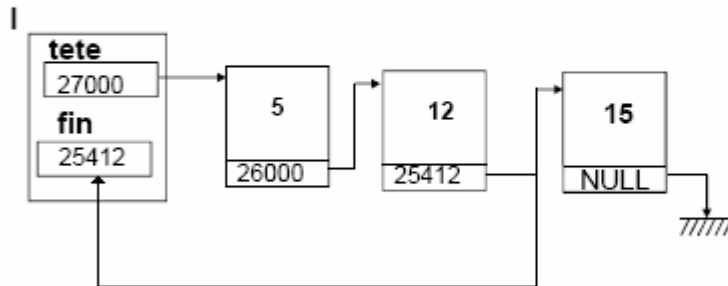


7.5.4. Chaînage en queue

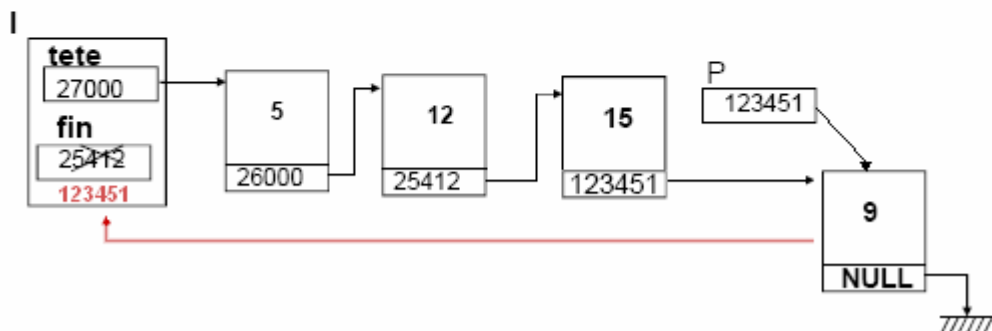
7.5.4.1. Création

utilisation de trois pointeurs

- ➔ tete
- ➔ fin
- ➔ p

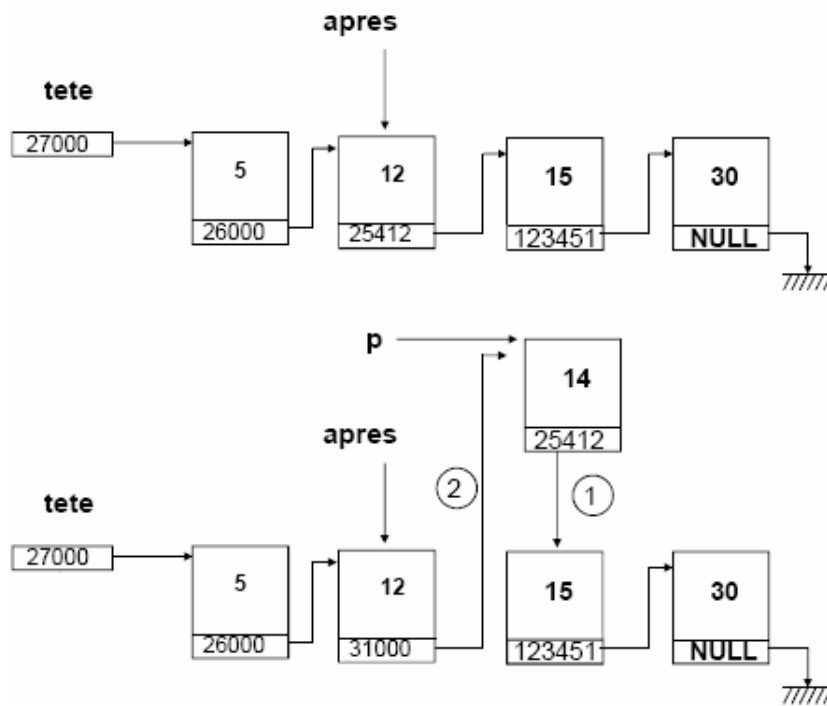


- ➔ $p \leftarrow \text{reserve Cellule}$
- ➔ $p \uparrow .\text{info} \leftarrow 9$
- ➔ $p \uparrow .\text{suivant} \leftarrow \text{null}$
- ➔ $(I.\text{fin}) \uparrow .\text{suivant} \leftarrow p$
- ➔ $I.\text{fin} \leftarrow p$



On n'est pas obligé d'avoir un pointeur de fin ! Dans ce cas, lors d'un ajout en queue (ici après 15), on va devoir parcourir la liste pour connaître l'adresse de la dernière cellule et ensuite ajouter l'élément.

7.5.4.2. Insertion



Pour l'insertion d'une nouvelle cellule, il suffit d'inverser les pointeurs :

La structure qui contient 12 aura son pointeur sur 14 et la structure qui contient 14 aura son pointeur sur 15. **Plus besoin de tout décaler comme on le faisait avec les tableaux.**

$p \rightarrow \text{suivant} = \text{apres} \rightarrow \text{suivant};$

$\text{apres} \rightarrow \text{suivant} = p;$

7.5.4.3. Destruction

Même procédé que le chaînage en tête.

7.5.5. Exemple de codes

On a choisi ici, d'avoir le nombre d'éléments, on aura donc une structure contenant nb_element. On va implémenter des fonctions de chaînage en Tête et en Queue.

Liste.h

```
struct sCellule {
    elem info;
    sCellule *suivant;
};
typedef struct sCellule Cellule;

struct sListe {
    Cellule *prem; /* Soit l une liste, on a : (*l.prem).info et (*l.prem).suivant , l.prem est l.tete */
    int nb_element;
};

typedef struct sListe Liste;
/* on a un pointeur sur la structure d'une cellule + notre nombre d'éléments */
Il faut rajouter les définitions de toutes les fonctions du fichier fonction.c.
```

Fonction .c

```
#include <stdio.h>
```

```
#include "liste.h"
```

```
#include "element.h"
```

```
void initialisation (Liste &l){ l.prem=NULL ; l.nb_element=0; }
```

```
/* Lors de l'initialisation on a pas encore créé de cellule donc on a pas d'adresse s'y rendre */
```

```
void testament(Liste &l){ videListe(l);}
```

```
bool estvide(const Liste &l){
```

```
    if(l.prem == NULL) /* l.prem = NULL donc *l.prem.info n'existe pas alors la liste est vide) */
```

```
        return true;
```

```
    else return false;}
```

```
elem premierElem (const Liste &l){return (*l.prem).info; }
```

```
/* on accède au contenu de la cellule tête par (*l.prem), puis à la variable info avec (*l.prem).info */
```

```
Cellule *adresseCelluleTete(const Liste &l){return l.prem;}
```

```
Cellule *celluleSuivante(const Liste &l, Cellule *c){return (*c).suivant;}
```

```
unsigned int getNbElements(const Liste &l){return l.nb_element;}
```

```
void afficheElement(const elem e){printf("%f \n", e);}
```

```
void afficherListe(const Liste &l){
```

```
    Cellule *c=l.prem; /* on associe l'adresse pointée par l.prem à l'adresse d'une cellule */
```

```
    while(c != NULL){
```

```
        afficheElement((*c).info);
```

```
        c=(*c).suivant; /* la variable c prend l'adresse de la prochaine cellule */
```

```
    }
```

```
}
```

```
void affichageListeDepuisCellule(const Liste &l, const Cellule *c){
```

```
    if(c != NULL)
```

```
        afficheElement((*c).info);
```

```
        affichageListeDepuisCellule(l,(*c).suivant);
```

```
}
```

```
/* version récursive */
```

```
void afficherListeRec(const Liste &l){
```

```
    printf("Liste \n");
```

```
    affichageListeDepuisCellule(l, l.prem);
```

```
    printf("Fin de la liste \n");
```

```
}
```

```
void ajoutEnTete(Liste &l, const elem e){
```

```
    Cellule *c= new Cellule;
```

```
    (*c).info=e;
```

```
    (*c).suivant=l.prem;
```

```
    l.prem=c; /* l'élément de tête devient la nouvelle cellule */
```

```
    l.nb_element++;
```

```
}
```

```

void supprimerEnTete(Liste &l){
    Cellule *c=l.prem;
    l.prem=(*c).suivant; /* on sauvegarde l'adresse de la cellule suivante */
    delete (c);          /* puis on efface la cellule actuelle */
    l.nb_element--;
}

void videListe(Liste &l){
    while(estvide(l) == false){
        supprimerEnTete(l);
        videListe(l);
    }
}

void ajoutEnQueueConnaissantCellule(const elem &e, Liste &l, Cellule *c){
    Cellule *tmp=c;
    while((*tmp).suivant != NULL){tmp=(*tmp).suivant;}
    (*tmp).suivant=new Cellule; /* on crée une nouvelle cellule */
    ((*tmp).suivant).info=e;    /* on y met e comme information */
    ((*tmp).suivant).suivant= NULL; /* et on met NULL dans .suivant pour l'adresse de la cellule
                                     suivante, car on chaine en queue */
    l.nb_element++;             /* on incrémente le nombre d'éléments de 1 */
}

void ajoutEnQueue(const elem &e, Liste &l){
    if(estvide(l))
        ajoutEnTete(l,e);      /* Cellule *c=l.prem => (*c).info=e et (*c).suivant= NULL */
    else
        ajoutEnQueueConnaissantCellule(e,l,l.prem); /* on ajoute la cellule en queue */
}

```

Fichier main.c

```

#include <stdio.h>
#include "fonctions.c"

int main(){
    Liste EnQueue; /* Déclaration de la liste */

    initialisation(EnQueue);
    afficherListe(EnQueue);
    supprimerEnTete(EnQueue);
    int nb_queue=getNbElements(EnQueue);
    printf("Nombre d'elements : %d\n", nb_queue);

    Liste EnTete; /* Déclaration de la liste */

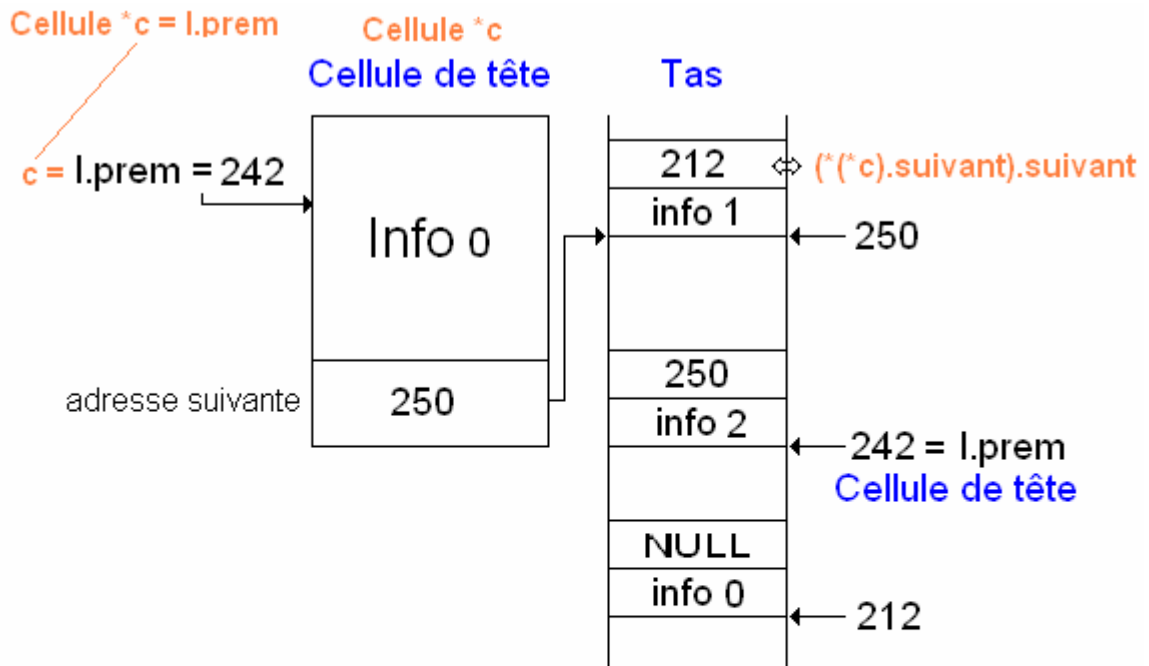
    initialisation(EnTete);
    ajoutEnTete(EnTete,4);

    testament(EnTete); /* ne pas oublier de supprimer les listes */
    testament(EnQueue);

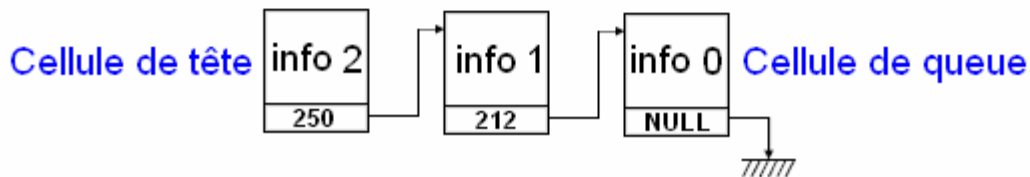
}

```

7.5.6. Bilan du chaînage



Equivaut à la liste chaînée



Attention : Quand on utilise "new Cellule" on va créer une cellule dans le tas. Comme on le voit sur le schéma ci-dessus, les cellules ne sont pas positionnées dans l'ordre, c'est la RAM qui gère cela.

« La structure cellule » contient une valeur enregistrée et un pointeur sur la valeur suivante.

Si cette «structure cellule» est la dernière de la liste, alors le pointeur est NULL (elle est en Queue).

l.tete qui est contenue dans la pile = l'adresse de la cellule précédente

p qui est contenue dans la pile = l'adresse de la cellule actuelle

Si la liste est vide ?!

Queue : attention on aura une case vide et un pointeur = NULL

Tête : pas de problème

On a la liste {9, 15, 12, 5} avec un chaînage en :

Queue : on obtient la liste {9, 15, 12, 5} → bon ordre

Tête : on obtient la liste {5, 12, 15, 9} → ordre inversé !

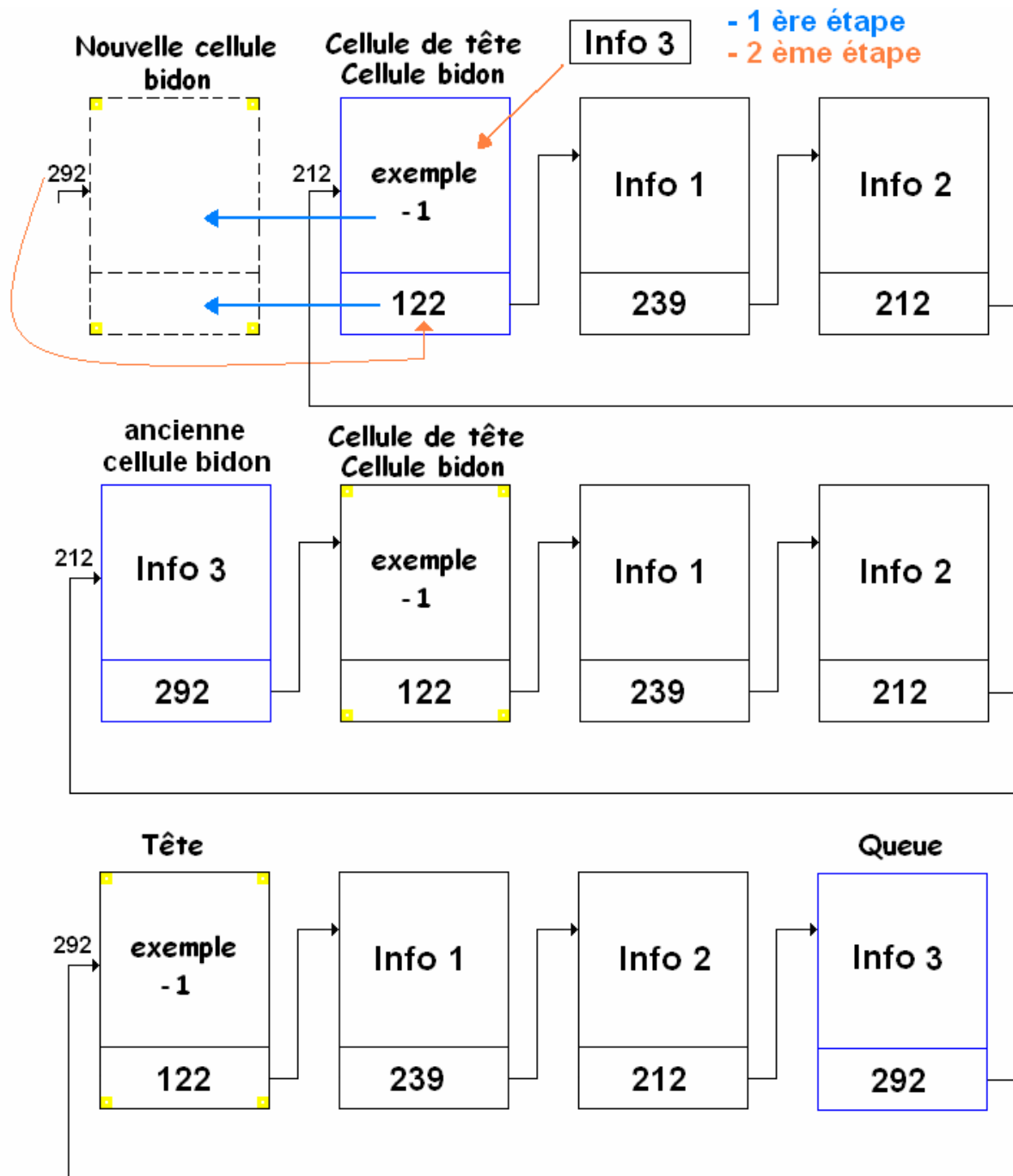
Coût des opérations

	Ajout en tete	Ajout en fin	insertion	Recherche
Listes chaînées	constant	?	linéaire	linéaire

7.5.7. Liste chaînée circulaire

Comme l'indique le nom on va chaîner en cercle, donc au lieu de mettre NULL dans la cellule en Queue, on va chaîner sur la cellule de tête.

Cas particulier : Dans une liste avec un seul élément, le 1er est en même temps le dernier.



```
Cellule *nouvelle = new Cellule // on crée une nouvelle cellule bidon, nouvelle bidon = 292
Cellule *bidon = l.prem // on se raccroche à l'ancienne cellule bidon, bidon = 212
(*nouvelle).info = (*bidon).info // on récupère l'info bidon (histoire d'avoir un élément, ici : -1 )
(*bidon).info = info 3 // on insère l'info 3 dans l'ancienne cellule bidon
(*nouvelle).suivant = (*bidon).suivant // adresse de la cellule suivante dans la nouvelle cellule bidon
(*bidon).suivante = l.prem // adresse de la nouvelle cellule bidon dans l'ancienne
l.prem = bidon; // la cellule de tête devient la nouvelle cellule bidon
```

Plus besoin de parcourir toute la liste pour ajouter un élément en queue !

Il faudra avoir une autre structure pour les chaînes circulaires:

```
typedef struct cListe {  
    element *debut;  
    element *fin; /* pour ne pas boucler à l'infini, ex: afficher la liste */  
    int taille;  
};  
  
typedef cListe Liste;
```

7.6. Makefile

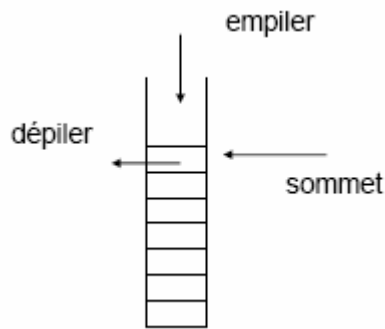
On peut utiliser des variables dans le makefile.

```
OPTIONS = -Wall -ansi -pedantic  
OBJETS = f1.o f2.o  
all : toto  
f1.o : f1.c f1.h  
    g++ $(OPTIONS) -c f1.c  
f2.o : f2.c f2.h f1.h  
    g++ $(OPTIONS) -c f2.c  
toto : $(OBJETS)  
    g++ $(OBJETS) -o toto  
clean:  
    rm *.o
```

Puis taper « **make all** » pour compiler. Si vous ne mettez pas « **all : toto** » alors tapez simplement « **make** » dans le terminal (juste ce qui est en bleu).

8. Chapitre : TDA Pile et File

8.1. TDA Pile



Opérations sur la pile :

Empiler (p: pile, e: Element)

Depiler (p: pile)

Consulter_sommet (p: pile)

Pile_vide (p: pile)

Implantation, deux possibilités :

✚ Utilisation de tableau dynamique

✚ Utilisation de listes chaînées

Exemple : Inverser les éléments d'une file, on va stocker les éléments de la file dans une pile et les réinsérer en ordre inverse (chaînage en Tête pour inverser les éléments).

2	4	3	5
5	3	4	2

Avant

Après

Element Fonction Consulter_sommet (f : file)

Pré condition : f initialisé

Post condition : f inversé

Paramètre : donné résultat

Variable local : p:pile

Début

Initialiser (p)

Tant Que (non (estvide (f)) faire

Empiler (p, consulterTette (f))

Defiler (p)

Fin Tant Que

Tant Que (non (estvide (p)) faire

Emfiler (f, consulterSommet (p))

Depiler (p)

Fin Tant Que

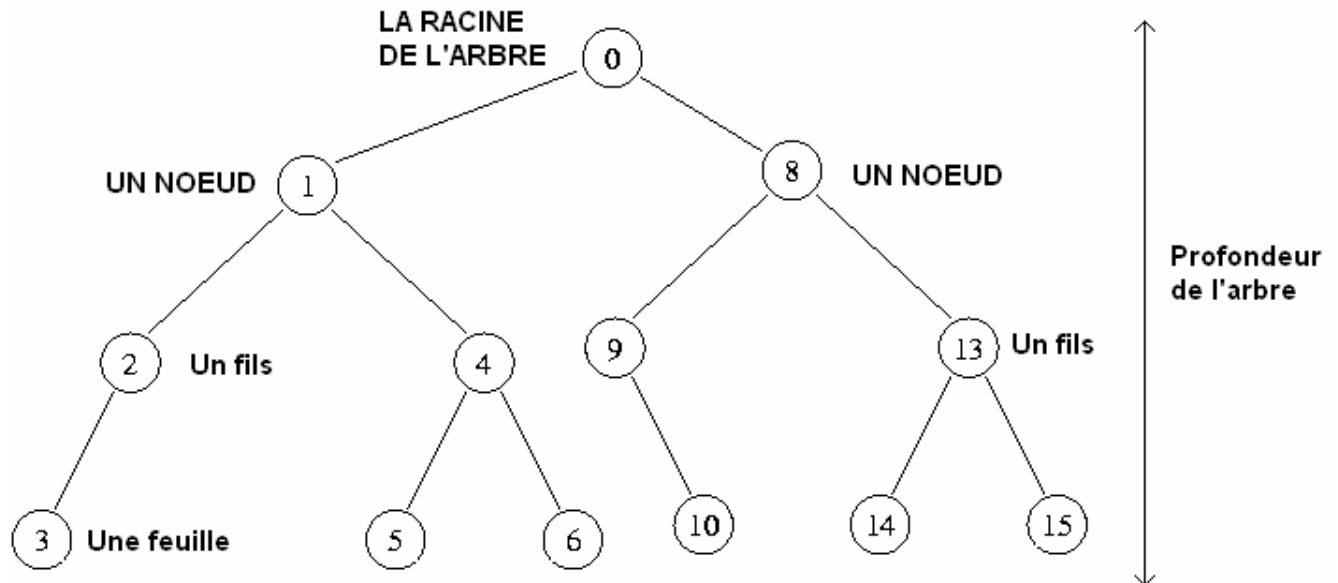
Testament (p)

Fin

Dans une première partie, on remplit la pile et dans une deuxième partie on remplit la file.

8.2. TDA File

8.2.1. Rappel sur les Arbres



Vocabulaire :

Les **nœuds** internes, éléments possédant des fils (sous-branches).

La **racine** de l'arbre est le nœud ne possédant pas de parent.

Les **feuilles**, éléments ne possédant pas de fils dans l'arbre.

La **hauteur/profondeur** d'un arbre est la longueur du plus grand chemin de la racine à une feuille.

Le **degré**, est le nombre de fils d'un nœud.

Wiki : Chaque nœud possède une **étiquette**, qui est en quelque sorte le « contenu » de l'arbre. L'étiquette peut être très simple: un nombre entier, par exemple. Elle peut également être aussi complexe que l'on veut : un objet, une instance d'une structure de données, un pointeur, etc. Il est presque toujours obligatoire de pouvoir comparer les étiquettes selon une relation d'ordre total, afin d'implanter les algorithmes sur les arbres.

8.2.2. Les arbres binaires

On dit qu'un arbre est binaire, si chaque nœud a **au plus deux fils** (donc de degré max = 2).

Les structures associées (Algorithme)

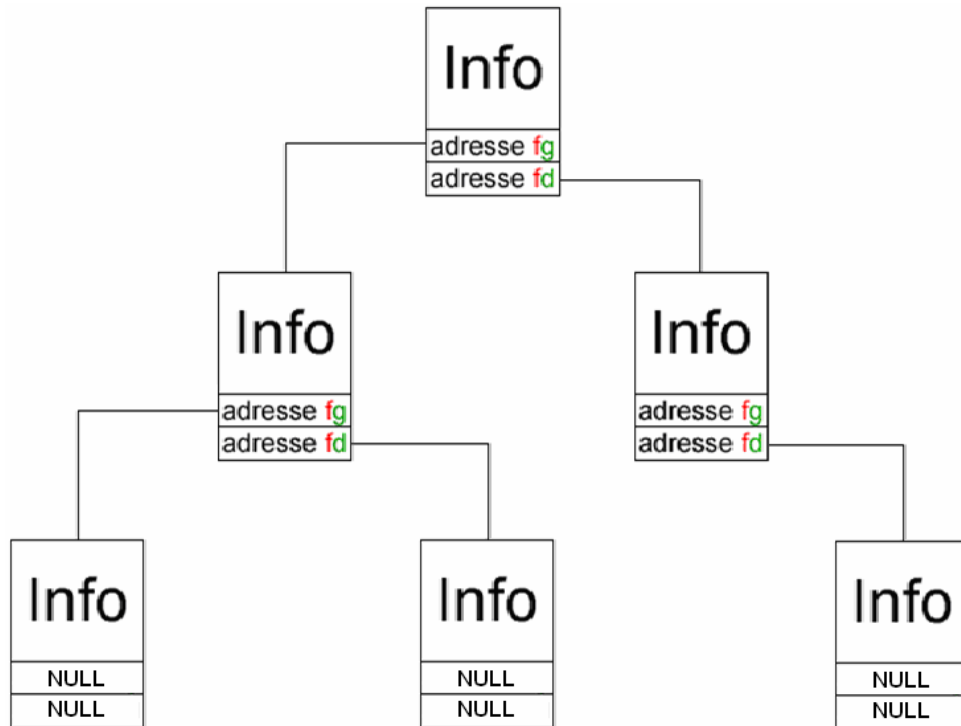
```
Cellule = Enregistrement
    Info : Element
    Pages File d'entiers
    fg, fd pointeur sur cellule
Fin Cellule

ArbreBin = Enregistrement
    Cellule *Racine
Fin arbrebin
```

fg = fils gauche.

fd = fils droit.

8.2.3. Arbre binaire avec les listes chaînées



8.2.3.1. Exemple de programme

En général **on utilise la récursivité** pour les arbres !

Fonction_Copie (Arbre &a1, const Arbre &a2, Cellule *c1, Cellule *c2)

Pré condition : l'arbre a2 n'est pas vide, ce qui équivaut à dire que Racine != NULL , attention les deux arbres doivent être indépendant l'un de l'autre.

Post condition : l'arbre a1 est une copie de l'arbre a2

Si c2 n'est pas NULL {

 Si c1 est NULL {

 On l'initialise avec une fonction init(a1)

 Racine = adresse d'une nouvelle cellule (qui sera la racine)

 c1 = Racine, la première cellule à remplir sera la racine. }

 On copie l'info c2.info dans la cellule actuelle c1.info

 Puis on teste si le fils gauche de a2 est vide ((*c2).fg == NULL) ?

 Oui { Alors (*c1).fg = NULL, le fils gauche n'a pas d'adresse }

 Non { Pour a1 → (*c1).fg = adresse d'une nouvelle cellule (new Cellule)

 Fonction_Copie (a1,a2, adresse du fg de a1, adresse du fg de a2)

 Puis on teste si le fils droit de a2 est vide ((*c2).fd == NULL) ?

 Oui { Alors (*c1).fd = NULL, le fils droit n'a pas d'adresse }

 Non { Pour a1 → (*c1).fd = adresse d'une nouvelle cellule (new Cellule)

 Fonction_Copie (a1,a2, adresse du fd de a1, adresse du fd de a2) }

Teste : Fonction_Copie (arbre1, arbre2, NULL, ArbreBin.Racine)

Au final si on a une feuille, les deux testes sur le fils gauche et droit mettrons NULL dans l'adresse du fils gauche et droit du nouvel arbre.

8.2.4. Trois façons de parcourir un arbre

ssa = sous arbre;

1. Préfixé **Racine**, ssaG, ssaD
2. Infixé ssaG, **Racine**, ssaD
3. Postfixé ssaG, ssaD, **Racine**

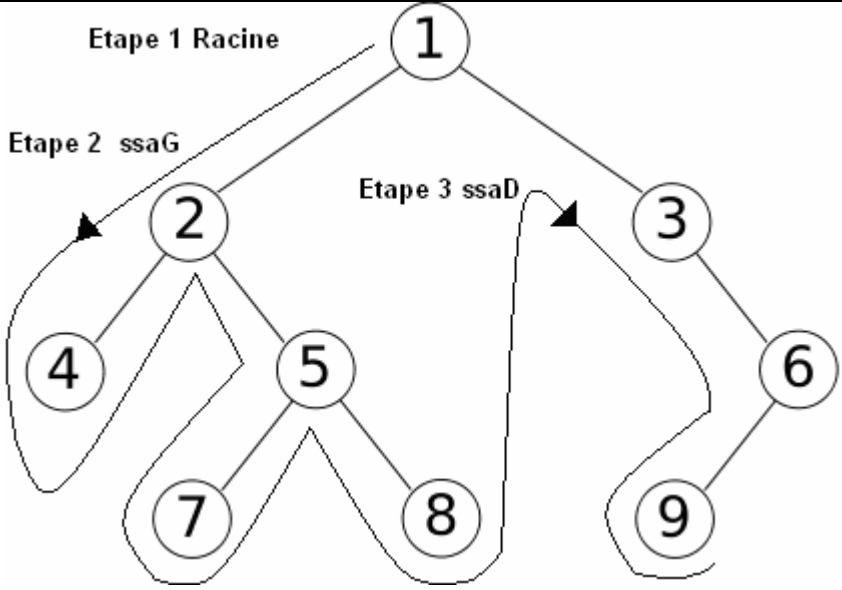
Souvent, il est souhaitable de visiter chacun des nœuds dans un arbre et d'y examiner la valeur. Il existe plusieurs ordres dans lesquels les nœuds peuvent être visités, et chacun a des propriétés utiles qui sont exploitées par les algorithmes basés sur les arbres binaires.

Comme départ pour chaque parcours, on utilisera l'appel suivant :

```
Procédure afficheArbre(donnée a : arbrebin)
  afficheApartirdeCellule(a.racine)
fin afficher
```

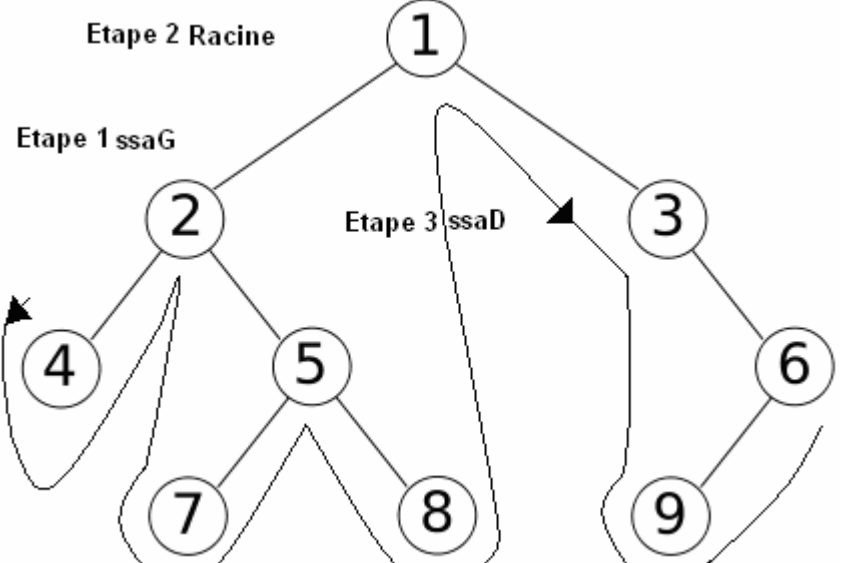
Soit une structure Arbre dont la racine est A et une référence gauche et droite à ses deux fils. Nous pouvons écrire les fonctions suivantes :

8.2.4.1. Parcours **Préfixé**

Wikipedia	
<pre>VisiterPréfixe(Arbre A) { Visiter(A) Si Non_Vide(gauche(A)) VisiterPréfixe(gauche(A)) Si Non_Vide(droite(A)) VisiterPréfixe(droite(A)) }</pre>	
Rendu du parcours	1, 2, 4, 5, 7, 8, 3, 6, 9

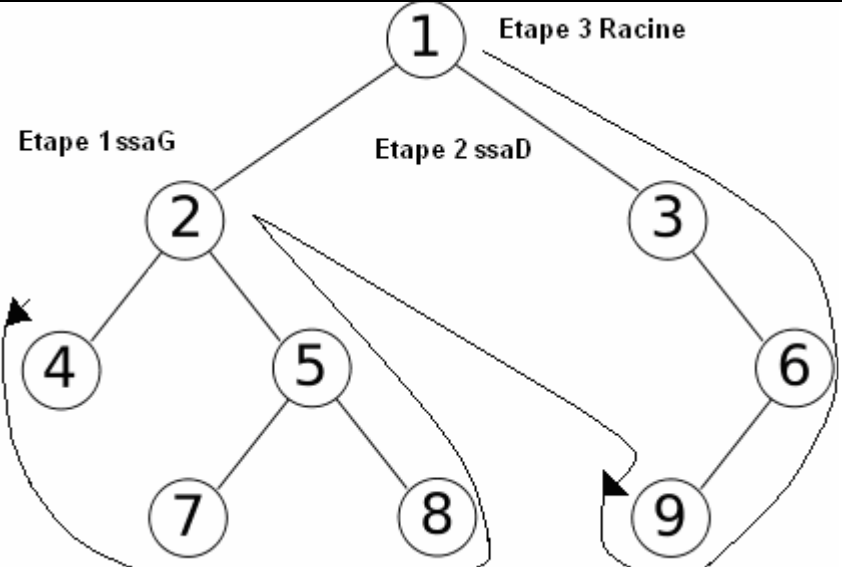
Cours (déduite)
<pre>Procédure afficheApartirdeCellule(p : pointeur sur Cellule) Début si (p <> nil) alors Affiche(p) afficheApartirdeCellule(p↑.fg) afficheApartirdeCellule(p↑.fd) fin si fin Procédure</pre>

8.2.4.2. Parcours Infixé

Wikipedia	
<pre> VisiterInfixe(Arbre A) { Si Non_Vide(gauche(A)) VisiterInfixe(gauche(A)) Visiter(A) Si Non_Vide(droite(A)) VisiterInfixe(droite(A)) } </pre>	
Rendu du parcours	4, 2, 7, 5, 8, 1, 3, 9, 6

Cours (donnée)
<pre> Procédure afficheApartirdeCellule(p : pointeur sur Cellule) Début si (p <> nil) alors afficheApartirdeCellule(p↑.fg) Affiche(p) afficheApartirdeCellule(p↑.fd) fin si fin Procédure </pre>

8.2.4.3. Parcours Postfixé

<pre> VisiterPostfixe(Arbre A) { Si Non_Vide(gauche(A)) VisiterPostfixe(gauche(A)) Si Non_Vide(droite(A)) VisiterPostfixe(droite(A)) Visiter(A) } </pre>	
Rendu du parcours	4, 7, 8, 5, 2, 9, 6, 3, 1

Cours (déduite)
Procédure afficheApartirdeCellule(p : pointeur sur Cellule) Début si (p <> nil) alors afficheApartirdeCellule(p↑.fg) afficheApartirdeCellule(p↑.fd) Affiche(p) finsi fin Procédure

8.2.5. Afficher un arbre par niveau (4^{ème} façon)

Ce parcours essaie toujours de visiter le nœud le plus proche de la racine qui n'a pas déjà été visité. En suivant ce parcours, on va d'abord visiter la racine, puis les nœuds à la profondeur 1, puis 2, etc. D'où le nom parcours en largeur.

On doit utiliser une structure de file d'attente (FIFO), on permute gauche et droite dans notre traitement.

Init_file (f : file) Enfile (f, a, racine) Tant que (non(filevide(f)) faire p ← consulter (f) //élément de tête Afficher (p) Défiler (p) Si (p↑fg ≠ null) alors Enfile (f, p↑fg) Si (p↑fd ≠ null) alors Enfile (f, p↑fd) Fin Tant que	
Rendu du parcours	1, 2, 3, 4, 5, 6, 7, 8, 9

8.3. Arbre binaire de recherche

Il faut que le sous arbre gauche et droit soit eux-mêmes des arbres binaire de recherche.

Valeur Sup(éléments du ssaG) < Valeur Noeud < Valeur Inf(éléments du ssaD)

Insertion d'un élément : si l'élément à insérer est déjà dans l'arbre, le programme s'achève.

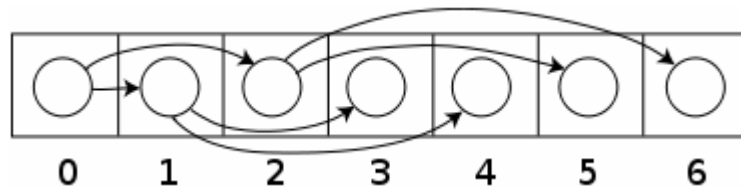
Supprimer un nœud : si les deux fils ne sont pas null, on va prendre le min(ssaG) où l'inf(ssaD) pour remplacer l'ancien noeud.

8.4. Arbres binaires et tableau (wikipedia)

Les arbres binaires peuvent aussi être rangés dans des tableaux, et si l'arbre est un arbre binaire complet, cette méthode ne gaspille pas de place, et la donnée structurée résultante est appelée **un tas**.

Dans cet arrangement compact, un nœud a un indice i , et (le tableau étant basé sur des zéros) ses fils se trouvent aux indices $2i+1$ et $2i+2$, tandis que son père se trouve (s'il existe) à l'indice $\text{floor}((i-1)/2)$. Cette méthode permet de bénéficier d'un encombrement moindre, et d'un meilleur référencement, en particulier durant un parcours préfixe. Toutefois, elle requiert une mémoire contiguë, elle est coûteuse s'il s'agit d'étendre l'arbre et l'espace perdu (dans le cas d'un arbre binaire non complet) est proportionnel à $2h - n$ pour un arbre de profondeur h avec n nœuds.

Un petit arbre binaire complet contenu dans un tableau



Pourquoi $\text{floor}((i-1)/2)$?

- ☞ Pour que l'on puisse commencer à 0.
- ☞ Quelque soit l'indice, paire ($2i+2$) ou impaire ($2i+1$), on retombera toujours sur le père.

L'indice i est un entier non signé :

Pair :

$$(2i+2-1)/2 = i + 1/2 \rightarrow \text{floor}(i) + \text{floor}(1/2) = i + 0 = i$$

Impair :

$$(2i+1-1)/2 = i \rightarrow \text{floor}(i) = i$$

FIN LIF5

