



Algorithmique et langage C

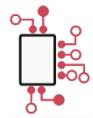
Spécialité ELEC 1A – 2017-2018

Jean-Jacques Schwartzmann

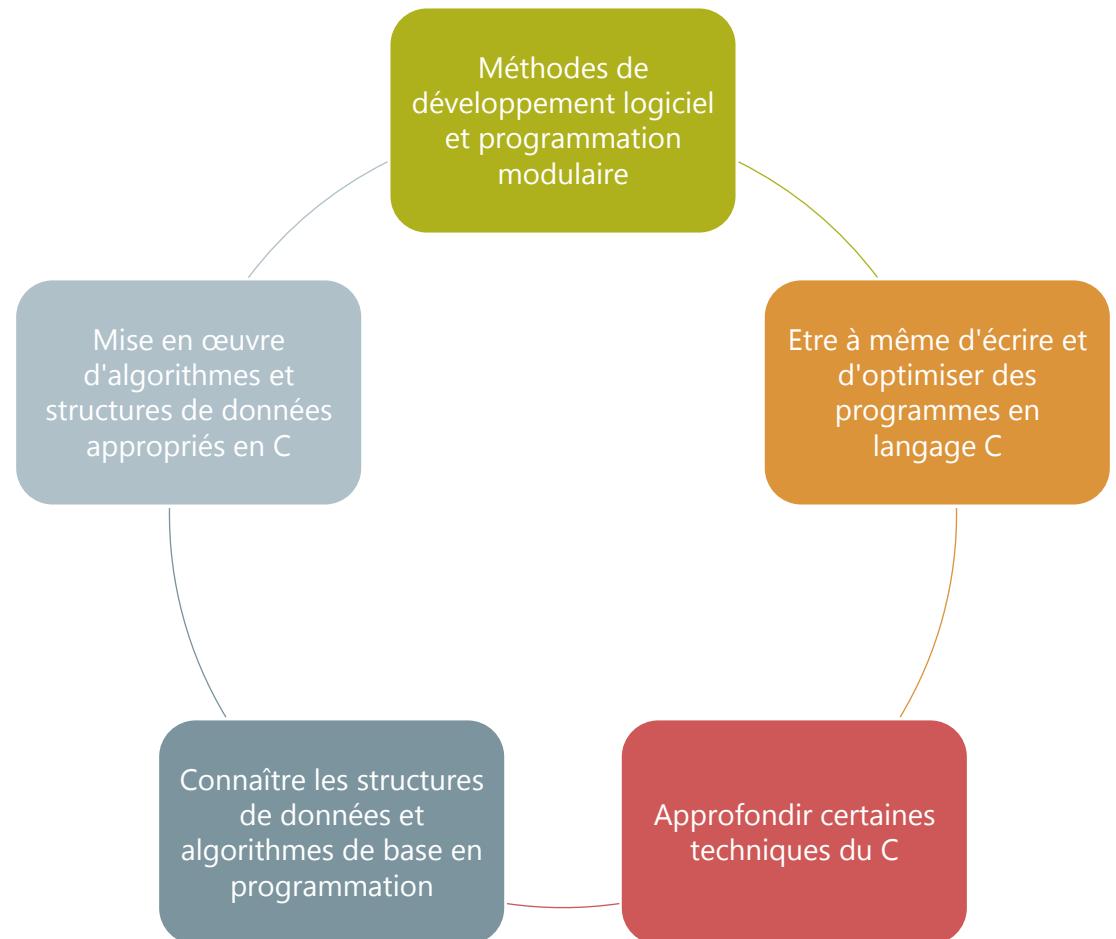
jean-jacques.schwartzmann@ensicaen.fr



L'École des INGÉNIEURS Scientifiques

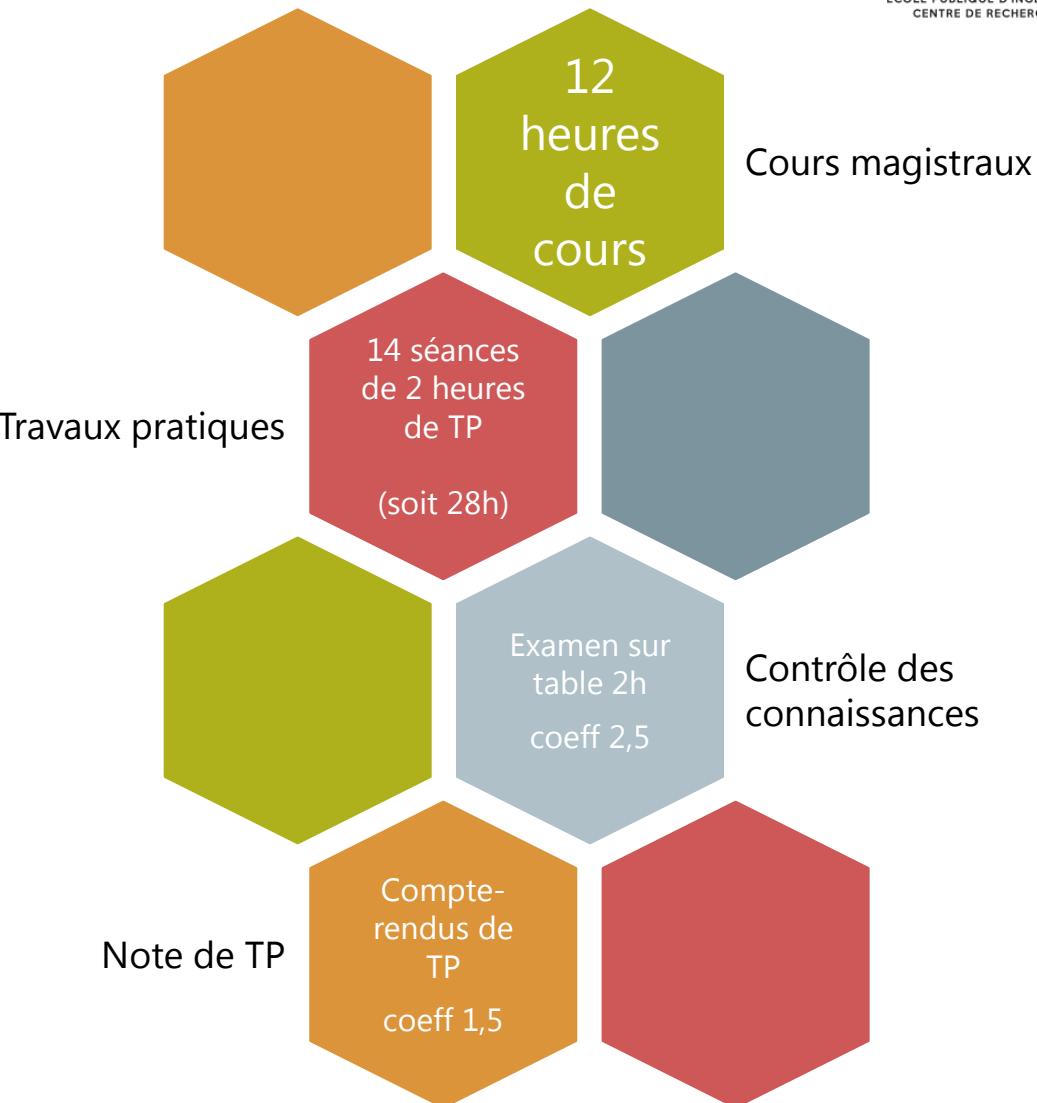


OBJECTIFS DU COURS





ORGANISATION DU COURS



RÉFÉRENCES

Livres

- Le langage C, Claude Delannoy, Eyrolles
- Apprenez à programmer en C, Mathieu Nebra, livre du zéro
- C en action, Yves Mettier, ENI
- Le langage C, Peter Aitken, Bradley L. Jones, Pearson



Liens web

- <http://c.developpez.com/cours/>
- <http://algo.developpez.com/cours/#sdd>
- <https://openclassrooms.com/courses?tags=c14955>
- <https://openclassrooms.com/courses?tags=algorithmique97302>
- <http://homepages.laas.fr/enjalber/AISEM2/coursAlgo.pdf>

PLAN DU COURS



Méthodes de développement logiciel

Rappels et approfondissements en C

Types abstraits de données

Listes chainées

Piles et files

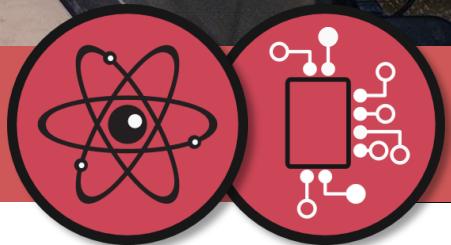
Algorithmes de tri

Tables de hachage

Arbres binaires



Méthodes de développement logiciel



MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Compilation

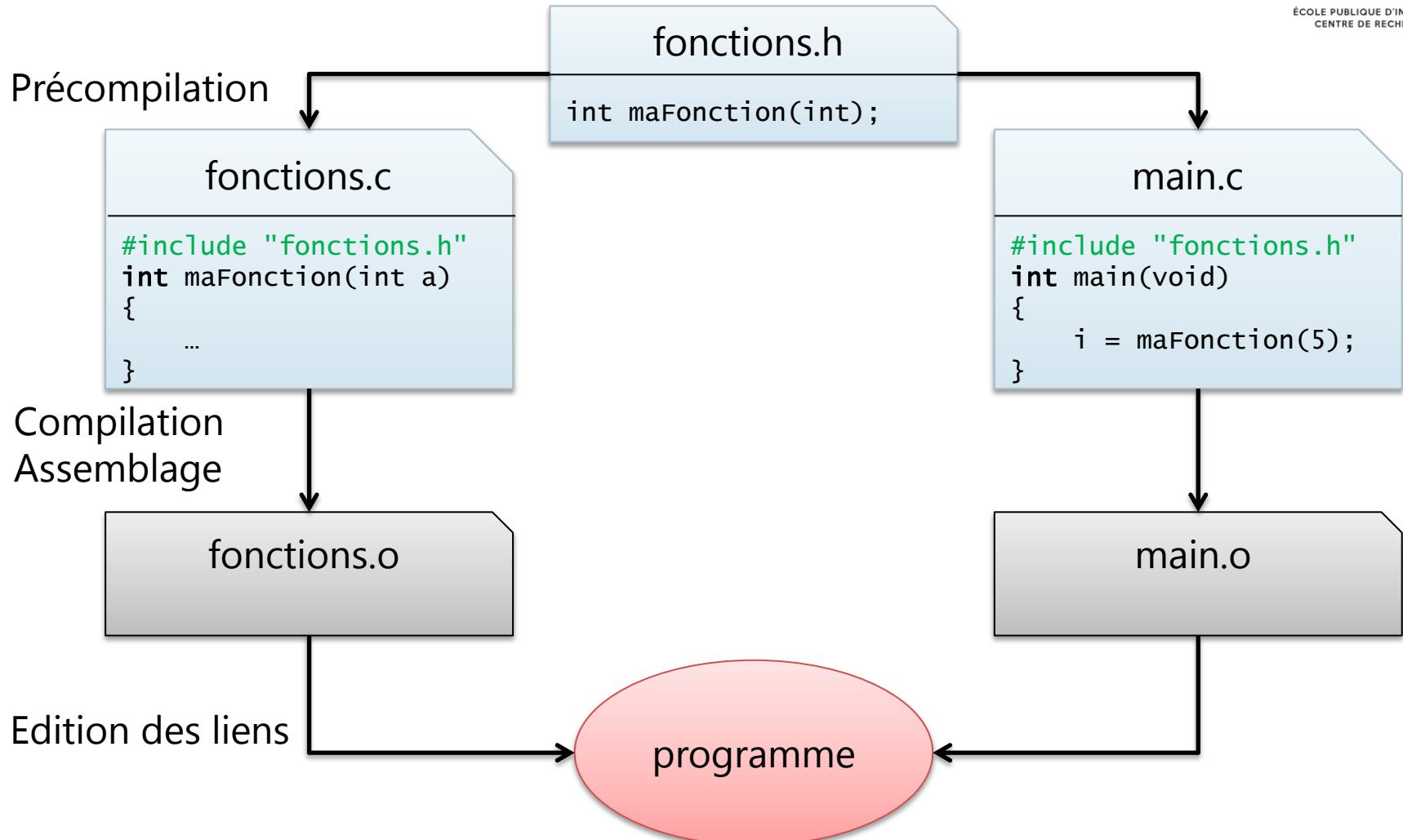
- Compilateur GCC
 - disponible sous Unix/Linux ou Windows (MinGW)
 - `gcc [options] fichier.c [-llibrairies]`
- Quelques options utiles :
 - `-o nom_fichier` : spécifie le nom du fichier de sortie produit, exécutable (`a.out` par défaut) ou fichier objet (`nom_source.o` par défaut)
 - `-c` : pas d'éditions de liens, produit un fichier objet (`fichier.o` ou `fichier.obj`)
 - `-g` : ajout d'information pour le débuggeur
 - `-O, -O1, -O2, -O3` : optimisation du code (vitesse d'exécution)
 - `-s, -Os` : optimisation du code (taille de l'exécutable)
 - `-W, -Wall, -Wextra` : affiche des warnings
 - `-ansi` : support du standard ANSI C (C89)
 - `-Wpedantic` : warning par rapport à la norme ANSI C strict

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Programmation modulaire

- Un projet en C est généralement constitué d'un ensemble de fichiers correspondant à deux catégories :
 - Code des fonctions dans des fichiers *.c
 - Prototypes de fonctions et déclarations de structures dans des fichiers *.h
- Intérêts :
 - Regroupement dans un même fichier du code correspondant à un ensemble logique de fonctionnalités,
 - Fichiers plus petits d'où un code plus lisible,
 - Réutilisation de code source (*réutilisabilité* et *généricité*),
 - Facilitation du travail en équipe.

MÉTHODES DE DÉVELOPPEMENT LOGICIEL



MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Headers

- Contenu du fichiers d'en-tête .h
 - Directives de préprocesseurs
 - Définitions de constantes symboliques
 - Déclarations des types synonymes
 - Déclarations des types composés
 - Déclarations des fonctions
 - ...
- Un fichier .h est inclus dans un fichier source avec la directive d'inclusion #include
 - `#include <stdio.h>` // en-tête des bibliothèques I/O standard
 - `#include "fonctions.h"` // en-tête d'un fichier personnel

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Inclusion des headers dans les sources

- L'inclusion de `<fonctions.h>` dans le fichier source correspondant `fonctions.c` est fortement recommandé même si elle n'est pas obligatoire : permet de s'assurer que la définition des fonctions dans `fonctions.c` est bien conforme aux *prototypes* déclarés dans `<fonctions.h>`
- L'inclusion de `<fonctions.h>` dans un autre fichier source permet d'utiliser les fonctions définies dans `fonctions.c`

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Directives de préprocesseur

- Il est possible de faire une compilation conditionnelle avec des directives de précompilation :
 - `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`
- Usages :
 - Inclusion de code conditionné à l'existence d'une constante symbolique
 - Inclusion de code en fonction de la valeur d'une constante symbolique
 - Exclusion de code (au lieu des commentaires)

```
#ifdef __unix__  
#include <unistd.h>  
#elif defined _WIN32  
#include <windows.h>  
#endif
```

```
#if VERBOSE >= 2  
printf(..., ...)  
#endif
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Constantes symboliques

- On peut ajouter des constantes symboliques au moment de la compilation avec les options de gcc
- Exemples :
 - `gcc -DVERBOSE=2 ...`
 - Affecte la valeur 2 à la constante symbolique VERBOSE
 - `gcc -DDEBUG ...`
 - Affecte la valeur 1 à la constante symbolique DEBUG
- Combiné avec la compilation conditionnelle, permet d'obtenir plusieurs exécutables en fonction des options de compilation

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Include guards

- Il se peut qu'un fichier .h fasse l'inclusion d'un autre fichier .h
- On a alors le risque d'inclure plusieurs fois le même fichier
 - Erreur lors de la compilation (déclarations multiples)
- Notion de *#include guards* pour éviter cela :

```
#ifndef _FONCTIONS_H
#define _FONCTIONS_H

int maFonction(int);

...
#endif
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Variables globales

- Les variables globales sont à déclarer dans un fichier .c et non un fichier .h, sinon, la variable serait déclarée plusieurs fois !
- Si on souhaite accéder à une variable globale définie dans un autre fichier, il faut la re-déclarer avec le mot clé **extern** :
 - Dans le fichier .h correspondant ou dans un autre fichier .c
 - Cela déclare que la variable existe et est déclaré dans un autre fichier source.

fonctions.c

```
#include "fonctions.h"

int varGlobale;
int maFonction (int i)
{...}
```

fonctions.h

```
extern int varGlobale;

int maFonction(int);
```

main.c

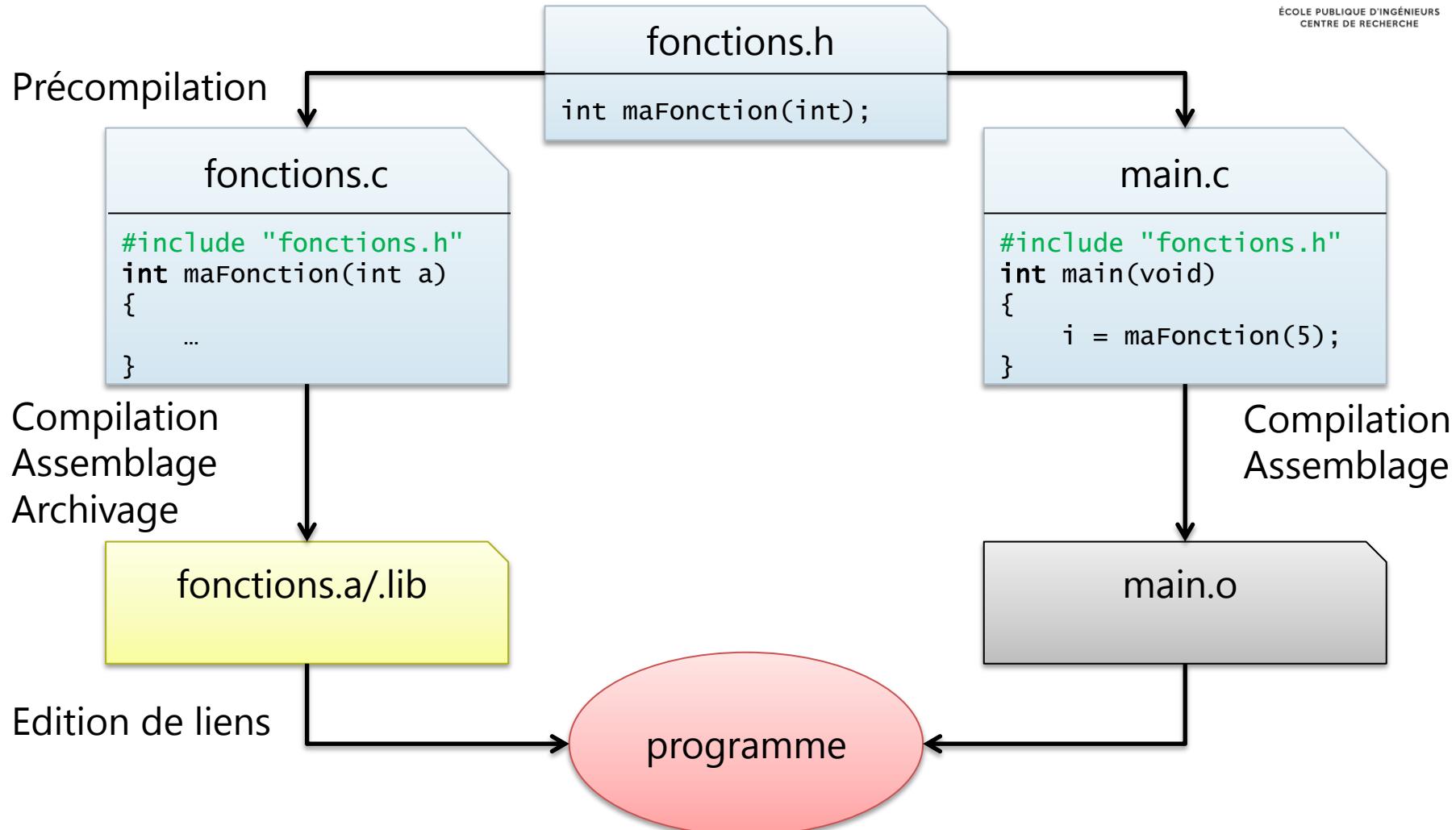
```
#include "fonctions.h"
int main(void)
{
    varGlobale = 42;
    i = maFonction(5);
}
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Bibliothèques statiques

- Il est possible de créer des bibliothèques (ou librairies) statiques :
 - Le code est compilé une fois pour toute.
 - Un fichier d'en tête .h permet de déclarer les fonctions de la bibliothèque dans un autre programme.
- Le code de la bibliothèque statique est incorporé dans le programme lors de l'édition des liens.
- Une bibliothèque est généralement composée de plusieurs fichiers sources : il s'agit en fait d'une archive contenant plusieurs fichiers .o

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

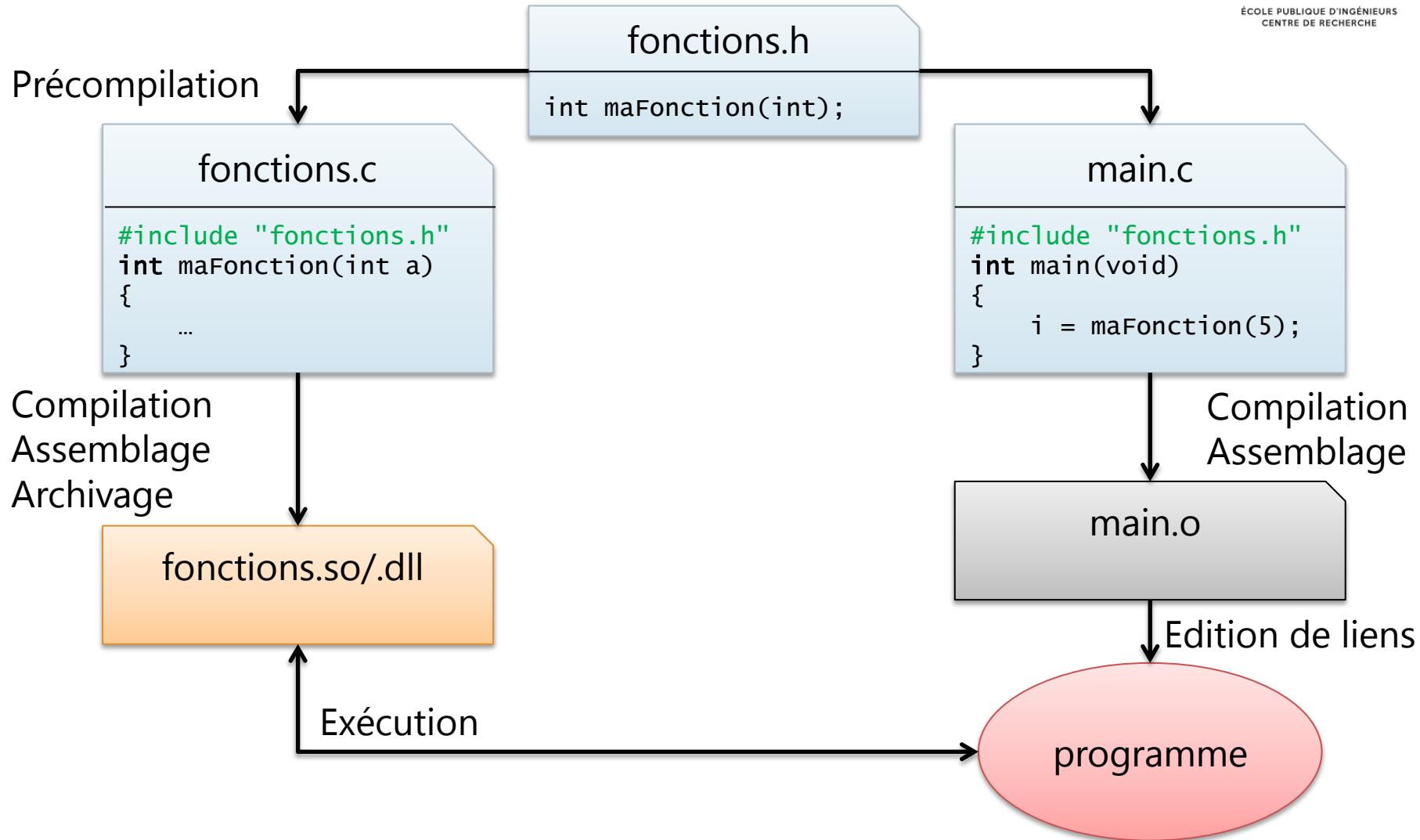


MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Bibliothèques dynamiques

- Il est possible de créer des bibliothèques partagées / dynamiques
 - Le code est compilé une fois pour toutes.
 - Un fichier d'en tête .h permet de déclarer les fonctions de la bibliothèque dans un autre programme.
- Le code de la bibliothèque est incorporé dans le programme lors de l'exécution du programme :
 - Au chargement du programme en mémoire,
 - ou bien à l'appel d'une fonction.
- Les bibliothèques dynamiques doivent être livrées avec le programme.

MÉTHODES DE DÉVELOPPEMENT LOGICIEL



MÉTHODES DE DÉVELOPPEMENT LOGICIEL

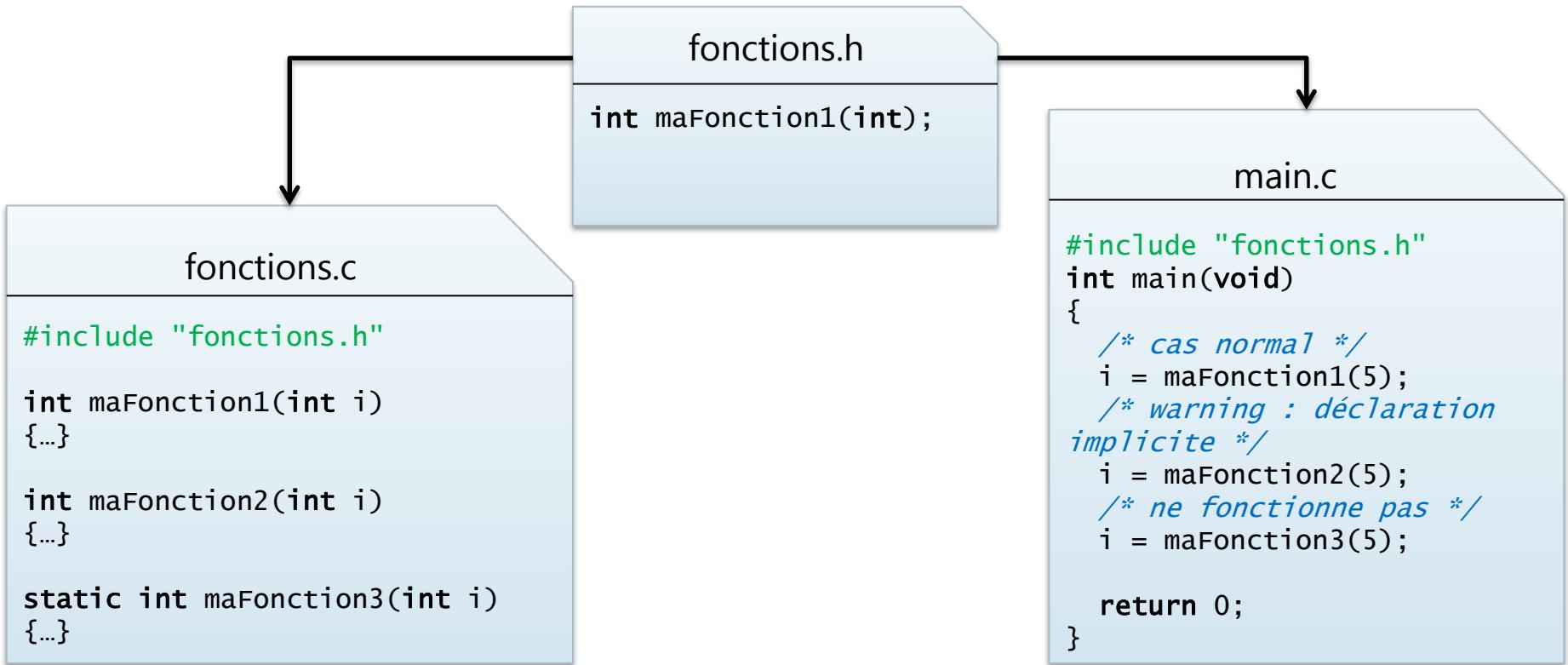
Bibliothèques dynamiques vs statiques

	Avantages	Inconvénients
Bibliothèques statiques	<ul style="list-style-type: none"> Application autonome Aucune action particulière de la part du système lors de l'exécution → embarqué 	<ul style="list-style-type: none"> Taille de l'application importante Correction d'une bibliothèque statique → recompilation de l'application nécessaire
Bibliothèques dynamiques	<ul style="list-style-type: none"> Taille de l'application réduite Correction d'une bibliothèque dynamique → correction de l'application 	<ul style="list-style-type: none"> Application dépendant de la bibliothèque Gestion de plusieurs bibliothèques dynamiques en même temps

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Fonctions *static*

- Le mot clé *static* appliqué à une fonction permet de cacher celle-ci.



MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Compilation d'un projet

1. Production des fichiers objets :

```
gcc -Wall -c fonctions.c  
gcc -Wall -c main.c
```

2. Production du fichier binaire :

```
gcc -o programme fonctions.o main.o → programme
```

- La commande *make* permet d'automatiser la phase de compilation en produisant un fichier de règles nommé *makefile* :

```
programme : fonctions.o main.o  
          gcc -o programme fonctions.o main.o  
main.o : main.c fonctions.h  
         gcc -Wall -c main.c  
fonctions.o : fonctions.c fonctions.h  
         gcc -Wall -c fonctions.c  
clean :  
       rm -f *.o
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Fichier makefile

- Dans de grands projets avec un grand nombre de fichiers C, le makefile permet de ne recompiler que les fichiers nécessaires, ce qui permet un gain de temps important à la compilation.
- Un fichier makefile est constitué d'un ensemble de règles de la forme :
cible : dépendance(s)
commande(s)
- Chaque commande doit être obligatoirement précédé d'une tabulation (pas d'espace !).
- Une règle n'a pas forcément de dépendance (par exemple la règle *clean*).

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Exécution d'un makefile

- L'exécution d'un makefile est réalisée par la commande : `make [cible]`. Si aucune cible n'est précisée, c'est la première cible du fichier makefile qui est prise en compte.
- Les dépendances d'une cible sont examinées. Pour chaque dépendance correspondant à la cible d'une autre règle, cette règle est évaluée. Ceci est fait de façon récursive, jusqu'à la résolution complète de toutes les dépendances.
- Si la cible est plus ancienne qu'une de ses dépendances, alors la règle est exécutée.
- Les règles *clean* et *mrproper* sont utiles pour forcer un projet à revenir à son état initiale. C'est indispensable lorsque c'est le fichier makefile lui-même qui est modifié ou que le projet est exporté vers une autre plateforme (par ex. de Linux à Windows).

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Makefile : utilisation de variables (1/2)

- On considère le makefile suivant :

```
all : final tests
final : final.o fonctions.o
        gcc final.o fonctions.o -o final
tests : tests.o fonctions.o
        gcc tests.o fonctions.o -o tests
tests.o : tests.c fonctions.h
        gcc -Wall -c tests.c
final.o : final.c fonctions.h
        gcc -Wall -c final.c
fonctions.o : fonctions.c fonctions.h
        gcc -Wall -c fonctions.c
clean :
        rm -f *~ *.o
mrproper : clean
        rm -f final tests
```

- Sur de grands projets, l'écriture du makefile devient vite fastidieuse...

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Makefile : utilisation de variables (2/2)

- Certaines variables sont prédéfinies :
 - Cible : \$@
 - Listes des dépendances : \$^
 - Première dépendance : \$<
- On peut aussi définir des variables utilisateurs avec la syntaxe :
VARIABLE=valeur

```
CC=gcc
CFLAGS=-wall -c
OBJECTS=fonctions.o
HEADERS=fonctions.h
```

- On peut ainsi réécrire les règles :

```
final : final.o fonctions.o
        gcc final.o fonctions.o -o final
final.o : final.c fonctions.h
        gcc -wall -c final.c
```



```
final : final.o $(OBJECTS)
        $(CC) $^ -o $@
final.o : final.c $(HEADERS)
        $(CC) $(CFLAGS) $< -o $@
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Simplification de l'écriture du makefile à l'aide des variables

```

all : final tests
final : final.o fonctions.o
    gcc final.o fonctions.o -o final
tests : tests.o fonctions.o
    gcc tests.o fonctions.o -o tests
tests.o : tests.c fonctions.h
    gcc -Wall -c tests.c
final.o : final.c fonctions.h
    gcc -Wall -c final.c
fonctions.o : fonctions.c fonctions.h
    gcc -Wall -c fonctions.c
clean :
    rm -f *~ *.o
mrproper : clean
    rm -f final tests
  
```



```

CC=gcc
CFLAGS=-Wall -c
EXEC=final tests
OBJECTS=fonctions.o
HEADERS=fonctions.h
all : $(EXEC)
final : final.o $(OBJECTS)
    $(CC) $^ -o @@
tests : tests.o $(OBJECTS)
    $(CC) $^ -o @@
tests.o : tests.c $(HEADERS)
    $(CC) $(CFLAGS) $< -o @@
final.o : final.c $(HEADERS)
    $(CC) $(CFLAGS) $< -o @@
fonctions.o : fonctions.c $(HEADERS)
    $(CC) $(CFLAGS) $< -o @@
clean :
    rm -f *~ *.o
mrproper : clean
    rm -f $(EXEC)
  
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Règles implicites et règles génériques

- Certaines règles sont déjà prédéfinies, par exemple une cible *fichier.o* est obtenue automatiquement à partir de la source *fichier.c* sans qu'il ne soit nécessaire de la mentionner dans le makefile.
- Il peut être nécessaire de préciser ou de redéfinir certaines règles implicites, par exemple la variable prédéfinie `CFLAGS` permet de redéfinir les options de compilation d'un fichier source C :

`CFLAGS=-Wall`

- Les dépendances des fichiers sources C peuvent être obtenues par la commande :
`gcc -MM *.c`
- Il est possible de les obtenir directement au moment de l'exécution du makefile avec la règle suivante :

```
SOURCES=final.c tests.c fonctions.c
depend :
    makedepend $(SOURCES)
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Simplification du makefile à l'aide des règles et variables prédéfinies

```

CC=gcc
CFLAGS=-Wall
EXECS=final tests
OBJECTS=fonctions.o
SOURCES=final.c tests.c fonctions.c

all : $(EXECS)

final : final.o $(OBJECTS)

tests : tests.o $(OBJECTS)

depend :
    makedepend $(SOURCES)

clean :
    rm -f *~ *.o

mrproper : clean
    rm -f $(EXECS)
  
```

```

$ make
gcc -Wall -c -o final.o final.c
gcc -Wall -c -o fonctions.o fonctions.c
gcc final.o fonctions.o -o final
gcc -Wall -c -o tests.o tests.c
gcc tests.o fonctions.o -o tests
$ 
  
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Doxxygen

- Doxxygen est un logiciel et une norme pour la documentation technique du code C et C++ (il supporte également d'autres langages de programmation).
- Il permet de documenter le code en ajoutant des commentaires particuliers afin de pouvoir produire une documentation technique au format HTML (ou PDF, RTF, LaTeX) automatiquement.
- Intérêts :
 - Documentation cohérente avec le code,
 - Incite les développeurs à commenter systématique le code, ce qui permet de ne pas avoir à le documenter ultérieurement.

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Doxxygen : exemple

```
#ifndef _TABELEM_H
#define _TABELEM_H

#include <stdlib.h>
#include "element.h"

/** @file tabelem.h
 * @author <a href="mailto:jean-jacques.schwartzmann@ensicaen.fr">Jean-Jacques S.</a>
 * @date 27/03/2017
 * @version 1.0
 * @brief Déclarations de fonctions de manipulation de tableaux d'éléments
 */

/** @brief Affichage d'un tableau de type TElement
 */
void printTab(PTElement, size_t);

/** @brief Génère un tableau d'éléments aléatoires entre 2 valeurs fixées
 * @return Un pointeur vers la tableau d'aléas générés
 */
PTElement initRandomTab(size_t, TElement, TElement);

#endif
```

MÉTHODES DE DÉVELOPPEMENT LOGICIEL

Doxygen : exemple de documentation produite

tabelem.h File Reference Functions

Déclarations de fonctions de manipulation de tableaux d'éléments. [More...](#)

`#include "element.h"`

[Go to the source code of this file.](#)

Functions

void printTab (PTElement, int)
Affichage d'un tableau de type TElement.

PTElement initRandomTab (int, TElement, TElement)
Génère un tableau d'éléments aléatoires entre 2 valeurs fixées. [More...](#)

```
string sInput,  
int iLength, iN;  
double dblTemp;  
bool again = true;  
  
while (again) {  
    iN = -1;  
    again = false;  
    getline(cin, sInput);  
    system("cls");  
    stringstream(sInput)  
    iLength = sInput.length();  
    if (iLength < 4) {  
        again = true;  
        continue;  
    } else if (sInput[iLength - 1] == '0')  
        again = true;  
        continue;  
    } while (++iN < iLength)  
        if (isdigit(sInput[iN]))  
            continue;  
        else if (iN == 1)  
            continue;
```

PLAN DU COURS

Méthodes de développement logiciel

Rappels et approfondissements en C

Types abstraits de données

Listes chainées

Piles et files

Algorithmes de tri

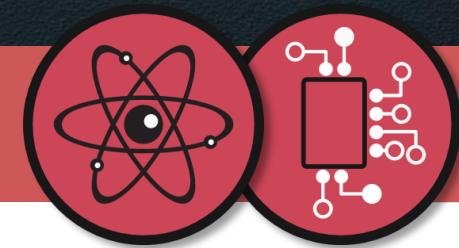
Tables de hachage

Arbres binaires

```
string sInput;
int iLength, iN;
double dblTemp;
bool again = true;

while (again) {
    iN = -1;
    again = false;
    getline(cin, sInput);
    system("cls");
    stringstream(sInput) >> dblTemp;
    iLength = sInput.length();
    if (iLength < 4) {
        again = true;
        continue;
    } else if (sInput[iLength - 3] != '.') {
        again = true;
        continue;
    } else if (++iN < iLength) {
        if (sInput[iN]) {
            again = true;
        }
    }
}
```

Rappels et approfondissements en C



RAPPELS ET APPROFONDISSEMENTS EN C

Les tableaux

- Un tableau est un ensemble fini d'éléments de *même type*, stockés en mémoire à des adresses contiguës.
- Syntaxe d'un tableau à une dimension
 - `<type> <nom_tableau>[<nombre_elements>];`
- *type* peut être n'importe quel type connu (char, short, int, float...)
- *nombre_elements* est une **constante entière** positive.
- Exemple
 - `int tab[10]; /* tableau de 10 éléments de type entier */`
 - Cette déclaration alloue en mémoire $10 * \text{sizeof(int)}$ octets consécutifs.

RAPPELS ET APPROFONDISSEMENTS EN C

Manipulations élémentaires sur les tableaux

- L'accès aux éléments se fait avec l'opérateur [i]
 - i est le numéro de l'élément que l'on souhaite récupérer : *indice* ou encore *index*.
 - i va de 0 à *nombre_elements* – 1
- Il est possible d'initialiser un tableau lors de sa déclaration par une liste de constantes
 - <type> <nom_tableau>[<N>] = { <const1>, <const2>, ..., <constN> }
- Exemple
 - `int tab[4] = { 1, 2 }; /* initialise les 2 premiers éléments de tab, le reste des éléments est mis à 0 */`
 - `tab[0] = 5; /* stocke la valeur 5 dans la première case du tableau */`
 - `j = tab[1]; /* renvoie 2 et l'affecte à la variable j */`
- Il n'est pas possible d'écrire *tab2 = tab1* !
 - Il faut effectuer l'affectation pour élément par élément (dans une boucle ou en utilisant les familles de fonctions *strcpy* ou *memcpy*)

RAPPELS ET APPROFONDISSEMENTS EN C

Organisation d'un tableau unidimensionnel en mémoire

```
#define N 5  
  
int tab1[N] = {5, 3, 2, 4, 1};
```

Adresses mémoire	Valeurs
0x00402000	5
0x00402004	3
0x00402008	2
0x0040200C	4
0x00402010	1

RAPPELS ET APPROFONDISSEMENTS EN C

Avantages/inconvénients des tableaux

- Le tableau est la structure de données la plus commune pour stocker et manipuler un volume conséquent de données.
- Avantages :
 - Accès dit aléatoire : accès direct aux données indépendamment de leur localisation.
- Inconvénients :
 - Taille fixe : on doit connaître la taille des données *à priori* (mais on peut toutefois utiliser des pointeurs et *alloquer dynamiquement* la mémoire).
 - Insertion et suppression de données coûteuses : déplacement des données avec recopies.
- Un élément dans un tableau a une adresse mémoire qui est référencée par un *pointeur*.

RAPPELS ET APPROFONDISSEMENTS EN C

Tableaux multidimensionnels

- Un tableau peut avoir plusieurs dimensions. Il y alors autant d'indices de positionnement que de dimensions.
- Cas d'un tableau à deux dimensions :
 - `<type> <nom_tableau>[<dimension1>][<dimension2>];`
- Initialisation d'un tableau à plusieurs dimensions lors de sa déclaration :
 - `<type> <nom_tableau>[<N1>] [<N2>]= { {<const1>, <const2>, ...,<constN2>} , { ... }, ... , { ... } }`
- Un tableau à deux dimensions est en fait un tableau à une dimension, dont chaque élément est lui-même un tableau à une dimension. On peut déclarer des tableaux à X dimensions, donc chaque élément est un tableau à X – 1 dimensions.

RAPPELS ET APPROFONDISSEMENTS EN C

Organisation d'un tableau bidimensionnel en mémoire

```
#define M 2
#define N 3

int tab2[M][N] = { {1, 2, 3}, {4, 5, 6} };
```

Adresses mémoire	Valeurs
0x00402000	1
0x00402004	2
0x00402008	3
0x0040200C	4
0x00402010	5
0x00402014	6

RAPPELS ET APPROFONDISSEMENTS EN C

Chaines de caractères

- Une chaîne est un tableau unidimensionnel de caractères se terminant par la valeur 0 (caractère '\0').
- Ce caractère est indispensable afin de pouvoir déterminer la longueur d'une chaîne sans devoir stocker, à part, sa longueur comme avec les tableaux classiques.
- Il existe un ensemble de fonctions déclarées dans le fichier `<string.h>` permettant la manipulation des chaînes de caractères comme `strlen` et `strcpy`.

RAPPELS ET APPROFONDISSEMENTS EN C

Initialisation d'une chaîne de caractères

- On peut initialiser une chaîne avec une chaîne de caractères littérale, ou bien avec une liste de caractères :
 - `char s[8] = "exemple";`
 - `char s[8] = {'e', 'x', 'e', 'm', 'p', 'l', 'e', '\0' };`
 - La chaîne de caractères est complétée par le compilateur avec le caractère nul '`\0`'. Il faut donc prendre en compte ce caractère dans la taille du tableau.
- Il est possible de ne pas spécifier la taille du tableau à l'initialisation. Par défaut, elle correspondra à la taille passée à l'initialisation.
 - `char s[] = "exemple"; /* ici taille du tableau = 8 */`
- **Il n'est pas possible d'écrire `s2 = s1` !**
 - On utilisera les fonctions `strcpy` ou `strncpy` pour copier le contenu d'une chaîne dans une autre.

RAPPELS ET APPROFONDISSEMENTS EN C

Organisation d'une chaîne en mémoire

```
char s1[] = "exemple";
```

Adresses mémoire	Valeurs
0x00402000	e
0x00402001	x
0x00402002	e
0x00402003	m
0x00402004	p
0x00402005	l
0x00402006	e
0x00402007	'\0'

RAPPELS ET APPROFONDISSEMENTS EN C

Les pointeurs

- Les pointeurs ont un rôle **essentiel** en langage C car ils permettent de manipuler directement la mémoire. Ils sont cependant la **principale source** de difficultés et d'erreurs pour le programmeur en langage C .
- Un pointeur désigne l'adresse mémoire d'une donnée.
- Pour déclarer un pointeur, le symbole * est utilisé.
- Le symbole & permet d'obtenir l'adresse mémoire d'une variable
- *Déréférencer* permet d'obtenir l'élément référencé par un pointeur.
- Pour *déréférencer* un pointeur, le symbole * est utilisé.
- **Attention** : le symbole * a deux significations différentes avec les pointeurs suivant la nature de l'utilisation.

RAPPELS ET APPROFONDISSEMENTS EN C

Exemple de manipulations sur les pointeurs

```
int i = 7;
int* i_ptr;          /* Déclaration d'un pointeur sur un int */
i_ptr = &i;           /* La variable i_ptr contient l'adresse de la
                      variable i */
int j = *i_ptr*2;    /* *i_ptr correspond dans ce cas à la valeur
                      de i */
(*i_ptr)++;          /* Attention : différent de *i_ptr++ !!! */
printf("%d %d %d %p\n", i, *i_ptr, j, i_ptr);
```

```
$ ./a.out
8 8 14 0xbfec84
```

RAPPELS ET APPROFONDISSEMENTS EN C

Pointeurs et tableaux

- Une variable de type tableau est en fait un pointeur sur le premier élément du tableau.

```
char tab[] = {'E', 'N', 'S', 'I', 'C', 'A', 'E', 'N', 0};
char* c_ptr = tab;
printf("%p\n%p\n%p\n%c\n", tab, c_ptr, &tab[0], *c_ptr);
```

```
$ ./a.out
0xbfd03244
0xbfd03244
0xbfd03244
0xbfd03244
E
```

- Ajouter une valeur x directement à un pointeur correspond à faire référence à l'élément x d'un tableau.

```
c_ptr += 4;
printf("%p %c\n", c_ptr, *c_ptr);
```

```
$ ./a.out
0xbfd03248 C
```

- L'arithmétique sur les pointeurs tient compte de la taille en octets des éléments référencés, c'est à dire la taille en octets d'un élément du tableau.

```
int* i_ptr = (int *)tab;
i_ptr++;
printf("%p %c %s\n", i_ptr, *i_ptr, (char *)i_ptr);
```

```
$ ./a.out
0xbfd03248 C CAEN
```

RAPPELS ET APPROFONDISSEMENTS EN C

Exercice sur l'arithmétique des pointeurs

```

int tab1[] = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89};
int* ptab1 = tab1;
double tab2[10];
double* ptab2 = tab2;
int i=0;
for (i=0; i<10; i++){
    printf("%i %p %p\n", i, ptab1, ptab2);
    *ptab2 = (float)(*ptab1 * *ptab1);
    ptab2++;
    ptab1++;
}
for (i=0; i<10; i++)
    printf("%.0lf ",tab2[i]);
printf("\n");
  
```

```

$ ./a.out
0 0xb81de650 0xb81de680
1 0xb81de654 0xb81de688
2 0xb81de658 0xb81de690
3 0xb81de65c 0xb81de698
4 0xb81de660 0xb81de6a0
5 0xb81de664 0xb81de6a8
6 0xb81de668 0xb81de6b0
7 0xb81de66c 0xb81de6b8
8 0xb81de670 0xb81de6c0
9 0xb81de674 0xb81de6c8
1 4 9 25 64 169 441 1156 3025 7921
$
```

RAPPELS ET APPROFONDISSEMENTS EN C

Tableaux multidimensionnels et pointeurs

- Un tableau multidimensionnel de dimension N est un tableau dont chaque élément est lui-même un tableau de dimension N – 1.
- En mémoire, il est organisé de manière linéaire.

```
#define M 2
#define N 3

int tab2[M][N] = { {1, 2, 3}, {4, 5, 6} };
int *i_ptr = (int *)tab2;
int i;
for (i = 0; i < M*N; i++)
    printf("%p %d\n", i_ptr + i, i_ptr[i]);
```

```
$ ./a.out
0xbfff6cf4 1
0xbfff6cf8 2
0xbfff6cfc 3
0xbfff6d00 4
0xbfff6d04 5
0xbfff6d08 6
```

RAPPELS ET APPROFONDISSEMENTS EN C

Tableaux et fonctions

- Il est possible de passer un tableau comme argument à une fonction.
- La fonction ne connaît pas à priori la taille du tableau, il faut donc passer également cette taille en argument.

```
void affiche_tab(char tab[], int taille) {
    int i = 0;
    printf("[");
    for (i = 0; i < taille; i++) {
        printf("0x%02x", tab[i]);
        if (i < taille-1) printf(", ");
    }
    printf("]\n");
}
```

```
$ ./a.out
[0x45, 0x4E, 0x53, 0x49, 0x43, 0x41, 0x45, 0x4E]
```

```
void main(void) {
    char *str = "ENSICAEN";
    affiche_tab(str, strlen(str));
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Tableaux et fonctions (suite)

- Pour ne pas avoir à passer la taille d'un tableau, une solution est d'utiliser une valeur dans le tableau marquant la fin du tableau.
- Inconvénient : réduit le nombre de valeurs "utiles" du tableau.
- Exemple : tableau d'entiers positifs terminé par la valeur – 1.
- C'est la convention utilisée pour les chaînes de caractères, exploitée par les fonctions *strlen*, *strcpy*, *strcmp*, *strchr*, *puts*, *printf*... (caractère '\0' terminal) :
 - `char *strcpy(char *dest, const char *source);`
 - `int strcmp(const char *s1, const char *s2);`

RAPPELS ET APPROFONDISSEMENTS EN C

La fonction main

- Il est possible de passer dans la ligne de commande des arguments après le nom de l'exécutable.
- Le main reçoit dans ce cas le nombre d'arguments et un tableau de chaînes de caractères dont les éléments correspondent aux arguments passés.

```
int main(int argc, char* argv[]) {
    printf("Nombre d'arguments %d\n", argc);
    printf("Nom du programme appelant : %s\n", argv[0]);
    int i;
    for (i = 1; i < argc; i++)
        printf("Argument %d: %s\n", i, argv[i]);
    return EXIT_SUCCESS;
}
```

```
$ ./a.out "Le C" a "des arguments !"
Nombre d'arguments 4
Nom du programme appelant : ./a.out
Argument 1: Le C
Argument 2: a
Argument 3: des arguments !
```

RAPPELS ET APPROFONDISSEMENTS EN C

Pointeurs et fonctions

- Il est possible de passer des pointeurs en arguments à une fonction : c'est le cas par exemple quand on passe un tableau en argument :
 - `void affiche_tab(char tab[], int taille);`
- Il peut être intéressant également de passer certaines variables *par adresse* plutôt que *par valeur* pour des raisons de gain en rapidité et en mémoire : en effet à l'appel d'une fonction, celle-ci **recopie** dans la pile les arguments qui lui sont passés. Si une variable est volumineuse en mémoire (cas d'une structure complexe par exemple), il sera plus avantageux de passer son adresse en argument pour éviter cette phase de recopie, coûteuse en temps et en mémoire.
- En pratique, il est recommandé de ne passer par valeur que les types de base du langage (entiers et réels), et de plutôt passer par adresse les variables d'autres types.

RAPPELS ET APPROFONDISSEMENTS EN C

Pointeurs et fonctions

- Enfin, **l'unique façon** en C de **modifier la valeur** d'une variable passée en argument est de passer son adresse, donc un pointeur vers cette variable.

```

void permuter_valeur(int a, int b) {
    int c = a;
    a = b;
    b = c; /* Aucun effet */
}
void permuter_adresse(int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
}
void main(void) {
    int a = 2, b = 5;
    printf("a = %d, b = %d\n", a, b);
    permuter_valeur(a, b);
    printf("a = %d, b = %d\n", a, b);
    permuter_adresse(&a, &b);
    printf("a = %d, b = %d\n", a, b);
}

```

```

$ ./a.out
a = 2, b = 5
a = 2, b = 5
a = 5, b = 2

```

RAPPELS ET APPROFONDISSEMENTS EN C

Pointeurs génériques

- Il est possible de déclarer des pointeurs non typés :
 - Exemple : `void *ptr;`
- On ne peut pas utiliser l'opérateur de déférencement `*` avec ces pointeurs.
- Pour affecter un pointeur typé à un pointeur générique, il faut utiliser un *cast* :
 - `void *malloc(size_t size); /* prototype de malloc */`
 - `int *i_ptr = (int *)malloc(10 * sizeof(int));`
- Il n'est pas nécessaire de caster les arguments de fonctions :
 - `void *free(void *ptr); /* prototype de free */`
 - `free(i_ptr);`

RAPPELS ET APPROFONDISSEMENTS EN C

Pointeurs et fonctions génériques

- Les pointeurs génériques permettent d'écrire des fonctions génériques comme les fonctions *memcpy*, *memcmp*, *memchr*, ...

```

void permuter(void *a, void *b, size_t taille) {
    void *c = malloc(taille);
    memcpy(c, a, taille); /* *c = *a interdit ! */
    memcpy(a, b, taille); /* *a = *b */
    memcpy(b, c, taille); /* *b = *c */
    free(c);
}

void main(void) {
    int a = 2, b = 5;
    double x = 3.14, y = 2.72;
    printf("a = %d, b = %d\n", a, b);
    printf("x = %.21f, y = %.21f\n", x, y);
    permuter(&a, &b, sizeof(a));
    permuter(&x, &y, sizeof(x));
    printf("a = %d, b = %d\n", a, b);
    printf("x = %.21f, y = %.21f\n", x, y);
}
  
```

```

$ ./a.out
a = 2, b = 5
x = 3.14, y = 2.72
a = 5, b = 2
x = 2.72, y = 3.14
  
```

RAPPELS ET APPROFONDISSEMENTS EN C

Allocation dynamique de pointeurs

- Limitation de l'utilisation des tableaux
 - Besoin de connaître *à priori* le nombre d'éléments à déclarer.
 - Si le nombre d'éléments n'est pas fixe, nécessité de créer des tableaux surdimensionnés → perte d'espace mémoire !
- Intérêts de l'allocation dynamique
 - Créer des variables / tableaux lors de l'exécution du programme
 - Optimiser l'utilisation de la mémoire
- Contraintes de l'allocation dynamique
 - En C, pas de *garbage collector* → gestion de la création / libération de la mémoire sous la responsabilité du programmeur.
 - Source importante d'erreurs :
 - Erreurs de segmentation
 - Fuite de mémoire



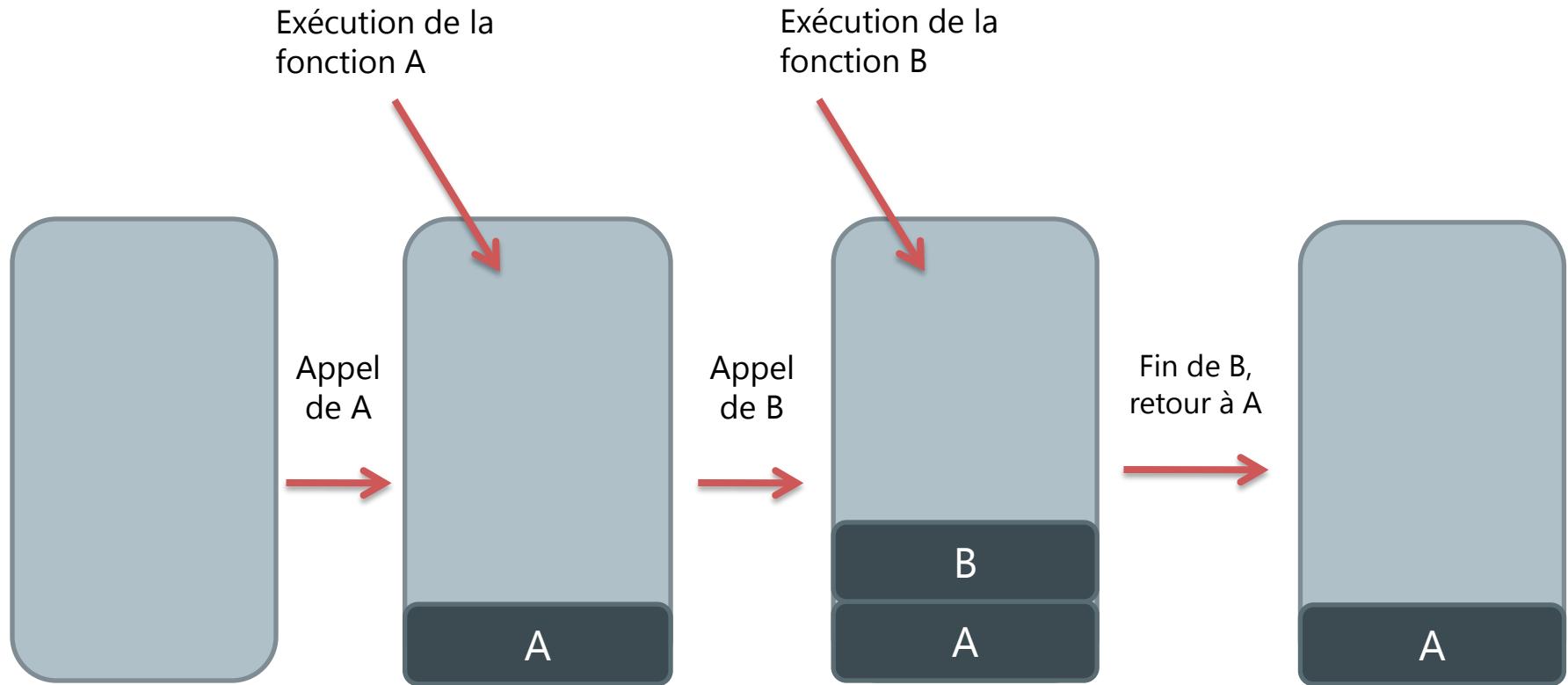
RAPPELS ET APPROFONDISSEMENTS EN C

Fonctions d'allocation dynamique

- L'allocation dynamique se gère à partir de 4 fonctions :
 - *malloc*
 - *calloc*
 - *realloc*
 - *free*
- Les variables créées avec l'allocation dynamique sont créées dans un segment mémoire appelé le tas (heap)
- Ces fonctions sont accessibles depuis la bibliothèque stdlib :
 - `#include <stdlib.h>`

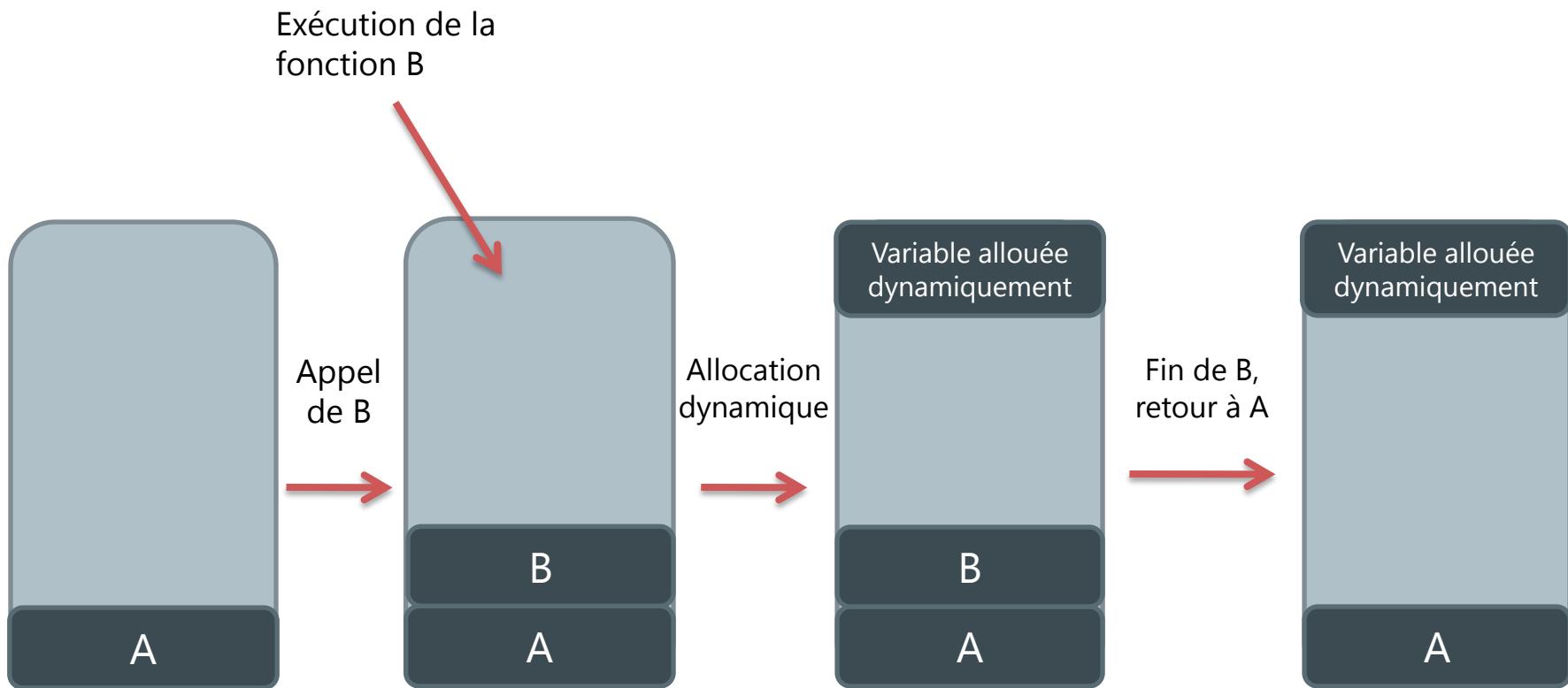
RAPPELS ET APPROFONDISSEMENTS EN C

Pile d'exécution (stack)



RAPPELS ET APPROFONDISSEMENTS EN C

Pile et tas



RAPPELS ET APPROFONDISSEMENTS EN C

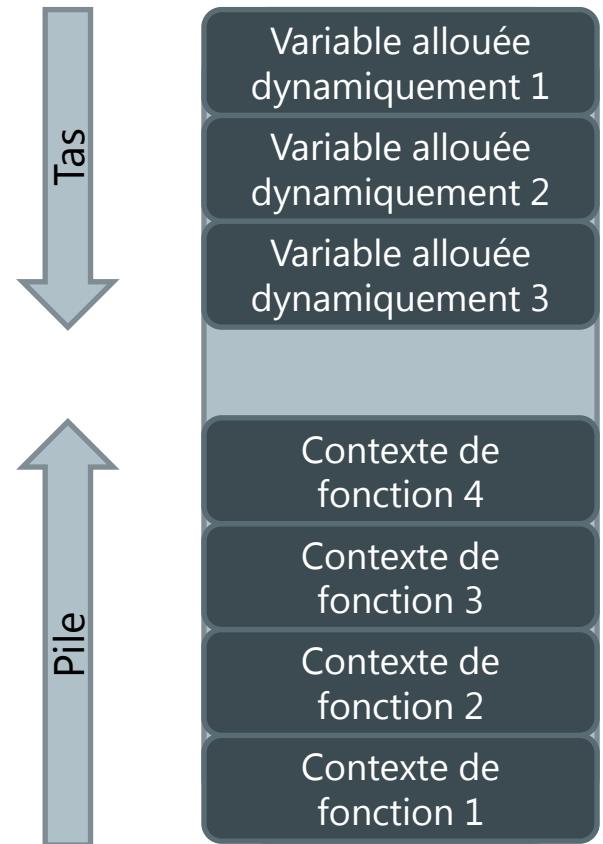
Fonction malloc

- La création d'une variable / d'un tableau dynamiquement se fait par l'utilisation de la fonction *malloc*.
- Prototype : `void *malloc(size_t <nombre d'octets>);`
- malloc réserve <nombre d'octets> en mémoire et renvoie un pointeur générique (`void*`) vers cette zone mémoire.
 - Utilisation d'un cast pour convertir ce pointeur en `<type *>`
 - Utilisation de `sizeof(<type>)` pour calculer le nombre d'octets nécessaires
 - Allocation d'un tableau de n variables de type `<type>` :
 - `<type> *ptr = (<type> *)malloc(n*sizeof(<type>));`

RAPPELS ET APPROFONDISSEMENTS EN C

Erreur à l'allocation

- Il se peut que l'allocation ne soit pas possible :
 - Plus d'adresse en mémoire vive disponible si trop d'allocation mémoire dans la pile et dans le tas,
 - Corruption du tas.
- *malloc* renvoie la valeur **NULL** dans les 2 cas.
- **Il faut tester** la valeur de retour de malloc pour vérifier s'il n'y a pas eu d'erreur !



RAPPELS ET APPROFONDISSEMENTS EN C

Gestion des erreurs d'allocation mémoire : exemple de code

```
#include <stdlib.h>

int main(void)
{
    int * pi = (int *)malloc(1000 * sizeof(int));

    if (pi == NULL) {
        perror("Erreur dans main(), malloc()");
        return EXIT_FAILURE;
    }

    ... /* exécution normale */

    return EXIT_SUCCESS;
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Fonction calloc

- *malloc* n'initialise pas la valeur de la zone mémoire réservée dans le tas → il faut l'initialiser manuellement (important pour la sécurité du code).
- *calloc* permet d'allouer dynamiquement un tableau et d'initialiser ses éléments à 0 en même temps.
- Prototype : **void *calloc(size_t <n_elems>, size_t <taille_type>);**
- *calloc* réserve un tableau *<n_elems>* éléments de taille *<taille_type>* en mémoire et renvoie un pointeur générique (**void***) vers la zone mémoire allouée.
- Exemple : allocation d'un tableau de 100 entiers :
 - **int *i_ptr = (int *)calloc(100, sizeof(int));**

RAPPELS ET APPROFONDISSEMENTS EN C

Fonction free

- La zone mémoire réservée par malloc/calloc n'est pas supprimée automatiquement quand on ne s'en sert plus.
 - Ceci est dû à l'absence de garbage collector (ramasse miette) en C, contrairement à d'autres langages comme Java/C#
- Il faut donc impérativement supprimer manuellement la mémoire lorsque l'on ne s'en sert plus ou à la fin du programme → utilisation de la fonction free
- Prototype : `void free(void *pointeur);`
- Attention à ne pas libérer plusieurs fois la même zone mémoire !
 - Source fréquente d'erreur de segmentation (segmentation fault)



RAPPELS ET APPROFONDISSEMENTS EN C

Erreurs courantes d'allocation

- L'erreur est l'affectation $pi = pj$
- La zone mémoire pointé par pi est perdu, il n'est donc plus possible de libérer cette zone mémoire → fuite de mémoire !
- pi et pj pointent vers la même zone mémoire, les deux free libèrent donc la même zone mémoire !
→ segmentation fault !

```
#include <stdlib.h>

void main(void)
{
    int * pi, * pj;

    pi = (int *)malloc(100*sizeof(int));
    pj = (int *)malloc(100*sizeof(int));

    pi = pj;      /* A éviter !!! */

    free(pi);
    free(pj);
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Fonction realloc

- *realloc* permet de redimensionner une zone mémoire alloué dynamiquement.
- Prototype : **void *realloc(void *ptr, size_t <nombre d'octets>);**
- <nombre d'octets> est le nombre d'octets à réservé, comme pour *malloc*.
- *realloc(ptr, 0)* équivaut à *free*.
- *realloc* conserve le contenu initial de la zone mémoire, mais n'initialise pas la nouvelle mémoire allouée.

RAPPELS ET APPROFONDISSEMENTS EN C

Retourner un espace alloué dynamiquement dans une fonction

- Si l'on souhaite effectuer une allocation dynamique dans une fonction, il faut la retourner à la fonction appelante pour pouvoir libérer l'espace mémoire alloué quand on n'en a plus besoin.
- Il faut donc renvoyer un pointeur vers l'espace alloué :
 - Soit comme valeur de retour de la fonction (comme dans malloc, calloc ou realloc),
 - Soit en passant ce pointeur comme argument dans la fonction.
- Exemple :

```
int *creerTableauEntiers(const int nbElems)
{
    return (int *)calloc(nbElems, sizeof(int));
}
```



RAPPELS ET APPROFONDISSEMENTS EN C

Erreur courante dans l'allocation de pointeurs passés en arguments

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

bool creerTableauEntiers(const int nbElems, int *Tab)
{
    Tab = (int *)calloc(nbElems, sizeof(int));
    if (Tab == NULL)
        return false;
    return true;
}

int main(void)
{
    int *Tab;
    printf("%p\n", Tab);
    if (!creerTableauEntiers(5, Tab)) return EXIT_FAILURE;
    printf("%p\n", Tab);
    free(Tab);
    return EXIT_SUCCESS;
}
```

```
$ ./a.out
0x8048543
0x8048543
*** Error in `./a.out': free(): invalid pointer: 0x08048543 ***
Abandon (core dumped)
```

RAPPELS ET APPROFONDISSEMENTS EN C

Allocation de pointeurs passés comme arguments dans une fonction



- Si on passe directement par valeur le pointeur à allouer comme argument dans la fonction, il sera inchangé à la sortie de la fonction ! Il faut donc lui passer **l'adresse du pointeur** à allouer.

```
bool creerTableauEntiers(const int nbElems, int **Tab) {  
    *Tab = (int *)calloc(nbElems, sizeof(int));  
    if (*Tab == NULL)  
        return false;  
    return true;  
}  
  
int main(void) {  
    int *Tab;  
    if (!creerTableauEntiers(100, &Tab)) return EXIT_FAILURE;  
    ...  
    free(Tab);  
    return EXIT_SUCCESS;  
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Organisation des variables en mémoire

- Variables allouées automatiquement
 - Création/destruction à l'entrée/sortie d'un bloc délimité par des { }

- Variables allouées dynamiquement
 - Création/destruction gérées par le programmeur avec malloc/free

- Variables allouées statiquement
 - Création automatique au chargement du programme en mémoire, destruction automatique en fin de programme

Adresses mémoire hautes

Segment pile



Segment tas



Segment réservé aux bibliothèques

Segment BSS

Segment de données

Segment de texte

Zone réservée (pointeur NULL)

Adresses mémoire basses

RAPPELS ET APPROFONDISSEMENTS EN C

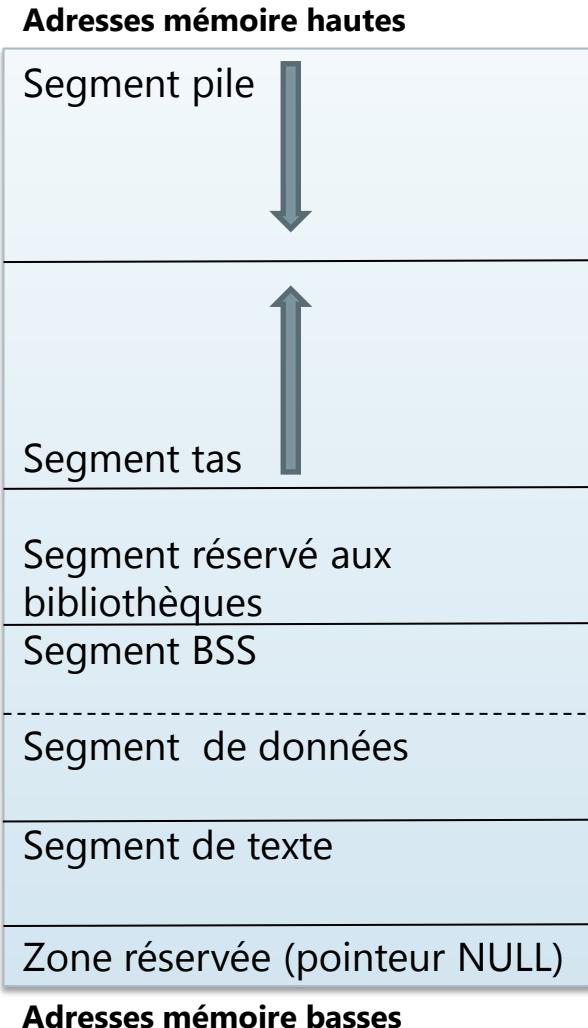
Organisation des variables et fonctions en mémoire : résumé

Mot clé	Appliqué à	Accès	Stockage
-	Variable locale	Uniquement la fonction de déclaration	Segment pile → non initialisée
-	Variable globale	Toutes fonctions, y compris celles déclarés dans d'autres fichiers → utilisation du mot clé extern	Segment données/BSS → Initialisée à 0
-	Variable allouée dynamiquement	Toutes fonctions, y compris celles déclarés dans d'autres fichiers → utilisation de pointeurs	Segment tas → non initialisée
-	Fonction	Toutes fonctions, y compris celles déclarés dans d'autres fichiers	Segment texte
static	Variable locale	Uniquement la fonction de déclaration	Segment données/BSS → Initialisée à 0
static	Variable globale	Toutes fonctions définies dans le même fichier	Segment données/BSS → Initialisée à 0
static	Fonction	Toutes fonctions définies dans le même fichier	Segment texte

RAPPELS ET APPROFONDISSEMENTS EN C

Organisation des variables en mémoire

- Variables allouées automatiquement →
- Variables allouées dynamiquement →
- Variables allouées statiquement →
- **Pointeurs sur fonctions** →



RAPPELS ET APPROFONDISSEMENTS EN C

Pointeurs sur fonctions

- Un pointeur sur une fonction correspond à l'adresse mémoire de la première instruction de la fonction dans le segment texte.
- Déclaration :
`<type de retour>(* nom pointeur)(<type des paramètres>)`
- Exemple :
`int (*p_fonction)(int x, int y);`
- L'intérêt des pointeurs sur fonctions est de pouvoir passer une fonction en paramètre d'une autre fonction → notion de *callback*.

RAPPELS ET APPROFONDISSEMENTS EN C

Passage en paramètre d'un pointeur sur fonction

- Il n'est pas nécessaire d'utiliser l'opérateur & pour récupérer l'adresse de la fonction passée en paramètre.

```
double carre(double);

void afficheOperateur(double(*fonction)(double), double);

void main(void)
{
    afficheOperateur(carre, 10);
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Utilisation d'un pointeur sur fonction passé en paramètre

- On peut utiliser la notation * pour utiliser la fonction passée en argument :

```
void afficheOperateur(double(*fonction)(double), double var) {  
    double resultat;  
    resultat = (*fonction)(var);  
    printf("%f", resultat);  
}
```

- Ou plus simplement, directement la notation usuelle :

```
void afficheOperateur(double(*fonction)(double), double var) {  
    double resultat;  
    resultat = fonction(var);  
    printf("%f", resultat);  
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Exemple complet

```
#include <stdio.h>
double carre(double nb){ return nb*nb; }
double cube(double nb){ return nb*nb*nb; }

void afficheOperateur(double(*fonction)(double), double var)
{
    double resultat;
    resultat = (*fonction)(var);
    printf("%lf\n", resultat);
}

void main(void)
{
    double x = 10;
    afficheOperateur(carre, x);
    afficheOperateur(cube, x);
}
```

```
$ ./a.out
100.000000
1000.000000
$
```

RAPPELS ET APPROFONDISSEMENTS EN C

Callback : exemple de la fonction *qsort*

- La fonction *qsort* est une fonction générique de la librairie C standard qui implémente l'algorithme de tri rapide "quick sort" pour trier des tableaux de n'importe quelle type.
- Prototype :

```
void qsort (void *base, size_t nelems, size_t size,
            int (*compar)(const void *, const void *));
```
- La fonction *qsort* trie les éléments du tableau pointé par *base* contenant *nelems* éléments de taille *size* en les triant avec la fonction pointé par *compar*.
- La fonction *compar* est un exemple de *fonction de callback*.
- Exemple de fonction *compar* : *strcmp* pour le tri des chaînes de caractères.

RAPPELS ET APPROFONDISSEMENTS EN C

Exemple de code : tri d'un tableau d'entiers avec *qsort*

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_ELEMS 10

int intcmp(const void *pa, const void *pb) {
    int a = *(const int *)pa;
    int b = *(const int *)pb;
    if (a < b) return (-1);
    if (a > b) return 1;
    return 0;
}
void afficheTabEntiers(const int *Tab, const int N) {
    int i;
    printf("Tab = [");
    for (i = 0; i < N-1; i++) printf(" %d,", Tab[i]);
    printf(" %d ]\n", Tab[i]);
}
int main(void) {
    int tabEntiers[NUM_ELEMS] = { 56, 85, -12, 0, 7, 96, 2, 9, -1, 18 };
    afficheTabEntiers(tabEntiers, NUM_ELEMS);
    qsort(tabEntiers, NUM_ELEMS, sizeof(int), intcmp);
    afficheTabEntiers(tabEntiers, NUM_ELEMS);
    return EXIT_SUCCESS;
}
```

```
$ ./a.out
Tab = [ 56, 85, -12, 0, 7, 96, 2, 9, -1, 18 ]
Tab = [ -12, -1, 0, 2, 7, 9, 18, 56, 85, 96 ]
```

RAPPELS ET APPROFONDISSEMENTS EN C

Récursivité

- Définition : « Une fonction récursive est une fonction récursive » ;-)
- « Vraie » définition : une fonction *récursive* est une fonction qui s'appelle elle-même (notion *d'auto-appel*). Les appels successifs de la fonction vont donc se trouver empilés l'un à la suite de l'autre dans la pile du programme.
- Une fonction récursive doit impérativement contenir une condition d'arrêt pour éviter de boucler indéfiniment (en fait jusqu'à ce qu'il se produise un dépassement de capacité ou un débordement de la pile).
- L'intérêt de la récursivité est de permettre la résolution de certains problèmes algorithmiques, de manière simple et élégante.

RAPPELS ET APPROFONDISSEMENTS EN C

Fonction factorielle : algorithme itératif

- La factorielle d'un entier positif n est définie par :
$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1,$$
 avec convention : $0! = 1$
- d'où le code itératif suivant :

```
unsigned long factorielle (unsigned long n)
{
    if (n == 0)
        return 1;
    unsigned long f = 1;
    for (int i = 1; i <= n; i++)
        f *= i;
    return f;
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Fonction factorielle : algorithme récursif

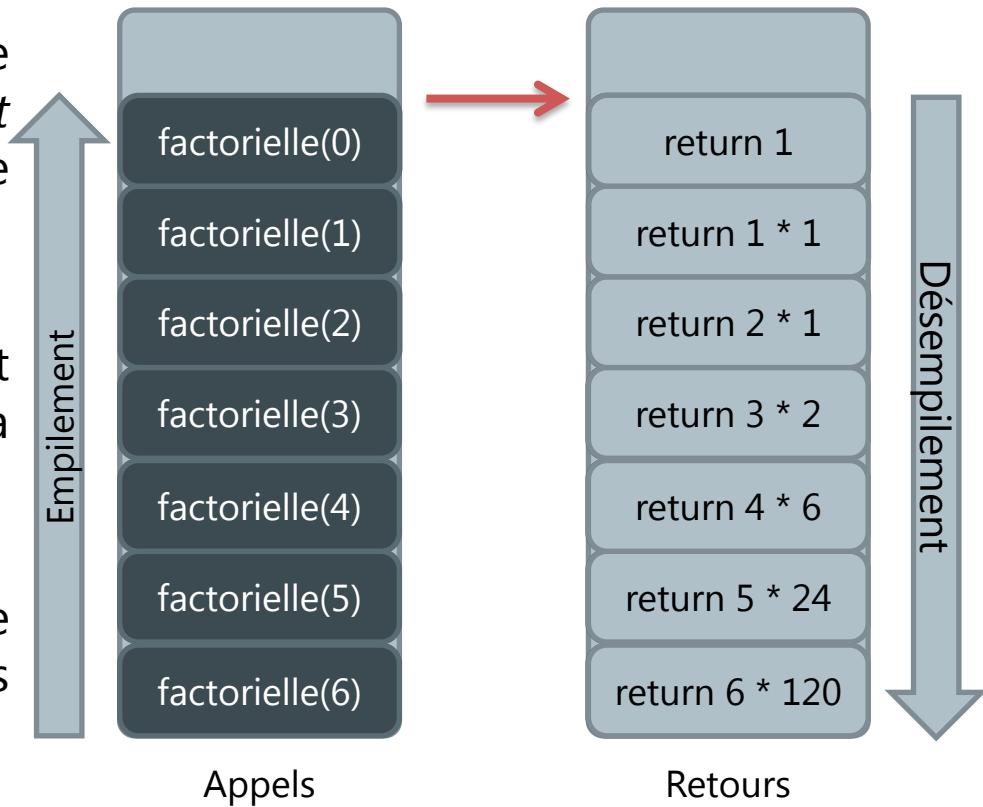
- On peut aussi remarquer que $(n - 1)! = (n - 1) \times (n - 2) \times \dots \times 2 \times 1$
- donc on peut ainsi définir la factorielle de n par :
$$n! = n \times (n - 1)!$$
- d'où le code récursif suivant :

```
unsigned long factorielle(unsigned long n)
{
    if (n == 0)
        return 1;
    return n * factorielle(n - 1);
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Pile de récursion

- L'exécution d'une fonction récursive passe par une phase *d'empilement* des appels, suivie d'une phase de *déempilement* lors des retours :
- Les appels à la fonction sont empilés jusqu'à l'atteinte de la condition d'arrêt.
- Les appels enregistrés sont ensuite déempilés au fur et à mesure des retours successifs.



RAPPELS ET APPROFONDISSEMENTS EN C

Fonctions récursives vs fonctions itératives

- Malgré une apparente simplicité dans le code, une fonction récursive peut demander beaucoup de ressources mémoire lorsque la pile de récursion devient profonde, du fait des auto-appels successifs et de l'empilement qui en résulte.
- C'est pourquoi, lorsqu'on a le choix, il est généralement préférable d'utiliser la version itérative d'une fonction, plutôt que sa version récursive.
- Cependant, certains problèmes se prêtent très bien à la récursivité et ne peuvent être implémentés que difficilement de manière itérative. C'est par exemple le cas des tris « divide and conquer » comme le tri fusion et le tri rapide, ou bien encore celui des parcours d'arbre.

RAPPELS ET APPROFONDISSEMENTS EN C

Suite de Fibonacci

- La suite de Fibonacci est une suite dont chaque terme est la somme des deux termes précédents. Elle est définie par :

$$F_{n+2} = F_{n+1} + F_n \text{ avec } F_0 = 0 \text{ et } F_1 = 1$$

- La définition étant elle-même récursive, on peut facilement implémenter cette suite sous forme de fonction récursive :

```
unsigned long fibonacci(unsigned long n)
{
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Récurivité terminale

- L'implémentation précédente n'est pas optimale car on se retrouve à calculer deux fois les mêmes valeurs. Le temps de calcul croît exponentiellement avec n .
- On peut l'optimiser en passant en paramètres les termes initiaux de la suite, de sorte que l'appel récursif soit la *dernière instruction à évaluer*, d'où la notion de *récursivité terminale*.

```
unsigned long fibonacci(unsigned long n,
                      unsigned long a,
                      unsigned long b)
{
    if (n == 0)
        return a;
    return fibonacci(n - 1, b, a + b);
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Intérêt de la récursivité terminale

- La récursivité terminale permet d'éliminer la phase de déempilement, puisque le dernier appel permet d'obtenir directement la valeur à calculer, contrairement à la récursivité simple où cette valeur est calculée lors des retours successifs.
- Les compilateurs comme GCC optimisent le code binaire produit, de sorte que les appels successifs à la fonction n'ont pas besoin d'être empilés car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.
- Lors de la compilation, la récursion terminale peut être transformée en itération, ce qui permet d'obtenir un code binaire efficace à partir d'un code source récursif plus simple, plus élégant et plus lisible que sa version itérative.

RAPPELS ET APPROFONDISSEMENTS EN C

Exemple d'utilisation de la fonction de Fibonacci à récursivité terminale

```
#include <stdlib.h>
#include <stdio.h>
unsigned long fibonacci(unsigned long n, unsigned long a, unsigned long b)
{
    if (n == 0) return a;
    return fibonacci(n - 1, b, a + b);
}
int main(int argc, char **argv)
{
    if (argc != 2) {
        printf("Usage : %s NUMELEMMS\n", argv[0]);
        return EXIT_FAILURE;
    }
    unsigned long N = atoi(argv[1]);
    if (N < 1) {
        printf("Usage : %s NUMELEMMS\n", argv[0]);
        return EXIT_FAILURE;
    }
    printf("Et voici les %lu premiers termes de la suite de Fibonacci : ", N);
    for (unsigned long n = 0; n < N; n++) printf("%lu, ", fibonacci(n, 0, 1));
    printf("\n");
    return EXIT_SUCCESS;
}
```

\$./fibonacci 15

Et voici les 15 premiers termes de la suite de Fibonacci :
 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,

RAPPELS ET APPROFONDISSEMENTS EN C

Récursivité croisée

- On parle de *récursion mutuelle*, ou *récurativité croisée*, lorsque deux fonctions sont mutuellement définies l'une en termes de l'autre.
- Soit par exemple deux suites u_n et v_n définies par :

$$\begin{cases} u_0 = 1 \text{ et } v_0 = 2 \\ u_{n+1} = 3u_n + 2v_n \\ v_{n+1} = 2u_n + 3v_n \end{cases}$$
- u_n et v_n seront implémentés comme indiqué ci-contre :

```
unsigned v(unsigned);
unsigned u(unsigned n)
{
    if (n == 0) return 1;
    return 3*u(n-1) + 2*v(n-1);
}

unsigned v(unsigned n)
{
    if (n == 0) return 2;
    return 2*u(n-1) + 3*v(n-1);
}
```

RAPPELS ET APPROFONDISSEMENTS EN C

Autre exemple (sans aucun intérêt pratique) de récursivité croisé

- *is-even* (respectivement *is_odd*) teste si un nombre est pair (respectivement impair)

```
bool is_odd(unsigned long);  
  
bool is_even(unsigned long n) {  
    if (n == 0)  
        return true;  
    else  
        return is_odd(n - 1);  
}  
  
bool is_odd(unsigned long n) {  
    if (n == 0)  
        return false;  
    else  
        return is_even(n - 1);  
}
```

PLAN DU COURS

Méthodes de développement logiciel

Rappels et approfondissements en C

Types abstraits de données

Listes chainées

Piles et files

Algorithmes de tri

Tables de hachage

Arbres binaires



Types abstraits de données

TYPES ABSTRAITS DE DONNÉES

Définition d'un TAD

- Définition mathématique : un TAD est un ensemble d'éléments distincts auquel est associé un ensemble d'opérations :
 - Opérations de création et de suppression permettant de créer un objet du type abstrait de données, et de le supprimer,
 - Opérations d'accès aux éléments permettant d'identifier un élément de l'ensemble,
 - Opérations de manipulation des éléments permettant de calculer d'autres éléments, internes ou externes au TAD.
- A rapprocher de la notion de *classe* et *d'objet* (POO) :
 - *constructeurs* et *destructeurs*,
 - *accesseurs* et *mutateurs*,
 - autres *méthodes*.

TYPES ABSTRAITS DE DONNÉES

Les champs du TAD

- **Nom** : nom du TAD,
- **Utilise** : énumération des types utilisés par le TAD,
- **Opérations** : prototypage de toutes les opérations, de la forme :
 $nom : type1 \times type2 \times \dots \times typeN \rightarrow type1' \times type2' \times \dots \times typeM'$
- **Axiomes** : description du comportement de chaque opération, sous la forme d'une proposition logique,
- **Préconditions** : conditions à respecter sur les arguments des opérations.

TYPES ABSTRAITS DE DONNÉES

Exemple de TAD : Point 2D à l'écran

- **Nom :** TPoint,
- **Utilise :** int, bool,
- **Opérations :**
 - New : int × int → TPoint
 - GetX : TPoint → int
 - GetY : TPoint → int
 - MoveTo: TPoint × int × int →
 - Show : TPoint →
 - Hide : TPoint →
 - IsVisible : TPoint → bool
 - Delete : TPoint →

TYPES ABSTRAITS DE DONNÉES

Type de données abstraits usuels

- Il existe différents TAD pour manipuler des ensembles de données :
 - Les tableaux
 - Les listes
 - Les piles
 - Les files
 - Les tables de hachage
 - Les arbres

TYPES ABSTRAITS DE DONNÉES

Programmation orientée objet (POO)

- Certains langages comme *C++*, *Objective-C*, *C#*, *Java*, *Delphi* (Pascal objet) et même les dernières versions de *Fortran* et de *COBOL* supportent la *programmation orientée objet*.
- Un *objet* est un conteneur symbolique et autonome qui contient des informations et des mécanismes concernant un sujet, manipulés dans un programme (Wikipedia)
- Une *classe* est une description des caractéristiques d'un ou de plusieurs objets. Chaque objet créé à partir de cette classe est une *instance* de la classe en question.
- Concrètement, on peut dire qu'une classe est une structure à laquelle on ajoute des fonctions (appelées *méthodes*) permettant de gérer cette structure.
- Un TAD peut être représenté par une classe

TYPES ABSTRAITS DE DONNÉES

Les classes en C++

- Outre l'ajout des fonctions membres, il existe trois grandes nouveautés par rapport aux structures du C :
 - **L'encapsulation** : les membres de classe, qu'ils soient des variables ou des fonctions, peuvent être privés c'est-à-dire inaccessibles en dehors de la classe (par opposition aux membres publics d'une structure C où tous ses membres sont accessibles). Les *attributs* (variables) privés ne peuvent être manipulés de l'extérieur que par les *méthodes* publiques.
 - **L'héritage** : une classe peut être dérivée. La classe dérivée hérite alors de toutes les propriétés et fonctions de la classe mère. Une classe peut d'ailleurs hériter de plusieurs classes simultanément (*héritage multiple*).
 - **Le polymorphisme** : un objet, instance d'une classe appartenant à une hiérarchie de classes, exécute les méthodes homonymes partagées à plusieurs niveaux d'une manière propre à cette classe. Autrement dit, une classe dérivée peut redéfinir les méthodes de la classe parente.
- Une classe se déclare via le mot-clé **class** suivi du nom de la classe, d'un bloc (accolades ouvrante et fermante) et d'un point virgule (ne pas l'oublier !). Les membres de la classe (variables ou fonctions) doivent être déclarés à l'intérieur de ce bloc, à la manière des structures en C.

TYPES ABSTRAITS DE DONNÉES

Exemple : implémentation classique en C d'un objet « point »

```
/* fichier point.h */
#ifndef _POINT_H
#define _POINT_H

#include <stdbool.h>

typedef struct {
    int X, Y;
    bool visible;
} TPoint;

typedef TPoint *PTPoint;

void TPoint_Init(PTPoint, int, int);

void TPoint_MoveTo(PTPoint, int, int);

void TPoint_Show(PTPoint);

void TPoint_Hide(PTPoint);

#endif
```

```
/* fichier point.c */
#include "point.h"
void TPoint_Init(PTPoint pPoint, int initX, int initY)
{
    pPoint->X = initX;
    pPoint->Y = initY;
    pPoint->visible = false;
}
void TPoint_MoveTo(PTPoint pPoint, int newX, int newY)
{
    TPoint_Hide(pPoint);
    pPoint->X = newX;
    pPoint->Y = newY;
    TPoint_Show(pPoint);
}
void TPoint_Show(PTPoint pPoint)
{
    pPoint->visible = true;
}
void TPoint_Hide(PTPoint pPoint)
{
    pPoint->visible = false;
}
```

TYPES ABSTRAITS DE DONNÉES

Implémentation orientée objet en C++ d'un objet point

```
// fichier point.h
#ifndef _POINT_H
#define _POINT_H

class TPoint {
private:
    int X, Y;
    bool visible;
public:
    TPoint(int = 0, int = 0); // Constructeur
    int GetX() { return X; }
    int GetY() { return Y; }
    void MoveTo(int , int);
    void Show() { visible = true; }
    void Hide() { visible = false; }
    bool IsVisible() { return visible; }
};

#endif
```

```
// fichier point.cpp
#include "point.h"

TPoint::TPoint(int initX, int initY)
{
    X = initX;
    Y = initY;
    visible = false;
}

void TPoint::MoveTo(int newX, int newY)
{
    Hide();
    X = newX;
    Y = newY;
    Show();
}
```

TYPES ABSTRAITS DE DONNÉES

C vs C++

```
/* fichier main.c */
#include <stdio.h>
#include <stdlib.h>
#include "point.h"

int main(void)
{
    TPoint O, A, B;
    TPoint_Init(&O, 0, 0);
    TPoint_Init(&A, 6, 0);
    TPoint_Init(&B, 5, 7);
    printf("O = (%d, %d)\n", O.X, O.Y);
    printf("A = (%d, %d)\n", A.X, A.Y);
    printf("B = (%d, %d)\n", B.X, B.Y);
    Tpoint_MoveTo(A, A.X, B.X);
    printf("A = (%d, %d)\n", A.X, A.Y);
    return EXIT_SUCCESS;
}
```

```
$ ./a.out
O = (0, 0)
A = (6, 0)
B = (5, 7)
A = (6, 7)
```

```
// fichier main.cpp
#include <stdio.h>
#include <stdlib.h>
#include "point.h"

int main(void)
{
    TPoint O, A(6), B(5, 7);
    printf("O = (%d, %d)\n", O.GetX(), O.GetY());
    printf("A = (%d, %d)\n", A.GetX(), A.GetY());
    printf("B = (%d, %d)\n", B.GetX(), B.GetY());
    A.MoveTo(A.GetX(), B.GetY());
    printf("A = (%d, %d)\n", A.GetX(), A.GetY());
    return EXIT_SUCCESS;
}
```

TYPES ABSTRAITS DE DONNÉES

Programmation orientée objet en C (1/2)

Le langage C n'est pas un langage orienté objet mais il est possible de se rapprocher de la POO en adoptant quelques règles d'écriture strictes :

1. Toutes les fonctions publiques doivent faire référence à une structure C (struct) qui représente l'objet. Concrètement, elles doivent toujours recevoir un pointeur sur la structure comme premier paramètre (*this*).
2. Toutes les fonctions publiques doivent être associé à un pointeur sur fonction dans la structure. Ces pointeurs de fonction sont l'équivalent des *méthodes* du C++.
3. À l'utilisation de la classe, on n'appelle jamais une fonction publique directement, mais à l'aide du pointeur sur fonction associé dans la structure.
4. À l'utilisation de la classe, on n'accède jamais aux données (*attributs*) de la structure, mais toujours à l'aide de *méthodes* de la structure : les *accesseurs* (getters) et les *mutateurs* (setters).

TYPES ABSTRAITS DE DONNÉES

Programmation orientée objet en C (2/2)

5. Pour éviter les redondances des noms de fonctions publiques, on préfixe leur nom du nom de la structure (*Type_*).
6. Les pointeurs sur fonction membres de la structure (les *méthodes*) auront par contre un nom court, non préfixé, puisque qu'il n'y a pas de risque de redondance.
7. Les objets créées sur la base de cette structure doivent l'être avec la fonction *Type_New* qui renvoie un pointeur vers l'objet créé. Cette fonction n'a pas de pointeur correspondant dans la structure, l'instance de la structure n'étant pas encore créée au moment de l'appel. Cette fonction est l'équivalent du *constructeur* du C++, c'est elle qui crée l'objet et doit donc être appelée avant toute utilisation d'un objet.
8. La mémoire allouée le constructeur sera libérée par la fonction *Type_Delete* quand l'objet ne sera plus utile. Cette fonction est l'équivalent du *destructeur* du C++.
9. Chaque "classe" doit être déclarée dans un fichier d'en-tête spécifique, et les fonctions correspondant aux méthodes doivent être implémentées dans un fichier source spécifique.

TYPES ABSTRAITS DE DONNÉES

POO en C : fichier *point.h*

```
#ifndef _POINT_H
#define _POINT_H

#include <stdbool.h>

typedef struct Point {
    int X, Y;
    bool visible;
    int(*GetX)(struct Point *);
    int(*GetY)(struct Point *);
    void(*MoveTo)(struct Point *, int, int);
    void(*Show)(struct Point *);
    void(*Hide)(struct Point *);
    bool(*IsVisible)(struct Point *);
    void(*Delete)(struct Point *); /* Destructeur */
} TPoint;

typedef TPoint *PTPoint;

PTPoint TPoint_New(int, int); /* Constructeur */

#endif
```

TYPES ABSTRAITS DE DONNÉES

POO en C : fichier *point.c* (constructeur)

```
#include <stdlib.h>
#include "point.h"

/* Déclaration des fonctions associés à la classe */
int TPoint_GetX(PTPoint);
int TPoint_GetY(PTPoint);
void TPoint_MoveTo(PTPoint, int, int);
void TPoint_Show(PTPoint);
void TPoint_Hide(PTPoint);
bool TPoint_IsVisible(PTPoint);
void TPoint_Delete(PTPoint);
/* Implémentation du constructeur */
PTPoint TPoint_New(int initX, int initY)
{
    PTPoint this = malloc(sizeof(TPoint));
    this->X = initX;
    this->Y = initY;
    this->Visible = false;
    this->GetX = (void *)TPoint_GetX;
    this->GetY = (void *)TPoint_GetY;
    this->MoveTo = (void *)TPoint_MoveTo;
    this->Show = (void *)TPoint_Show;
    this->IsVisible = (void *)TPoint_IsVisible;
    this->Delete = (void *)TPoint_Delete;
    return this;
}
```

TYPES ABSTRAITS DE DONNÉES

POO en C : fichier *point.c* (destructeur et autres fonctions)

```
int TPoint_GetX(PTPoint this) {
    return this->X;
}
int TPoint_GetY(PTPoint this) {
    return this->Y;
}
void TPoint_MoveTo(PTPoint this, int newX, int newY)
{
    TPoint_Hide(this);
    this->X = newX;
    this->Y = newY;
    TPoint_Show(this);
}
void TPoint_Show(PTPoint this) {
    this->Visible = true;
}
void TPoint_Hide(PTPoint this) {
    this->Visible = false;
}
bool TPoint_IsVisible(PTPoint this) {
    return this->Visible;
}
void TPoint_Delete(PTPoint this) /* Implementation du destructeur */
{
    free(this);
}
```

TYPES ABSTRAITS DE DONNÉES

POO en C : exemple de programme

```
#include <stdio.h>
#include <stdlib.h>
#include "point.h"

int main(void)
{
    PTPoint O = TPoint_New(0, 0);
    PTPoint A = TPoint_New(6, 0);
    PTPoint B = TPoint_New(5, 7);
    printf("O = (%d, %d)\n", O->GetX(O), O->GetY(O));
    printf("A = (%d, %d)\n", A->GetX(A), A->GetY(A));
    printf("B = (%d, %d)\n", B->GetX(B), B->GetY(B));
    A->MoveTo(A, A->GetX(A), B->GetY(B));
    printf("A = (%d, %d)\n", A->GetX(A), A->GetY(A));
    O->Delete(O);
    A->Delete(A);
    B->Delete(B);
    return EXIT_SUCCESS;
}
```

```
$ ./a.out
O = (0, 0)
A = (6, 0)
B = (5, 7)
A = (6, 7)
```

TYPES ABSTRAITS DE DONNÉES

L'héritage en C++

- **L'héritage** : on peut dériver une classe fille d'une classe de base. La classe dérivée hérite alors de toutes les propriétés et fonctions de la classe mère.
- Exemple :

```
class TPixel : public TPoint { // TPixel dérive publiquement de TPoint
private:
    byte R, G, B;
public:
    TPixel(int = 0, int = 0, int = 0x808080); // Constructeur
    void SetColor(int color);
    int GetColor();
};
```

- **Le polymorphisme** : une classe dérivée peut redéfinir les méthodes de la classe parente. Un objet de la classe dérivée pourra donc exécuter les méthodes homonymes de la classe de base à sa manière propre.

TYPES ABSTRAITS DE DONNÉES

Héritage en C++ : la classe *TPixel*

```
// fichier pixel.h
#ifndef _PIXEL_H
#define _PIXEL_H

#include "point.h"

typedef unsigned char byte;

class TPixel : public TPoint {
private:
    byte R, G, B;
public:
    // Constructeur
    TPixel(int = 0, int = 0, int = 0x808080);
    void SetColor(int color);
    int GetColor();
};

#endif
```

```
// fichier pixel.cpp
#include "pixel.h"

TPixel::TPixel(int initX, int initY, int initColor):
    TPoint(initX, initY)
{
    R = (byte)(initColor >> 16);
    G = (byte)(initColor >> 8);
    B = (byte)(initColor);
}

void TPixel::SetColor(int color)
{
    R = (byte)(color >> 16);
    G = (byte)(color >> 8);
    B = (byte)(color);
}

int TPixel::GetColor()
{
    return (int)R << 16 | (int)G << 8 | (int)B;
}
```

TYPES ABSTRAITS DE DONNÉES

Héritage en C : fichier *pixel.h*

- Comme en C++ la classe héritée reprend toutes les propriétés et méthodes de la classe de base.
- Le début de la classe héritée est un copier-coller de la classe de base.
- Les champs supplémentaires sont à ajouter à la suite.
- Les méthodes étant appelées par l'intermédiaire de pointeurs de fonctions, le polymorphisme est tout à fait possible !

```

#ifndef _PIXEL_H
#define _PIXEL_H

#include <stdbool.h>
#include <stdint.h>

typedef struct Pixel {
    int X, Y;
    bool Visible;
    int(*GetX)(struct Pixel *);
    int(*GetY)(struct Pixel *);
    void(*MoveTo)(struct Pixel *, int, int);
    void(*Show)(struct Pixel *);
    void(*Hide)(struct Pixel *);
    bool(*IsVisible)(struct Pixel *);
    void(*Delete)(struct Pixel *); /* Destructeur */
    byte R, G, B;
    void (*SetColor)(struct Pixel *, int);
    int (*GetColor)(struct Pixel *);
} TPixel;

typedef TPixel *PTPixel;

PTPixel TPixel_New(int, int, int); /* Constructeur */

#endif

```

TYPES ABSTRAITS DE DONNÉES

Héritage en C : fichier *pixel.c* : déclarations des prototypes

```
#include <stdlib.h>
#include "pixel.h"

/* Prototypes des fonctions de la classe de base */
extern int TPoint_GetX(PTPoint);
extern int TPoint_GetY(PTPoint);
extern void TPoint_MoveTo(PTPoint, int, int);
extern void TPoint_Show(PTPoint);
extern void TPoint_Hide(PTPoint);
extern bool TPoint_IsVisible(PTPoint);
extern void TPoint_Delete(PTPoint);

/* Prototypes des fonctions de la classe héritée */
void TPixel_SetColor(PTPixel, int);
int TPixel_GetColor(PTPixel);
void TPixel_Delete(PTPixel);
```

TYPES ABSTRAITS DE DONNÉES

Héritage en C : fichier *pixel.c* : constructeur

```
/* Implementation du constructeur */
TPixel *TPixel_New(int initX, int initY, int initColor)
{
    PTPixel this = malloc(sizeof(TPixel));
    this->X = initX;
    this->Y = initY;
    this->visible = false;
    this->GetX = (void *)TPoint_GetX;
    this->GetY = (void *)TPoint_GetY;
    this->MoveTo = (void *)TPoint_MoveTo;
    this->>Show = (void *)TPoint_Show;
    this->IsVisible = (void *)TPoint_IsVisible;
    this->R = (byte)(initColor >> 16);
    this->G = (byte)(initColor >> 8);
    this->B = (byte)(initColor);
    this->SetColor = (void *)TPixel_SetColor;
    this->GetColor = (void *)TPixel_GetColor;
    this->Delete = (void *)TPixel_Delete;
    return this;
} /* Constructeur */
```

TYPES ABSTRAITS DE DONNÉES

Héritage en C : fichier *pixel.c* : destructeur et autres fonctions

```
void TPixel_SetColor(PTPixel this, int color)
{
    this->R = (byte)(color >> 16);
    this->G = (byte)(color >> 8);
    this->B = (byte)(color);
}

int TPixel_GetColor(PTPixel this)
{
    return (int)this->R << 16 | (int)this->G << 8 |
           (int)this->B;
}

/* Implementation du destructeur */
void TPixel_Delete(PTPixel this)
{
    free(this);
}
```

PLAN DU COURS



Méthodes de développement logiciel

Rappels et approfondissements en C

Types abstraits de données

Listes chainées

Piles et files

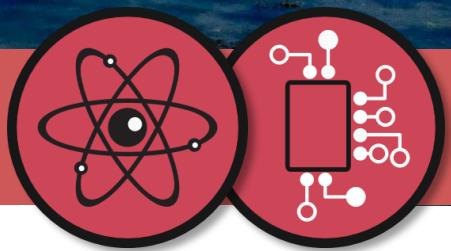
Algorithmes de tri

Tables de hachage

Arbres binaires



Listes chainées



Implémentation d'une liste simple avec un pointeur

Implémentation à l'aide d'une structure

Implémentation orientée objet - Listes doublement chainées

LES LISTES CHAINÉES

Définition d'une liste

- Une liste chaînée est une structure de données représentant une collection ordonnée et de *taille arbitraire* d'éléments de même type, voire de types différents. L'accès aux éléments d'une liste se fait de *manière séquentielle* : chaque élément permet l'accès au suivant.
- Les listes ont de nombreux avantages sur les tableaux :
 - Pas de taille fixée à priori,
 - Possibilité d'insérer et de supprimer des éléments de manière efficace, sans de coûteuses recopies de données.
- Par contre, l'accès à un élément se fait de manière séquentielle, alors que les tableaux sont indexés : il faut parcourir tous les éléments précédents celui à atteindre.

LES LISTES CHAINÉES

Implémentation d'une liste chaînée simple en C

- Une liste chaînée peut être implémentée simplement à l'aide d'une structure C désignée sous le terme de *Nœud* (*Node*) :

```
typedef struct Node {
    TElement Element;
    struct Node *Next;
} TNode;
```

- La liste peut alors être représentée par un simple pointeur vers le premier élément, le dernier élément pointant vers NULL :



```
typedef TNode *PTNode;
typedef PTNode TList;
typedef TList *PTList;
TList Liste;
```

LES LISTES CHAINÉES

Opérations sur le TAD liste

- **Nom :** TList,
- **Utilise :** int, bool, TElement, TNode
- **Opérations :**
 - New → TList
 - IsEmpty : TList → bool
 - Length : TList → int
 - GoTo : TList × int → TNode
 - InsertFirst : TList × TElement → TNode × TList
 - RemoveFirst : TList → bool × TList
 - Add : TList × TElement → TNode × TList
 - RemoveLast : TList → bool
 - Display : TList →
 - Delete : TList → TList

LES LISTES CHAINÉES

Fichier *list.h* : déclarations des types et des fonctions

```
#ifndef _LIST_H
#define _LIST_H

#include <stdbool.h>
#include "element.h"

typedef struct Node {
    TElement Element;
    struct Node *Next;
} TNode;
/* Déclaration de la structure Noeud */

typedef TNode *PTNode;
/* Type Noeud */
typedef PTNode TList;
/* Type liste */
typedef TList *PTList;
/* Type pointeur vers Liste */

TList TList_New(void);
/* Retourne une liste vide */
bool TList_IsEmpty(const PTList);
/* Vrai si la liste est vide */
int TList_Length(const PTList);
/* Retourne la taille de la liste */
PTNode TList_GoTo(const PTList, int);
/* Va à une position donnée dans la liste */
PTNode TList_InsertFirst(PTList, TElement);
/* Insère un élément en début de liste */
bool TList_RemoveFirst(PTList);
/* Retire l'élément en début de liste */
PTNode TList_Add(PTList, TElement);
/* Ajoute un élément en fin de liste */
bool TList_RemoveLast(PTList);
/* Retire le dernier élément de la liste */
void TList_Display(const PTList);
/* Affiche tous les éléments de la liste */
void TList_Delete(PTList);
/* Supprime complètement la liste */

#endif
```

LES LISTES CHAINÉES

Fonctions : création d'une liste vide – teste si liste vide – taille de la liste

```
#include <stdlib.h>
#include "list.h"

TList TList_New(void)          /* Création d'une nouvelle liste vide */
{
    return NULL;
}

bool TList_IsEmpty(const PTLList this) /* Teste si la liste est vide */
{
    return (*this == NULL);
}

int TList_Length(const PTLList this) /* Renvoie la taille de la liste */
{
    int length = 0;
    PTNode pNode = *this;
    while (pNode) {
        pNode = pNode->Next;
        length++;
    }
    return length;
}
```

LES LISTES CHAINÉES

Fonction TList_Goto : aller à une position donnée dans la liste

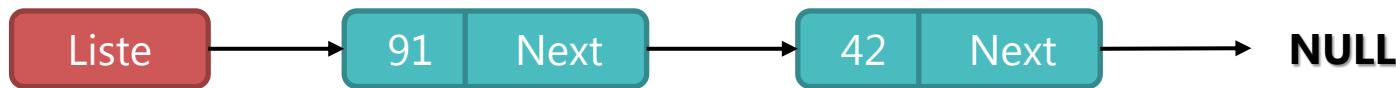
- Va à l'élément situé à la position *Pos* dans la liste
- Retourne un pointeur sur l'élément trouvé ou NULL en cas d'échec

```
#include <stdlib.h>
#include "list.h"

PTNode TList_GoTo(const PTList this, int Pos)
{
    if (TList_IsEmpty(this)) return NULL;
    PTNode pNode = *this;
    int index = 0;
    while (index < Pos) {
        if (pNode->Next == NULL) return NULL;
        pNode = pNode->Next;
        index++;
    }
    return pNode;
}
```

LES LISTES CHAINÉES

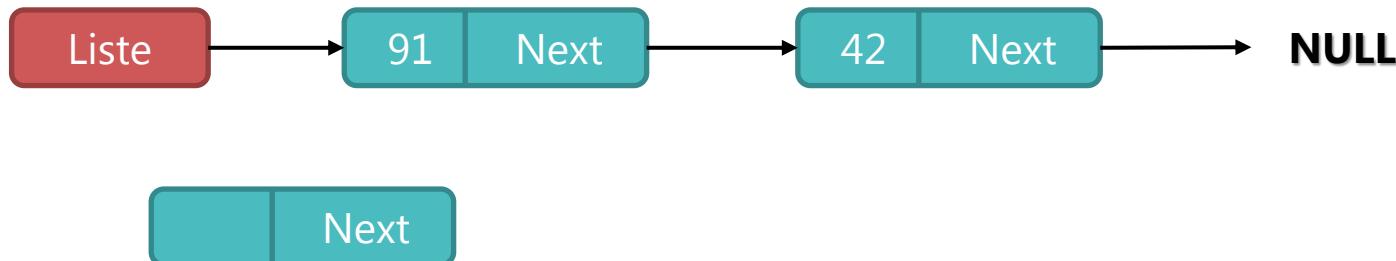
Fonction `TList_InsertFirst` : insertion d'un élément en début de liste (1/5)



```
PTNode TList_InsertFirst(PTList this, TElement Element)  
{
```

LES LISTES CHAINÉES

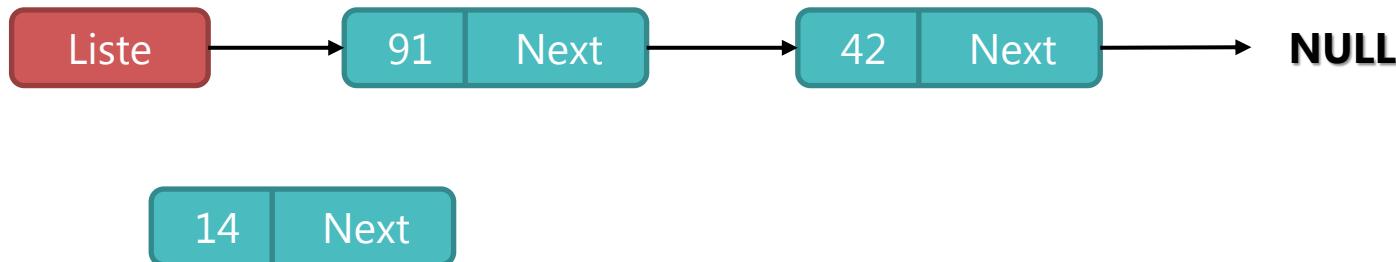
Fonction `TList_InsertFirst` : insertion d'un élément en début de liste (2/5)



```
PTNode TList_InsertFirst(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
```

LES LISTES CHAINÉES

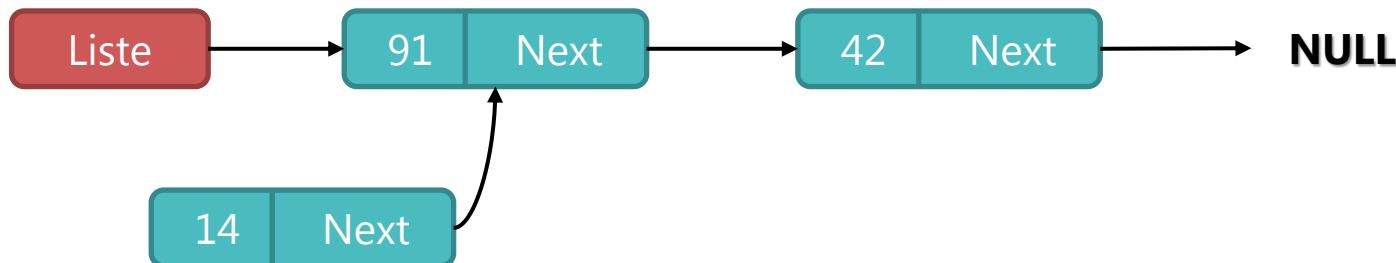
Fonction `TList_InsertFirst` : insertion d'un élément en début de liste (3/5)



```
PTNode TList_InsertFirst(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : insertion d'un élément en début de liste (4/5)

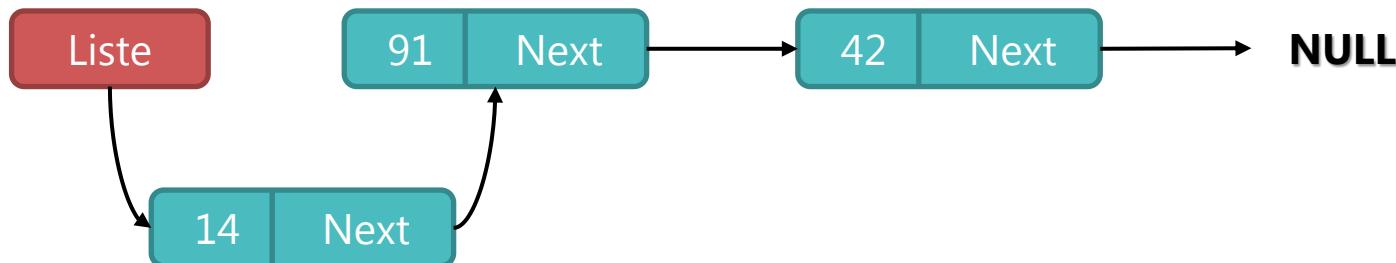


```

PTNode TList_InsertFirst(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = *this;
  
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : insertion d'un élément en début de liste (5/5)

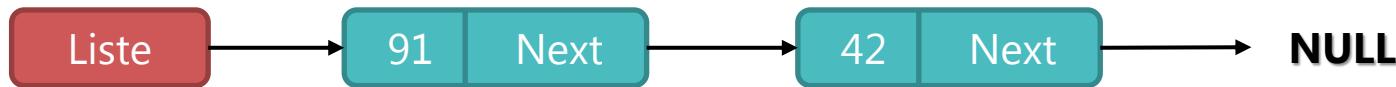


```

PTNode TList_InsertFirst(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = *this;
    *this = newNode;
    return newNode;
}
  
```

LES LISTES CHAINÉES

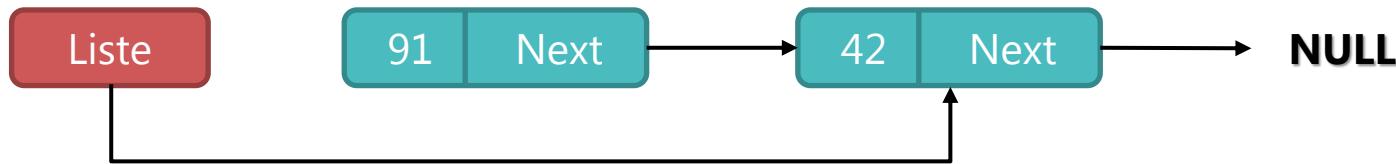
Fonction `TList_RemoveFirst` : retrait de l'élément en début de liste (1/3)



```
bool TList_RemoveFirst(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = *this, nextNode = pNode->Next;
```

LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait de l'élément en début de liste (2/3)



```
bool TList_RemoveFirst(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = *this, nextNode = pNode->Next;
    *this = nextNode;
```

LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait de l'élément en début de liste (3/3)

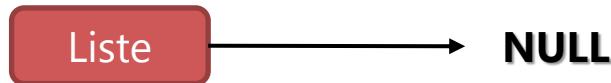


```
bool TList_RemoveFirst(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = *this, nextNode = pNode->Next;
    *this = nextNode;
    free(pNode);
    return true;
}
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (1/8)

1) Liste vide

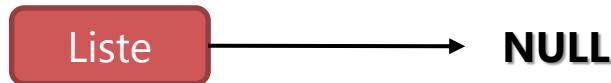


```
PTNode TList_Add(PTList this, TElement Element)  
{
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (2/8)

1) Liste vide

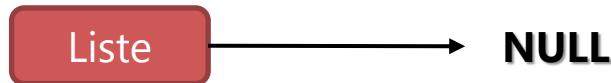


```
PTNode TList_Add(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (3/8)

1) Liste vide



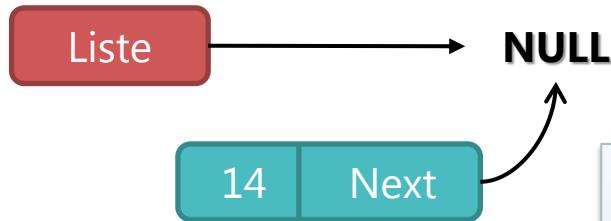
14 Next

```
PTNode TList_Add(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (4/8)

1) Liste vide

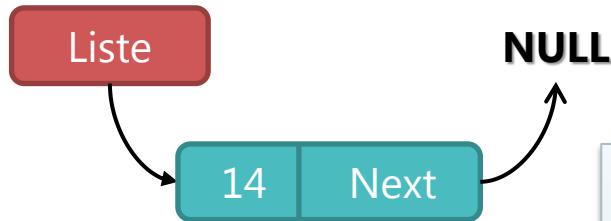


```
PTNode TList_Add(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (5/8)

1) Liste vide : similaire à l'insertion en début de liste

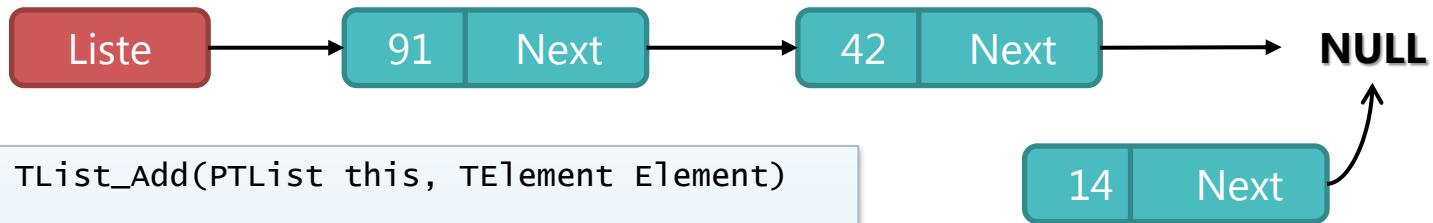


```
PTNode TList_Add(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
    if (TList_IsEmpty(this))
        *this = newNode;
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (6/8)

2) Liste non vide

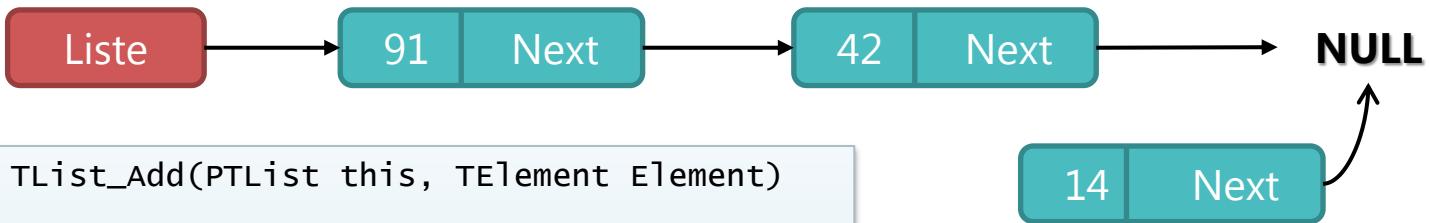


```
PTNode TList_Add(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
    if (TList_IsEmpty(this))
        *this = newNode;
    else {
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (7/8)

2) Liste non vide : avancer jusqu'au dernier élément

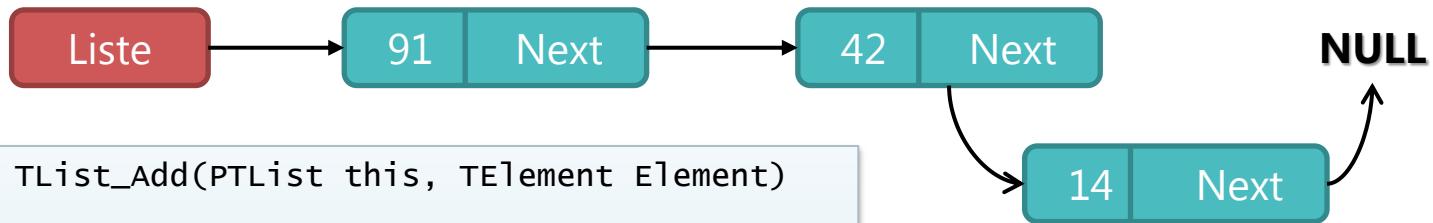


```
PTNode TList_Add(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
    if (TList_IsEmpty(this))
        *this = newNode;
    else {
        PTNode pNode = *this;
        while (pNode->Next) pNode = pNode->Next;
    }
}
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (8/8)

2) Liste non vide : l'ajout ne change pas la valeur du pointeur *Liste*



```
PTNode TList_Add(PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
    if (TList_IsEmpty(this))
        *this = newNode;
    else {
        PTNode pNode = *this;
        while (pNode->Next) pNode = pNode->Next;
        pNode->Next = newNode;
    }
    return newNode;
}
```

LES LISTES CHAINÉES

Fonction `TList_RemoveLast` : retrait d'un élément en fin de liste (1/6)

1) Un seul élément dans la liste



```
bool TList_RemoveLast(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode prevNode = *this, pNode = prevNode->Next;
```

LES LISTES CHAINÉES

Fonction TList_RemoveLast : retrait d'un élément en fin de liste (2/6)

1) Un seul élément dans la liste



```
bool TList_RemoveLast(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode prevNode = *this, pNode = prevNode->Next;
    if (pNode == NULL) { /* 1 seul élément dans la liste */
        *this = NULL;
```

LES LISTES CHAINÉES

Fonction TList_RemoveLast : retrait d'un élément en fin de liste (3/6)

1) Un seul élément dans la liste

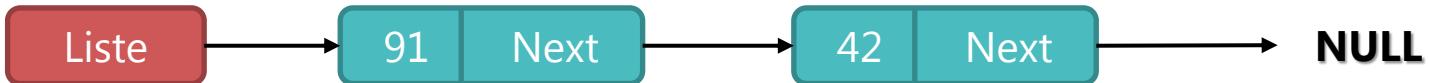


```
bool TList_RemoveLast(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode prevNode = *this, pNode = prevNode->Next;
    if (pNode == NULL) { /* 1 seul élément dans la liste */
        *this = NULL;
        free(prevNode);
    }
}
```

LES LISTES CHAINÉES

Fonction `TList_RemoveLast` : retrait d'un élément en fin de liste (4/6)

2) Plus d'un élément dans la liste



```

bool TList_RemoveLast(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode prevNode = *this, pNode = prevNode->Next;
    if (pNode == NULL) { /* 1 seul élément dans la liste */
        *this = NULL;
        free(prevNode);
    }
    else {
        while (pNode->Next) {
            prevNode = pNode;
            pNode = pNode->Next;
        }
    }
}
  
```

LES LISTES CHAINÉES

Fonction `TList_RemoveLast` : retrait d'un élément en fin de liste (5/6)

2) Plus d'un élément dans la liste

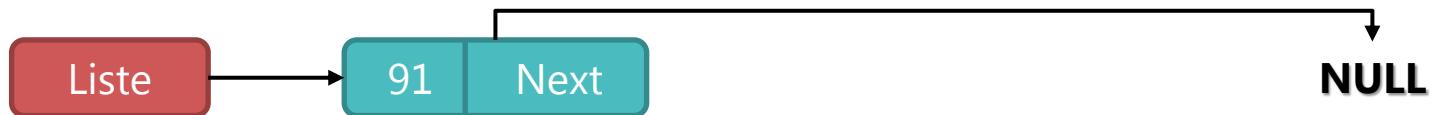


```
bool TList_RemoveLast(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode prevNode = *this, pNode = prevNode->Next;
    if (pNode == NULL) { /* 1 seul élément dans la liste */
        *this = NULL;
        free(prevNode);
    }
    else {
        while (pNode->Next) {
            prevNode = pNode;
            pNode = pNode->Next;
        }
        prevNode->Next = NULL;
    }
}
```

LES LISTES CHAINÉES

Fonction `TList_RemoveLast` : retrait d'un élément en fin de liste (6/6)

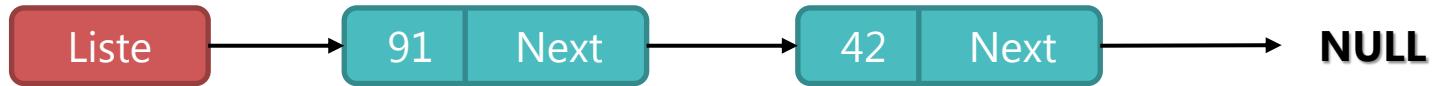
2) Plus d'un élément dans la liste



```
bool TList_RemoveLast(PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode prevNode = *this, pNode = prevNode->Next;
    if (pNode == NULL) { /* 1 seul élément dans la liste */
        *this = NULL;
        free(prevNode);
    }
    else {
        while (pNode->Next) {
            prevNode = pNode;
            pNode = pNode->Next;
        }
        prevNode->Next = NULL;
        free(pNode);
    }
    return true;
}
```

LES LISTES CHAINÉES

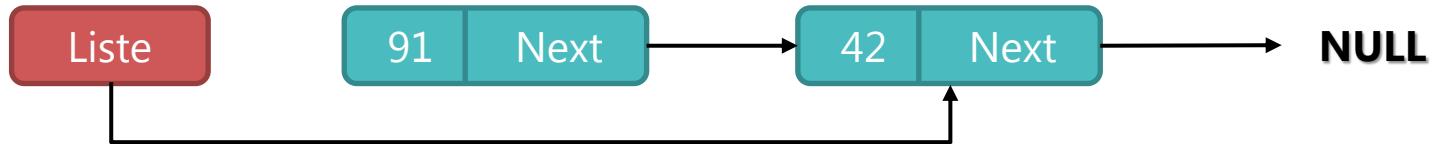
Fonction TList_Delete : suppression complète de la liste (1/5)



```
void TList_Delete(PTList this)
{
    while (TList_RemoveFirst(this));
}
```

LES LISTES CHAINÉES

Fonction TList_Delete : suppression complète de la liste (2/5)



```
void TList_Delete(PTList this)
{
    while (TList_RemoveFirst(this));
}
```

LES LISTES CHAINÉES

Fonction TList_Delete : suppression complète de la liste (3/5)



```
void TList_Delete(PTList this)
{
    while (TList_RemoveFirst(this));
}
```

LES LISTES CHAINÉES

Fonction TList_Delete : suppression complète de la liste (4/5)



```
void TList_Delete(PTList this)
{
    while (TList_RemoveFirst(this));
}
```

LES LISTES CHAINÉES

Fonction TList_Delete : suppression complète de la liste (5/5)

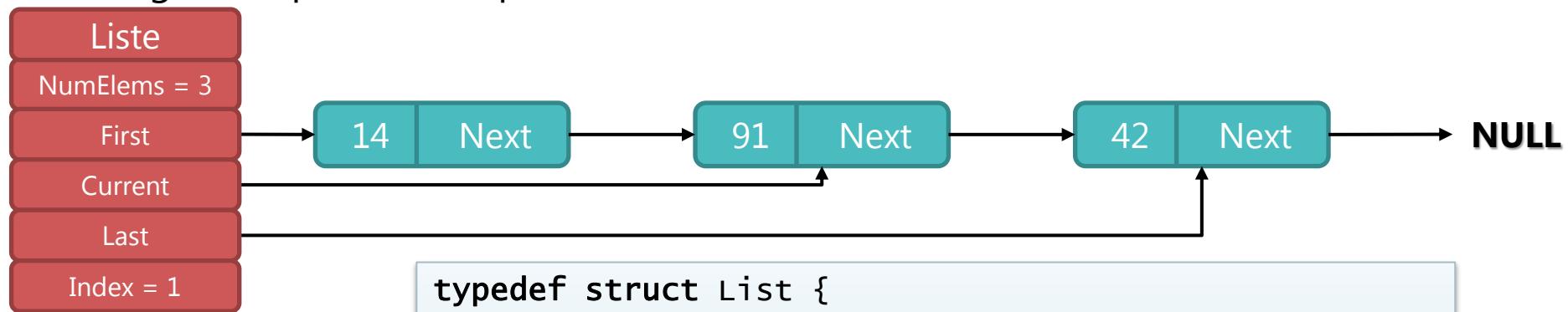


```
void TList_Delete(PTList this)
{
    while (TList_RemoveFirst(this));
}
```

LES LISTES CHAINÉES

Nouvelle implémentation à l'aide d'une structure

- Une représentation du type *Liste* sous forme de structure C permet une gestion plus fine et plus efficace de la liste :



```

typedef struct List {
    int NumElems;      /* Nombre d'éléments */
    PTNode First;      /* Pointe sur le premier élément */
    PTNode Last;       /* Pointe sur le dernier élément */
    PTNode Current;   /* Pointe sur l'élément courant */
    int Index;         /* Indice de l'élément courant */
} TList;

typedef TList *PTList;
  
```

LES LISTES CHAINÉES

Opérations sur le TAD liste

- **Nom :** TList,
- **Utilise :** int, bool, TElement, TNode
- **Opérations :**
 - New → TList
 - IsEmpty : TList → bool
 - Length : TList → int
 - GetIndex : TList → int
 - GoTo : TList x int → TNode
 - InsertFirst : TList x TElement → TNode
 - RemoveFirst : TList → bool
 - Add : TList x TElement → TNode
 - RemoveLast : TList → bool
 - Insert : TList x TElement → TNode
 - RemoveCurrent : TList → bool
 - Display : TList →
 - Delete : TList →

LES LISTES CHAINÉES

Fichier *list.h* : déclarations des types (1/2)

```
#ifndef _LIST_H
#define _LIST_H

#include <stdbool.h>
#include "element.h"

typedef struct Node {
    TElement Element;
    struct Node *Next;
} TNode;

typedef TNode *PTNode;

/* Déclaration de la structure Noeud */

typedef struct List {
    int NumElems;
    PTNode First;
    PTNode Last;
    PTNode Current;
    int Index;
} TList;

typedef TList *PTList;

/* Type Noeud */
/* Type pointeur vers Noeud */

/* Déclaration de la structure Liste */
/* Nombre d'éléments de la liste */
/* Pointeur vers le premier élément */
/* Pointeur vers le dernier élément */
/* Pointeur vers l'élément courant */
/* Indice de l'élément courant */
/* Type Liste */

/* Type pointeur vers Liste */
```

LES LISTES CHAINÉES

Fichier *list.h* : déclarations des fonctions (2/2)

```

TList TList_New(void);           /* Crée une nouvelle liste vide */
bool TList_IsEmpty(const PTList); /* Vrai si la liste est vide */
int TList_Length(const PTList);  /* Retourne la taille de la liste */
int TList_GetIndex(const PTList); /* Retourne la position courante */
PTNode TList_GoTo(const PTList, int); /* Va à une position donnée dans la liste */

/* Insère un élément en début de liste */
PTNode TList_InsertFirst(const PTList, TElement);
/* Retire l'élément en début de liste */
bool TList_RemoveFirst(const PTList);
/* Ajoute un élément en fin de liste */
PTNode TList_Add(const PTList, TElement);
/* Retire le dernier élément de la liste */
bool TList_RemoveLast(const PTList);
/* Insère un élément à la position courante*/
PTNode TList_Insert(const PTList, TElement);
/* Retire l'élément courant */
bool TList_RemoveCurrent(const PTList);

void TList_Display(const PTList); /* Affiche tous les éléments de la liste */
void TList_Delete(const PTList);  /* Supprime complètement la liste */

#endif

```

LES LISTES CHAINÉES

Fonctions : création d'une liste vide – teste si liste vide – taille de la liste

```
#include <stdlib.h>
#include "list.h"

TList TList_New(void)          /* Création d'une nouvelle liste vide */
{
    PTList this = malloc(sizeof(TList));
    this->NumElems = 0;
    this->First = NULL;
    this->Last = NULL;
    this->Current = NULL;
    this->Index = -1;
    return this;
}

bool TList_IsEmpty(const PTList this) /* Teste si la liste est vide */
{
    return (this->NumElems == 0);
}

int TList_Length(const PTList this)   /* Renvoie la taille de la liste */
{
    return this->NumElems;
}
```

LES LISTES CHAINÉES

Parcours de la liste

- *GoTo* positionne le pointeur courant sur l'élément situé à la position *Pos* dans la liste et met à jour l'indice, puis renvoie le pointeur courant.
- *GetIndex* est un accesseur qui retourne l'indice courant.

```
#include <stdlib.h>
#include "list.h"

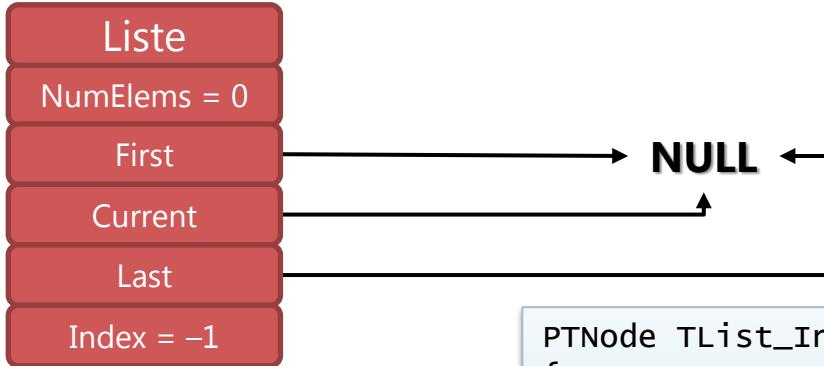
PTNode TList_GoTo(const PTList this, int Pos)
{
    /* Position inatteignable ! */
    if (this->NumElems <= Pos) return NULL;
    /* On part du début de la liste */
    this->Current = this->First;
    this->Index = 0;
    while (this->Index < Pos) {
        this->Current = this->Current->Next;
        this->Index++;
    }
    return this->Current;
}

int TList_GetIndex(const PTList this)
{
    return this->Index;
}
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : ajout d'un élément en début de liste (1/11)

1) Liste vide

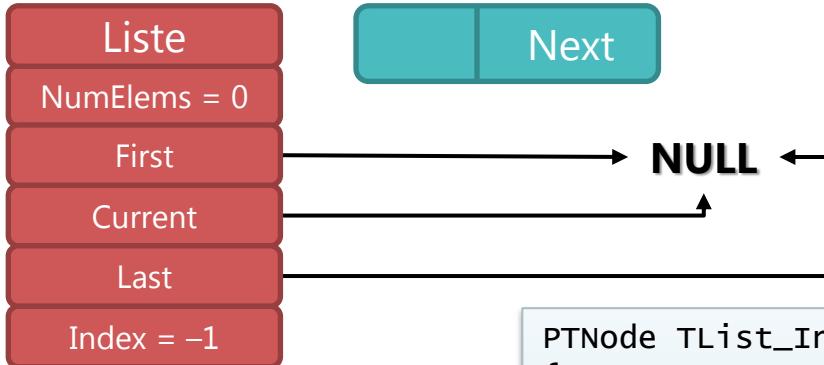


```
PTNode TList_InsertFirst(const PTLList this, TElement Element)  
{
```

LES LISTES CHAINÉES

Fonction TList_InsertFirst : ajout d'un élément en début de liste (2/11)

1) Liste vide

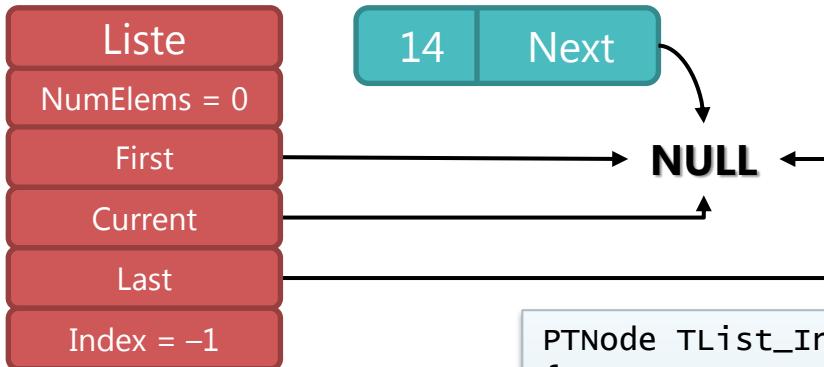


```
PTNode TList_InsertFirst(const PTLList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : ajout d'un élément en début de liste (3/11)

1) Liste vide

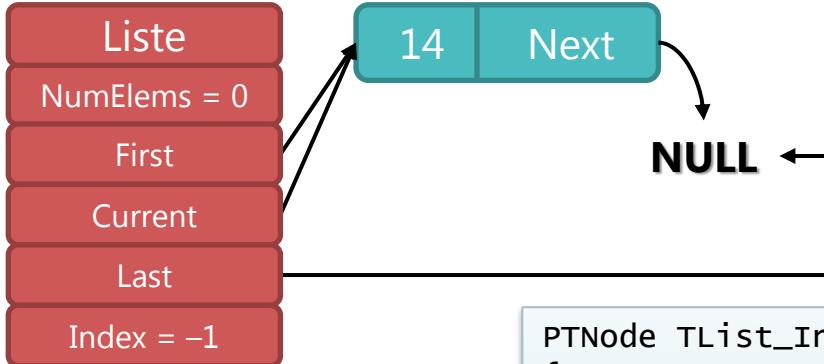


```
PTNode TList_InsertFirst(const PTLList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = this->First;
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : ajout d'un élément en début de liste (4/11)

1) Liste vide

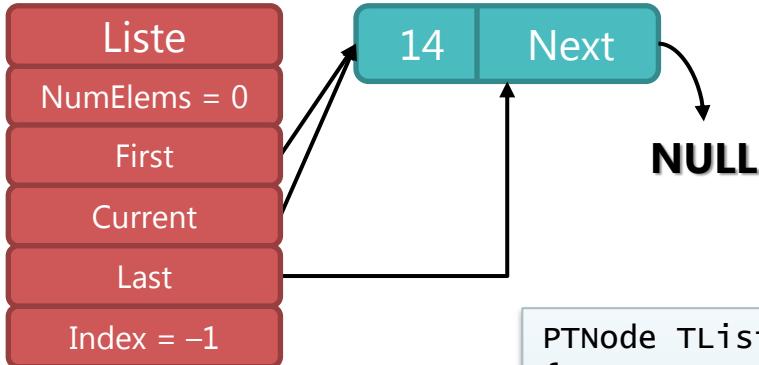


```
PTNode TList_InsertFirst(const PTLList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = this->First;
    this->Current = this->First = newNode;
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : ajout d'un élément en début de liste (5/11)

1) Liste vide

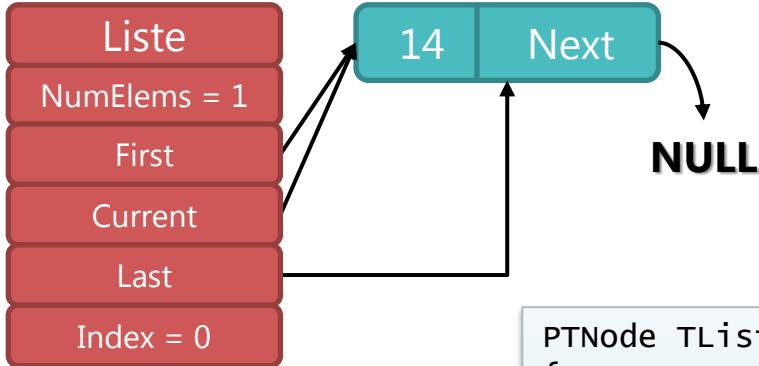


```
PTNode TList_InsertFirst(const PTLList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = this->First;
    this->Current = this->First = newNode;
    if (TList_IsEmpty(this)) this->Last = newNode;
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : ajout d'un élément en début de liste (6/11)

1) Liste vide

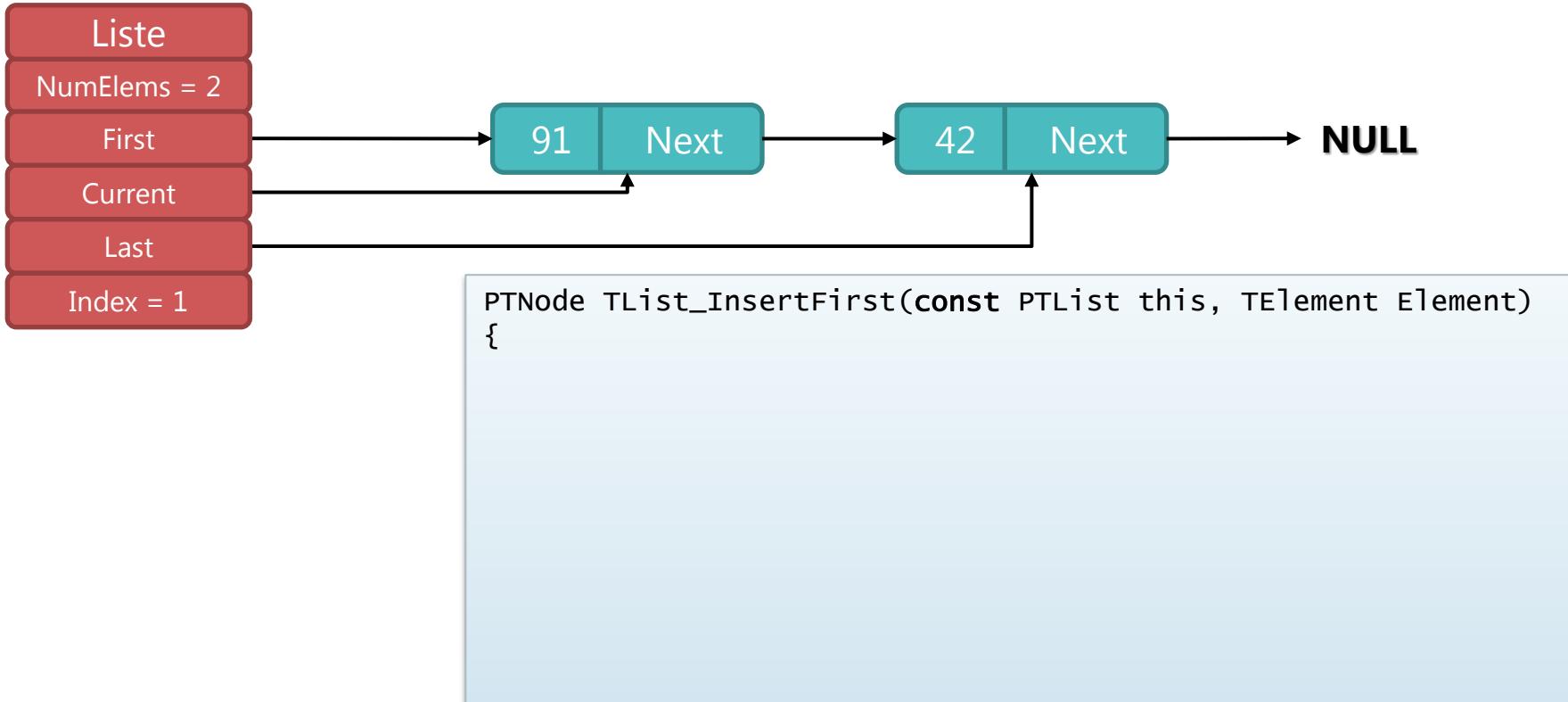


```
PTNode TList_InsertFirst(const PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = this->First;
    this->Current = this->First = newNode;
    if (TList_IsEmpty(this)) this->Last = newNode;
    this->NumElems++;
    this->Index = 0;
    return newNode;
}
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : ajout d'un élément en début de liste (7/11)

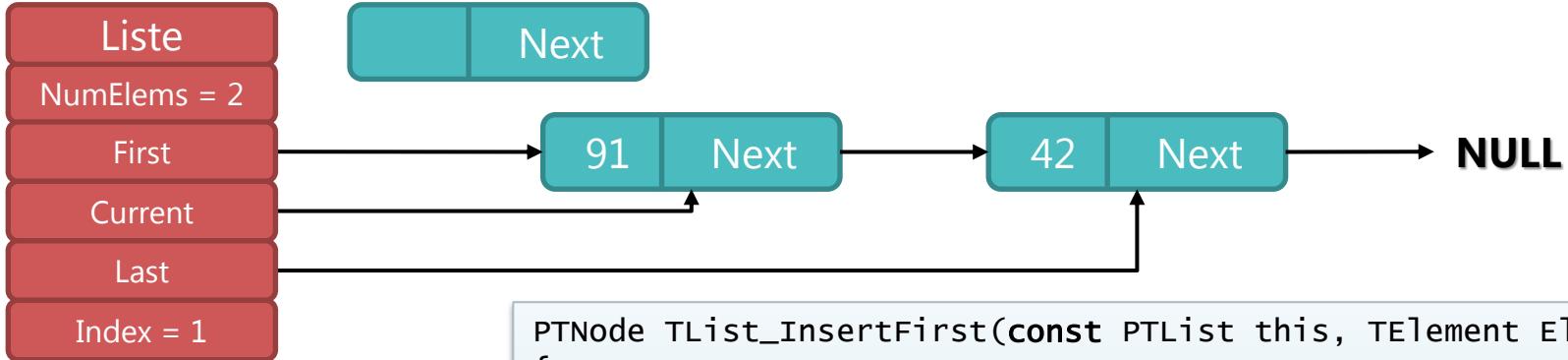
2) Liste non vide



LES LISTES CHAINÉES

Fonction TList_InsertFirst : ajout d'un élément en début de liste (8/11)

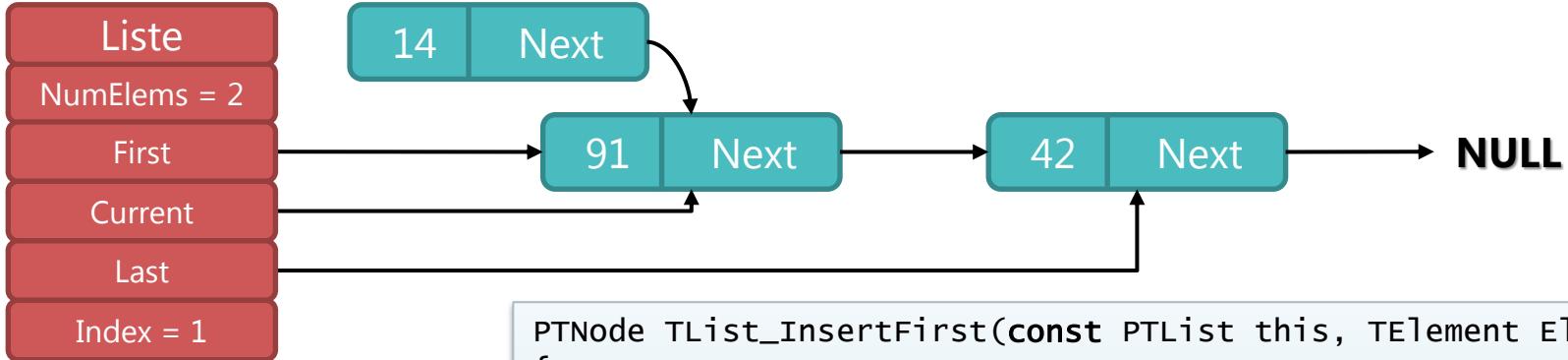
2) Liste non vide



LES LISTES CHAINÉES

Fonction TList_InsertFirst : ajout d'un élément en début de liste (9/11)

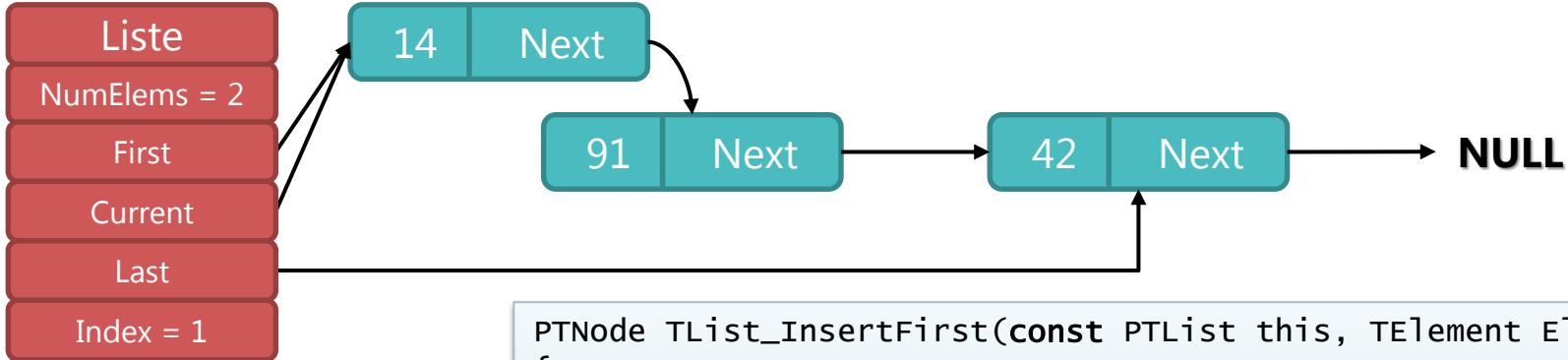
2) Liste non vide



LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : ajout d'un élément en début de liste (10/11)

2) Liste non vide

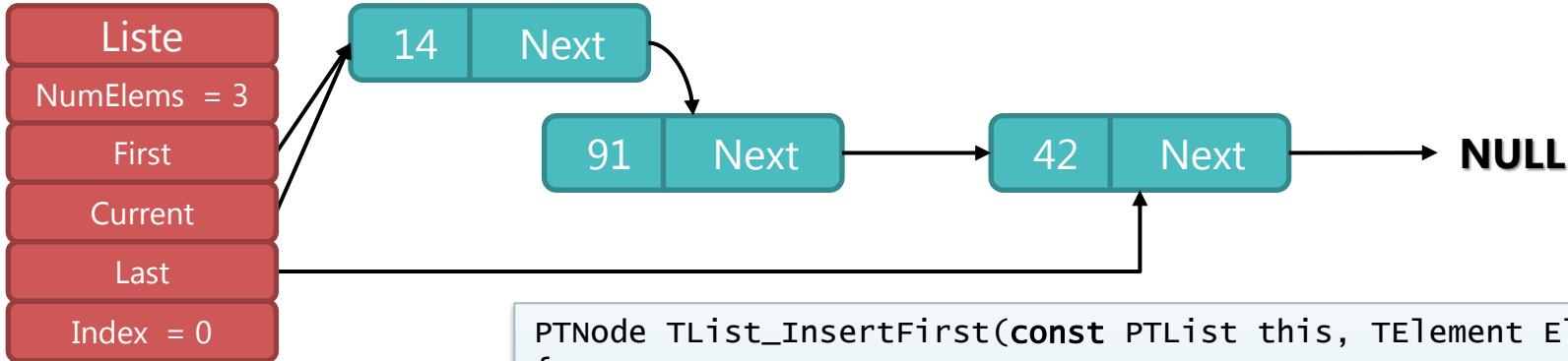


```
PTNode TList_InsertFirst(const PTLList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = this->First;
    this->Current = this->First = newNode;
```

LES LISTES CHAINÉES

Fonction `TList_InsertFirst` : ajout d'un élément en début de liste (11/11)

2) Liste non vide



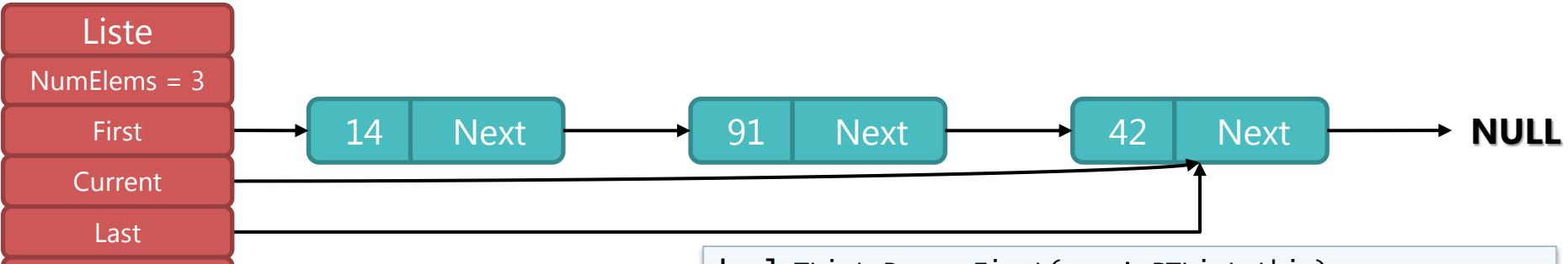
```

PTNode TList_InsertFirst(const PTLList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->element = Element;
    newNode->Next = this->First;
    this->Current = this->First = newNode;
    if (TList_IsEmpty(this)) this->Last = newNode;
    this->NumElems++;
    this->Index = 0;
    return newNode;
}
  
```

LES LISTES CHAINÉES

Fonction TList_RemoveFirst : retrait d'un élément en début de liste (1/9)

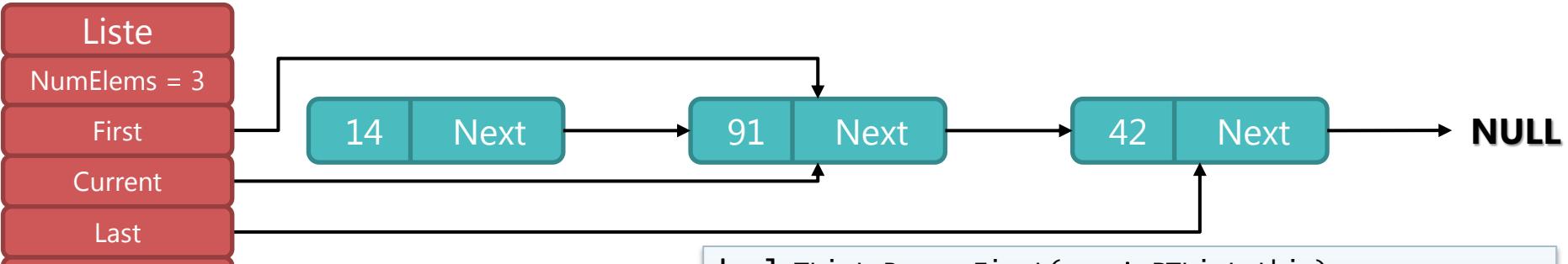
1) Plus d'un élément dans la liste



LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait d'un élément en début de liste (2/9)

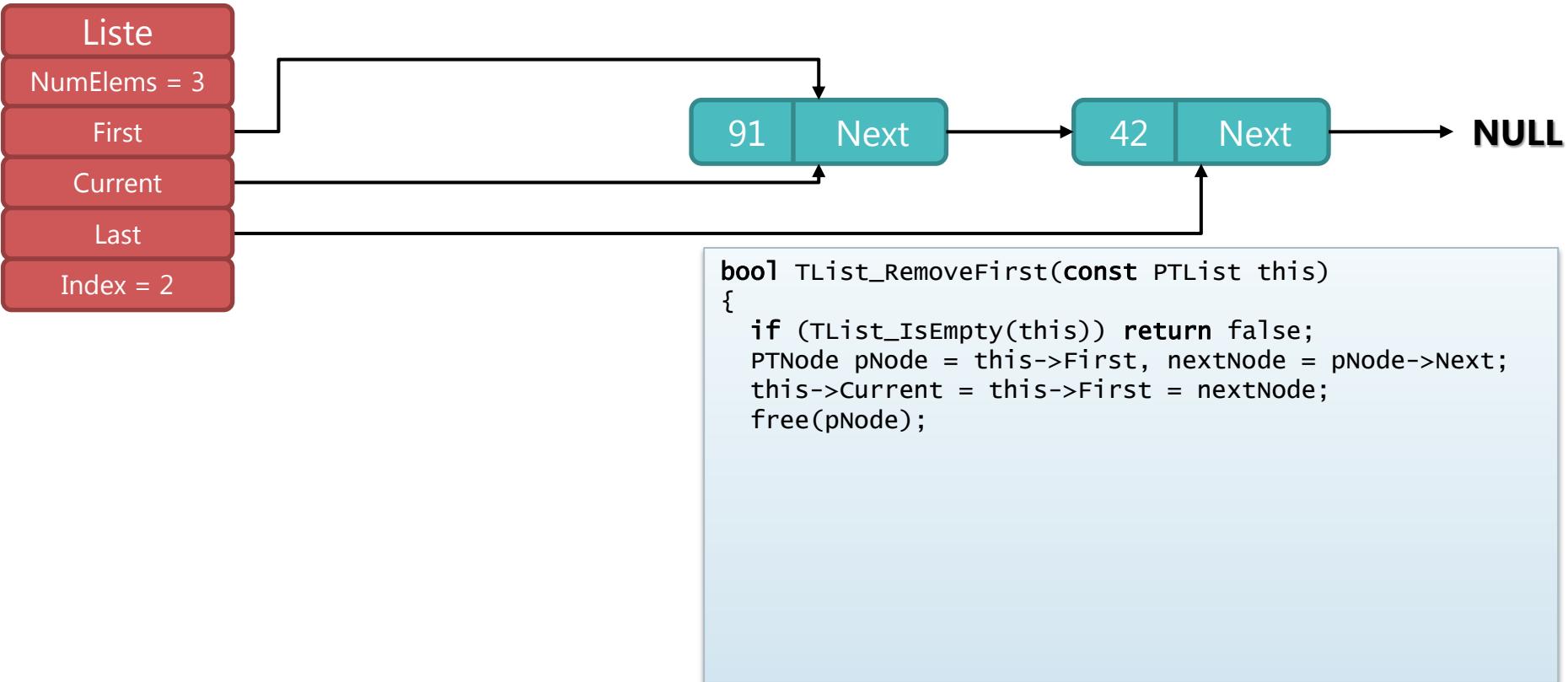
1) Plus d'un élément dans la liste



LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait d'un élément en début de liste (3/9)

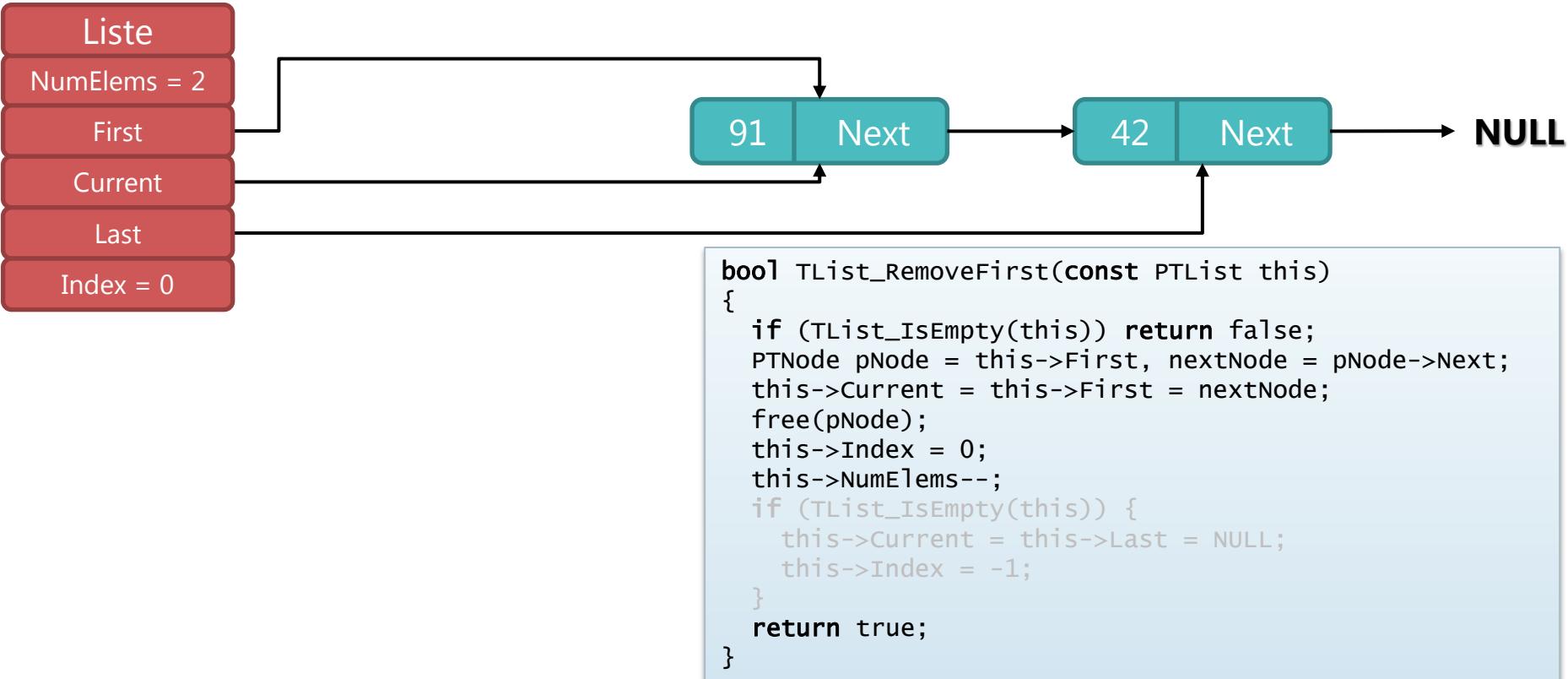
1) Plus d'un élément dans la liste



LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait d'un élément en début de liste (4/9)

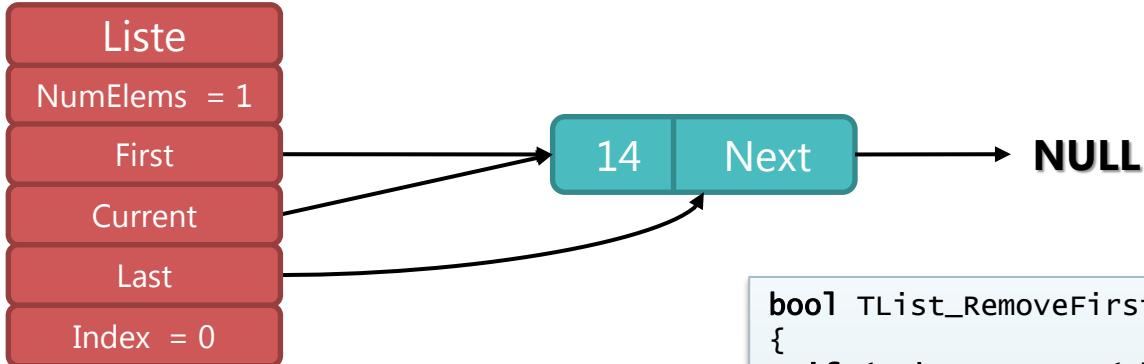
1) Plus d'un élément dans la liste



LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait d'un élément en début de liste (5/9)

2) Un seul élément dans la liste

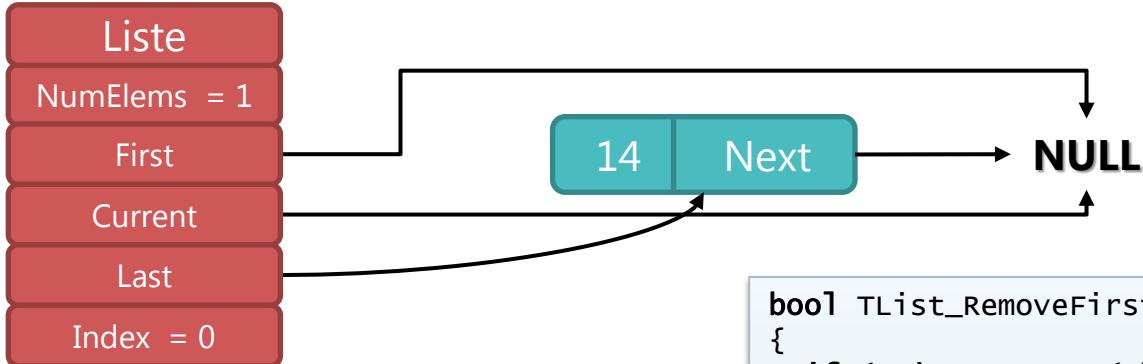


```
bool TList_RemoveFirst(const PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = this->First, nextNode = pNode->Next;
```

LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait d'un élément en début de liste (6/9)

2) Un seul élément dans la liste

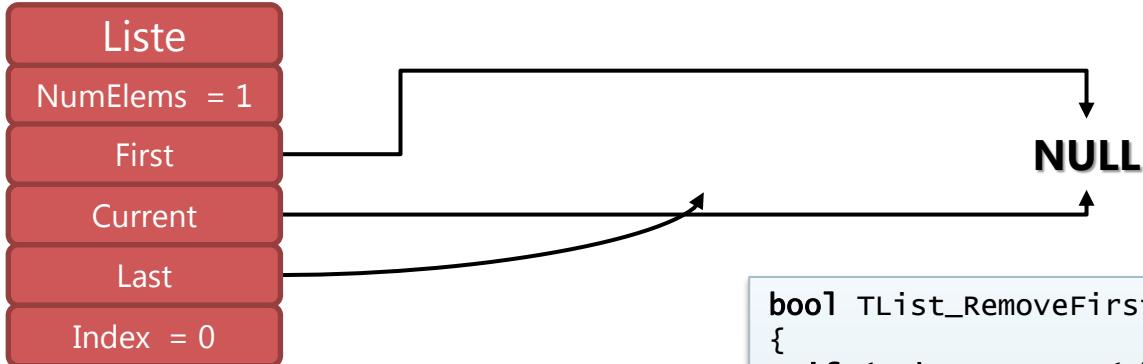


```
bool TList_RemoveFirst(const PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = this->First, nextNode = pNode->Next;
    this->Current = this->First = nextNode;
```

LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait d'un élément en début de liste (7/9)

2) Un seul élément dans la liste

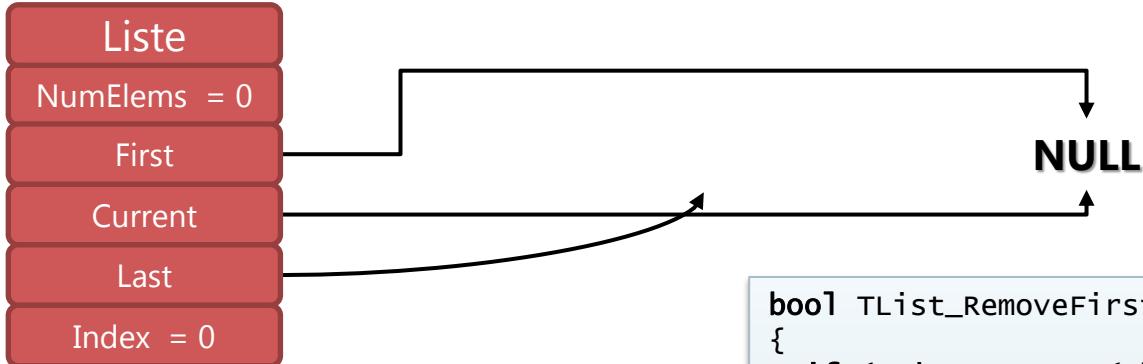


```
bool TList_RemoveFirst(const PTLList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = this->First, nextNode = pNode->Next;
    this->Current = this->First = nextNode;
    free(pNode);
```

LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait d'un élément en début de liste (8/9)

2) Un seul élément dans la liste

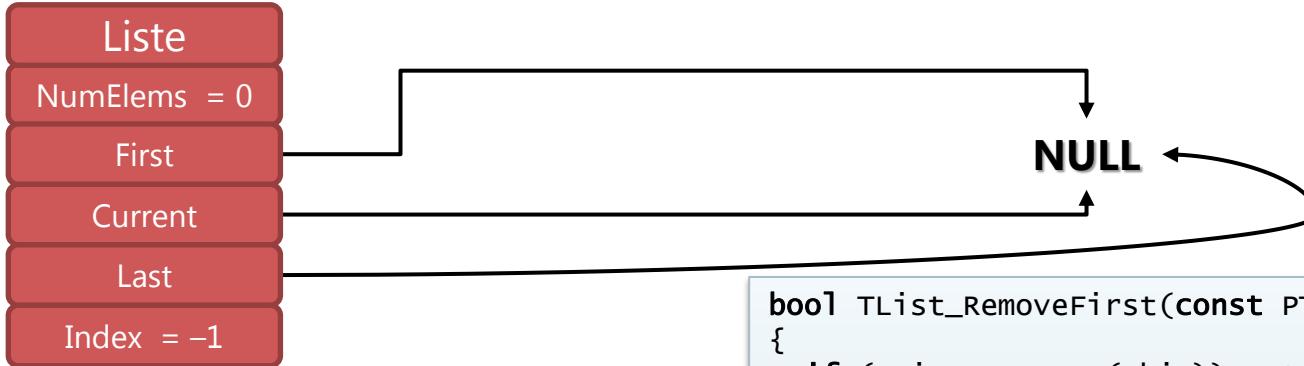


```
bool TList_RemoveFirst(const PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = this->First, nextNode = pNode->Next;
    this->Current = this->First = nextNode;
    free(pNode);
    this->Index = 0;
    this->NumElems--;
}
```

LES LISTES CHAINÉES

Fonction `TList_RemoveFirst` : retrait d'un élément en début de liste (9/9)

2) Un seul élément dans la liste

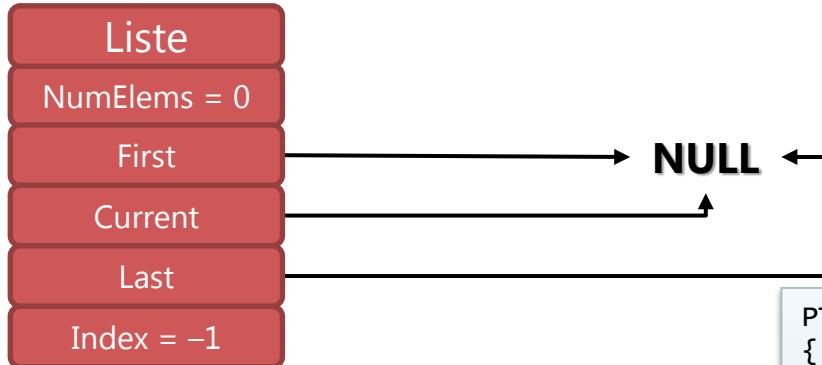


```
bool TList_RemoveFirst(const PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = this->First, nextNode = pNode->Next;
    this->Current = this->First = nextNode;
    free(pNode);
    this->Index = 0;
    this->NumElems--;
    if (TList_IsEmpty(this)) {
        this->Current = this->Last = NULL;
        this->Index = -1;
    }
    return true;
}
```

LES LISTES CHAINÉES

Fonction TList_Add : ajout d'un élément en fin de liste (1/10)

1) Liste vide

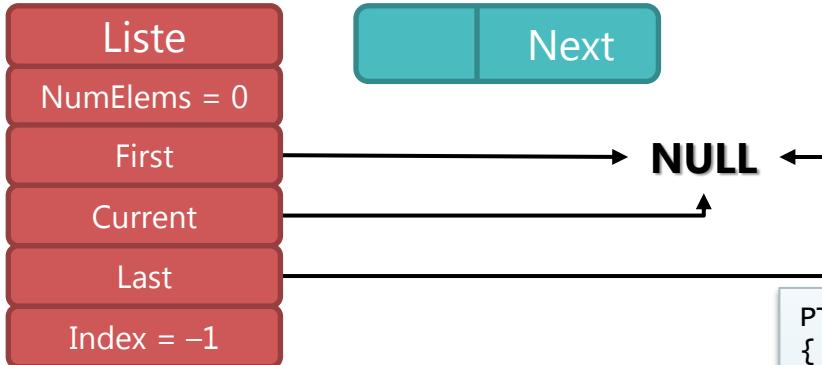


```
PTNode TList_Add(const PTLList this, TElement Element)  
{
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (2/10)

1) Liste vide

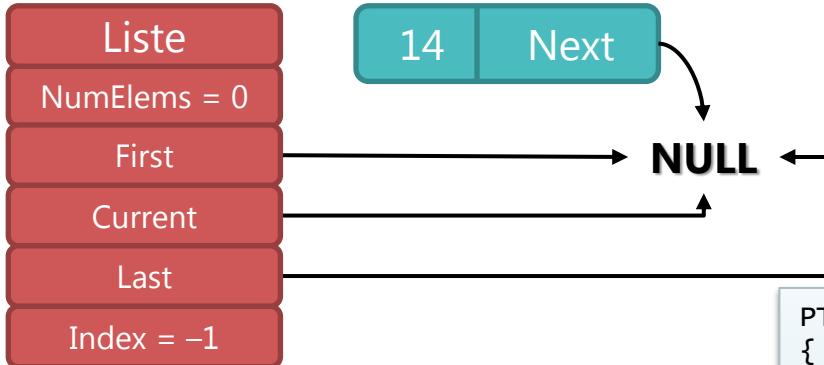


```
PTNode TList_Add(const PTLList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
```

LES LISTES CHAINÉES

Fonction TList_Add : ajout d'un élément en fin de liste (3/10)

1) Liste vide

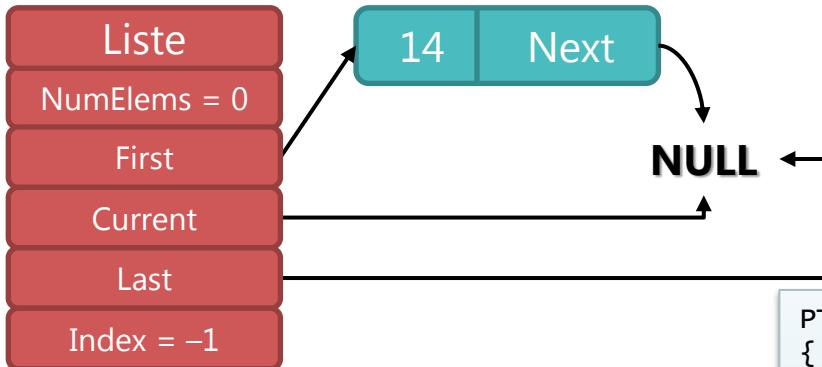


```
PTNode TList_Add(const PTLList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
```

LES LISTES CHAINÉES

Fonction TList_Add : ajout d'un élément en fin de liste (4/10)

1) Liste vide

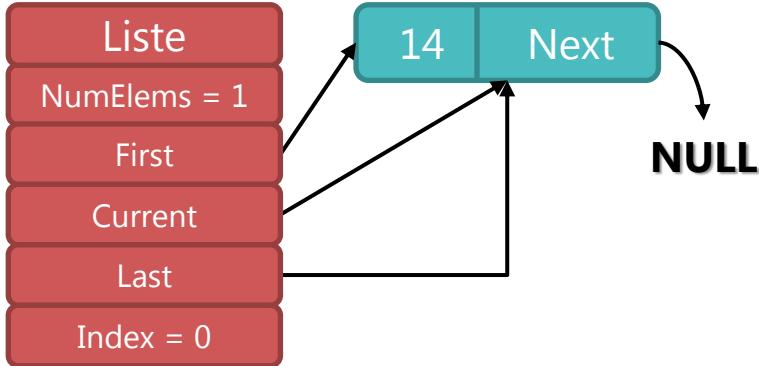


```
PTNode TList_Add(const PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
    if (TList_IsEmpty(this))
        this->First = newNode;
    else
        this->Last->Next = newNode;
```

LES LISTES CHAINÉES

Fonction TList_Add : ajout d'un élément en fin de liste (5/10)

1) Liste vide

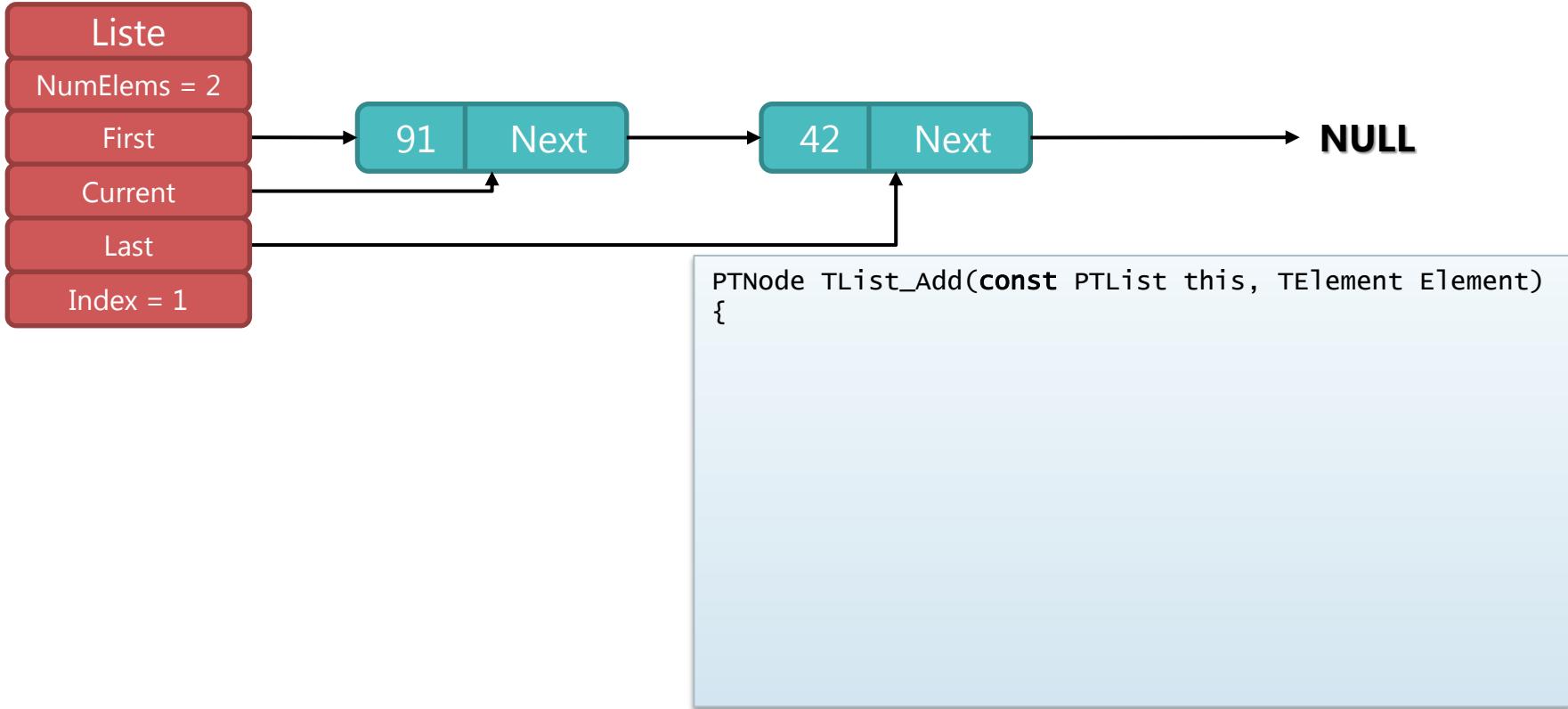


```
PTNode TList_Add(const PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
    if (TList_IsEmpty(this))
        this->First = newNode;
    else
        this->Last->Next = newNode;
    this->Current = this->Last = newNode;
    this->Index = this->NumElems++;
    return newNode;
}
```

LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (6/10)

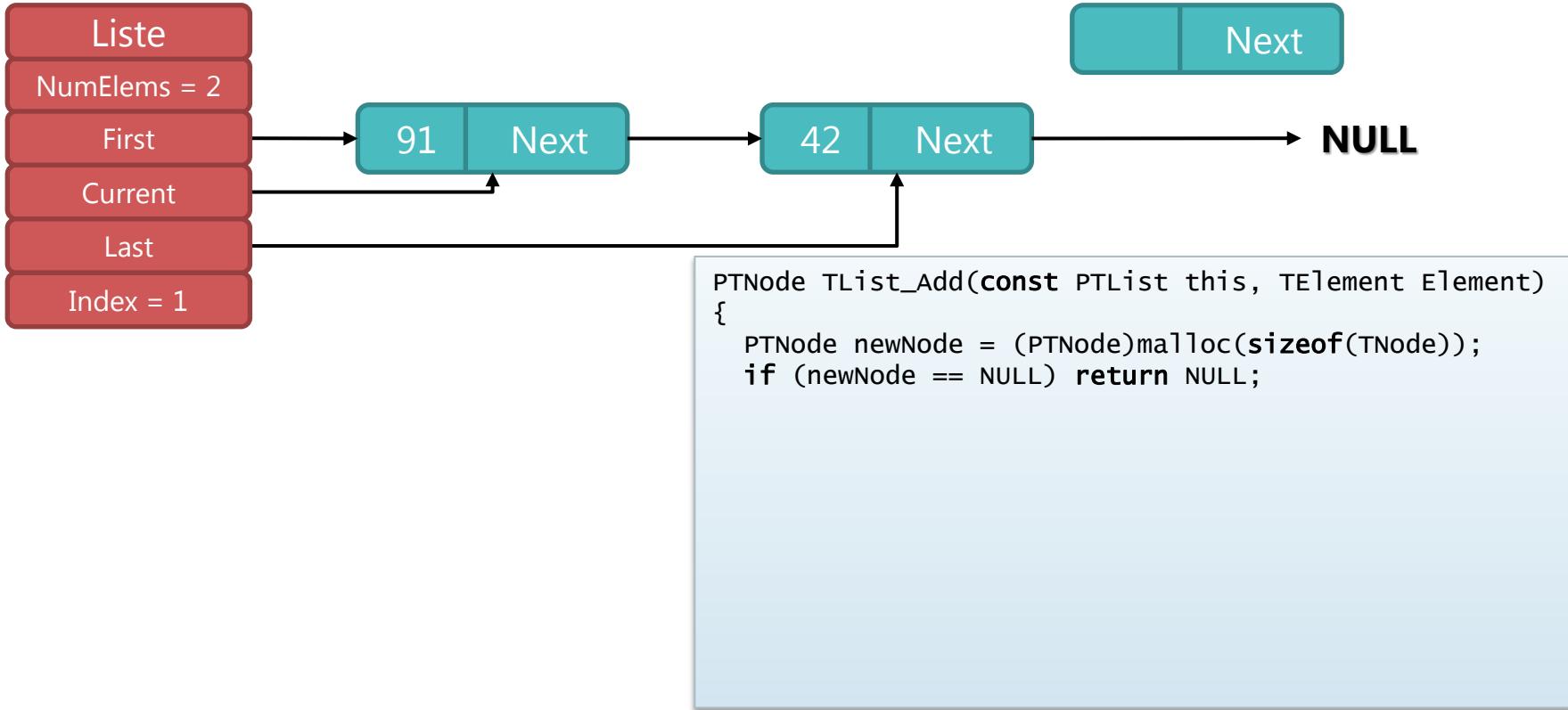
2) Liste non vide



LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (7/10)

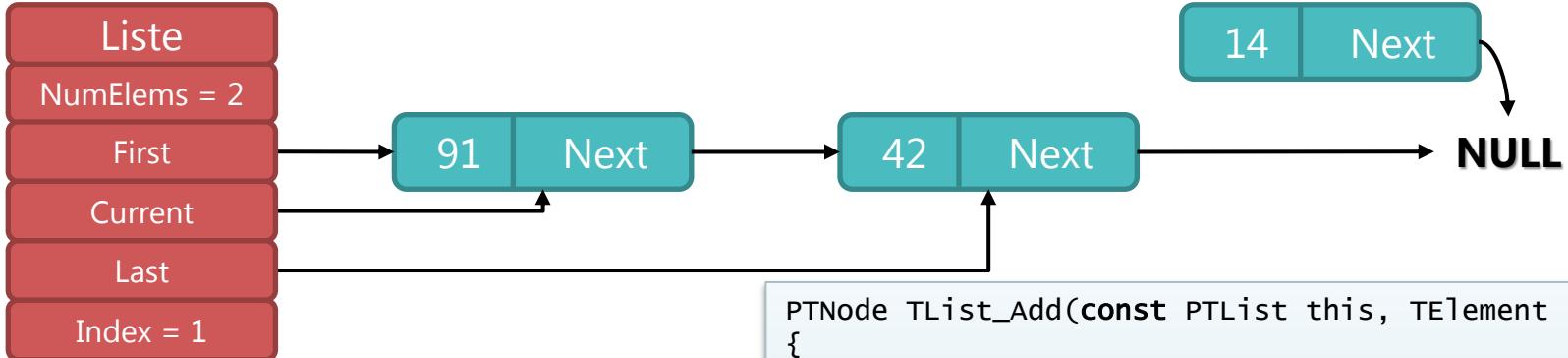
2) Liste non vide



LES LISTES CHAINÉES

Fonction `TList_Add` : ajout d'un élément en fin de liste (8/10)

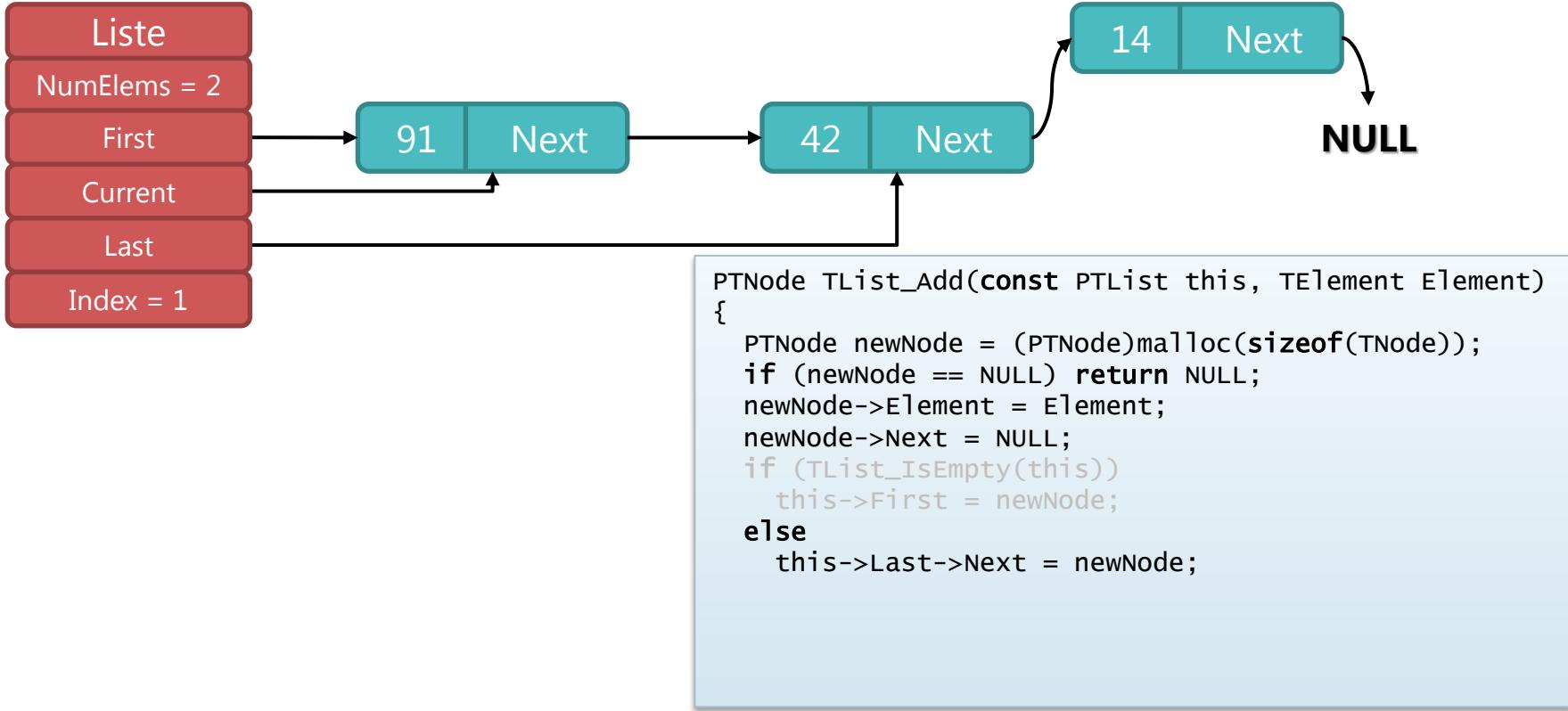
2) Liste non vide



LES LISTES CHAINÉES

Fonction TList_Add : ajout d'un élément en fin de liste (9/10)

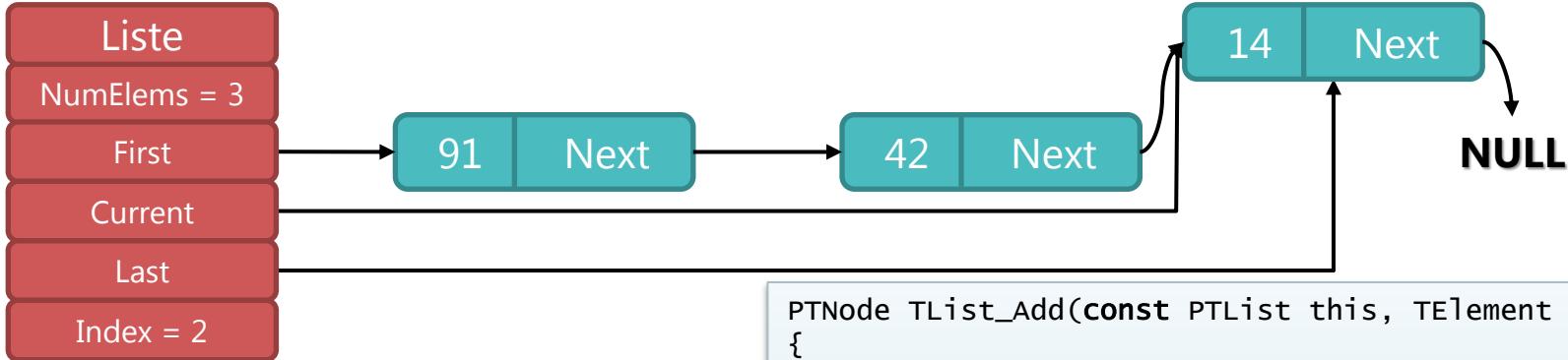
2) Liste non vide



LES LISTES CHAINÉES

Fonction TList_Add : ajout d'un élément en fin de liste (10/10)

2) Liste non vide

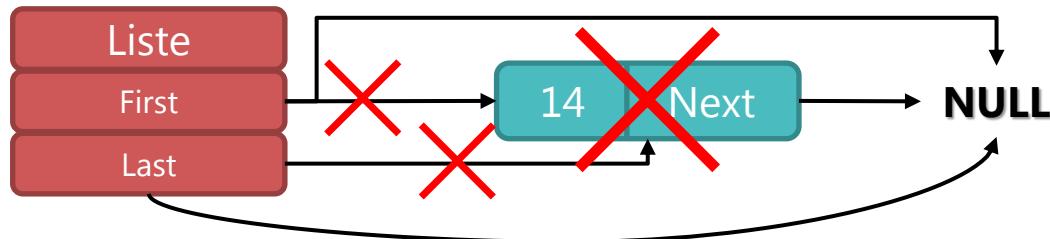


```
PTNode TList_Add(const PTList this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Next = NULL;
    if (TList_IsEmpty(this))
        this->First = newNode;
    else
        this->Last->Next = newNode;
    this->Current = this->Last = newNode;
    this->Index = this->NumElems++;
    return newNode;
}
```

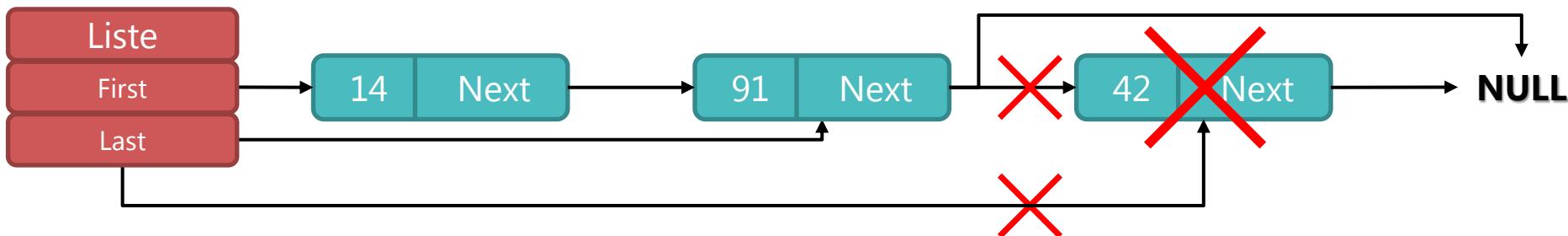
LES LISTES CHAINÉES

Fonction Tlist_RemoveLast : suppression d'un élément en fin de liste

- 1) Un seul élément dans la liste (idem suppression en début de liste)



- 2) Plus d'un élément dans la liste



LES LISTES CHAINÉES

Fonction Tlist_RemoveLast : code C

```

bool TList_RemoveLast(const PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode prevNode = this->First, pNode = prevNode->Next;
    if (pNode == NULL) {           /* 1 seul élément dans la liste */
        free(prevNode);
        this->First = this->Current = this->Last = NULL;
        this->Index = -1;
    }
    else {
        while (pNode->Next) {
            prevNode = pNode;
            pNode = pNode->Next;
        }
        free(pNode);
        prevNode->Next = NULL;
        this->Last = prevNode;
    }
    this->NumElems--;
    this->Index = this->NumElems - 1;
    return true;
}
  
```

LES LISTES CHAINÉES

Fonction TList_Delete : suppression complète de la liste

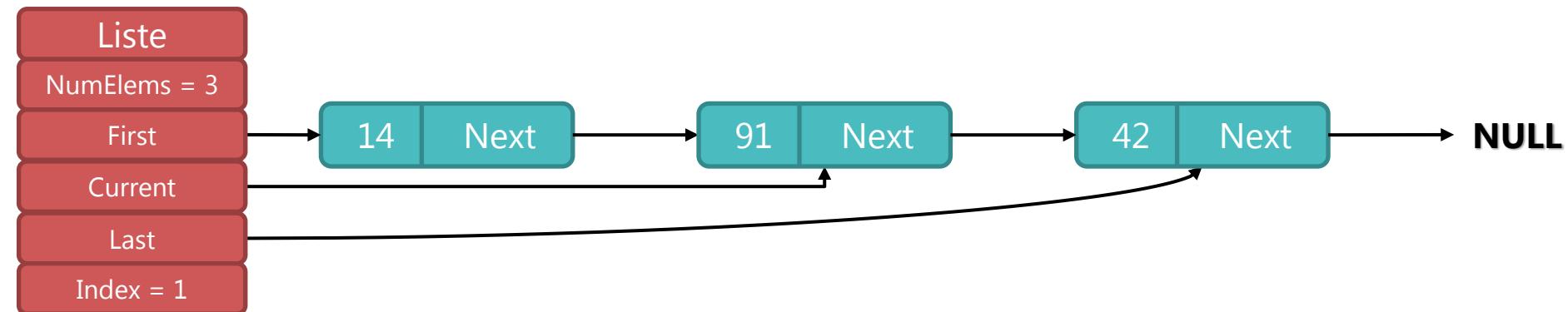
- Comme dans l'implémentation précédente, la suppression complète de la liste s'opère par la suppression successive des éléments de tête (moins coûteuse en temps) jusqu'au dernier élément.
- Il faut ensuite libérer la mémoire allouée à la structure liste.

```
bool TList_Delete(const PTLList this)
{
    while (TList_RemoveFirst(this));
    free(this);
}
```

LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant l'élément courant (1/5)

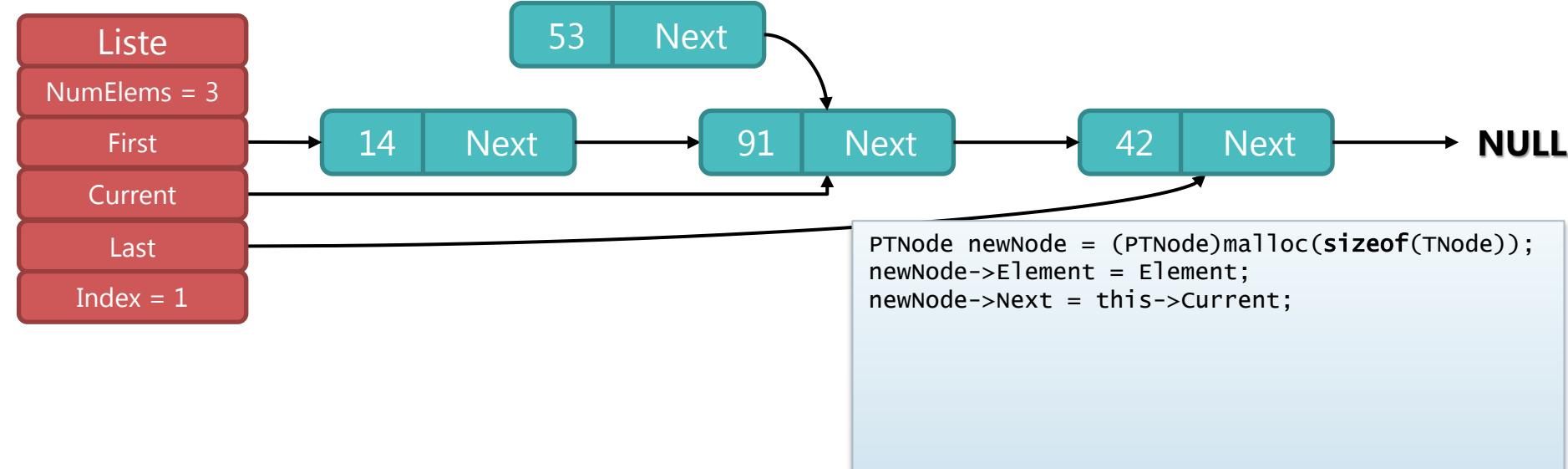
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst*,



LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant l'élément courant (2/5)

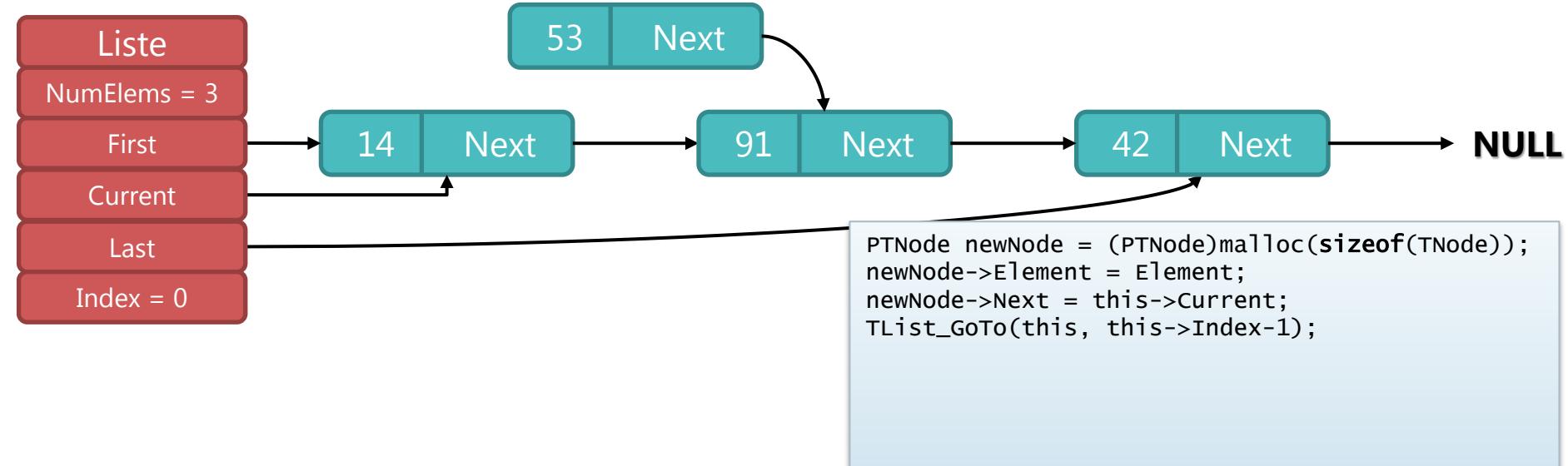
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst*,
- 2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant,



LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant l'élément courant (3/5)

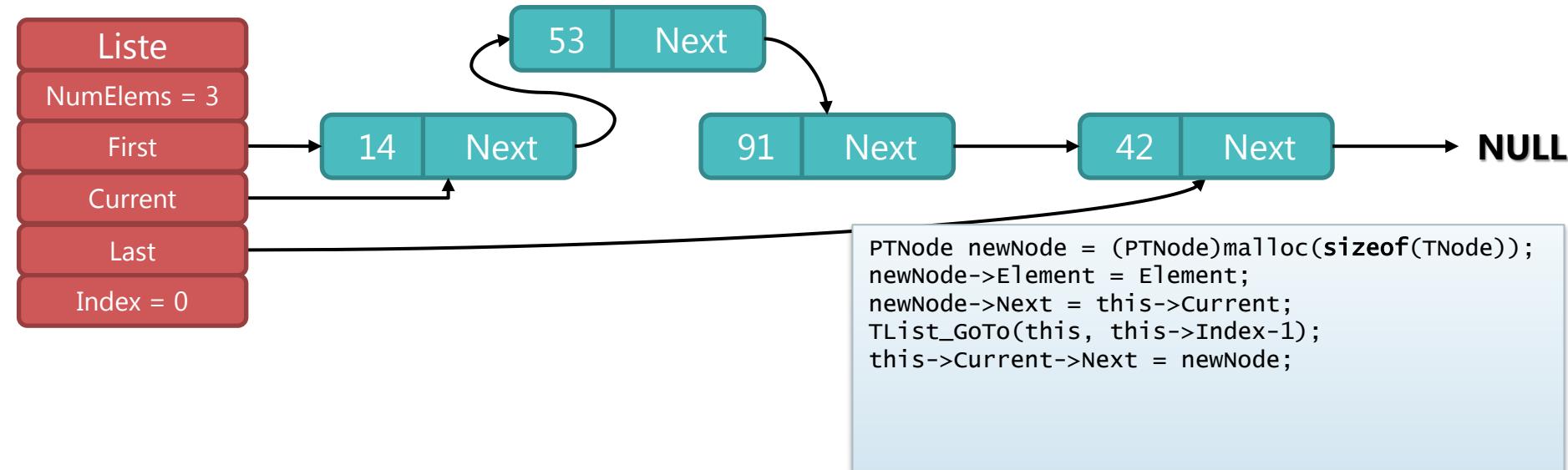
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst*,
- 2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant,
- 3) puis aller au nœud précédent (fonction *GoTo*),



LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant l'élément courant (4/5)

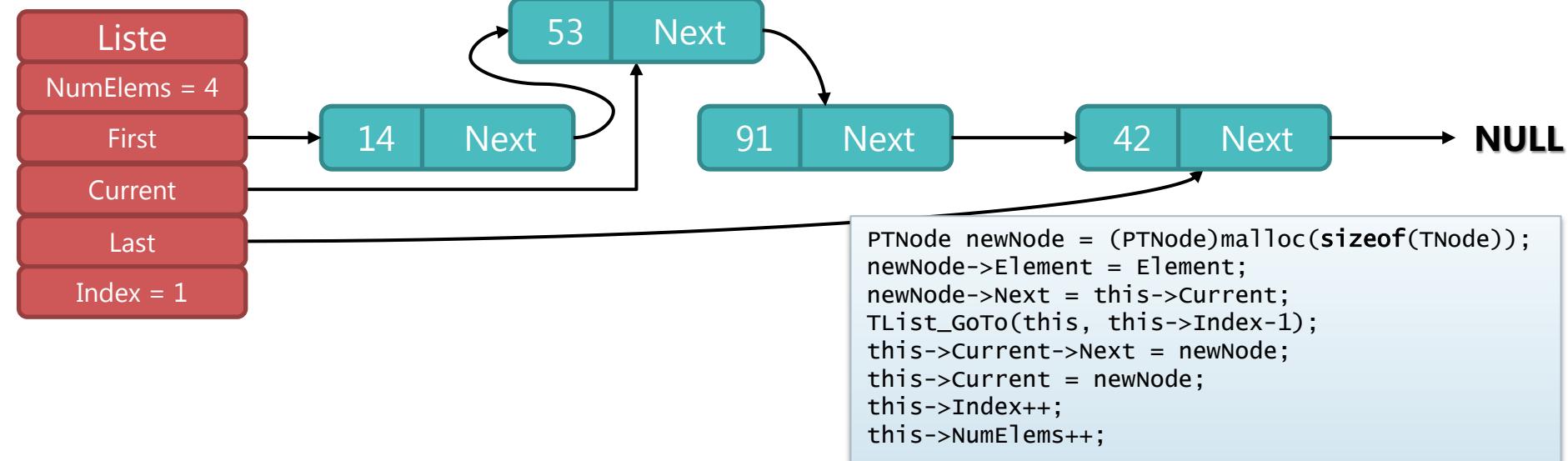
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst*,
- 2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant,
- 3) puis aller au nœud précédent (fonction *GoTo*),
- 4) pour le faire pointer vers le nouveau nœud,



LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant l'élément courant (5/5)

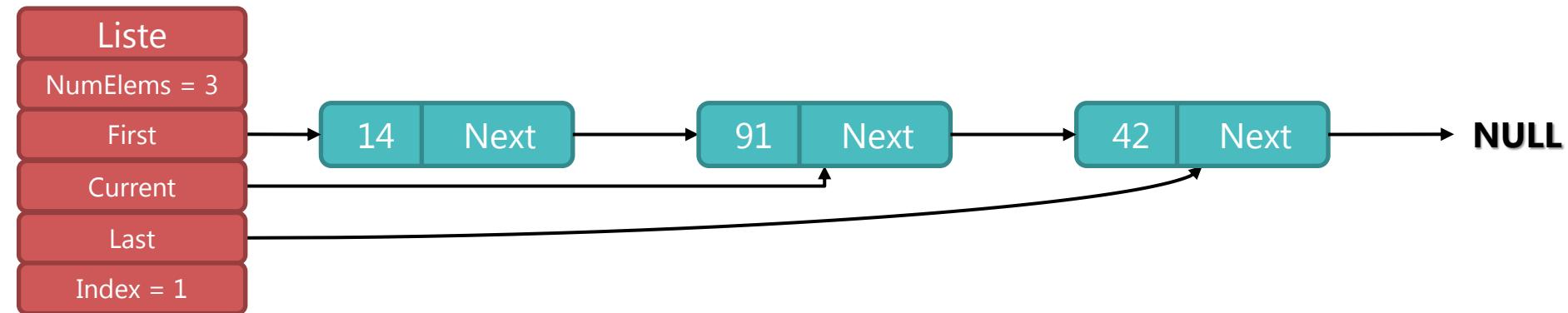
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst*,
- 2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant,
- 3) puis aller au nœud précédent (fonction *GoTo*),
- 4) pour le faire pointer vers le nouveau nœud,
- 5) qui devient alors le nœud courant (index initial inchangé).



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait de l'élément courant (1/5)

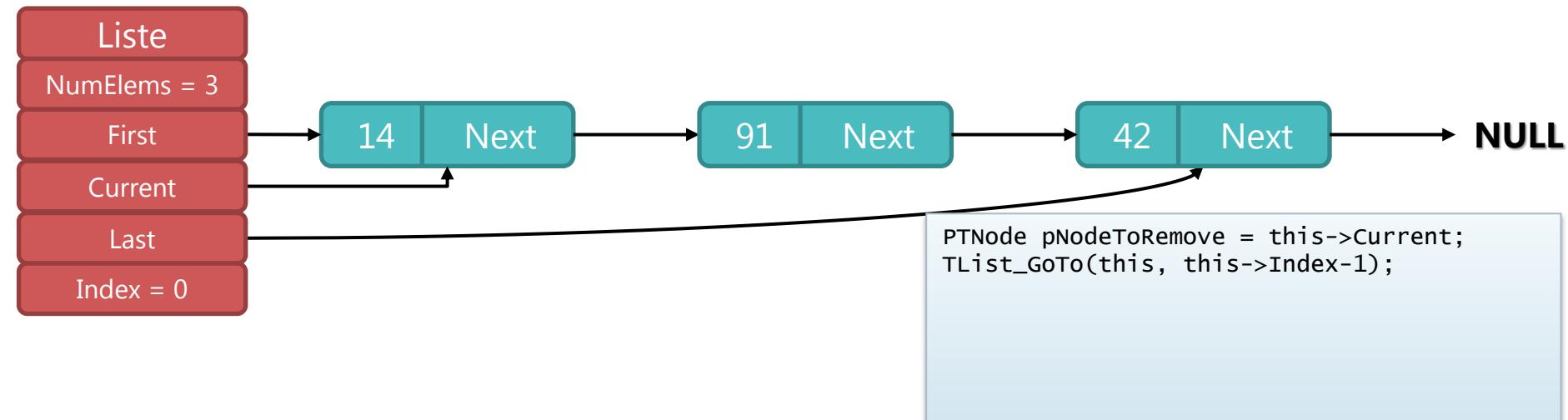
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst*,
- 2) Sinon si le nœud courant est le dernier de la liste, appeler la fonction *RemoveLast*,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait de l'élément courant (2/5)

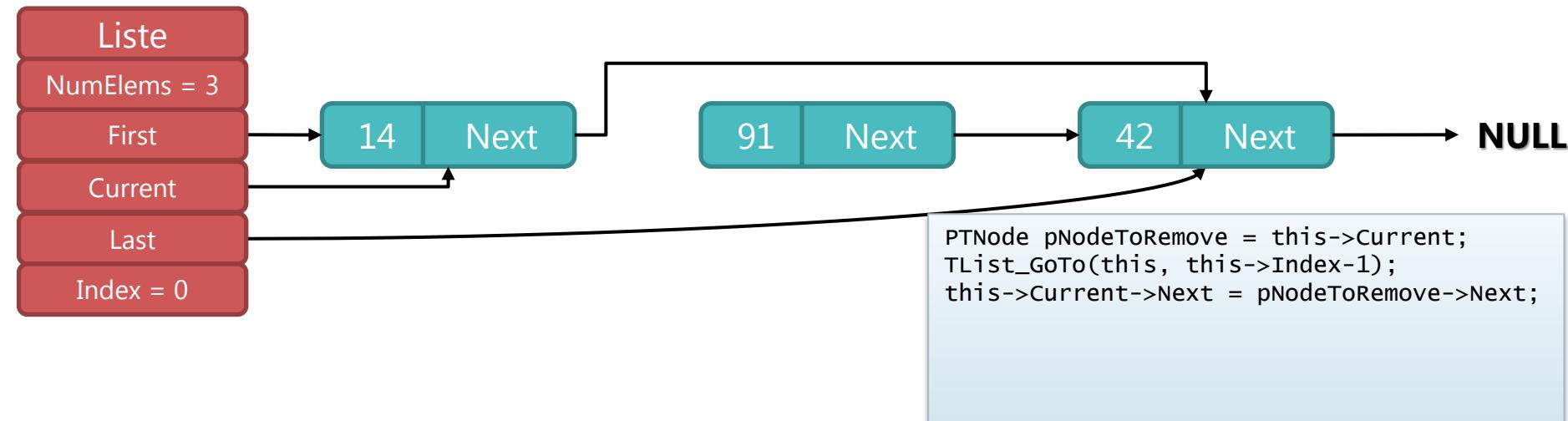
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst*,
- 2) Sinon si le nœud courant est le dernier de la liste, appeler la fonction *RemoveLast*,
- 3) Sinon aller au nœud précédent le nœud à supprimer,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait de l'élément courant (3/5)

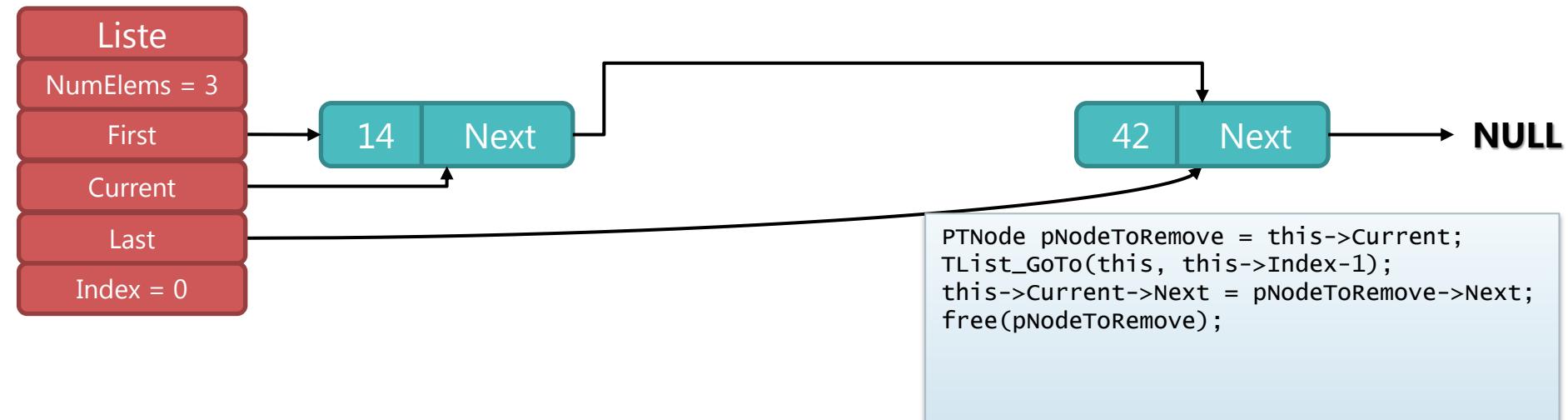
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst*,
- 2) Sinon si le nœud courant est le dernier de la liste, appeler la fonction *RemoveLast*,
- 3) Sinon aller au nœud précédent le nœud à supprimer,
- 4) pour le faire pointer vers son suivant,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait de l'élément courant (4/5)

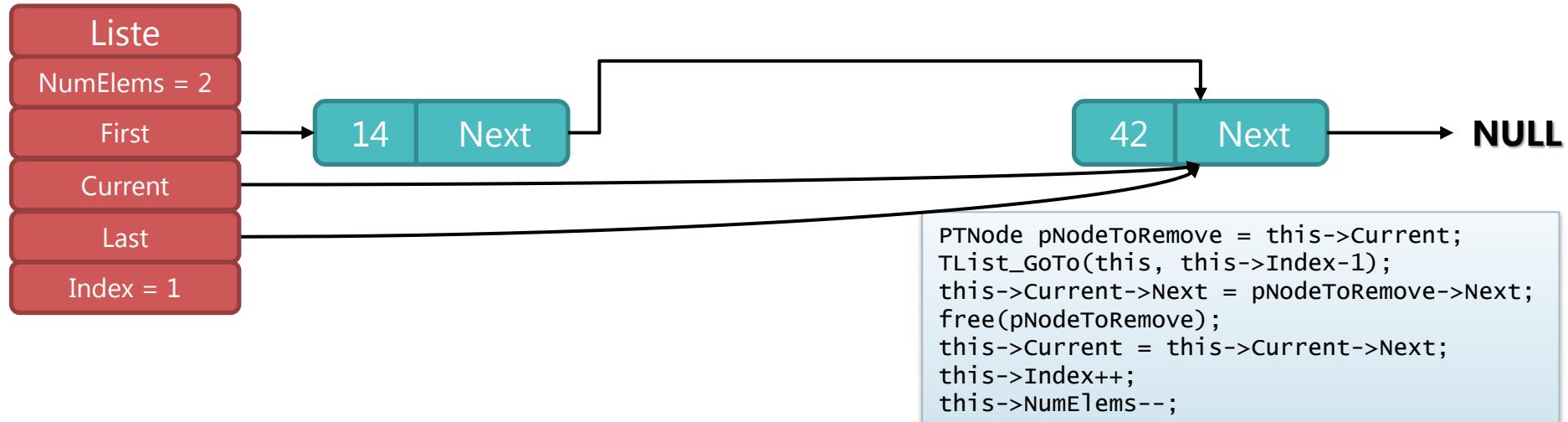
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst*,
- 2) Sinon si le nœud courant est le dernier de la liste, appeler la fonction *RemoveLast*,
- 3) Sinon aller au nœud précédent le nœud à supprimer,
- 4) pour le faire pointer vers son suivant,
- 5) et supprimer le nœud,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait de l'élément courant (5/5)

- 1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst*,
- 2) Sinon si le nœud courant est le dernier de la liste, appeler la fonction *RemoveLast*,
- 3) Sinon aller au nœud précédent le nœud à supprimer,
- 4) pour le faire pointer vers son suivant,
- 5) et supprimer le nœud,
- 6) le suivant devenant alors le nœud courant (indice initial inchangé).



LES LISTES CHAINÉES

Implémentation orientée objet d'une liste : "classe" TList (dans *list.h*)

```

typedef struct List {
    int NumElems;
    PTNode First;
    PTNode Last;
    PTNode Current;
    int Index;
    bool(*IsEmpty)(const struct List * );
    int(*Length)(const struct List * );
    int(*GetIndex)(const struct List * );
    PTNode(*GoTo)(const struct List *, int);
    PTNode(*InsertFirst)(const struct List *, TElement);
    bool(*RemoveFirst)(const struct List * );
    PTNode(*Add)(const struct List *, TElement);
    bool(*RemoveLast)(const struct List * );
    PTNode(*Insert)(const struct List *, TElement);
    bool(*RemoveCurrent)(const struct List * );
    void(*Display)(const struct List * );
    void(*Delete)(const struct List *); /* Destructeur */
} TList;

typedef TList *PTList;

PTList TList_New(void); /* Constructeur */

```

LES LISTES CHAINÉES

Constructeur (dans *list.c*) : création et initialisation d'une liste vide

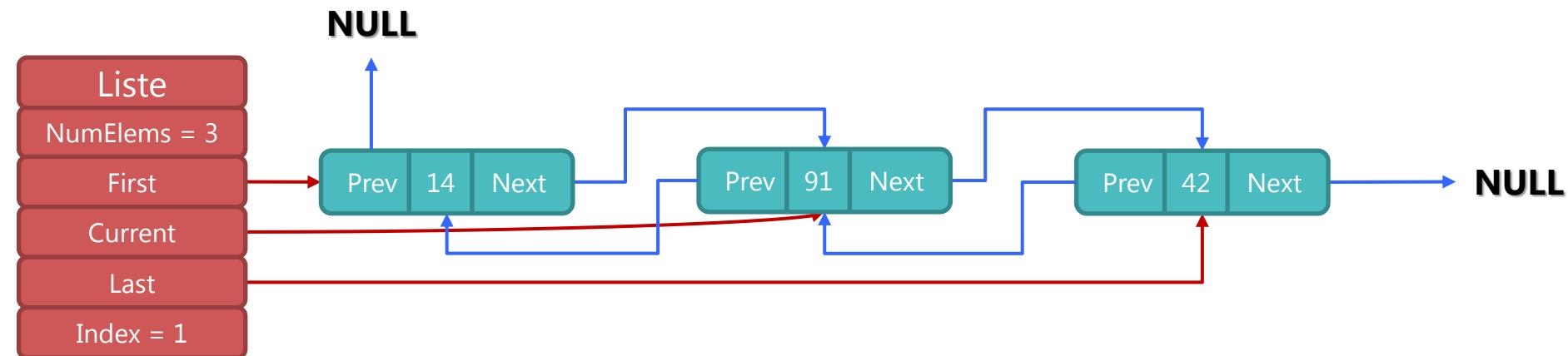
```
PTList TList_New(void)
{
    PTList this = malloc(sizeof(TList));
    this->NumElems = 0;
    this->First = NULL;
    this->Last = NULL;
    this->Current = NULL;
    this->Index = -1;
    this->IsEmpty = (void *)TList_IsEmpty;
    this->Length = (void *)TList_Length;
    this->GetIndex = (void *)TList_Index;
    this->GoTo = (void *)TList_GoTo;
    this->InsertFirst = (void *)TList_InsertFirst;
    this->RemoveFirst = (void *)TList_RemoveFirst;
    this->Add = (void *)TList_Add;
    this->RemoveLast = (void *)TList_RemoveLast;
    this->Insert = (void *)TList_Insert;
    this->RemoveCurrent = (void *)TList_RemoveCurrent;
    this->Display = (void *)TList_Display;
    this->Delete = (void *)TList_Delete;
    return this;
}
```

LES LISTES CHAINÉES

Liste doublement chainée

- Permet le parcours de liste dans les deux sens.
- Chaque nœud connaît son suivant et son précédent : ajouts et suppressions plus simples à implémenter.

```
typedef struct Node {
    TElement Element;
    struct Node *Prev;
    struct Node *Next;
} TNode;
```



LES LISTES CHAINÉES

Parcours d'une liste double

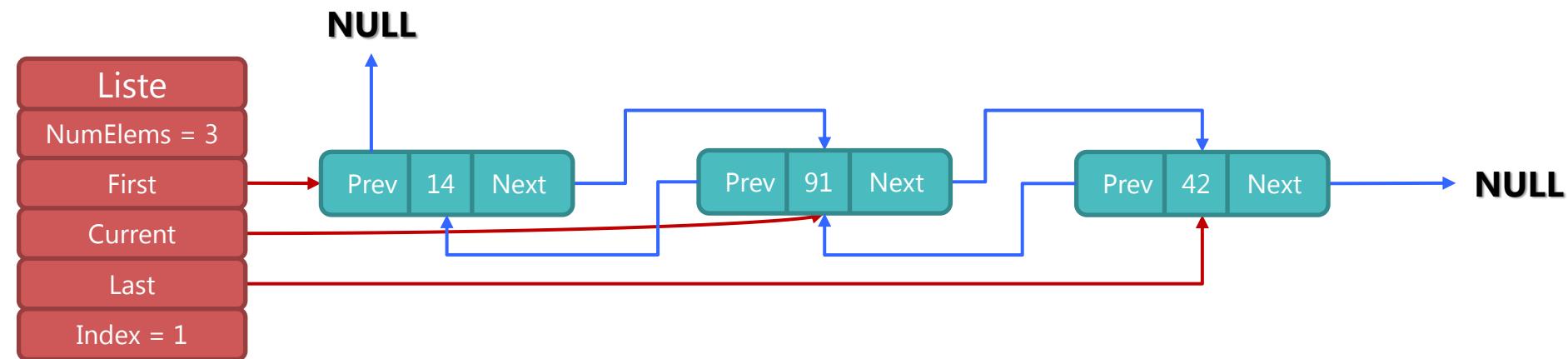
- Le temps moyen de recherche d'un élément dans une liste chaînée simple de n éléments est proportionnel à $n/2$.
- Dans une liste double on peut positionner le pointeur initial au début ou à la fin de la liste selon la position demandée.
- Le temps moyen d'accès à un élément donné est divisé par 2 par rapport à la liste chaînée simple.

```
PTNode TList_GoTo(const PTList this, int Pos)
{
    /* Position inatteignable ! */
    if (this->NumElems <= Pos) return NULL;
    /* On part du début ou de la fin de
     * la liste selon la position désirée */
    if (Pos < this->NumElems/2) {
        this->Current = this->First;
        this->Index = 0;
        while (this->Index < Pos) {
            this->Current = this->Current->Next;
            this->Index++;
        }
    }
    else {
        this->Current = this->Last;
        this->Index = this->NumElems - 1;
        while (this->Index > Pos) {
            this->Current = this->Current->Prev;
            this->Index--;
        }
    }
    return this->Current;
}
```

LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant le nœud courant (1/4)

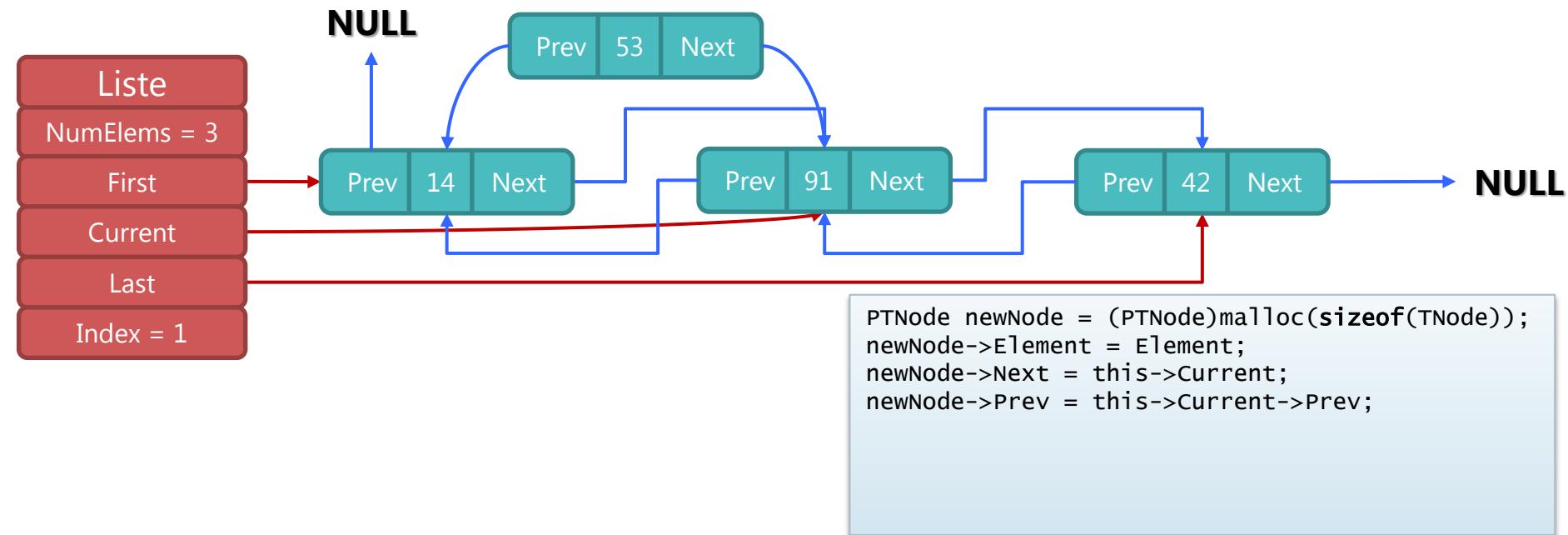
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst*,



LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant le nœud courant (2/4)

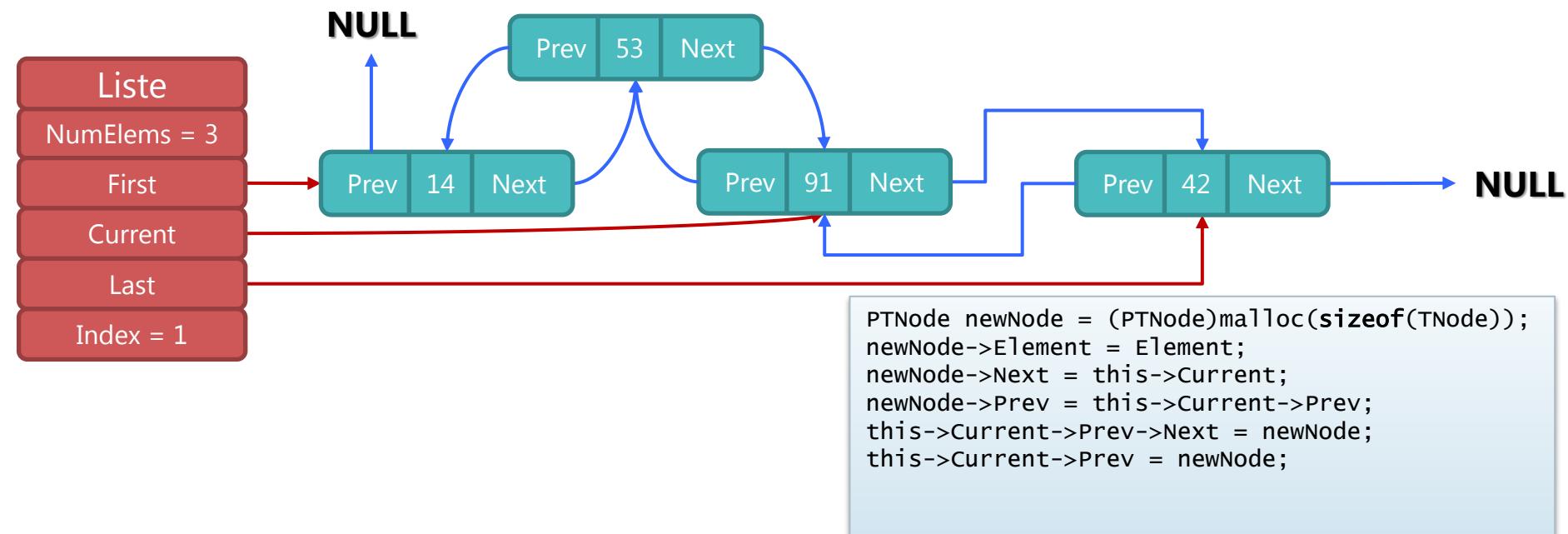
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst*,
- 2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant, et son précédent,



LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant le nœud courant (3/4)

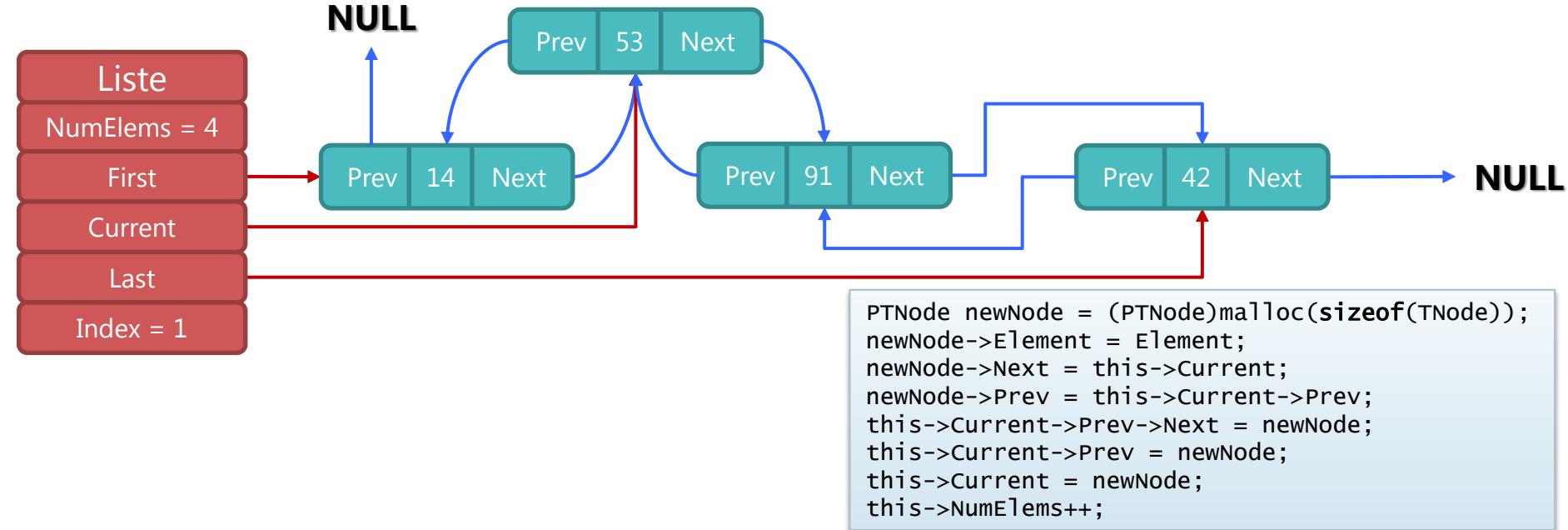
- 1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst*,
- 2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant, et son précédent,
- 3) Les nœuds précédent et courant pointent vers le nouveau nœud,



LES LISTES CHAINÉES

Fonction Tlist_Insert : insertion avant le nœud courant (4/4)

- 1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst*,
- 2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant, et son précédent,
- 3) Les nœuds précédent et courant pointent vers le nouveau nœud,
- 4) Le nouveau nœud devient le nœud courant.



LES LISTES CHAINÉES

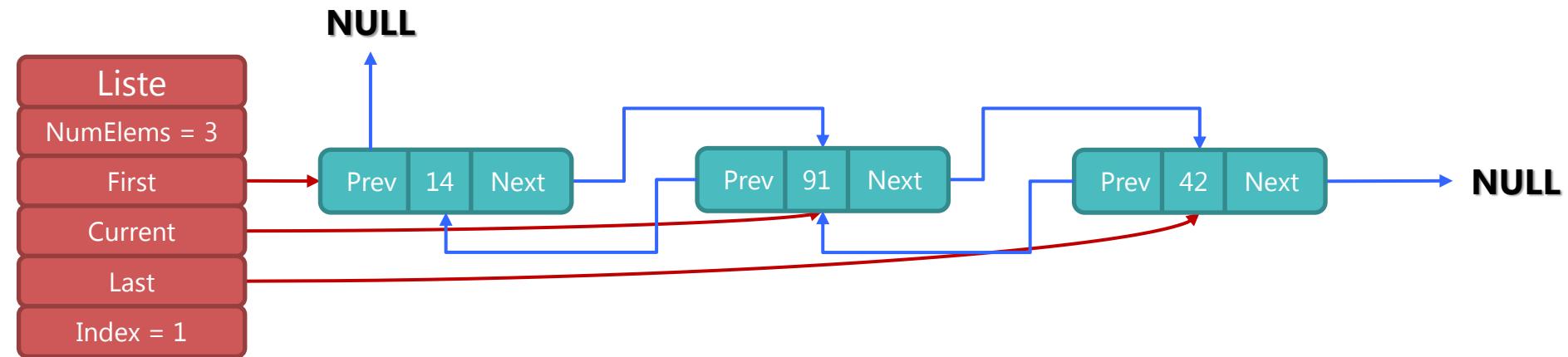
Fonction Tlist_Insert : code C

```
PTNode TList_Insert(const PTList this, TElement Element)
{
    if (this->Current == NULL) return NULL;
    if (this->Current == this->First)
        this->Current = TList_InsertFirst(this, Element);
    else {
        PTNode newNode = (PTNode)malloc(sizeof(TNode));
        if (newNode == NULL) return NULL;
        newNode->Element = Element;
        newNode->Next = this->Current;
        newNode->Prev = this->Current->Prev;
        this->Current->Prev->Next = newNode;
        this->Current->Prev = newNode;
        this->Current = newNode;
        this->NumElems++;
    }
    return this->Current;
}
```

LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (1/9)

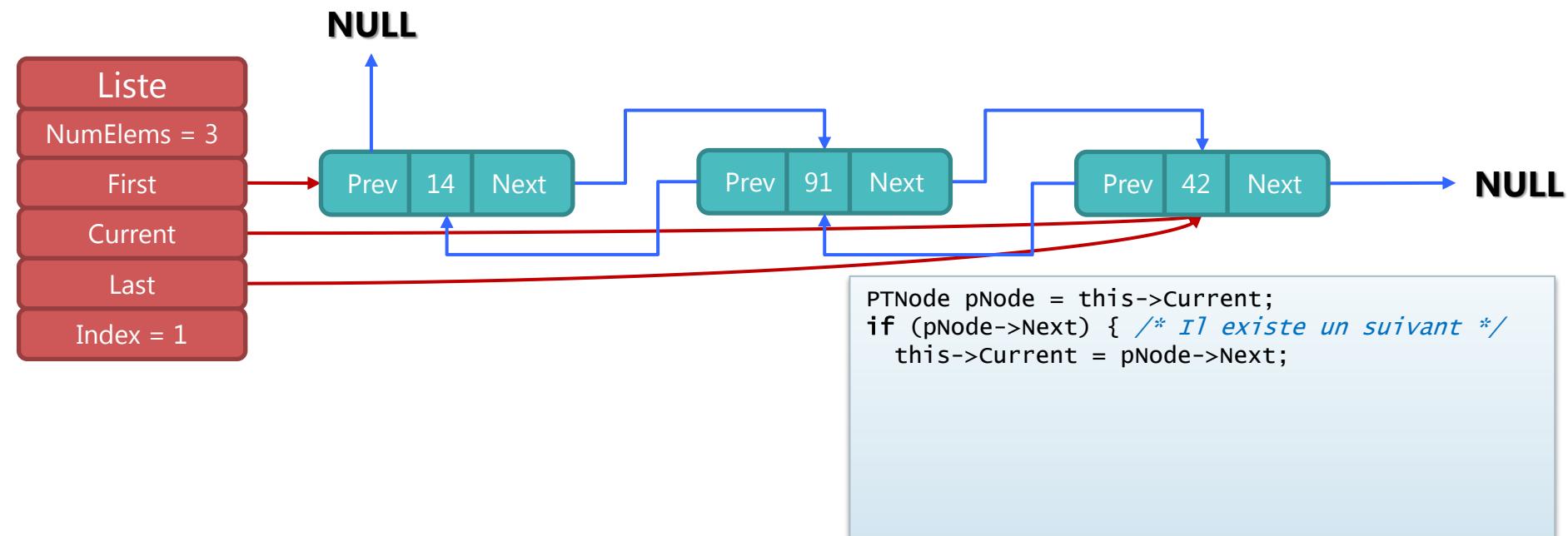
1) S'il existe un suivant,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (2/9)

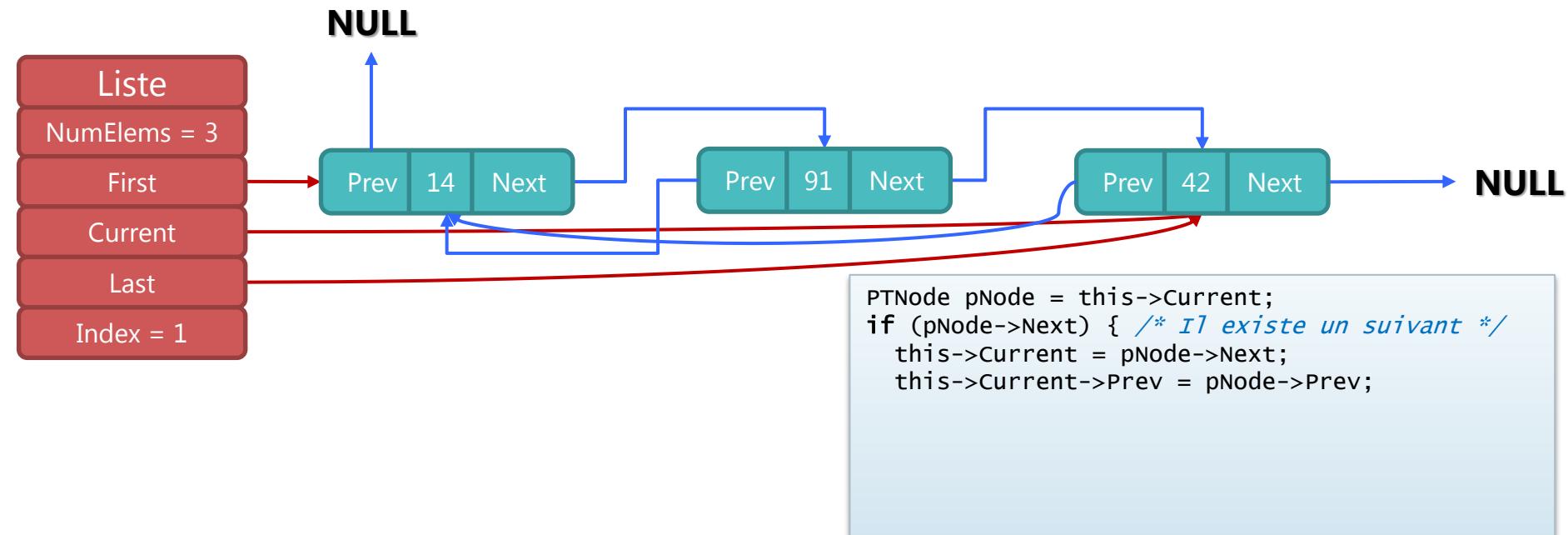
- 1) S'il existe un suivant, il devient le nouveau nœud courant,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (3/9)

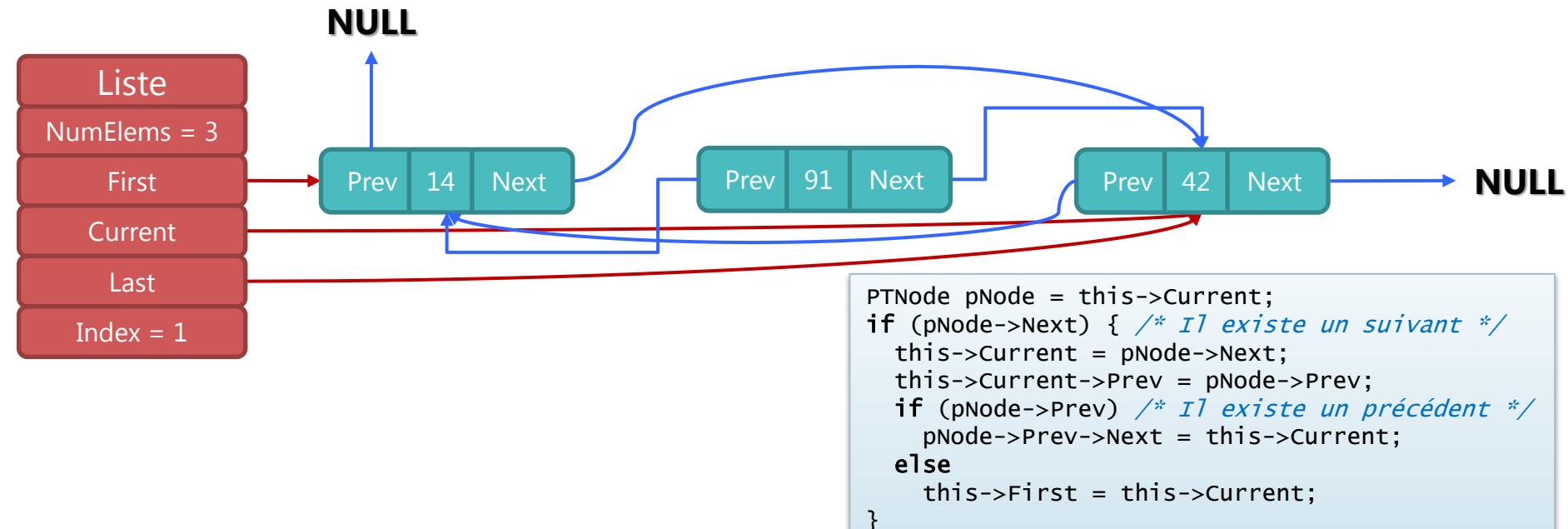
- 1) S'il existe un suivant, il devient le nouveau nœud courant,
- 2) Et pointe vers le nœud précédent,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (4/9)

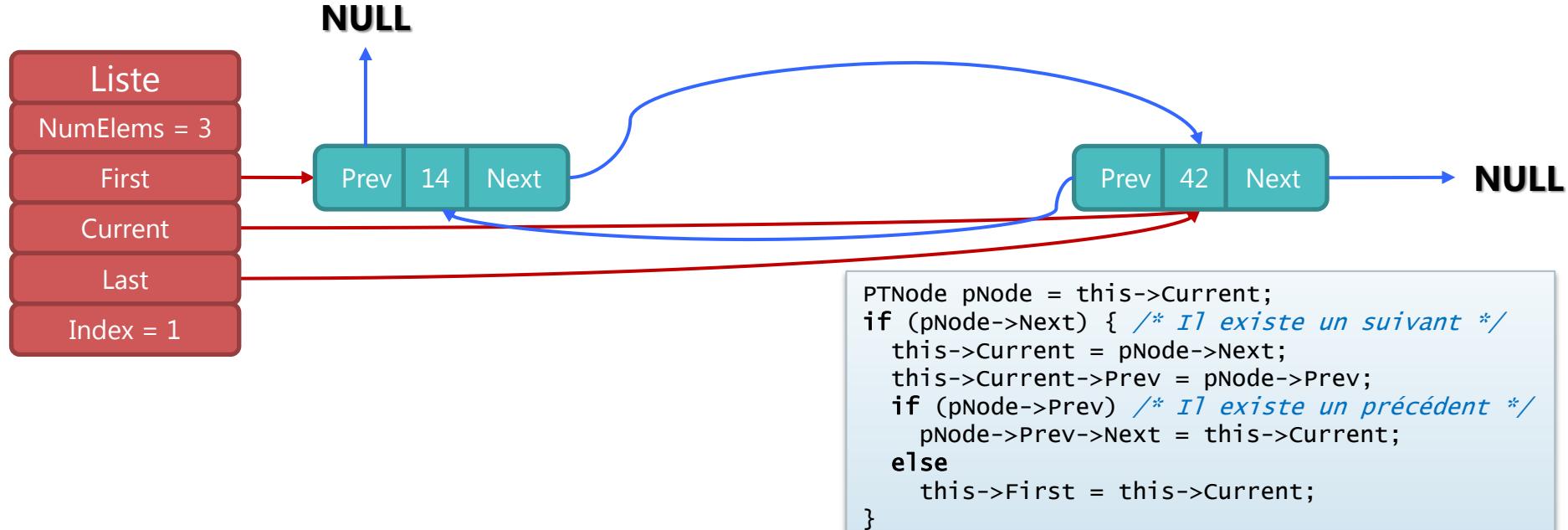
- 1) S'il existe un suivant, il devient le nouveau nœud courant,
- 2) Et pointe vers le nœud précédent,
- 3) qui pointe (si non NULL) à son tour vers le nouveau nœud courant,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (5/9)

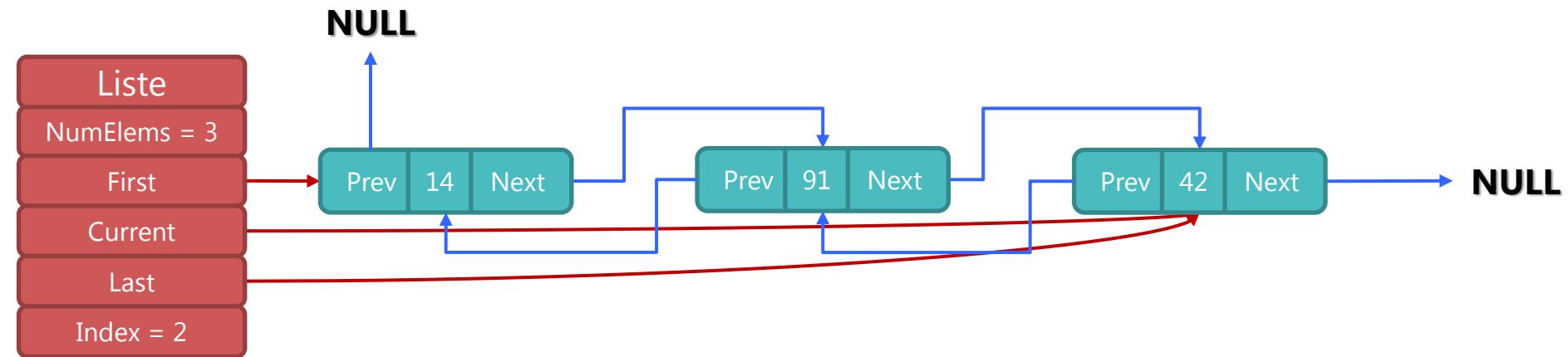
- 1) S'il existe un suivant, il devient le nouveau nœud courant,
- 2) Et pointe vers le nœud précédent,
- 3) qui pointe (si non NULL) à son tour vers le nouveau nœud courant,
- 4) Et on peut maintenant supprimer l'ancien nœud courant.



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (6/9)

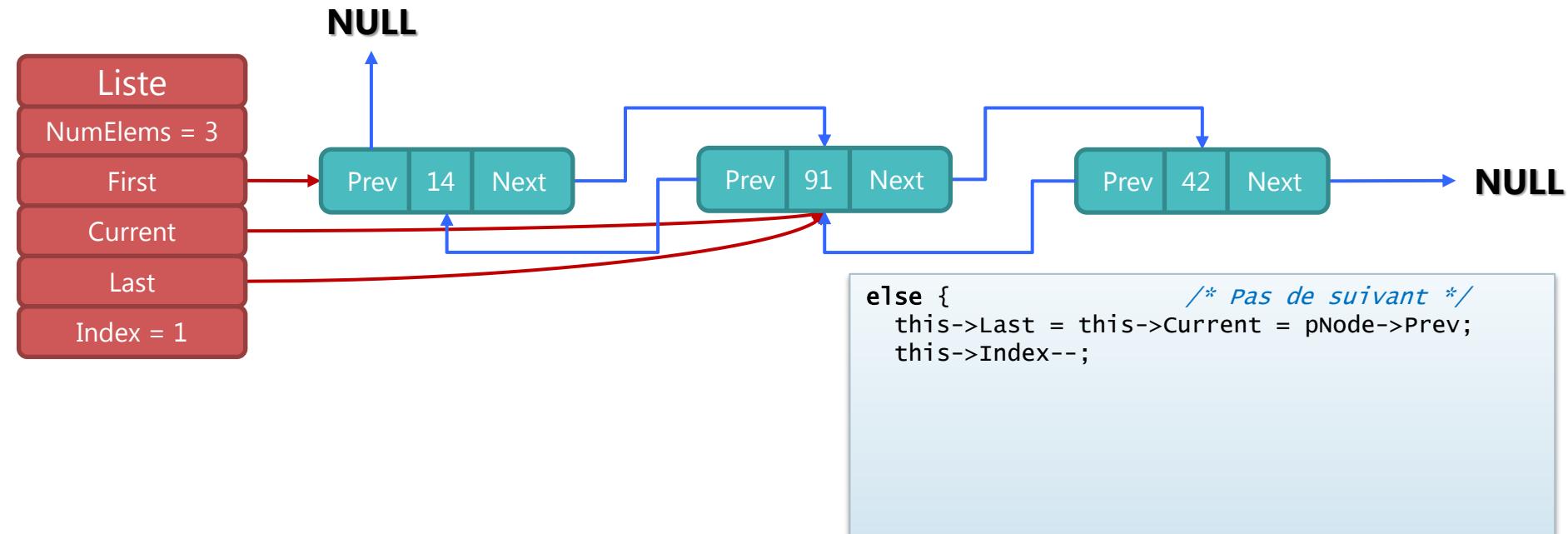
5) Si pas de suivant,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (7/9)

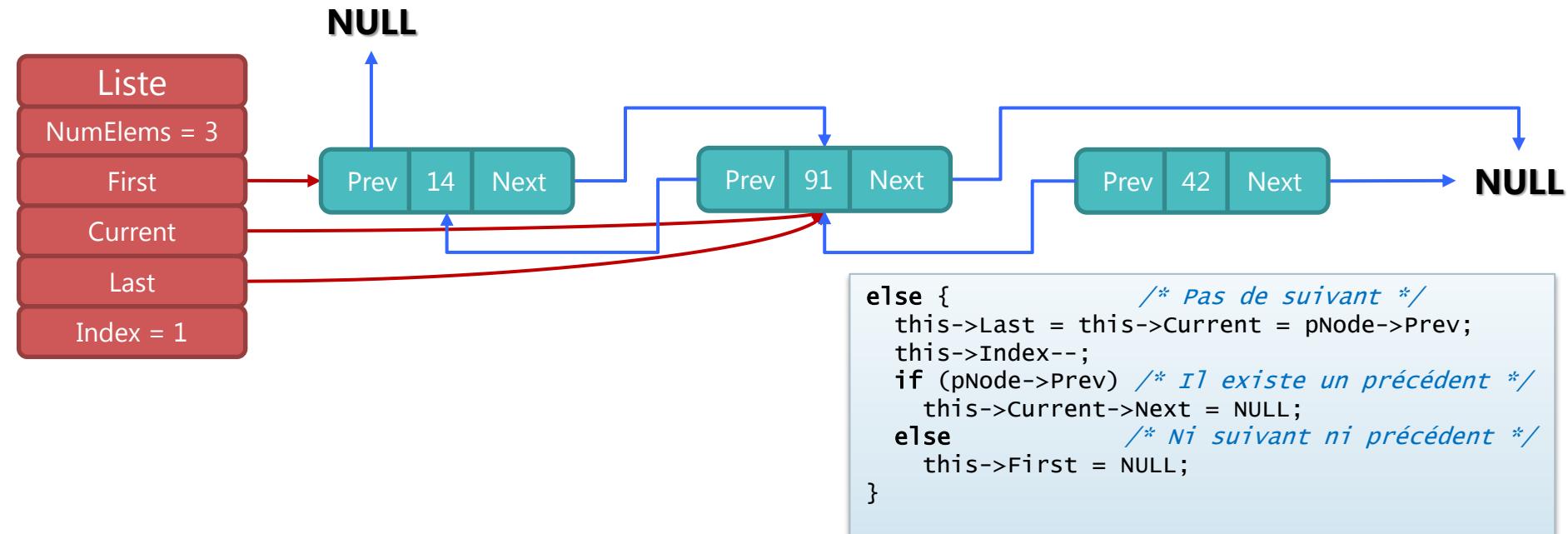
- 5) Si pas de suivant, c'est le précédent qui devient le nouveau nœud courant,



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (8/9)

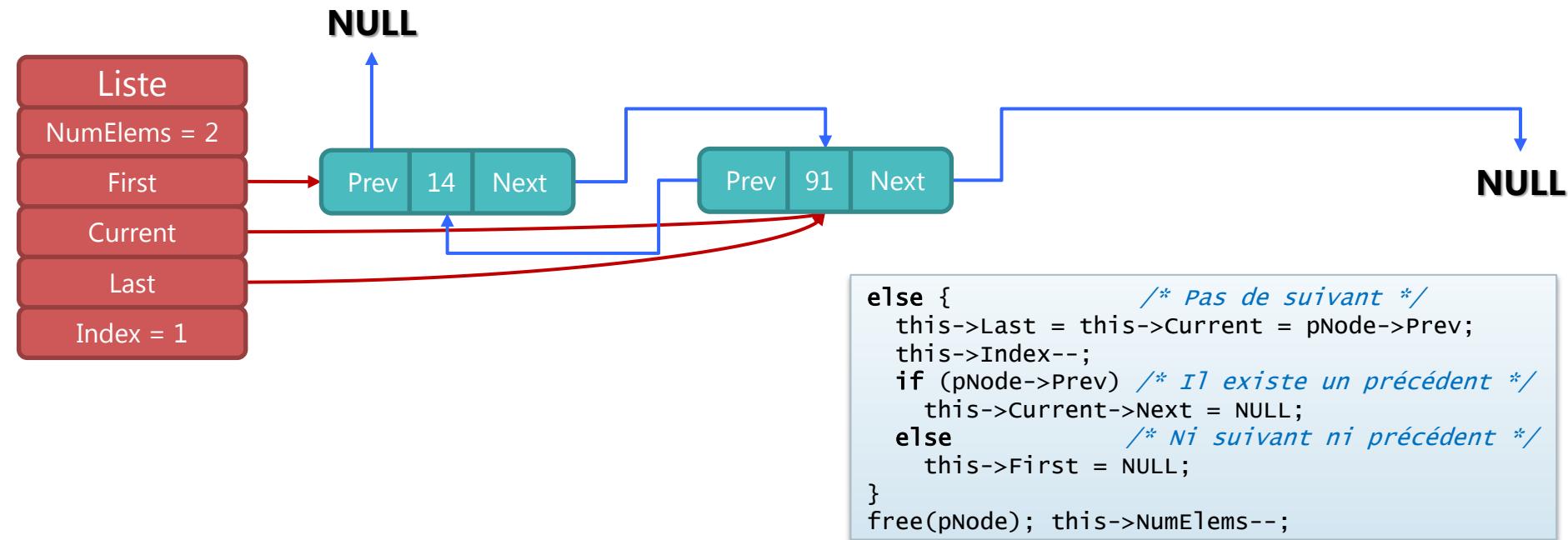
- 5) Si pas de suivant, c'est le précédent qui devient le nouveau nœud courant,
- 6) et pointe vers NULL (s'il n'est pas lui-même égal à NULL, auquel cas la liste serait vide)



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : retrait du nœud courant (9/9)

- 5) Si pas de suivant, c'est le précédent qui devient le nouveau nœud courant,
- 6) et pointe vers NULL (s'il n'est pas lui-même égal à NULL, auquel cas la liste serait vide)
- 7) On peut maintenant supprimer l'ancien nœud courant.



LES LISTES CHAINÉES

Fonction Tlist_RemoveCurrent : code C

```

bool TList_RemoveCurrent(const PTList this)
{
    if (TList_IsEmpty(this)) return false;
    PTNode pNode = this->Current;
    if (pNode->Next) {           /* Il existe un suivant */
        this->Current = pNode->Next;
        this->Current->Prev = pNode->Prev;
        if (pNode->Prev)          /* Il existe un précédent */
            pNode->Prev->Next = this->Current;
        else
            this->First = this->Current;
    }
    else {                      /* Pas de suivant */
        this->Last = this->Current = pNode->Prev;
        this->Index--;
        if (pNode->Prev)          /* Il existe un précédent */
            this->Current->Next = NULL;
        else                      /* Ni suivant ni précédent */
            this->First = NULL;
    }
    free(pNode);
    this->NumElems--;
    return true;
}

```

PLAN DU COURS

Méthodes de développement logiciel

Rappels et approfondissements en C

Types abstraits de données

Listes chainées

Piles et files

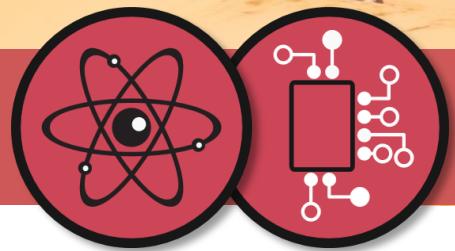
Algorithmes de tri

Tables de hachage

Arbres binaires



Piles et files



Les piles - Implémentation à l'aide d'une liste chaînée

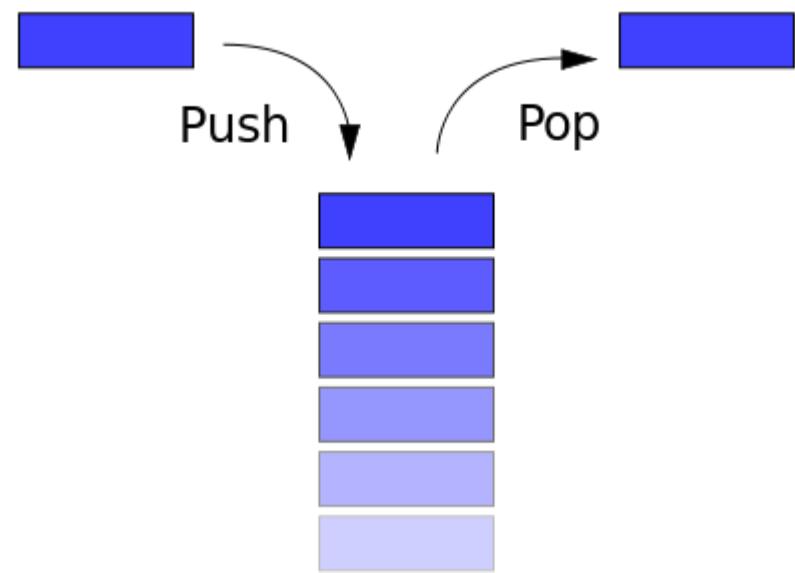
Les files - Implémentation à l'aide d'une liste chaînée simple

Implémentation d'une file à l'aide d'un tableau circulaire

PILES ET FILES

Les piles (stack)

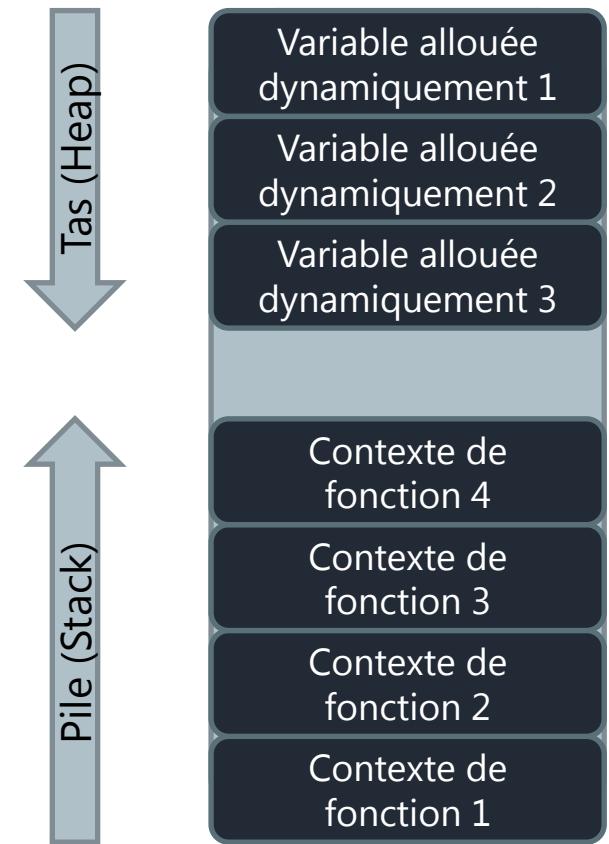
- La *pile* est un TAD permettant l'ajout ou la suppression d'un élément à partir d'un ensemble de données généralement de même type.
- La manière dont les données sont traitées dans une pile est de type *LIFO* : *Last In, First Out*.
- Le terme *pile* vient du fait que les données sont empilées comme une pile d'objets dont seul le sommet est accessible.



PILES ET FILES

Importance des piles en informatique

- La fonction *Undo* d'un éditeur de texte mémorise dans une pile les modifications apportées au texte.
- Dans un browser web, une pile sert à mémoriser les pages web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « *Page précédente* ».
- Les microprocesseurs gèrent nativement une pile. Elle correspond alors à une zone de la mémoire, et le processeur retient l'adresse du dernier élément.
- La *pile d'exécution (call stack)* est l'endroit où sont empilés les paramètres d'appel des fonctions. Par ailleurs, on y crée un espace pour des variables locales.



PILES ET FILES

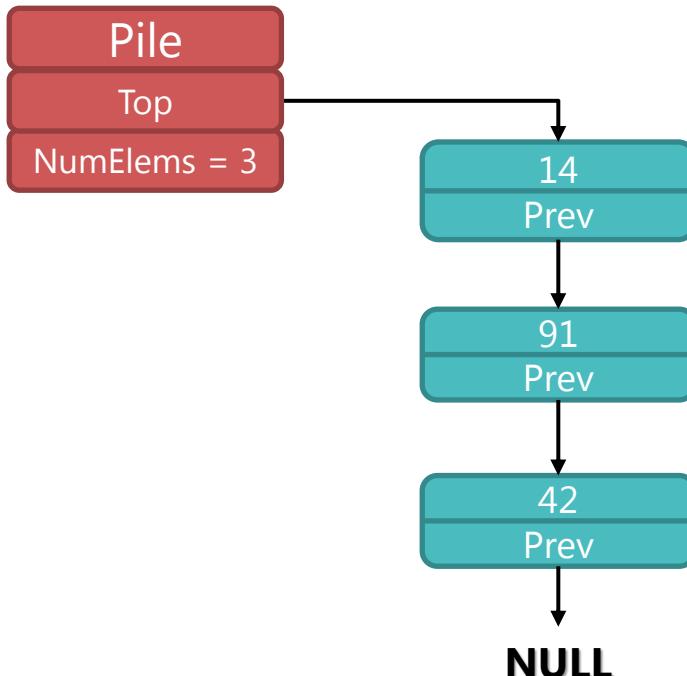
Implémentation d'une pile à l'aide d'une liste chaînée simple

- L'implémentation d'une pile est directe à partir d'une liste
- Le sommet de la pile correspond à la tête de liste avec une liste simple
- Les équivalences de fonctions sont les suivantes :
 - Push \leftrightarrow InsertFirst
 - Pop \leftrightarrow RemoveFirst

PILES ET FILES

Structure de pile en C

- On peut représenter une pile avec juste un pointeur qui contient l'adresse du sommet.
- Une structure a l'avantage de permettre de mémoriser en plus la taille de la pile.



```

typedef struct Node {
    TElement Element;
    struct Node *Prev;
} TNode;

typedef TNode *PTNode;

typedef struct Stack {
    int NumElems; /* Nombre d'éléments */
    PTNode Top; /* Sommet */
} TStack;

typedef TStack *PTStack;
  
```

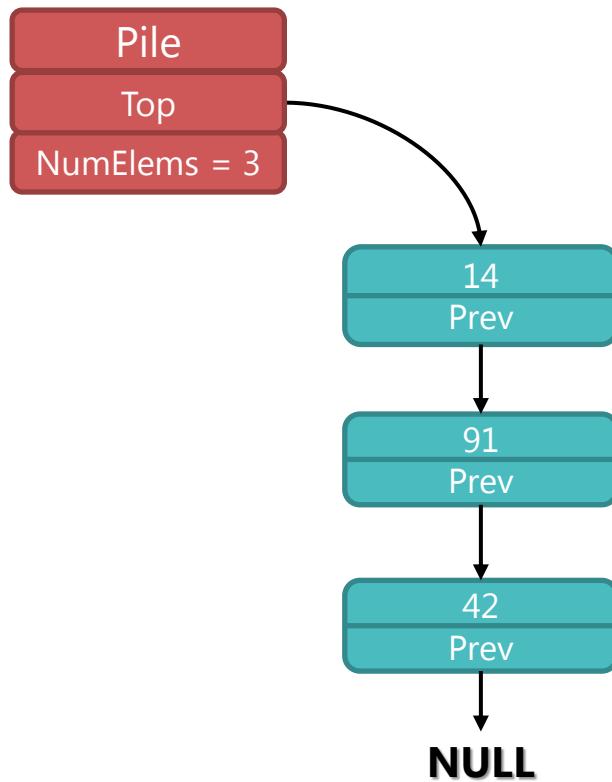
PILES ET FILES

Opérations sur le TAD pile

- **Nom :** TStack,
- **Utilise :** int, bool, TElement, TNode
- **Opérations :**
 - New → TStack
 - IsEmpty : TStack → bool
 - Size : TStack → int
 - Top : TStack → TNode
 - Push : TStack × TElement → TNode
 - Pop : TStack → bool × TElement
 - Clear : TStack →
 - Display : TStack →
 - Delete : TStack →

PILES ET FILES

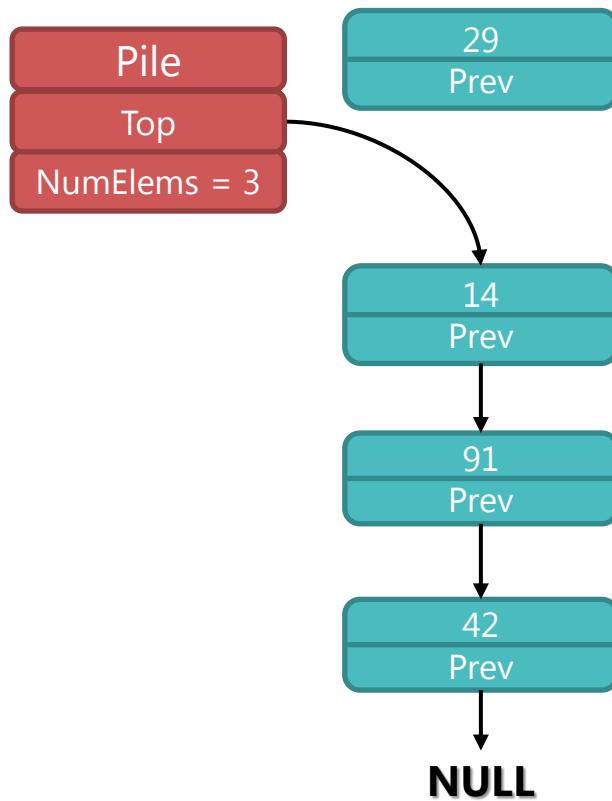
Fonction TStack_Push : ajout d'un élément au sommet de la pile (1/5)



```
PTNode TStack_Push(const PTStack this, TElement Element)
{
```

PILES ET FILES

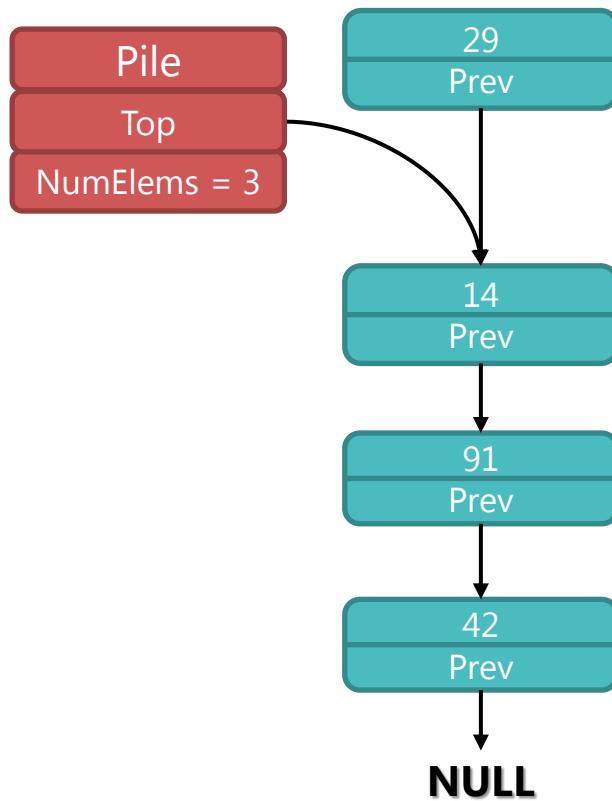
Fonction TStack_Push : ajout d'un élément au sommet de la pile (2/5)



```
PTNode TStack_Push(const PTStack this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
```

PILES ET FILES

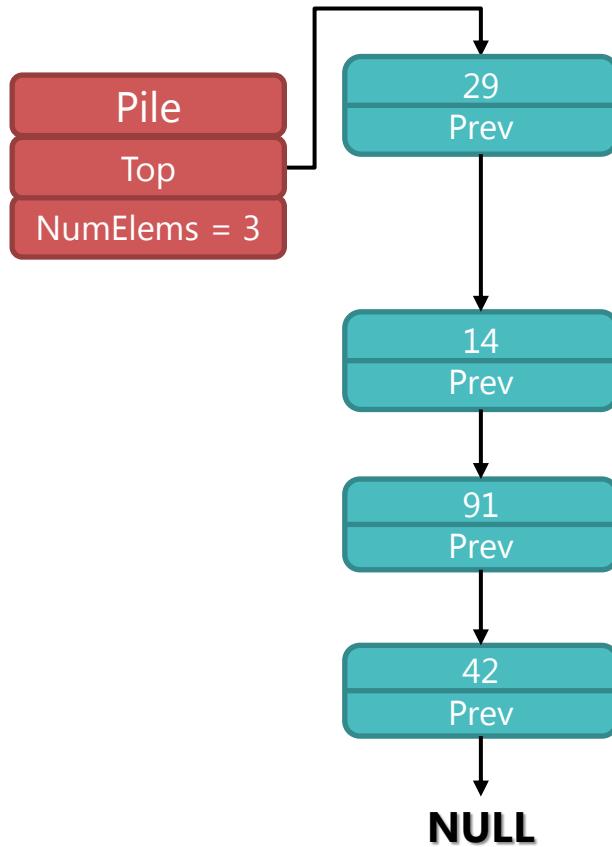
Fonction TStack_Push : ajout d'un élément au sommet de la pile (3/5)



```
PTNode TStack_Push(const PTStack this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Prev = this->Top;
```

PILES ET FILES

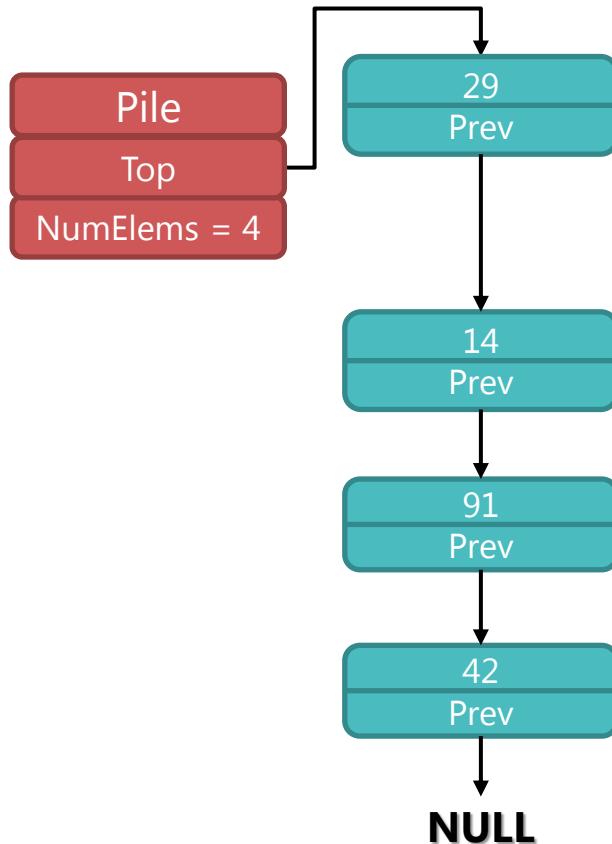
Fonction TStack_Push : ajout d'un élément au sommet de la pile (4/5)



```
PTNode TStack_Push(const PTStack this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Prev = this->Top;
    this->Top = newNode;
```

PILES ET FILES

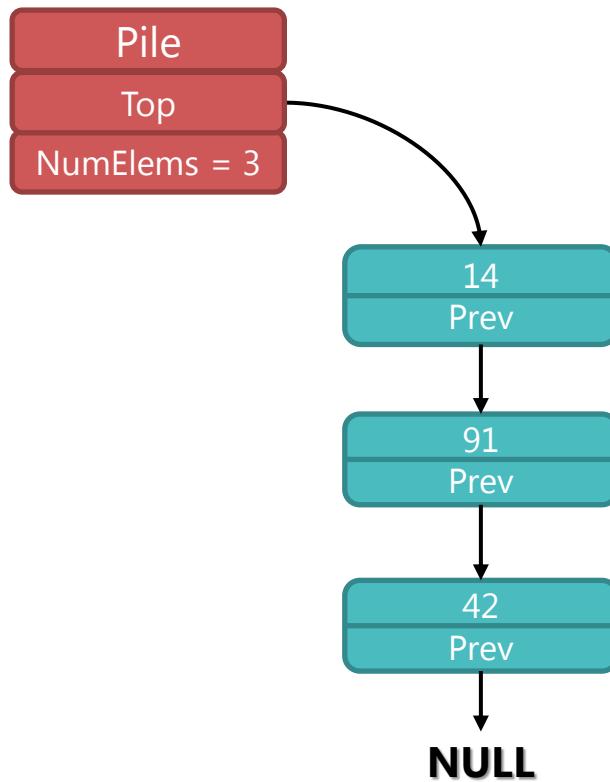
Fonction TStack_Push : ajout d'un élément au sommet de la pile (5/5)



```
PTNode TStack_Push(const PTStack this, TElement Element)
{
    PTNode newNode = (PTNode)malloc(sizeof(TNode));
    if (newNode == NULL) return NULL;
    newNode->Element = Element;
    newNode->Prev = this->Top;
    this->Top = newNode;
    this->NumElems++;
    return newNode;
}
```

PILES ET FILES

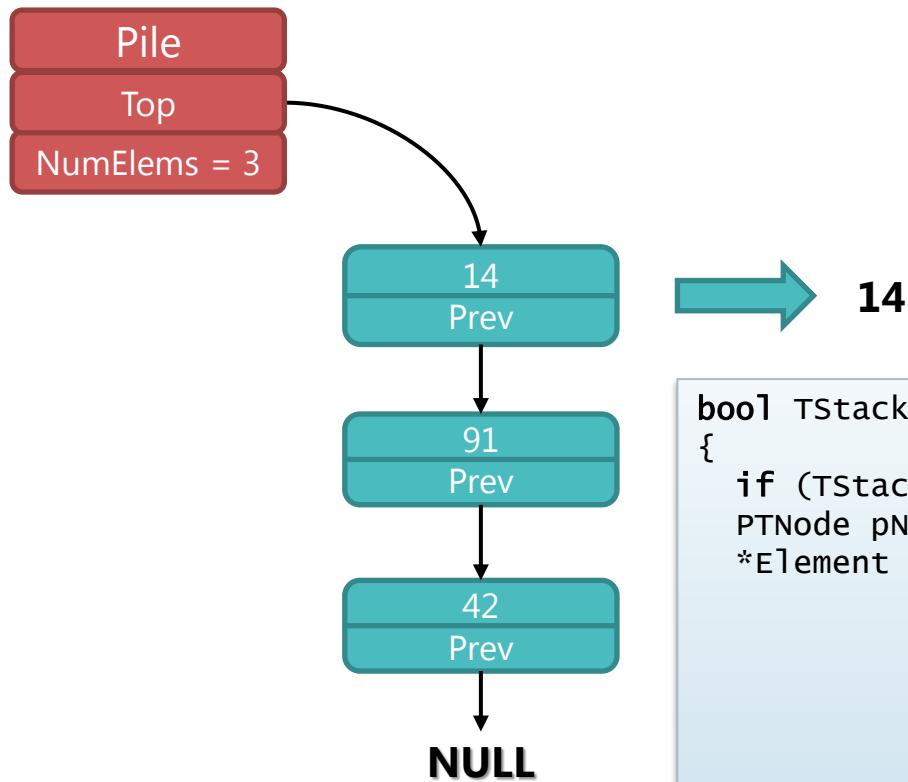
Fonction TStack_Pop : retrait de l'élément au sommet de la pile (1/5)



```
bool TStack_Pop(const PTStack this, TElement *Element)
{
    if (TStack_IsEmpty(this)) return false;
    PTNode pNode = this->Top, prevNode = pNode->Prev;
```

PILES ET FILES

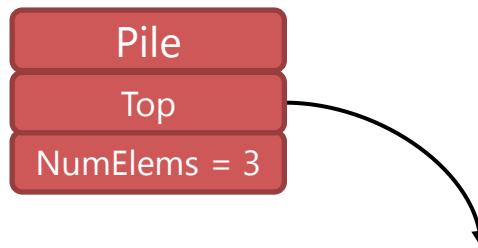
Fonction TStack_Pop : retrait de l'élément au sommet de la pile (2/5)



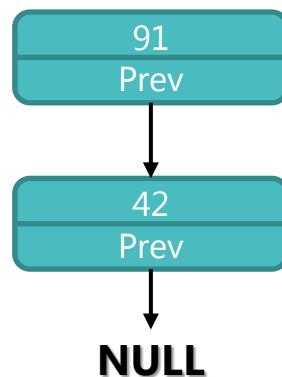
```
bool TStack_Pop(const PTStack this, TElement *Element)
{
    if (TStack_IsEmpty(this)) return false;
    PTNode pNode = this->Top, prevNode = pNode->Prev;
    *Element = pNode->Element;
```

PILES ET FILES

Fonction TStack_Pop : retrait de l'élément au sommet de la pile (3/5)



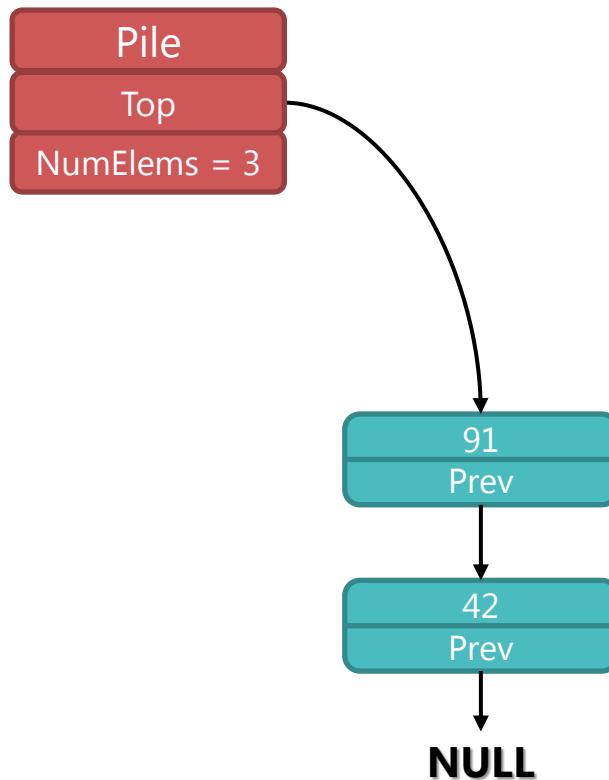
14



```
bool TStack_Pop(const PTStack this, TElement *Element)
{
    if (TStack_IsEmpty(this)) return false;
    PTNode pNode = this->Top, prevNode = pNode->Prev;
    *Element = pNode->Element;
    free(pNode);
```

PILES ET FILES

Fonction TStack_Pop : retrait de l'élément au sommet de la pile (4/5)



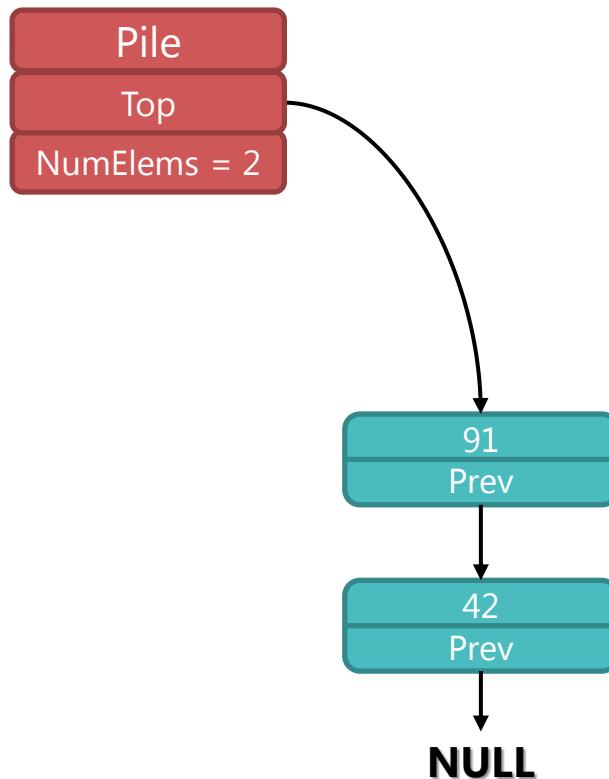
14

```

bool TStack_Pop(const PTStack this, TElement *Element)
{
    if (TStack_IsEmpty(this)) return false;
    PTNode pNode = this->Top, prevNode = pNode->Prev;
    *Element = pNode->Element;
    free(pNode);
    this->Top = prevNode;
}
  
```

PILES ET FILES

Fonction TStack_Pop : retrait de l'élément au sommet de la pile (5/5)



14

```

bool TStack_Pop(const PTStack this, TElement *Element)
{
    if (TStack_IsEmpty(this)) return false;
    PTNode pNode = this->Top, prevNode = pNode->Prev;
    *Element = pNode->Element;
    free(pNode);
    this->Top = prevNode;
    this->NumElems--;
    return true;
}
  
```

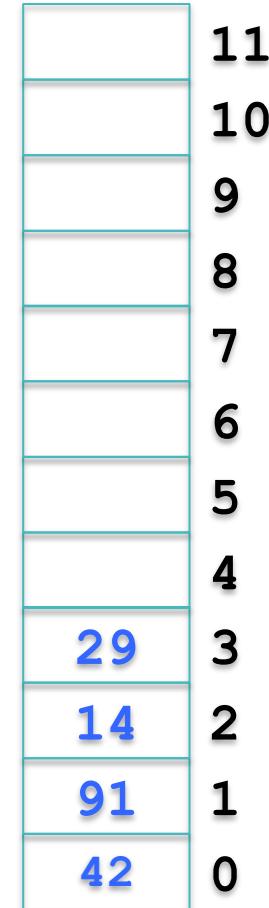
PILES ET FILES

Implémentation d'une pile à l'aide d'un tableau

- Une pile peut également être représentée par un tableau.
- Une structure C permet de mémoriser le nombre d'éléments et la taille maximale.

```
typedef struct Stack {
    int NumElements;           /* Nombre d'éléments */
    int MaxSize;               /* Taille maximale */
    TElement *Table;           /* Contenu */
} TStack;

typedef TStack *PTStack;
```



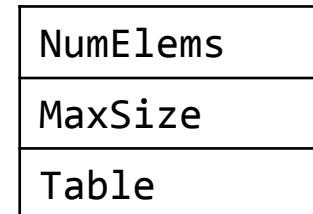
PILES ET FILES

/ Attention aux allocations mémoire ! */*

```
PTStack TStack_New(int maxSize)
{
    PTStack this = (PTStack)malloc(sizeof(TStack));
    if (this == NULL) return NULL;
    this->Table = (PTElement)calloc(maxSize, sizeof(TElement));
    if (this->Table == NULL) {
        free(this);
        return NULL;
    }
    this->NumElems = 0;
    this->MaxSize = maxSize;
    return this;
}
```

/ Libération dans l'ordre inverse */*

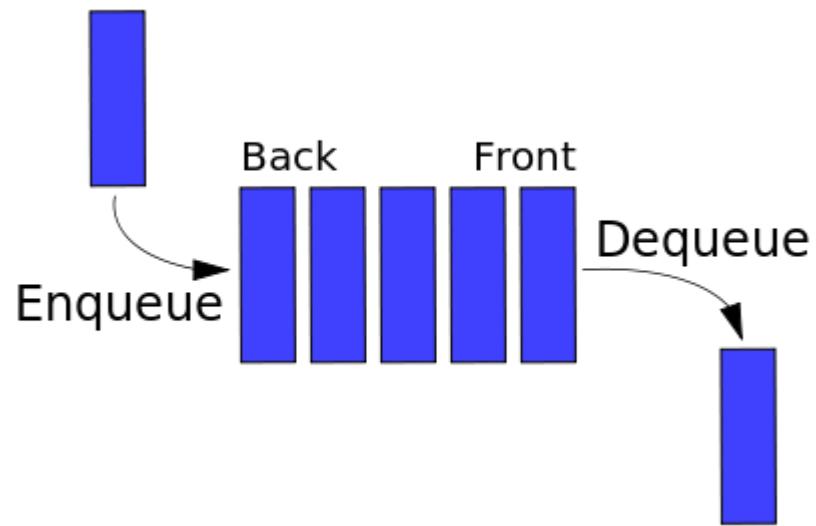
```
void TStack_Delete(const PTStack this)
{
    free(this->Table);
    free(this);
}
```



PILES ET FILES

Les files (queue)

- La *file*, dite aussi *file d'attente* ou *queue* est un TAD assez semblable à la pile, mais contrairement à cette dernière, la manière dont les données sont traitées dans une file est de type FIFO : *First In, First Out*.
- Le terme *file* ou *queue* vient du fait que ce sont les données entrées les premières qui ressortent en premier, à la manière des personnes dans une qui « font la queue » dans une file d'attente.
- Un exemple d'utilisation est la file d'attente des tâches à imprimer dans un *spooler* d'impression



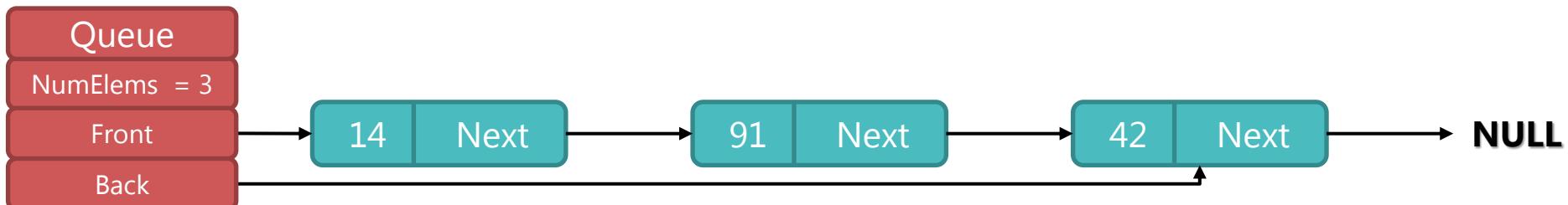
PILES ET FILES

Implémentation d'une file à l'aide d'une liste chaînée simple

- L'implémentation d'une file est directe à partir d'une liste chaînée simple (cf. chapitre sur les listes)
- Les équivalences de fonctions sont les suivantes :
 - Enqueue \leftrightarrow Add
 - Dequeue \leftrightarrow RemoveFirst
- Ces choix doivent permettre une implémentation efficace des files d'attente en C.

PILES ET FILES

Structure de file en C



```

typedef struct Node {
    TElement Element;
    struct Node *Next;
} TNode;

typedef TNode *PTNode;

typedef struct Queue {
    int NumElems; /* Nombre d'éléments */
    PTNode Front; /* Tête */
    PTNode Back; /* Queue */
} TQueue;

typedef TQueue *PTQueue;
  
```

PILES ET FILES

Opérations sur le TAD file

- **Nom :** TQueue,
- **Utilise :** int, bool, TElement, TNode
- **Opérations :**
 - New → TQueue
 - IsEmpty : TQueue → bool
 - Length : TQueue → int
 - Enqueue : TQueue × TElement → TNode
 - Dequeue : TQueue → bool × TElement
 - Clear : TQueue →
 - Display : TQueue →
 - Delete : TQueue →

PILES ET FILES

Implémentation d'une file à l'aide d'un tableau circulaire

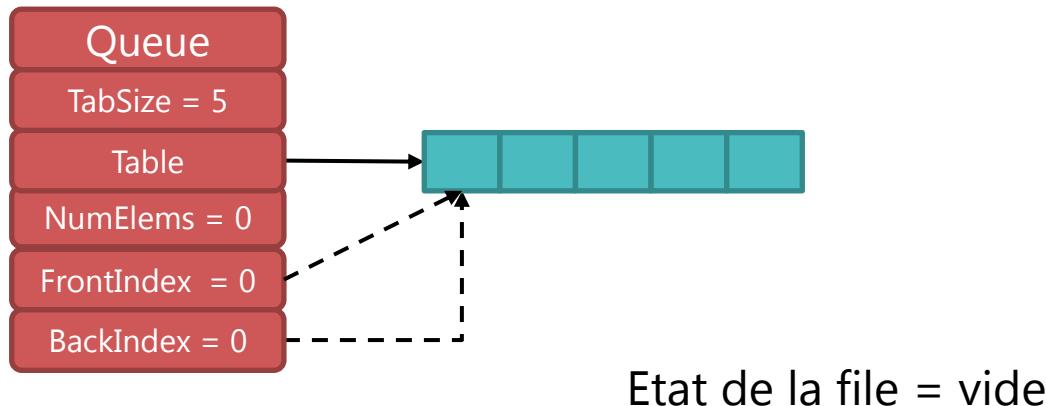
- L'implémentation d'une file est possible avec un tableau, mais cependant beaucoup plus complexe que l'implémentation d'une pile.
- Il faut en effet gérer simultanément deux indices, correspondant respectivement à la tête et à la queue de la file.
- La structure à déclarer est la suivante :

```
typedef struct Queue {  
    int TabSize;           /* Taille maxi alloué au tableau */  
    TElement *Table;       /* Tableau contenant les éléments de la file */  
    int NumElems;          /* Nombre d'éléments dans la file */  
    int FrontIndex;        /* Indice du 1er élément de la file */  
    int BackIndex;         /* Indice du dernier élément de la file */  
} TQueue ;
```

PILES ET FILES

Organisation des éléments de la file dans le tableau (1/2)

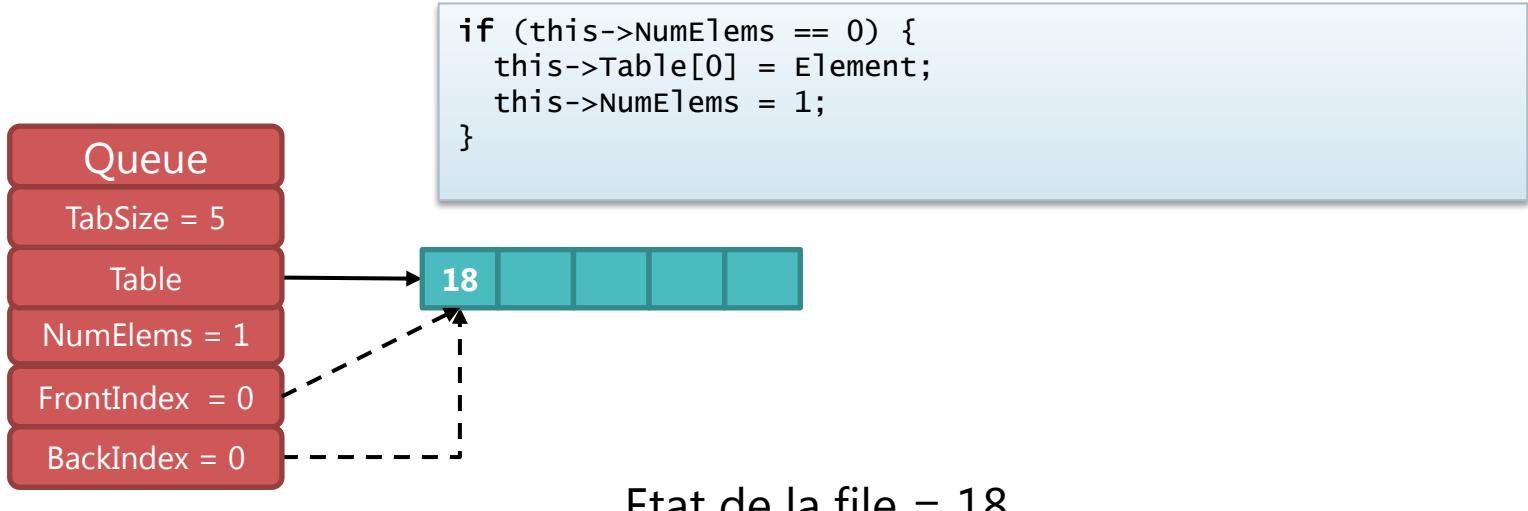
- Initialement, un espace de $TabSize$ éléments est alloué au tableau $Table$. Les indices $FrontIndex$ et $BackIndex$ correspondent respectivement à l'indice du premier et du dernier élément de la file. A l'initialisation la file est vide et ces deux indices sont nuls.



PILES ET FILES

Organisation des éléments de la file dans le tableau (2/2)

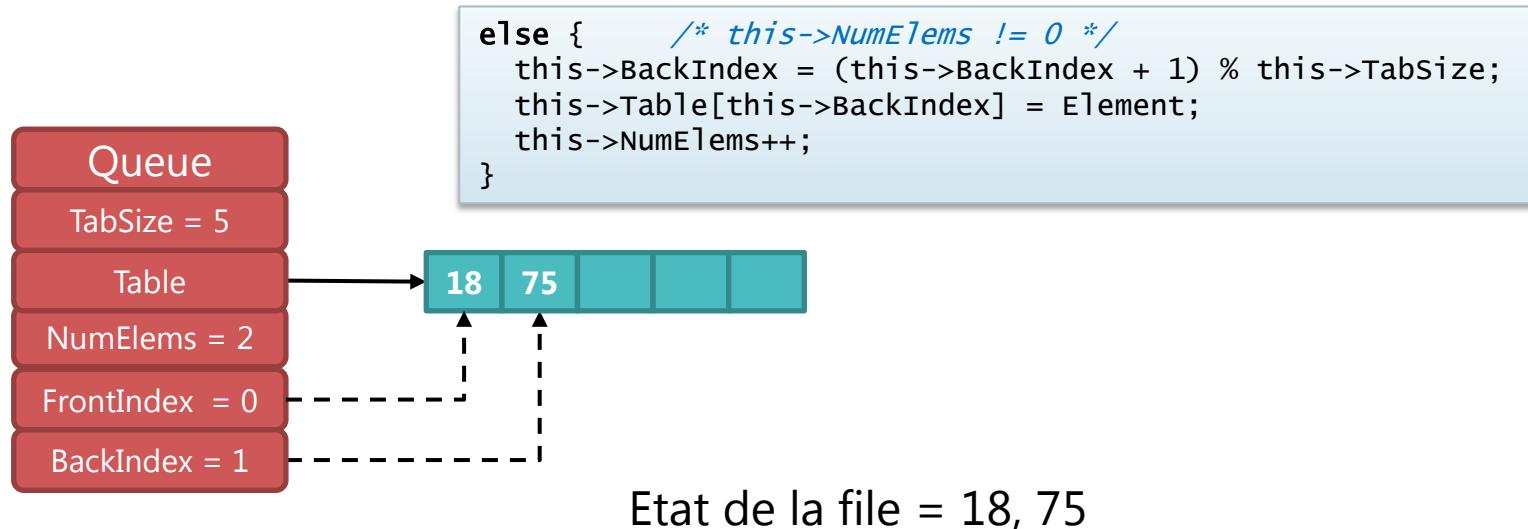
- Initialement, un espace de *TabSize* éléments est alloué au tableau *Table*. Les indices *FrontIndex* et *BackIndex* correspondent respectivement à l'indice du premier et du dernier élément de la file. A l'initialisation la file est vide et ces deux indices sont nuls.
- Dans cette file vide, un premier élément sera placé dans la première case du tableau, ce qui laisse inchangées les valeurs de *FrontIndex* et *BackIndex*.



PILES ET FILES

Ajout de nouveaux éléments dans la file (1/3)

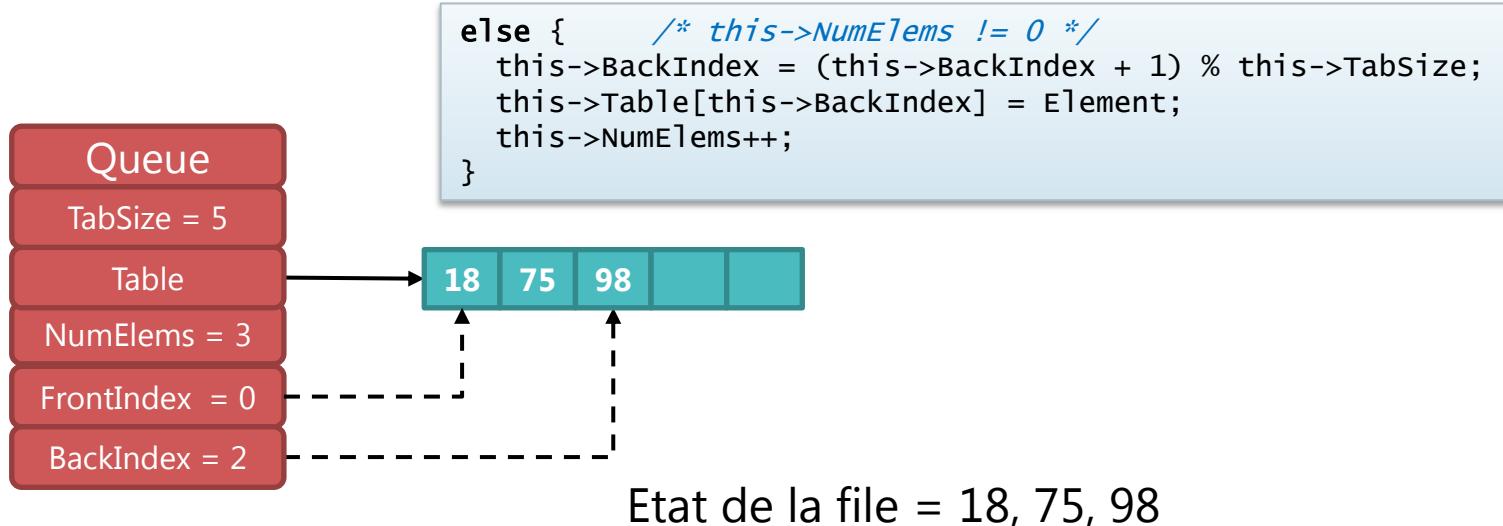
- Les éléments ajoutés en queue de file sont ajoutés dans les cases suivantes du tableau, et l'indice *BackIndex* est donc incrémenté à chaque ajout.



PILES ET FILES

Ajout de nouveaux éléments dans la file (2/3)

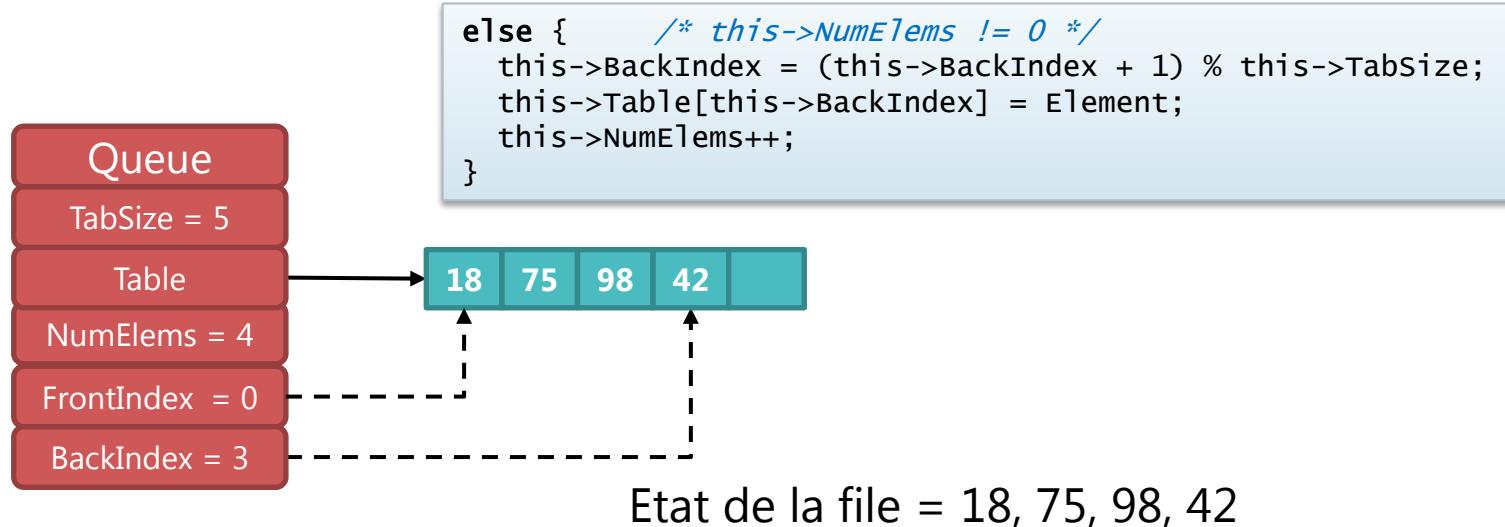
- Les éléments ajoutés en queue de file sont ajoutés dans les cases suivantes du tableau, et l'indice *BackIndex* est donc incrémenté à chaque ajout.



PILES ET FILES

Ajout de nouveaux éléments dans la file (3/3)

- Les éléments ajoutés en queue de file sont ajoutés dans les cases suivantes du tableau, et l'indice *BackIndex* est donc incrémenté à chaque ajout.

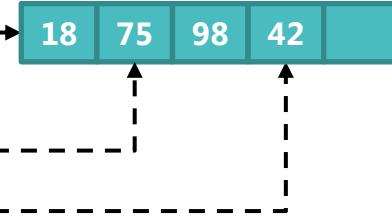


PILES ET FILES

Retrait d'un élément de la file (1/2)

- Pour retirer un élément de la file, il suffit d'incrémenter l'indice *FrontIndex*. Au fur et à mesure de l'évolution de la file, ses éléments vont donc se décaler vers la droite du tableau.

```
bool TQueue_Dequeue(const PTQueue this, TElement *Element)
{
    if (this->NumElems == 0) return false;
    *Element = this->Table[this->FrontIndex];
    this->FrontIndex = (this->FrontIndex + 1) % this->TabSize;
    this->NumElems--;
    if (this->NumElems == 0) /* Si file vide, RAZ des indices */
        this->FrontIndex = this->BackIndex = 0;
    return true;
}
```



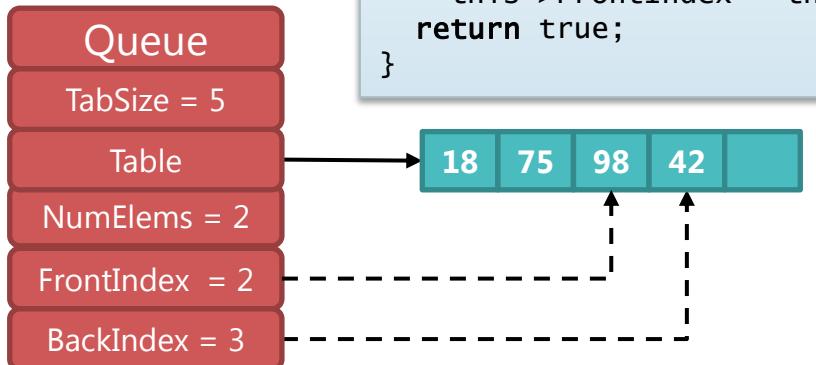
Etat de la file = 75, 98, 42

PILES ET FILES

Retrait d'un élément de la file (2/2)

- Pour retirer un élément de la file, il suffit d'incrémenter l'indice *FrontIndex*. Au fur et à mesure de l'évolution de la file, ses éléments vont donc se décaler vers la droite du tableau.

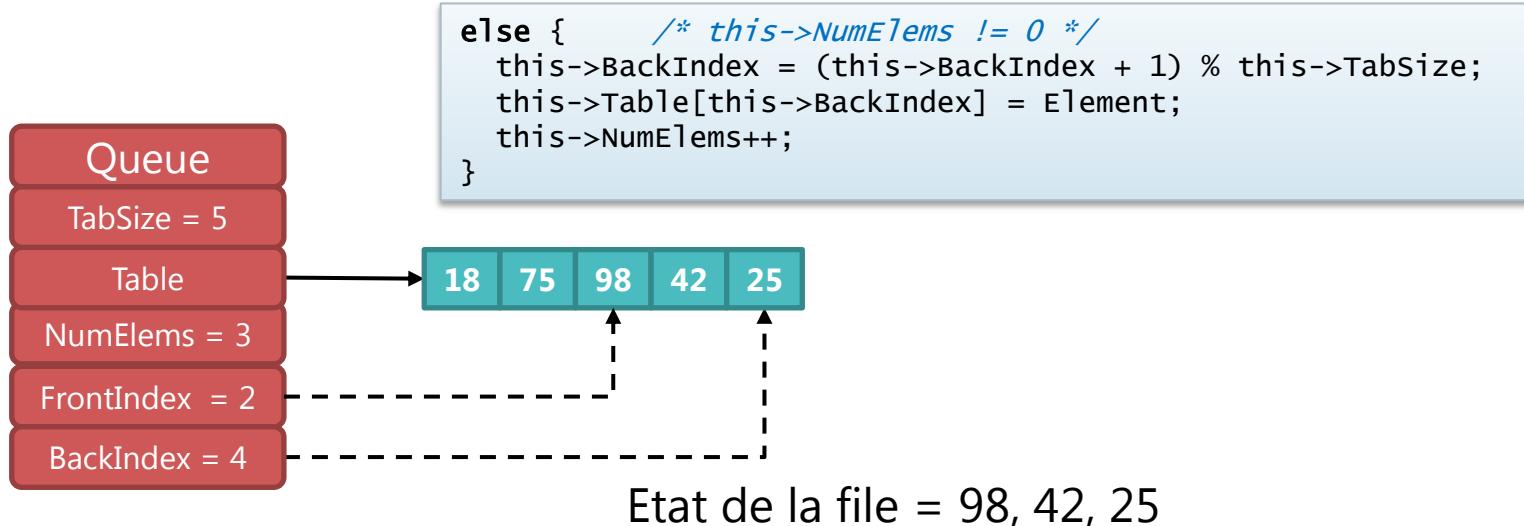
```
bool TQueue_Dequeue(const PTQueue this, TElement *Element)
{
    if (this->NumElems == 0) return false;
    *Element = this->Table[this->FrontIndex];
    this->FrontIndex = (this->FrontIndex + 1) % this->TabSize;
    this->NumElems--;
    if (this->NumElems == 0) /* Si file vide, RAZ des indices */
        this->FrontIndex = this->BackIndex = 0;
    return true;
}
```



PILES ET FILES

Ajout d'un élément au-delà de l'indice maximal du tableau (1/2)

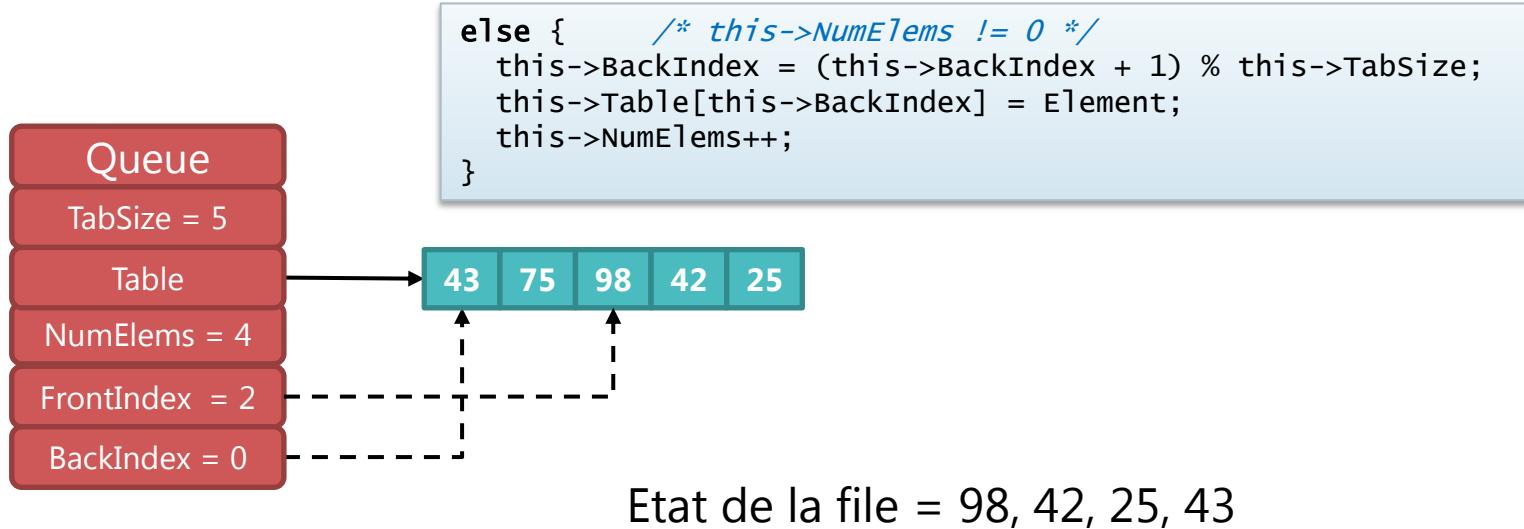
- Comme on ne peut indéfiniment décaler la file vers la droite, lorsque qu'on ajoute un élément en queue de file alors que l'on a atteint l'indice maximal du tableau, cet élément est placé au début du tableau comme si la première et la dernière case étaient adjacentes : on emploie le terme de « tableau circulaire ».



PILES ET FILES

Ajout d'un élément au-delà de l'indice maximal du tableau (2/2)

- Comme on ne peut indéfiniment décaler la file vers la droite, lorsque qu'on ajoute un élément en queue de file alors que l'on a atteint l'indice maximal du tableau, cet élément est placé au début du tableau comme si la première et la dernière case étaient adjacentes : on emploie le terme de « tableau circulaire ».



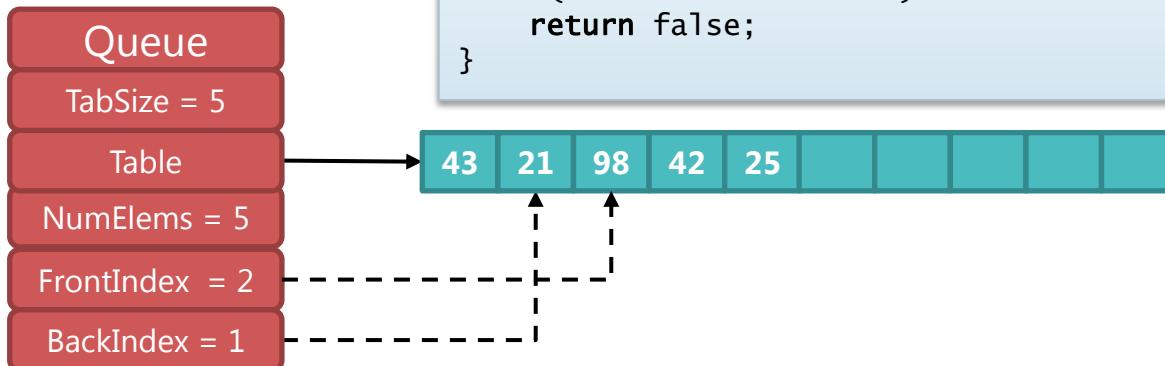
PILES ET FILES

Réorganisation des éléments de la file après réallocation dynamique (1/3)

- Pour continuer à accepter de nouveaux éléments dans la file lorsqu'elle est pleine, il faut augmenter dynamiquement la taille du tableau.
- Si l'indice *FrontIndex* est différent de 0, il faudra alors réorganiser le tableau après la réallocation.

```

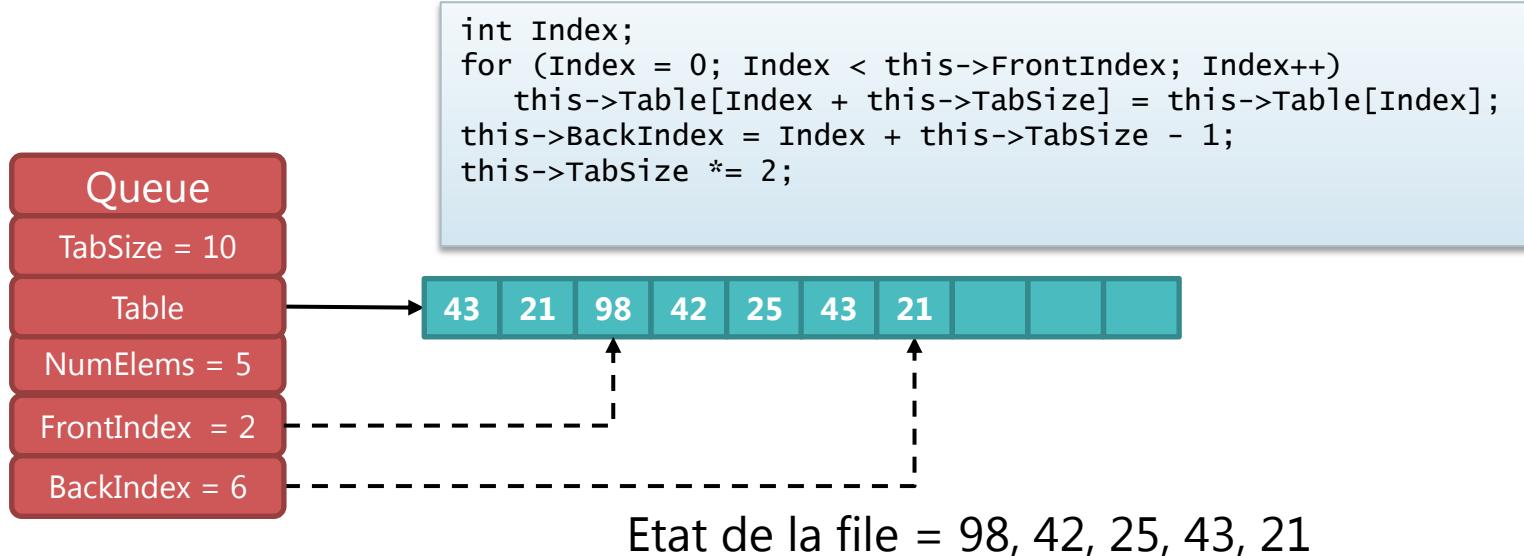
if (this->NumElems == this->TabSize) {
    this->Table = (TElement *)realloc(this->Table,
                                         this->TabSize*2*sizeof(TElement));
    if (this->Table == NULL)
        return false;
}
  
```



PILES ET FILES

Réorganisation des éléments de la file après réallocation dynamique (2/3)

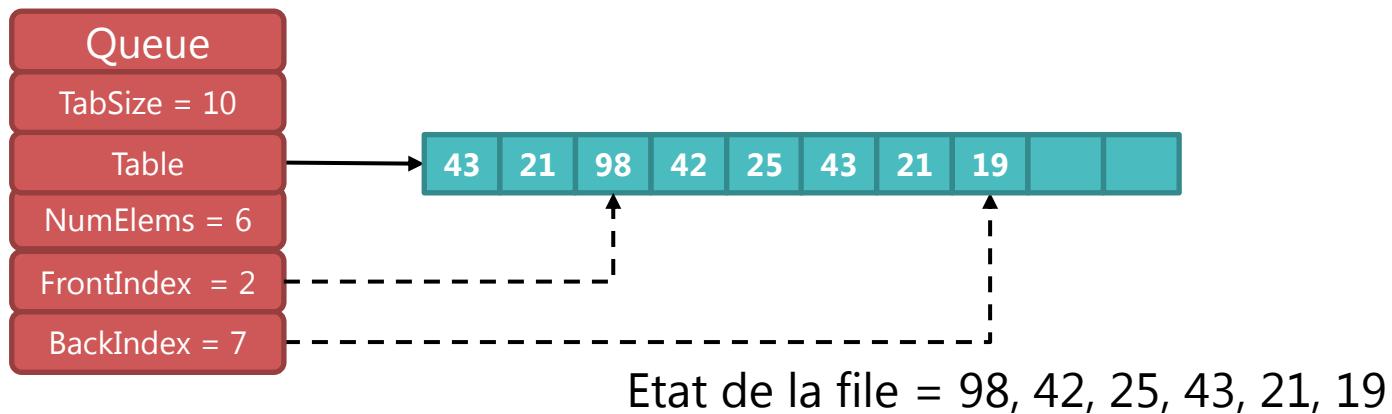
- Une manière simple de réorganiser les données à minima est de déplacer les éléments de la queue de la file situés au début du tableau, à la suite des autres éléments.
- Après réorganisation :



PILES ET FILES

Réorganisation des éléments de la file après réallocation dynamique (3/3)

- Une manière simple de réorganiser les données à minima est de déplacer les éléments de la queue de la file situés au début du tableau, à la suite des autres éléments.
- Après réorganisation, on peut ensuite accepter de nouveaux éléments en queue de file.



PLAN DU COURS



Méthodes de développement logiciel

Rappels et approfondissements en C

Types abstraits de données

Listes chainées

Piles et files

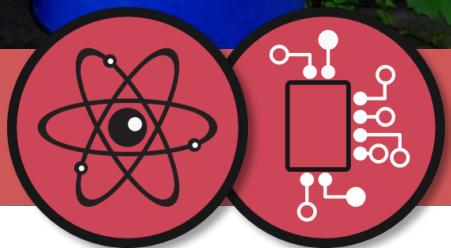
Algorithmes de tri

Tables de hachage

Arbres binaires



Algorithmes de tri



Complexité algorithmique - *Tri par sélection* - *Tri par insertion*

Tri à bulles - *Tri fusion* - *Tri rapide (quick sort)*

Comparaisons des algorithmes de tri

ALGORITHMES DE TRI

Motivations

- Pour la résolution d'un grand nombre de problèmes, il est généralement important que les données soient triées, ce qui implique l'existence d'une *notion d'ordre*.
- L'affichage d'un ensemble de données peut être beaucoup plus lisible, si celles-ci sont au préalable triées.
- La recherche d'un élément dans un ensemble de données est plus efficace (plus rapide) si ces données sont triées au préalable.
- Il existe de nombreux algorithmes de tri :
 - http://fr.wikipedia.org/wiki/Algorithme_de_tri
- L'efficacité, et donc le choix d'un algorithme de tri va dépendre entre autres :
 - de la taille des données,
 - du TAD employé pour les stocker : tableau ou liste,
 - de l'arrangement initial des données,
 - de la possibilité de paralléliser le traitement,
 - ...

ALGORITHMES DE TRI

Complexité algorithmique

- Estimation du coût en nombre d'opérations, en fonction du nombre n de données (bits, octets ou mots) à traiter. On dit que la complexité est $O(f(n))$.
- Classes de complexité :
 - constante : $O(1)$
 - logarithmique : $O(\log(n))$
 - linéaire : $O(n)$ ou $O(n\log(n))$
 - polynomiale : $O(n^k)$ avec $k \geq 2$ (*quadratique* si $k = 2$, *cubique* si $k = 3$)
 - exponentielle : $O(2^n)$
 - factorielle : $O(n!)$

ALGORITHMES DE TRI

Problèmes traitables et non-traitables

- Les problèmes que l'on peut résoudre par un algorithme de complexité sous-exponentielle sont dits *traitables*, alors que ceux que l'on ne peut résoudre que par un algorithme de complexité exponentielle sont dits *intraitables* (impossible à résoudre en pratique quand n augmente).
- On peut mesurer l'efficacité d'un algorithme relativement à un problème donné : on peut estimer par exemple qu'un algorithme de tri de complexité quadratique est inefficace s'il existe d'autres algorithmes de tri de complexité linéaire. Par contre, un algorithme de complexité cubique peut être considéré comme efficace si c'est le meilleur par rapport au problème posé (c'est le cas par exemple de *l'exponentiation modulaire rapide*).

ALGORITHMES DE TRI

Exemple de complexité algorithmique : recherche dans une collection

- Problème : rechercher une information particulière dans une collection de n éléments.
- Recherche séquentielle : consiste à parcourir la collection à partir du début jusqu'à trouver l'élément recherché. Il faut effectuer en moyenne $n/2$ comparaisons, donc la complexité est en $O(n)$. Cette méthode fonctionne sur une collection triée ou non.
- Recherche dichotomique : on compare l'élément recherché avec celui du milieu de l'ensemble. On recommence récursivement avec la moitié gauche (si l'élément recherché est inférieur) ou droite (si l'élément recherché est supérieur) de la collection de données, jusqu'à trouver l'élément recherché dans la collection. On divise ainsi l'ensemble de recherche par 2 à chaque itération : la complexité est donc en $O(\log(n))$. Cette méthode ne fonctionne que sur une collection triée, d'où l'intérêt de disposer d'ensemble de données triés.

ALGORITHMES DE TRI

Autres caractéristiques des algorithmes de tri

- Caractère en place : un algorithme de tri est dit *en place* s'il modifie directement la structure de données qu'il est en train de trier, sans passer par un espace mémoire intermédiaire.
- Stabilité : un algorithme de tri est dit *stable* s'il garde l'ordre relatif des éléments égaux pour la relation d'ordre considérée, c'est-à-dire l'ordre de ces éléments avant l'exécution de l'algorithme. Exemples d'algorithmes stables :
 - *tri par insertion*,
 - *tri à bulles*,
 - *tri fusion*

ALGORITHMES DE TRI

Tri par sélection

- Principe :
 - rechercher parmi les n éléments à trier celui de plus petite valeur et l'échanger avec celui placé en première position,
 - répéter itérativement la même procédure avec les $n - 1$ éléments restants, en éliminant celui venant d'être positionné.
- Complexité : le nombre d'éléments à parcourir est n , puis $n - 1$, ..., 3, 2, et enfin 1. Soit un total de $n(n - 1)/2$ éléments lus. La complexité de cette algorithme est donc en $O(n^2)$.
- Même s'il est simple dans sa conception et son implémentation, c'est l'algorithme de tri le moins efficace.
- Chaque étape nécessitant que l'étape précédente soit terminé, cet algorithme n'est pas parallélisable dans sa version de base.

ALGORITHMES DE TRI

Tri par sélection : exemple

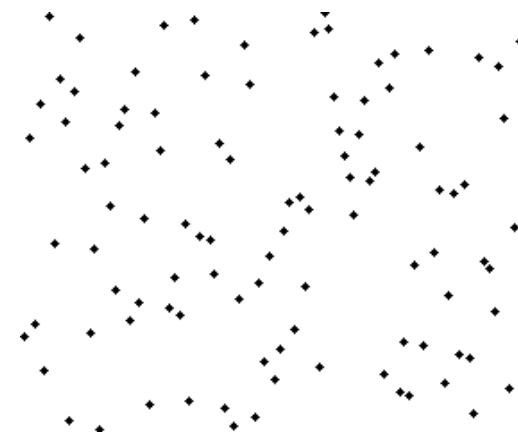
- 18 75 98 42 25 31 55 24 7 31 12 7
- 7 75 98 42 25 31 55 24 18 31 12 7
- 7 7 98 42 25 31 55 24 18 31 12 75
- 7 7 12 42 25 31 55 24 18 31 98 75
- 7 7 12 18 25 31 55 24 42 31 98 75
- 7 7 12 18 24 31 55 25 42 31 98 75
- 7 7 12 18 24 25 55 31 42 31 98 75
- 7 7 12 18 24 25 31 55 42 31 98 75
- 7 7 12 18 24 25 31 31 42 55 98 75
- 7 7 12 18 24 25 31 31 42 55 98 75
- 7 7 12 18 24 25 31 31 42 55 75 98
- 7 7 12 18 24 25 31 31 42 55 75 98

ALGORITHMES DE TRI

Tri par sélection d'un tableau : implémentation en C

```
void SelectionSort(TElement *tabElems, int numElems)
{
    for (int offset = 0; offset < numElems - 1; offset++) {
        TElement selectedElem = tabElems[offset];
        selectedIndex = offset;
        for (int Index = offset + 1; Index < numElems; Index++) {
            if (tabElems[Index] < selectedElem) {
                selectedElem = tabElems[Index];
                selectedIndex = Index;
            }
        }
        if (selectedIndex != offset) {
            TElement tempElem = tabElems[offset];
            tabElems[offset] = selectedElem;
            tabElems[selectedIndex] = tempElem;
        }
    }
}
```

ALGORITHMES DE TRI



Tri par insertion

- Méthode utilisée intuitivement par les humains : exemple des cartes à jouer où on insère chaque carte à sa place dans une *main* déjà triée.
- Facile à concevoir et à implémenter.
- Complexité $O(n^2)$ en moyenne, mais cette complexité dépend de l'arrangement initiale des données :
 - Si les données sont initialement triées à l'envers (pire des cas), il faut $n^2/2$ comparaisons et affectations,
 - Si les données sont distribuées aléatoirement, leurs permutations sont équiprobales ($n/2$) et il faut alors $n^2/4$ comparaisons et affectations,
 - Si les données sont déjà triées, il faut $n - 1$ comparaisons (complexité linéaire).
- Utilisé de façon brute, il est non parallélisable.

ALGORITHMES DE TRI

Tri par insertion : exemple

- 18 75 98 42 25 31 55 24 7 31 12 7
- 18 75 98 42 25 31 55 24 7 31 12 7
- 18 75 98 42 25 31 55 24 7 31 12 7
- 18 42 75 98 25 31 55 24 7 31 12 7
- 18 25 42 75 98 31 55 24 7 31 12 7
- 18 25 31 42 75 98 55 24 7 31 12 7
- 18 25 31 42 55 75 98 24 7 31 12 7
- 18 24 25 31 42 55 75 98 7 31 12 7
- 7 18 24 25 31 42 55 75 98 31 12 7
- 7 18 24 25 31 31 42 55 75 98 12 7
- 7 12 18 24 25 31 31 42 55 75 98 7
- 7 7 12 18 24 25 31 31 42 55 75 98
- 7 7 12 18 24 25 31 31 42 55 75 98

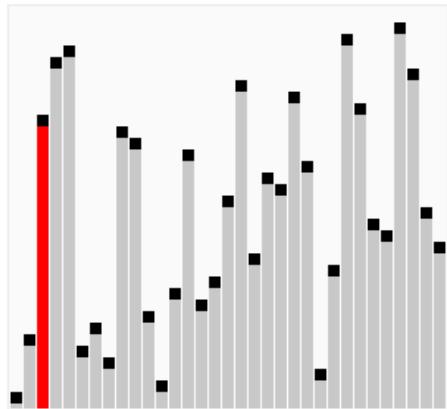
ALGORITHMES DE TRI

Tri par insertion d'un tableau : implémentation en C

```
void InsertionSort(TElement *tabElems, int numElems)
{
    for (int Index = 1; Index < numElems; Index++) {
        if (tabElems[Index-1] > tabElems[Index]) {
            TElement elementToInsert = tabElems[Index];
            int backIndex;
            for (backIndex = Index;
                 backIndex > 0 && (tabElems[backIndex-1] > elementToInsert);
                 backIndex--)
            {
                tabElems[backIndex] = tabElems[backIndex-1];
            }
            tabElems[backIndex] = elementToInsert;
        }
    }
}
```

ALGORITHMES DE TRI

Tri à bulles



- Appelé également *tri par propagation*, cette méthode consiste à parcourir la structure de données et à échanger systématiquement les éléments adjacents qui sont mal ordonnés. On répète l'opération tant que la structure n'est complètement triée, c'est-à-dire jusqu'à ce qu'il n'y ait plus d'éléments à échanger.
- Les éléments les plus grands migrent vers la droite (dans le cas d'un tri croissant) au fur et à mesure de la progression de l'algorithme, comme des bulles qui remontent à la surface d'un liquide, d'où le nom de *tri à bulles*.
- Simple dans son principe et facile à implémenter, mais algorithme lent. Intérêt purement pédagogique.
- Complexité algorithmique en $O(n^2)$ en moyenne, identique au tri par insertion

ALGORITHMES DE TRI

Tri à bulles : exemple

- 18 75 98 42 25 31 55 24 7 31 12 7
- 18 75 42 25 31 55 24 7 31 12 7 98
- 18 42 25 31 55 24 7 31 12 7 75 98
- 18 25 31 42 24 7 31 12 7 55 75 98
- 18 25 31 24 7 31 12 7 42 55 75 98
- 18 25 24 7 31 12 7 31 42 55 75 98
- 18 24 7 25 12 7 31 31 42 55 75 98
- 18 7 24 12 7 25 31 31 42 55 75 98
- 7 18 12 7 24 25 31 31 42 55 75 98
- 7 12 7 18 24 25 31 31 42 55 75 98
- 7 7 12 18 24 25 31 31 42 55 75 98
- 7 7 12 18 24 25 31 31 42 55 75 98

ALGORITHMES DE TRI

Tri à bulles sur un tableau : implémentation en C

```
void Bubblesort(TElement *tabElems, int numElems)
{
    bool sortDone = false;
    for (int subSize = numElems; subSize > 0 && !sortDone; subSize--)
    {
        sortDone = true;
        for (int Index = 1; Index < subSize; Index++) {
            if (tabElems[Index-1] > tabElems[Index]) {
                sortDone = false;
                TElement tempElem = tabElems[Index];
                tabElems[Index] = tabElems[Index-1];
                tabElems[Index-1] = tempElem;
            }
        }
    }
}
```

ALGORITHMES DE TRI

Tri à bulles : variantes

- Constat : les éléments les plus grands (*lièvres*) sont propagés rapidement vers la droite (dans le cas d'un tri croissant), alors que les plus petits (*tortues*) ne se déplacent que d'une position à la fois. Des variantes du tri à bulles peuvent améliorer cet état de fait.
- 1^{ère} variante : Tri cocktail (*Shaker sort*)
 - Inverse le sens du parcours à chaque itération. Le tri est légèrement plus rapide dans la majorité des cas mais la complexité reste en $O(n^2)$.
- 2^{ème} variante : Tri en peigne (*Combsort*)
 - Au lieu de comparer les éléments adjacents, on compare au début de l'algorithme les éléments séparés par un certain intervalle m . Cet intervalle se réduit jusqu'à 1 au cours des itérations successives.
 - L'amélioration est significative, cette méthode peut même s'avérer dans certaines configurations (grands volumes de données presque triées) plus rapide que le *Quick sort*.

ALGORITHMES DE TRI

Tri fusion

- Cet algorithme utilise le principe « *diviser pour régner* » qui consiste à diviser un problème en sous-problèmes plus simples à résoudre, puis à trouver une solution au problème initial à partir des sous-problèmes résolus.
- Principe de la fusion : à partir de deux collections de données (tableaux ou listes) triées, construire une collection triée comportant tous les éléments des deux collections initiales. Un simple parcours de gauche à droite des deux collections avec ajout à chaque itération de l'élément le plus petit de ces deux collections dans la nouvelle collection permet de réaliser cette fusion.
- La collection initiale est divisée récursivement par deux afin de produire des couples de sous-collections de taille $m = 1$. Dans la pratique et à des fins d'optimisation, cette taille minimale peut être > 1 (par exemple $m = 10$). Chaque sous-collection de taille m est alors triée par un algorithme classique efficace sur de petits volumes de données, comme le *tri par insertion*.
- Algorithme efficace, sa complexité algorithmique est $O(n \log(n))$.
- Il est par ailleurs *stable* et *non en place*.

ALGORITHMES DE TRI

Principe de la fusion (1/11)

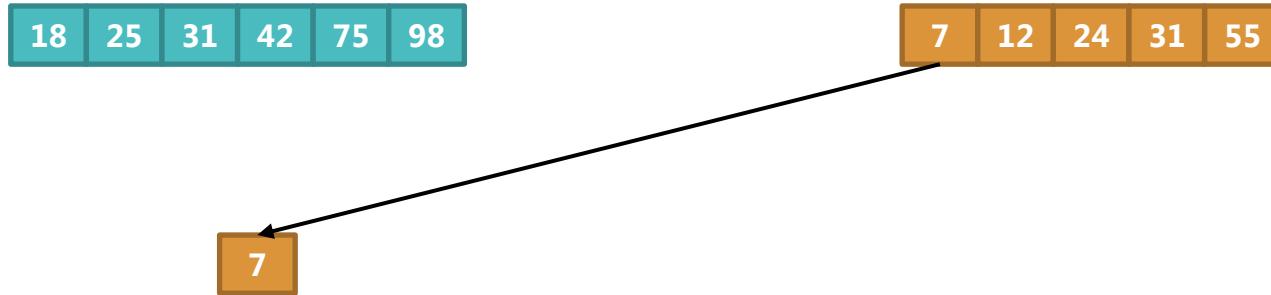
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (2/11)

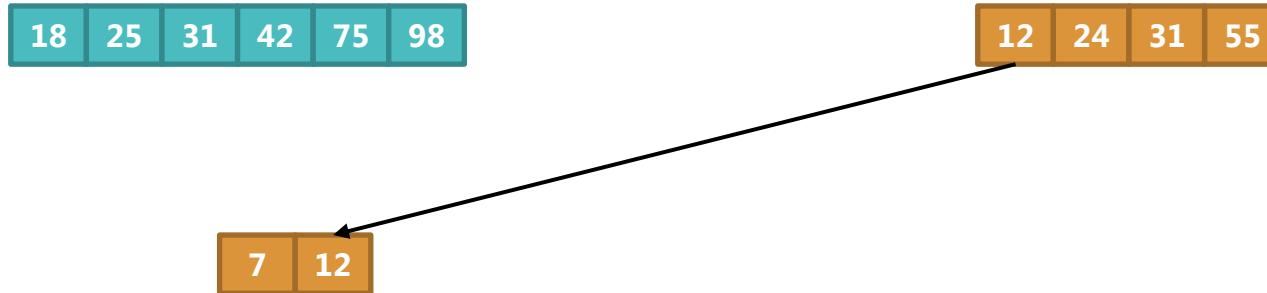
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (3/11)

- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (4/11)

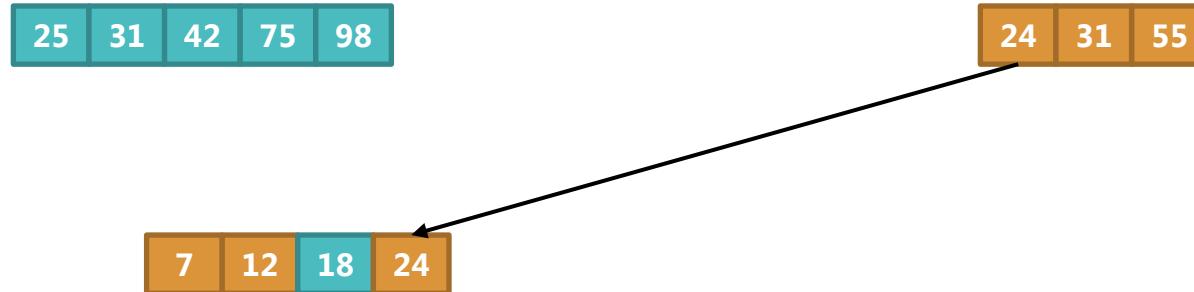
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (5/11)

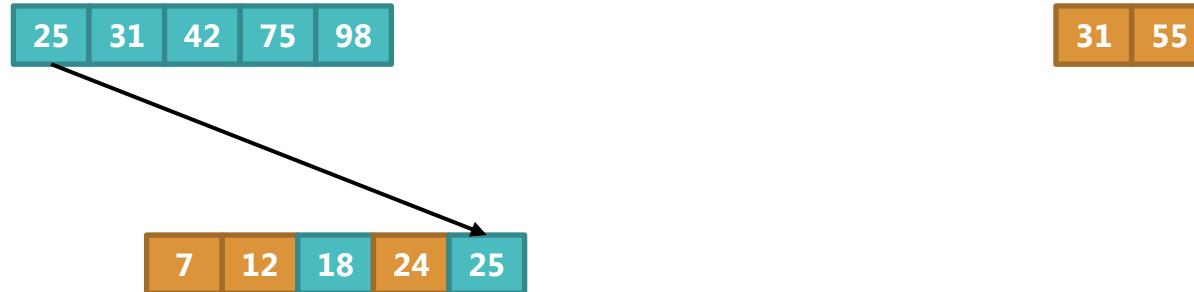
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (6/11)

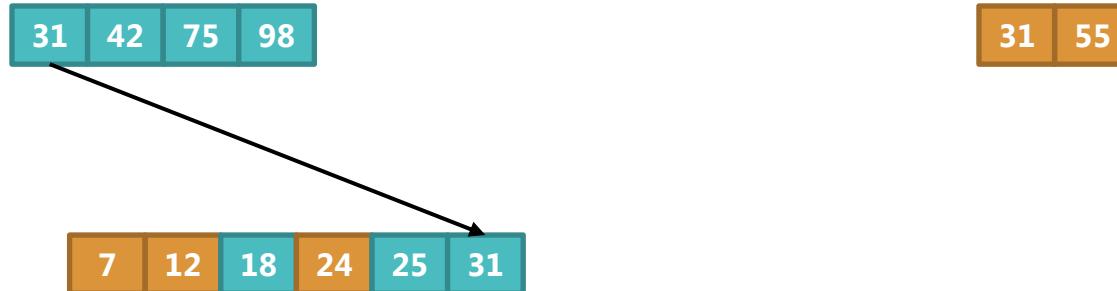
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (7/11)

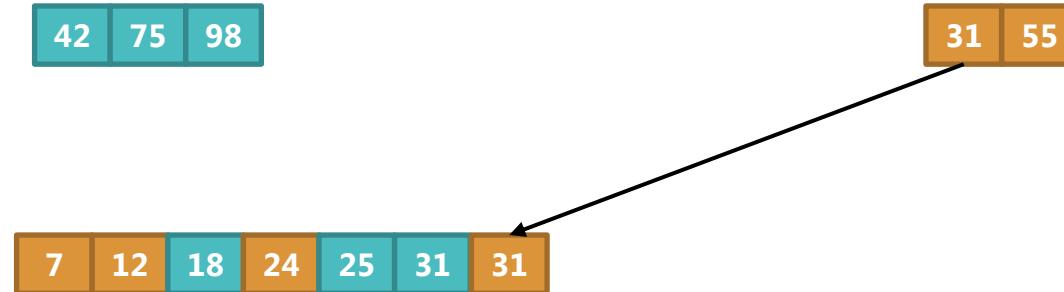
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (8/11)

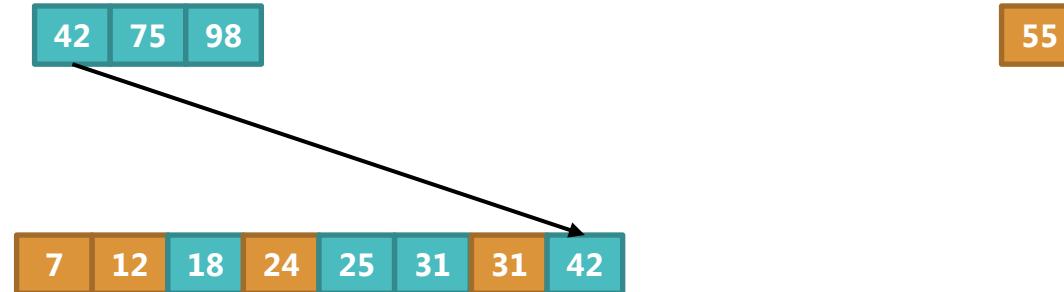
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (9/11)

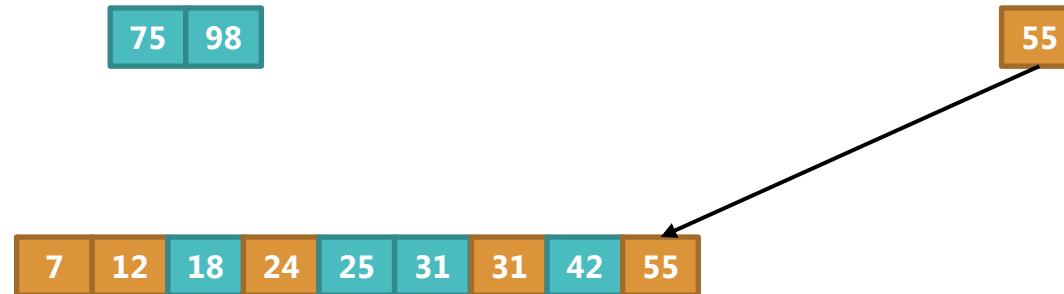
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Principe de la fusion (10/11)

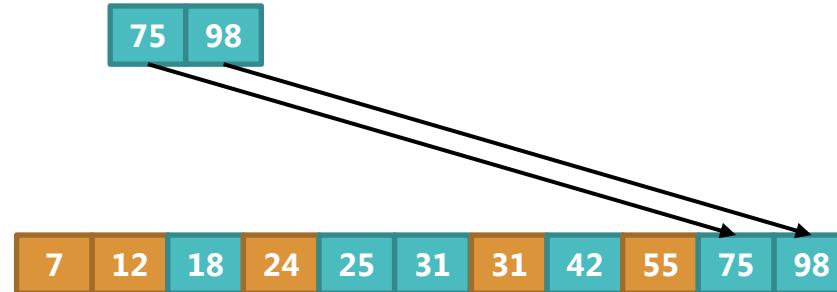
- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

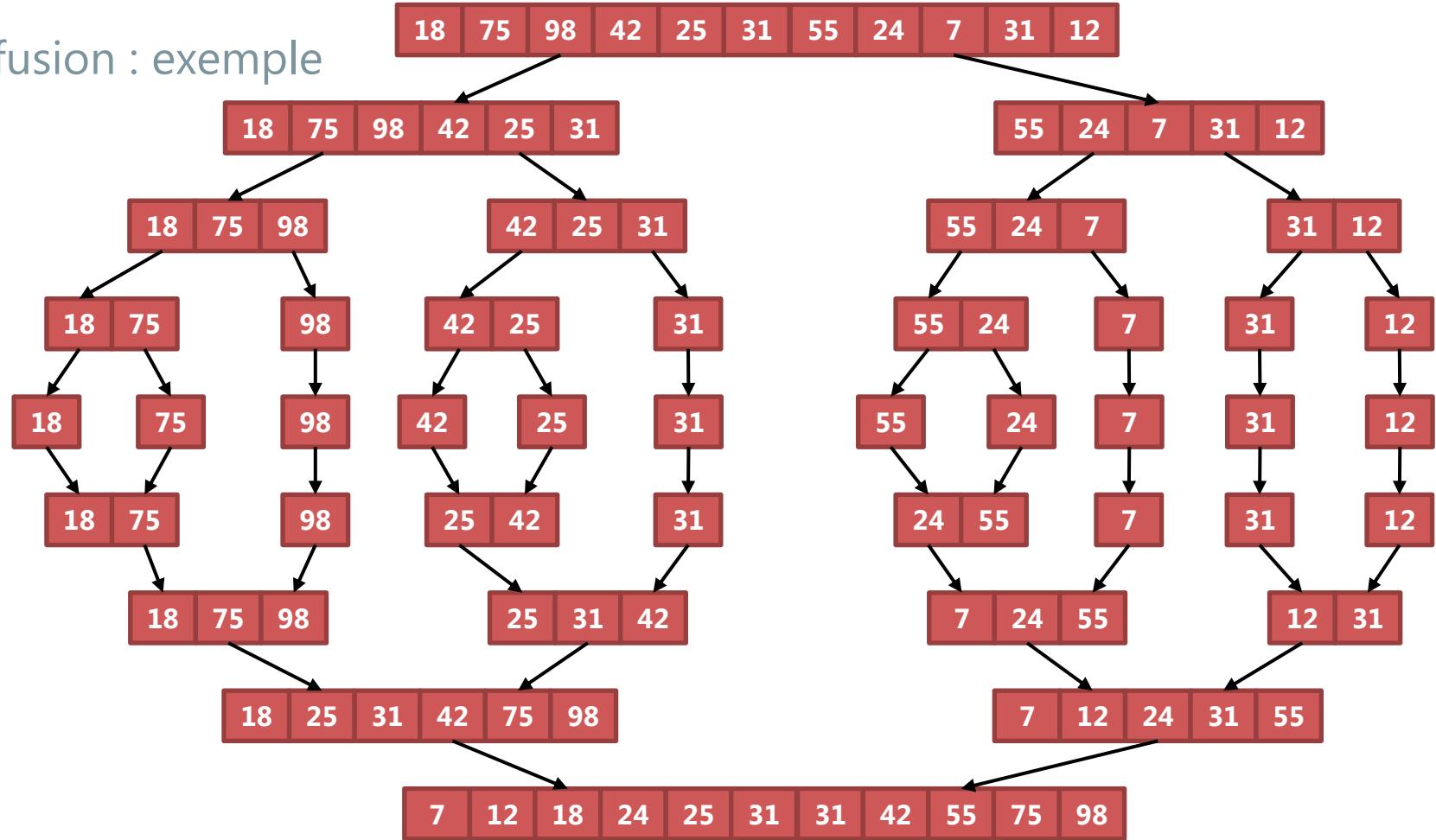
Principe de la fusion (11/11)

- Partant du principe que le premier élément de chaque collection est aussi le plus petit, on compare entre eux les deux premiers éléments de chacune de ces deux collections et on ajoute le plus petit à la collection triée.
- On itère ensuite jusqu'à épuisement de l'une des deux collections.



ALGORITHMES DE TRI

Tri fusion : exemple



ALGORITHMES DE TRI

Algorithme de fusion de deux tableaux triés : pseudo-code

```
tableau algo_fusion(tableau gauche, tableau droit, entier tailleG, entier tailleD)
début
    entier taille = tailleG + tailleD
    tableau fusion[taille]
    entier i, g = 0, d = 0
    pour i = 0 à taille-1 faire
        si gauche[g] > droit[d] alors
            fusion[i] = droit[d]; d = d + 1
        sinon
            fusion[i] = gauche[g]; g = g + 1
        fsi
        si (d == tailleD) alors
            tant_que g < tailleG
                i = i + 1; fusion[i] = gauche[g]; g = g + 1
            fin
        sinon si (g == tailleG) alors
            tant_que d < tailleD
                i = i + 1; fusion[i] = droit[d]; d = d + 1
            fin
        fsi
    fin
    renvoyer fusion
fin
```

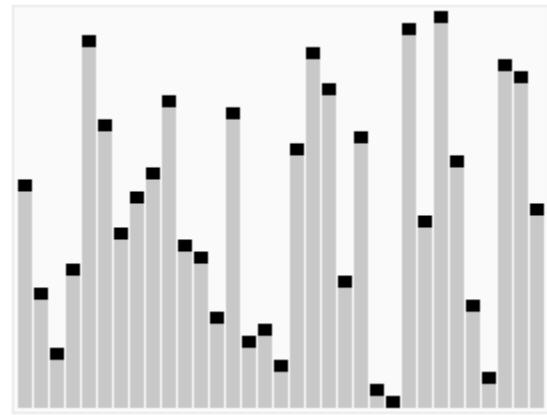
ALGORITHMES DE TRI

Tri fusion sur un tableau : pseudo-code récursif

```
procedure tri_fusion(tableau T, entier n, entier m)
debut
    si n <= m alors
        si n > 1 alors
            tri_insertion(T, n)
        fsi
    fsi
    entier tailleD = n/2
    entier tailleG = n - tailleD
    tableau droit = T + taille G
    tableau gauche = T
    si tailleD > 1 alors tri_fusion(droit, tailleD, m)
    si tailleG > 1 alors tri_fusion(gauche, tailleG, m)
    tableau fusion = algo_fusion(gauche, droit, tailleG, tailleD)
    recopier(T, fusion, n) // recopie dans T les n éléments triés
fin
```

ALGORITHMES DE TRI

Quick sort

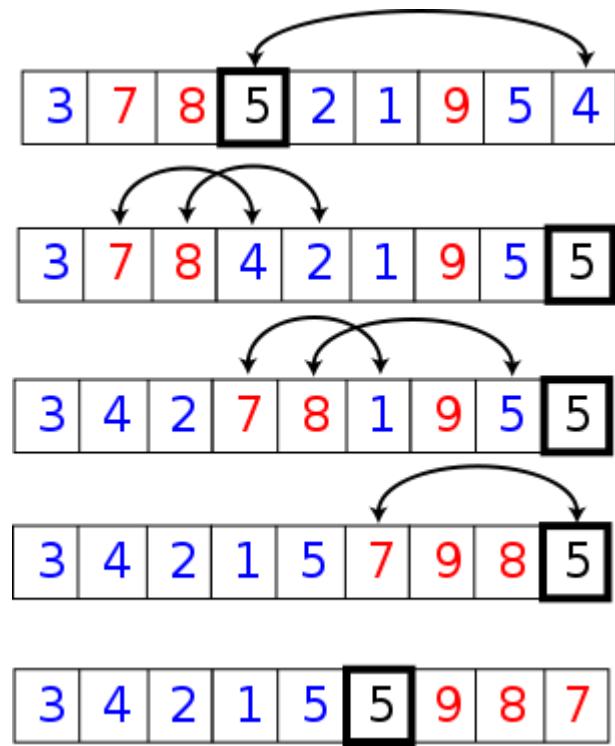


- Comme le *tri fusion*, c'est un algorithme basé sur le principe *diviser pour régner*.
- Comme le *tri fusion*, il se prête bien à une implémentation récursive.
- Tri *en place*, mais *instable*.
- Très efficace (un des plus rapides) sur des données rangées aléatoirement, sa complexité moyenne est en $O(n \log(n))$. Sur des données presque triées, il est par contre moins rapide que le *tri par insertion* qui tire avantage de ce fait. Si les données sont triées à l'envers, la méthode devient très lente et sa complexité devient $O(n^2)$.
- La fonction *qsort* est une fonction générique de la librairie standard du C qui implémente une version optimisée de cet algorithme pour trier des tableaux de n'importe quelle type.
- Prototype : **void qsort(void *base, size_t nelems, size_t size,
int (*compar)(const void *, const void *));**

ALGORITHMES DE TRI

Description du Quick sort

- Partitionnement : il s'agit de choisir un élément appelé *pivot* que l'on place à son emplacement définitif en plaçant tous les éléments inférieurs à sa gauche et tous les éléments supérieurs à sa droite.
- Cette opération est répétée récursivement : on choisit un nouveau pivot pour chaque sous-collection jusqu'à ce que le tableau initial soit entièrement trié.
- Concrètement, pour chaque sous-collection :
 - Echanger le pivot avec le dernier élément,
 - Placer tous les éléments inférieurs ou égaux au début,
 - Echanger le pivot avec le 1^{er} élément supérieur.



ALGORITHMES DE TRI

Quick sort : pseudo-code (version de base du partitionnement)

```
entier partition(tableau T, entier indice_debut, entier indice_fin)
debut
    entier temp, pivot = T[indice_fin]
    entier i = indice_debut - 1, j
    pour j = indice_debut à (indice_fin - 1) faire
        si T[j] <= pivot alors
            i = i + 1
            temp = T[i]; T[i] = T[j]; T[j] = temp
        fsi
    fin
    temp = T[i + 1]; T[i + 1] = T[indice_fin]; T[indice_fin] = temp
    renvoyer i + 1
fin

procedure quick_sort(tableau T, entier indice_debut, entier indice_fin)
debut
    entier pivot = partition(T, indice_debut, indice_fin)
    quick_sort(T, indice_debut, pivot - 1)
    quick_sort(T, pivot + 1, indice_fin)
fin
```

ALGORITHMES DE TRI

Quick sort : pseudo-code (version alternative plus rapide)

```

procedure quick_sort(tableau T, entier n)
début
  entier temp, pivot = T[n - 1], g = 0, d = n - 2
  tant_que g <= d
    tant_que g <= d et T[g] < pivot
      g = g + 1
    fin
    tant_que g <= d et T[d] >= pivot
      d = d - 1
    fin
    si g < d alors
      temp = T[g]; T[g] = T[d]; T[d] = temp
      g = g + 1; d = d - 1
    fsi
  fin
  T[n - 1] = T[g]
  T[g] = pivot
  si g > 1 alors quick_sort(T, g) fsi
  si n > g + 2 alors quick_sort(T + g + 1, n - g - 1) fsi
fin
  
```

ALGORITHMES DE TRI

Choix du pivot

- Arbitraire : on choisit systématiquement comme pivot le premier ou le dernier élément de la collection. La complexité moyenne est alors en $O(n\log(n))$, mais si la collection initiale est presque triée, on est dans le pire des cas avec une complexité en $O(n^2)$.
- Aléatoire : on choisit le pivot aléatoirement de manière uniforme parmi les éléments de la collection. La complexité moyenne est également en $O(n\log(n))$, dans le pire des cas on a une complexité en $O(n^2)$, mais l'algorithme s'écarte peu de la complexité moyenne.
- Optimal : on choisit un pivot égal à la valeur médiane de la collection. Ce choix permet d'atteindre une complexité en $O(n\log(n))$ même dans les cas défavorables, mais nécessite de calculer préalablement la médiane.

ALGORITHMES DE TRI

Autres algorithmes

- Tri de Shell : c'est une amélioration du tri par insertion basé sur deux constats :
 1. Le tri par insertion est efficace sur une liste presque triée,
 2. Il est inefficace dans le cas moyen car on ne déplace qu'un élément à la fois.Il consiste donc à effectuer un tri par insertion sur des éléments séparés d'un gap m puis de recommencer en réduisant m jusqu'à $m = 1$. Empiriquement les valeurs optimales pour m sont 701, 301, 132, 57, 23, 10, 4, 1.
- Le tri par tas ou *heapsort* est basé sur le fait que le tableau à trier peut être vu comme un arbre binaire possédant les propriétés d'un *tas*.
- *Introsort* (tri introspectif) : c'est une amélioration du quick sort, basé sur un compteur de récursion. Si la profondeur de récursion devient trop élevée, un autre algorithme comme le tri par tas est utilisé pour trier les sous-tableaux.

ALGORITHMES DE TRI

Comparaison récapitulative de l'efficacité des algorithmes de tri

Algorithme	Cas optimal	Cas moyen	Pire des cas	Stable	En place
Tri par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$	Non	Oui
Tri par insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Oui	Oui
Tri à bulles	$O(n)$	$O(n^2)$	$O(n^2)$	Oui	Oui
Combsort	$O(n)$	$O(n\log(n))$	$O(n^2)$	Non	Oui
Tri fusion	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	Oui	Non
Quick sort	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	Non	Oui

ALGORITHMES DE TRI

Résultats expérimentaux : temps en s sur des données aléatoires

Taille	Tri par sélection	Tri par insertion	Tri à bulles	Tri fusion	Tri rapide	Fonction qsort
100	0,000037	0,000019	0,000139	0,000012	0,000011	0,000009
200	0,000142	0,000111	0,000464	0,000059	0,000038	0,000023
400	0,000468	0,000220	0,001676	0,000049	0,000066	0,000047
800	0,001920	0,000971	0,007893	0,000115	0,000113	0,000091
1600	0,007560	0,003938	0,031460	0,000277	0,000247	0,000203
3200	0,024701	0,019259	0,117024	0,000441	0,000561	0,000359
6400	0,093677	0,062188	0,443486	0,001232	0,000991	0,000819
12800	0,364757	0,254169	1,807667	0,002208	0,002616	0,001782
25600	1,445812	1,009818	7,271801	0,005946	0,006223	0,003549
51200	5,733326	4,049834	28,642312	0,011675	0,013151	0,008255

ALGORITHMES DE TRI

Résultats expérimentaux : temps en s sur des données déjà triées

Taille	Tri par sélection	Tri par insertion	Tri à bulles	Tri fusion	Tri rapide	Fonction qsort
100	0,000022	0,000003	0,000002	0,000019	0,000004	0,000003
200	0,000081	0,000002	0,000001	0,000008	0,000006	0,000004
400	0,000316	0,000003	0,000003	0,000016	0,000013	0,000008
800	0,001310	0,000004	0,000004	0,000035	0,000028	0,000043
1600	0,005744	0,000008	0,000008	0,000080	0,000064	0,000040
3200	0,024715	0,000015	0,000015	0,000190	0,000163	0,000095
6400	0,092966	0,000027	0,000026	0,000360	0,000320	0,000138
12800	0,364839	0,000051	0,000052	0,000513	0,000683	0,000289
25600	1,452664	0,000102	0,000102	0,002289	0,001241	0,000615
51200	5,782145	0,000244	0,000237	0,003927	0,003310	0,002164

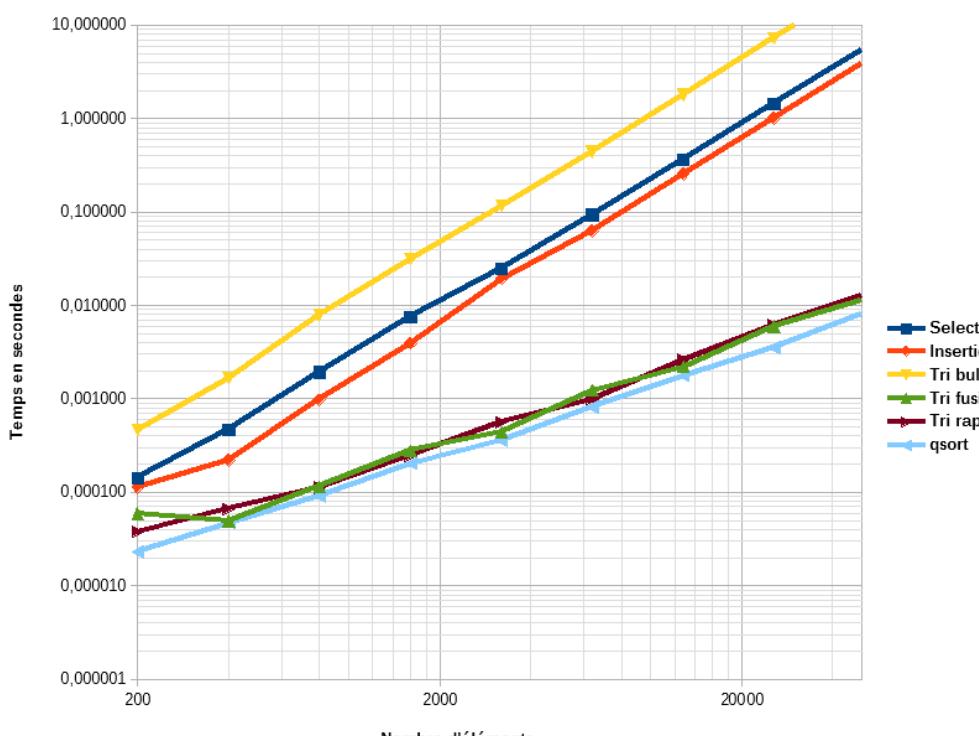
ALGORITHMES DE TRI

Résultats expérimentaux : temps en s sur des données triées à l'envers

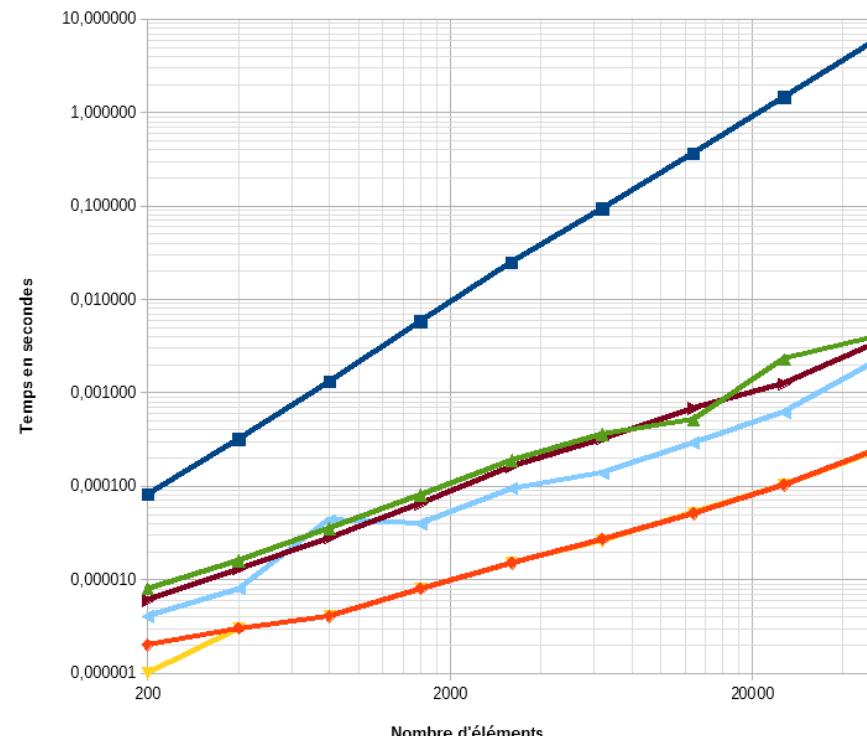
Taille	Tri par sélection	Tri par insertion	Tri à bulles	Tri fusion	Tri rapide	Fonction qsort
100	0,000025	0,000031	0,000165	0,000009	0,000006	0,000004
200	0,000110	0,000128	0,000570	0,000016	0,000010	0,000006
400	0,000375	0,000241	0,002861	0,000062	0,000042	0,000028
800	0,001370	0,001669	0,010450	0,000073	0,000044	0,000022
1600	0,005811	0,008692	0,040582	0,000157	0,000092	0,000064
3200	0,024411	0,033932	0,155037	0,000308	0,000240	0,000089
6400	0,097459	0,125908	0,616874	0,000692	0,000432	0,000201
12800	0,379662	0,507022	2,483861	0,001494	0,001038	0,000397
25600	1,584609	2,108329	9,875538	0,002572	0,002225	0,000954
51200	6,061650	8,264900	39,485193	0,006433	0,004273	0,002038

ALGORITHMES DE TRI

Données aléatoires vs données déjà triées



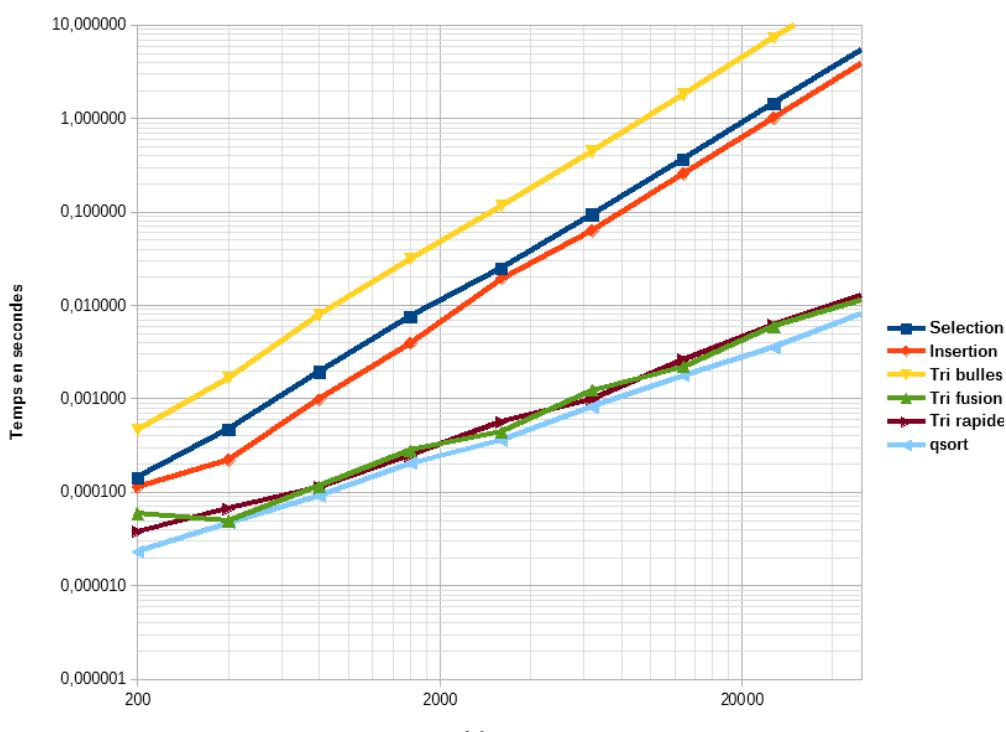
**Données aléatoires
(cas moyen)**



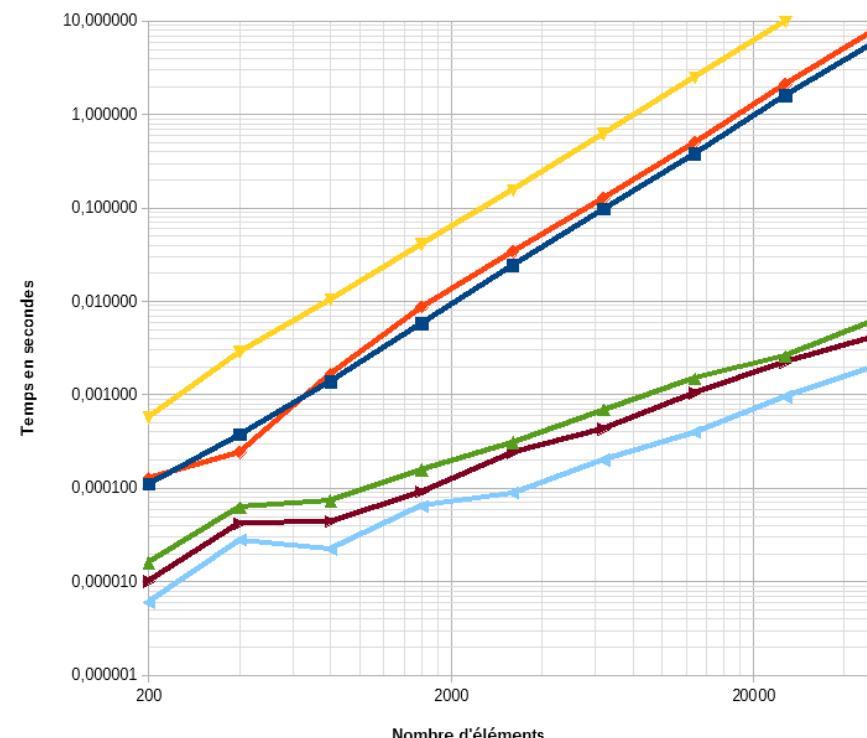
Données déjà triées

ALGORITHMES DE TRI

Données aléatoires vs données triées à l'envers



**Données aléatoires
(cas moyen)**



Données triées à l'envers

PLAN DU COURS

Méthodes de développement logiciel

Rappels et approfondissements en C

Types abstraits de données

Listes chainées

Piles et files

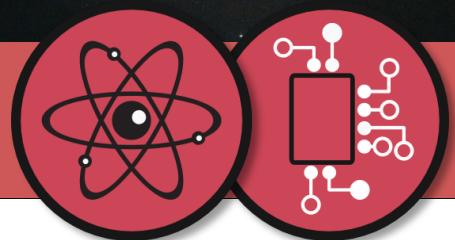
Algorithmes de tri

Tables de hachage

Arbres binaires



Tables de hachage



Tableaux associatifs - Listes d'association

Concept de table de hachage - Fonctions de hachage

Résolution des collisions - Opérations sur le TAD table de hachage

TABLES DE HACHAGE

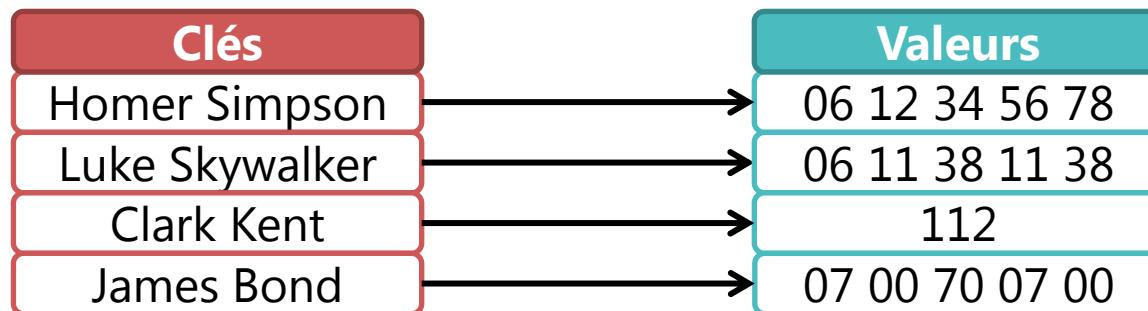
Motivations : tableaux vs listes chainées

- Les listes chaînées permettent d'insérer ou de supprimer facilement des éléments, mais la recherche d'un élément précis impose de parcourir la liste du début jusqu'à l'élément recherché : il n'est possible d'atteindre un élément donné que par recherche séquentielle.
- Les tableaux permettent l'accès direct à un élément mais manquent de souplesse : l'insertion ou la suppression d'un élément nécessite de nombreuses recopies. La recherche d'un élément précis dans un tableau trié peut s'opérer rapidement par recherche dichotomique, bien plus efficace que la recherche séquentielle. On peut accéder directement à un élément donné dans le tableau, mais uniquement par son adresse (indice entier dans le tableau).
- Pour palier à ces inconvénients, il est nécessaire de concevoir un TAD permettant à la fois d'insérer et de supprimer des éléments, et de rechercher rapidement un élément donné.

TABLES DE HACHAGE

Tableaux associatifs

- Un tableau associatif est un TAD permettant d'associer une clé à une valeur : un exemple courant est le répertoire des contacts dans un téléphone mobile.

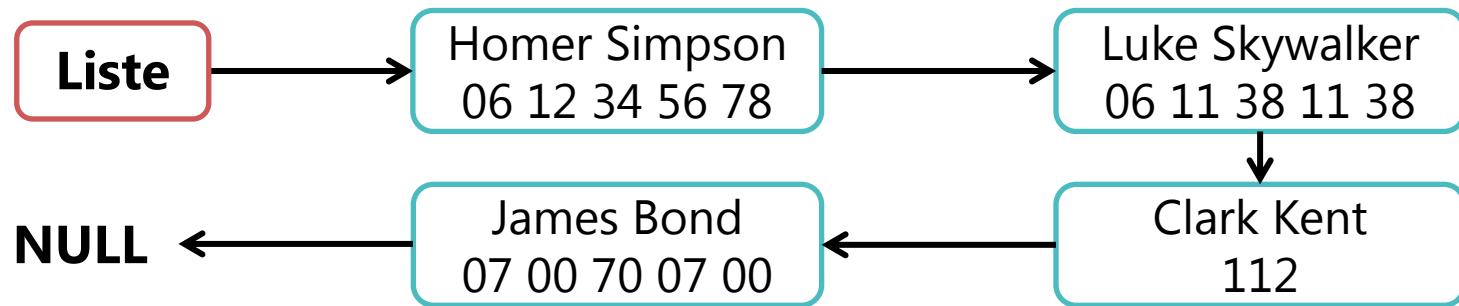


- Le tableau associatif est une généralisation de la notion de tableau traditionnel qui associe des entiers consécutifs (les indices) à des valeurs.
- Cependant, alors que les tableaux traditionnels permettent l'accès direct à un élément via son indice dans le tableau, il n'est pas possible d'accéder directement à un élément via une clé sous forme de chaîne de caractère (par exemple).

TABLES DE HACHAGE

Liste d'association

- Un tableau associatif peut être représenté sous forme d'une liste chaînée de couples (clé/valeur) :



- Ajout/suppression en $O(1)$, en tête de liste (cf. Listes chaînées).
- Pas de tri, ni de fonction spécialisée nécessaire sur les clés.
- La recherche de l'élément à partir de la clé se fait de manière séquentielle, la complexité de la recherche est donc en $O(n)$.
- Satisfaisant pour un petit nombre d'éléments, mais pas pour de grands volumes de données.

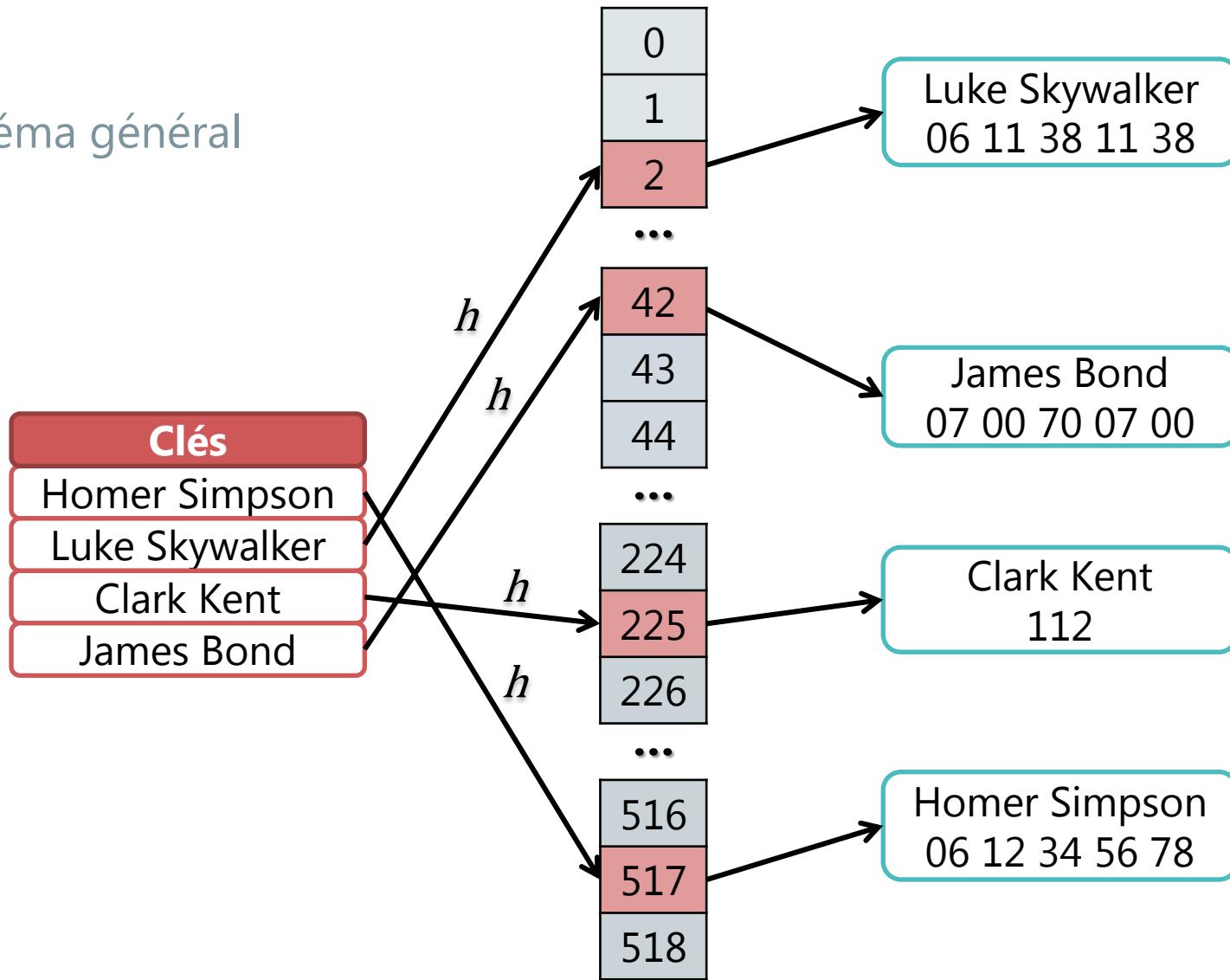
TABLES DE HACHAGE

Définitions

- Principe du *hachage* : convertir la clé en un nombre entier qui servira d'indice pour permettre un accès direct dans un tableau. Cette transformation se fait par le biais d'une *fonction de hachage* notée h . Le résultat est appelé *hash code*.
- Concrètement, une table de hachage est donc un tableau dont les éléments sont placés aux indices correspondant à leurs *hash code*.
- Contrairement à un tableau traditionnel, les éléments ne sont pas placés les uns à la suite des autres dans le tableau, mais sont autant que possible dispersés dans celui-ci de manière *uniforme* : les cases inoccupées sont appelées buckets, et les tableaux ainsi remplis sont désignés par le terme *bucket array*. Le tableau doit donc contenir plus de cases disponibles que d'éléments à stocker.
- Le rapport entre le nombre d'éléments stockés et le nombres de cases disponibles s'appelle *facteur de compression* ou *facteur de charge*. Afin d'optimiser la mémoire utilisée, les cases du tableau peuvent contenir des pointeurs vers les valeurs des éléments stockés au lieu des valeurs elles-mêmes.

TABLES DE HACHAGE

Schéma général



TABLES DE HACHAGE

Fonctions de hachage

- Une fonction de hachage $h : \{0,1\}^t \longrightarrow \{0,1\}^n$ est une fonction qui, à partir d'une donnée x de taille quelconque t en entrée, produit en sortie un résultat appelé *condensé* ou *empreinte* $y = h(x)$ de taille fixe n .
- Propriétés souhaitées :
 - 1) $\forall x_1, x_2 \in \{0,1\}^t, x_1 = x_2 \Rightarrow h(x_1) = h(x_2)$
 - 2) $\forall x_1, x_2 \in \{0,1\}^t, h(x_1) = h(x_2) \Rightarrow x_1 = x_2$
- La 1^{ère} propriété est triviale : deux clés identiques donnent le même *hash code* (heureusement !).
- Par contre la 2^{nde} propriété (injectivité) ne l'est pas car le cardinal de l'ensemble de départ est supérieur au cardinal de l'ensemble d'arrivée : il existe donc une possibilité de *collisions*.

TABLES DE HACHAGE

Paradoxe des anniversaires

- Qualifié de « paradoxe » car il contredit l'intuition, mais ce n'en est pas un.
- Problème posé : combien faut-il réunir de personnes dans une pièce pour avoir au moins une chance sur deux que deux de ces personnes aient leur anniversaire le même jour ?
- Réponse = 23



TABLES DE HACHAGE

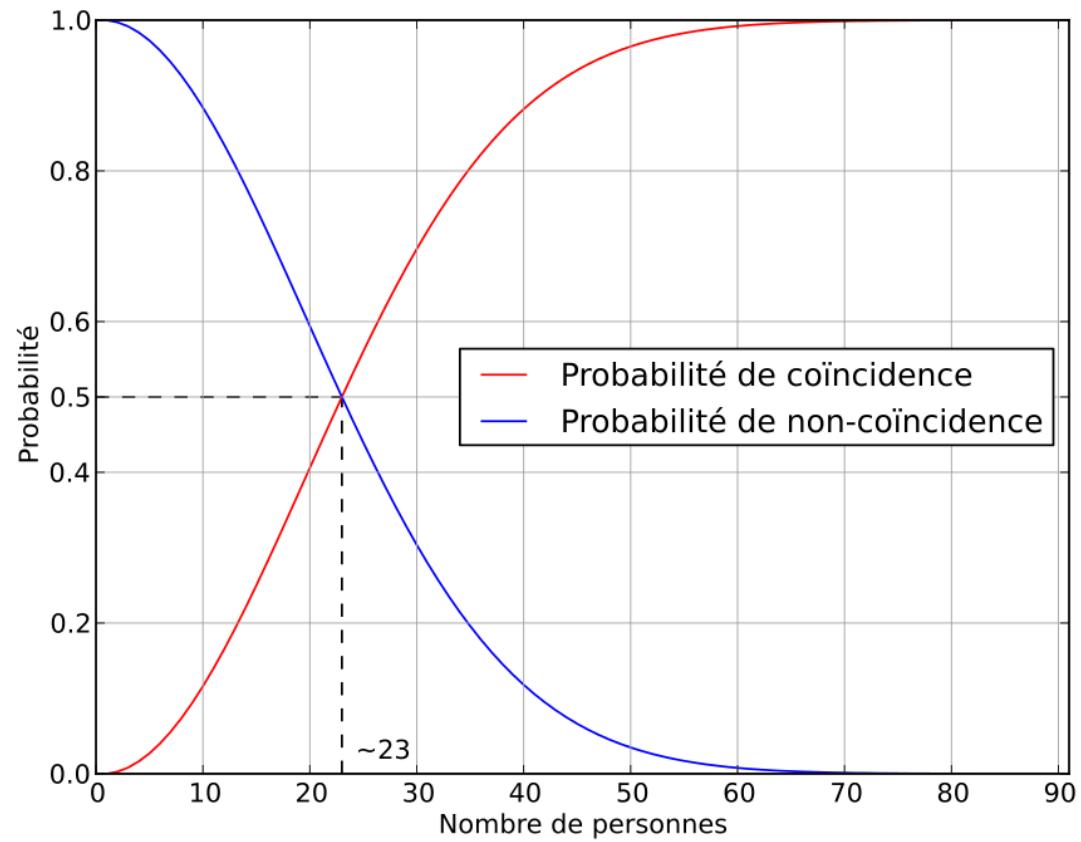
Probabilité de coïncidence

- Il y a k personnes dans la pièce : quelle est la probabilité \bar{p}_k de non-coïncidence, c'est-à-dire la probabilité pour que ces k personnes soient toutes nées un jour différent ? ($N = 365$ est le nombre de jours dans une année, on ne tient pas compte des années bissextiles)
- $k = 2 : \bar{p}_2 = \frac{N-1}{N}$
- $k = 3 : \bar{p}_3 = \frac{N-1}{N} \times \frac{N-2}{N}$
-
- pour k quelconque on trouve : $\bar{p}_k = \frac{N!}{(N-k)!N^k}$
- d'où la probabilité de coïncidence : $p_k = 1 - \bar{p}_k = 1 - \frac{N!}{(N-k)!N^k}$

TABLES DE HACHAGE

Probabilité de coïncidence

- $k = 5 : p_k = 2,71 \%$
- $k = 10 : p_k = 11,69 \%$
- **$k = 23 : p_k = 50,73 \%$**
- $k = 40 : p_k = 89,12 \%$
- Avec 60 personnes dans la pièce, la probabilité de coïncidence est de 99,41 % !



TABLES DE HACHAGE

Collisions

- On montre par le calcul que si une fonction de hachage produit un condensé de n bits, il est possible de trouver une collision avec 86% de probabilité avec $2^{(1+n/2)}$ valeurs.
- Exemple : on dispose d'une table de hachage de 1024 entrées, la fonction de hachage produit donc un condensé de 10 bits. On peut donc estimer qu'on aura une probabilité de collision de 86% avec seulement $2^{(1+10/2)} = 2^6 = 64$ valeurs !
- On doit donc choisir une fonction de hachage minimisant le risque de collision. Les *fonctions de hachage cryptographiques* ont cette propriété de résistance aux collisions, mais ne sont pas applicables dans le contexte des tables de hachage.
- Les collisions étant inévitables à partir d'un certain *facteur de charge*, il est nécessaire de mettre en place des méthodes de *Résolution des collisions*.

TABLES DE HACHAGE

Résolution des collisions

- *Chaînage* : au lieu d'une valeur unique, la case pointe vers une liste chaînée (une *liste d'association*) que l'on parcourt jusqu'à trouver le bon couple clé/valeur. Augmente le temps de recherche en cas de collisions multiples.
- *Adressage ouvert* : en cas de collision, consiste à stocker les valeurs concurrentes dans des cases contiguës. La position de ces cases est déterminée par une méthode de *sondage* :
 - *Sondage linéaire* : consiste à placer la donnée dans la première case libre qui suit la case déterminée par le hachage. Sensible à l'effet de *clustering*.
 - *Sondage quadratique* : semblable à la méthode précédente, mais l'intervalle entre les cases successives augmente linéairement pour éviter le *clustering*.
 - *Double hachage* : l'adresse de la case est donnée par une seconde fonction de hachage, différente de la première.

TABLES DE HACHAGE

Choix d'une fonction de hachage

- Pour être utilisable dans le contexte d'une table de hachage, une fonction de hachage doit posséder les propriétés suivantes :
 - Rapide à calculer pour ne pas pénaliser le temps de recherche,
 - Sortir une empreinte de taille raisonnable afin de limiter la taille de la table,
 - Permettre une répartition uniforme des empreintes afin d'éviter l'effet de *clustering* dans la table.
 - Avoir de bonnes *propriétés de diffusion* : une modification mineure de la clé entraîne une modification importante de l'empreinte en sortie (*effet d'avalanche*).
 - Limiter les collisions.
- Exemple de fonction de hachage :
 - Calculer la somme des valeurs ASCII des caractères de la clé, pondéré en fonction de la position des caractères :
$$h(k) = [k_0B^{n-1} + k_1B^{n-2} + \dots + k_{n-2}B + k_{n-1}] \bmod N$$
 - N est la taille de la table, on choisit un nombre premier pour éviter les diviseurs communs, ce qui permet de minimiser les collisions.
 - B est une puissance de 2.

TABLES DE HACHAGE

Fonctions de hachage cryptographiques (pour info uniquement)

- Ces fonctions produisent des empreintes de taille importante (256 bits recommandés) et sont utilisées pour :
 - Contrôle d'intégrité des transmissions de données (empreinte MD5 128 bits),
 - Codes d'authentification de message (MAC ou HMAC),
 - Signatures électroniques,
 - Stockage des mots de passes.
- Exemples : MD5 (128 bits, obsolète), SHA-1 (160 bits, obsolète), SHA-2 (224, 256, 384, 512 bits), SHA-3 (224, 256, 384, 512 bits, nouveau standard), Whirlpool (512 bits)
- Propriétés :
 - Résistance à la première pré-image,
 - Résistance à la seconde pré-image,
 - Résistance aux collisions.
- Malgré leurs bonnes propriétés, ces fonctions sont *impropres* à l'utilisation dans les tables de hachage en raison de la taille trop importante de leurs empreintes.

TABLES DE HACHAGE

Opérations sur le TAD table de hachage

- **Nom :** THashTable,
- **Utilise :** int, bool, TKey, TElement
- **Opérations :**
 - New → THashTable
 - IsEmpty : THashTable → bool
 - Size : THashTable → int
 - Put : THashTable × TKey × TElement → bool
 - Remove : THashTable × TKey → bool
 - Get : THashTable × TKey → TElement
 - Contains : THashTable × TKey → bool
 - Clear : THashTable →
 - Delete : THashTable →

PLAN DU COURS

Méthodes de développement logiciel

Rappels et approfondissements en C

Types abstraits de données

Listes chainées

Piles et files

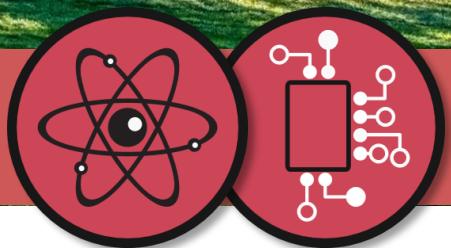
Algorithmes de tri

Tables de hachage

Arbres binaires



Arbres binaires



Notions d'arbre, arbre binaire, définitions

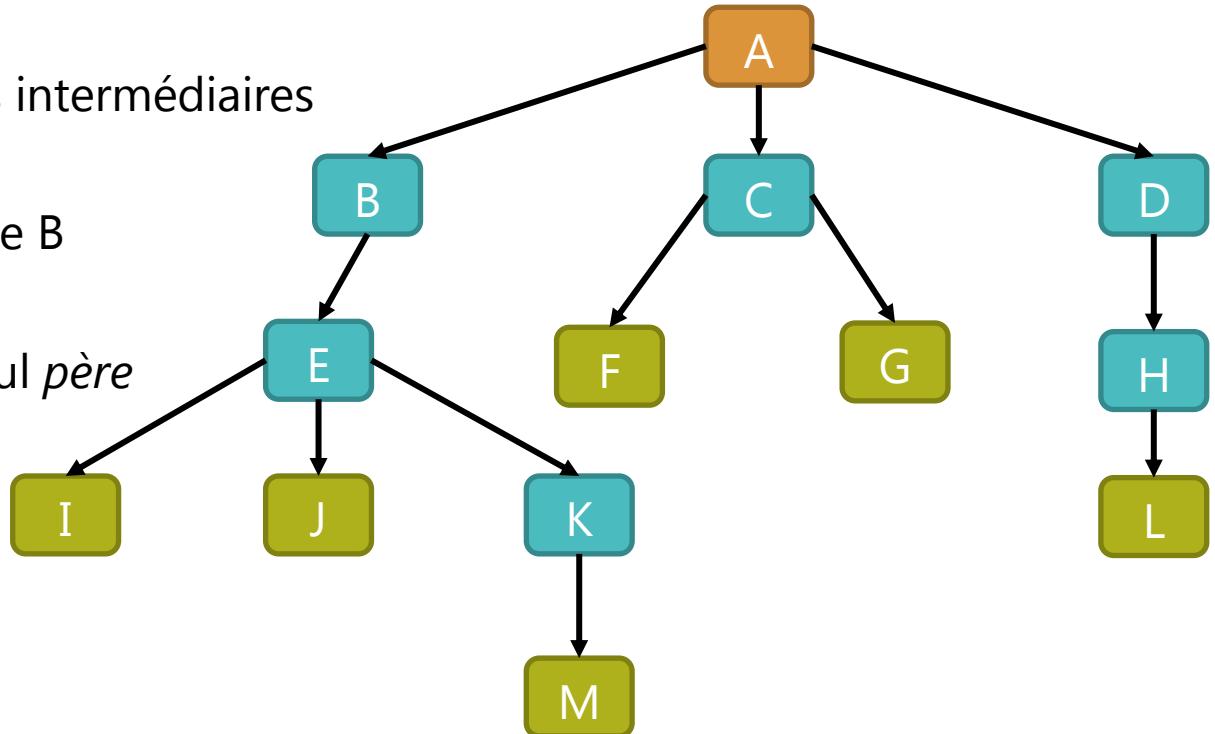
Parcours d'arbre - Arbres binaires de recherche - Arbres AVL

Tas et tri par tas

ARBRES BINAIRES

Arbre

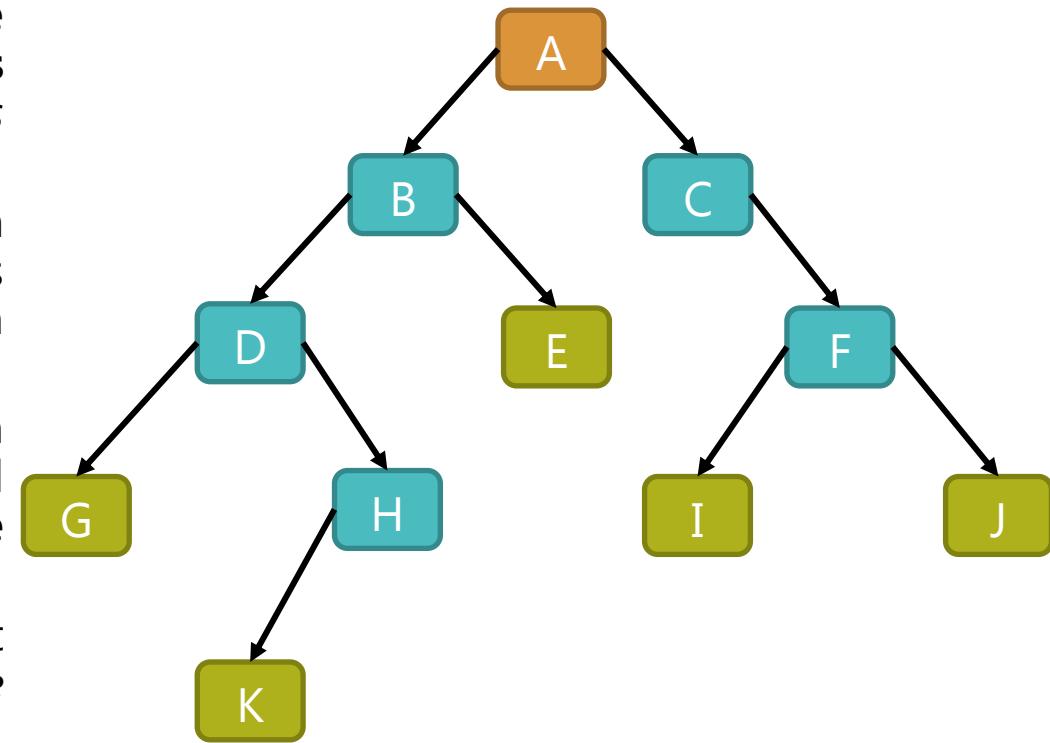
- Un *arbre* est une structure de données composée de *nœuds* (comme une liste chaînée) et de *feuilles* qui sont les nœuds terminaux.
- A : racine
- B, C, D, E, H, K : nœuds intermédiaires
- F, G, I, J, L, M : feuilles
- M est un *descendant* de B
- J est un *fils* de E
- Un nœud n'a qu'un seul *père*



ARBRES BINAIRES

Arbre binaire

- Un *arbre binaire* est un arbre dont tous les nœuds ont **au plus** deux fils : un *fils gauche* et un *fils droit*.
- Un *arbre binaire entier* est un arbre binaire dont tous les nœuds ont soit deux, soit aucun fils.
- Un *arbre binaire parfait* est un arbre binaire entier dans lequel toutes les feuilles sont à la même hauteur.
- Remarque : une liste simplement chaînée est un *arbre unaire* (chaque nœud a un fils unique).



ARBRES BINAIRES

Définitions

- *Sous-arbre direct de l'arbre* : arbre ayant pour racine le fils gauche ou le fils droit de la racine.
- *Profondeur d'un nœud* : longueur du chemin de la racine au nœud.
- *Niveau* : ensemble des nœuds de même profondeur.
- *Hauteur de l'arbre* : distance pour aller de la racine à la feuille la plus profonde.
- *Hauteur d'un nœud* : longueur du chemin du nœud à sa feuille descendante la plus profonde.
- *Taille de l'arbre* : nombre de nœuds de l'arbre, y compris la racine.
- *Taille d'un nœud* : nombre de descendants du nœud + 1 pour le nœud lui-même.

ARBRES BINAIRES

Implémentation à base de pointeurs

- Chaque nœud contient :
 - une instance du type de base,
 - un pointeur vers le fils gauche,
 - un pointeur vers le fils droit.
- L'arbre est décrit par une structure contenant :
 - la taille de l'arbre,
 - un pointeur vers la racine,
 - un pointeur vers le nœud courant.

```

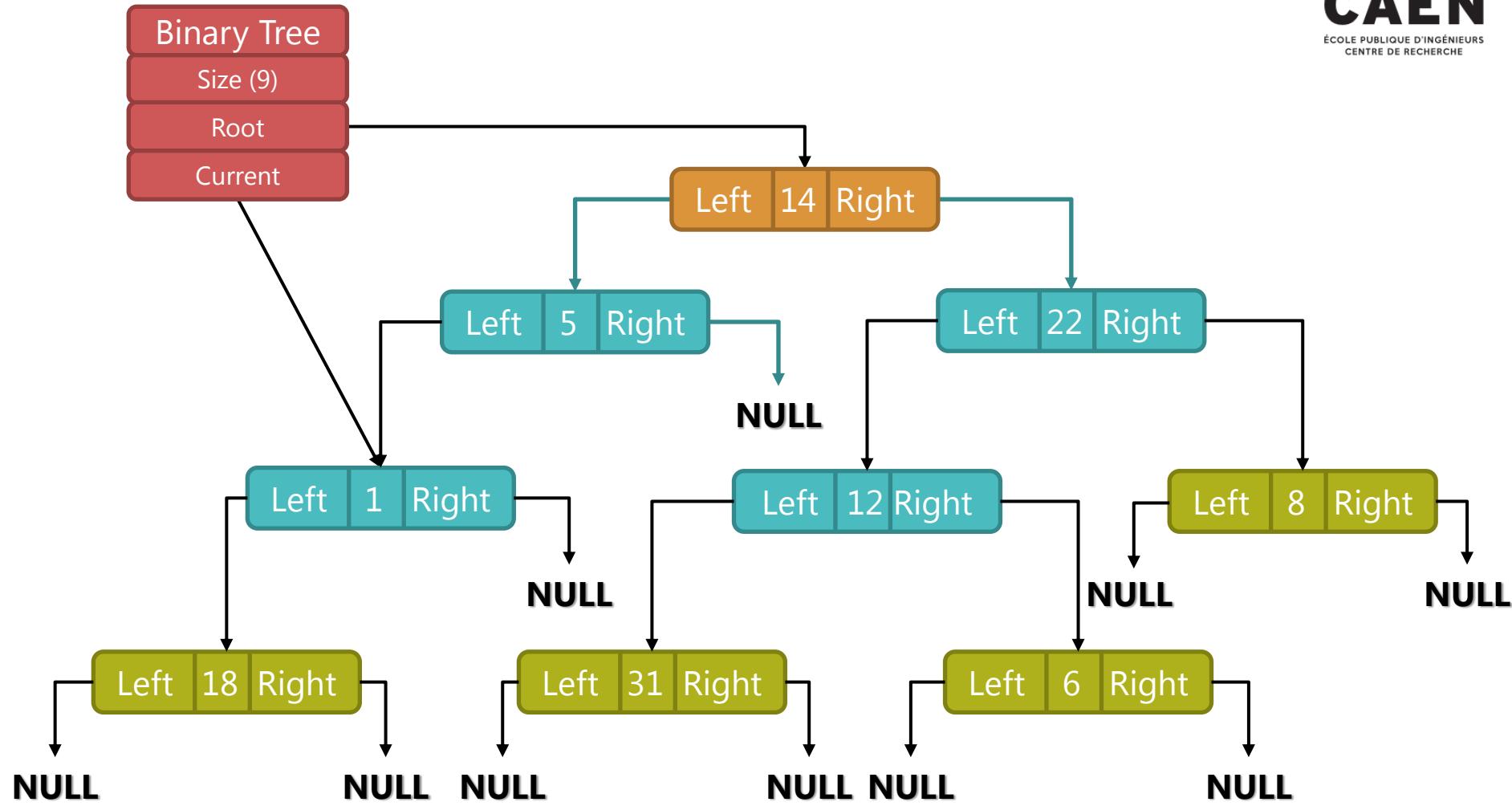
typedef struct Node {
    TElement Key;
    struct Node *LeftChild;
    struct Node *RightChild;
} TNode;

typedef TNode *PTNode;

typedef struct BinaryTree {
    int Size;          /* Nombre de noeuds */
    PTNode Root;       /* Racine */
    PTNode Current;   /* Courant */
} TBinaryTree;

typedef TBinaryTree *PTBinaryTree;
  
```

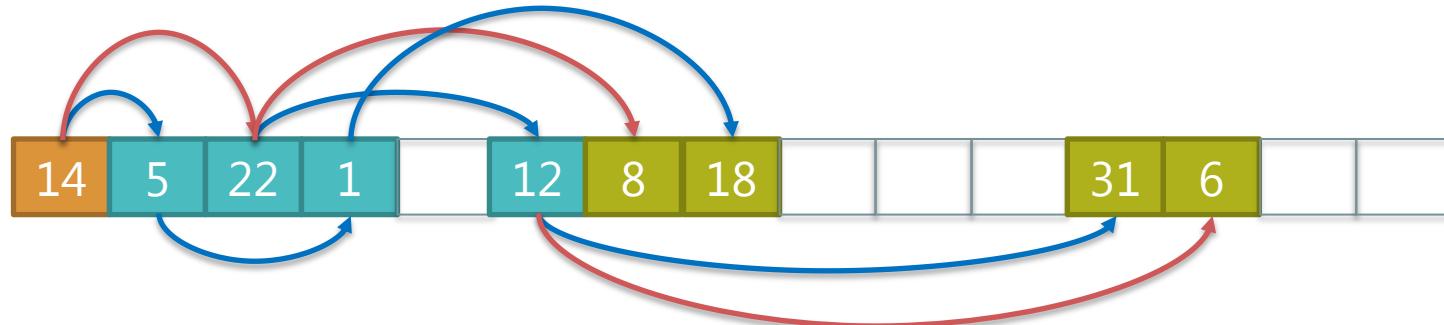
ARBRES BINAIRES



ARBRES BINAIRES

Implémentation à base de tableau

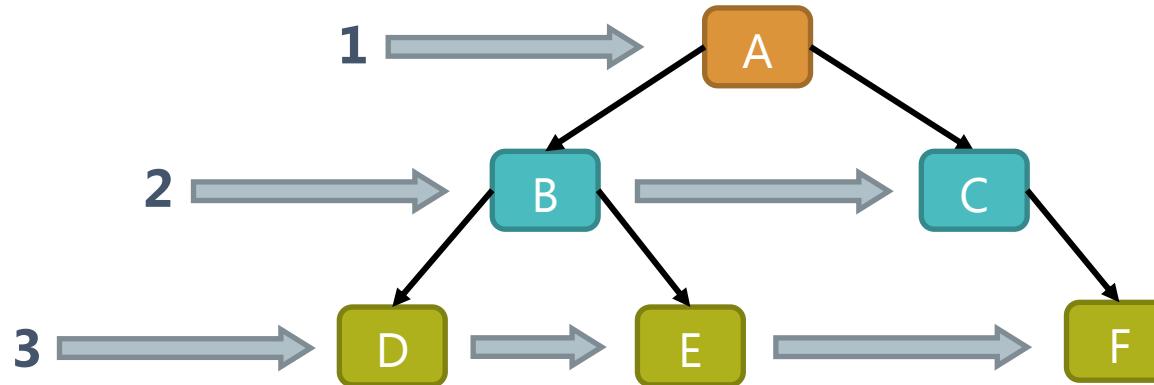
- Les nœuds de l'arbre (en fait leurs contenus) se trouvent dans les cases du tableau.
- Soit le nœud d'indice i :
 - Son fils gauche se trouve à l'indice $2i + 1$
 - Son fils droit se trouve à l'indice $2i + 2$
- Intéressant pour les arbres binaires complets car toutes les cases sont remplies, mais peut générer un gaspillage de mémoire dans le cas contraire.



ARBRES BINAIRES

Parcours en largeur (*Breadth-first search*)

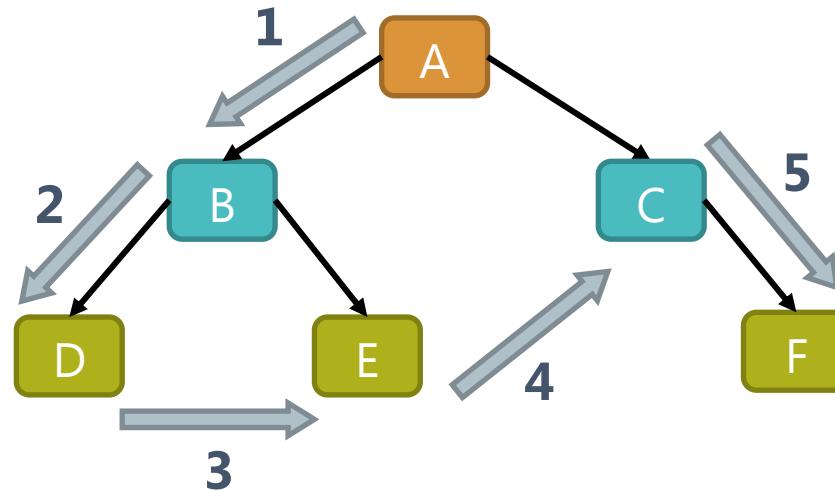
- Chaque niveau de l'arbre est visité hiérarchiquement de gauche à droite.
(parcours ABCDEF sur la figure)



ARBRES BINAIRES

Parcours en profondeur (*Depth-first search*)

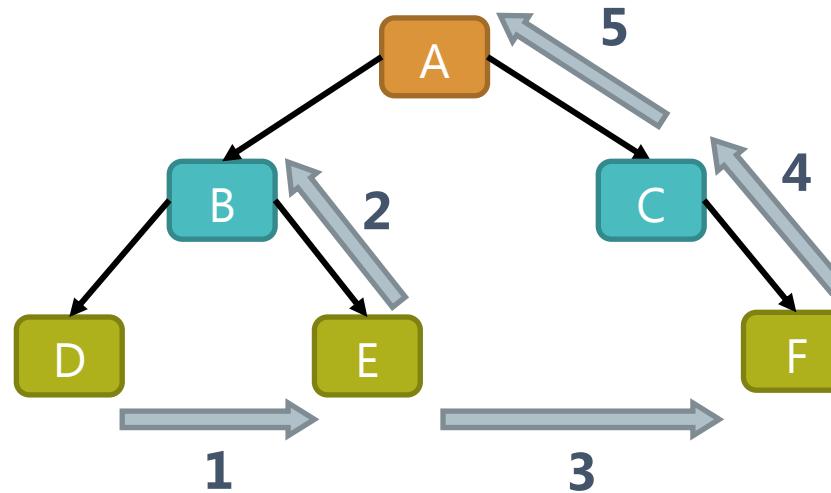
- Préfixé (*Pre-order*) : chaque nœud est visité ainsi que chacun de ses fils : gauche, puis droit : ABDECF



ARBRES BINAIRES

Parcours en profondeur (*Depth-first search*)

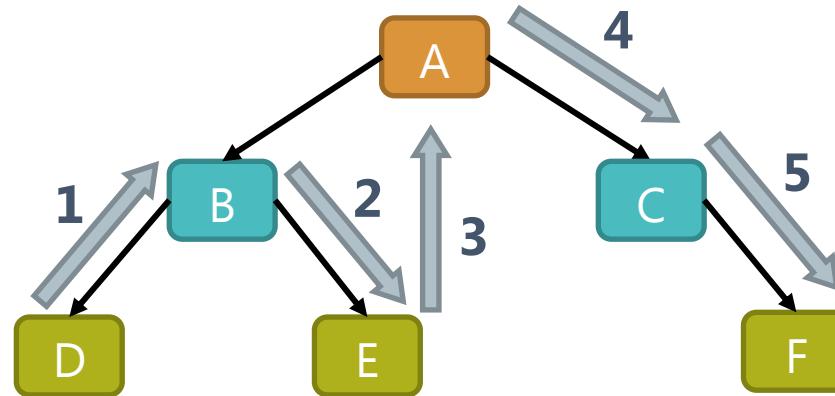
- Suffixé (*Post-order*) : chaque nœud est visité après avoir visité chacun de ses fils : gauche, puis droit : DEBFCA



ARBRES BINAIRES

Parcours en profondeur (*Depth-first search*)

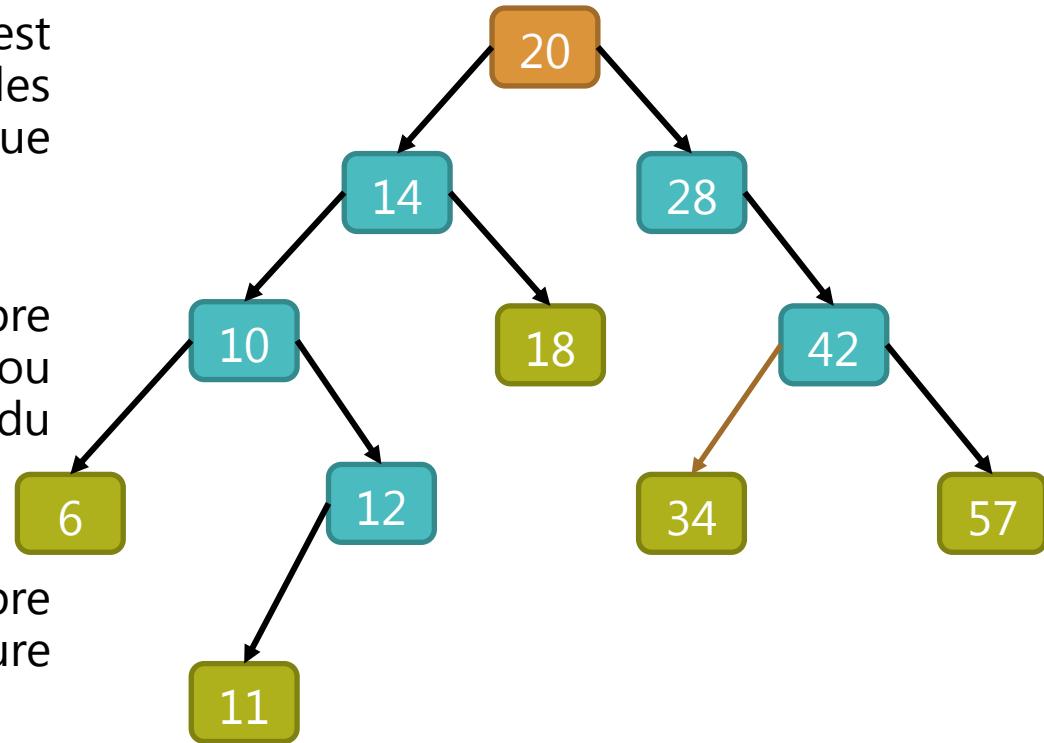
- Infixé (*In-order*) : visite chaque nœud entre les nœuds de son sous-arbre gauche et les nœuds de son sous-arbre droit : DBEACF



ARBRES BINAIRES

Arbre binaire de recherche (ABR)

- Un *arbre binaire de recherche* est un arbre binaire qui a les propriétés suivantes pour chaque nœud :
 - Chaque nœud du sous-arbre gauche a une valeur (*clé* ou *étiquette*) inférieure à celle du nœud considéré.
 - Chaque nœud du sous-arbre droit a une valeur supérieure à celle du nœud considéré.



ARBRES BINAIRES

Propriétés des arbres binaires de recherche

- L'insertion/suppression d'un élément se fait de manière efficace et préserve la relation d'ordre entre les éléments.
- Un *parcours en profondeur infixé* de l'arbre permet de visiter tous les nœuds dans l'ordre croissant des clés.
- La recherche d'un élément donné se trouve simplifiée, avec une complexité en $O(\log(n))$ dans le cas moyen, mais peut tendre vers $O(n)$ si l'arbre n'est pas équilibré.
- Afin de préserver cette efficacité, il peut être nécessaire de rééquilibrer l'arbre après plusieurs insertions et/ou suppressions.

ARBRES BINAIRES

Insertion des éléments dans un ABR (1/12)

- Eléments à insérer : 20, 14, 28, 10, 6, 18, 42, 34, 12, 11, 57

ARBRES BINAIRES

Insertion des éléments dans un ABR (2/12)

- Eléments restant à insérer : 14, 28, 10, 6, 18, 42, 34, 12, 11, 57

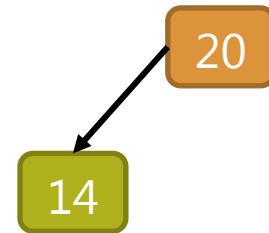
20

- Le premier élément est inséré en tant que racine de l'arbre

ARBRES BINAIRES

Insertion des éléments dans un ABR (3/12)

- Eléments restant à insérer : 28, 10, 6, 18, 42, 34, 12, 11, 57

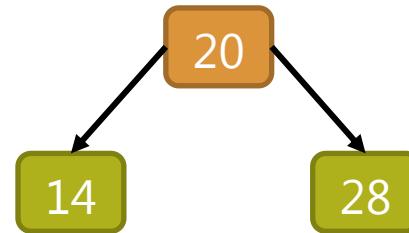


- $14 < 20 \rightarrow$ inséré à gauche

ARBRES BINAIRES

Insertion des éléments dans un ABR (4/12)

- Eléments restant à insérer : 10, 6, 18, 42, 34, 12, 11, 57

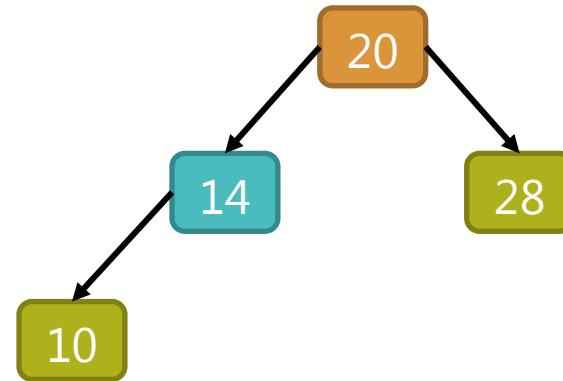


- $28 > 20 \rightarrow$ inséré à droite

ARBRES BINAIRES

Insertion des éléments dans un ABR (5/12)

- Eléments restant à insérer : 6, 18, 42, 34, 12, 11, 57

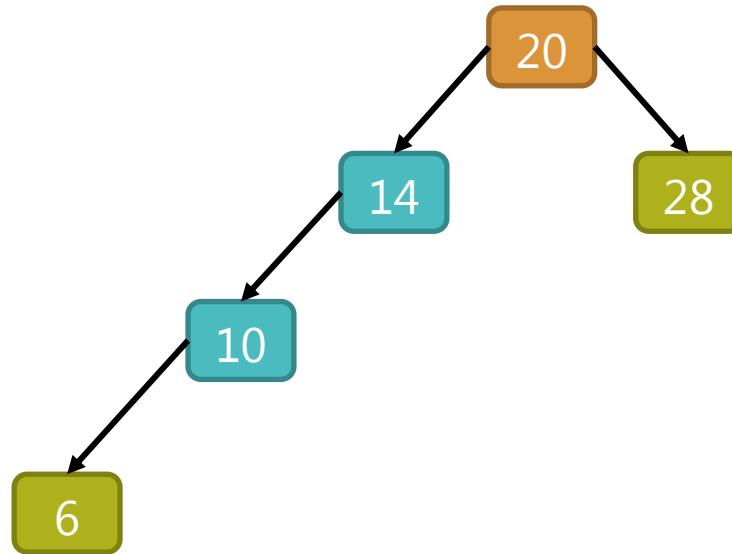


- $10 < 20 \rightarrow$ à gauche ; $10 < 14 \rightarrow$ inséré à gauche

ARBRES BINAIRES

Insertion des éléments dans un ABR (6/12)

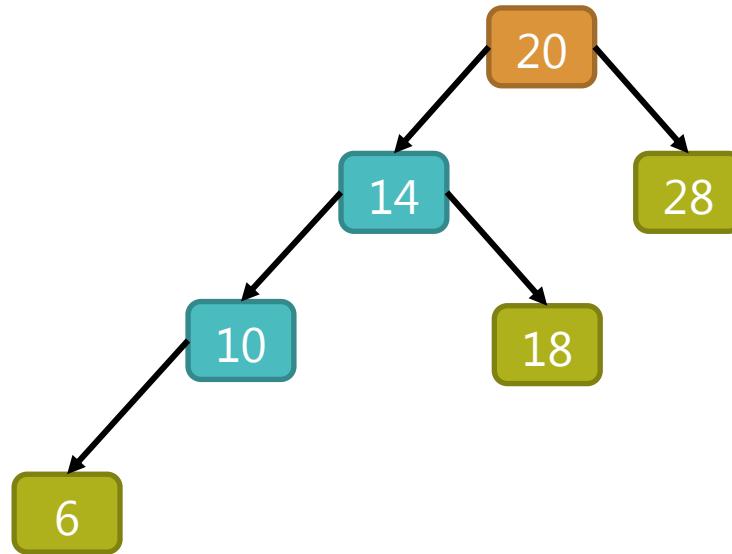
- Eléments restant à insérer : 18, 42, 34, 12, 11, 57



ARBRES BINAIRES

Insertion des éléments dans un ABR (7/12)

- Eléments restant à insérer : 42, 34, 12, 11, 57

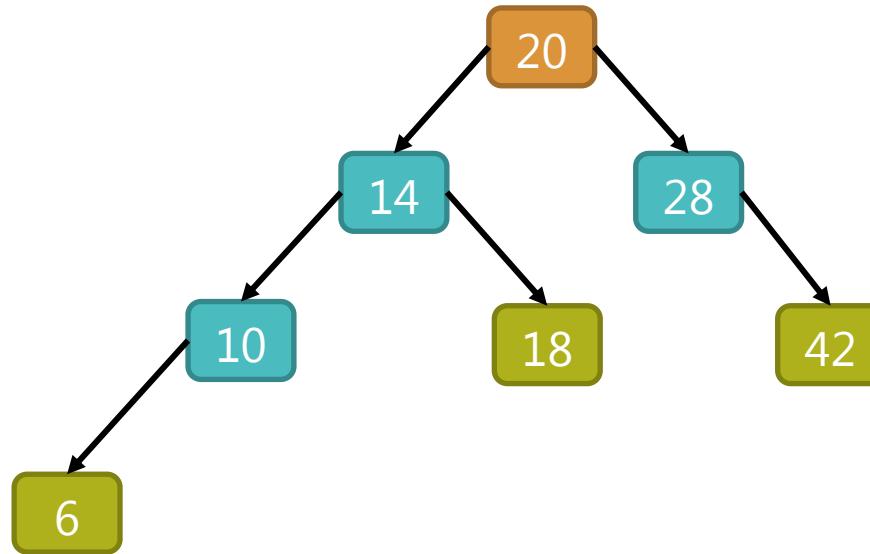


- $18 < 20 \rightarrow$ à gauche ; $18 > 14 \rightarrow$ inséré à gauche

ARBRES BINAIRES

Insertion des éléments dans un ABR (8/12)

- Eléments restant à insérer : 34, 12, 11, 57

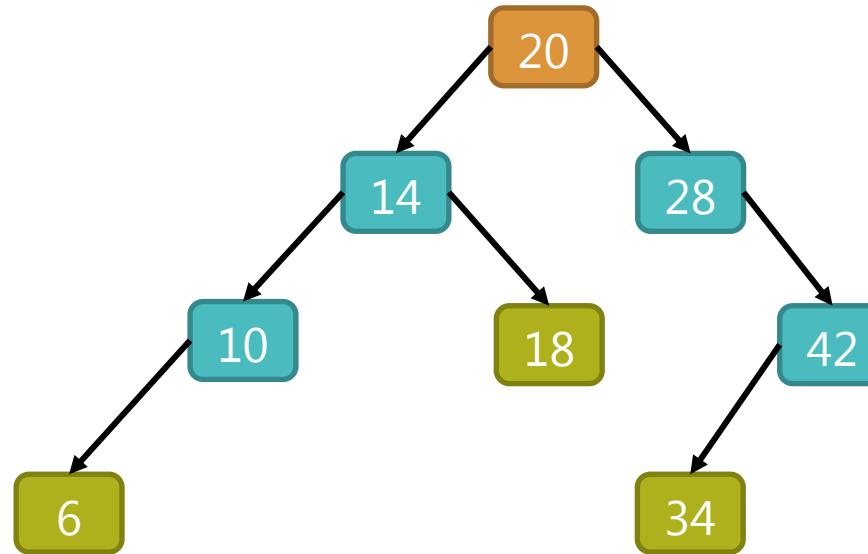


- $42 > 20 \rightarrow$ à droite ; $42 > 28 \rightarrow$ inséré à droite

ARBRES BINAIRES

Insertion des éléments dans un ABR (9/12)

- Eléments restant à insérer : 12, 11, 57

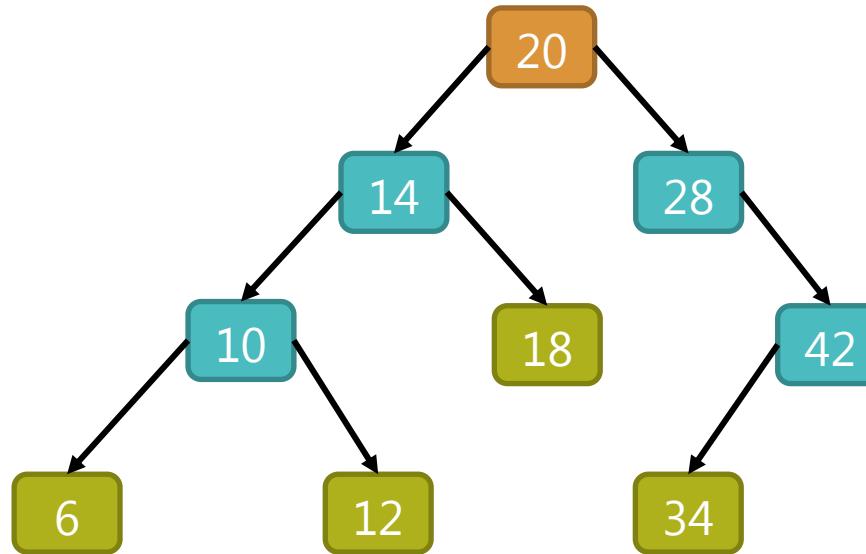


- $34 > 20 \rightarrow$ à droite ; $34 > 28 \rightarrow$ à droite ; $34 < 42 \rightarrow$ à gauche

ARBRES BINAIRES

Insertion des éléments dans un ABR (10/12)

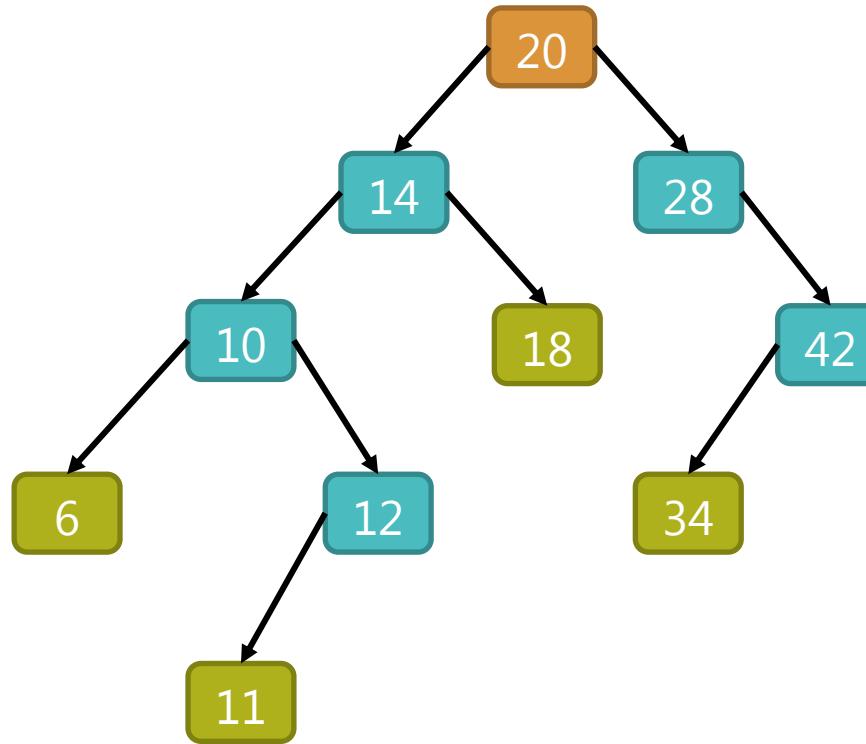
- Eléments restant à insérer : 11, 57



ARBRES BINAIRES

Insertion des éléments dans un ABR (11/12)

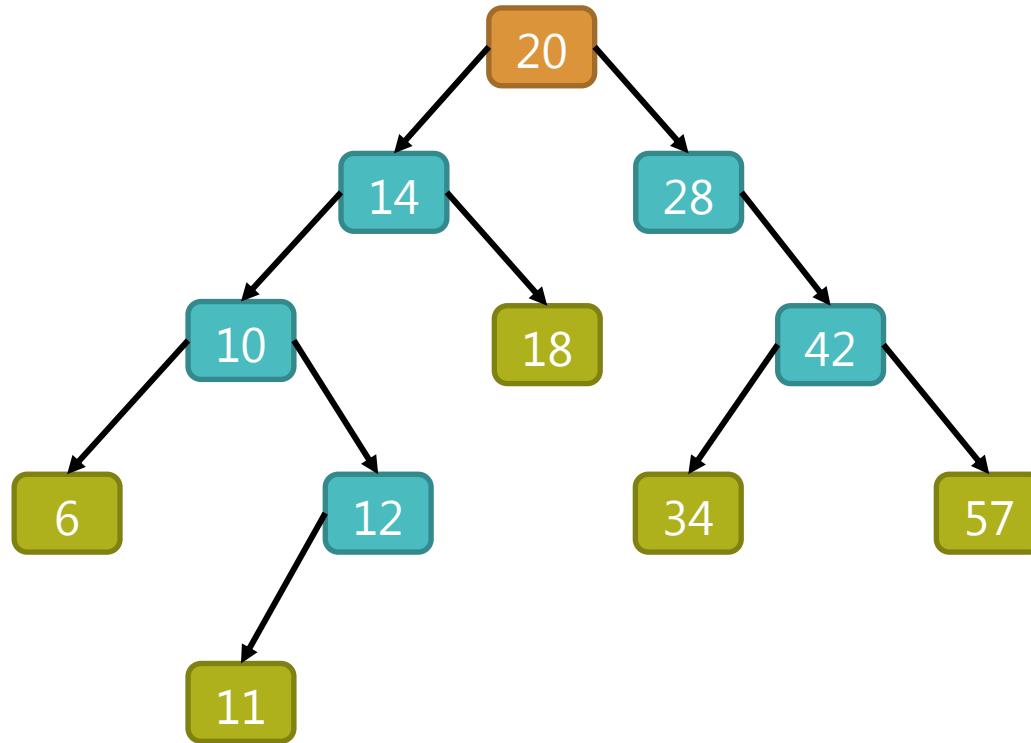
- Eléments restant à insérer : 57



ARBRES BINAIRES

Insertion des éléments dans un ABR (12/12)

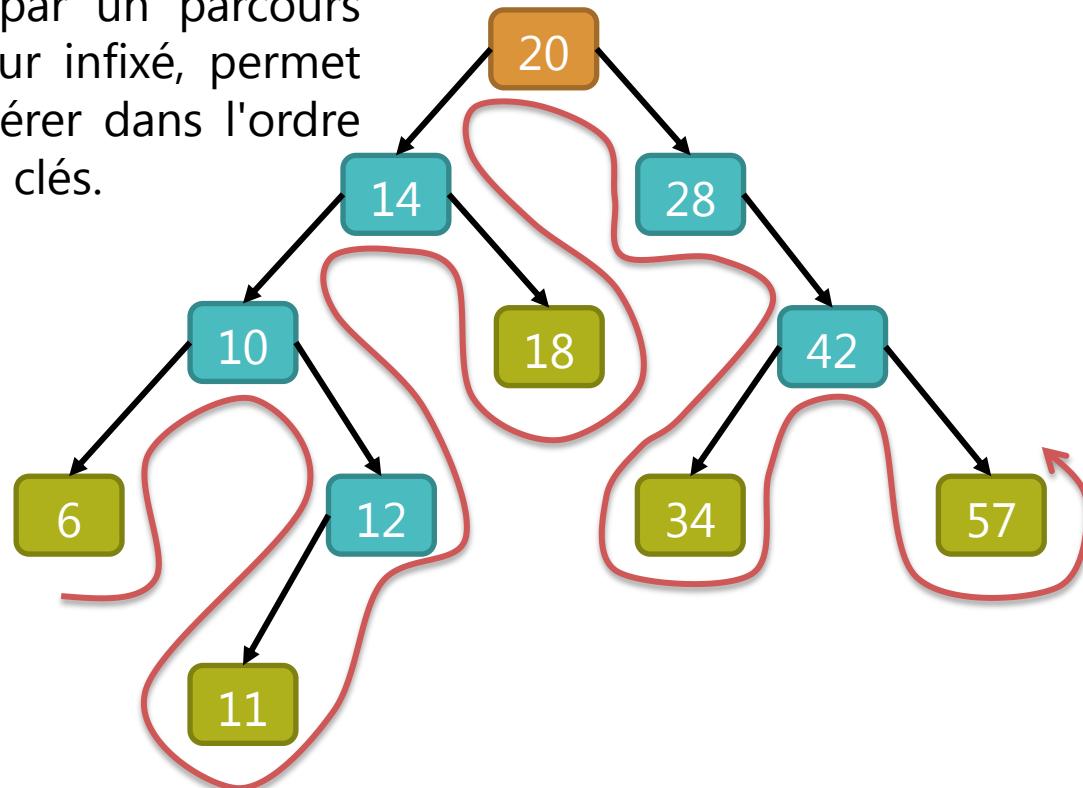
- Eléments restant à insérer :



ARBRES BINAIRES

Tri par ABR

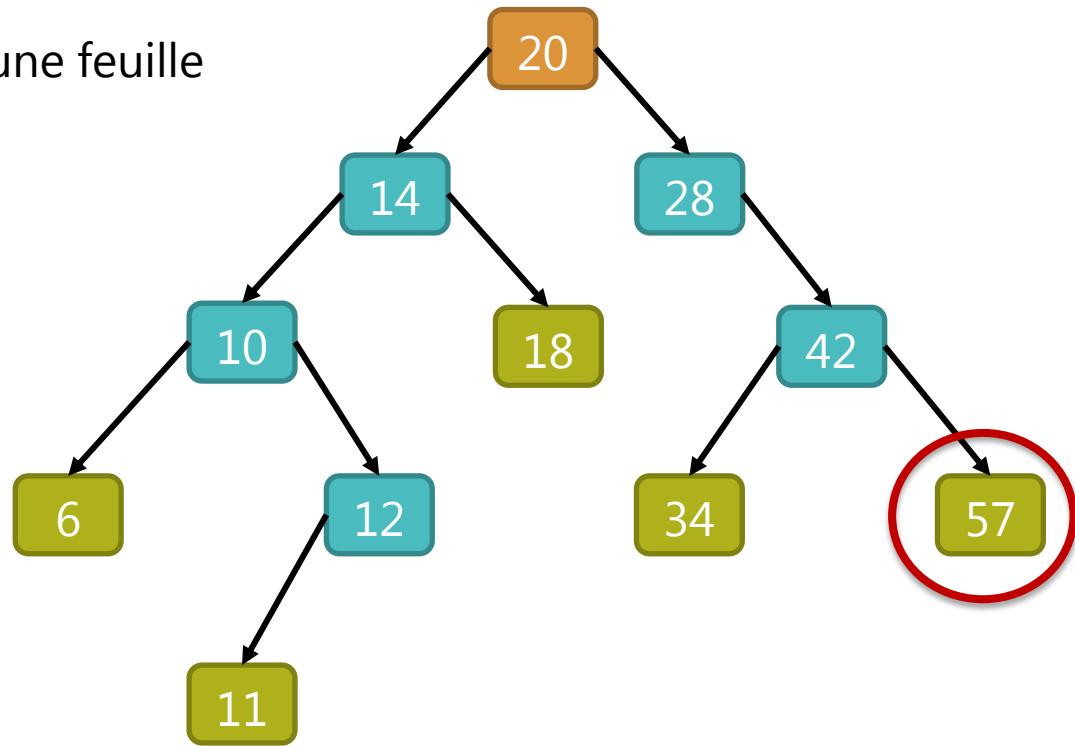
- L'insertion des éléments dans l'ABR, suivi par un parcours en profondeur infixé, permet de les récupérer dans l'ordre croissant des clés.



ARBRES BINAIRES

Suppression d'un noeud (1/8)

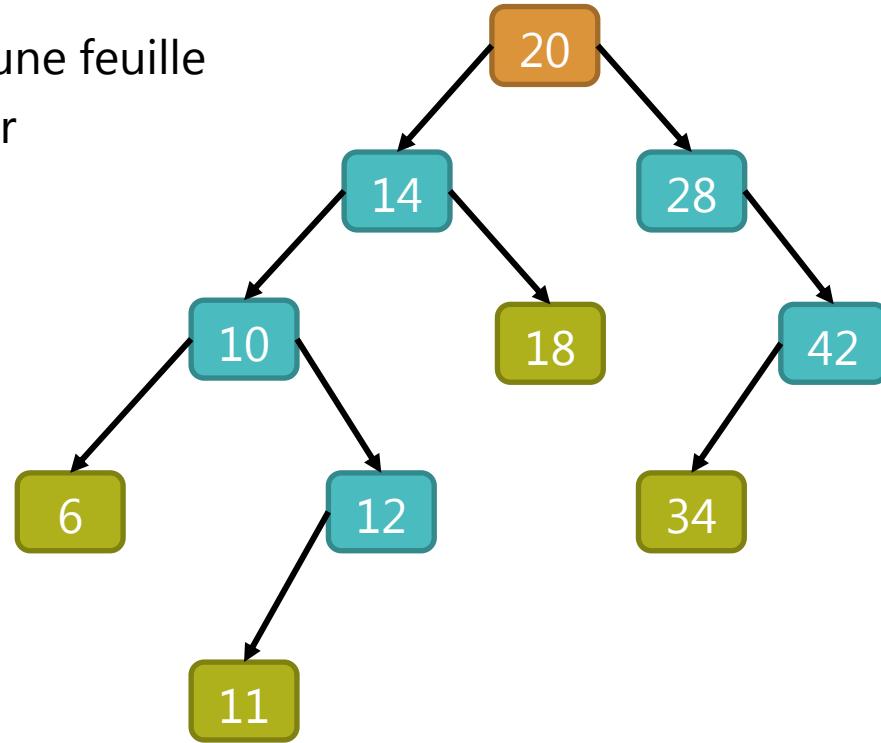
- 3 cas à considérer :
 - 1) Le nœud à supprimer est une feuille



ARBRES BINAIRES

Suppression d'un noeud (2/8)

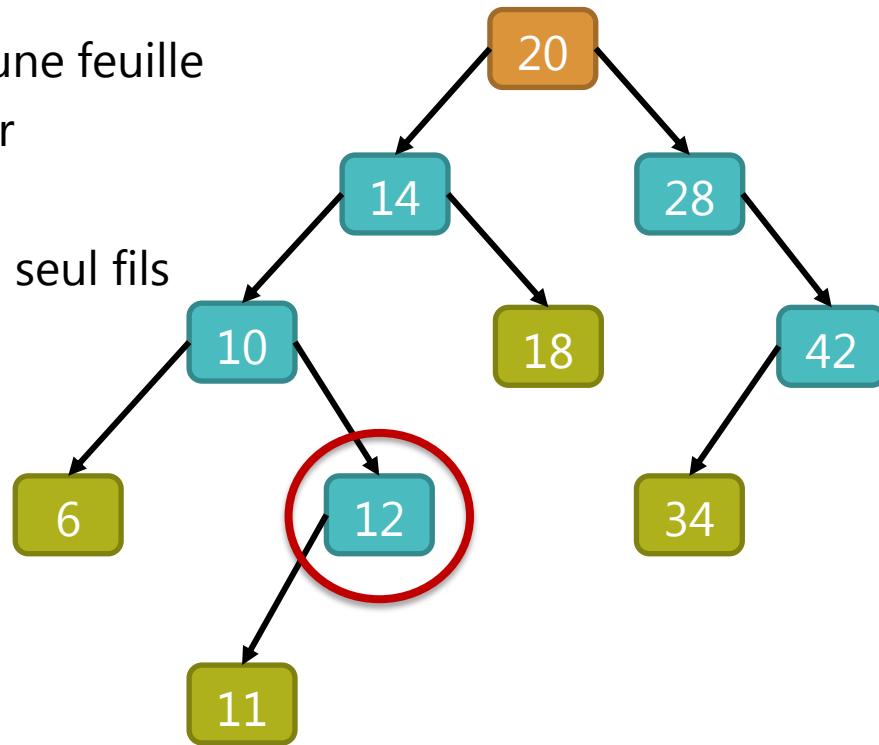
- 3 cas à considérer :
 - 1) Le nœud à supprimer est une feuille
 - Il suffit de la supprimer



ARBRES BINAIRES

Suppression d'un noeud (3/8)

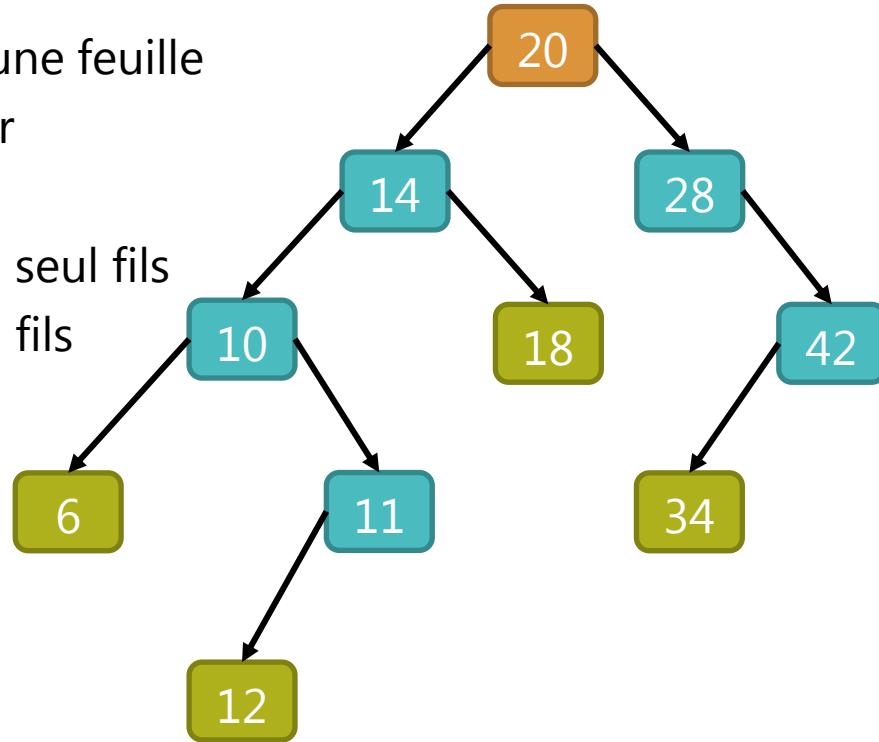
- 3 cas à considérer :
 - 1) Le nœud à supprimer est une feuille
 - Il suffit de la supprimer
 - 2) Le nœud à supprimer a un seul fils



ARBRES BINAIRES

Suppression d'un noeud (4/8)

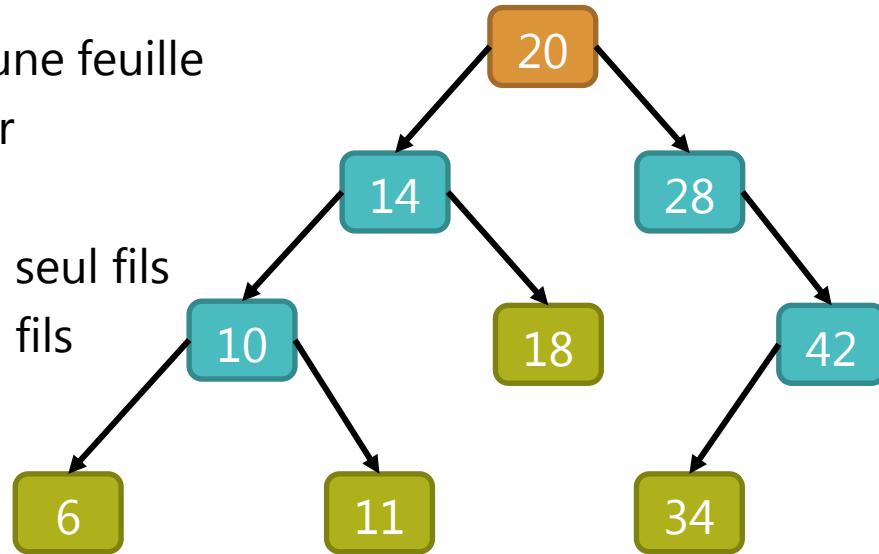
- 3 cas à considérer :
 - 1) Le nœud à supprimer est une feuille
 - Il suffit de la supprimer
 - 2) Le nœud à supprimer a un seul fils
 - On l'échange avec son fils



ARBRES BINAIRES

Suppression d'un noeud (5/8)

- 3 cas à considérer :
 - 1) Le nœud à supprimer est une feuille
 - Il suffit de la supprimer
 - 2) Le nœud à supprimer a un seul fils
 - On l'échange avec son fils
 - On supprime la feuille

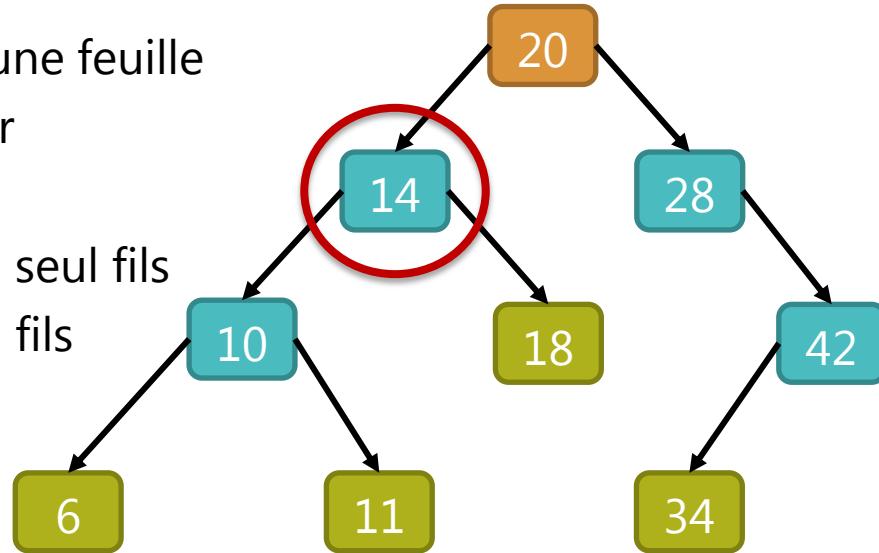


ARBRES BINAIRES

Suppression d'un noeud (6/8)

- 3 cas à considérer :

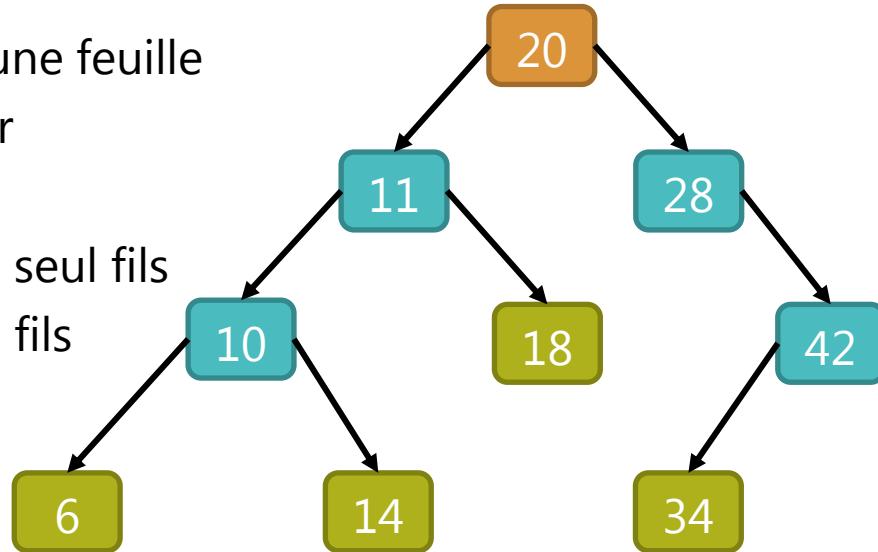
 - 1) Le nœud à supprimer est une feuille
 - Il suffit de la supprimer
 - 2) Le nœud à supprimer a un seul fils
 - On l'échange avec son fils
 - On supprime la feuille
 - 3) Le nœud a deux fils



ARBRES BINAIRES

Suppression d'un noeud (7/8)

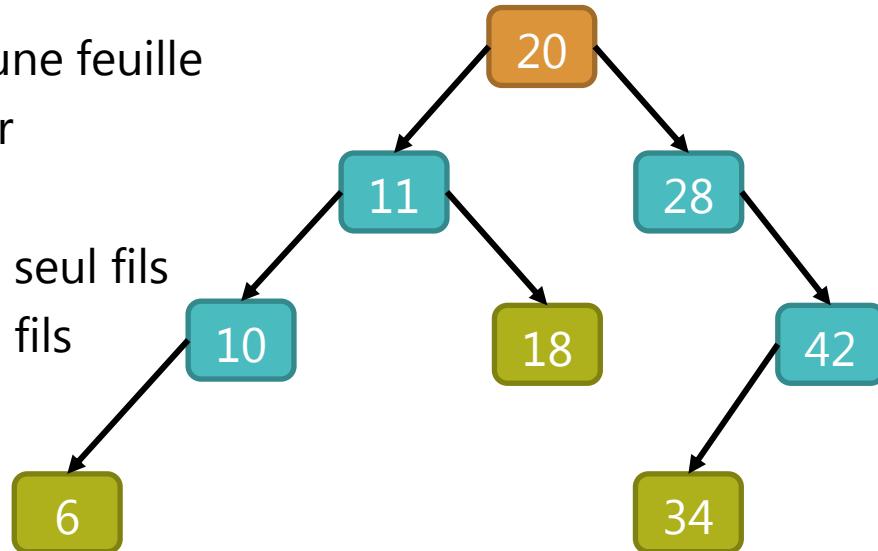
- 3 cas à considérer :
 - 1) Le nœud à supprimer est une feuille
 - Il suffit de la supprimer
 - 2) Le nœud à supprimer a un seul fils
 - On l'échange avec son fils
 - On supprime la feuille
 - 3) Le nœud a deux fils
 - On l'échange avec son + proche prédécesseur ou + proche successeur



ARBRES BINAIRES

Suppression d'un noeud (8/8)

- 3 cas à considérer :
 - 1) Le noeud à supprimer est une feuille
 - Il suffit de la supprimer
 - 2) Le noeud à supprimer a un seul fils
 - On l'échange avec son fils
 - On supprime la feuille
 - 3) Le noeud a deux fils
 - On l'échange avec son + proche prédécesseur ou + proche successeur
 - Puis on supprime le noeud devenu feuille.



ARBRES BINAIRES

Opérations sur le TAD arbre binaire de recherche

- **Nom :** TSearchTree,
- **Utilise :** **int, bool, TElement, TNode**
- **Opérations :**
 - New → TSearchTree
 - IsEmpty : TSearchTree → **bool**
 - Size : TSearchTree → **int**
 - AddNode : TSearchTree × TElement → TNode
 - SearchKey : TSearchTree × TElement → TNode
 - RemoveNode : TSearchTree × TElement → **bool**
 - Traversal : TSearchTree →
 - Clear : TSearchTree →
 - Delete : TSearchTree →

ARBRES BINAIRES

Implémentation orientée objet d'un ABR : "classe" TNode

```
/* Structure noeud générique */

typedef struct Node {
    void *pKey;           /* Pointe vers le contenu */
    struct Node *LeftChild; /* Fils gauche */
    struct Node *RightChild; /* Fils droit */
} TNode;

typedef TNode *PTNode;
```

ARBRES BINAIRES

Implémentation orientée objet d'un ABR : "classe" TSearchTree

```

typedef struct SearchTree {
    size_t NumNodes;           /* Nombre de noeuds */
    size_t SizeOfKey;          /* Taille d'une clé */
    PTNode Root;               /* Racine */
    PTNode Current;            /* Noeud courant */
    bool(*IsEmpty)(const struct SearchTree * );
    size_t(*Size)(const struct SearchTree * );
    PTNode(*AddNode)(const struct SearchTree *, const void *,
                     int(*compare)(const void *, const void *));
    PTNode(*SearchKey)(const struct SearchTree *, const void *,
                       int(*compare)(const void *, const void *));
    bool(*RemoveNode)(const struct SearchTree *, const void *,
                        int(*compare)(const void *, const void *));
    PTNode(*TraversalInOrder)(const PTNode subTree,
                             void(*action)(const void *));
    void Clear(const struct SearchTree * );
    void Delete(const struct SearchTree * );
} TSearchTree;

typedef TSearchTree *PTSearchTree;

PTSearchTree TSearchTree_New(size_t keySize);
  
```

ARBRES BINAIRES

Ajout d'une clé dans l'ABR

```

PTNode TSearchTree_AddNode(const PSearchTree this, const void *newKey,int(*compare)(const void *, const void *)) {
    PTNode newNode = malloc(sizeof(TNode));           /* Initialisation du nouveau noeud */
    newNode->pKey = malloc(this->SizeOfKey);        /* avec la valeur de la clé */
    memcpy(newNode->pKey, newKey, this->SizeOfKey);  /* passée en paramètre */
    newNode->LeftChild = newNode->RightChild = NULL; /* Le nouveau noeud n'a aucun fils */
    if (this->NumNodes == 0) this->Current = this->Root = newNode; /* Si arbre vide, racine = nouveau noeud */
    else { /* sinon commencer le parcours de l'arbre */
        this->Current = this->Root;
        bool done = false;
        while (!done) {
            if (compare(newKey, this->Current->pKey) < 0) { /* clé à insérer < clé courante */
                if (this->Current->LeftChild) this->Current = this->Current->LeftChild;
                else {
                    this->Current->LeftChild = newNode;
                    done = true;
                }
            }
            else { /* clé à insérer >= clé courante */
                if (this->Current->RightChild) this->Current = this->Current->RightChild;
                else {
                    this->Current->Right = newNode;
                    done = true;
                }
            }
        } /* while */
    } /* else */
    this->NumNodes++;
    return newNode;
}

```

ARBRES BINAIRES

Recherche d'une clé dans l'ABR

- La recherche commence à la racine, elle est orienté vers le sous-arbre gauche (resp. droit) selon que la clé recherchée est inférieure (resp. supérieure), puis on recommence itérativement jusqu'à coïncidence (clé trouvée) ou jusqu'à la feuille la plus profonde (clé introuvable).

```

PTNode TSearchTree_SearchKey(const PTSearchTree this, const void *pKey,
                             int(*compare)(const void *, const void *))
{
    PTNode currNode = this->Root;
    while (currNode) {
        if (compare(pKey, currNode->pKey) == 0)          /* clé recherchée = clé courante */
            return currNode;
        if (compare(pKey, currNode->pKey) < 0)          /* clé recherchée < clé courante */
            currNode = currNode->Left;                   /* Aller à gauche */
        else
            currNode = currNode->Right;                 /* sinon aller à droite */
    }
    return currNode;
}
  
```

ARBRES BINAIRES

Parcours infixé de l'ABR ou de l'un de ses sous-arbres

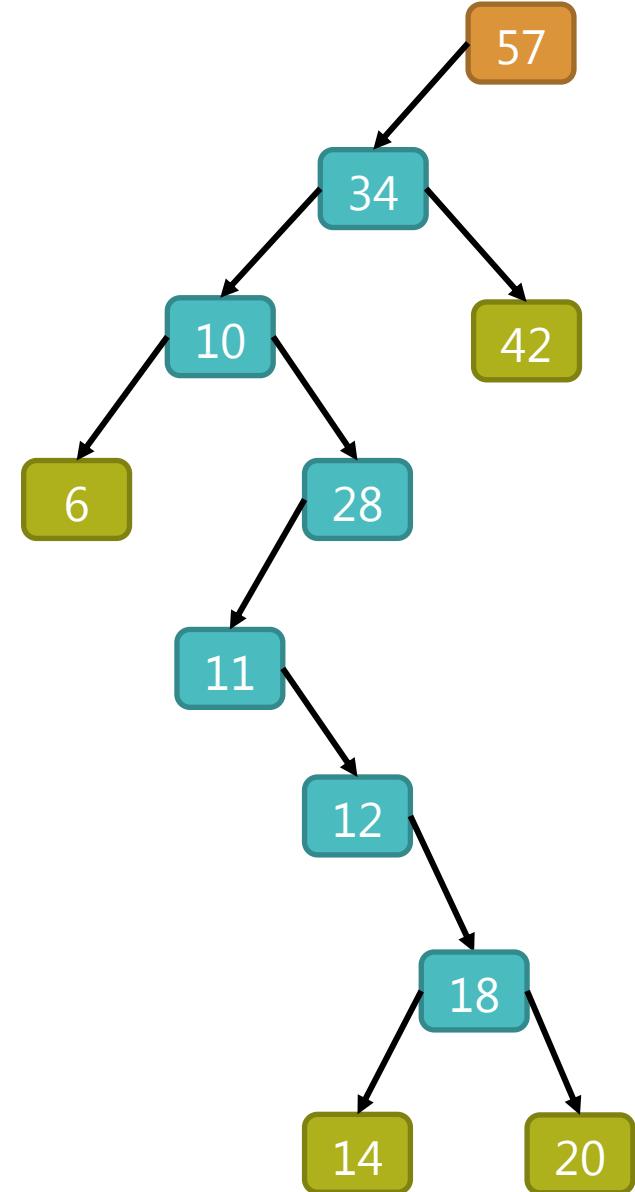
- Cette fonction parcourt récursivement un sous-arbre (éventuellement l'arbre entier à partir de la racine), d'abord à gauche, puis à droite.
- L'action à réaliser est définie par la fonction de callback *action* passée en second paramètre.

```
void TSearchTree_TraversalInOrder(const PTNode subTree, void(*action)(const void *))  
{  
    if (subTree) {  
        if (subTree->Left)  
            TSearchTree_TraversalInOrder(subTree->Left, action);  
        action(subTree->pKey);  
        if (subTree->Right)  
            TSearchTree_TraversalInOrder(subTree->Right, action);  
    }  
}
```

ARBRES BINAIRES

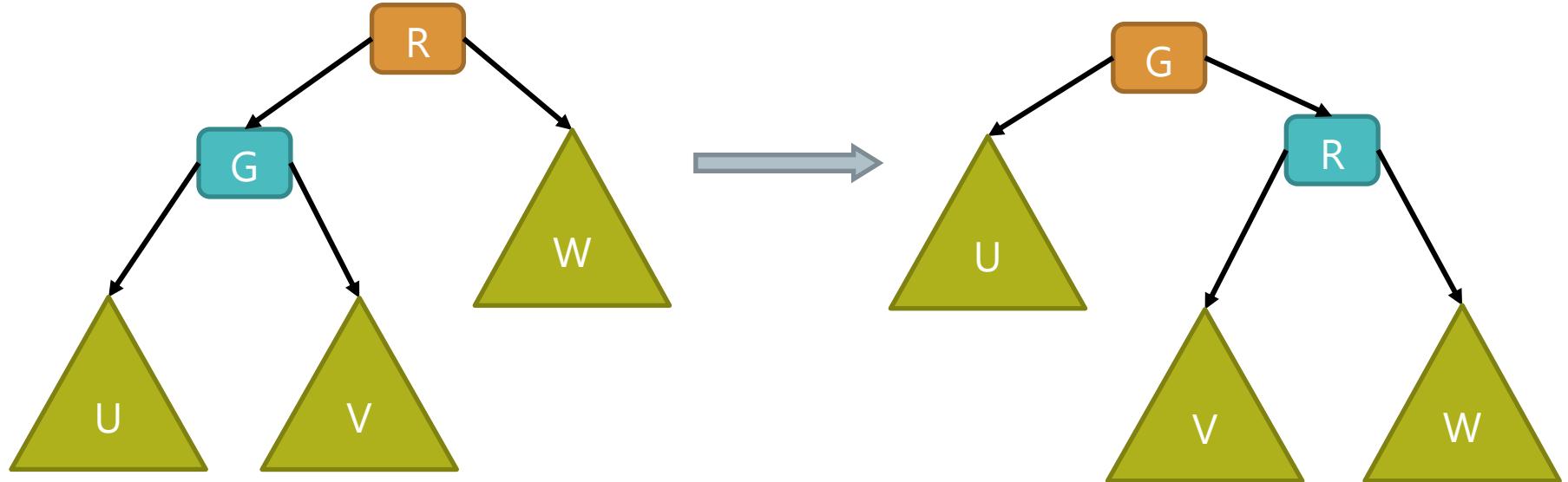
Complexité de la recherche dans un ABR

- La recherche d'un élément (une clé) dans un ABR est similaire à la recherche par dichotomie dans une collection triée, et la complexité est en $O(\log(n))$.
- Si l'arbre est mal équilibré, la complexité peut tendre vers $O(n)$, comme dans une liste chainée.
- Il peut alors être nécessaire de rééquilibrer l'arbre *par rotation*.



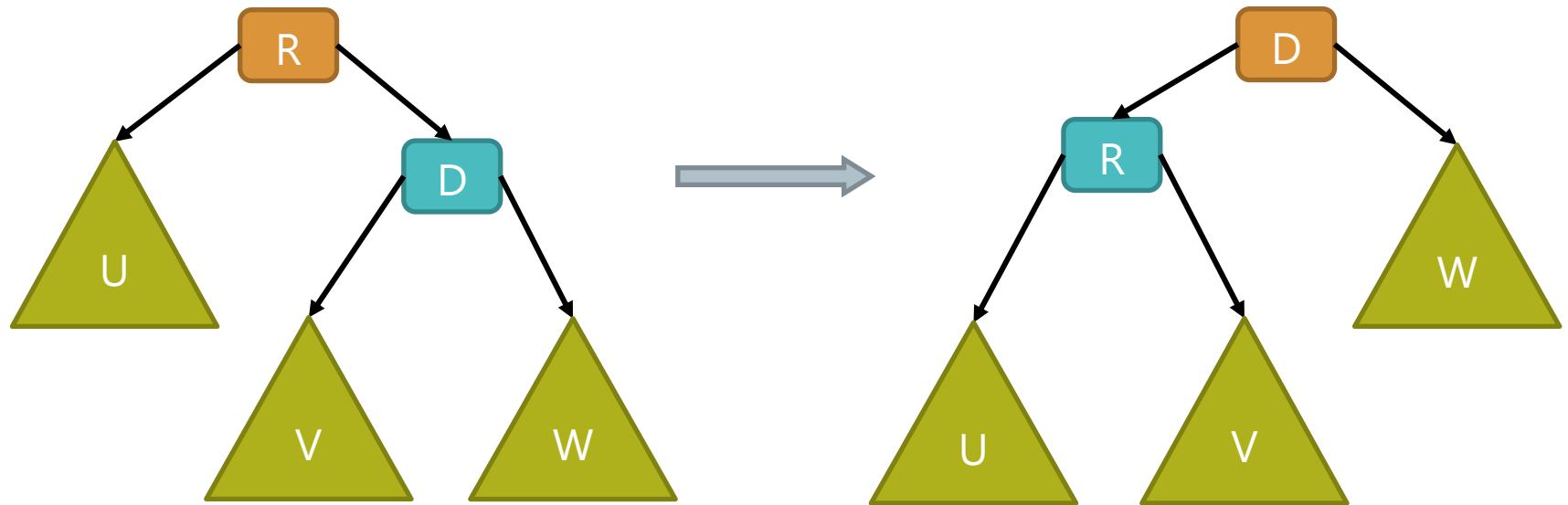
ARBRES BINAIRES

Simple rotation à droite



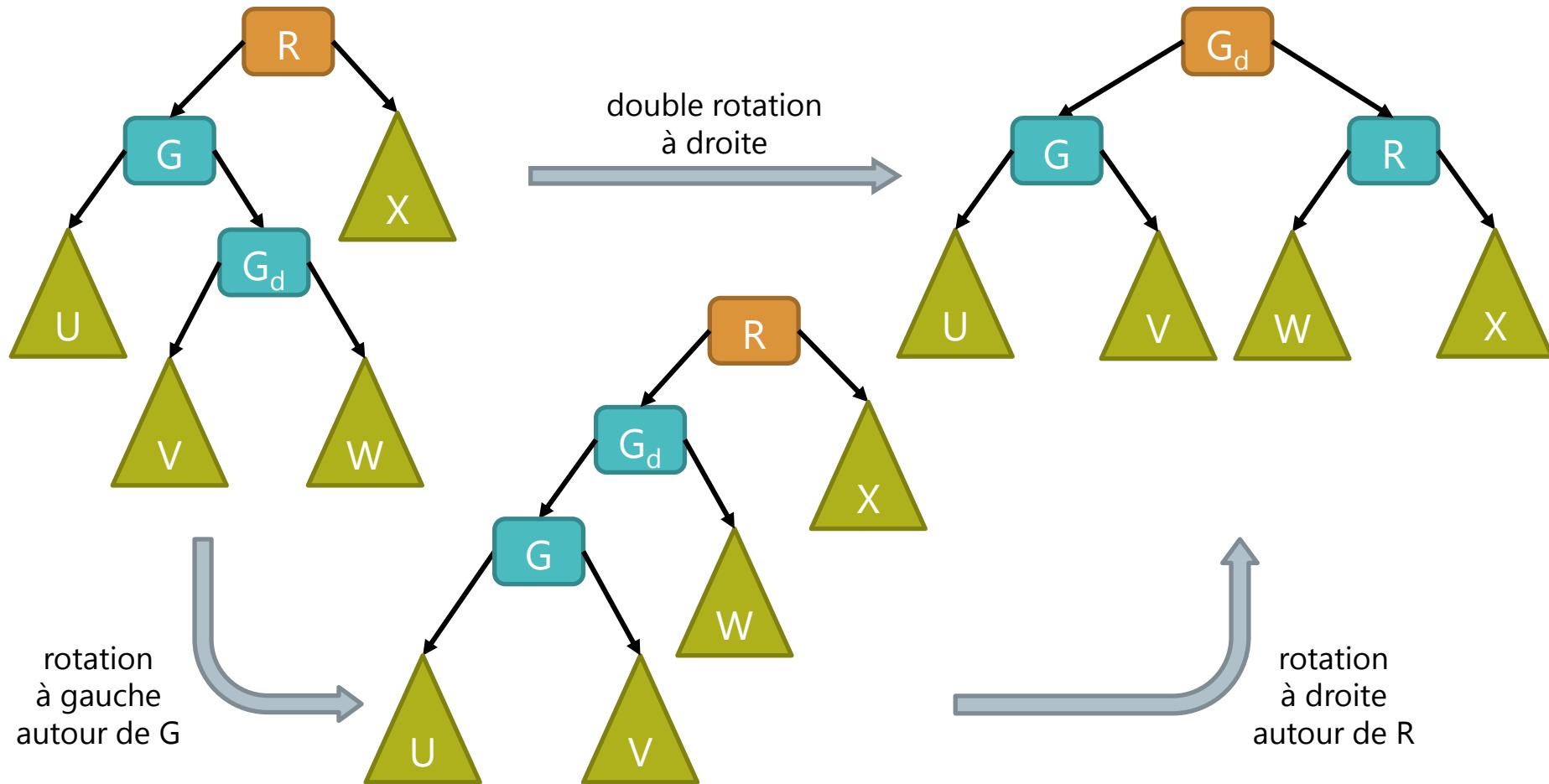
ARBRES BINAIRES

Simple rotation à gauche



ARBRES BINAIRES

Double rotation à droite



ARBRES BINAIRES

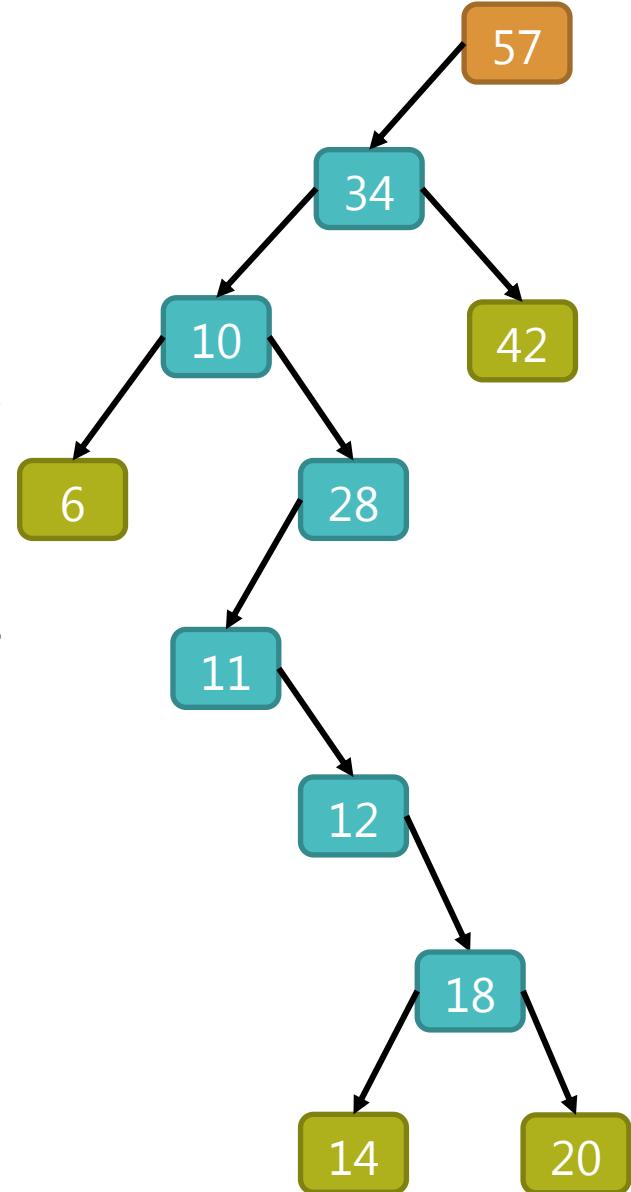
Arbres AVL

- Les arbres AVL sont des ABR automatiquement équilibrés.
- Inventés en 1962 par (*Georgii Adelson-Velsky* et *Evguenii Landis*).
- Facteur d'équilibrage d'un nœud : différence entre la hauteur de son sous-arbre gauche et la hauteur de son sous-arbre droit.
- Un nœud équilibré a un facteur d'équilibrage de -1, 0, ou +1.
- Opérations similaires aux opérations sur les ABR, complétés par une rotation à droite ou à gauche en cas de déséquilibre.

ARBRES BINAIRES

Arbre binaire déséquilibré

- L'insertion des éléments suivants dans un ABR (dans cet ordre) : 57, 34, 10, 42, 28, 6, 11, 12, 18, 14, 20, abouti à un arbre déséquilibré (ci-contre)
- Dans un arbre AVL, l'arbre est rééquilibré par rotation en cas de nécessité à chaque opération d'insertion ou de suppression d'un nœud, ce qui évite ce genre de situation.



ARBRES BINAIRES

Insertion des éléments dans un AVL (1/20)

- Eléments à insérer dans cet ordre : 57, 34, 10, 42, 28, 6, 11, 12, 18, 14, 20

ARBRES BINAIRES

Insertion des éléments dans un AVL (2/20)

- Eléments restant à insérer : 34, 10, 42, 28, 6, 11, 12, 18, 14, 20

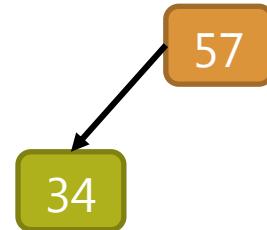
57

- Le premier élément est inséré en tant que racine de l'arbre

ARBRES BINAIRES

Insertion des éléments dans un AVL (3/20)

- Eléments restant à insérer : 10, 42, 28, 6, 11, 12, 18, 14, 20

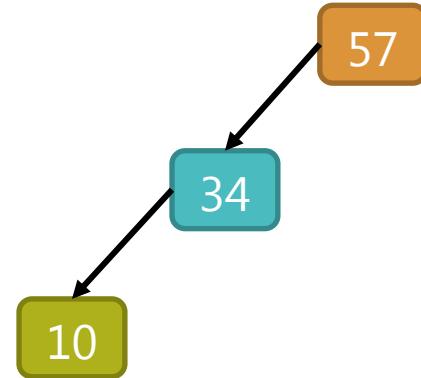


- $34 < 57 \rightarrow$ inséré à gauche

ARBRES BINAIRES

Insertion des éléments dans un AVL (4/20)

- Eléments restant à insérer : 42, 28, 6, 11, 12, 18, 14, 20

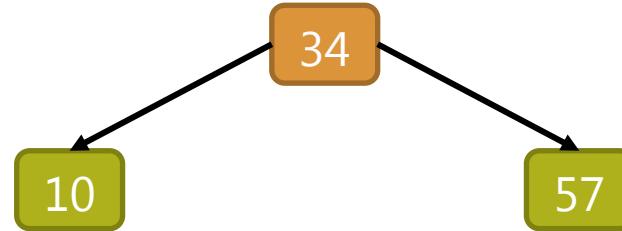


- $10 < 57 \rightarrow$ à gauche
 - $10 < 34 \rightarrow$ inséré à gauche
- déséquilibre au niveau racine

ARBRES BINAIRES

Insertion des éléments dans un AVL (5/20)

- Eléments restant à insérer : 42, 28, 6, 11, 12, 18, 14, 20

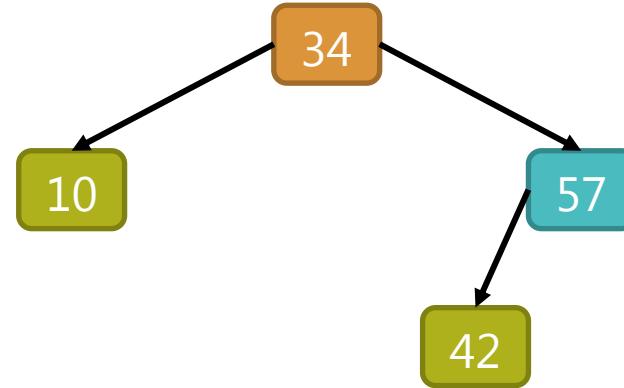


- arbre rééquilibré après rotation à droite

ARBRES BINAIRES

Insertion des éléments dans un AVL (6/20)

- Eléments restant à insérer : 28, 6, 11, 12, 18, 14, 20

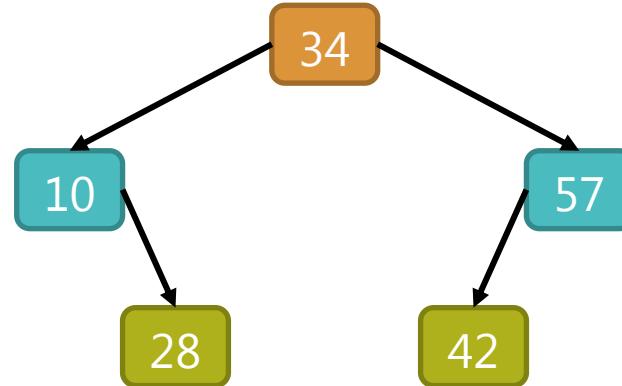


- $42 > 34 \rightarrow$ à droite
- $42 < 57 \rightarrow$ inséré à gauche

ARBRES BINAIRES

Insertion des éléments dans un AVL (7/20)

- Eléments restant à insérer : 6, 11, 12, 18, 14, 20

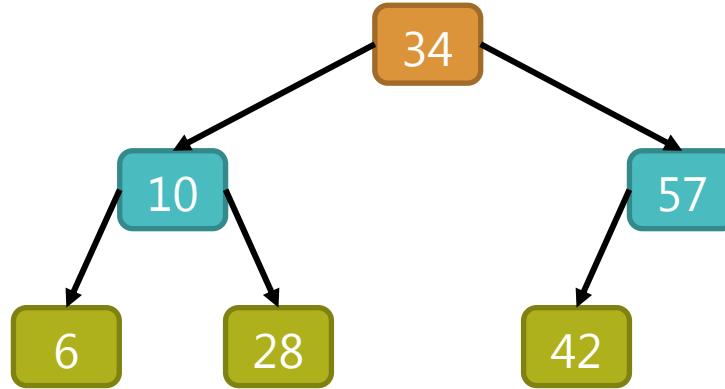


- $28 < 34 \rightarrow$ à gauche
- $28 < 10 \rightarrow$ inséré à droite

ARBRES BINAIRES

Insertion des éléments dans un AVL (8/20)

- Eléments restant à insérer : 11, 12, 18, 14, 20

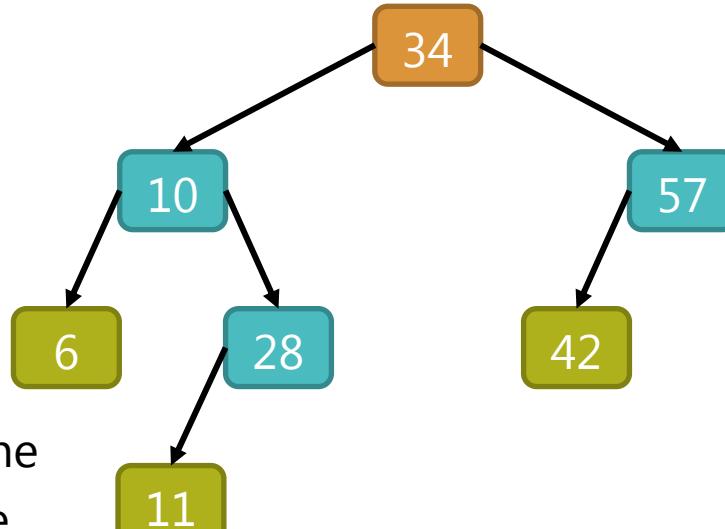


- $6 < 34 \rightarrow$ à gauche
- $6 < 10 \rightarrow$ inséré à gauche

ARBRES BINAIRES

Insertion des éléments dans un AVL (9/20)

- Eléments restant à insérer : 12, 18, 14, 20



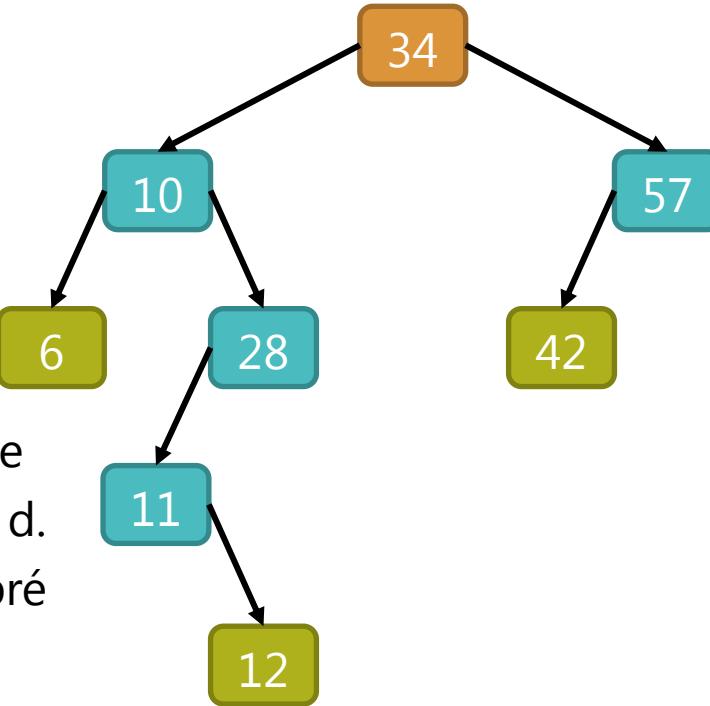
- $11 < 34 \rightarrow$ à gauche
- $11 > 10 \rightarrow$ à droite
- $11 < 28 \rightarrow$ inséré à gauche

ARBRES BINAIRES

Insertion des éléments dans un AVL (10/20)

- Eléments restant à insérer : 18, 14, 20

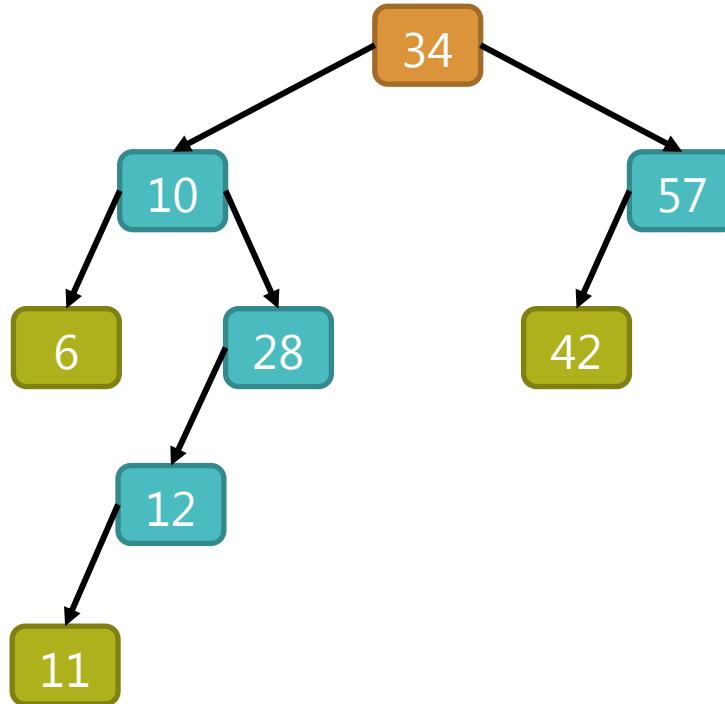
- $12 < 34 \rightarrow$ à g.
 - $12 > 10 \rightarrow$ à droite
 - $12 < 28 \rightarrow$ à gauche
 - $12 > 11 \rightarrow$ inséré à d.
- nœud 28 déséquilibré



ARBRES BINAIRES

Insertion des éléments dans un AVL (11/20)

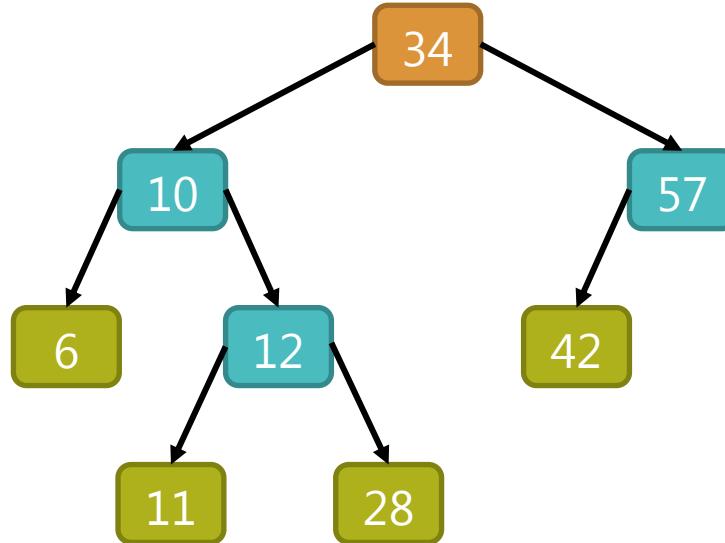
- Rotation à droite pour rééquilibrer le sous-arbre



ARBRES BINAIRES

Insertion des éléments dans un AVL (12/20)

- Eléments restant à insérer : 18, 14, 20



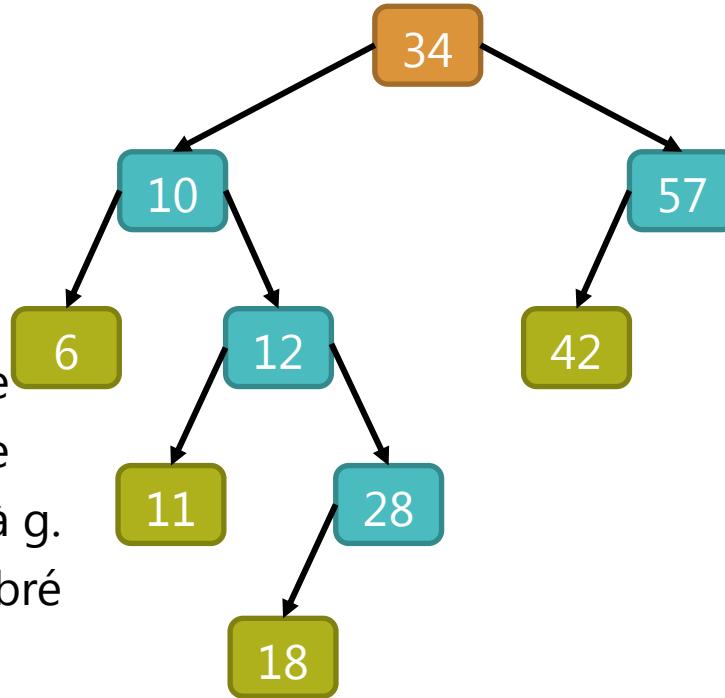
- sous-arbre rééquilibré après rotation à droite

ARBRES BINAIRES

Insertion des éléments dans un AVL (13/20)

- Eléments restant à insérer : 14, 20

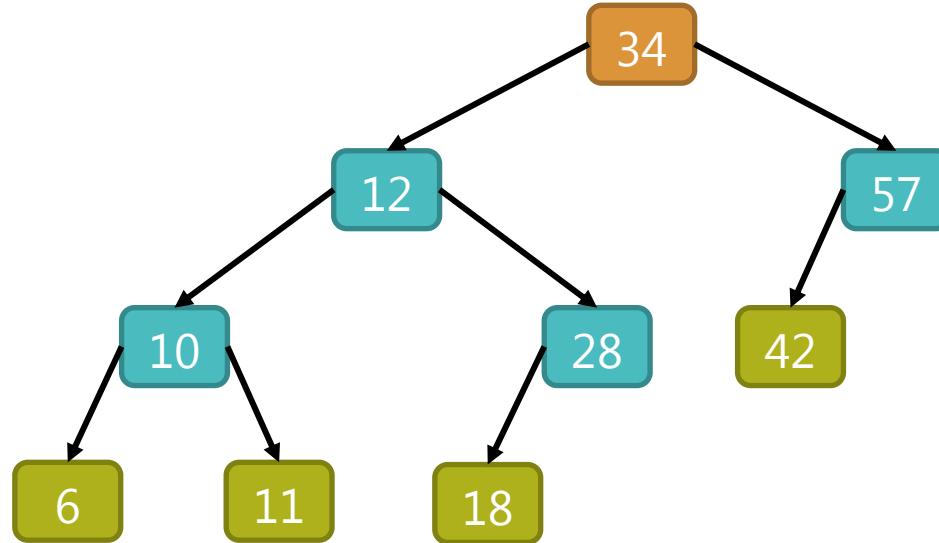
- $18 < 34 \rightarrow$ à g.
 - $18 > 10 \rightarrow$ à droite
 - $18 < 12 \rightarrow$ à droite
 - $18 < 28 \rightarrow$ inséré à g.
- nœud 10 déséquilibré



ARBRES BINAIRES

Insertion des éléments dans un AVL (14/20)

- Eléments restant à insérer : 14, 20

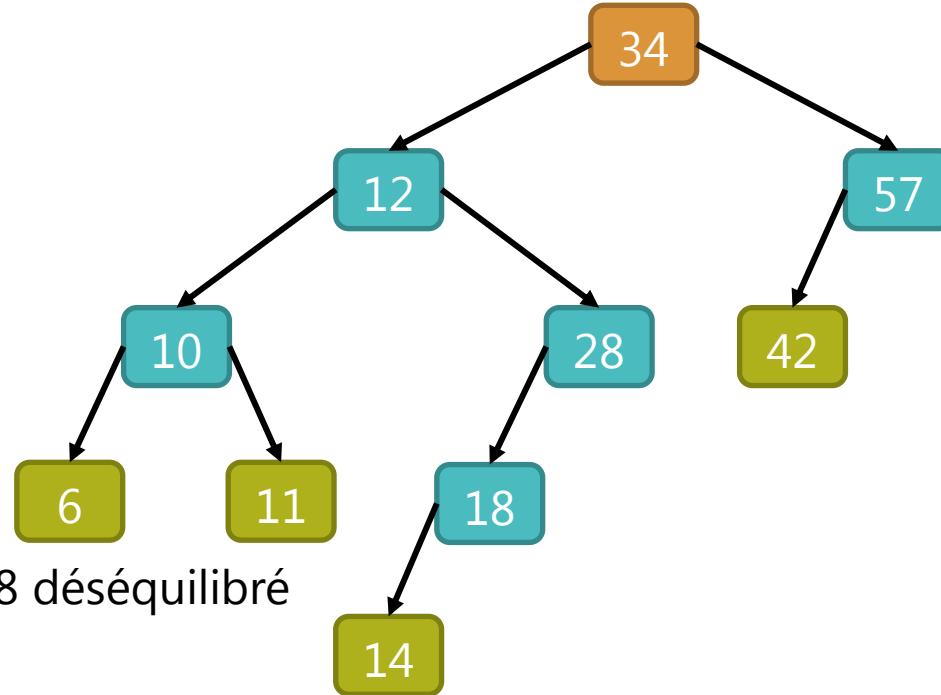


- sous-arbre rééquilibré après rotation à gauche

ARBRES BINAIRES

Insertion des éléments dans un AVL (15/20)

- Eléments restant à insérer : 20

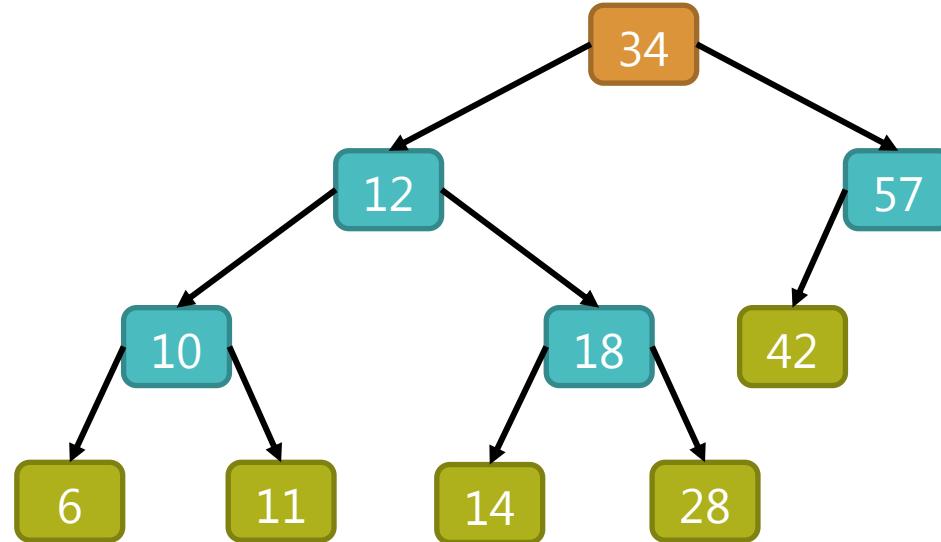


- nœud 28 déséquilibré

ARBRES BINAIRES

Insertion des éléments dans un AVL (16/20)

- Eléments restant à insérer : 20

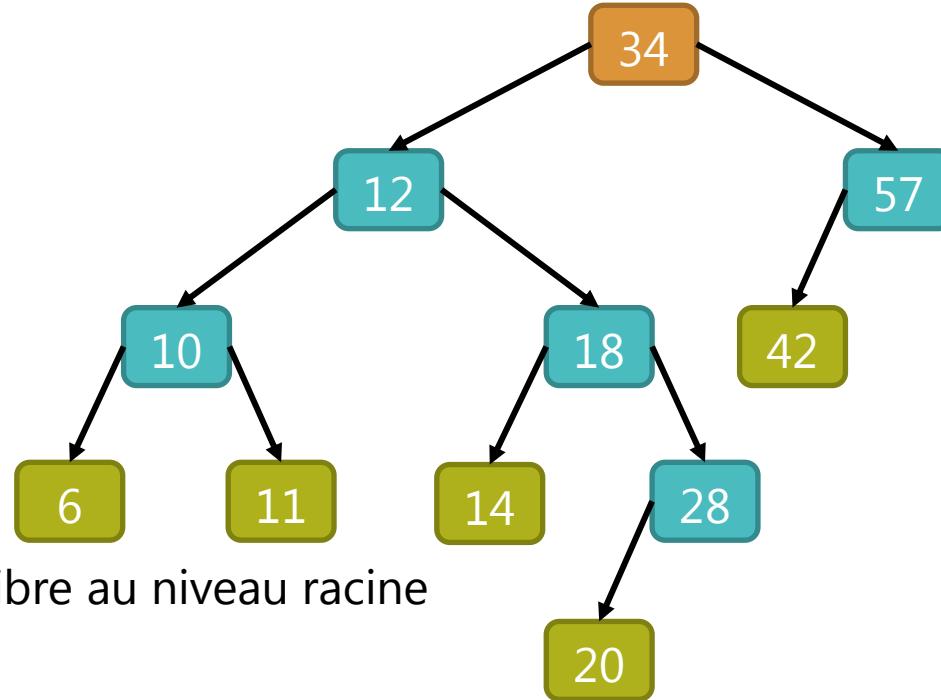


- sous-arbre rééquilibré après rotation à gauche

ARBRES BINAIRES

Insertion des éléments dans un AVL (17/20)

- après insertion du dernier élément (20)

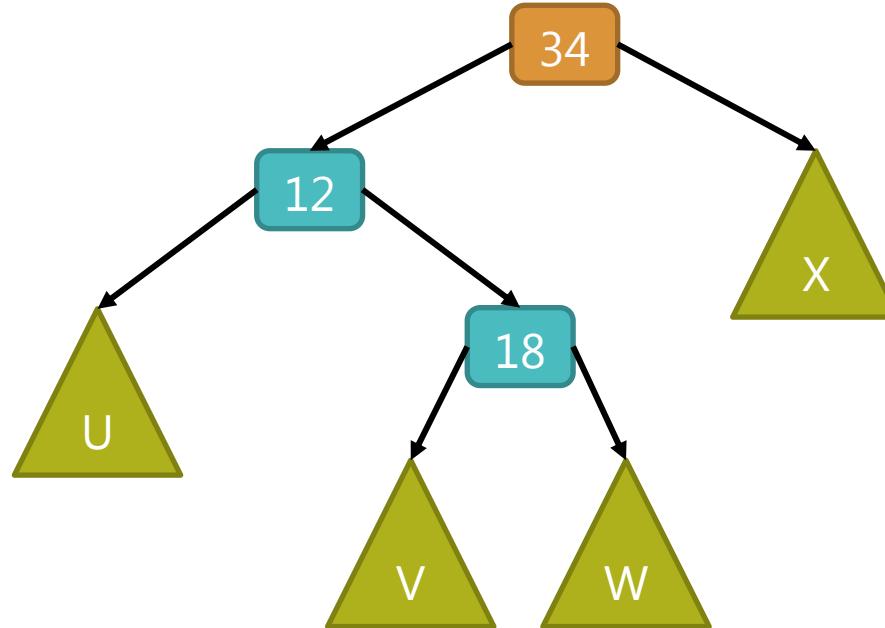


→ déséquilibre au niveau racine

ARBRES BINAIRES

Insertion des éléments dans un AVL (18/20)

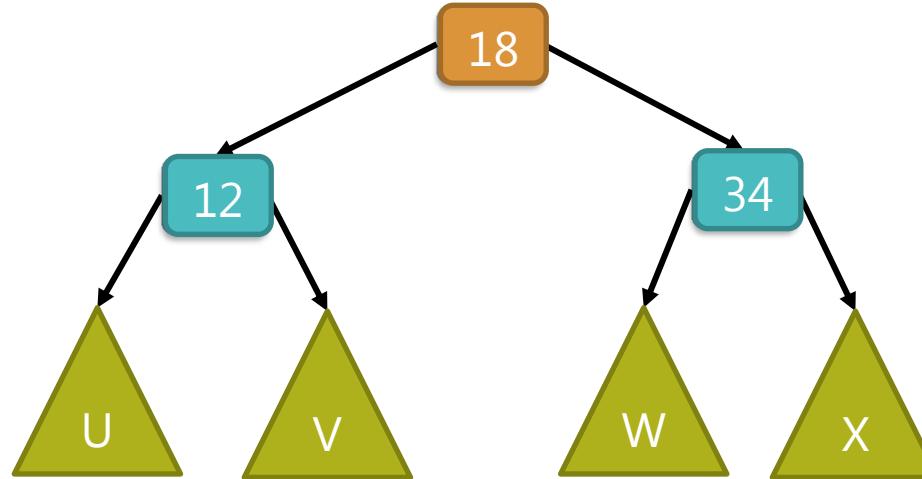
- la solution pour rééquilibrer l'arbre est une double rotation à droite



ARBRES BINAIRES

Insertion des éléments dans un AVL (19/20)

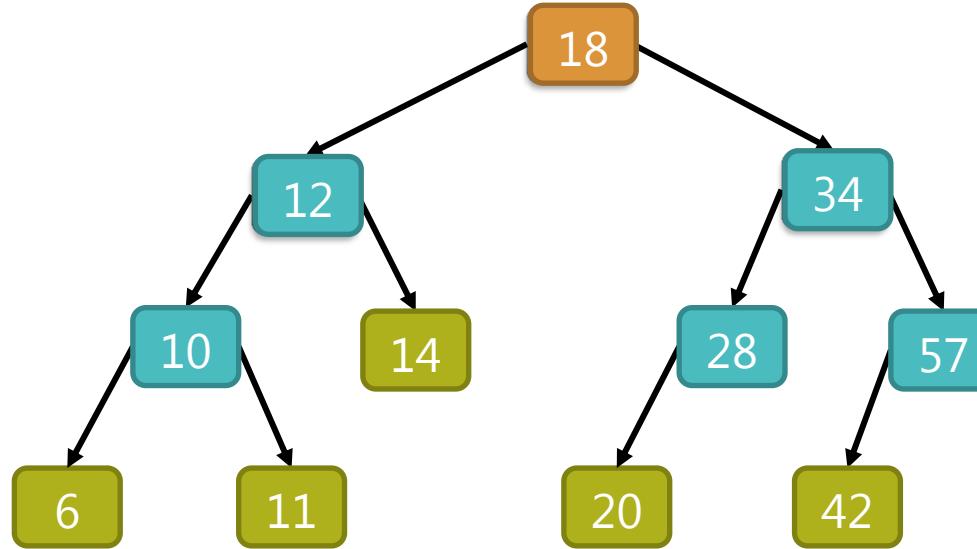
- après rotation à droite autour de la racine l'arbre est rééquilibré.



ARBRES BINAIRES

Insertion des éléments dans un AVL (20/20)

- après double rotation à droite autour de la racine l'arbre est rééquilibré : on peut remarquer que la rotation préserve l'ordre infixé.



ARBRES BINAIRES

Implémentation d'un arbre AVL

- Comment savoir s'il faut rééquilibrer l'arbre quand on insère un nouveau nœud ?
- Il faut connaître le facteur d'équilibrage du nœud :
 - en calculant la différence de hauteur entre chaque sous-arbre du nœud,
 - ou bien stocker et mettre à jour ce facteur dans le nœud.

```

typedef struct Node {
    void *pKey;           /* Pointeur vers la clé */
    int balance;          /* Facteur d'équilibrage */
    struct Node *leftChild;
    struct Node *rightChild;
} TNode;

typedef TNode *PTNode;

typedef struct AVLTree {
    int size;              /* Nombre de noeuds */
    int sizeofkey;         /* Taille unitaire de clé */
    PTNode Root;           /* Nœud racine */
    PTNode Current;        /* Nœud courant */
} TAVLTree;

typedef TAVLTree *PTAVLTree;
  
```

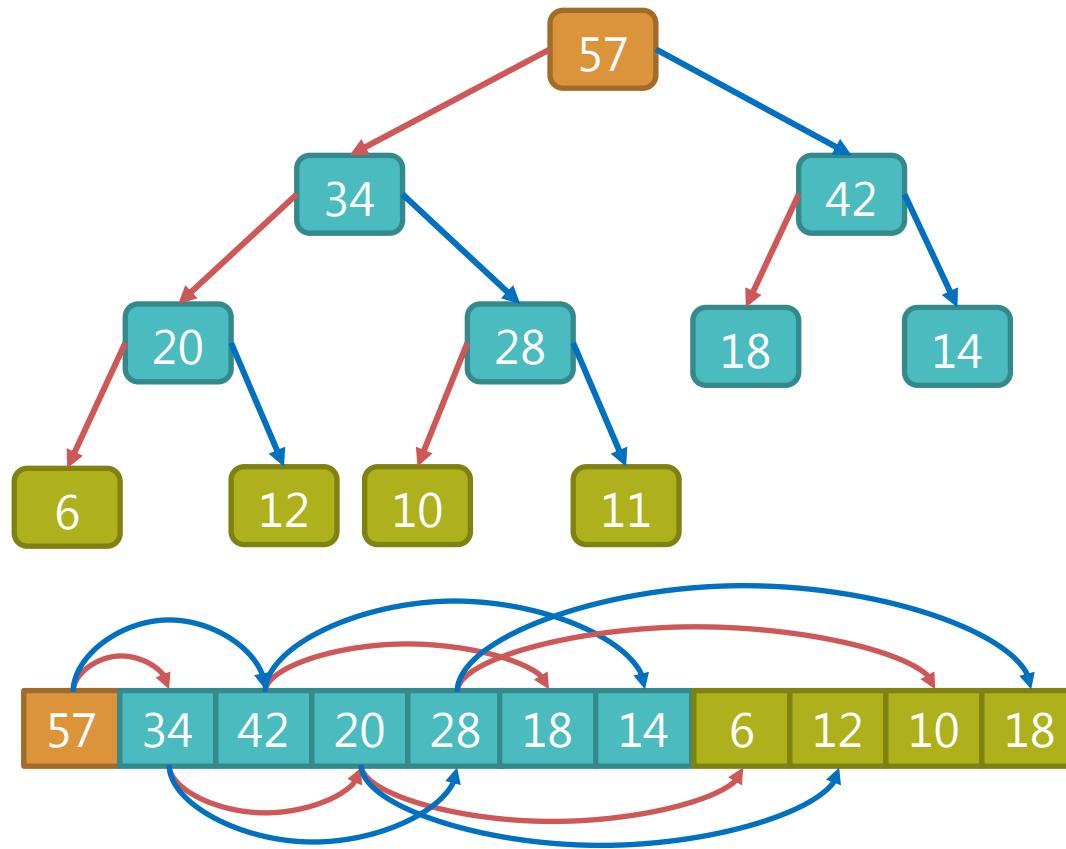
ARBRES BINAIRES

Structure de tas binaire

1. Un arbre tassé ou tas (*heap*) est un arbre binaire complet : tous les niveaux sont remplis, sauf éventuellement le dernier. Si le dernier niveau n'est pas rempli, il doit l'être de gauche à droite.
2. La clé de chaque nœud doit être \geq (respectivement \leq) aux clés de chacun de ses fils.
 - Si la relation d'ordre est " \geq " on a un *tas-max (max-heap)*
 - Si la relation d'ordre est " \leq " on a un *tas-min (min-heap)*
- Un tas étant un arbre binaire complet, il peut être représenté sous forme de tableau.

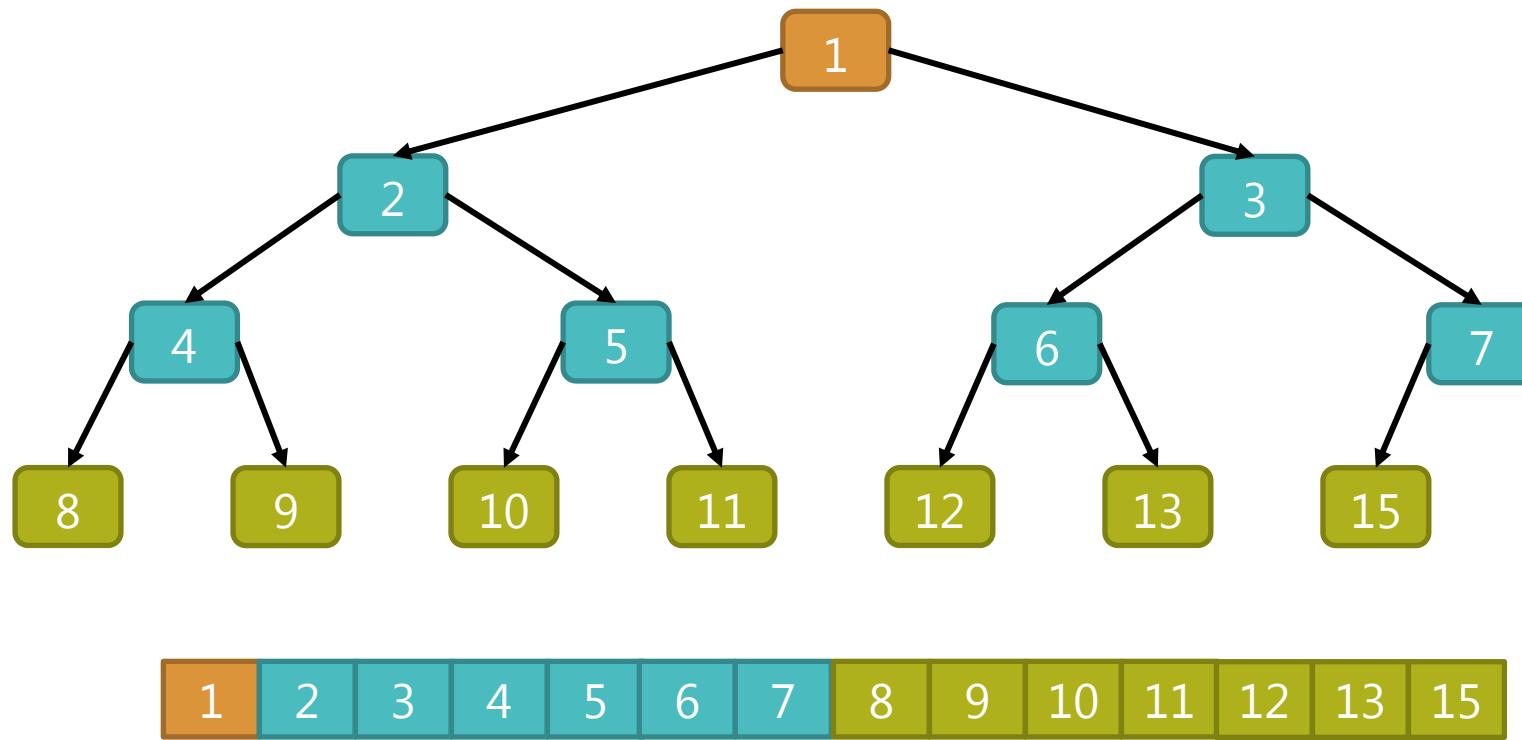
ARBRES BINAIRES

Exemple de représentation d'un tas en arbre ou en tableau



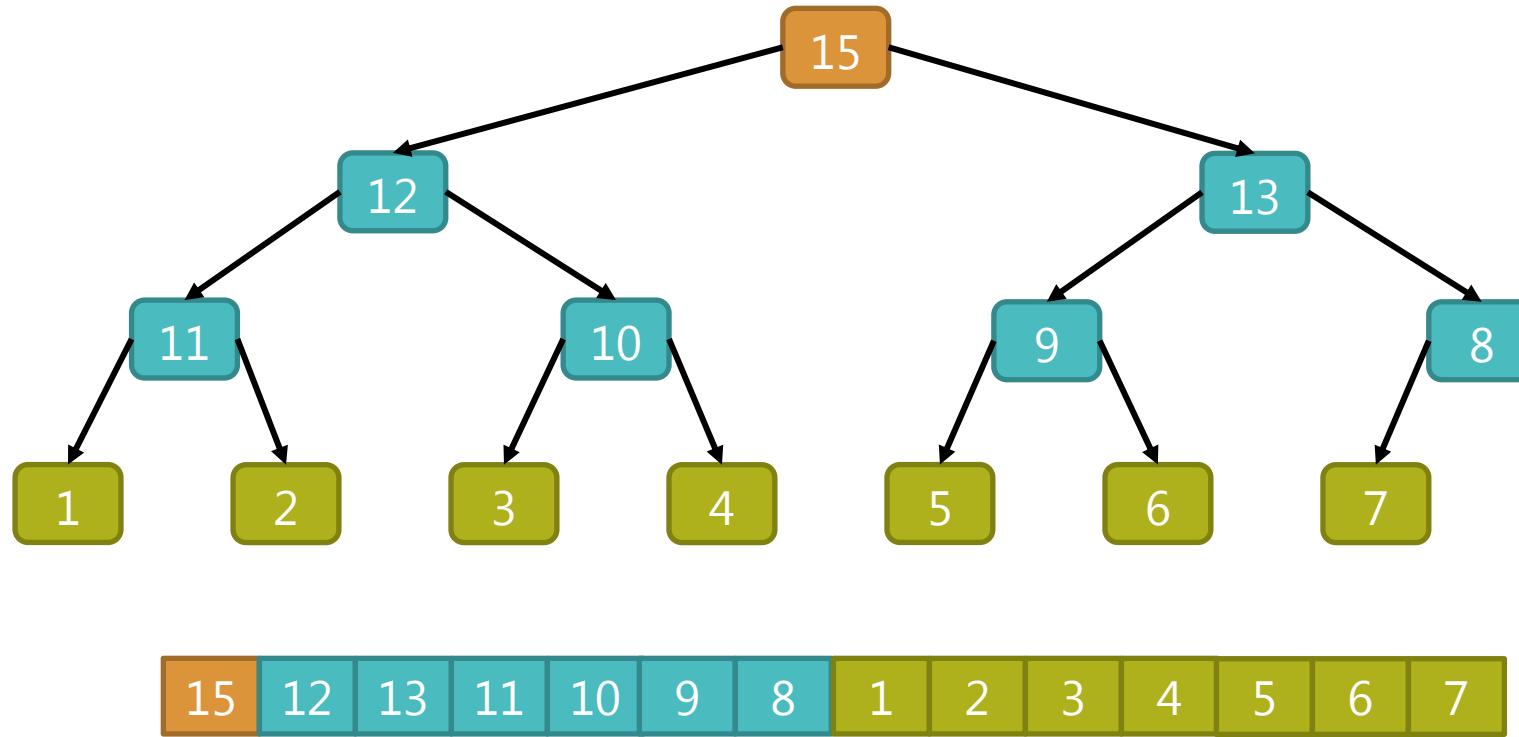
ARBRES BINAIRES

Organisation en *tas-min* (*min-heap*)



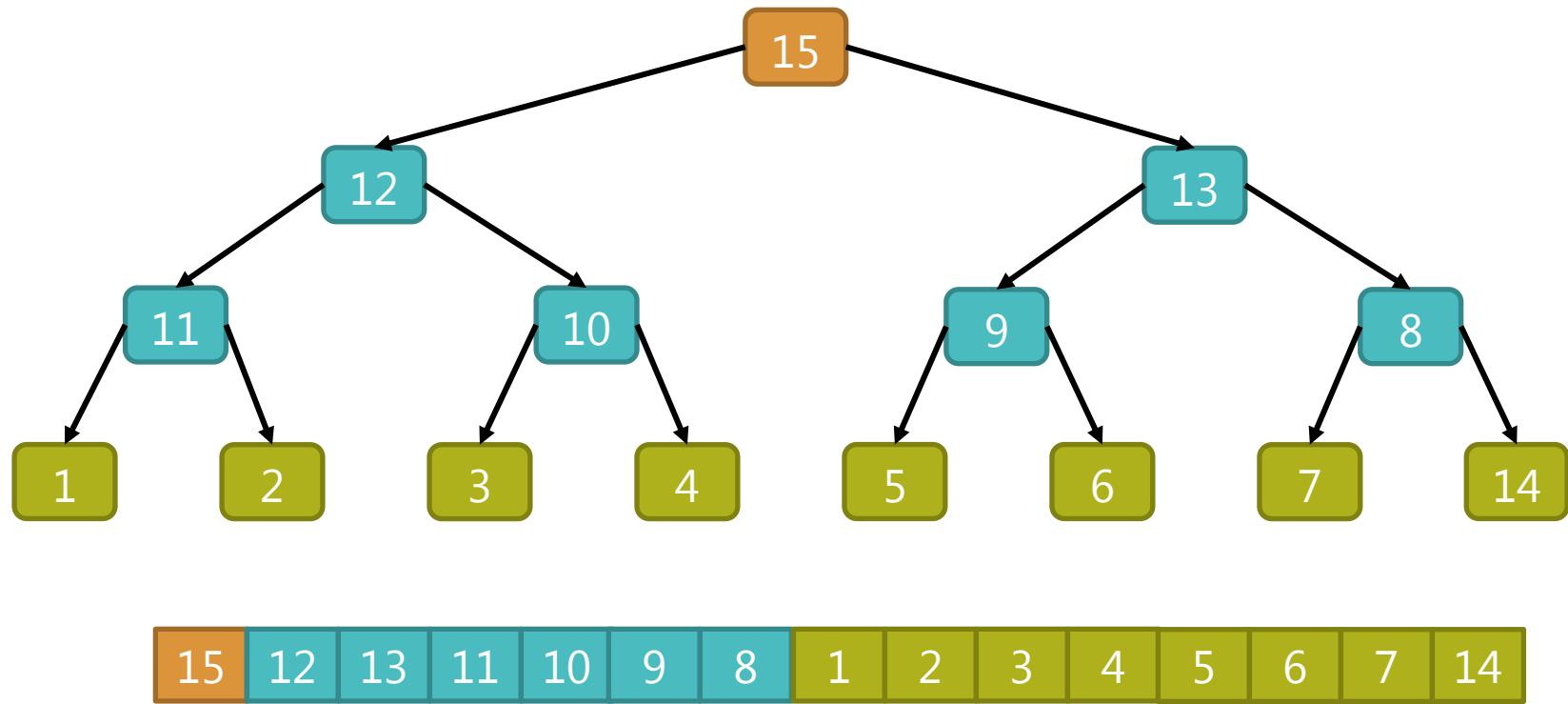
ARBRES BINAIRES

Organisation en *tas-max* (*max-heap*)



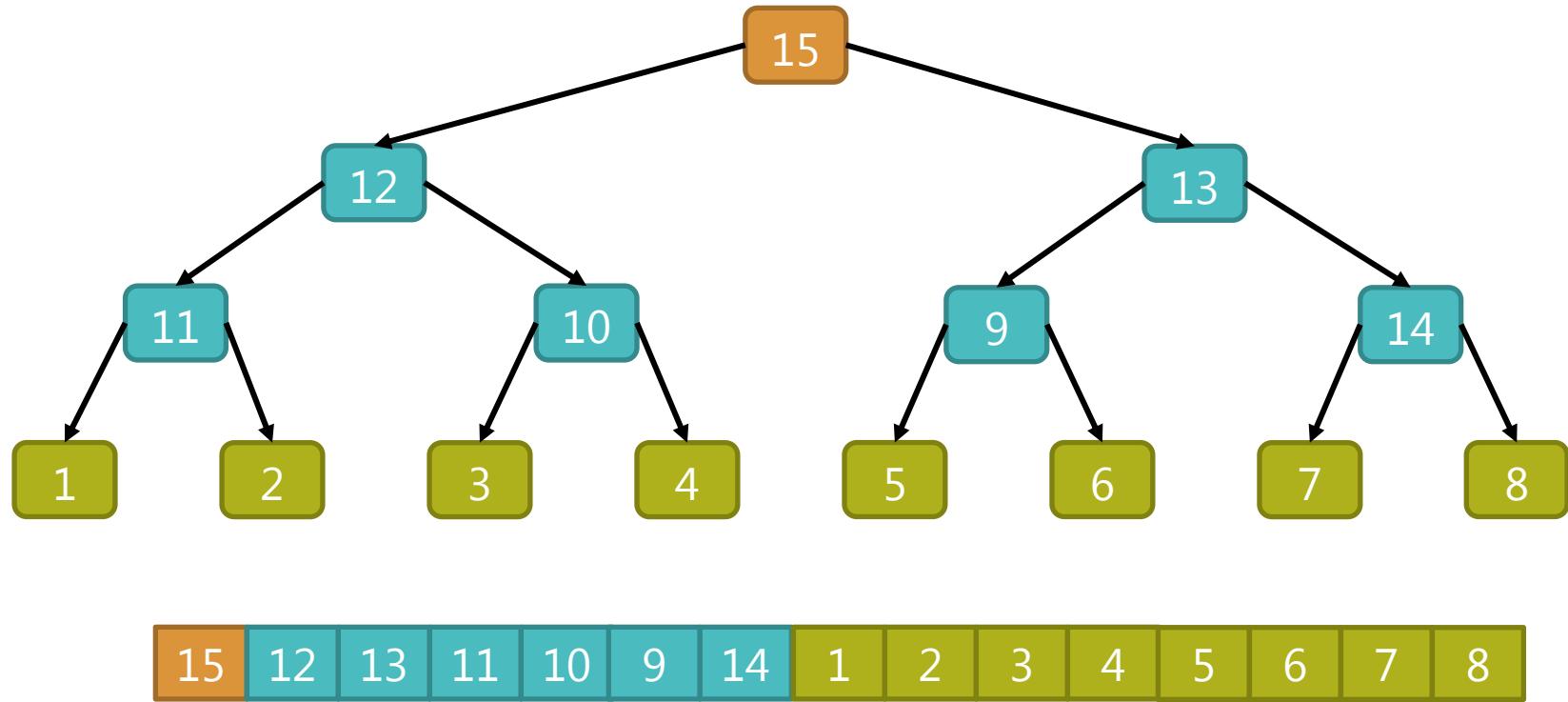
ARBRES BINAIRES

Ajout d'un élément : on l'ajoute en tant que feuille



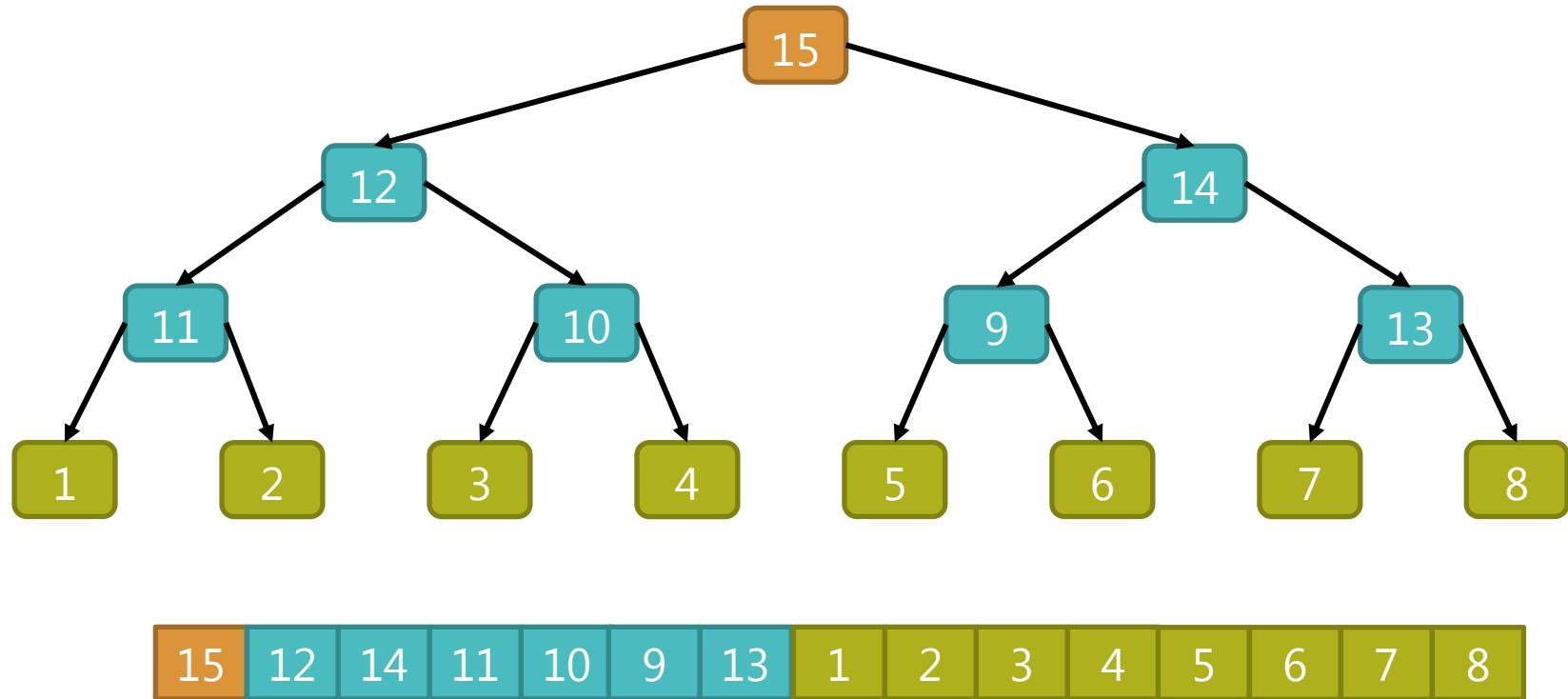
ARBRES BINAIRES

Ajout d'un élément : s'il est $>$ à son parent, on l'échange avec celui-ci



ARBRES BINAIRES

Ajout d'un élément : on recommence jusqu'à retrouver une structure de tas



ARBRES BINAIRES

Tas et file de priorité

- Le 1^{er} élément du tas (la racine) est toujours le plus grand.
- Cette propriété du tas le rend idéal pour implémenter une *file de priorité*.
- Lorsqu'on retire le 1^{er} élément, il faut réorganiser le tas : si l'élément suivant est inférieur à au moins un de ses fils, on l'échange avec son fils le plus grand. On recommence l'opération jusqu'à ce que l'élément soit enfin à sa place.
- Cette opération s'appelle le *tamisage* ou la *percolation*. Elle est à la base du *tri par tas (heapsort)*.

ARBRES BINAIRES

Tri pas tas

- On commence par insérer les éléments du tableau dans une structure de tas

6	2	5	1	8	3	7	4
---	---	---	---	---	---	---	---

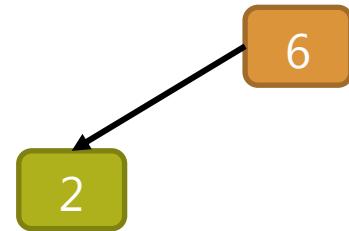
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (1/12)



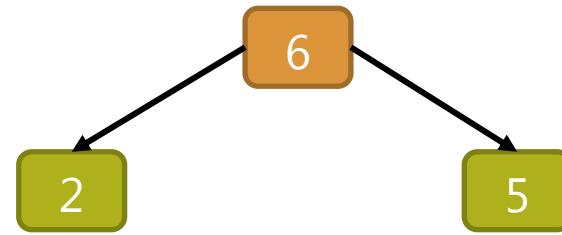
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (2/12)



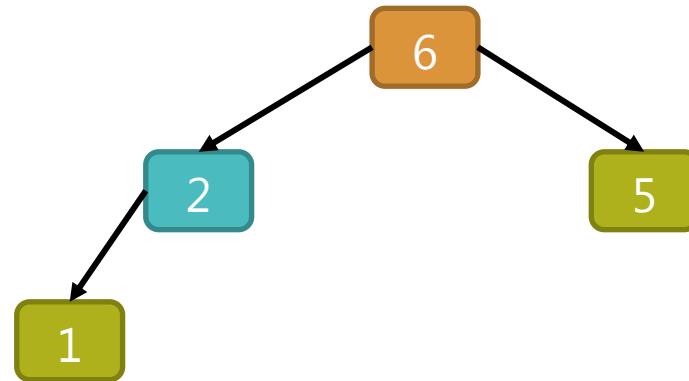
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (3/12)



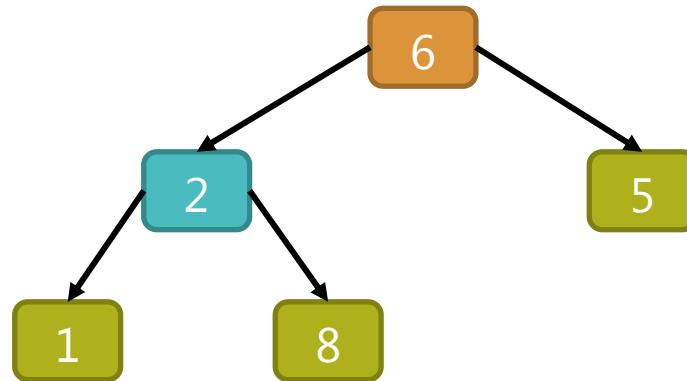
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (4/12)



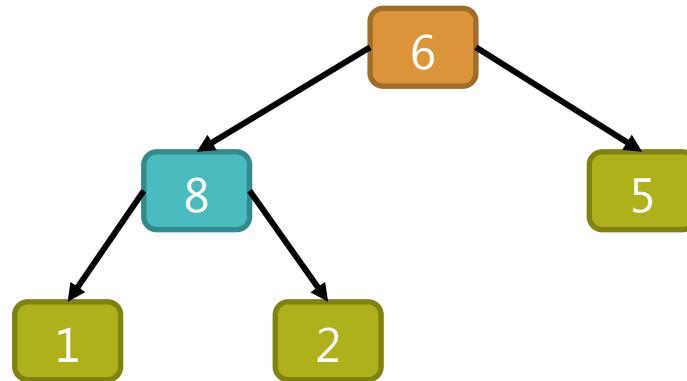
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (5/12)



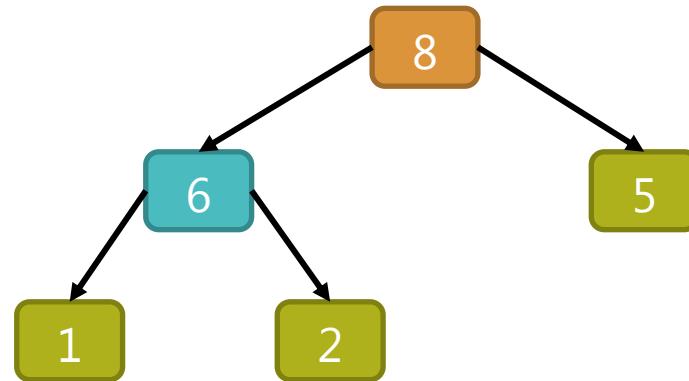
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (6/12)



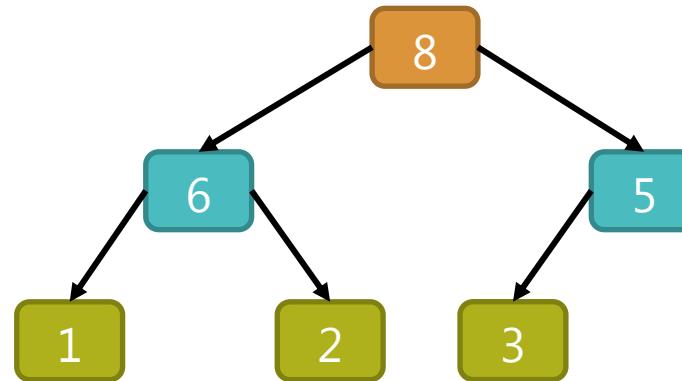
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (7/12)



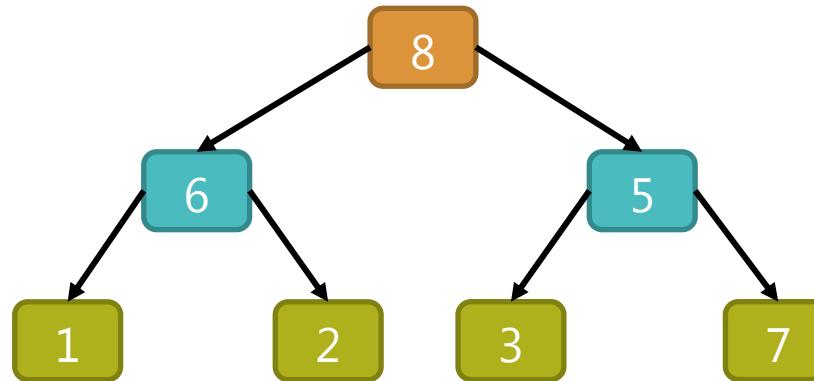
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (8/12)



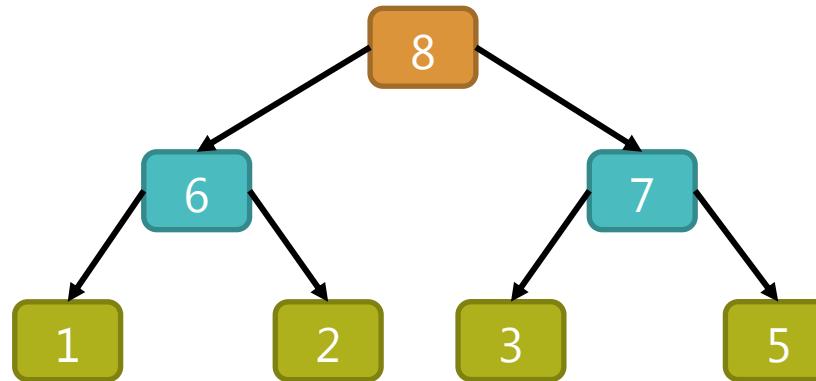
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (9/12)



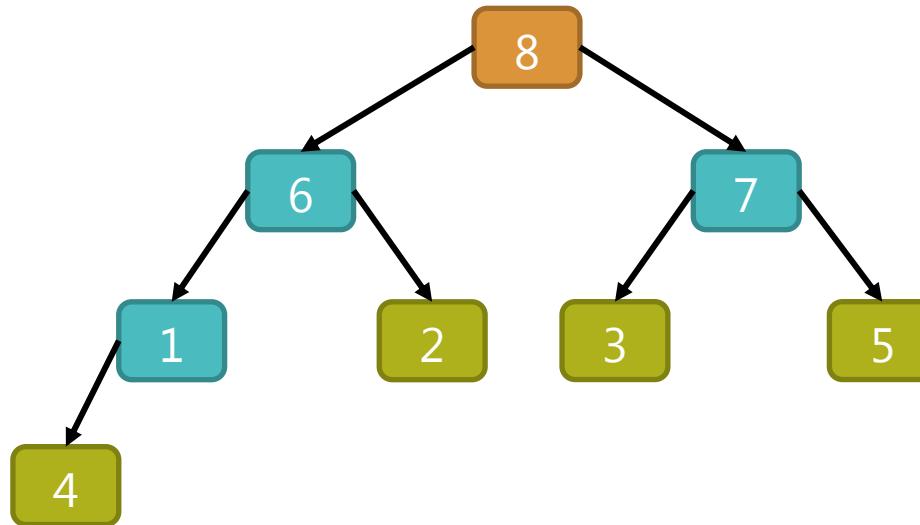
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (10/12)



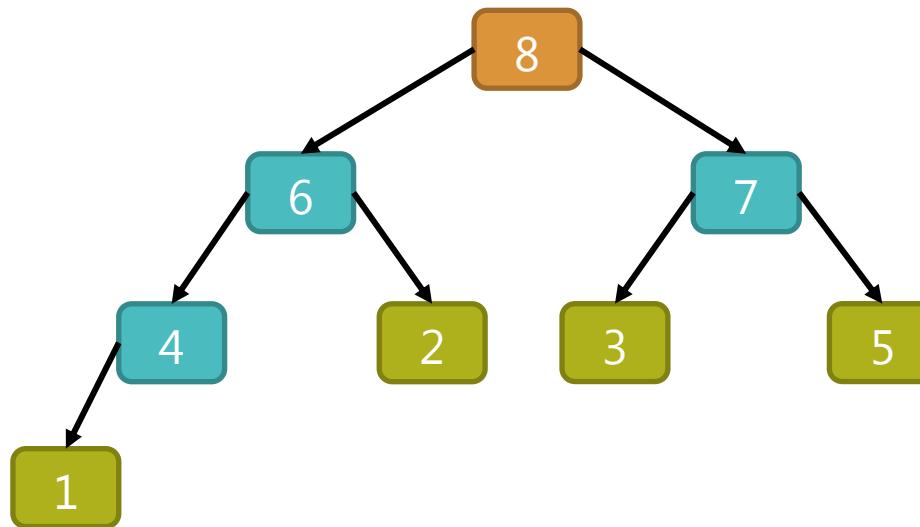
ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (11/12)



ARBRES BINAIRES

Tri pas tas : insertion des éléments dans le tas (12/12)



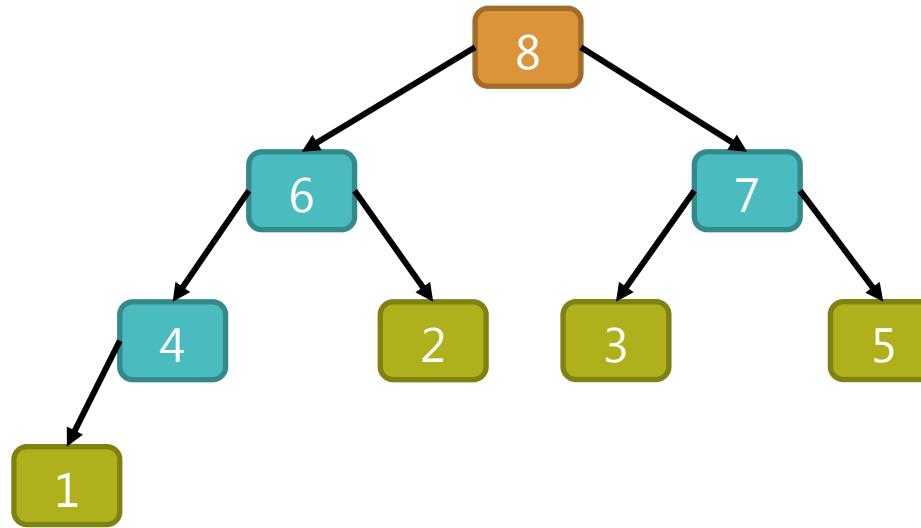
ARBRES BINAIRES

Tamisage

- A la fin de l'insertion dans le tas, les éléments ne sont pas pour autant triés : on sait seulement que la racine est l'élément de plus grand de la collection.
- On échange donc le dernier élément du tableau avec la racine (1^{er} élément du tableau) qui se retrouve de fait à sa place définitive. Si on considère maintenant le tableau des éléments restants, on a une structure de tas dont seul la racine est éventuellement mal placée.
- On utilise l'opération de tamisage qui consiste à échanger la racine avec le plus grand de ses fils jusqu'à ce qu'elle soit à sa place.
- On recommence ensuite l'opération jusqu'à ce que toute la collection soit triée.

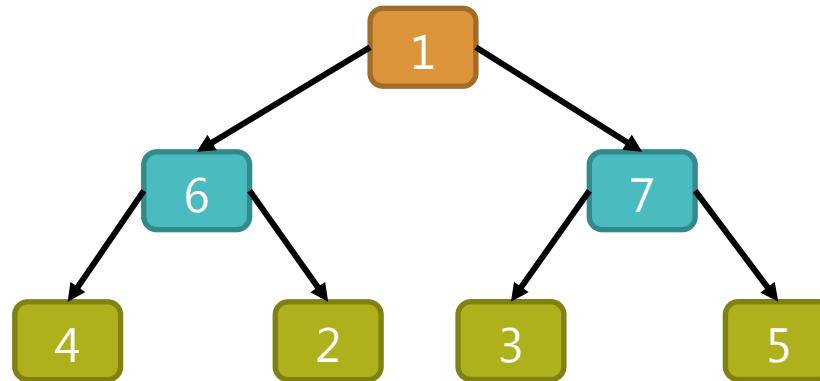
ARBRES BINAIRES

Tri pas tas : tamisages successifs (1/17)



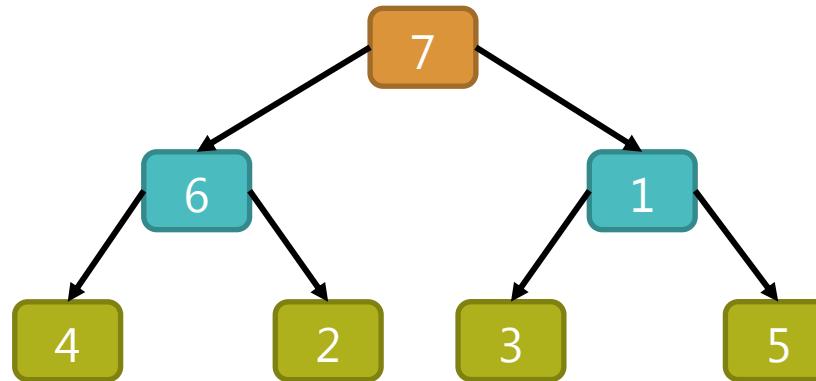
ARBRES BINAIRES

Tri pas tas : tamisages successifs (2/17)



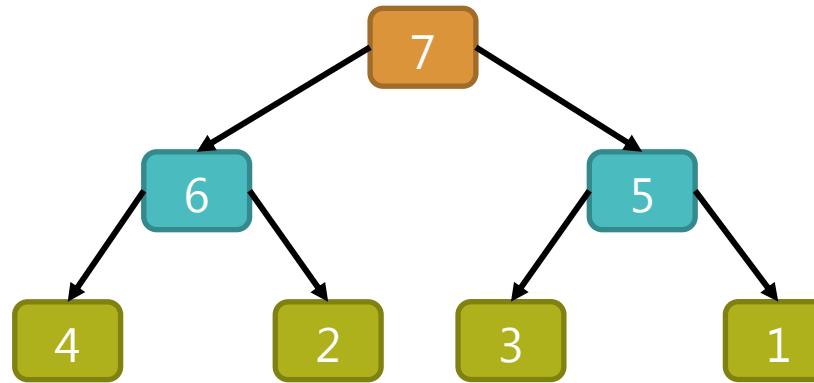
ARBRES BINAIRES

Tri pas tas : tamisages successifs (3/17)



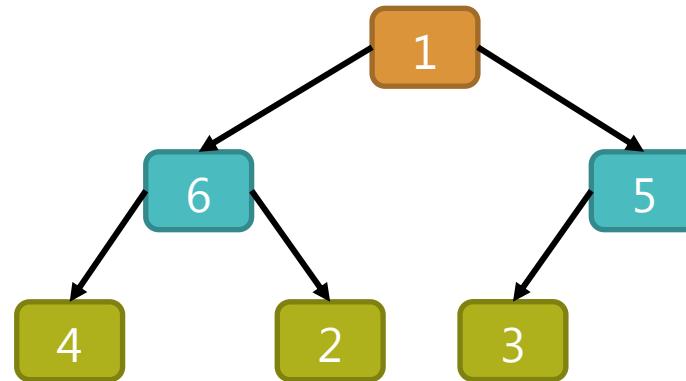
ARBRES BINAIRES

Tri pas tas : tamisages successifs (4/17)



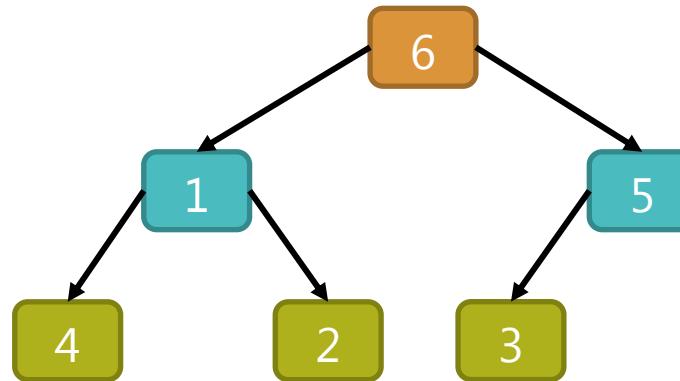
ARBRES BINAIRES

Tri pas tas : tamisages successifs (5/17)



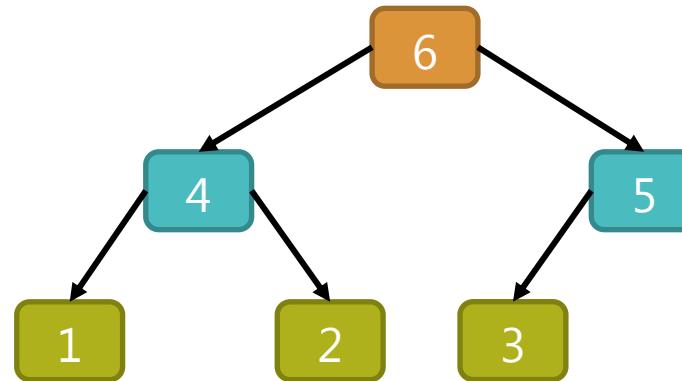
ARBRES BINAIRES

Tri pas tas : tamisages successifs (6/17)



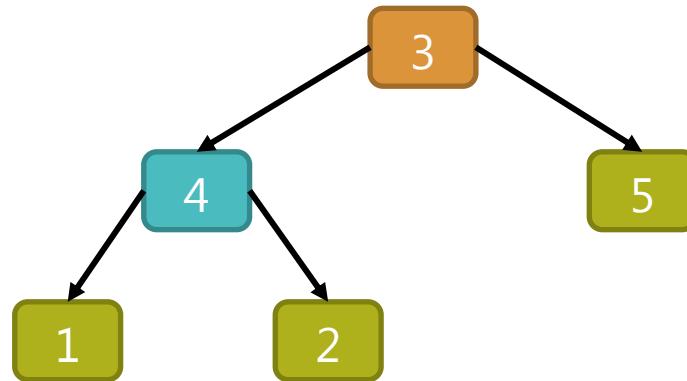
ARBRES BINAIRES

Tri pas tas : tamisages successifs (7/17)



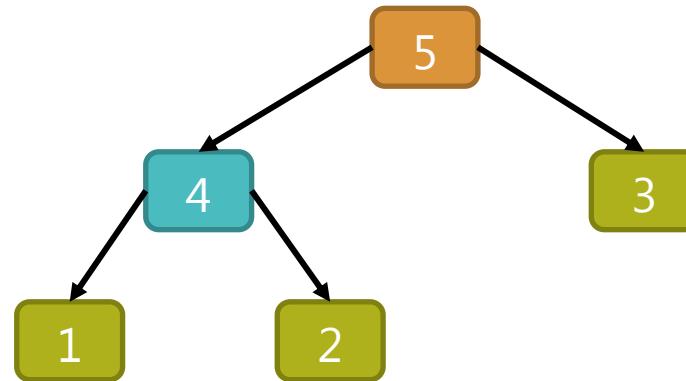
ARBRES BINAIRES

Tri pas tas : tamisages successifs (8/17)



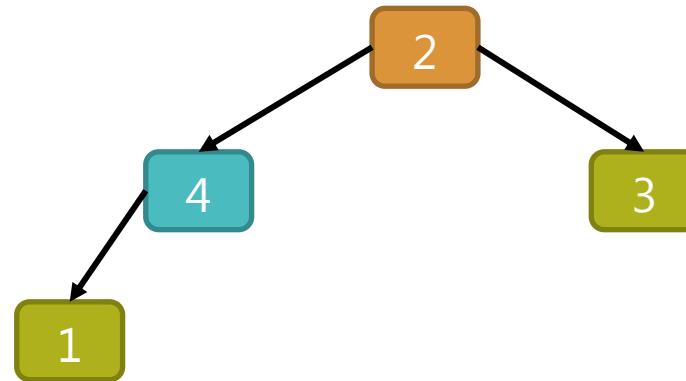
ARBRES BINAIRES

Tri pas tas : tamisages successifs (9/17)



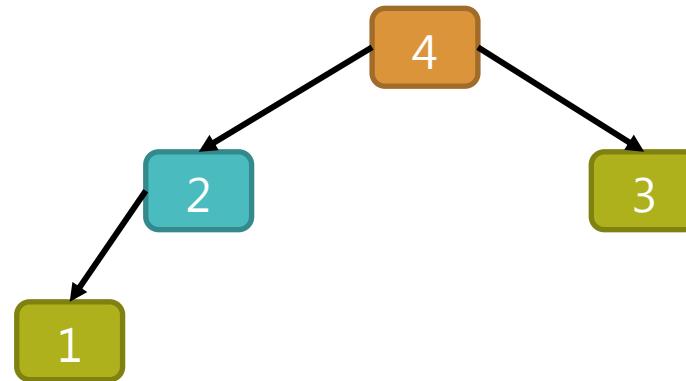
ARBRES BINAIRES

Tri pas tas : tamisages successifs (10/17)



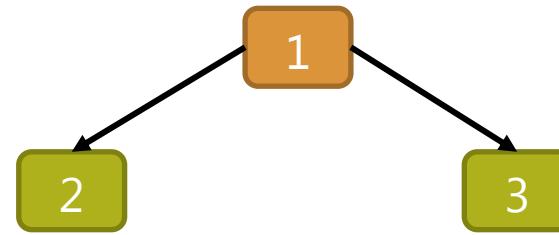
ARBRES BINAIRES

Tri pas tas : tamisages successifs (11/17)



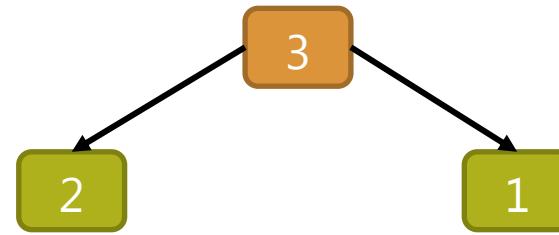
ARBRES BINAIRES

Tri pas tas : tamisages successifs (12/17)



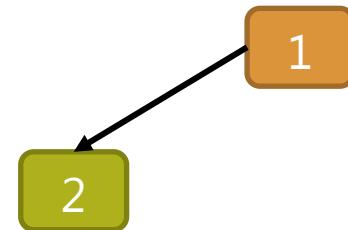
ARBRES BINAIRES

Tri pas tas : tamisages successifs (13/17)



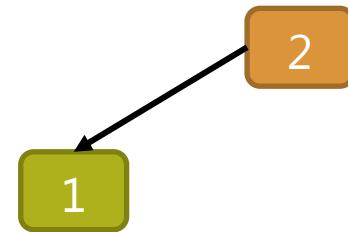
ARBRES BINAIRES

Tri pas tas : tamisages successifs (14/17)



ARBRES BINAIRES

Tri pas tas : tamisages successifs (15/17)



ARBRES BINAIRES

Tri pas tas : tamisages successifs (16/17)

1

1 2 3 4 5 6 7 8

ARBRES BINAIRES

Tri pas tas : tamisages successifs (17/17)



ARBRES BINAIRES

Optimisations du tri par tas

- La complexité du tri par tas est en $O(n \log(n))$
- Si les données sont déjà triées, l'algorithme les mélange d'abord dans la 1^{ère} phase avant de les retrier dans la phase de tamisages successifs. Le *smoothsort* permet d'éviter cet inconvénient.
- À la fin du tri, l'algorithme effectue plusieurs fois de suite les mêmes inversions, ce qui est inutile. On peut plutôt arrêter l'algorithme quand il reste peu d'éléments puis effectuer un tri par insertion.



MERCI

Jean-Jacques Schwartzmann

Maître de conférences associé

jean-jacques.schwartzmann@ensicaen.fr



L'École des INGÉNIEURS Scientifiques

