Algorithmique et langage C

Spécialité ELEC 1A – 2018-2019

Jean-Jacques Schwartzmann

jean-jacques.schwartzmann@ensicaen.fr

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

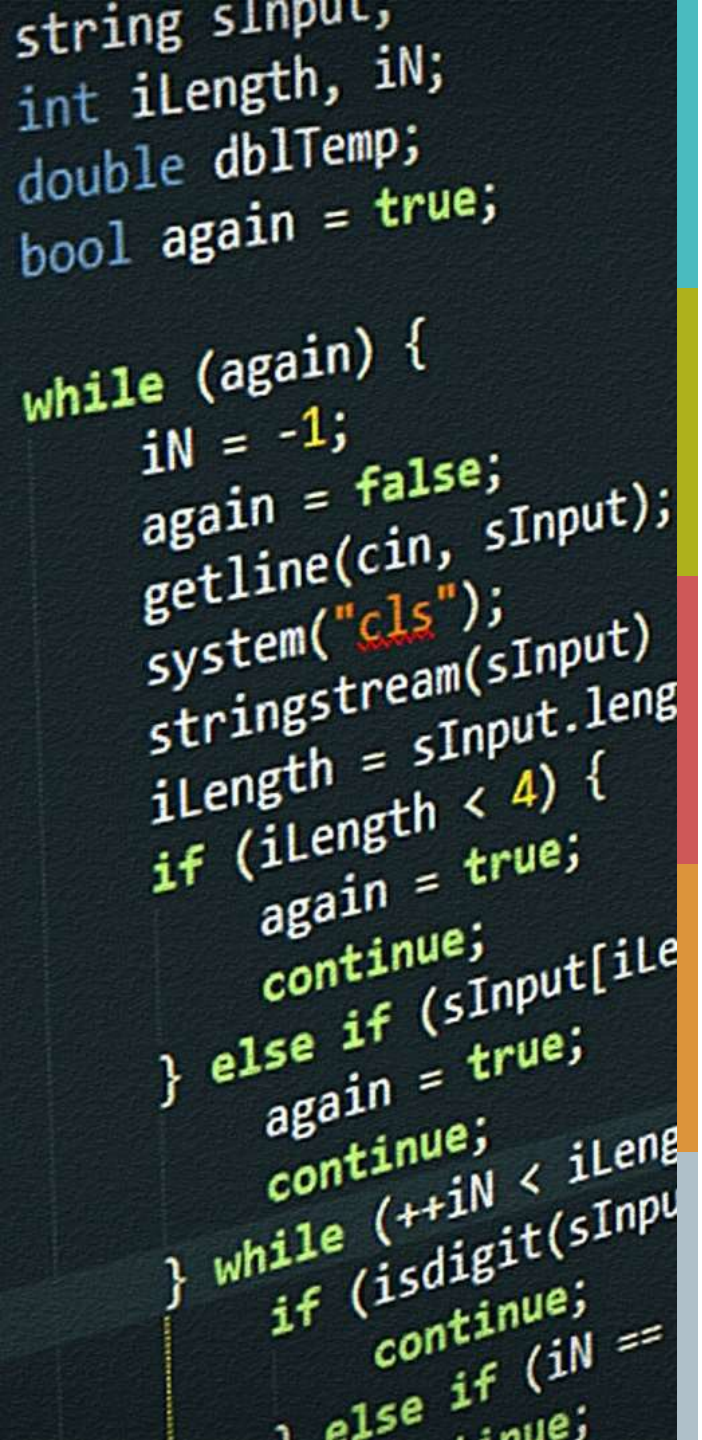
L'École des INGÉNIEURS Scientifiques

OBJECTIFS DU COURS

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Mise en œuvre d'algorithmes et structures de données appropriés en langage C

Connaître les structures de données et algorithmes de base en programmation



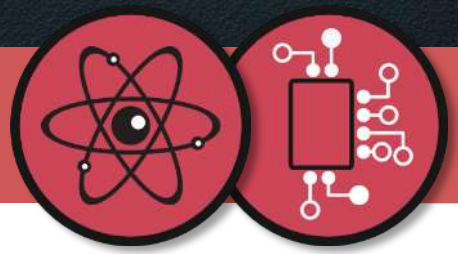
PLAN DU COURS

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

- Rappels et approfondissements en C
- 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;
    while (++iN < iLength) {
        isdigit(sInput[iN])) {
                                3) ) {
```

Rappels et approfondissements en C



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Tableaux, chaines, pointeurs - Tableaux et fonctions

Pointeurs et fonctions génériques - Allocation mémoire

Organisation de la mémoire - Pointeurs sur fonctions – Récursivité

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*sizeof(int) octets consécutifs.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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*)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

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 connaitre la taille des données *à priori* (mais on peut toutefois utiliser des pointeurs et *allouer 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*.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

Chaines de caractères

- Une chaine 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*.

Initialisation d'une chaine de caractères

- On peut initialiser une chaine avec une chaine 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 chaine dans une autre.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Organisation d'une chaine en mémoire

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

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

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.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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 0xbfecec84
```

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
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
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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
$
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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
```

Tableaux et fonctions

- Il est possible de passer un tableau comme argument à une fonction.
- La fonction ne connait 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");
}

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

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

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 chaines 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);`

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 !
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

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
```

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);`

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 = %.2lf, x = %.2lf\n", x, y);
  permuter(&a, &b, sizeof(a));
  permuter(&x, &y, sizeof(x));
  printf("a = %d, b = %d\n", a, b);
  printf("x = %.2lf, x = %.2lf\n", x, y);
}
```

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

Allocation dynamique de pointeurs

- Limitation de l'utilisation des tableaux
  - Besoin de connaitre *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

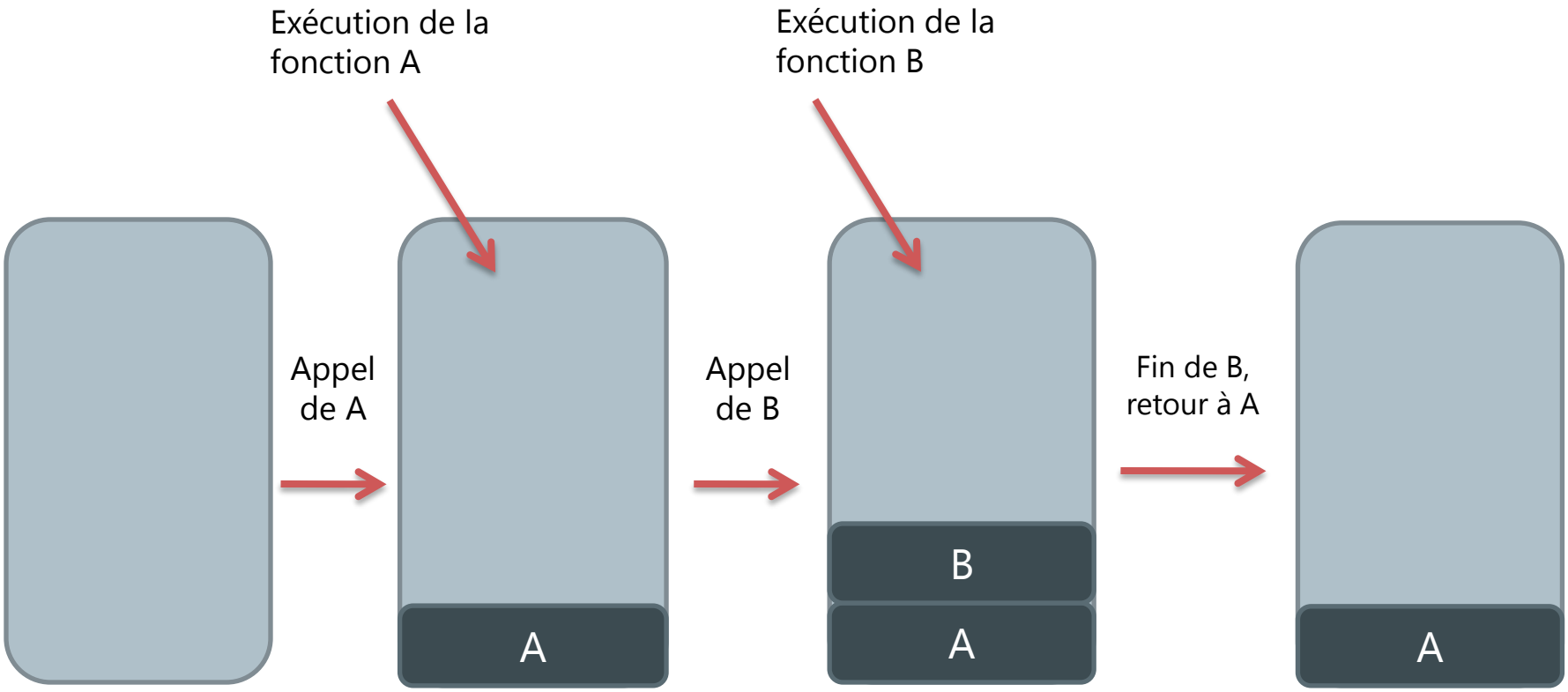
Fonctions d'allocation dynamique

- L'allocation dynamique se gère à partir de 4 fonctions :
  - *malloc*
  - *calloc*
  - *realloc*
  - *free*

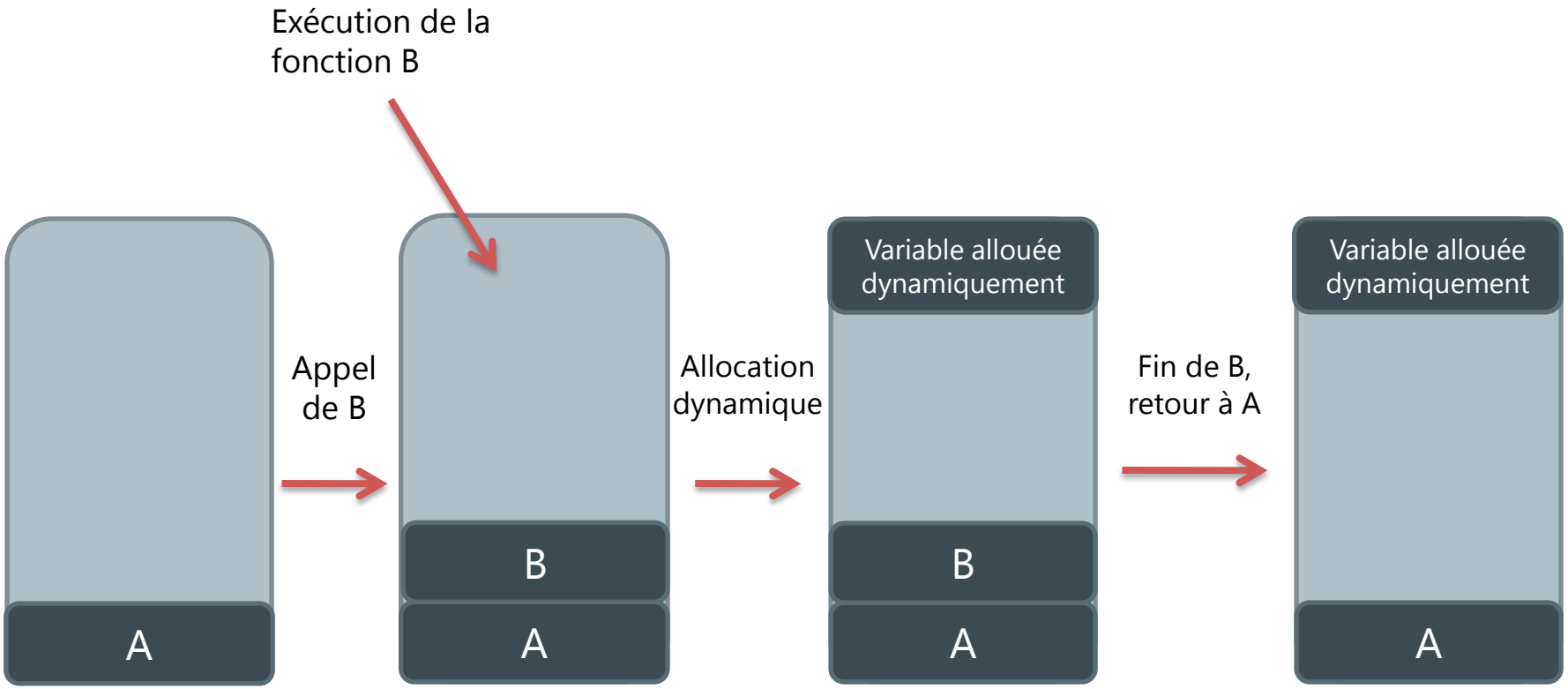
- Les variables crées avec l'allocation dynamique sont crées dans un segment mémoire appelé le tas (heap)

- Ces fonctions sont accessibles depuis la bibliothèque stdlib :
  - `#include <stdlib.h>`

Pile d'exécution (stack)



Pile et tas


Exécution de la fonction B
Appel de B
Allocation dynamique
Fin de B, retour à A
Variable allouée dynamiquement
Variable allouée dynamiquement
A
B
A
B
A
A

Fonction malloc

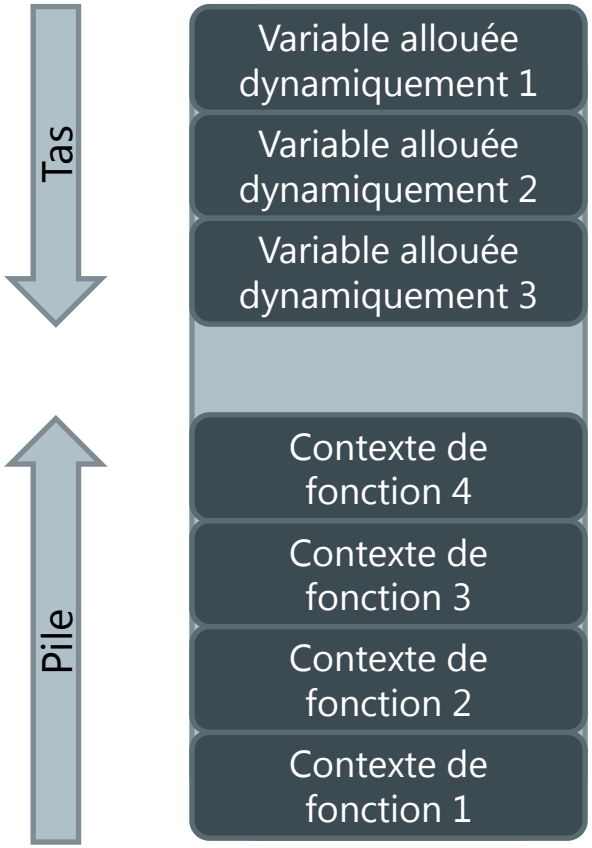
- La création d'une variable / d'un tableau dynamique se fait par l'utilisation de la fonction *malloc*.

- 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>));`

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 !



Gestion des erreurs d'allocation mémoire : exemple de code

```c
#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;
}
```

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));`

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)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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 d'accéder à cette zone mémoire, ni de la libérer → fuite de mémoire !
- *pi* et *pj* pointent vers la même zone mémoire, les deux free libèrent donc la même zone mémoire !

  → segmentation fault !

```c
#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);
}
```

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 réserver, 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.

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));
}
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Erreur courante dans l'allocation de pointeurs passés en arguments

```c
#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)
```

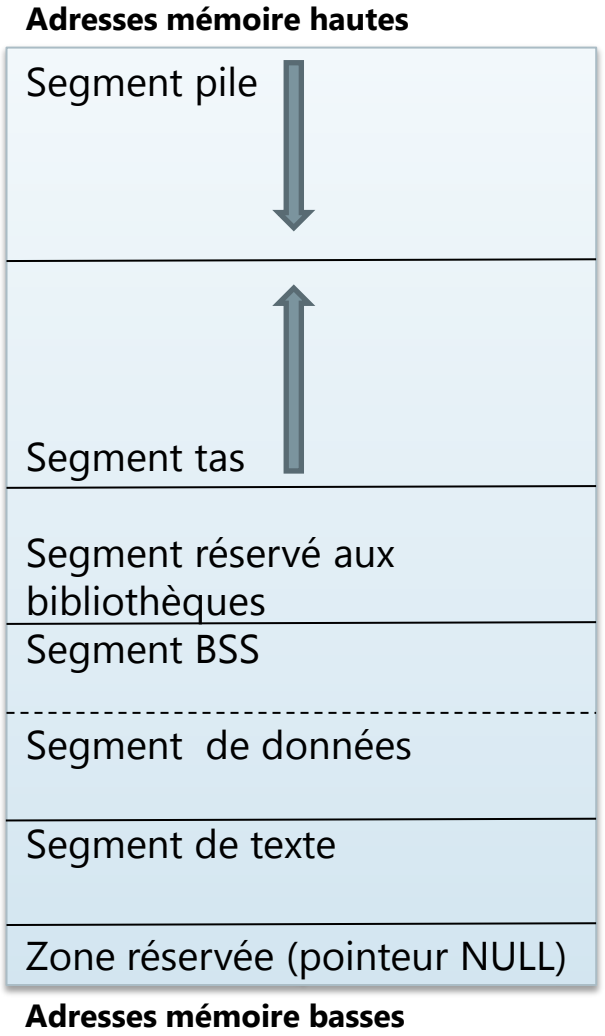
Allocation de pointeurs passés comme arguments dans une fonction

- 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;
}
```

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



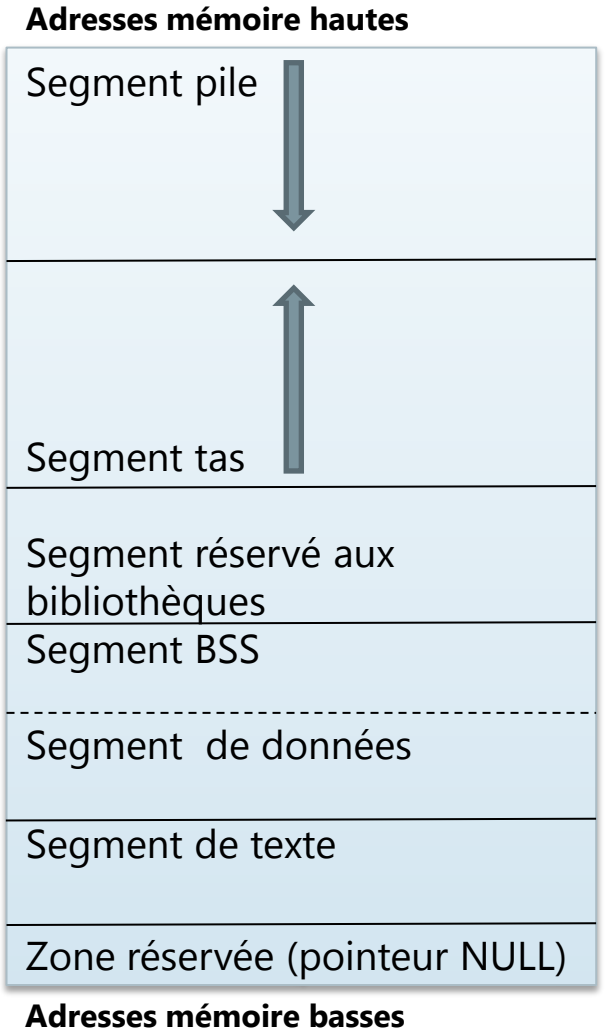
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

Organisation des variables en mémoire

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



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*.

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);
}
```

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);
}
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Exemple complet

```c
#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
$
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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 chaines de caractères.

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

```c
#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 ]
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

Fonction factorielle : algorithme itératif

- La factorielle d'un entier positif *n* est définie par :

  $n ! = n \times (n - 1) \times (n - 2) \times \ldots \times 2 \times 1$, avec par 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;
}
```

Fonction factorielle : algorithme récursif

- On peut aussi remarquer que $(n-1)! = (n-1) \times (n-2) \times \ldots \times 2 \times 1$
- donc on peut ainsi définir la factorielle de n par :

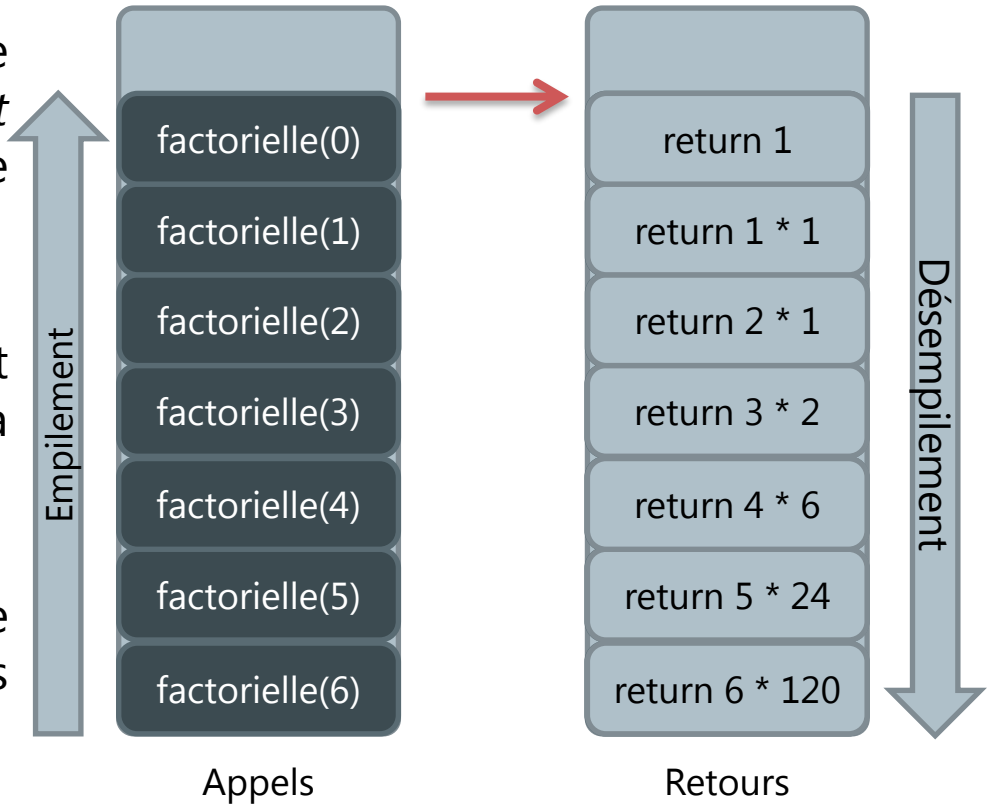
$$n ! = n \times (n-1)!$$

- d'où le code récursif suivant :

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

Pile de récursion

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



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Suite de Fibonacci

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

$$\mathcal{F}_{n+2} = \mathcal{F}_{n+1} + \mathcal{F}_n \text{ avec } \mathcal{F}_0 = 0 \text{ et } \mathcal{F}_1 = 1$$

- 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);
}
```

Récursivité 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);
}
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Intérêt de la récursivité terminale

- La récursivité terminale permet d'éliminer la phase de désempilement, puisque le dernier appel permet d'obtenir directement la valeur à calculer, contrairement à la récursivité simple ou 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'êtres 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.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

```c
#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 NUMELEMS\n", argv[0]);
    return EXIT_FAILURE;
  }
  unsigned long N = atoi(argv[1]);
  if (N < 1) {
    printf("Usage : %s NUMELEMS\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,
```

Récursivité croisée

- On parle de *récursion mutuelle*, ou *récursivité 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);
}
```

Autre exemple (sans aucun intérêt pratique) de récursivité croisé

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

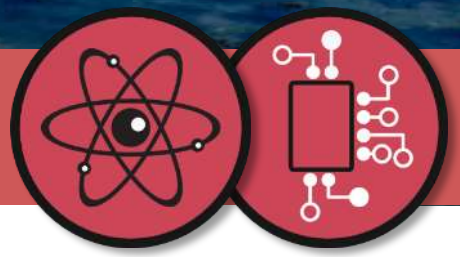
```
bool is_odd(unsigned long);

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

- Rappels et approfondissements en C
- Listes chainées
- Piles et files
- Algorithmes de tri
- Tables de hachage
- Arbres binaires

Listes chainées



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Types abstrait de données

Implémentation d'une liste simple avec un pointeur – avec d'une structure

Listes doublement chainées

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 et de modifier 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*.

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 x type2 x … x typeN → type1' x type2' x … x 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 x int → TPoint
GetX : TPoint → int
GetY : TPoint → int
MoveTo: TPoint x int x 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

Parallèle avec la 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,

- 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

Règles d'écriture en C (conventions propres à ce cours)

1. Toutes les fonctions implémentant les opérations associées au TAD doivent faire référence à une structure C (struct) qui représente l'objet à manipuler : concrètement, elles doivent toujours recevoir un pointeur sur la structure comme premier paramètre que l'on nommera *this* par convention.
2. Pour éviter les redondances des noms de fonctions et donc les confusions, on préfixe leur nom du type de la structure (Type_).
3. 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 est l'équivalent du *constructeur* en POO, c'est elle qui crée l'objet et doit donc être appelée avant toute utilisation d'un objet.
4. La mémoire allouée par le constructeur sera libérée par la fonction *Type_Delete* quand l'objet ne sera plus utile. Cette fonction est l'équivalent du *destructeur* en POO
5. Chaque TAD doit être déclaré dans un fichier d'en-tête spécifique, et les fonctions correspondant aux opérations associées doivent être implémentées dans un fichier source spécifique.

Réutilisabilité de code : le type *TElement*

- Afin de produire du code réutilisable, on définira dans un fichier d'en-tête *element.h* le type *TElement*, qui redéfinit le type des éléments du TAD :

```
typedef int TElement;
```

```
typedef struct etudiant {
  char nom[25];
  char prenom[25];
  int age;
} TElement;
```

LES LISTES CHAINÉES

Définition d'une liste chaînée

- 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 être alors 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 x int → TNode
InsertFirst : TList x TElement → TNode x TList
RemoveFirst : TList → bool x TList
Add : TList x TElement → TNode x TList
RemoveLast : TList → bool
Display : TList →
Delete : TList → TList
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

```
#ifndef _LIST_H
#define _LIST_H

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

typedef struct Node {                          /* Déclaration de la structure Noeud */
  TElement Element;
  struct Node *Next;
} TNode;                                       /* Type Noeud */
typedef TNode *PTNode;                         /* Type pointeur vers 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
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

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

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

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

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

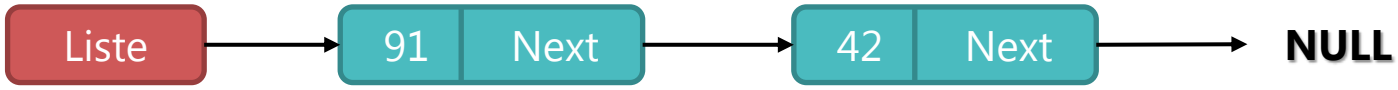
Fontion TList_Goto : aller à une position donnée dans la liste

- 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

```c
#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;
}
```

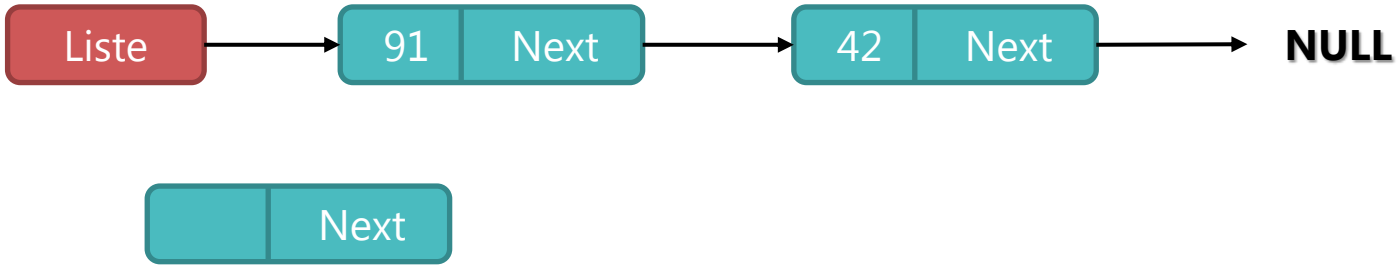
LES LISTES CHAINÉES

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



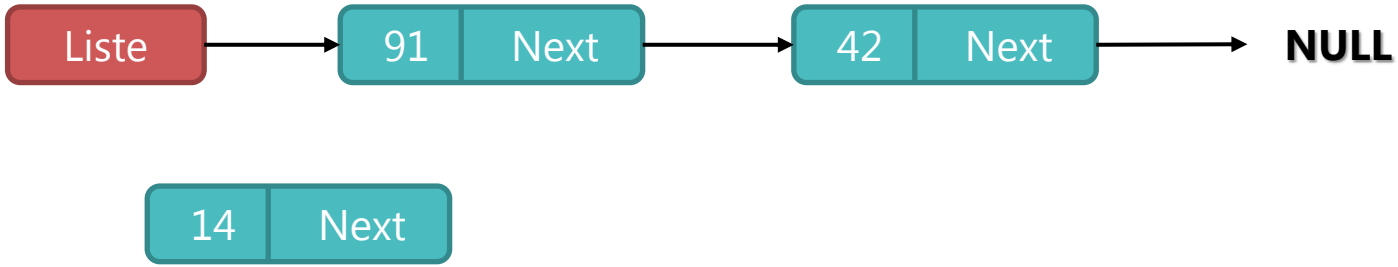

```
PTNode TList_InsertFirst(PTList this, TElement Element)
{
```

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



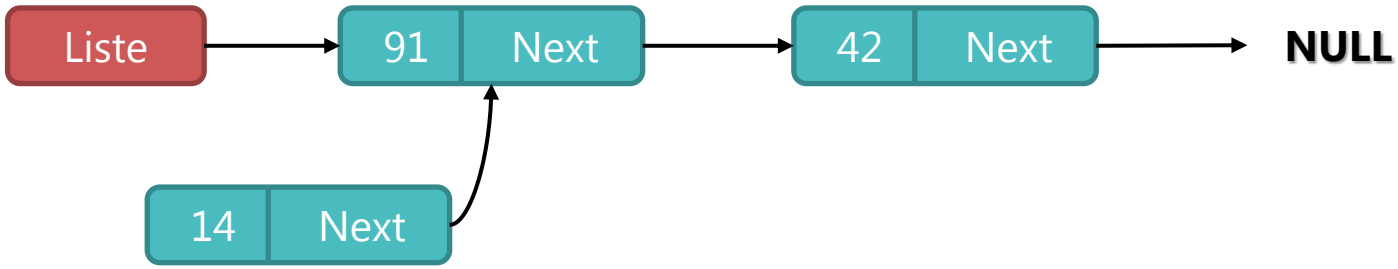

```
PTNode TList_InsertFirst(PTList this, TElement Element)
{
  PTNode newNode = (PTNode)malloc(sizeof(TNode));
  if (newNode == NULL) return NULL;
```

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



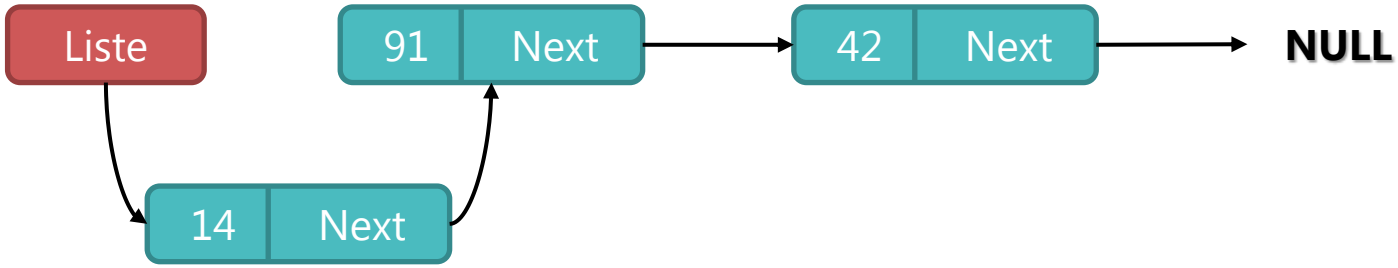

```
PTNode TList_InsertFirst(PTList this, TElement Element)
{
  PTNode newNode = (PTNode)malloc(sizeof(TNode));
  if (newNode == NULL) return NULL;
  newNode->Element = Element;
```

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;
```

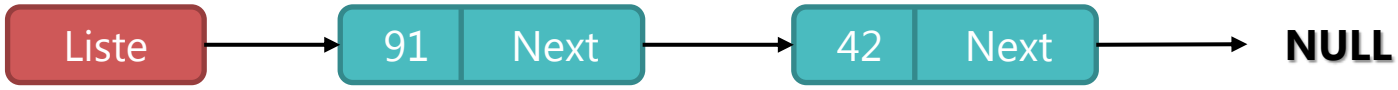
Fonction TList_InsertFirst : insertion d'un élément en début de liste (5/5)




```
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;
}
```

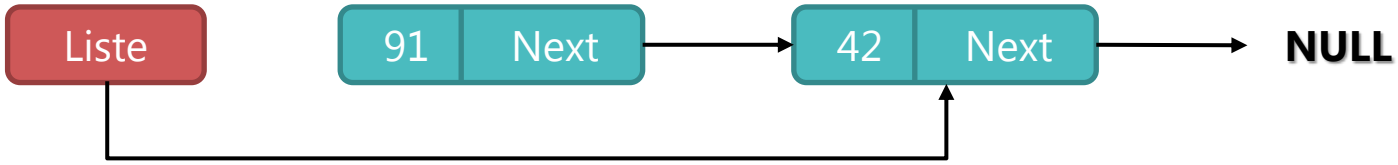
LES LISTES CHAINÉES

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;
```

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



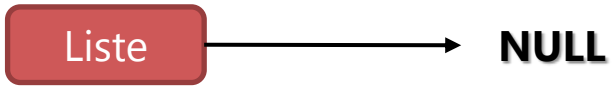

```
bool TList_RemoveFirst(PTList this)
{
  if (TList_IsEmpty(this)) return false;
  PTNode pNode = *this, nextNode = pNode->Next;
  *this = nextNode;
```

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;
}
```

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

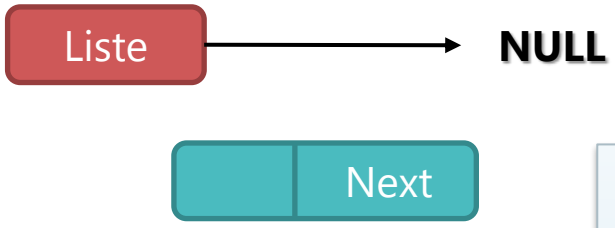
1) Liste vide




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

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

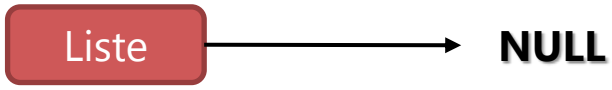
1) Liste vide




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

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

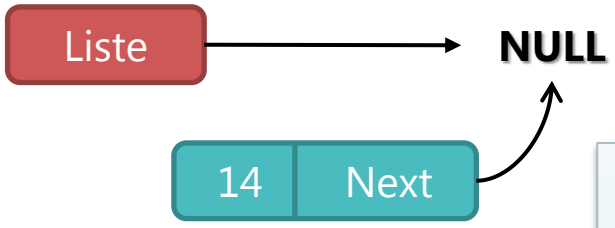
1) Liste vide




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

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

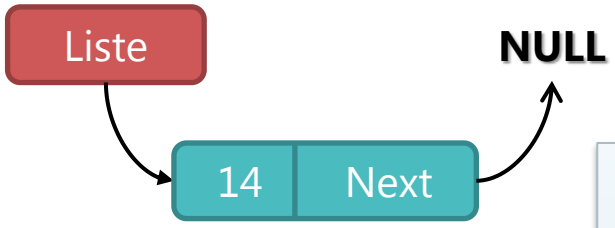
1) Liste vide




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

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

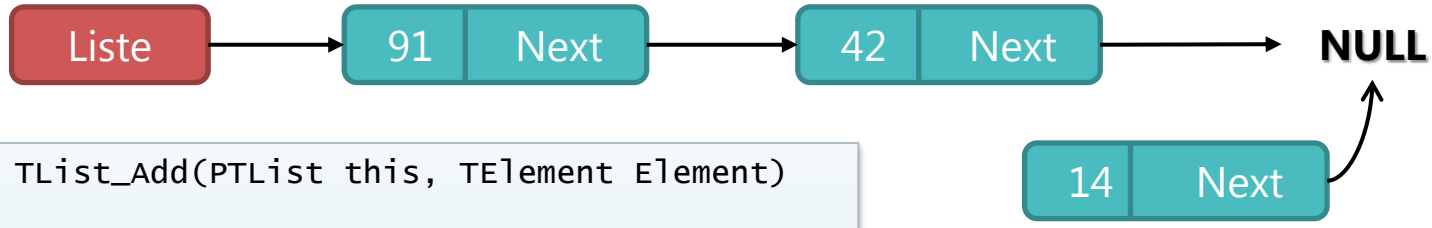
1) Liste vide : similaire à l'insertion en début de liste




```
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;
```

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

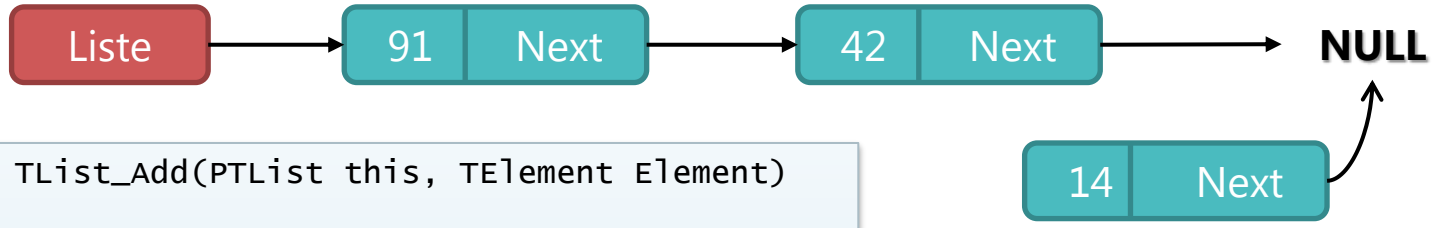
2) Liste non vide




```
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 {
```

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

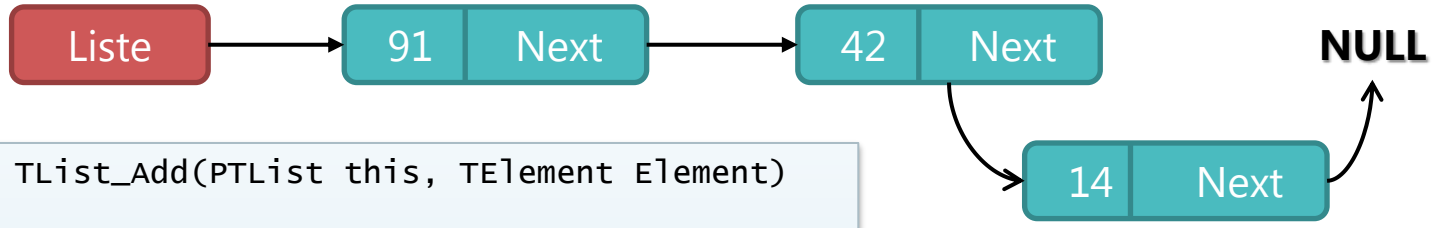
2) Liste non vide : avancer jusqu'au dernier élément




```
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;
```

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;
}
```

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;
```

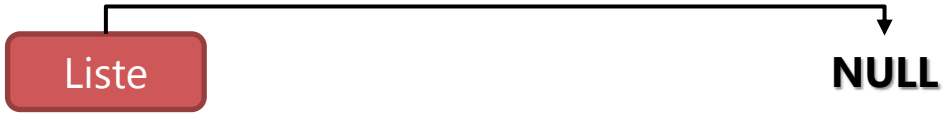
Fonction TList_RemoveLast : retrait d'un élément en fin de liste (2/6)

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;
```

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

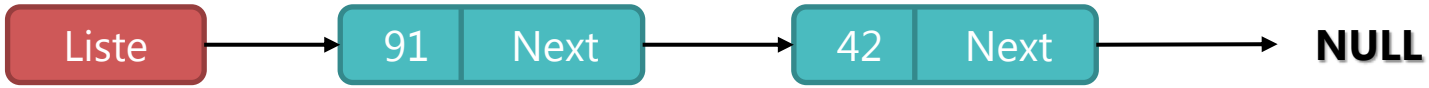
1) Un seul élément dans la liste




```
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);
  }
```

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;
    }
```

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

2) Plus d'un élément dans la liste

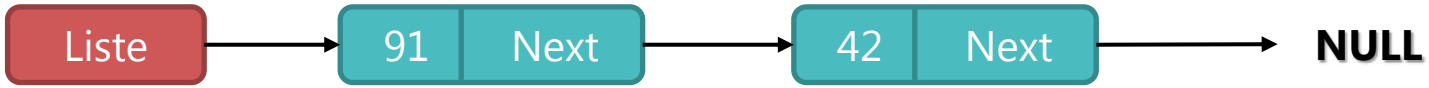

```
bool TList_RemoveLast(PTList this)
{
  if (TList_IsEmpty(this)) return false;
  PTNode prevNode = *this, pNode = prevNode->Next;
  if (pNode == NULL) {  /* 1 seul élément dans la liste */
    *this = NULL;
    free(prevNode);
  }
  else {
    while (pNode->Next) {
      prevNode = pNode;
      pNode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

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

2) Plus d'un élément dans la liste

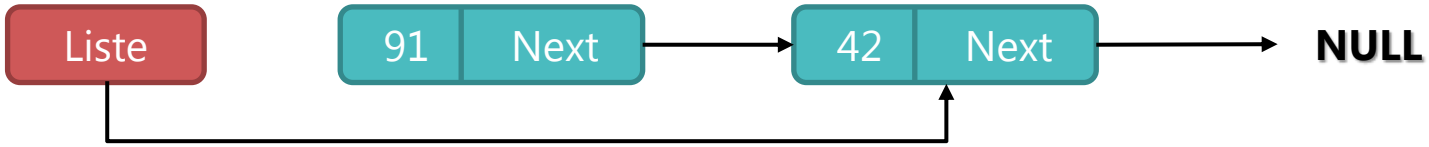

```
bool TList_RemoveLast(PTList this)
{
  if (TList_IsEmpty(this)) return false;
  PTNode prevNode = *this, pNode = prevNode->Next;
  if (pNode == NULL) {  /* 1 seul élément dans la liste */
    *this = NULL;
    free(prevNode);
  }
  else {
    while (pNode->Next) {
      prevNode = pNode;
      pNode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

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




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

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




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

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


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

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

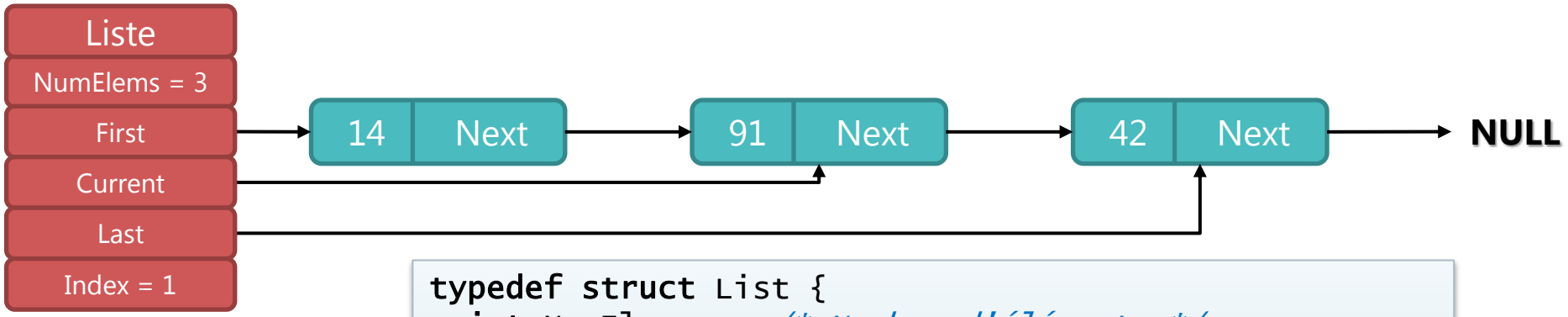

```
void TList_Delete(PTList this)
{
  while (TList_RemoveFirst(this));
}
```

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


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

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;
```

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 →
```

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

```
#ifndef _LIST_H
#define _LIST_H

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

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

typedef TNode *PTNode;                   /* Type pointeur vers Noeud */

typedef struct List {                    /* Déclaration de la structure Liste */
  int NumElems;                          /* Nombre d'éléments de la liste */
  PTNode First;                          /* Pointeur vers le premier élément */
  PTNode Last;                           /* Pointeur vers le dernier élément */
  PTNode Current;                        /* Pointeur vers l'élément courant */
  int Index;                             /* Indice de l'élément courant */
} TList;                                 /* Type Liste */

typedef TList *PTList;                   /* Type pointeur vers Liste */
```

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
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

```c
#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;
}
```

Parcours de la liste

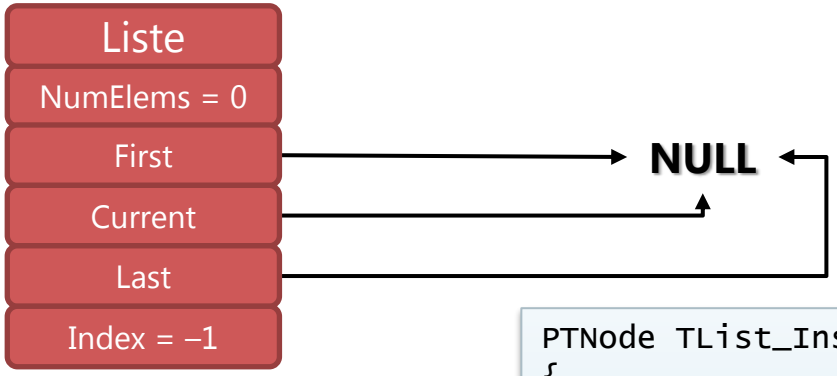
- *GoTo* positionne le pointeur courant sur l'élément situé à la position *Pos* dans la liste et met à jour l'indice, puis renvoie le pointeur courant.

- *GetIndex* est un accesseur qui retourne l'indice courant.

```c
#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;
}
```

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

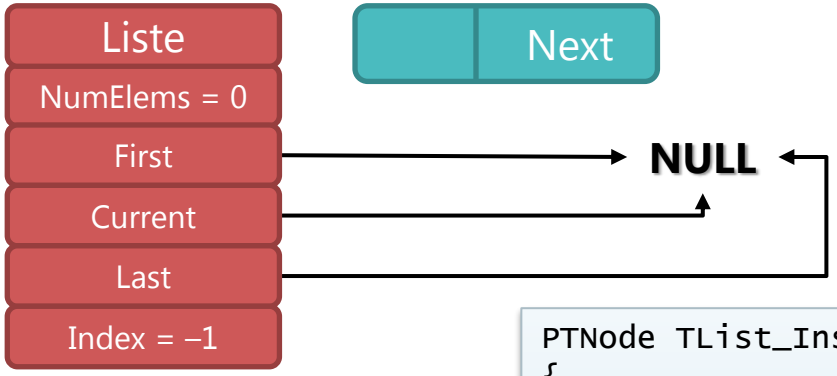
1) Liste vide




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

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

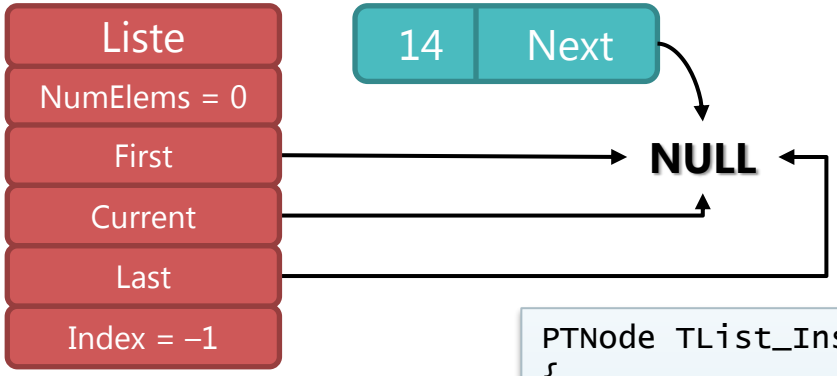
1) Liste vide




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

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

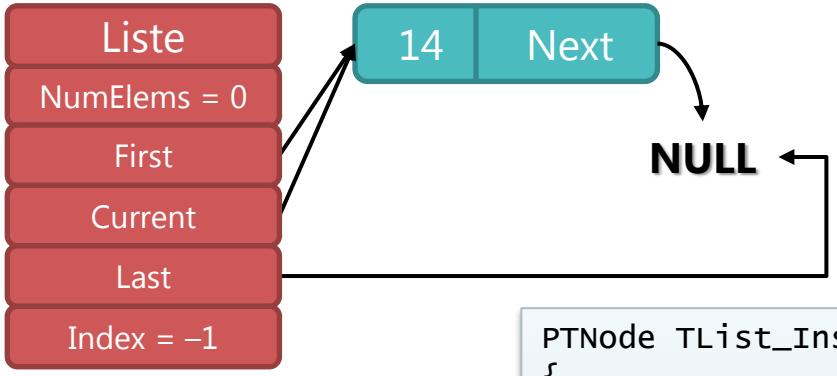
1) Liste vide




```
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;
```

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

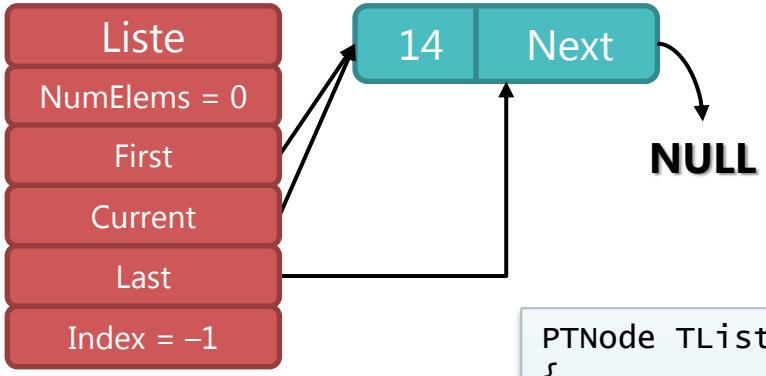
1) Liste vide




```
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;
```

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

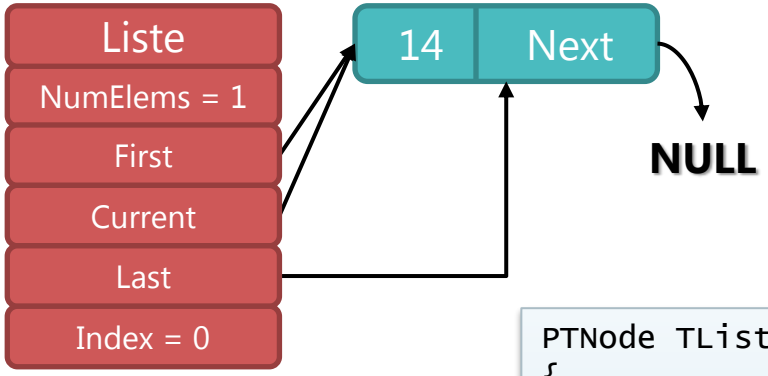
1) Liste vide




```
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;
```

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

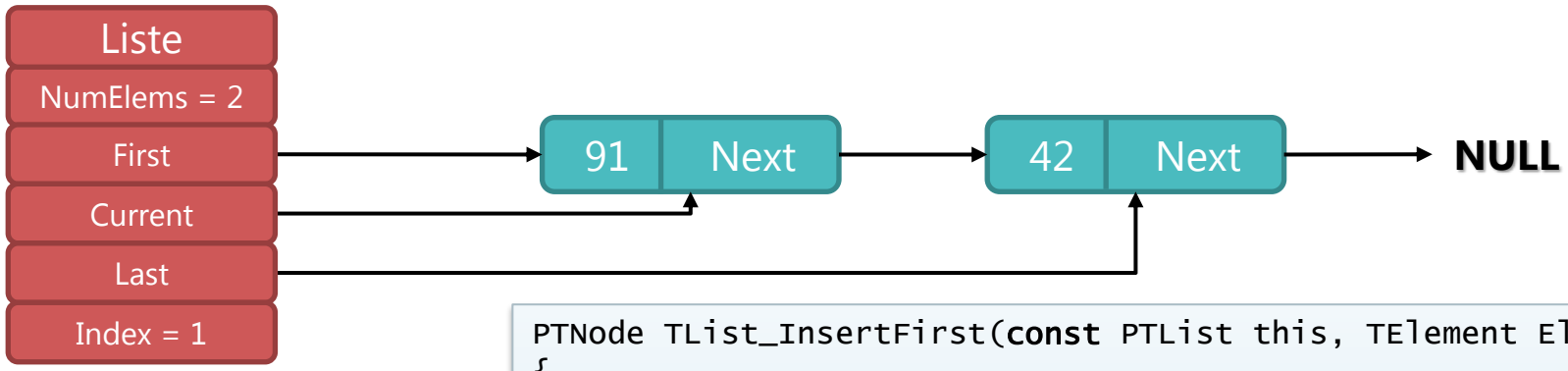
1) Liste vide




```
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;
}
```

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

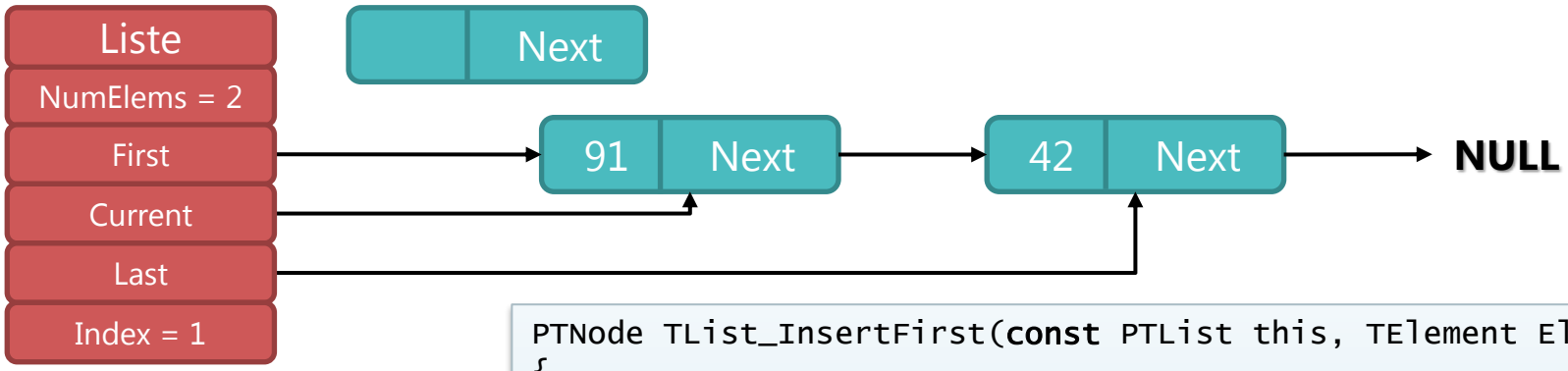
2) Liste non vide




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

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

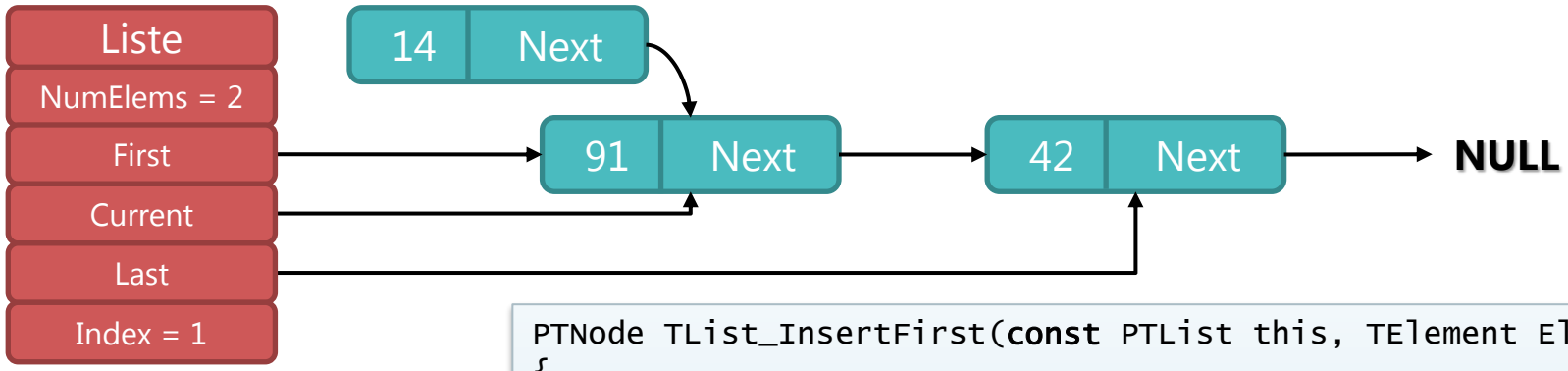
2) Liste non vide




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

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

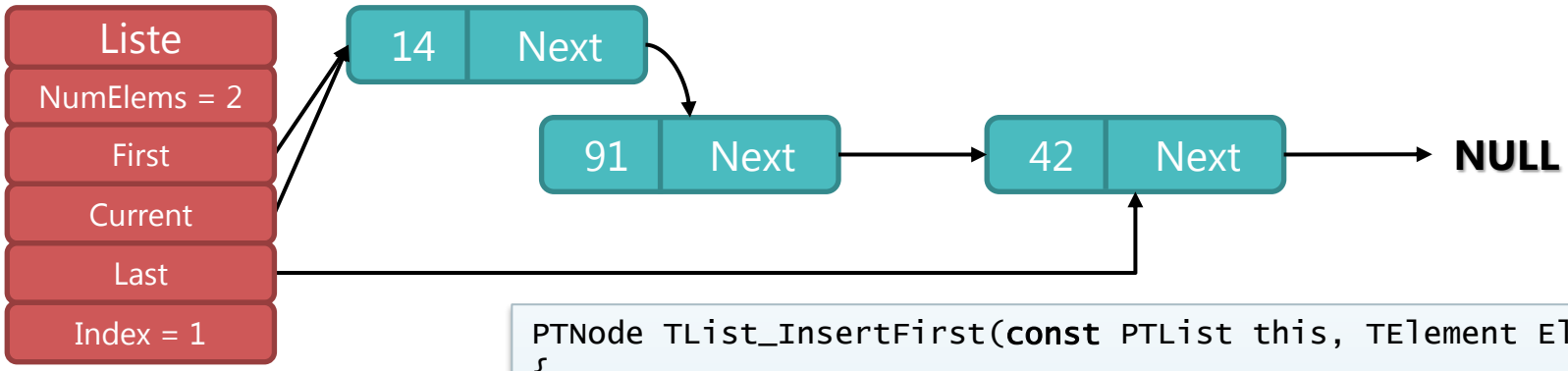
2) Liste non vide




```
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;
```

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

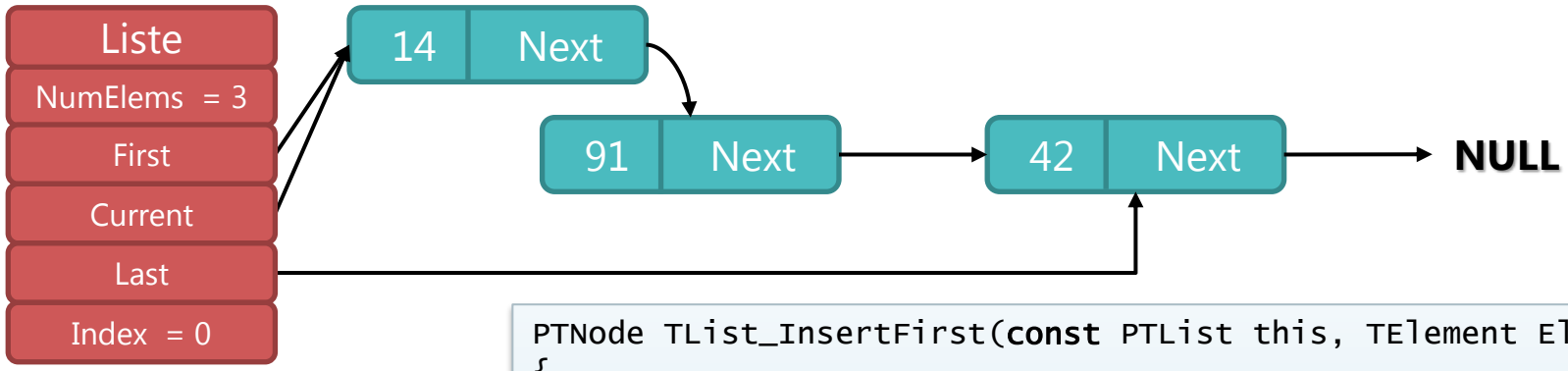
2) Liste non vide




```
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;
```

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

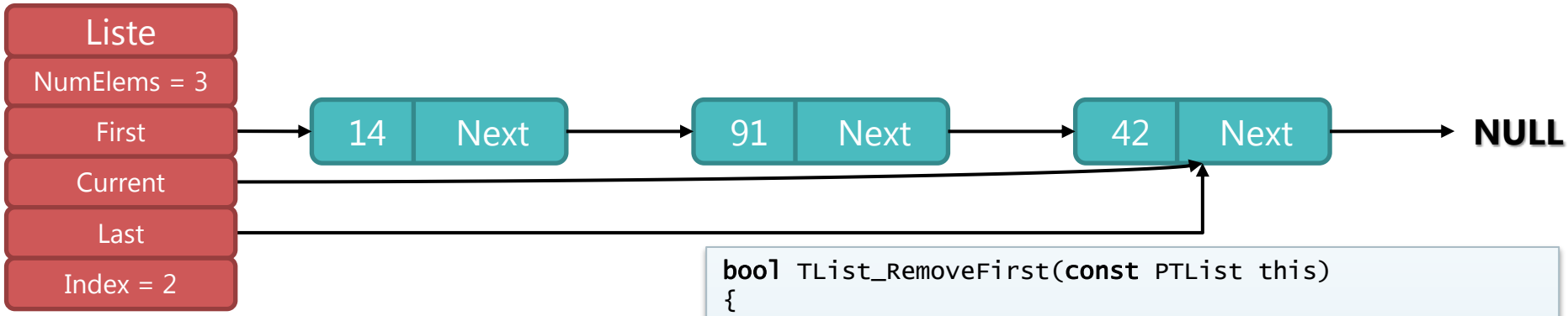
2) Liste non vide




```
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;
}
```

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

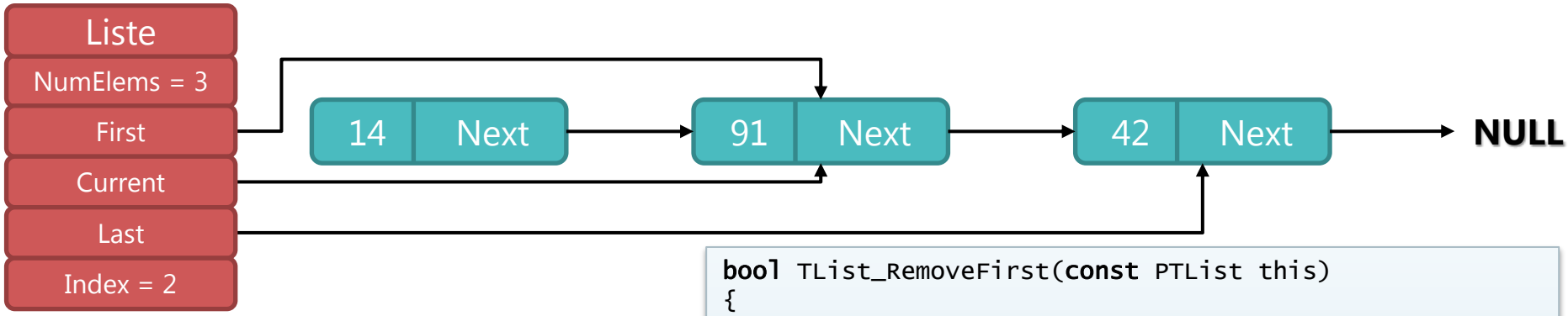
1) Plus d'un élément dans la liste




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

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

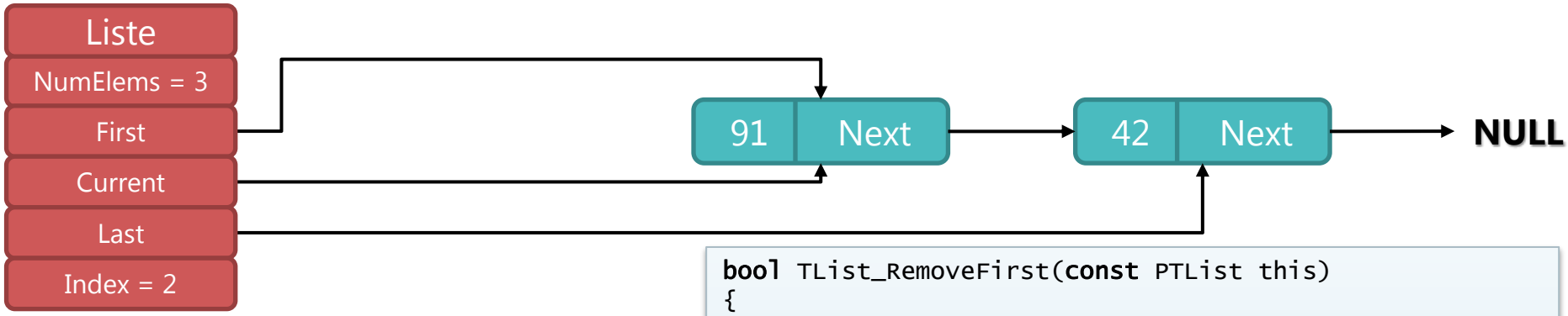
1) Plus d'un élément dans la liste




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

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

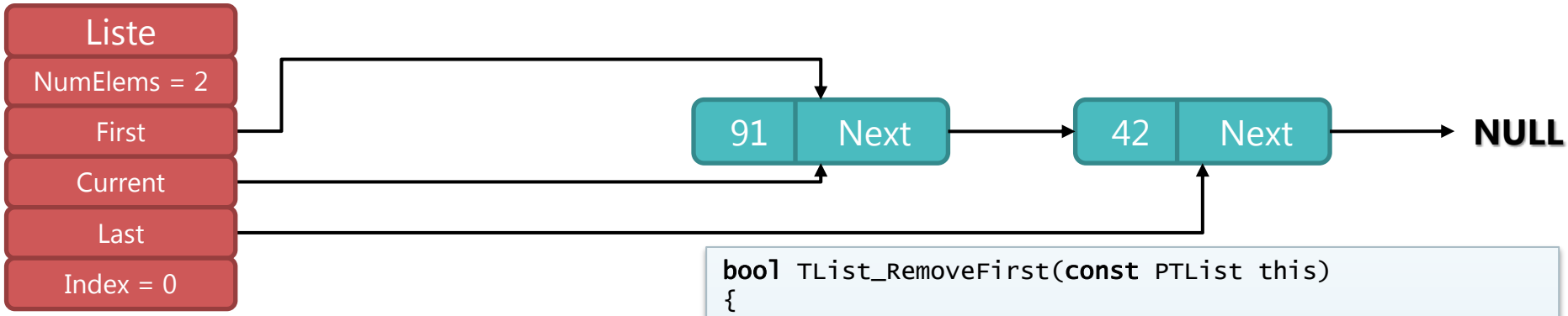
1) Plus d'un élément dans la liste




```
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);
```

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

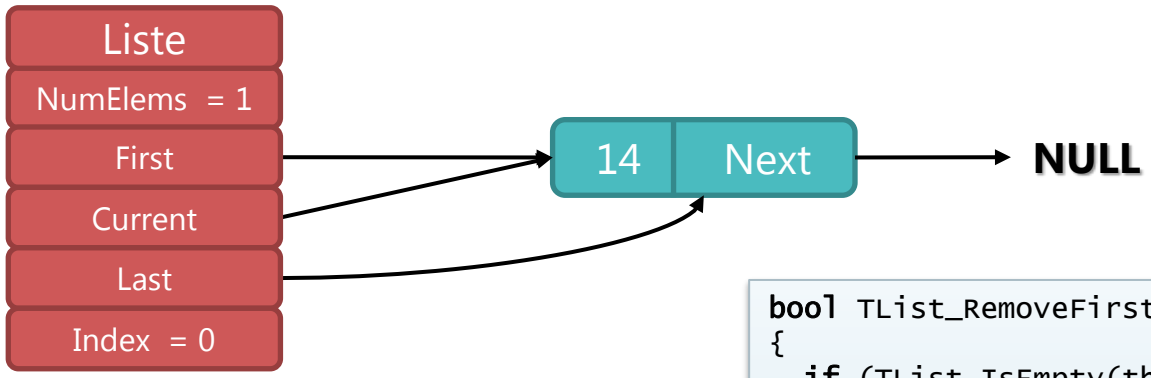
1) Plus d'un élément dans la liste




```
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;
}
```

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

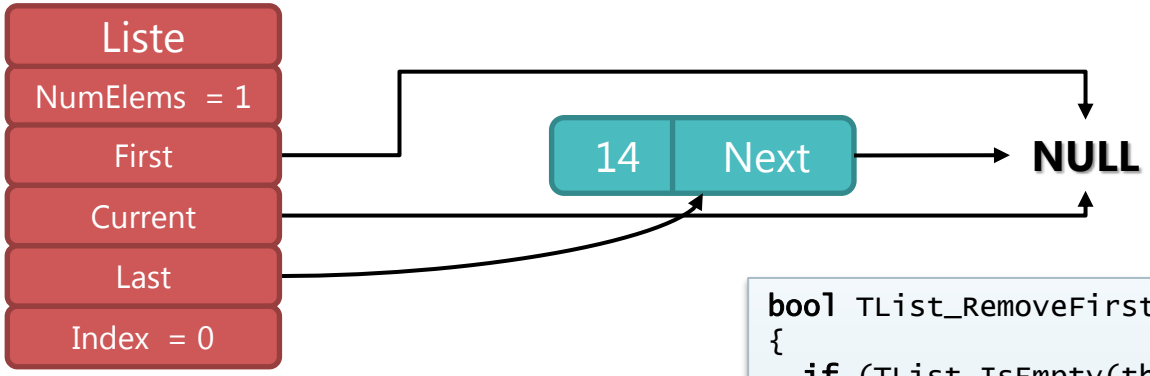
2) Un seul élément dans la liste




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

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

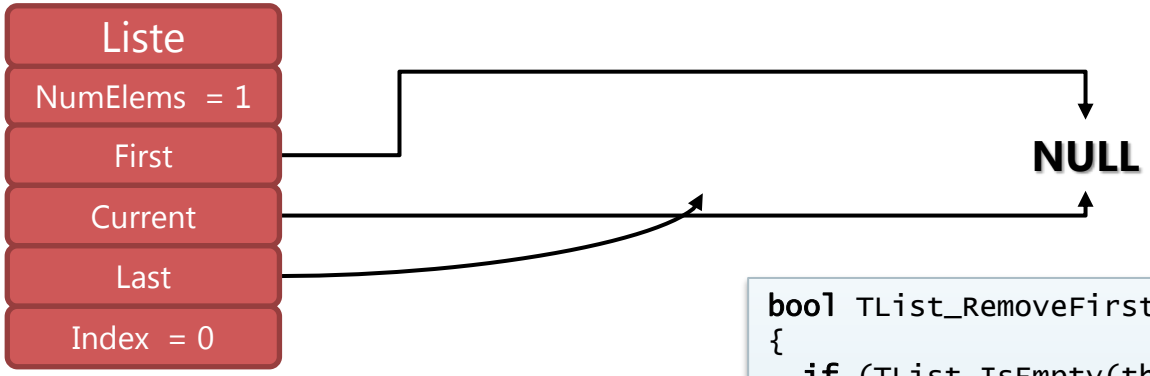
2) Un seul élément dans la liste




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

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

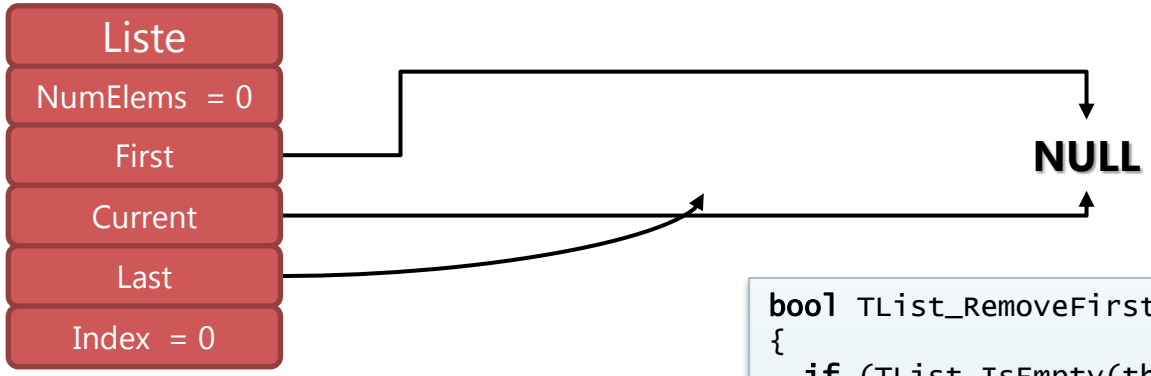
2) Un seul élément dans la liste




```
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);
```

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

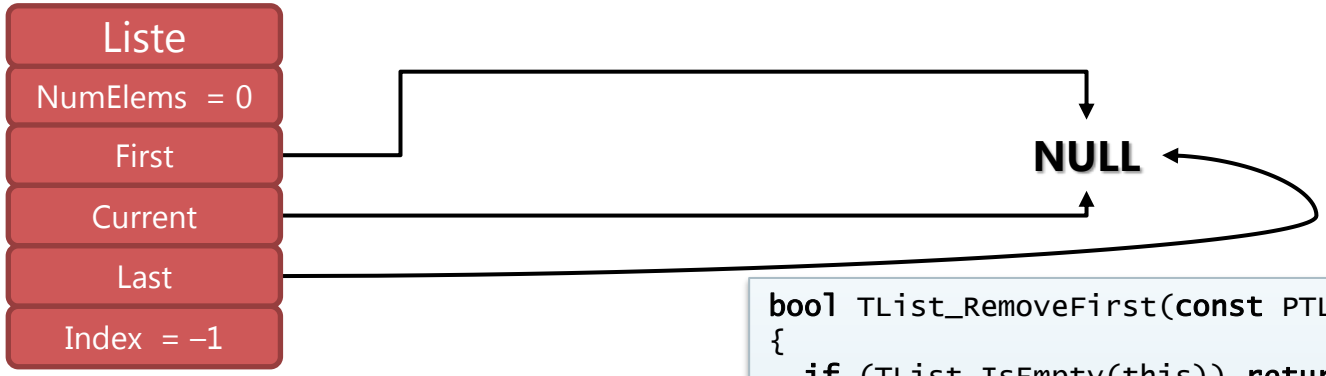
2) Un seul élément dans la liste




```
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--;
```

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

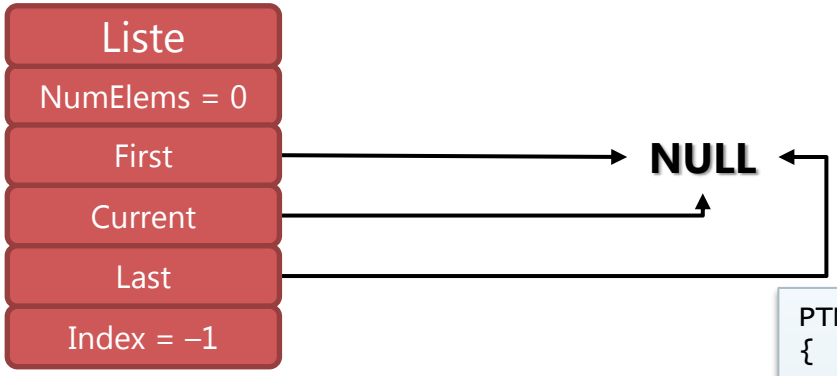
2) Un seul élément dans la liste




```
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;
}
```

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

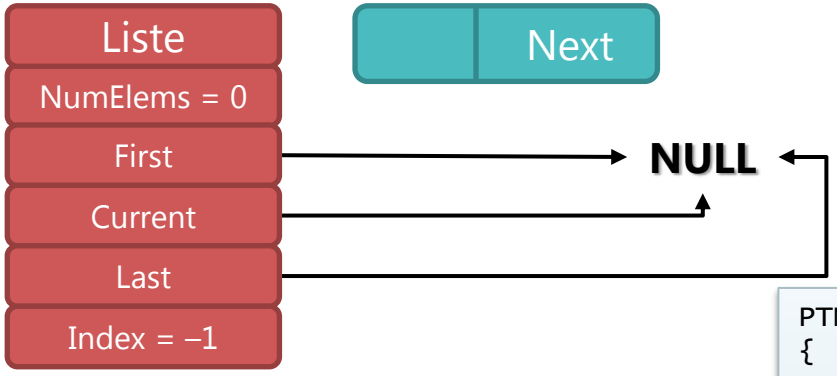
1) Liste vide




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

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

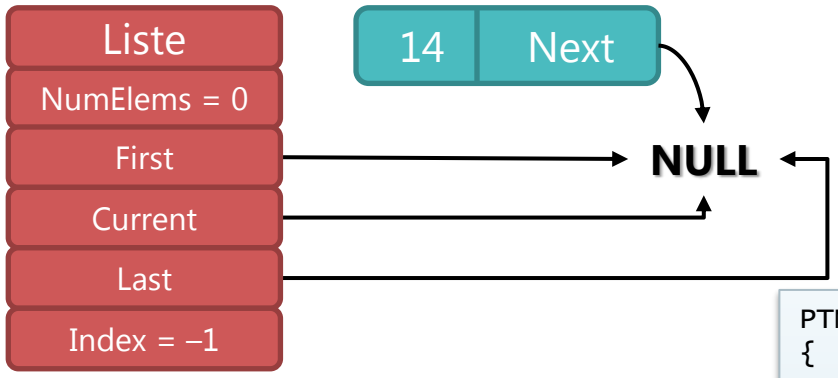
1) Liste vide




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

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

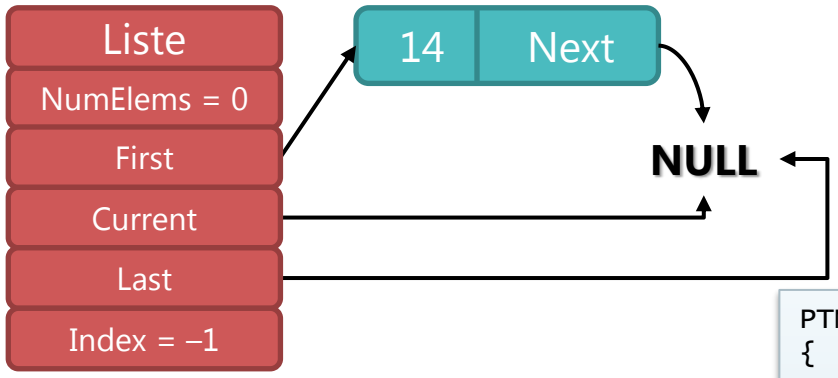
1) Liste vide




```
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;
```

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

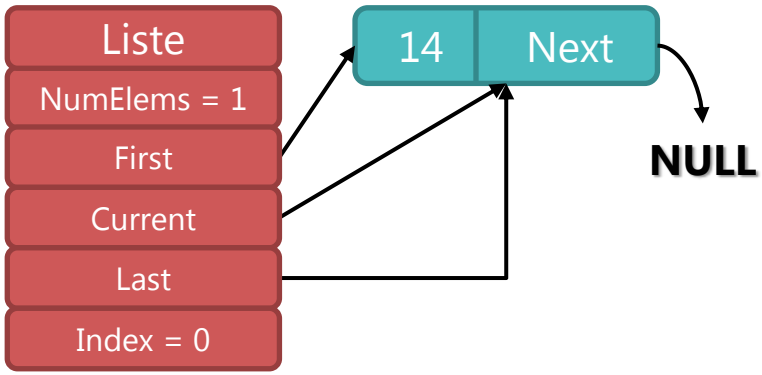
1) Liste vide




```
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;
```

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

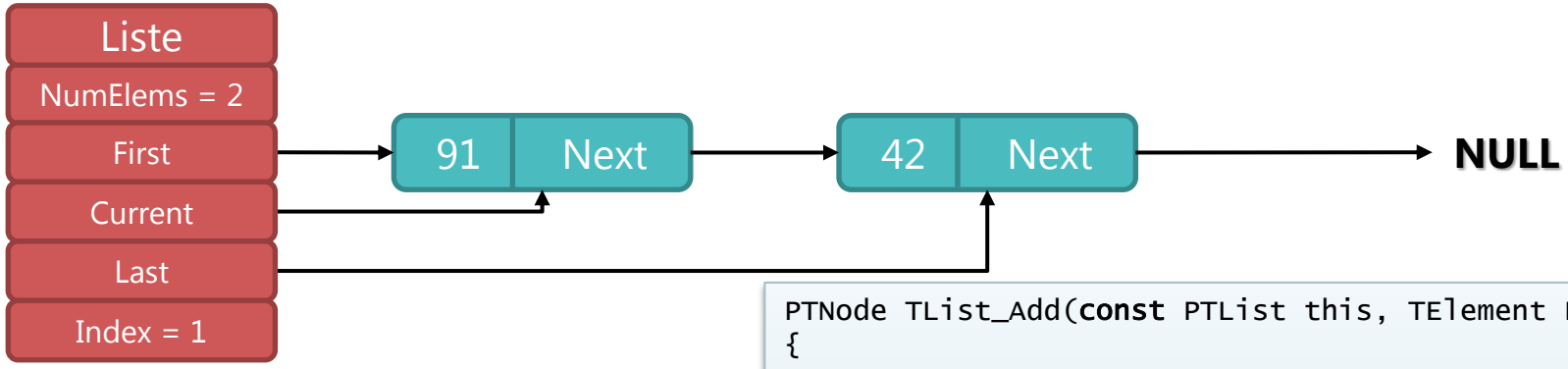
1) Liste vide




```
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;
}
```

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

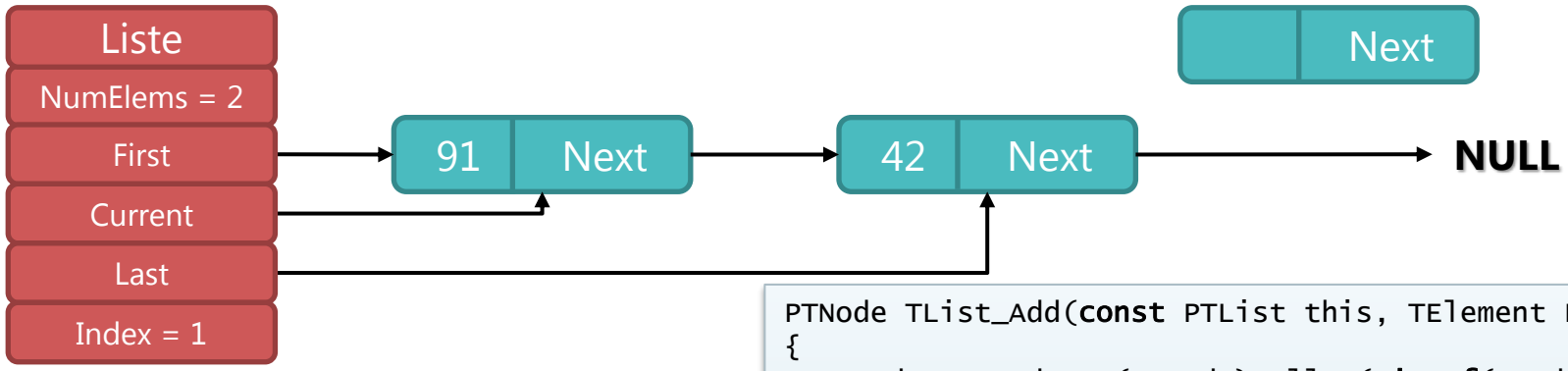
2) Liste non vide




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

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

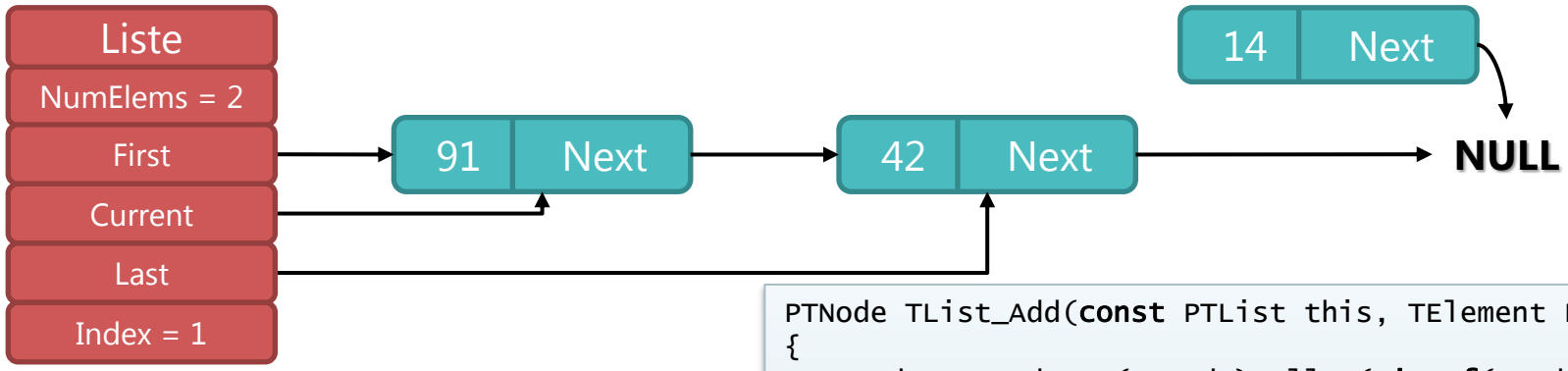
2) Liste non vide




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

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

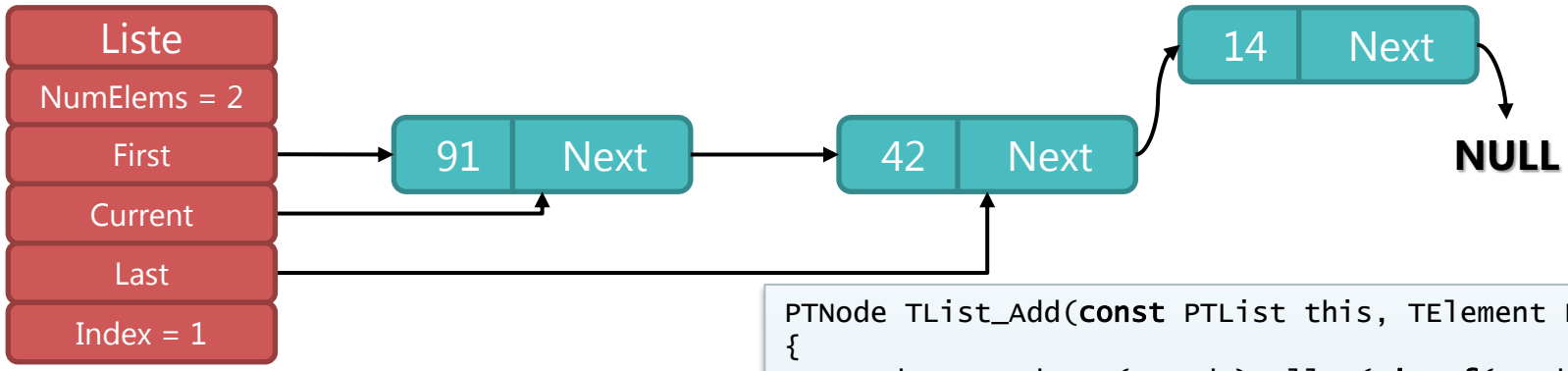
2) Liste non vide




```
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;
```

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

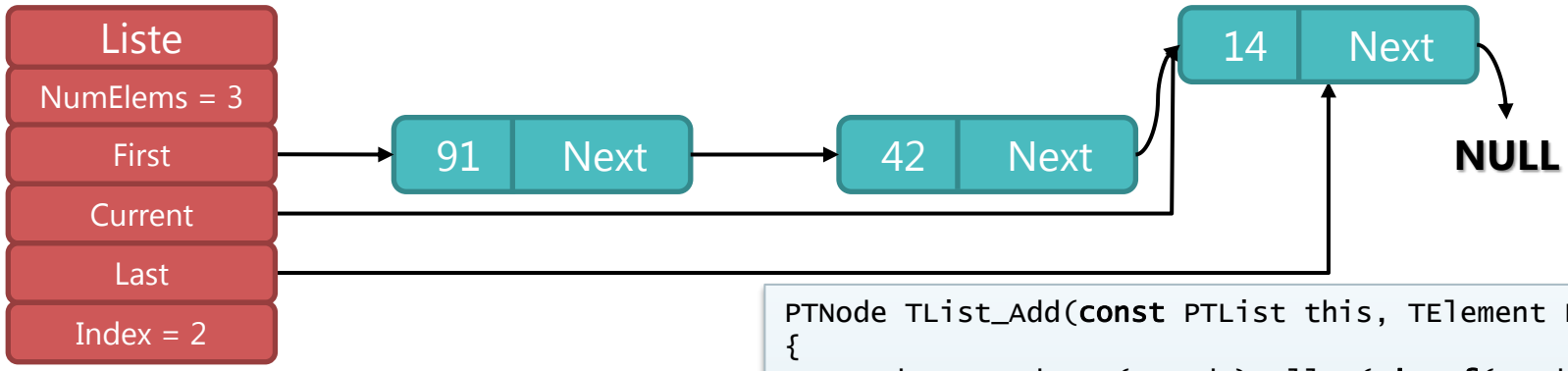
2) Liste non vide




```
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;
```

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

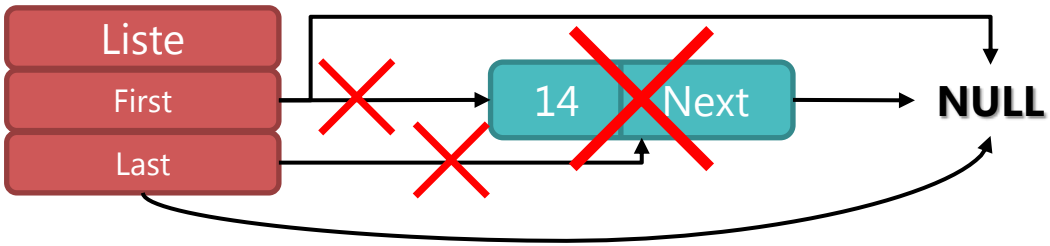
2) Liste non vide



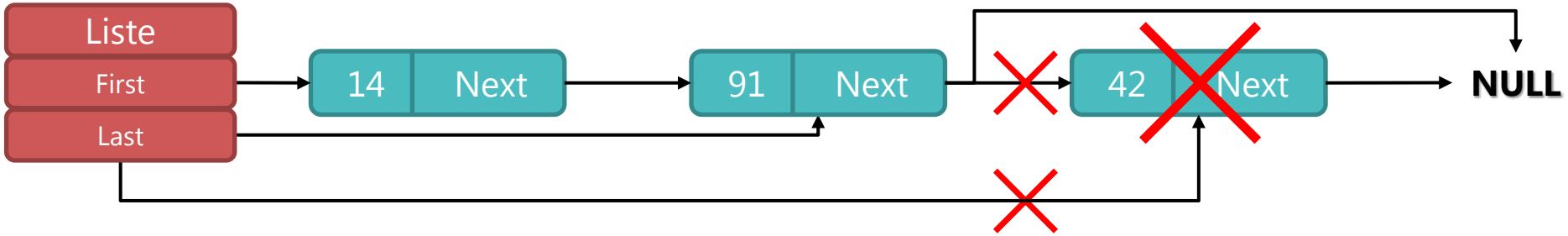

```
PTNode TList_Add(const PTList this, TElement Element)
{
  PTNode newNode = (PTNode)malloc(sizeof(TNode));
  if (newNode == NULL) return NULL;
  newNode->Element = Element;
  newNode->Next = NULL;
  if (TList_IsEmpty(this))
    this->First = newNode;
  else
    this->Last->Next = newNode;
  this->Current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

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



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;
}
```

Fonction TList_Delete : suppression complète de la liste

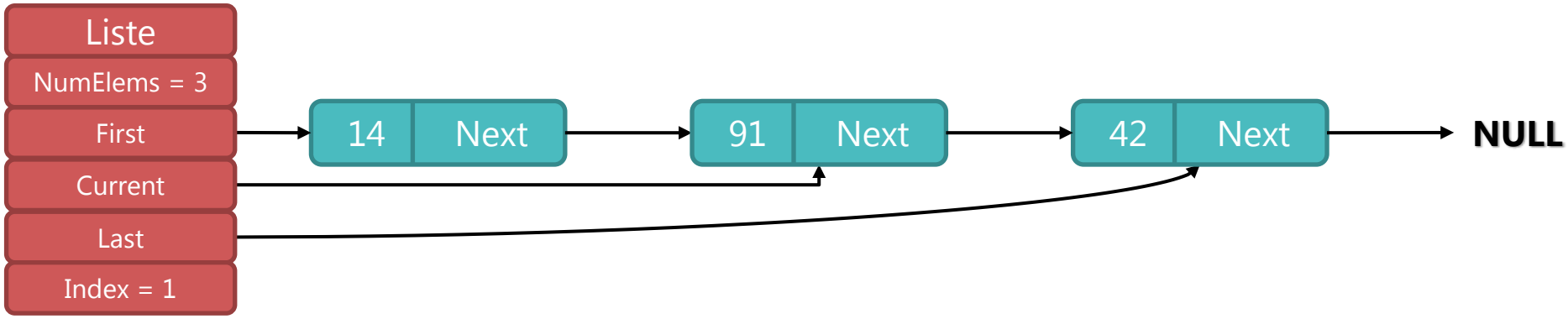
- Comme dans l'implémentation précédente, la suppression complète de la liste s'opère par la suppression successive des éléments de tête (moins coûteuse en temps) jusqu'au dernier élément.

- Il faut ensuite libérer la mémoire allouée à la structure liste.

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

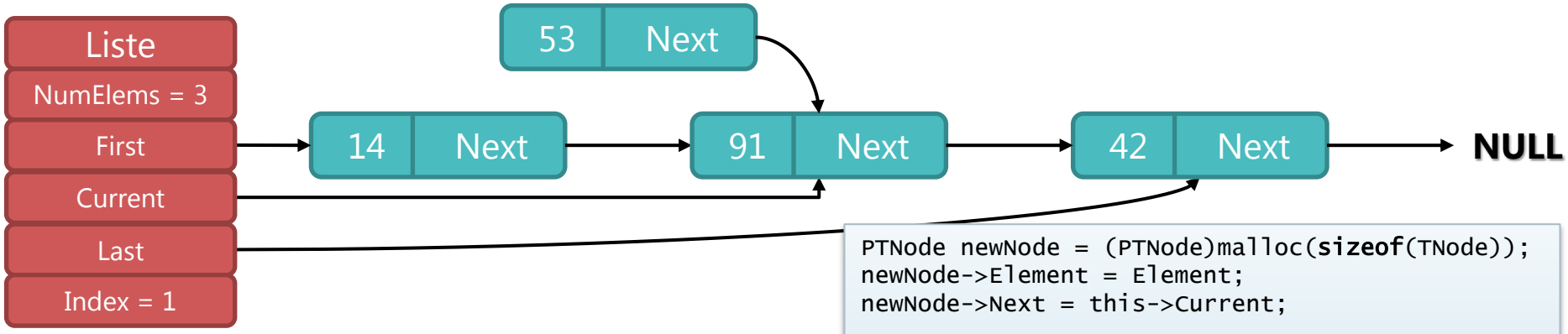
Fonction Tlist_Insert : insertion avant l'élément courant (1/5)

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



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

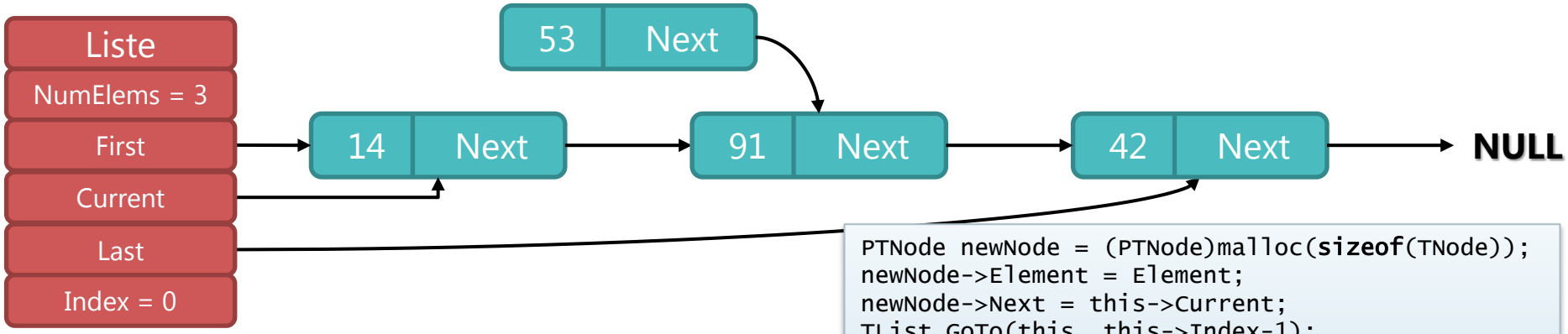
1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst,*
2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant,




```
PTNode newNode = (PTNode)malloc(sizeof(TNode));
newNode->Element = Element;
newNode->Next = this->Current;
```

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

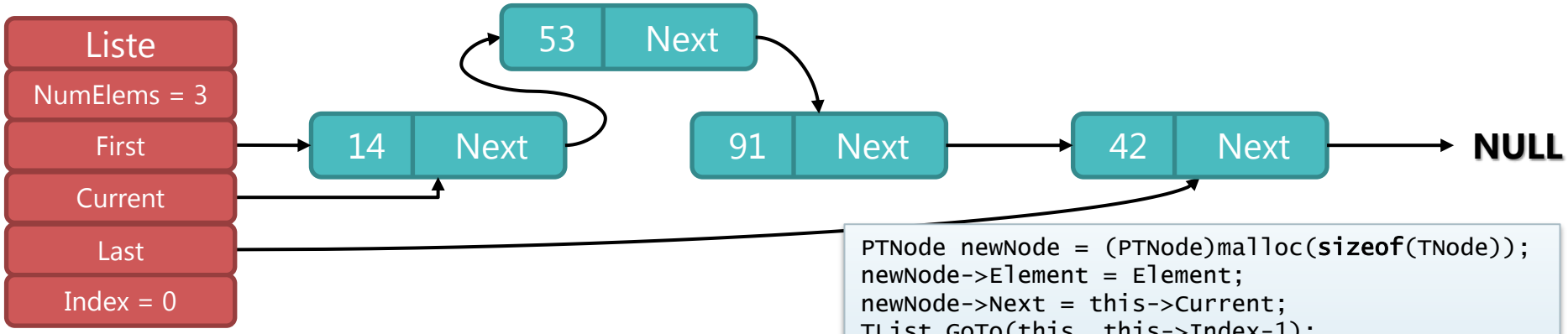
1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst,*
2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant,
3) puis aller au nœud précédent (fonction GoTo),




```
PTNode newNode = (PTNode)malloc(sizeof(TNode));
newNode->Element = Element;
newNode->Next = this->Current;
TList_GoTo(this, this->Index-1);
```

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

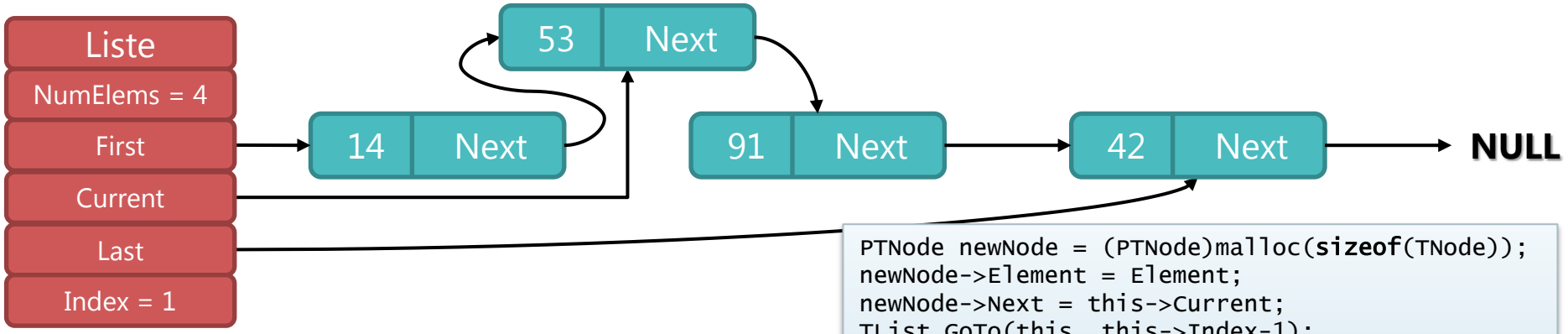
1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst,*
2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant,
3) puis aller au nœud précédent (fonction GoTo),
4) pour le faire pointer vers le nouveau nœud,




```
PTNode newNode = (PTNode)malloc(sizeof(TNode));
newNode->Element = Element;
newNode->Next = this->Current;
TList_GoTo(this, this->Index-1);
this->Current->Next = newNode;
```

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

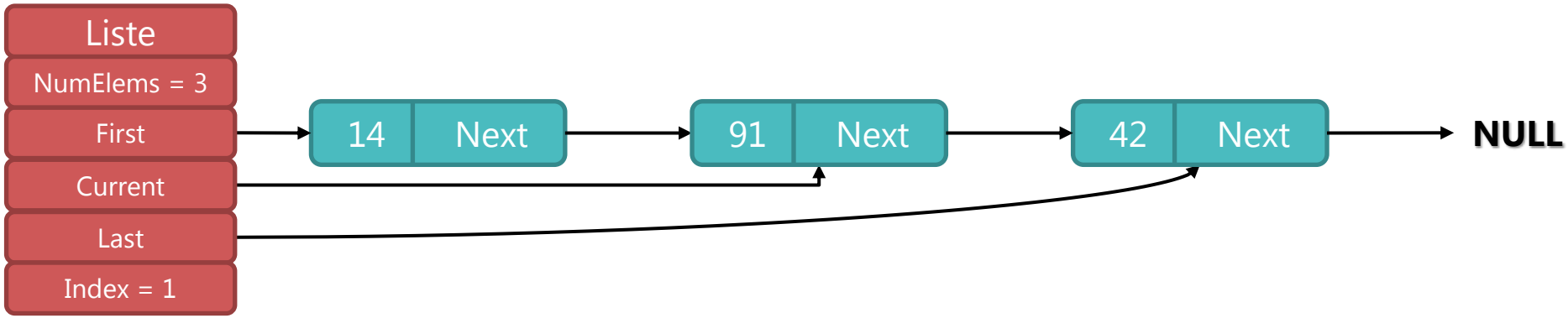
1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst,*
2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant,
3) puis aller au nœud précédent (fonction GoTo),
4) pour le faire pointer vers le nouveau nœud,
5) qui devient alors le nœud courant (index initial inchangé).




```
PTNode newNode = (PTNode)malloc(sizeof(TNode));
newNode->Element = Element;
newNode->Next = this->Current;
TList_GoTo(this, this->Index-1);
this->Current->Next = newNode;
this->Current = newNode;
this->Index++;
this->NumElems++;
```

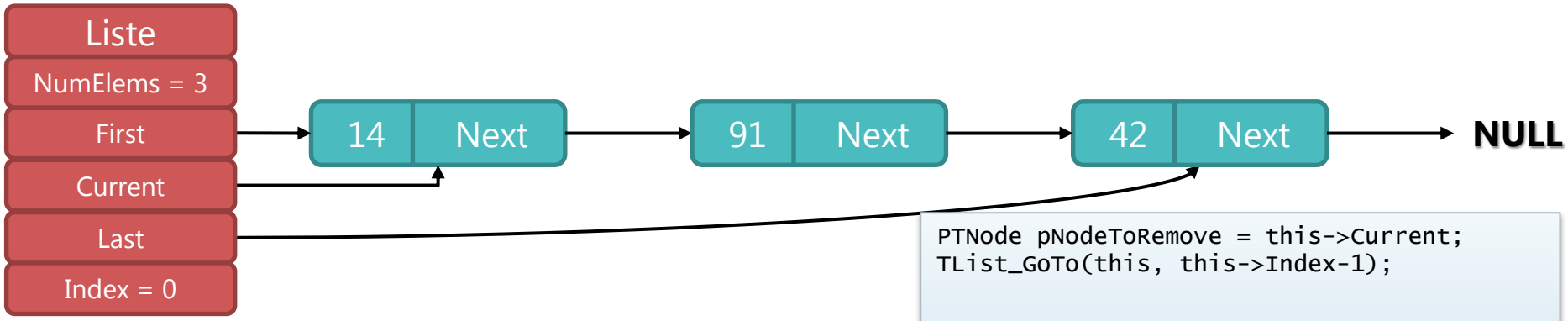
Fonction Tlist_RemoveCurrent : retrait de l'élément courant (1/5)

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*,



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

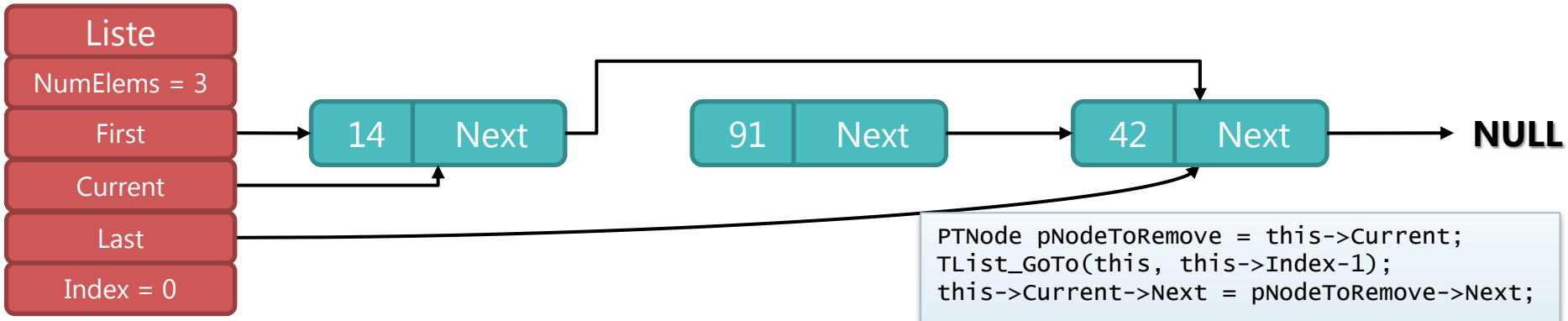
1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst,*
2) Sinon si le nœud courant est le dernier de la liste, appeler la fonction *RemoveLast*,
3) Sinon aller au nœud précédent le nœud à supprimer,




```
PTNode pNodeToRemove = this->Current;
TList_GoTo(this, this->Index-1);
```

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

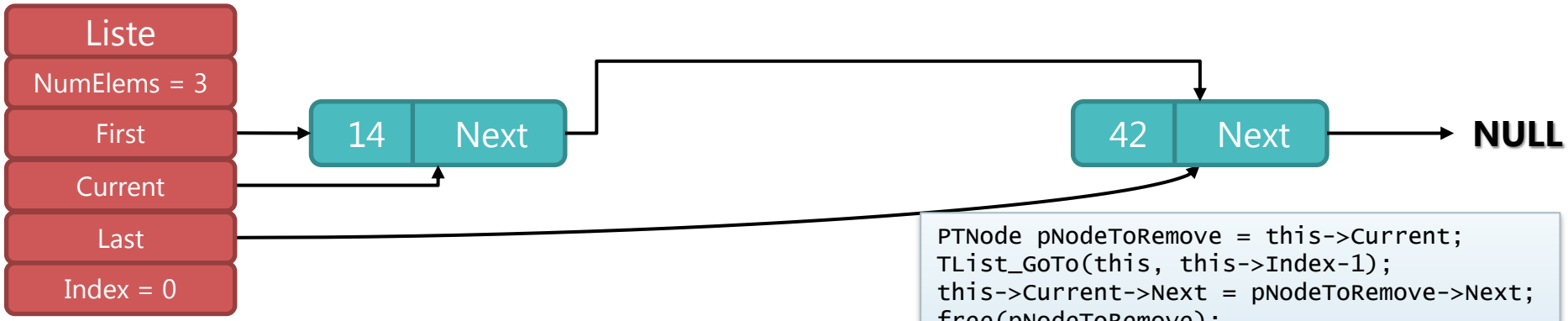
1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst,*
2) Sinon si le nœud courant est le dernier de la liste, appeler la fonction *RemoveLast*,
3) Sinon aller au nœud précédent le nœud à supprimer,
4) pour le faire pointer vers son suivant,




```
PTNode pNodeToRemove = this->Current;
TList_GoTo(this, this->Index-1);
this->Current->Next = pNodeToRemove->Next;
```

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

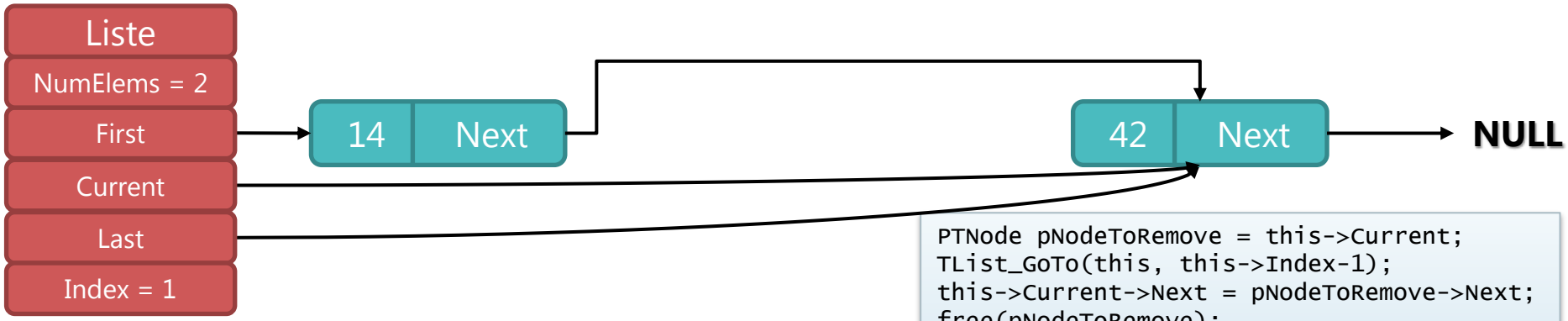
1) Si le nœud courant est le premier de la liste , appeler la fonction *RemoveFirst,*
2) Sinon si le nœud courant est le dernier de la liste, appeler la fonction *RemoveLast*,
3) Sinon aller au nœud précédent le nœud à supprimer,
4) pour le faire pointer vers son suivant,
5) et supprimer le nœud,




```
PTNode pNodeToRemove = this->Current;
TList_GoTo(this, this->Index-1);
this->Current->Next = pNodeToRemove->Next;
free(pNodeToRemove);
```

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).

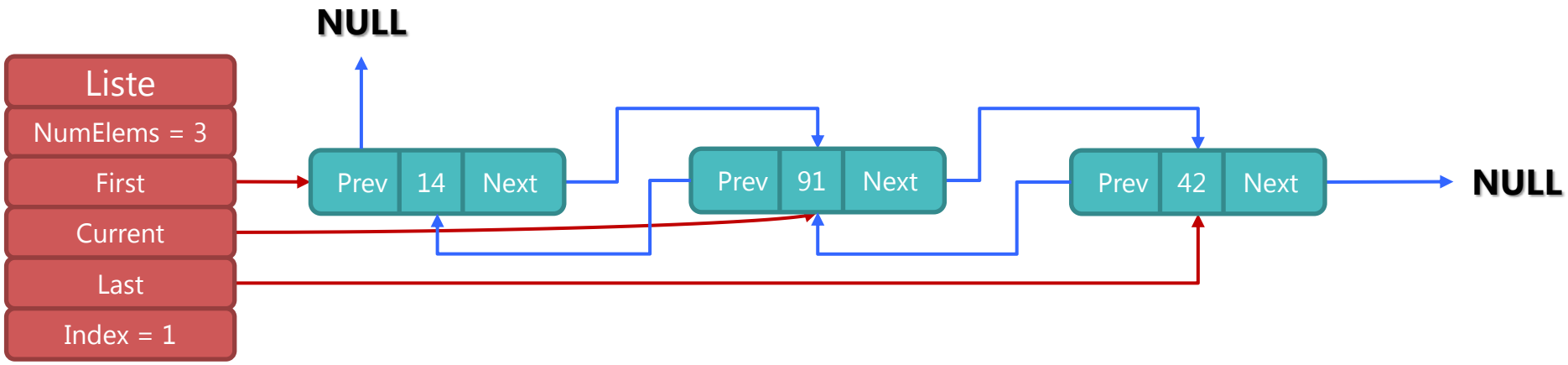



```
PTNode pNodeToRemove = this->Current;
TList_GoTo(this, this->Index-1);
this->Current->Next = pNodeToRemove->Next;
free(pNodeToRemove);
this->Current = this->Current->Next;
this->Index++;
this->NumElems--;
```

Liste doublement chainée

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

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



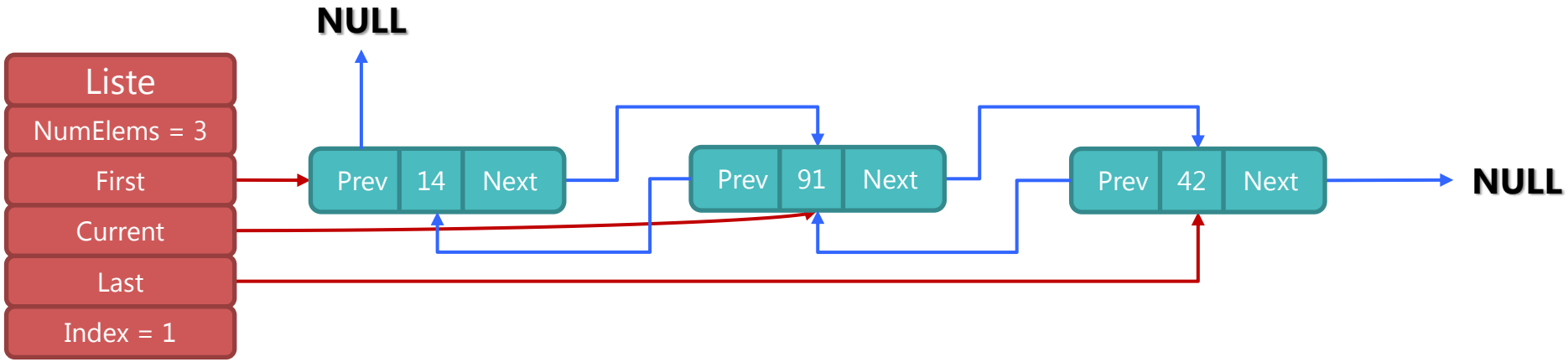
Parcours d'une liste double

- 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;
}
```

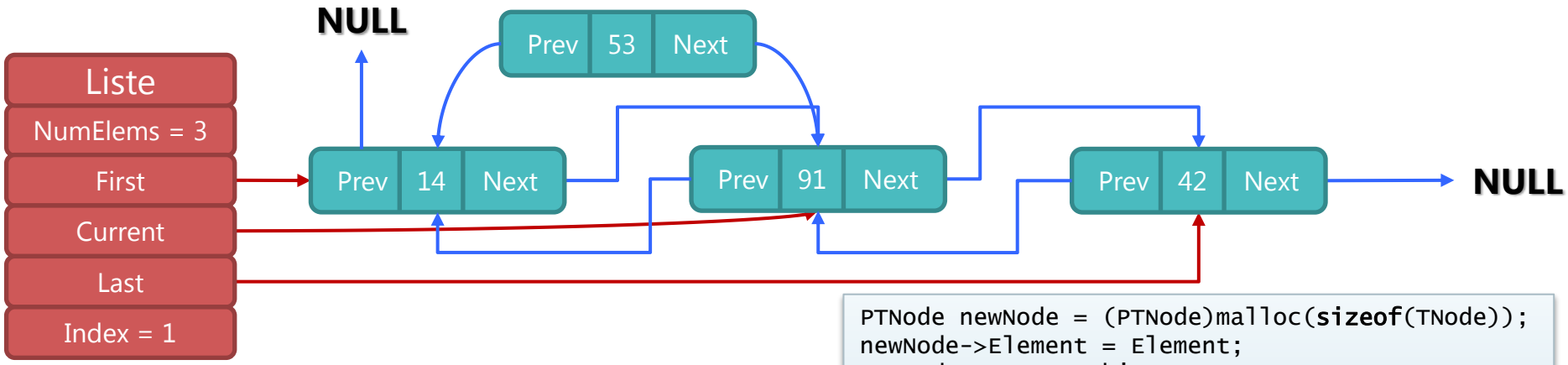
Fonction Tlist_Insert : insertion avant le nœud courant (1/4)

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



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

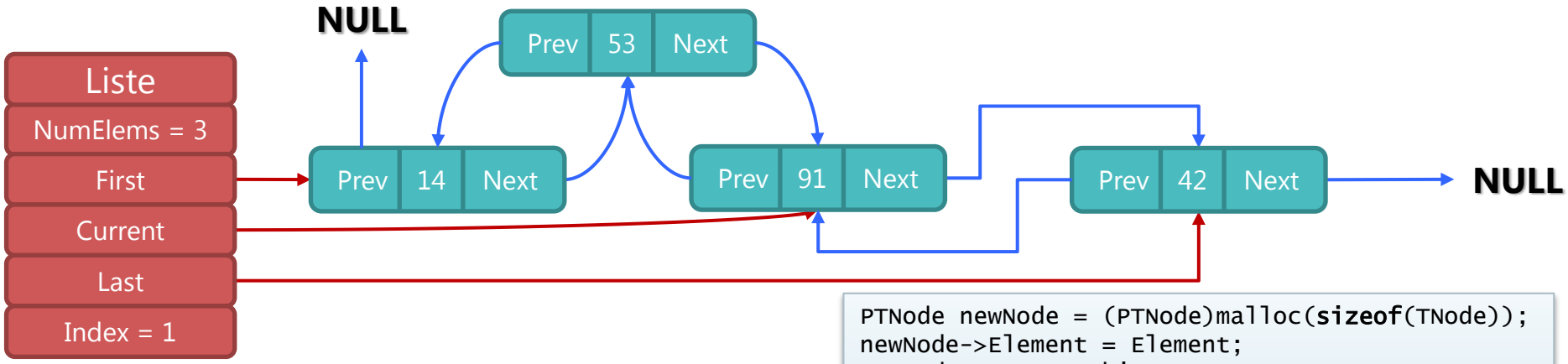
1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst,*
2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant, et son précédent,




```
PTNode newNode = (PTNode)malloc(sizeof(TNode));
newNode->Element = Element;
newNode->Next = this->Current;
newNode->Prev = this->Current->Prev;
```

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

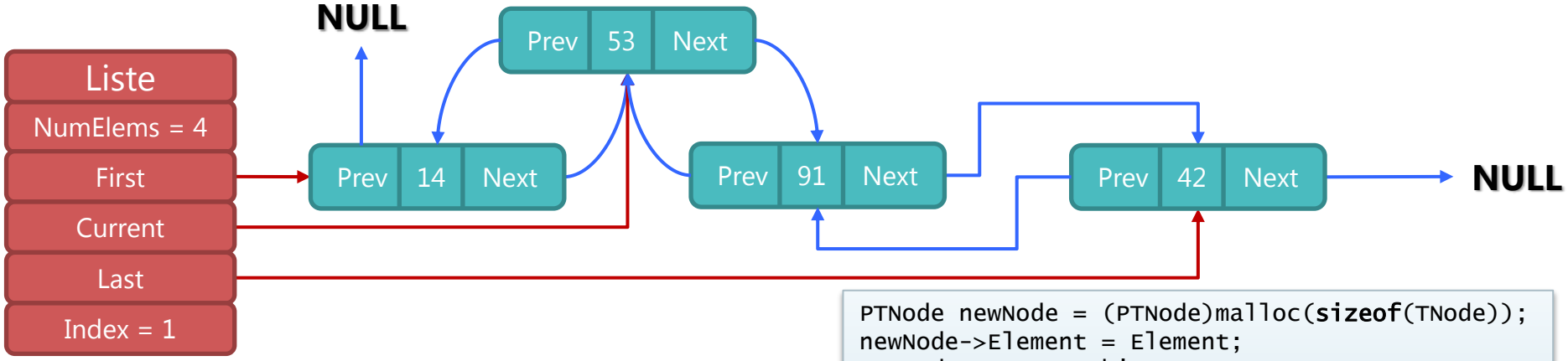
1) Si le nœud courant est le premier de la liste , appeler la fonction *InsertFirst,*
2) Sinon créer le nouveau nœud, qui pointe vers le nœud courant, et son précédent,
3) Les nœuds précédent et courant pointent vers le nouveau nœud,




```
PTNode newNode = (PTNode)malloc(sizeof(TNode));
newNode->Element = Element;
newNode->Next = this->Current;
newNode->Prev = this->Current->Prev;
this->Current->Prev->Next = newNode;
this->Current->Prev = newNode;
```

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.



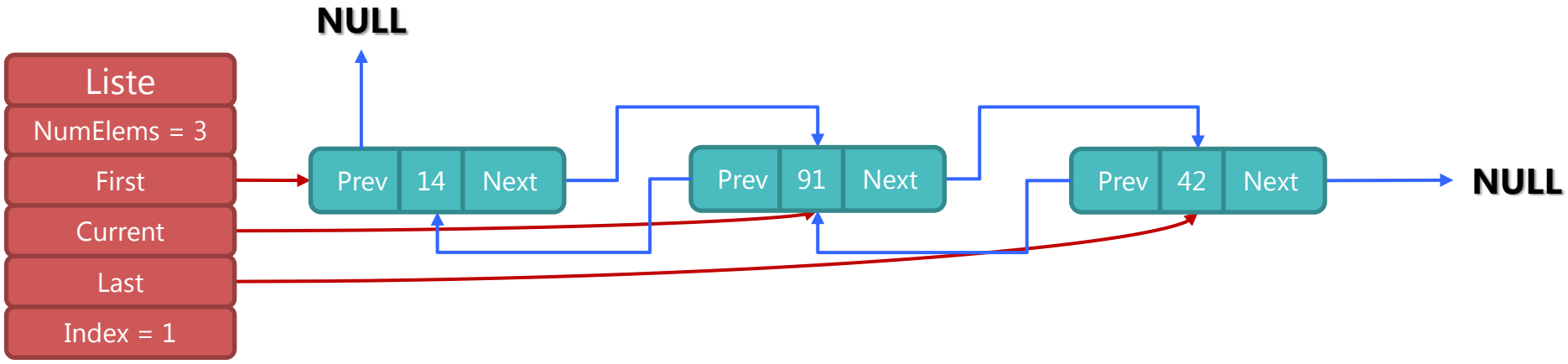

```
PTNode newNode = (PTNode)malloc(sizeof(TNode));
newNode->Element = Element;
newNode->Next = this->Current;
newNode->Prev = this->Current->Prev;
this->Current->Prev->Next = newNode;
this->Current->Prev = newNode;
this->Current = newNode;
this->NumElems++;
```

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;
}
```

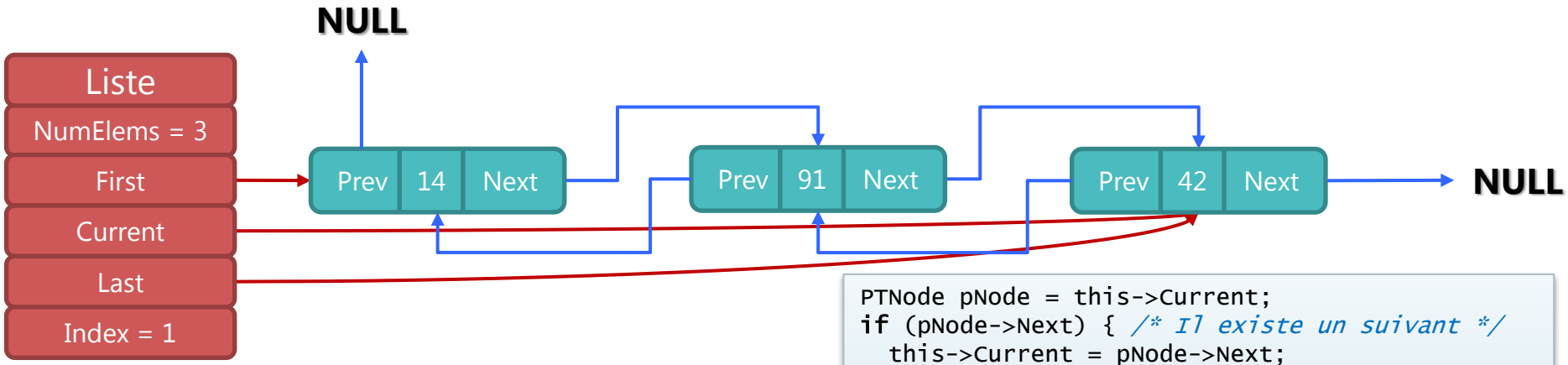
Fonction Tlist_RemoveCurrent : retrait du nœud courant (1/9)

1) S'il existe un suivant,



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

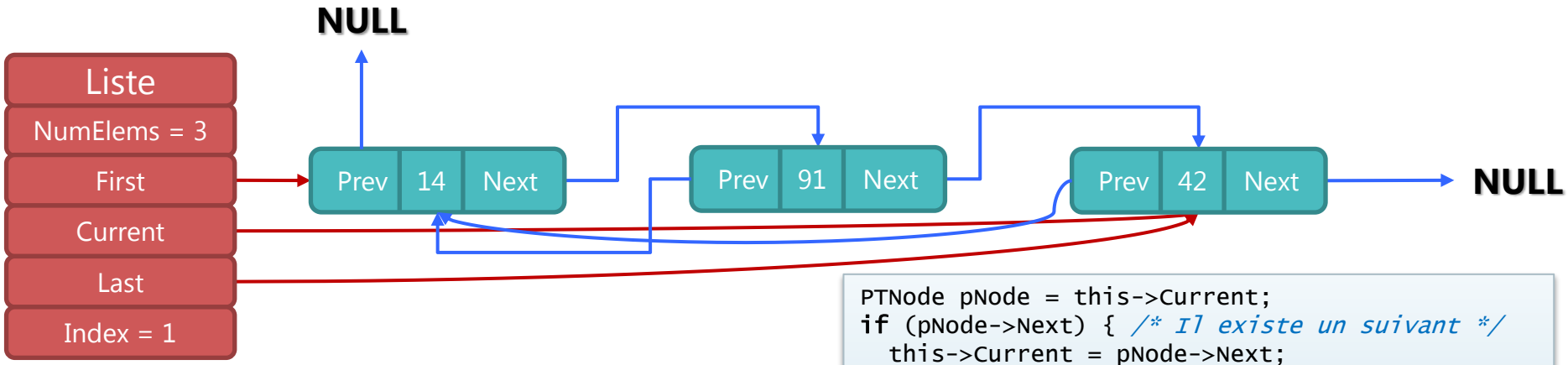
1) S'il existe un suivant, il devient le nouveau nœud courant,




```
PTNode pNode = this->Current;
if (pNode->Next) { /* Il existe un suivant */
  this->Current = pNode->Next;
```

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

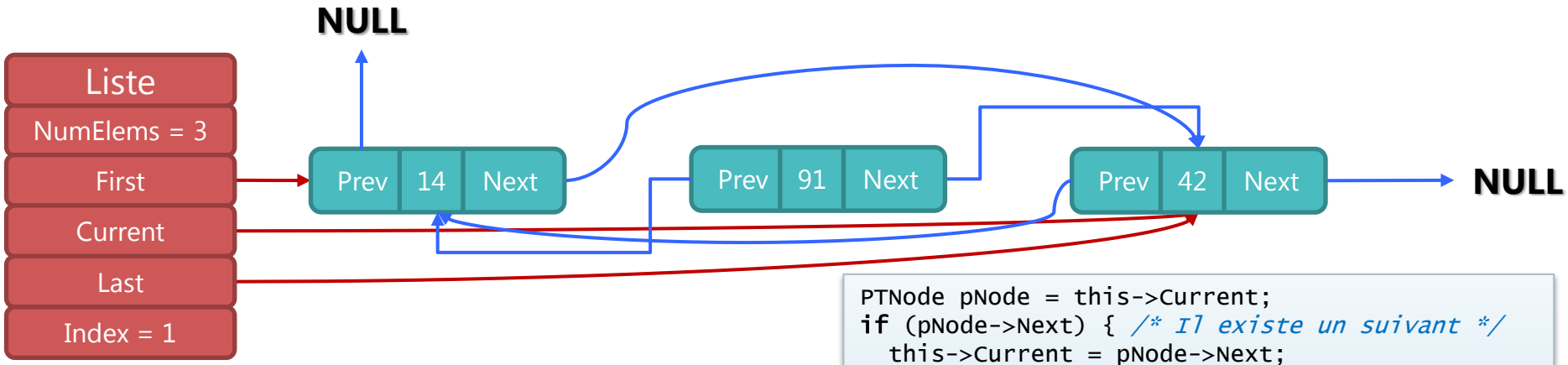
1) S'il existe un suivant, il devient le nouveau nœud courant,
2) Et pointe vers le nœud précédent,




```
PTNode pNode = this->Current;
if (pNode->Next) { /* Il existe un suivant */
  this->Current = pNode->Next;
  this->Current->Prev = pNode->Prev;
```

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

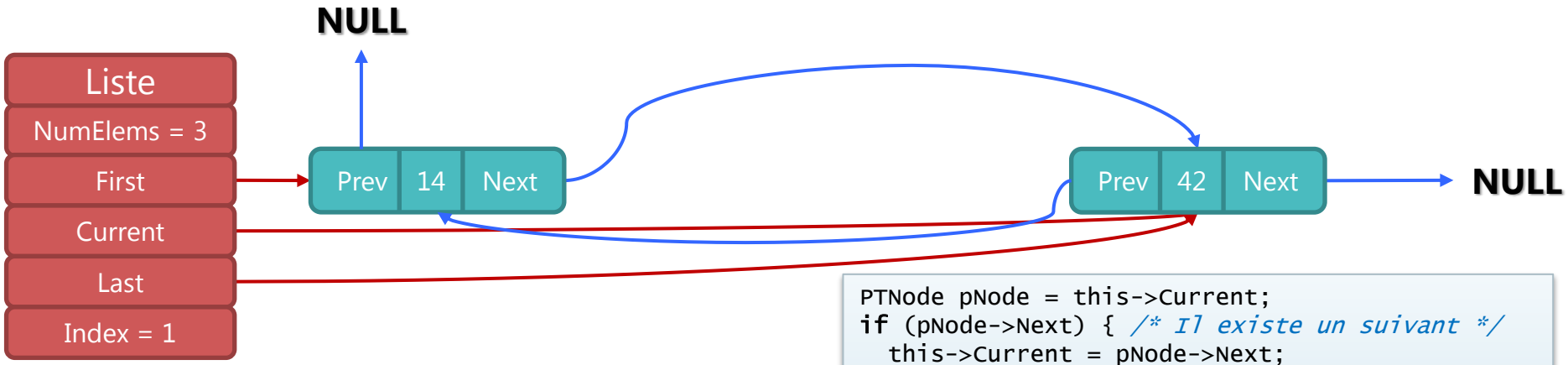
1) S'il existe un suivant, il devient le nouveau nœud courant,
2) Et pointe vers le nœud précédent,
3) qui pointe (si non NULL) à son tour vers le nouveau nœud courant,




```
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;
}
```

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

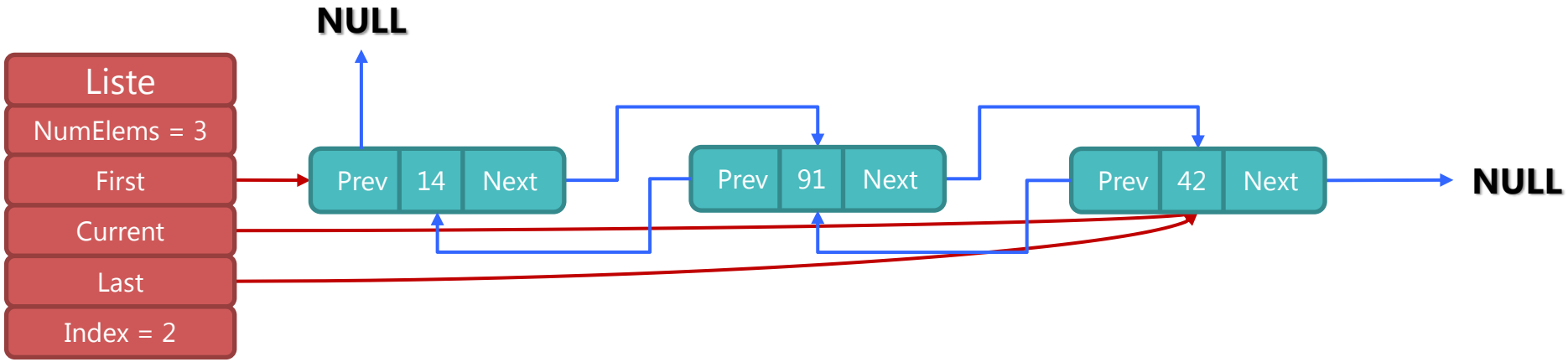
1) S'il existe un suivant, il devient le nouveau nœud courant,
2) Et pointe vers le nœud précédent,
3) qui pointe (si non NULL) à son tour vers le nouveau nœud courant,
4) Et on peut maintenant supprimer l'ancien nœud courant.




```
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;
}
```

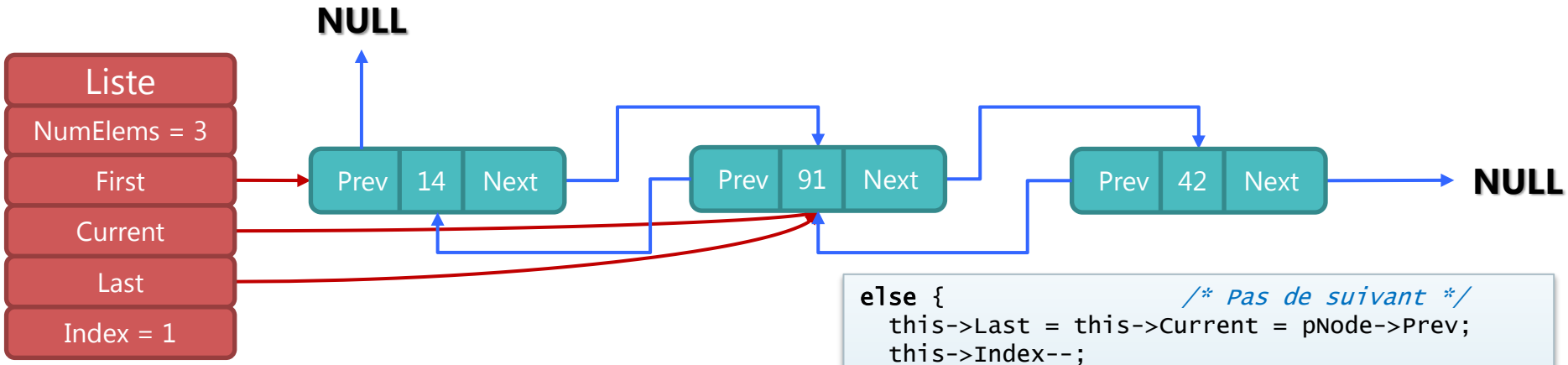
Fonction Tlist_RemoveCurrent : retrait du nœud courant (6/9)

5) Si pas de suivant,



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

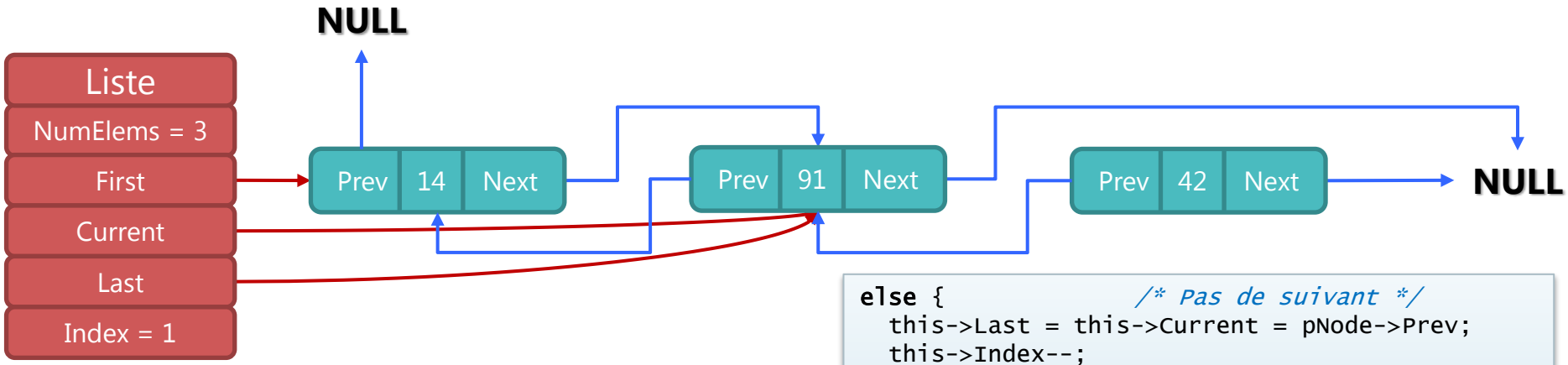
5) Si pas de suivant, c'est le précédent qui devient le nouveau nœud courant,




```
else {                          /* Pas de suivant */
  this->Last = this->Current = pNode->Prev;
  this->Index--;
```

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

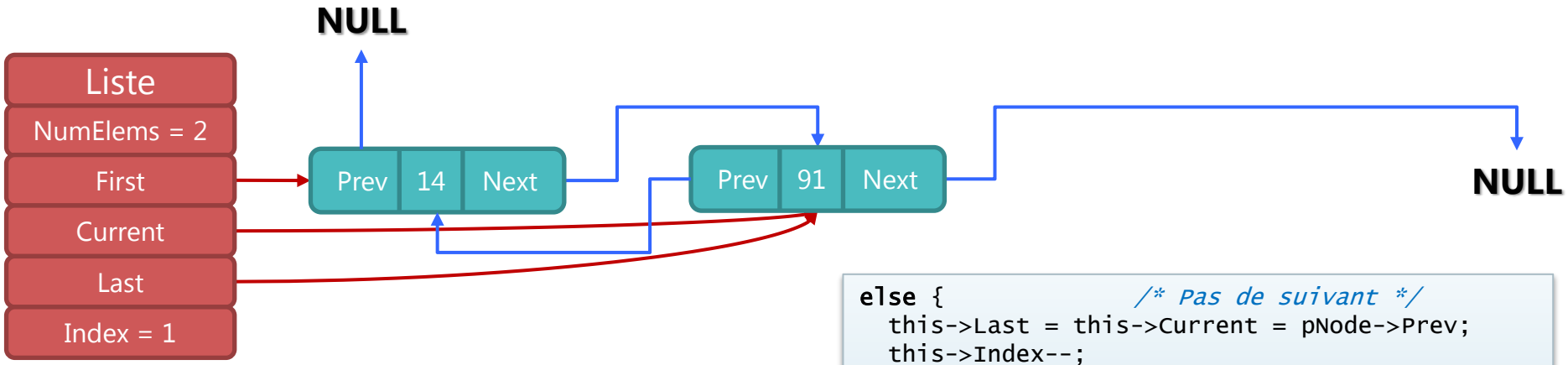
5) Si pas de suivant, c'est le précédent qui devient le nouveau nœud courant,
6) et pointe vers NULL (s'il n'est pas lui-même égal à NULL, auquel cas la liste serait vide)




```
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;
}
```

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.




```
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--;
```

Fonction Tlist_RemoveCurrent : code C

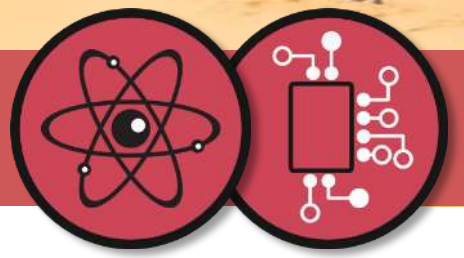
```
bool TList_RemoveCurrent(const PTList this)
{
  if (TList_IsEmpty(this)) return false;
  PTNode pNode = this->Current;
  if (pNode->Next) {          /* Il existe un suivant */
    this->Current = pNode->Next;
    this->Current->Prev = pNode->Prev;
    if (pNode->Prev)          /* Il existe un précédent */
      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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

- Rappels et approfondissements en C
- Listes chainées
- Piles et files
- Algorithmes de tri
- Tables de hachage
- Arbres binaires

Piles et files



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

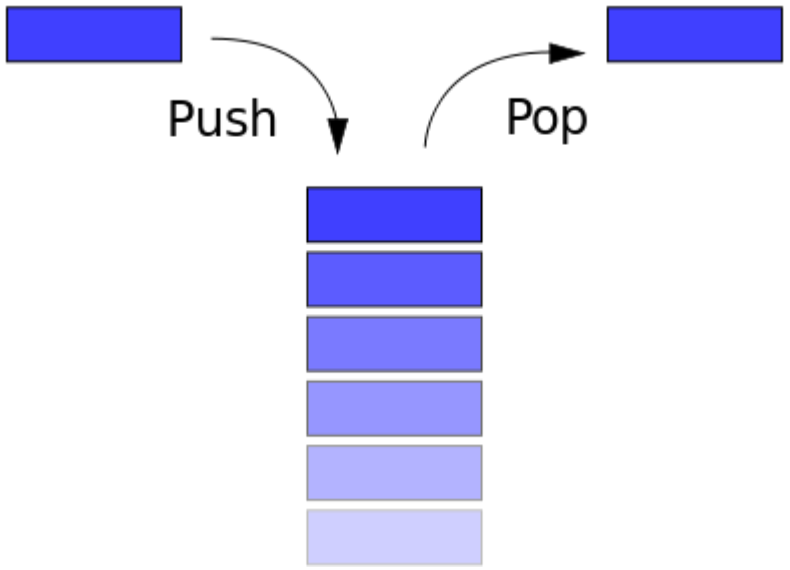
Les piles - Implémentation à l'aide d'une liste chaînée

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

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

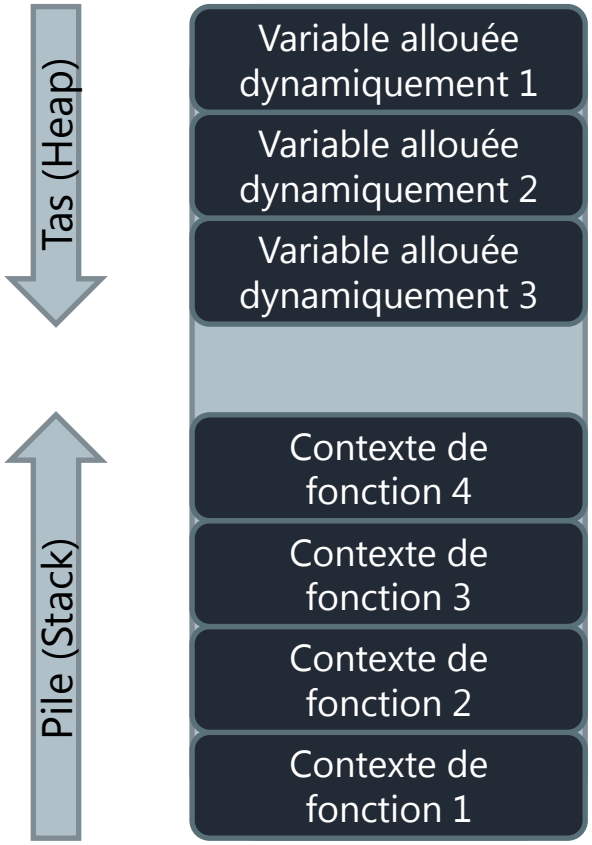
Les piles (stack)

- 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.



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.

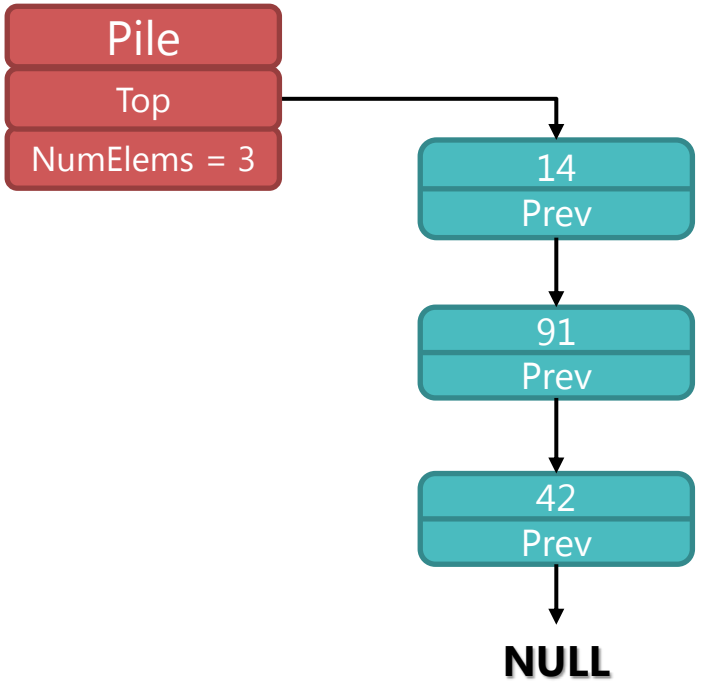


Implémentation d'une pile à l'aide d'une liste chaînée simple

- 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` ↔ `InsertFirst`
  - `Pop` ↔ `RemoveFirst`

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.



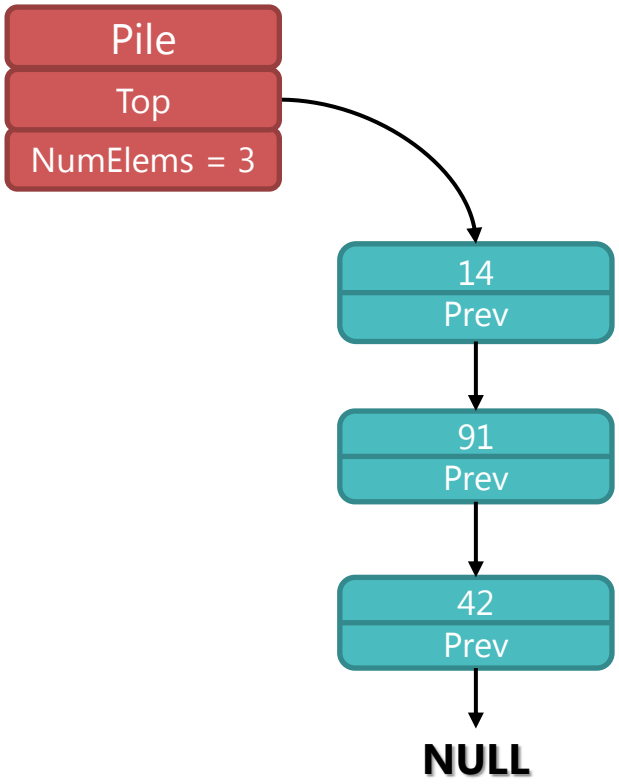

```
typedef struct Node {
  TElement Element;
  struct Node *Prev;
} TNode;

typedef TNode *PTNode;

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

typedef TStack *PTStack;
```

Opérations sur le TAD pile

- **Nom** : TStack,
- **Utilise** : **int**, **bool**, TElement, TNode
- **Opérations** :

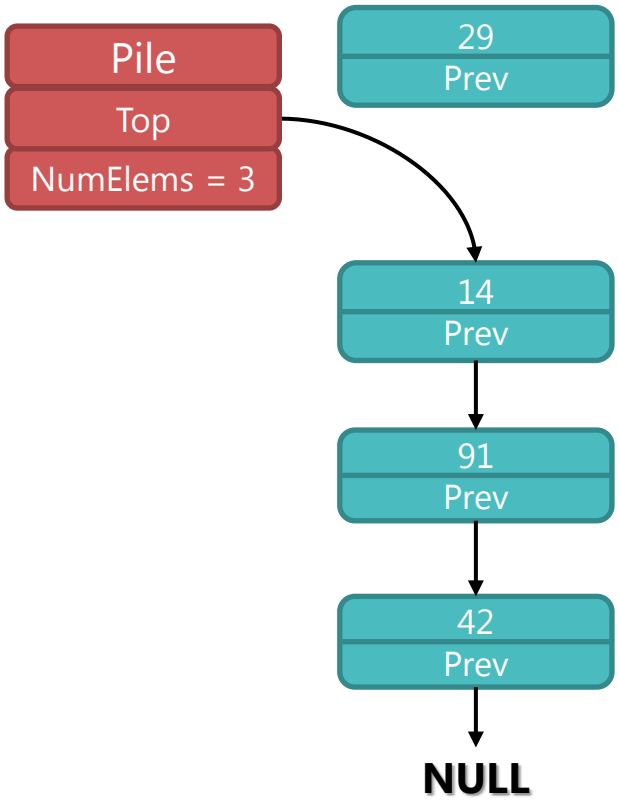
```
New → TStack
IsEmpty : TStack → bool
Size : TStack → int
Top : TStack → TNode
Push : TStack x TElement → TNode
Pop : TStack → bool x TElement
Clear : TStack →
Display : TStack →
Delete : TStack →
```

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



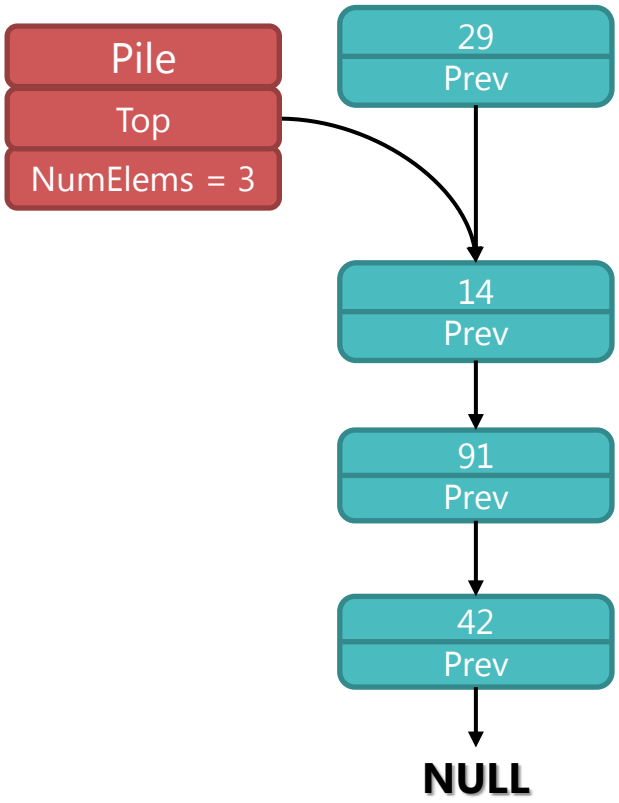

```
PTNode TStack_Push(const PTStack this, TElement Element)
{
```

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



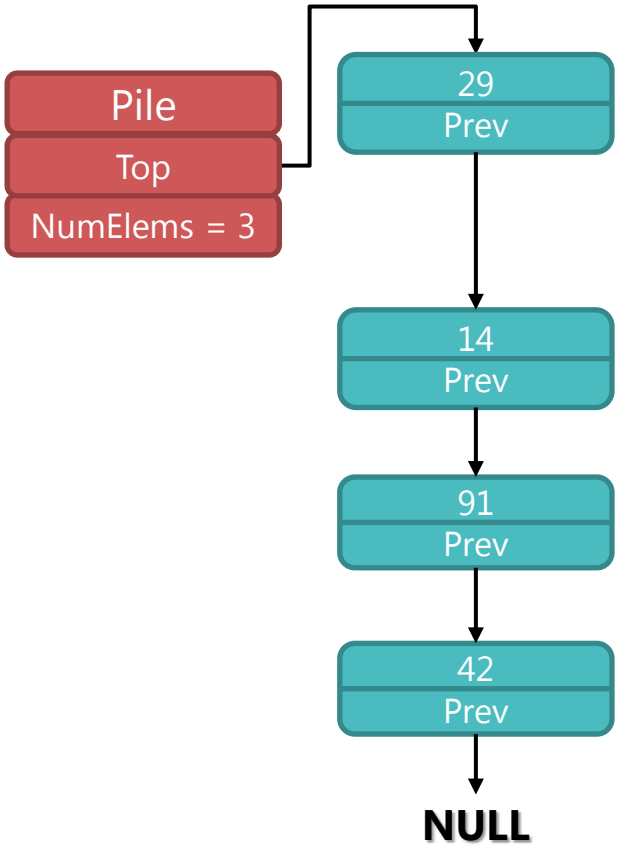

```
PTNode TStack_Push(const PTStack this, TElement Element)
{
  PTNode newNode = (PTNode)malloc(sizeof(TNode));
  if (newNode == NULL) return NULL;
  newNode->Element = Element;
```

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



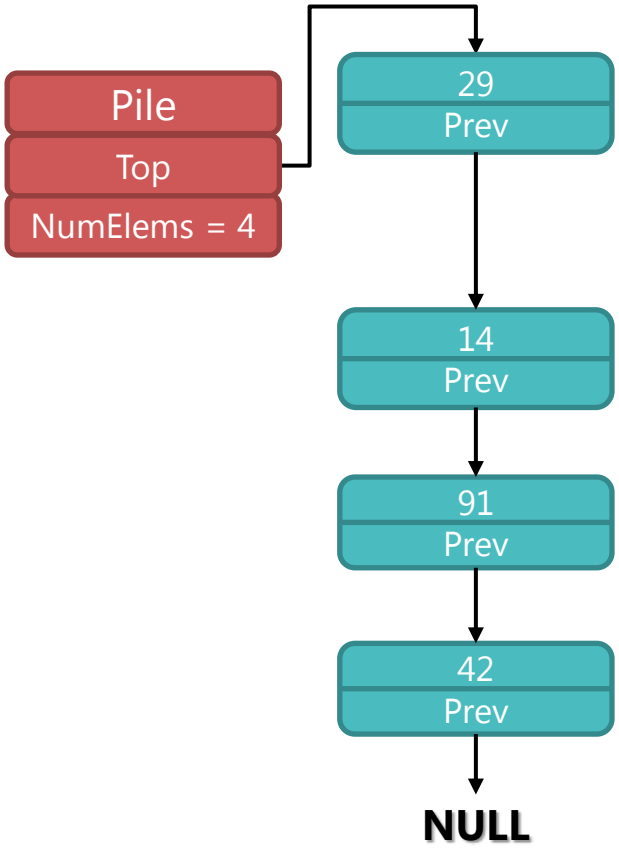

```
PTNode TStack_Push(const PTStack this, TElement Element)
{
  PTNode newNode = (PTNode)malloc(sizeof(TNode));
  if (newNode == NULL) return NULL;
  newNode->Element = Element;
  newNode->Prev = this->Top;
```

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



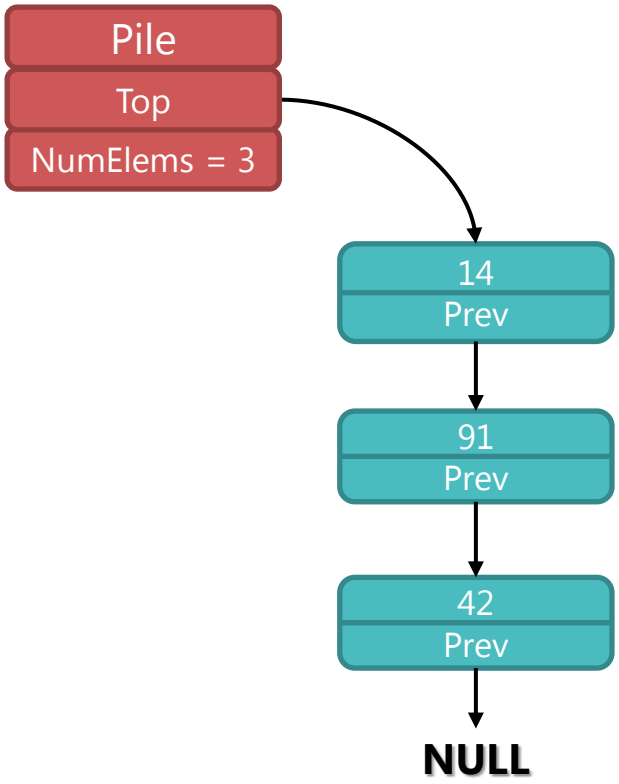

```
PTNode TStack_Push(const PTStack this, TElement Element)
{
  PTNode newNode = (PTNode)malloc(sizeof(TNode));
  if (newNode == NULL) return NULL;
  newNode->Element = Element;
  newNode->Prev = this->Top;
  this->Top = newNode;
```

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



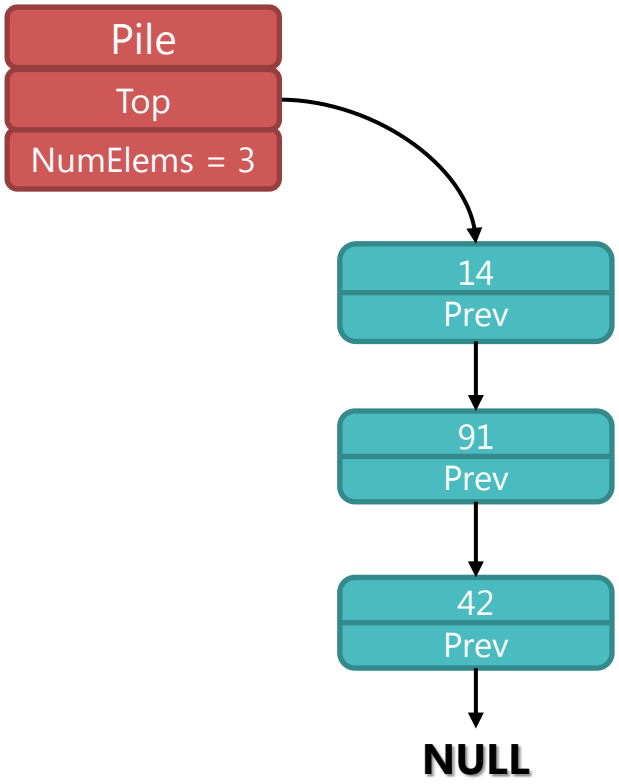

```
PTNode TStack_Push(const PTStack this, TElement Element)
{
  PTNode newNode = (PTNode)malloc(sizeof(TNode));
  if (newNode == NULL) return NULL;
  newNode->Element = Element;
  newNode->Prev = this->Top;
  this->Top = newNode;
  this->NumElems++;
  return newNode;
}
```

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;
```

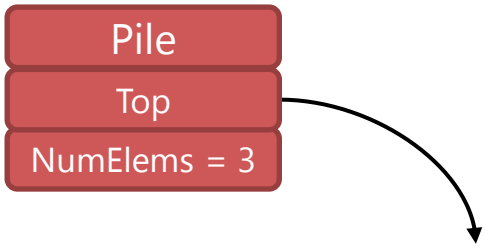
Fonction TStack_Pop : retrait de l'élément au sommet de la pile (2/5)



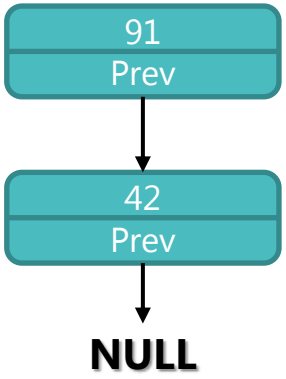
14

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

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

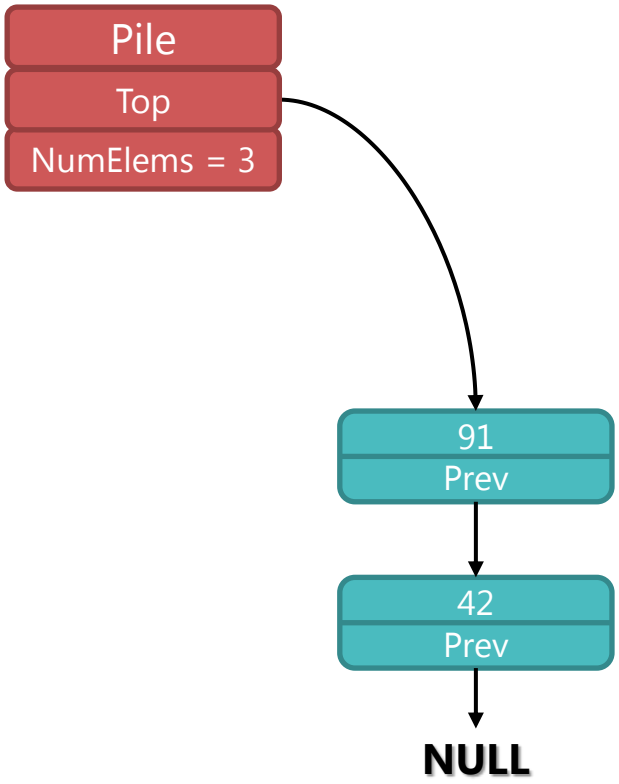


14




```
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);
```

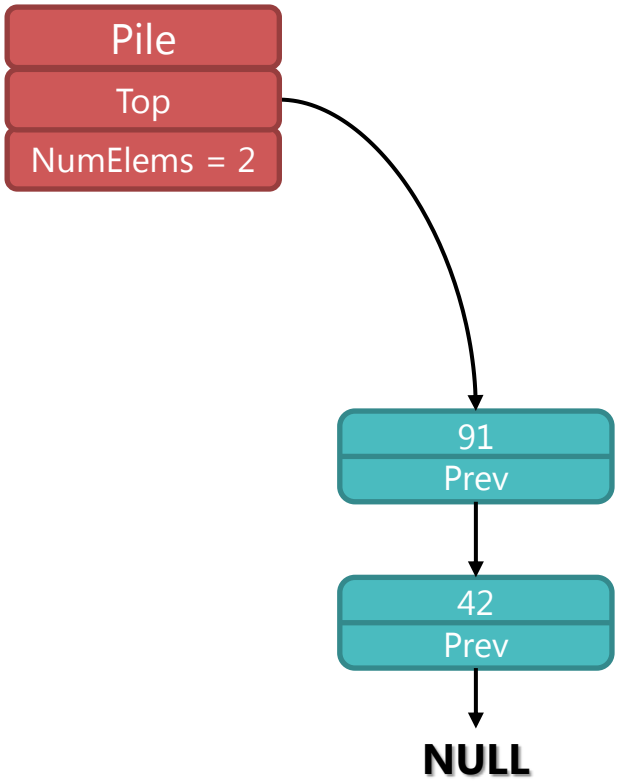
Fonction TStack_Pop : retrait de l'élément au sommet de la pile (4/5)



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;
```

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



14

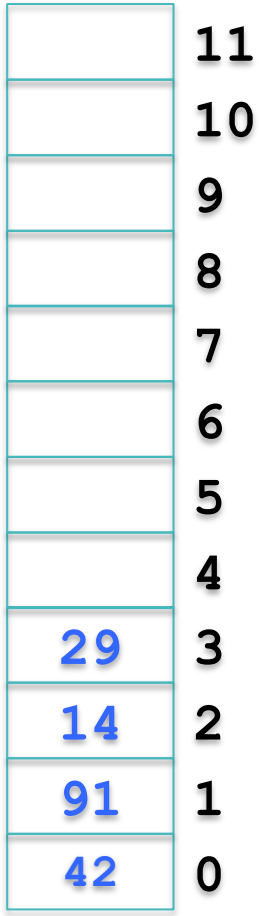
```
bool TStack_Pop(const PTStack this, TElement *Element)
{
  if (TStack_IsEmpty(this)) return false;
  PTNode pNode = this->Top, prevNode = pNode->Prev;
  *Element = pNode->Element;
  free(pNode);
  this->Top = prevNode;
  this->NumElems--;
  return true;
}
```

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.

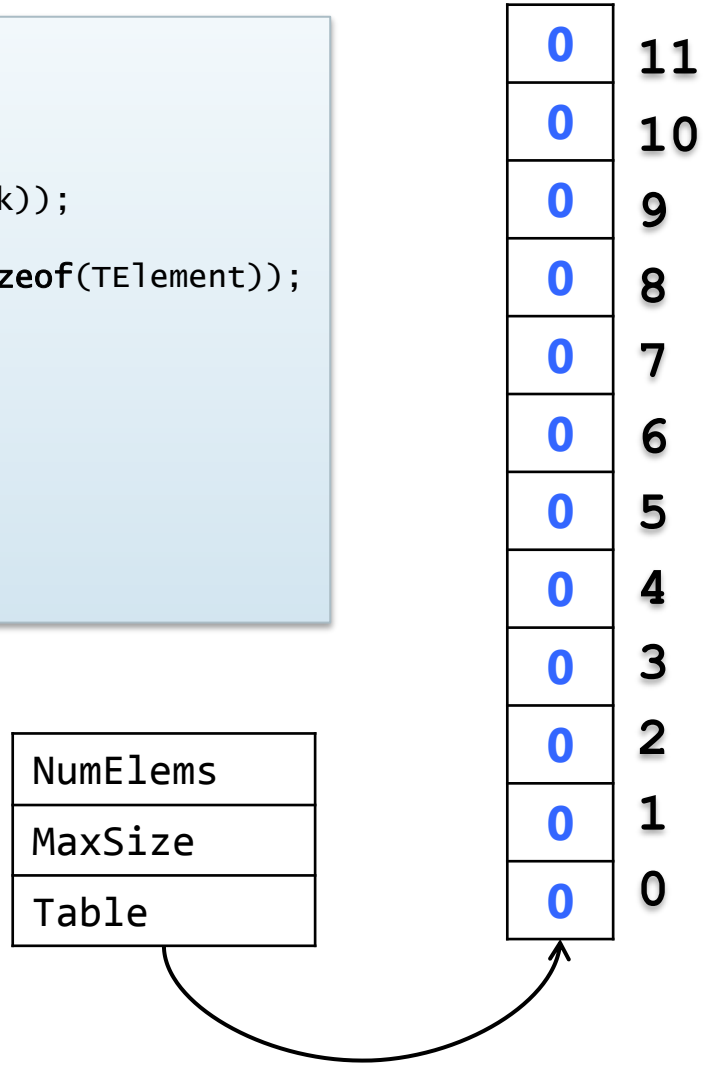
```
typedef struct Stack {
  int NumElems;              /* Nombre d'éléments */
  int MaxSize;               /* Taille maximale */
  TElement *Table;           /* Contenu */
} TStack;

typedef TStack *PTStack;
```



PILES ET FILES

```c
/* 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;
}
```

```c
/* Libération dans l'ordre inverse */

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

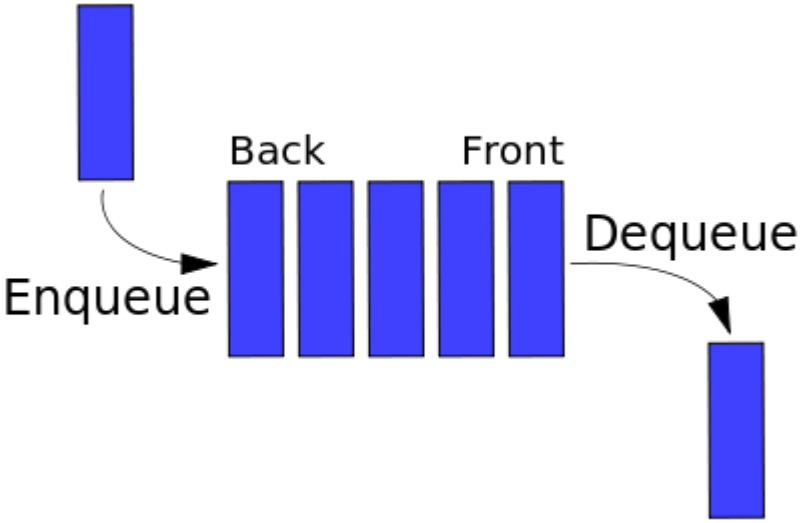


Les files (queue)

- 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 : *Fisrt 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.

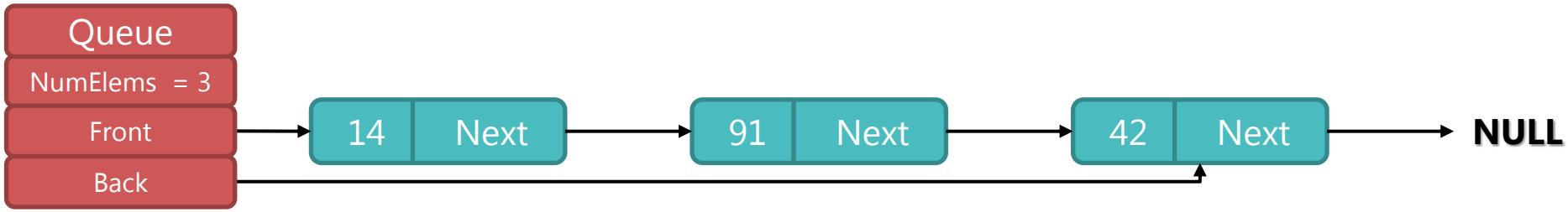
- Un exemple d'utilisation est la file d'attente des tâches à imprimer dans un *spooler* d'impression



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` ↔ `Add`
  - `Dequeue` ↔ `RemoveFirst`
- Ces choix doivent permettre une implémentation efficace des files d'attente en C.

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;
```

Opérations sur le TAD file

- **Nom** : TQueue,
- **Utilise** : **int**, **bool**, TElement, TNode
- **Opérations** :

```
New → TQueue
IsEmpty : TQueue → bool
Length : TQueue → int
Enqueue : TQueue x TElement → TNode
Dequeue : TQueue → bool x TElement
Clear : TQueue →
Display : TQueue →
Delete : TQueue →
```

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.

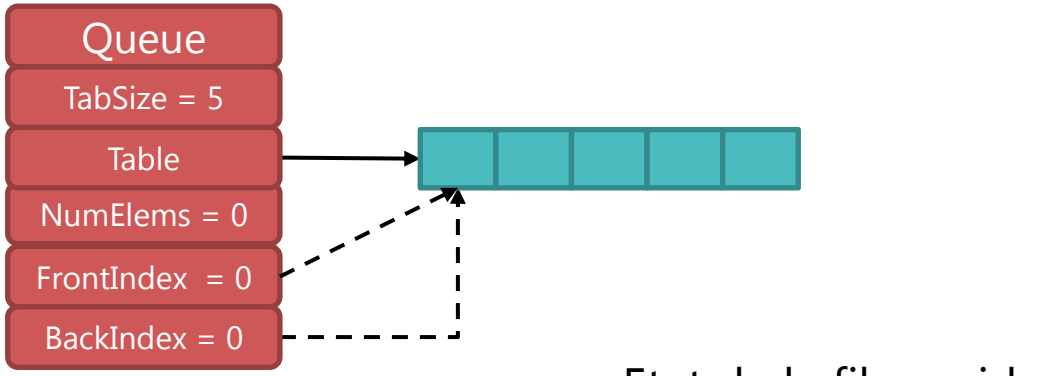
- Il faut en effet gérer simultanément deux indices, correspondant respectivement à la tête et à la queue de la file.

- 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 ;
```

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.

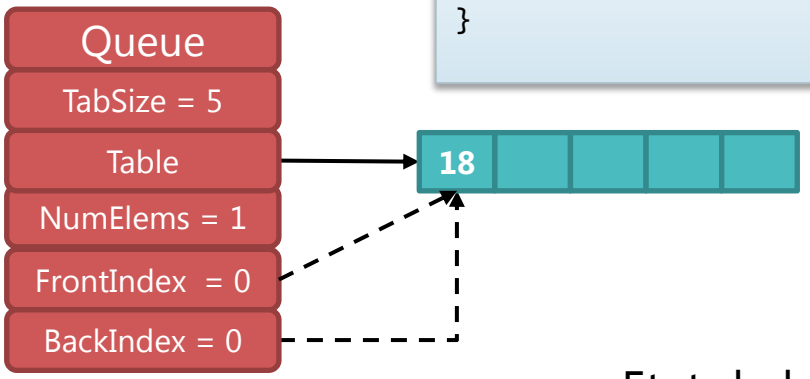


Etat de la file = vide

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*.

```
if (this->NumElems == 0) {
  this->Table[0] = Element;
  this->NumElems = 1;
}
```

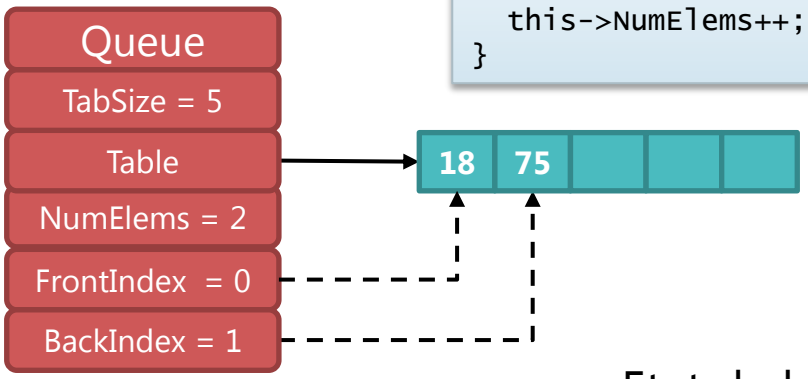


Etat de la file = 18

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.

```
else {      /* this->NumElems != 0 */
  this->BackIndex = (this->BackIndex + 1) % this->TabSize;
  this->Table[this->BackIndex] = Element;
  this->NumElems++;
}
```

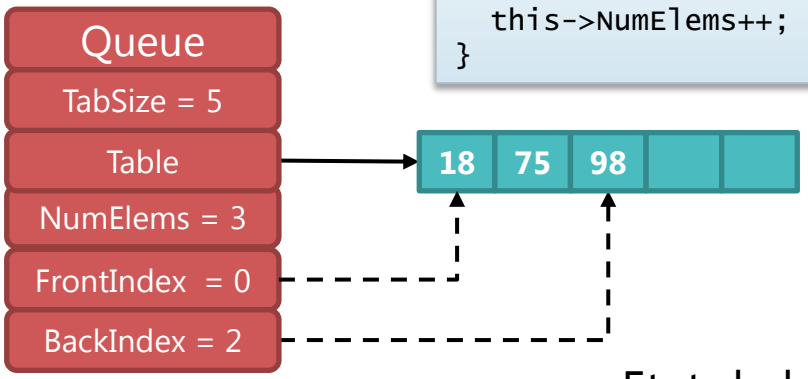


Etat de la file = 18, 75

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.

```
else {      /* this->NumElems != 0 */
  this->BackIndex = (this->BackIndex + 1) % this->TabSize;
  this->Table[this->BackIndex] = Element;
  this->NumElems++;
}
```

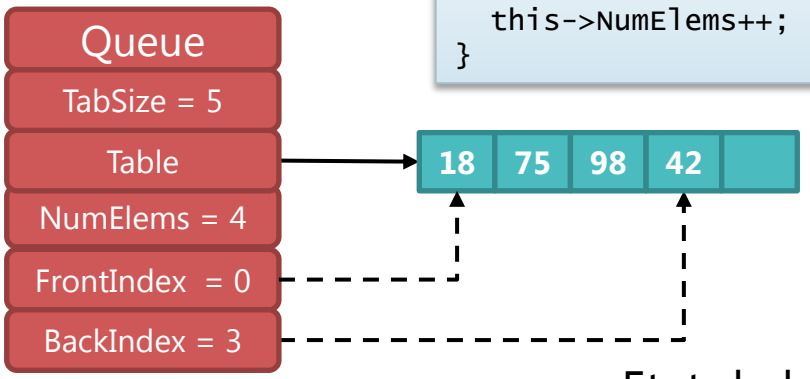


Etat de la file = 18, 75, 98

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.

```
else {     /* this->NumElems != 0 */
  this->BackIndex = (this->BackIndex + 1) % this->TabSize;
  this->Table[this->BackIndex] = Element;
  this->NumElems++;
}
```

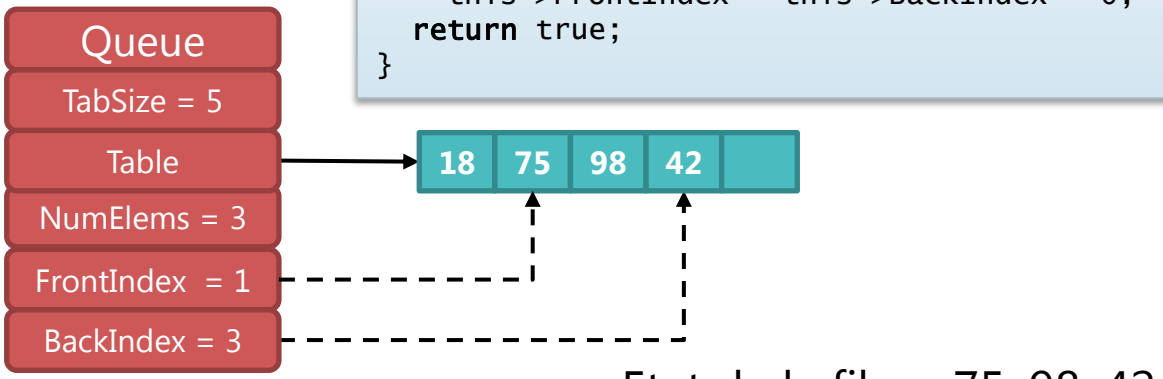


Etat de la file = 18, 75, 98, 42

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;
}
```

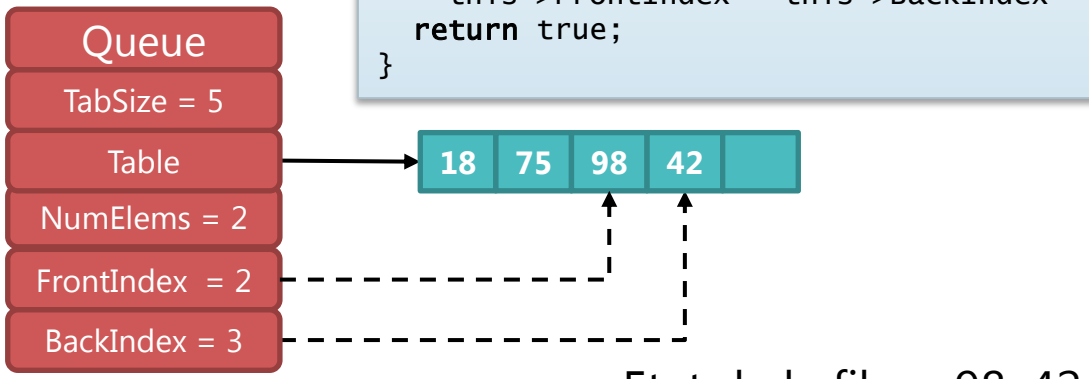


Etat de la file = 75, 98, 42

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;
}
```

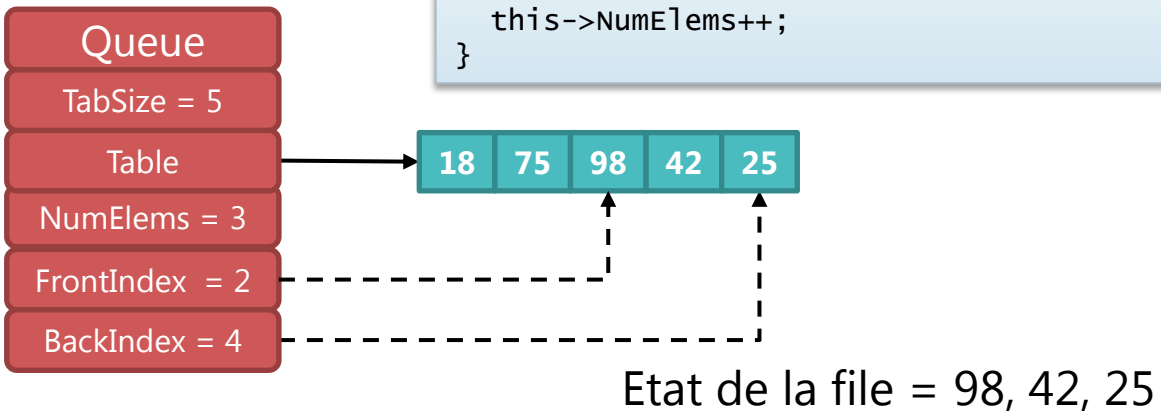


Etat de la file = 98, 42

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 ».

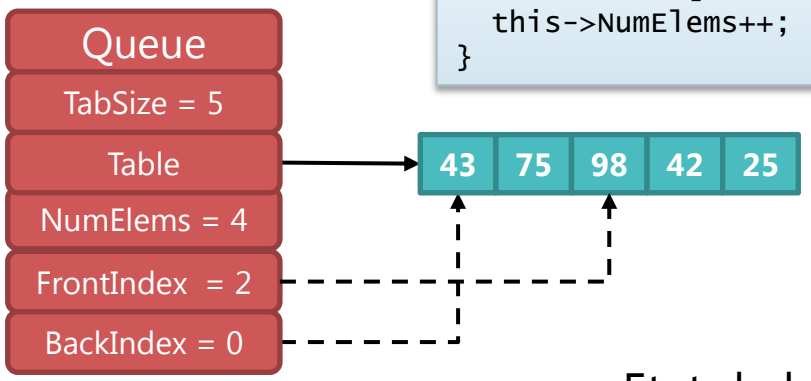
```
else {      /* this->NumElems != 0 */
  this->BackIndex = (this->BackIndex + 1) % this->TabSize;
  this->Table[this->BackIndex] = Element;
  this->NumElems++;
}
```



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 ».

```
else {      /* this->NumElems != 0 */
  this->BackIndex = (this->BackIndex + 1) % this->TabSize;
  this->Table[this->BackIndex] = Element;
  this->NumElems++;
}
```



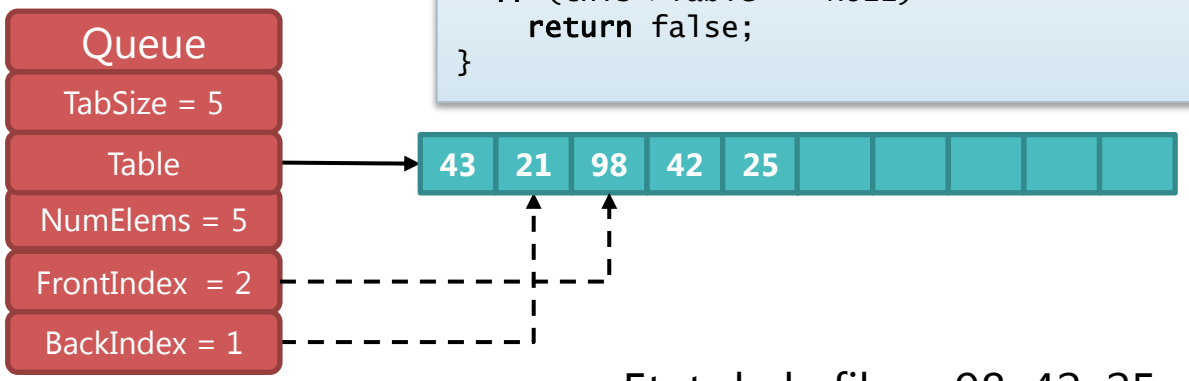
Etat de la file = 98, 42, 25, 43

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;
}
```



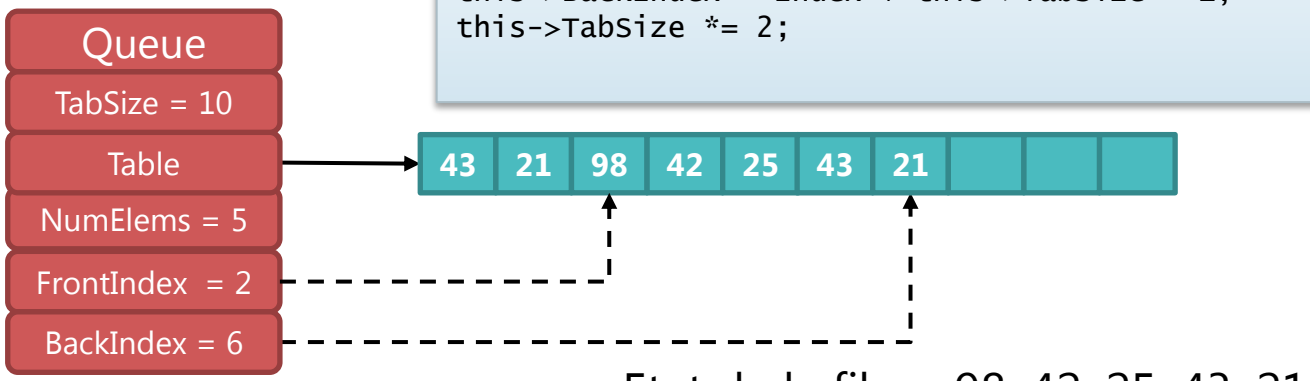
Etat de la file = 98, 42, 25, 43, 21

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 :

```
int Index;
for (Index = 0; Index < this->FrontIndex; Index++)
   this->Table[Index + this->TabSize] = this->Table[Index];
this->BackIndex = Index + this->TabSize - 1;
this->TabSize *= 2;
```

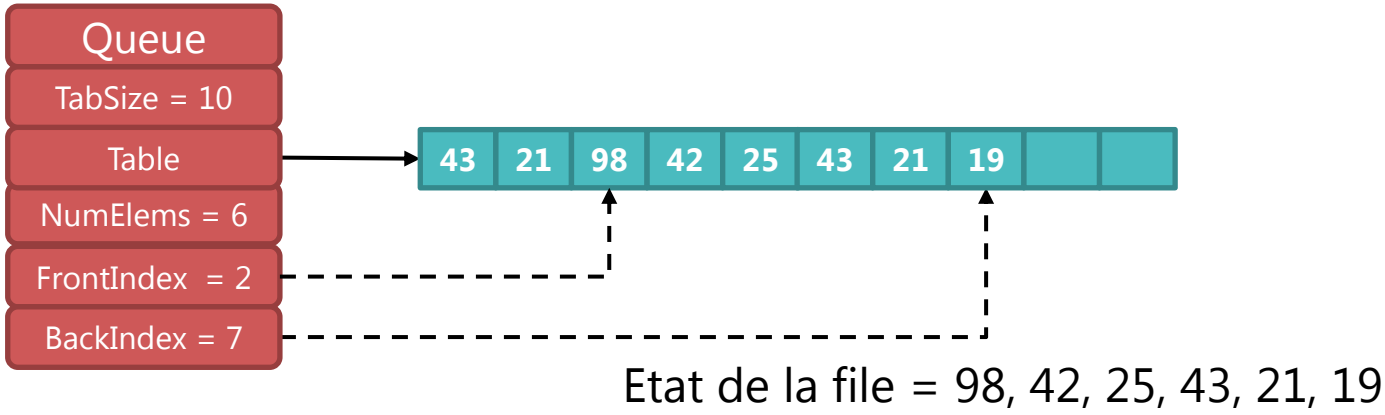


Etat de la file = 98, 42, 25, 43, 21

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

- Rappels et approfondissements en C
- 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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

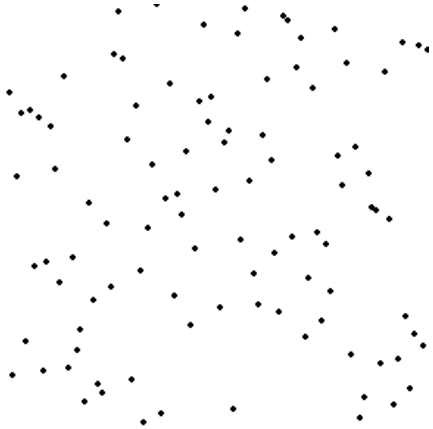
ALGORITHMES DE TRI

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

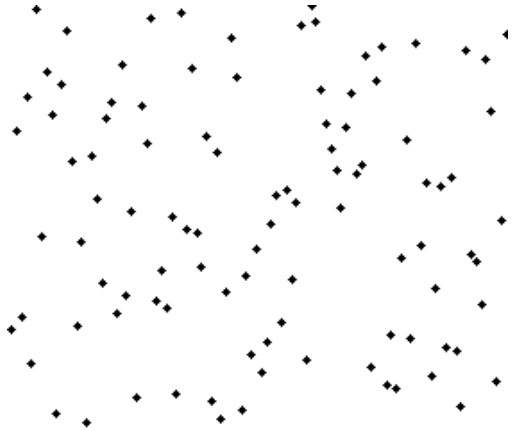
- 18 75 98 42 25 31 55 24 7 31 12 7
- 7 75 98 42 25 31 55 24 18 31 12 7
- 7 7 98 42 25 31 55 24 18 31 12 75
- 7 7 12 42 25 31 55 24 18 31 98 75
- 7 7 12 18 25 31 55 24 42 31 98 75
- 7 7 12 18 24 31 55 25 42 31 98 75
- 7 7 12 18 24 25 55 31 42 31 98 75
- 7 7 12 18 24 25 31 55 42 31 98 75
- 7 7 12 18 24 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 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];
        int 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 équiprobables ($n/2$) 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

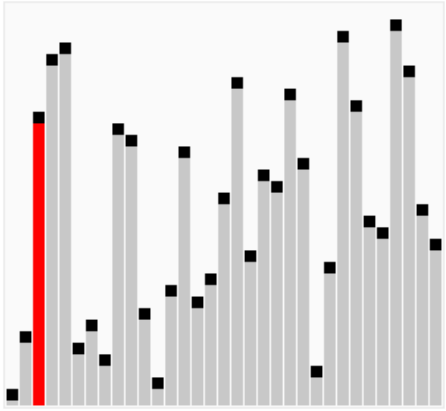
- 18 75 98 42 25 31 55 24 7 31 12 7
- 18 75 98 42 25 31 55 24 7 31 12 7
- 18 75 98 42 25 31 55 24 7 31 12 7
- 18 42 75 98 25 31 55 24 7 31 12 7
- 18 25 42 75 98 31 55 24 7 31 12 7
- 18 25 31 42 75 98 55 24 7 31 12 7
- 18 25 31 42 55 75 98 24 7 31 12 7
- 18 24 25 31 42 55 75 98 7 31 12 7
- 7 18 24 25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

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
- 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;
      }
    }
  }
}
```

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.
- 1ère 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 $O(n^2)$.
- 2ème 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*.

Tri fusion

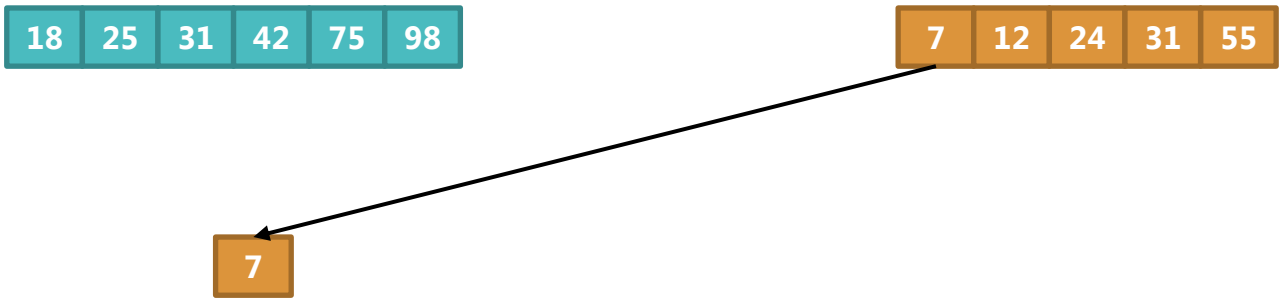
- Cet algorithme utilise le principe « *diviser pour régner* » qui consiste à diviser un problème en sous-problèmes plus simples à résoudre, puis à trouver une solution au problème initial à partir des sous-problèmes résolus.
- Principe de la fusion : à partir de deux collections de données (tableaux 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*.

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.

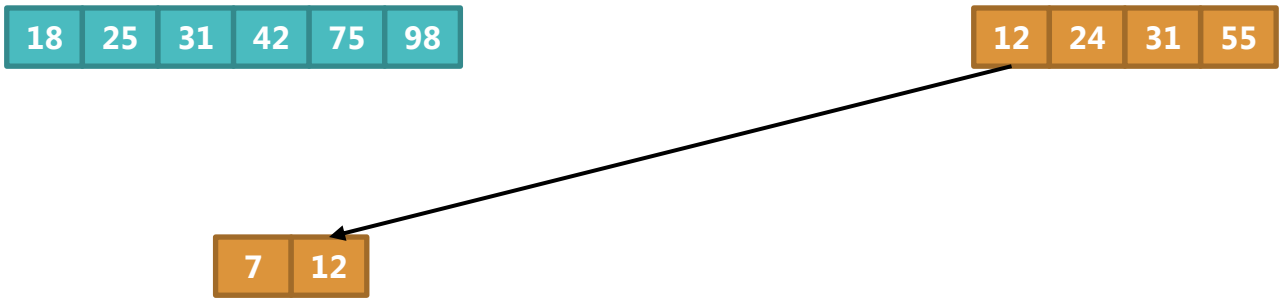
Principe de la fusion (2/11)

- 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.



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.

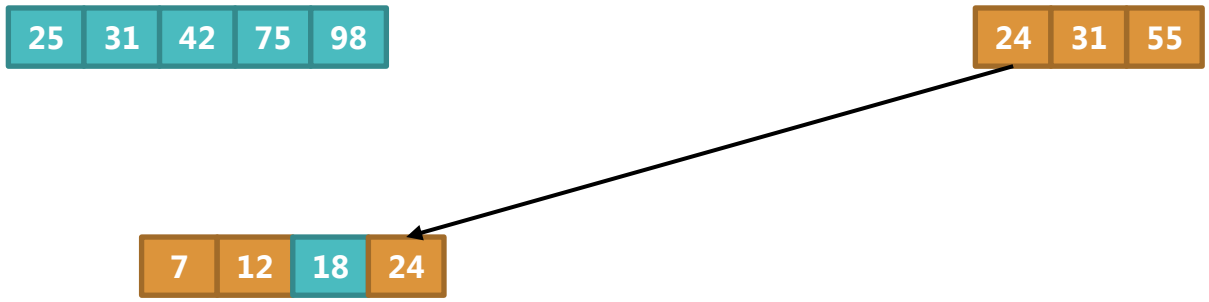


Principe de la fusion (4/11)

- 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.

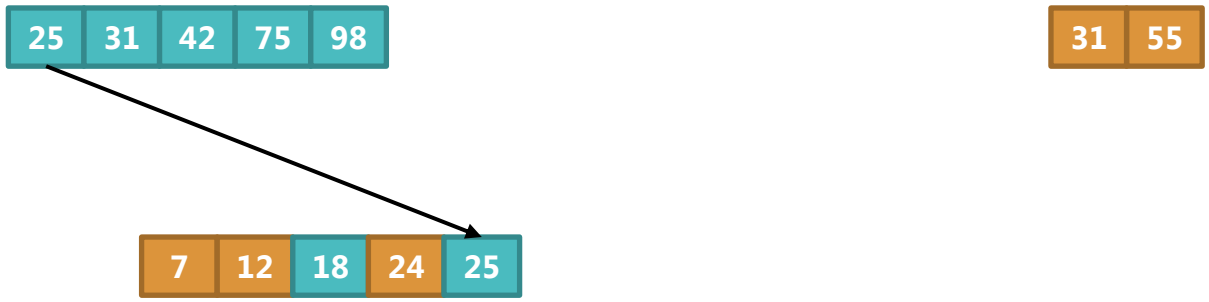
Principe de la fusion (5/11)

- 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.



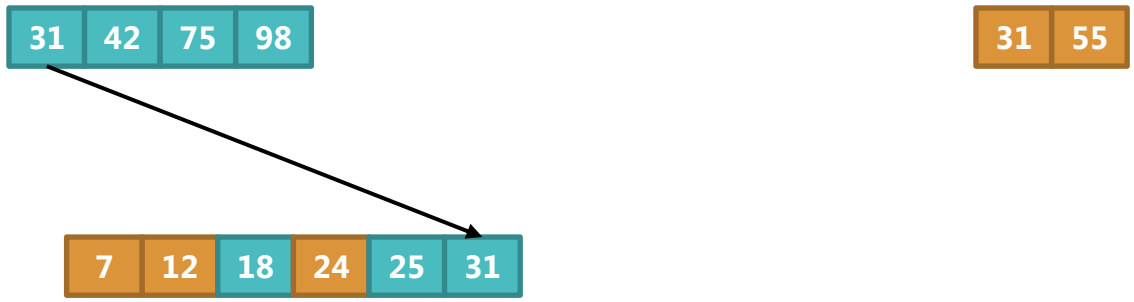
Principe de la fusion (6/11)

- 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.



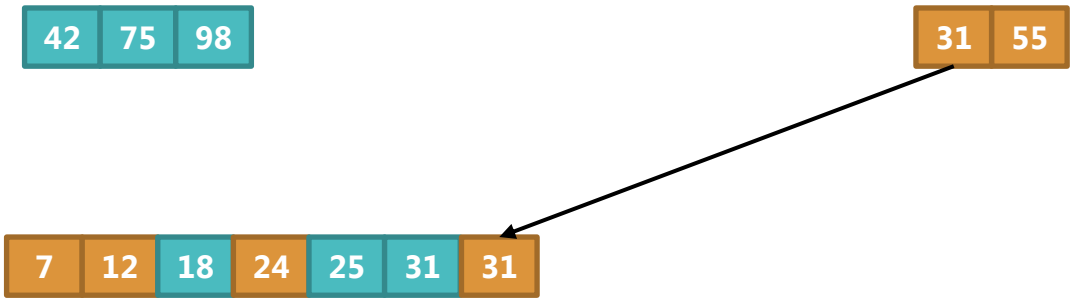
Principe de la fusion (7/11)

- 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.



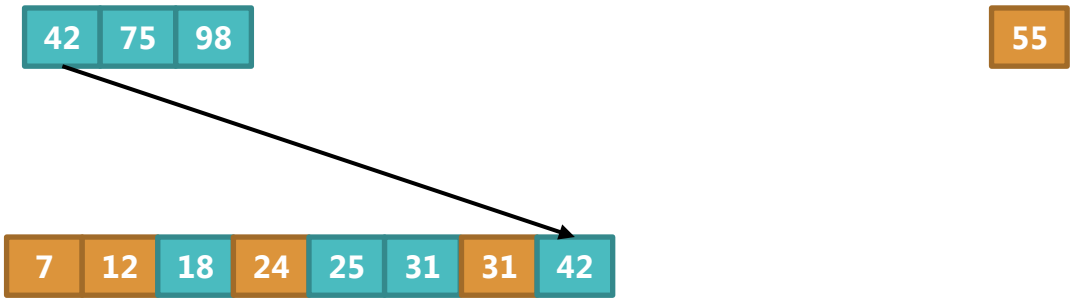
Principe de la fusion (8/11)

- 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.



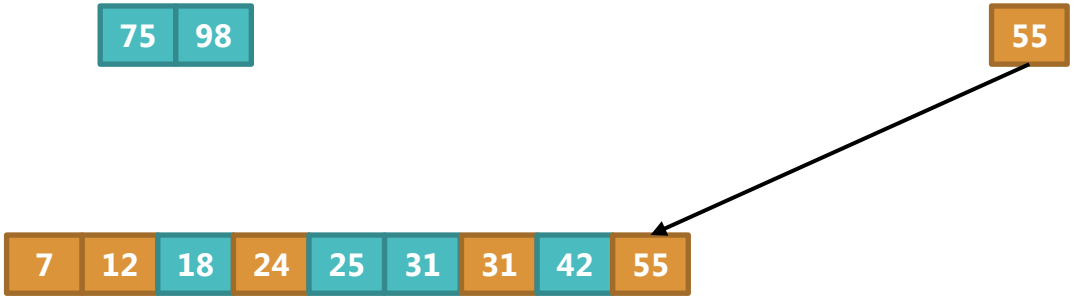
Principe de la fusion (9/11)

- 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.



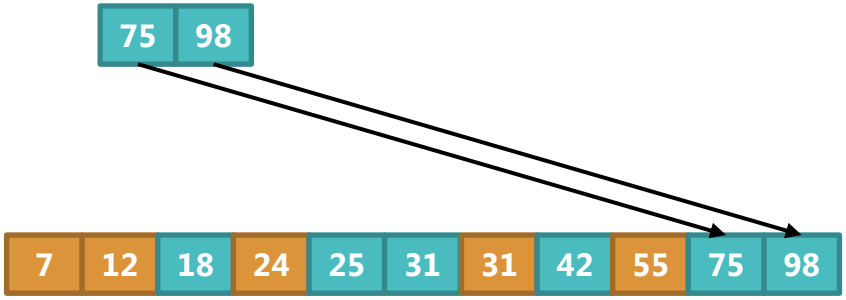
Principe de la fusion (10/11)

- 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.



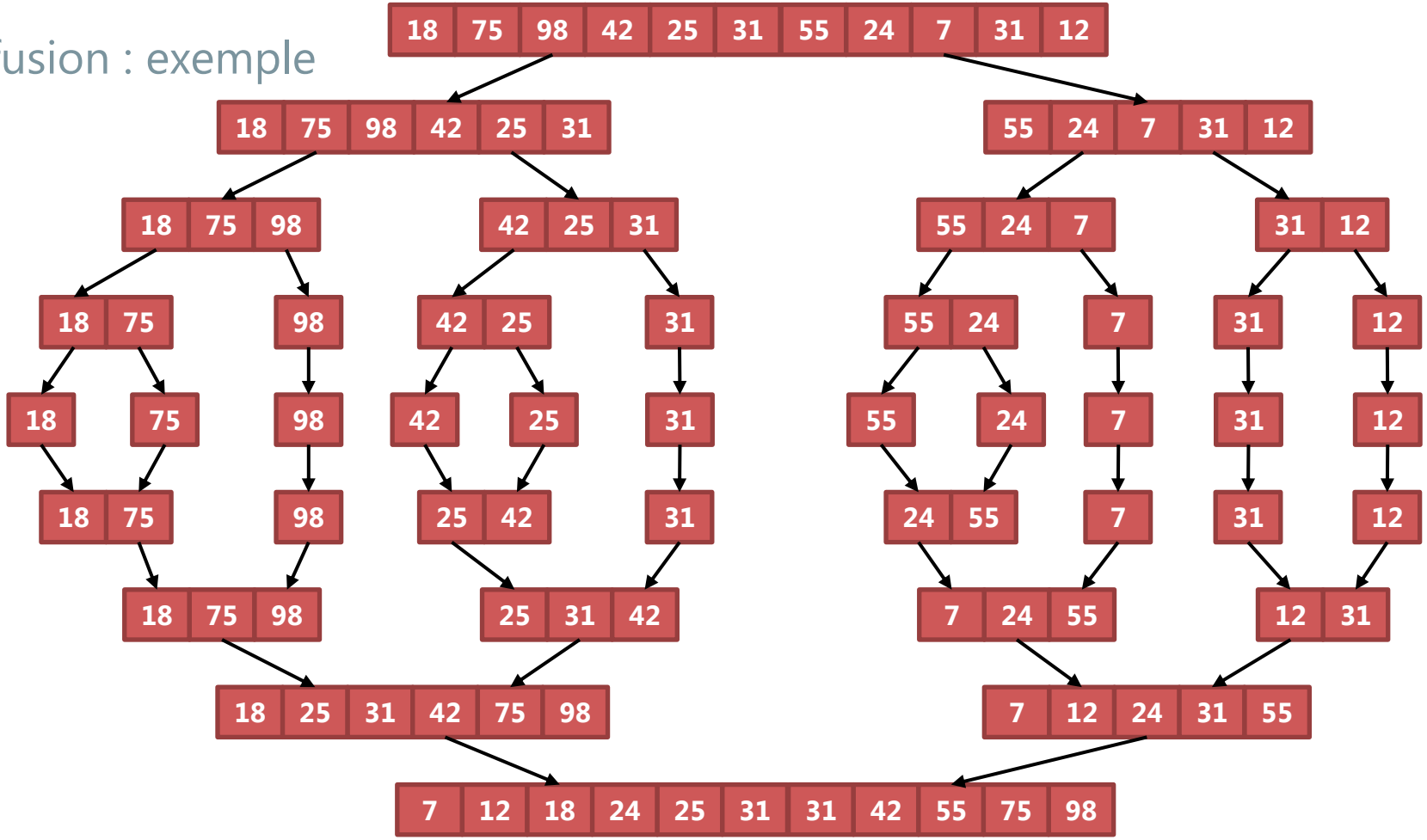
Principe de la fusion (11/11)

- 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



Algorithme de fusion de deux tableaux triés : pseudo-code

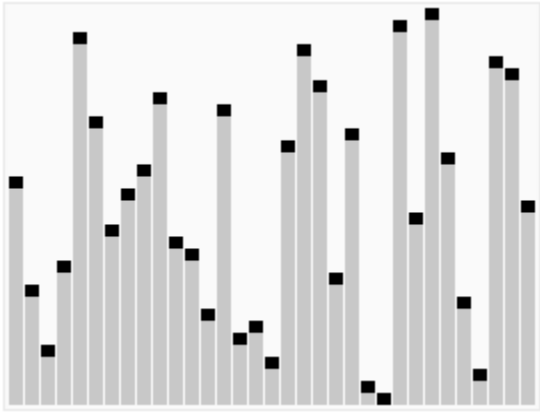
```
tableau algo_fusion(tableau gauche, tableau droit, entier tailleG, entier tailleD)
debut
  entier taille = tailleG + tailleD
  tableau fusion[taille]
  entier i, g = 0, d = 0
  pour i = 0 à taille-1 faire
    si gauche[g] > droit[d] alors
      fusion[i] = droit[d]; d = d + 1
    sinon
      fusion[i]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



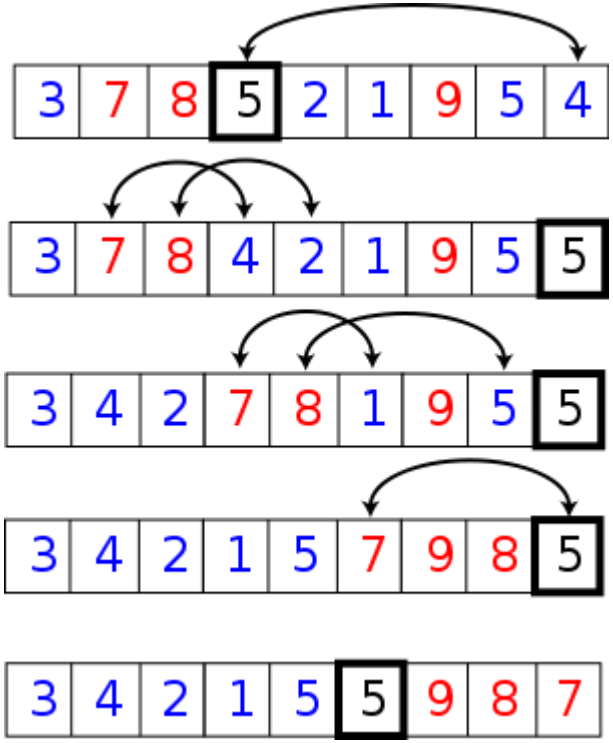
Quick sort

- 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 *));
```

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 élément 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 :
  1. Echanger le pivot avec le dernier élément,
  2. Placer tous les éléments inférieurs ou égaux au début,
  3. Echanger le pivot avec le 1er élément supérieur.



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
```

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

```
procedure quick_sort(tableau T, entier n)
debut
  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
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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 élément 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.

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

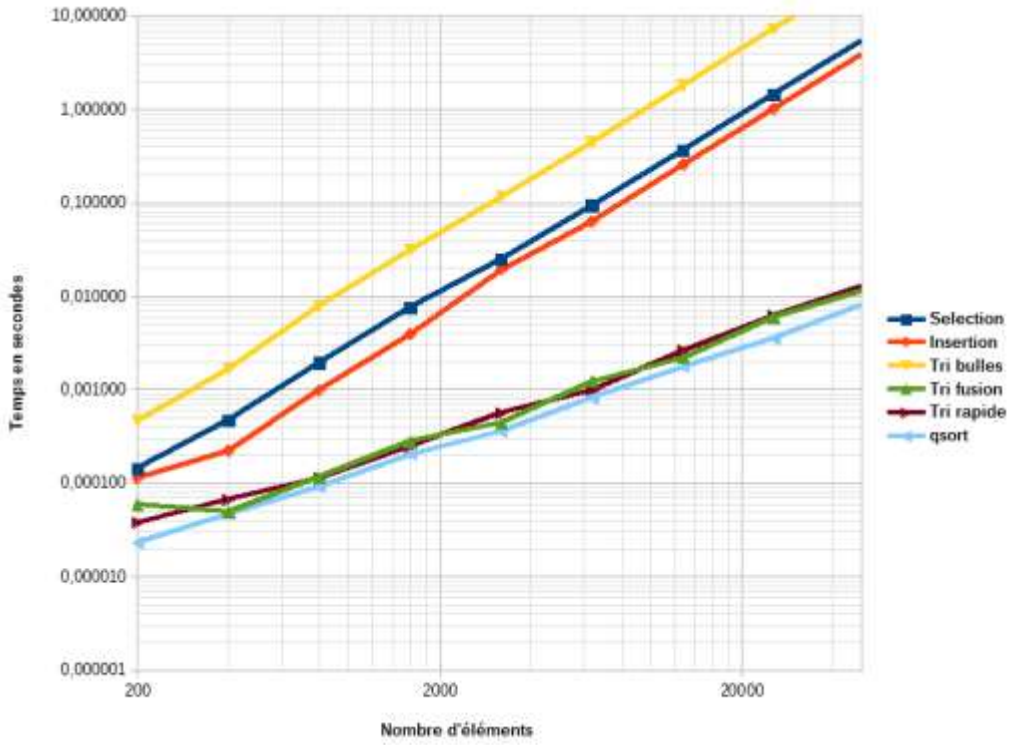
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

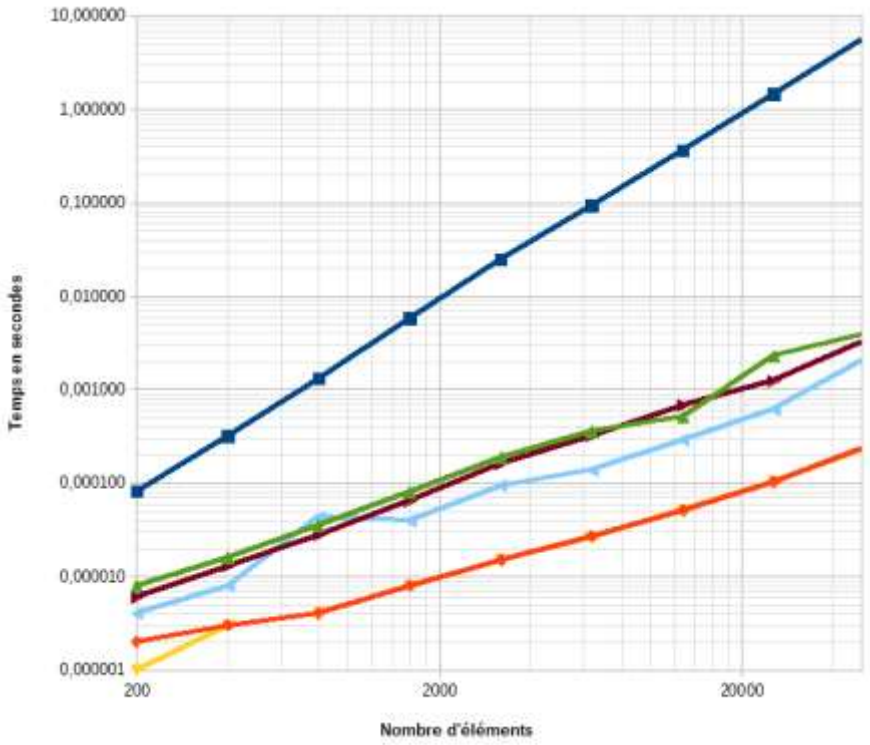
Taille	Tri par sélection	Tri par insertion	Tri à bulles	Tri fusion	Tri rapide	Fonction qsort
100	0,000025	0,000031	0,000165	0,000009	0,000006	0,000004
200	0,000110	0,000128	0,000570	0,000016	0,000010	0,000006
400	0,000375	0,000241	0,002861	0,000062	0,000042	0,000028
800	0,001370	0,001669	0,010450	0,000073	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



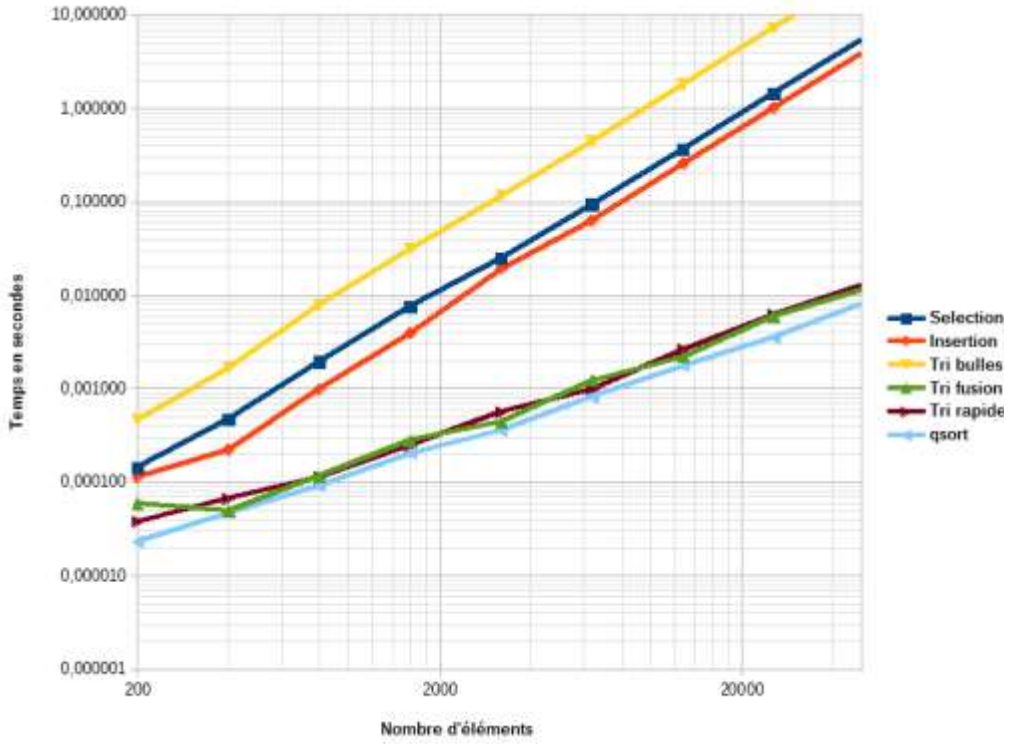
Données aléatoires (cas moyen)



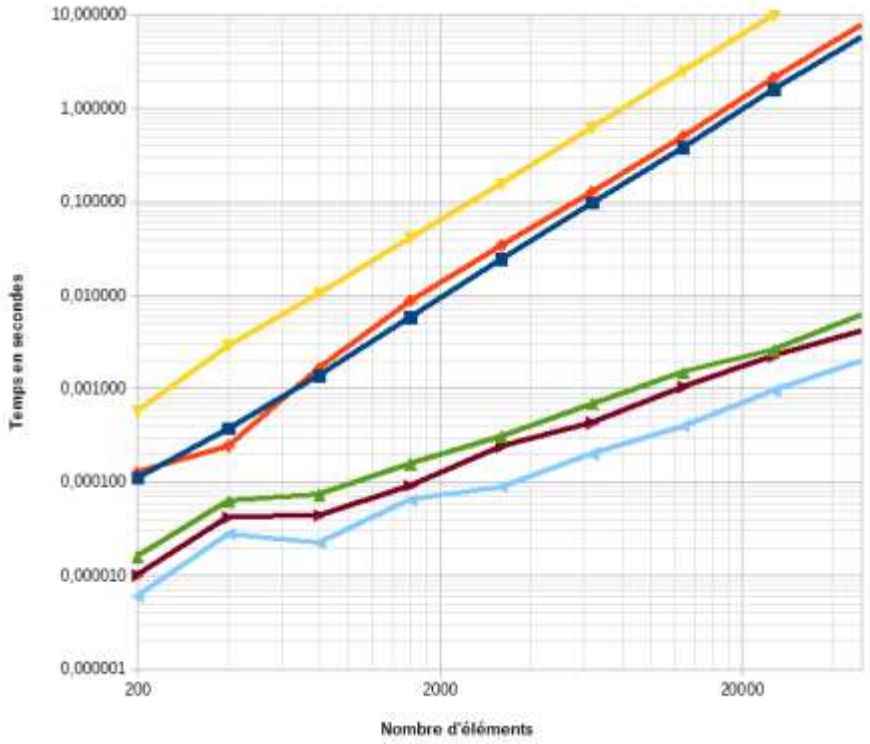
Données déjà triées

ALGORITHMES DE TRI

Données aléatoires vs données triées à l'envers



Données aléatoires (cas moyen)



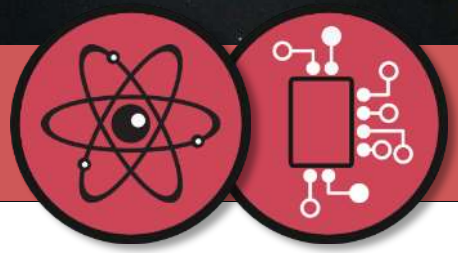
Données triées à l'envers

PLAN DU COURS

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

- Rappels et approfondissements en C
- Listes chainées
- Piles et files
- Algorithmes de tri
- **Tables de hachage**
- Arbres binaires

Tables de hachage



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

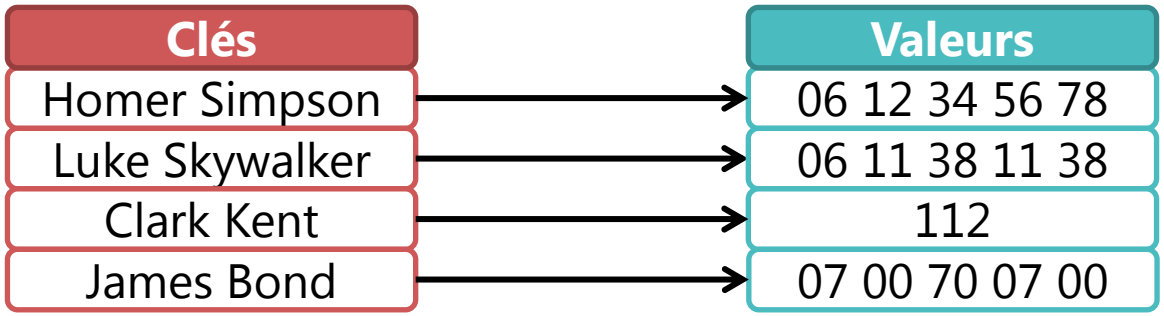
Motivations : tableaux vs listes chainées

- 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

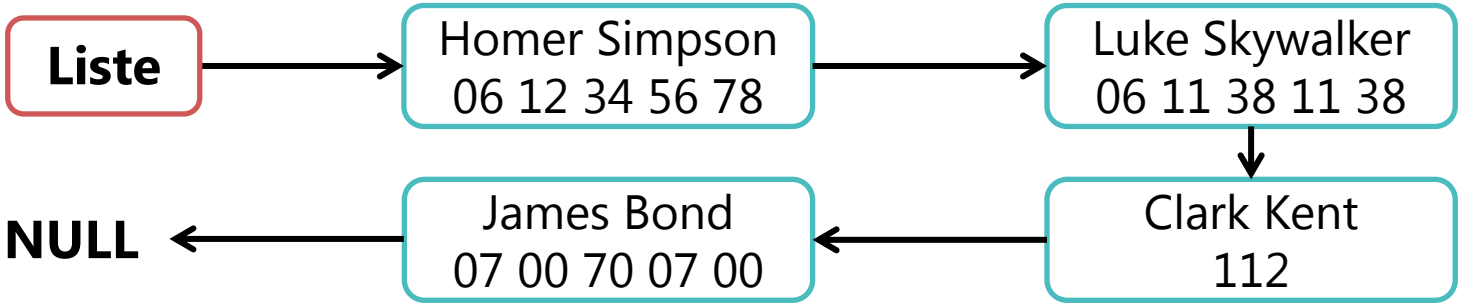
- Un tableau associatif est un TAD permettant d'associer une clé à une valeur : un exemple courant est le répertoire des contacts dans un téléphone mobile.



- 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).

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.

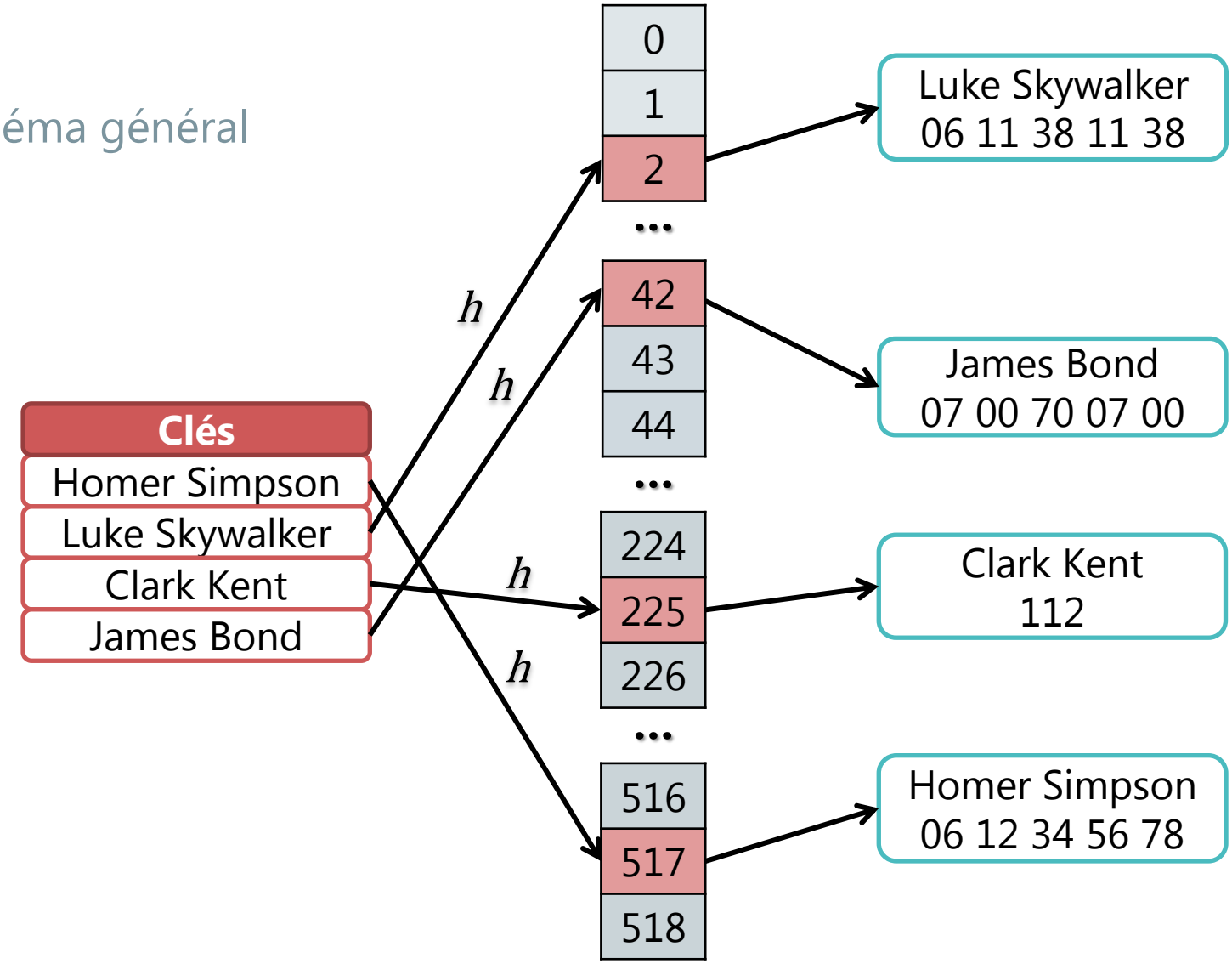
TABLES DE HACHAGE

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 1ère propriété est triviale : deux clés identiques donnent le même *hash code* (heureusement !).
- Par contre la 2nde 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*.

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

Probabilité de coïncidence

- Il y a k personnes dans la pièce : quelle est la probabilité $\overline{p_k}$ 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)

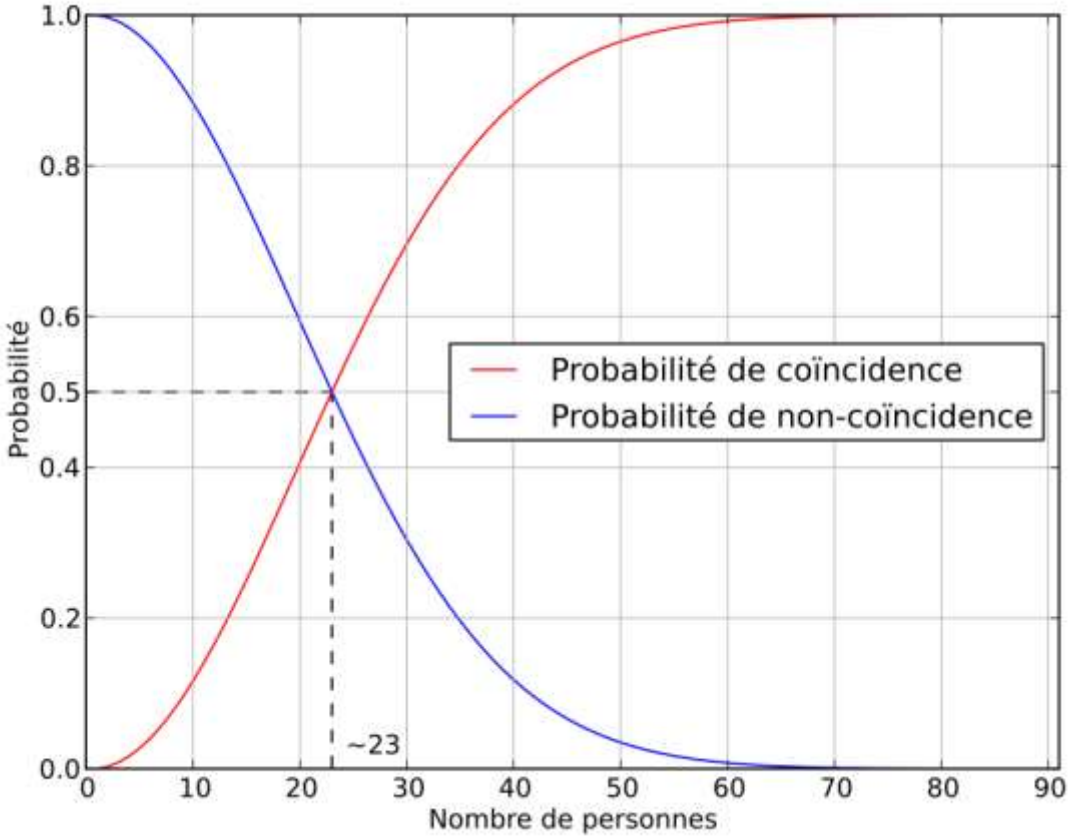
- $k = 2 : \overline{p}_2 = \frac{N-1}{N}$
- $k = 3 : \overline{p}_3 = \frac{N-1}{N} \times \frac{N-2}{N}$
- ....
- pour k quelconque on trouve : $\overline{p}_k = \frac{N!}{(N-k)!N^k}$

- d'où la probabilité de coïncidence : $p_k = 1 - \overline{p}_k = 1 - \frac{N!}{(N-k)!N^k}$

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 % !



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

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 entraine 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 :

$$h(k) = [\ k_0 B^{n-1} + k_1 B^{n-2} + \ldots + k_{n-2} B + k_{n-1}\ ] \bmod N$$

  - 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

Opérations sur le TAD table de hachage

- **Nom** : THashTable,
- **Utilise** : **int**, **bool**, TKey, TElement
- **Opérations** :

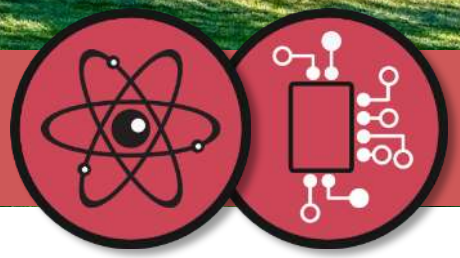
```
New → THashTable
IsEmpty : THashTable → bool
Size : THashTable → int
Put : THashTable x Tkey x TElement → bool
Remove : THashTable x Tkey → bool
Get : THashTable x TKey → TElement
Contains : THashTable x TKey → bool
Clear : THashTable →
Delete : THashTable →
```

PLAN DU COURS

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

- Rappels et approfondissements en C
- 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

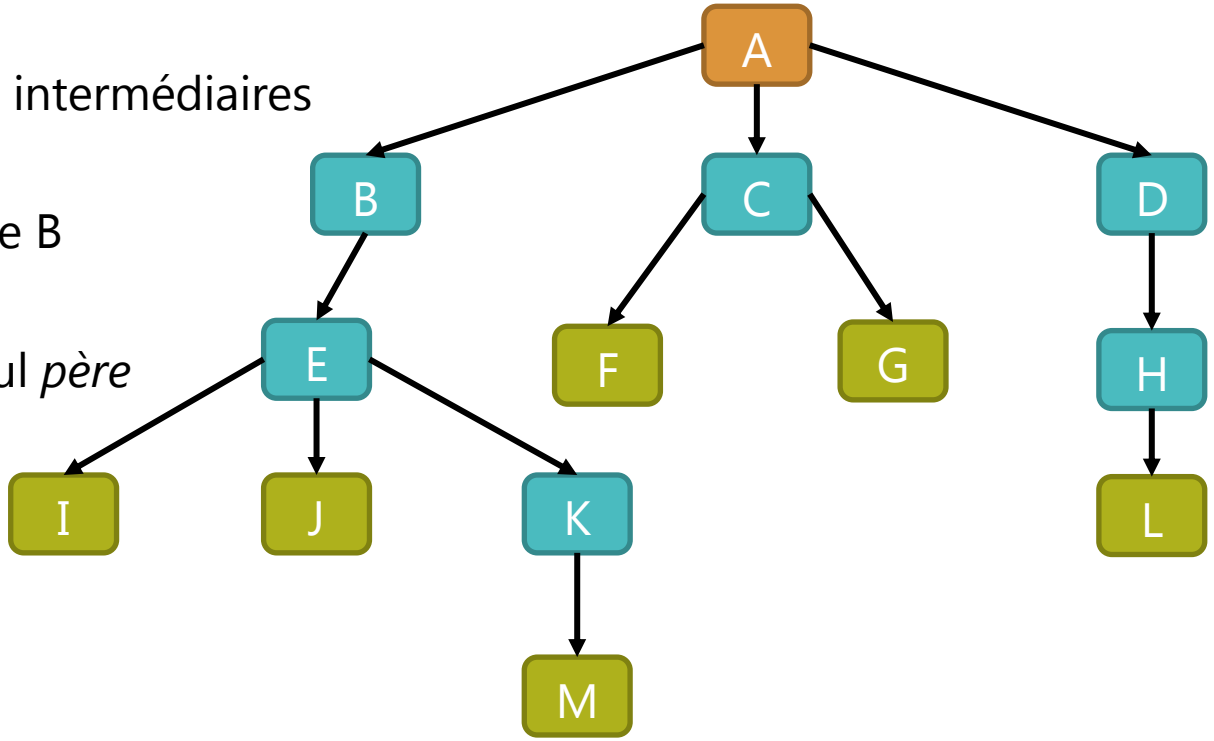
Tas et tri par tas

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

ARBRES BINAIRES

Arbre

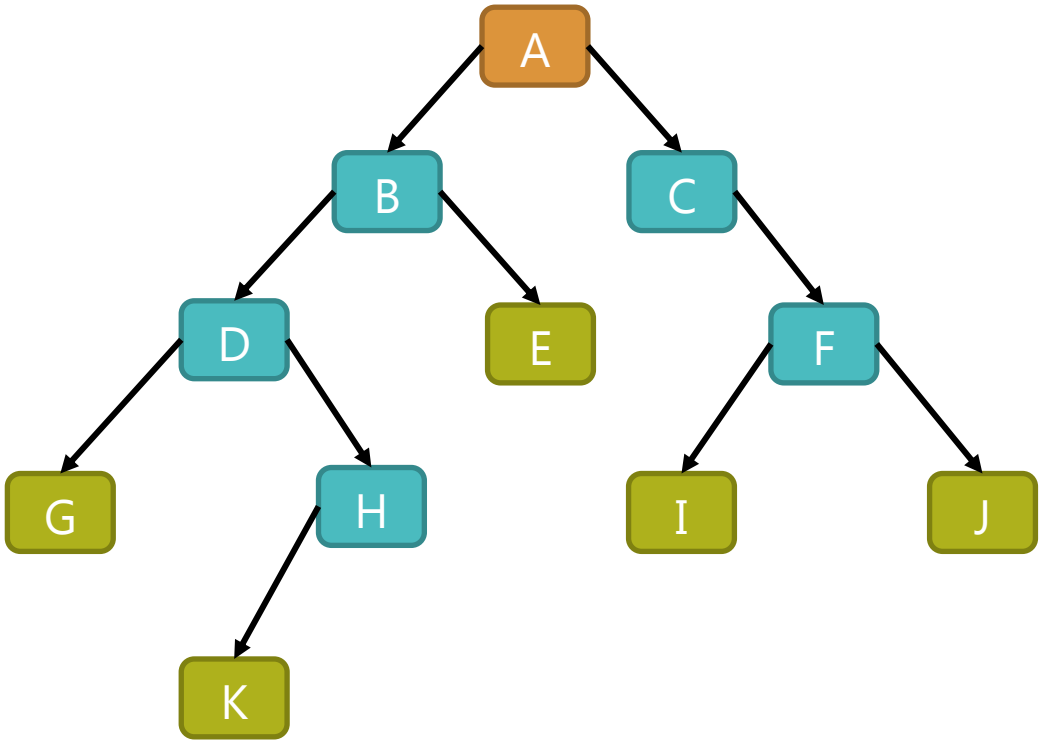
- Un *arbre* est une structure de données composée de *nœuds* (comme une liste chaînée) et de *feuilles* qui sont les nœuds terminaux.
- A : racine
- B, C, D, E, H, K : nœuds intermédiaires
- F, G, I, J, L, M : feuilles
- M est un *descendant* de B
- J est un *fils* de E
- Un nœud n'a qu'un seul *père*



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

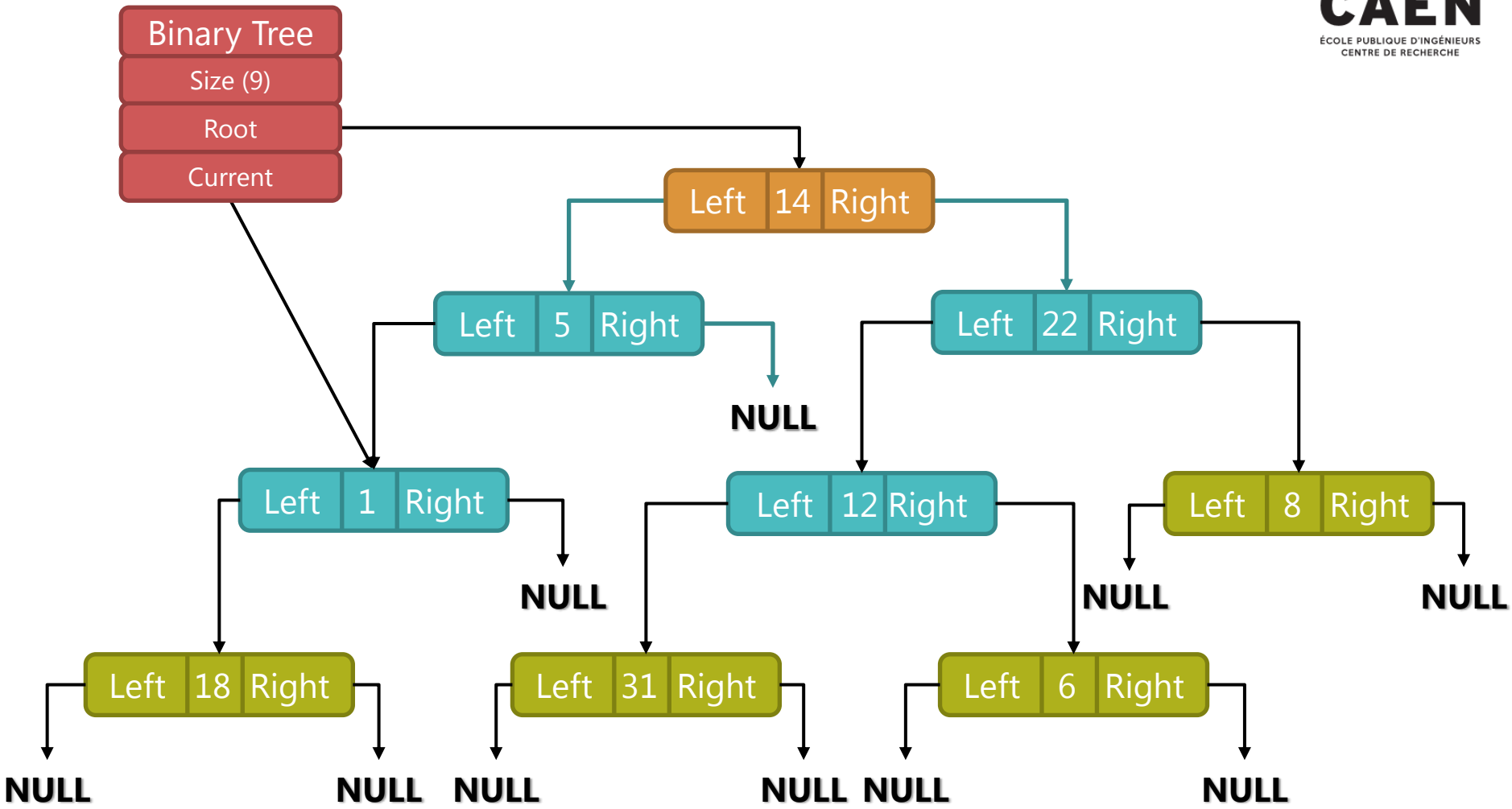
Implémentation à base de pointeurs

- 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.

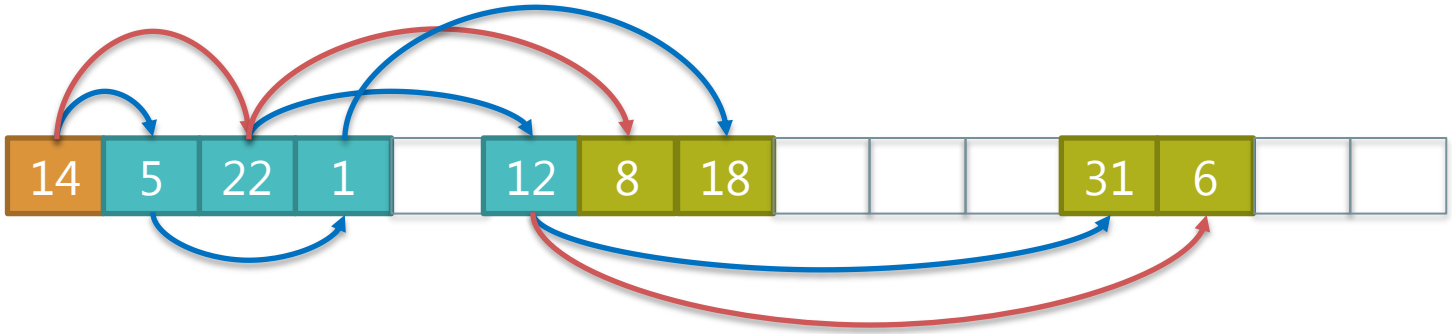
```
typedef struct Node {
  TElement Key;
  struct Node *LeftChild;
  struct Node *RightChild;
} TNode;

typedef TNode *PTNode;

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

typedef TBinaryTree *PTBinaryTree;
```

ARBRES BINAIRES



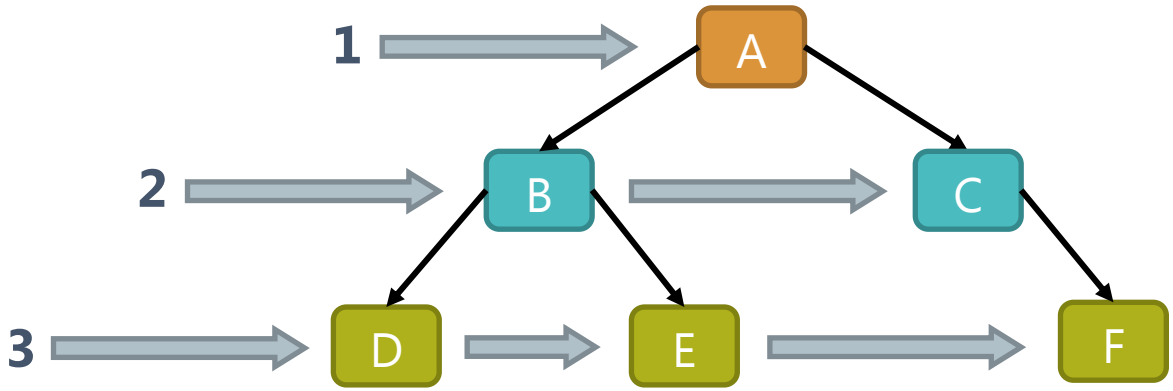
Implémentation à base de tableau

- 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.



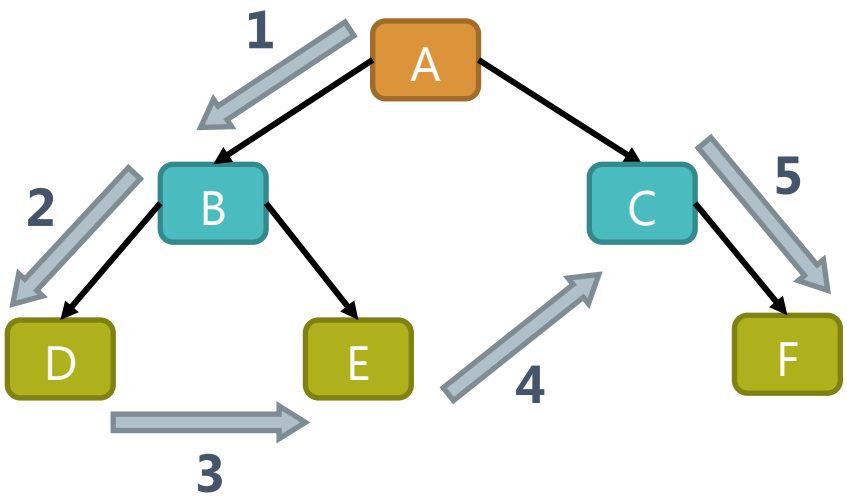
Parcours en largeur (*Breadth-first search*)

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



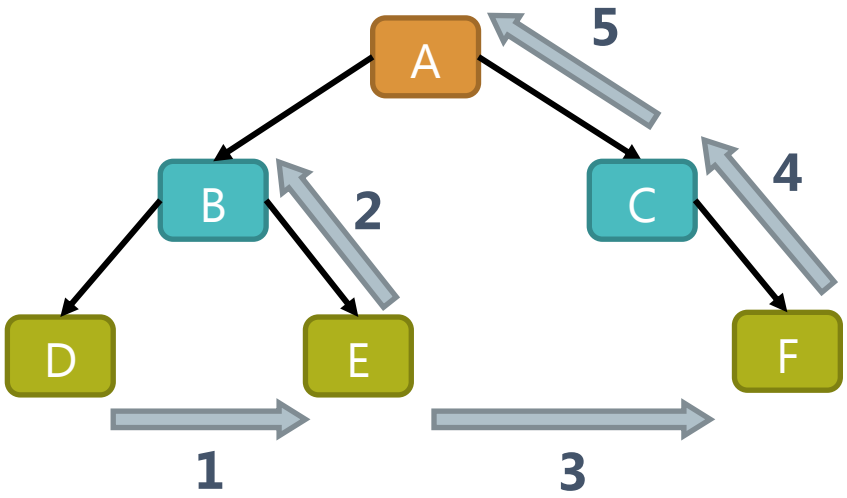
Parcours en profondeur (*Depth-first search*)

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



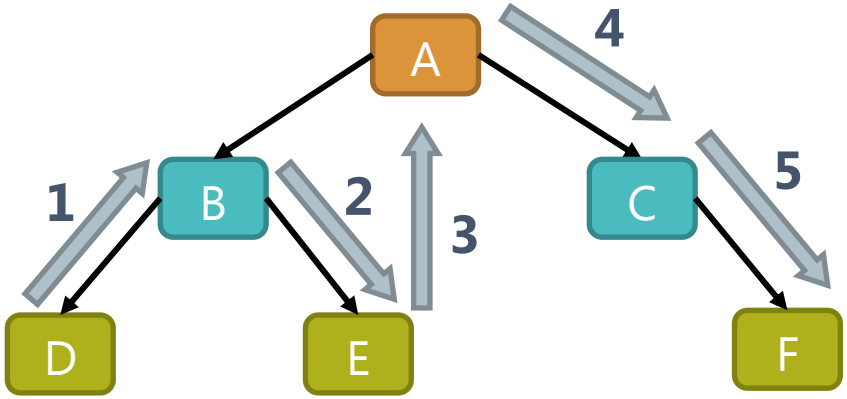
Parcours en profondeur (*Depth-first search*)

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



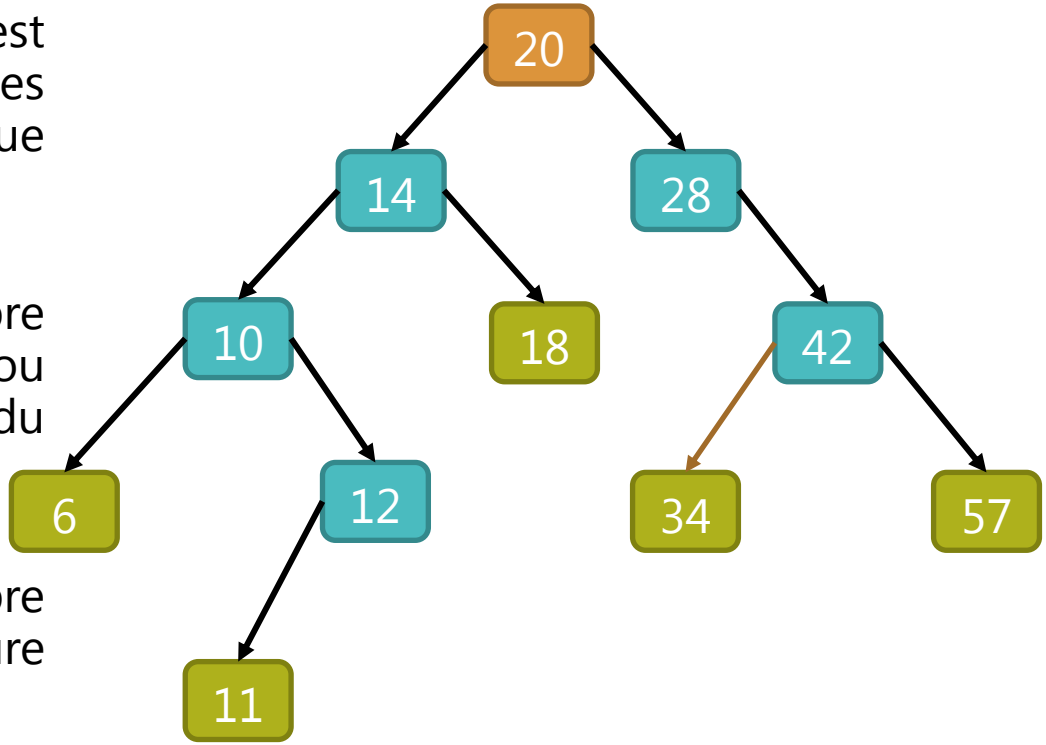
Parcours en profondeur (*Depth-first search*)

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



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.



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

Insertion des éléments dans un ABR (1/12)

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

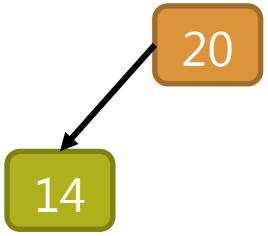
Insertion des éléments dans un ABR (2/12)

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

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

Insertion des éléments dans un ABR (3/12)

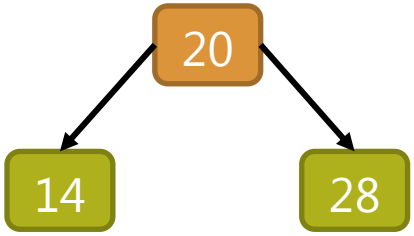
- Eléments restant à insérer : 28, 10, 6, 18, 42, 34, 12, 11, 57



- 14 < 20 → inséré à gauche

Insertion des éléments dans un ABR (4/12)

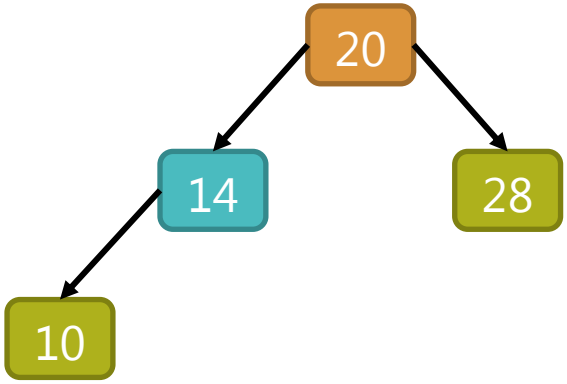
- Eléments restant à insérer : 10, 6, 18, 42, 34, 12, 11, 57



- 28 > 20 → inséré à droite

Insertion des éléments dans un ABR (5/12)

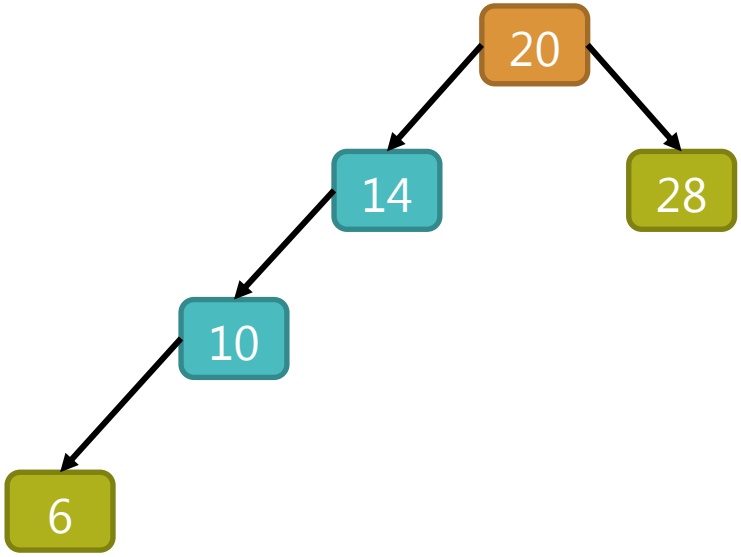
- Eléments restant à insérer : 6, 18, 42, 34, 12, 11, 57



- 10 < 20 → à gauche ; 10 < 14 → inséré à gauche

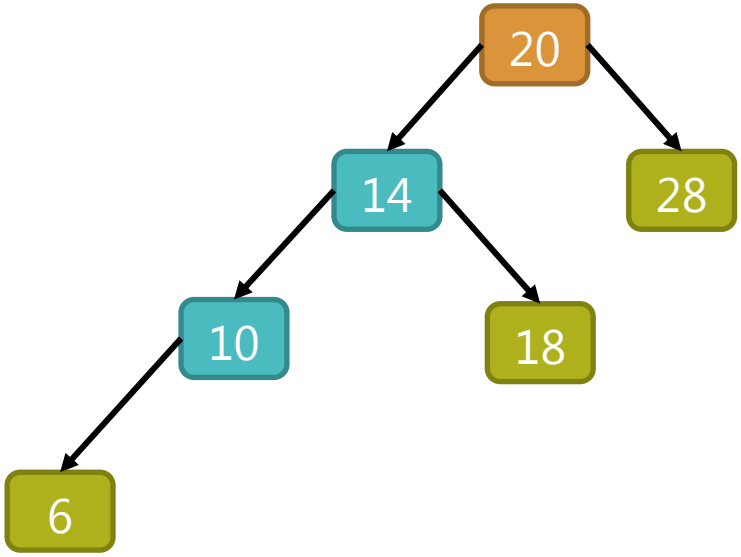
Insertion des éléments dans un ABR (6/12)

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



Insertion des éléments dans un ABR (7/12)

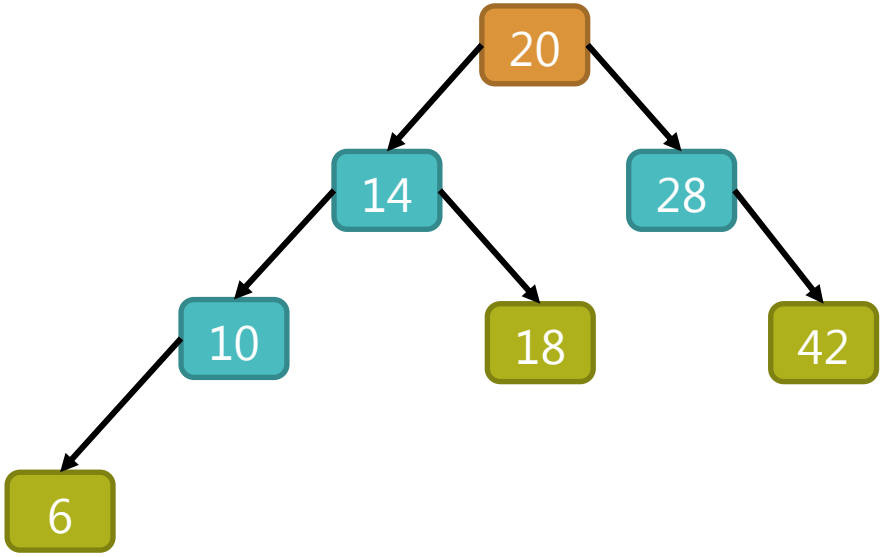
- Eléments restant à insérer : 42, 34, 12, 11, 57



- 18 < 20 → à gauche ; 18 > 14 → inséré à gauche

Insertion des éléments dans un ABR (8/12)

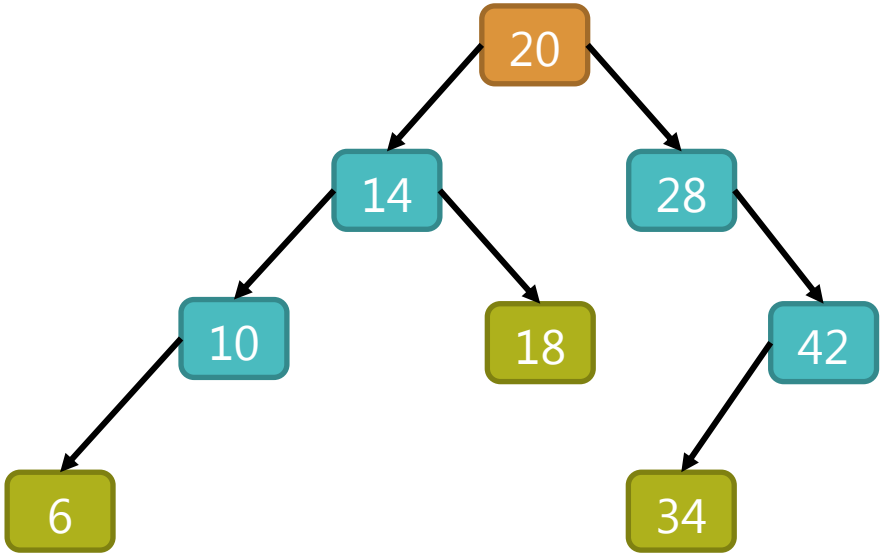
- Eléments restant à insérer : 34, 12, 11, 57



- 42 > 20 → à droite ; 42 > 28 → inséré à droite

Insertion des éléments dans un ABR (9/12)

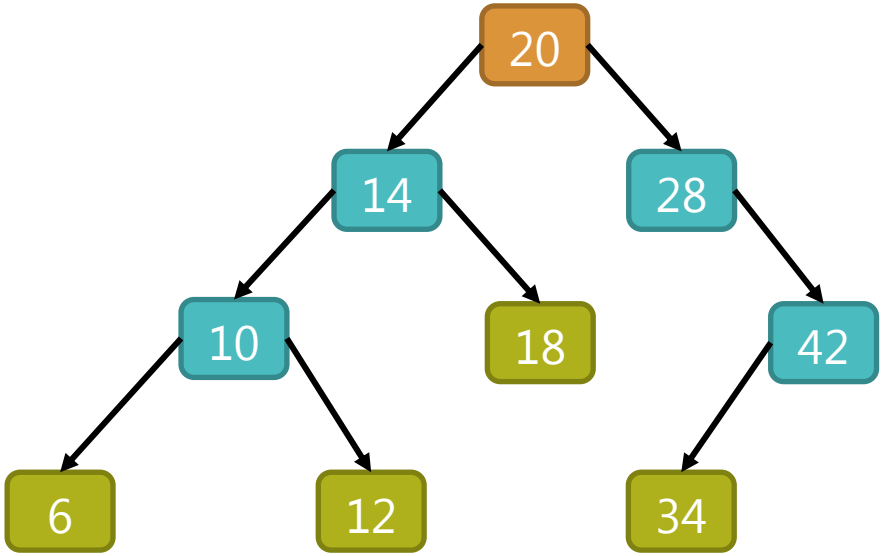
- Eléments restant à insérer : 12, 11, 57



- 34 > 20 → à droite ; 34 > 28 → à droite ; 34 < 42 → à gauche

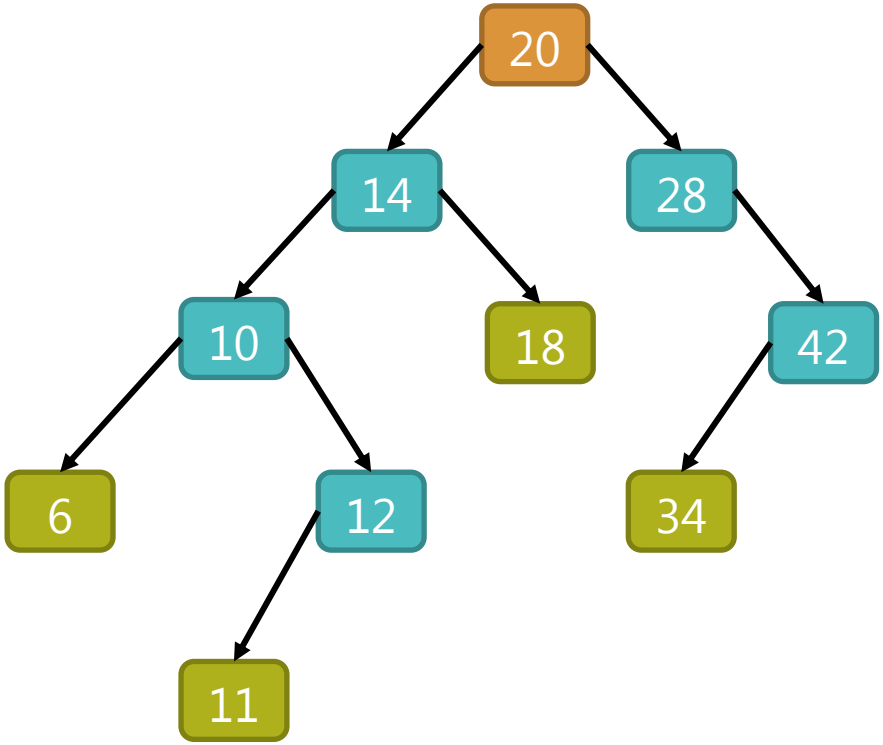
Insertion des éléments dans un ABR (10/12)

- Eléments restant à insérer : 11, 57



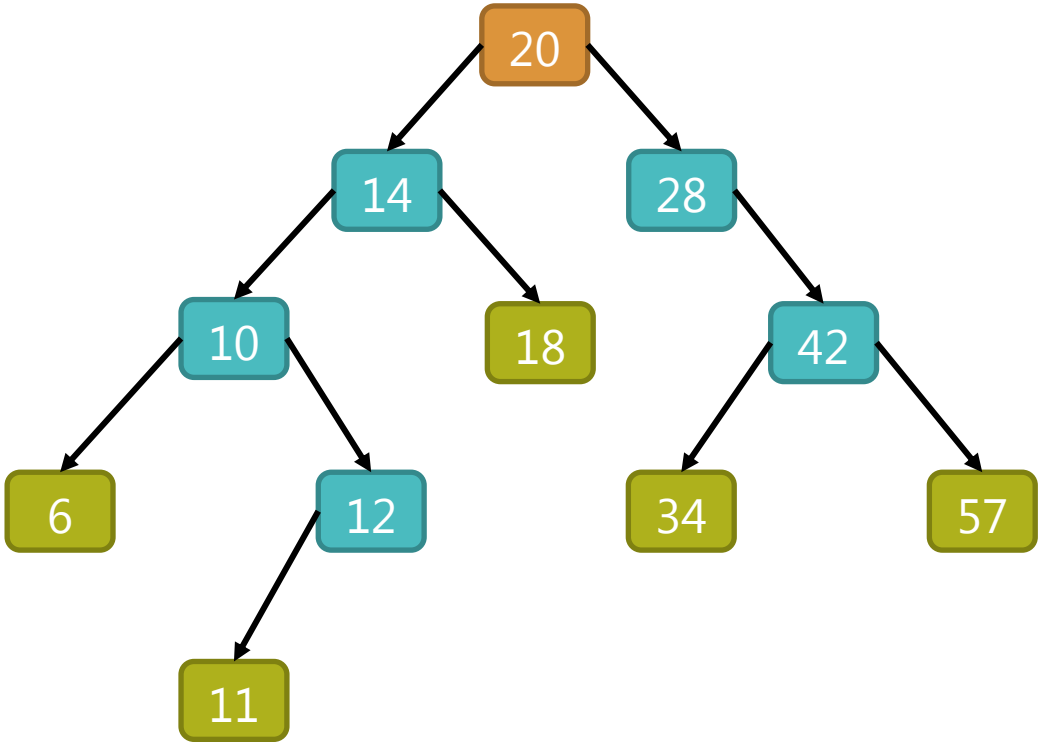
Insertion des éléments dans un ABR (11/12)

- Eléments restant à insérer : 57



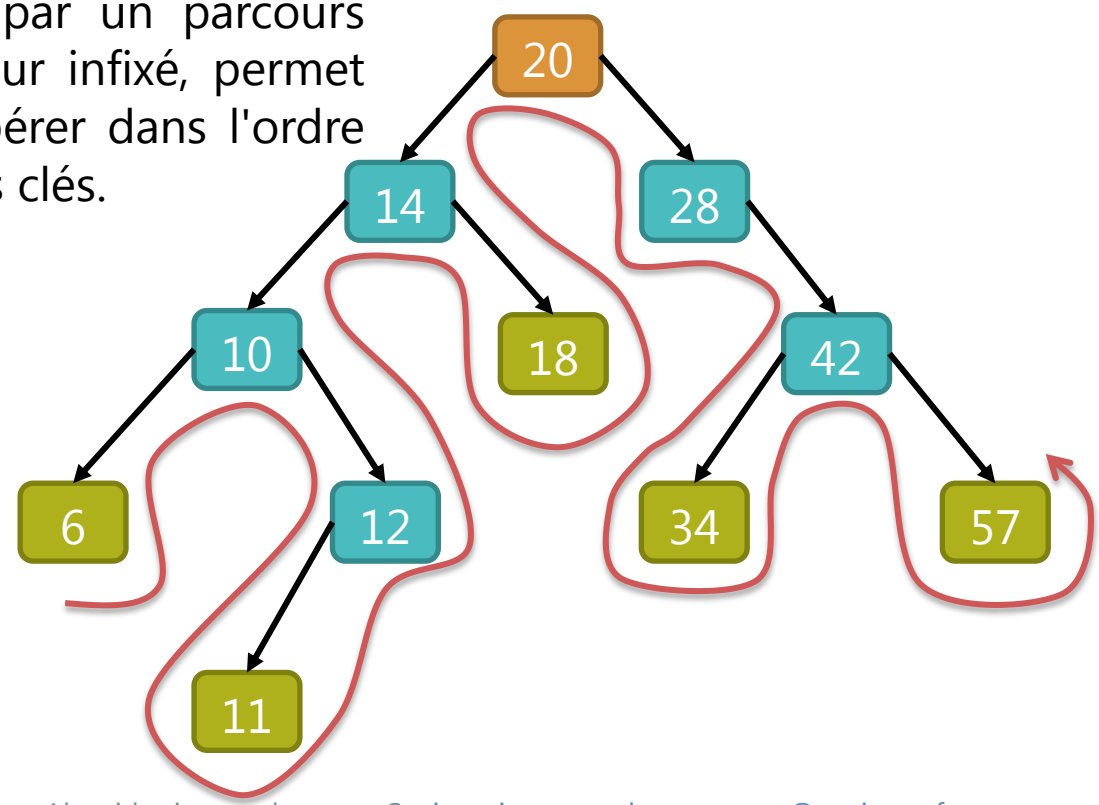
Insertion des éléments dans un ABR (12/12)

- Eléments restant à insérer :



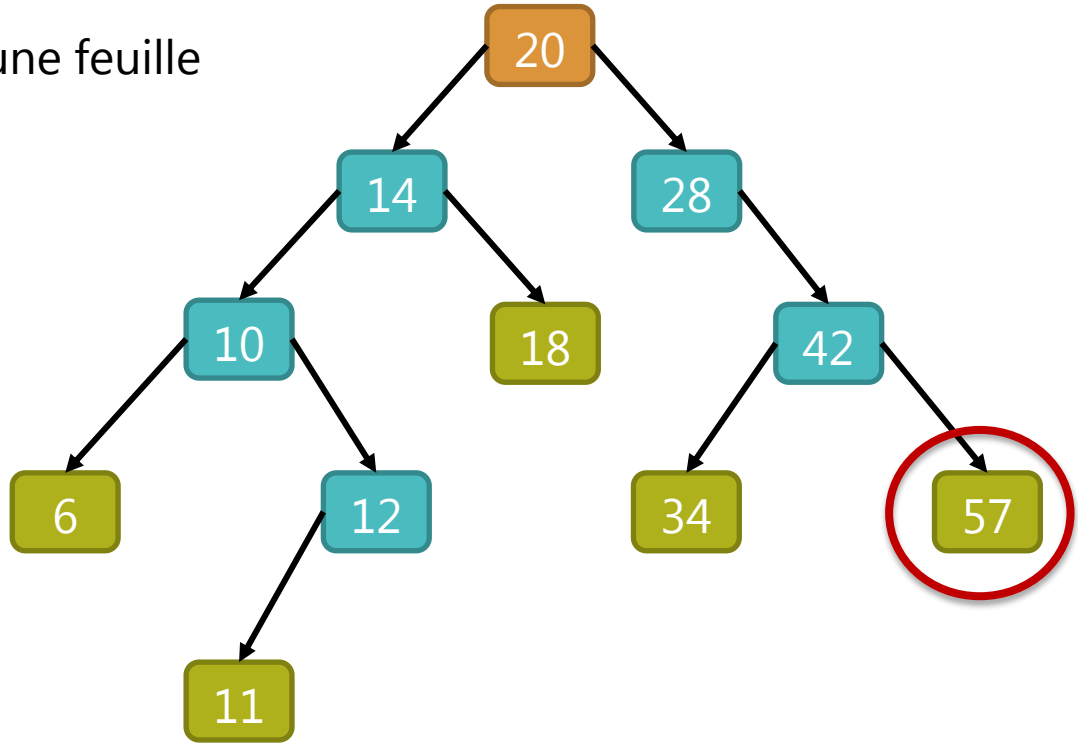
Tri par ABR

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



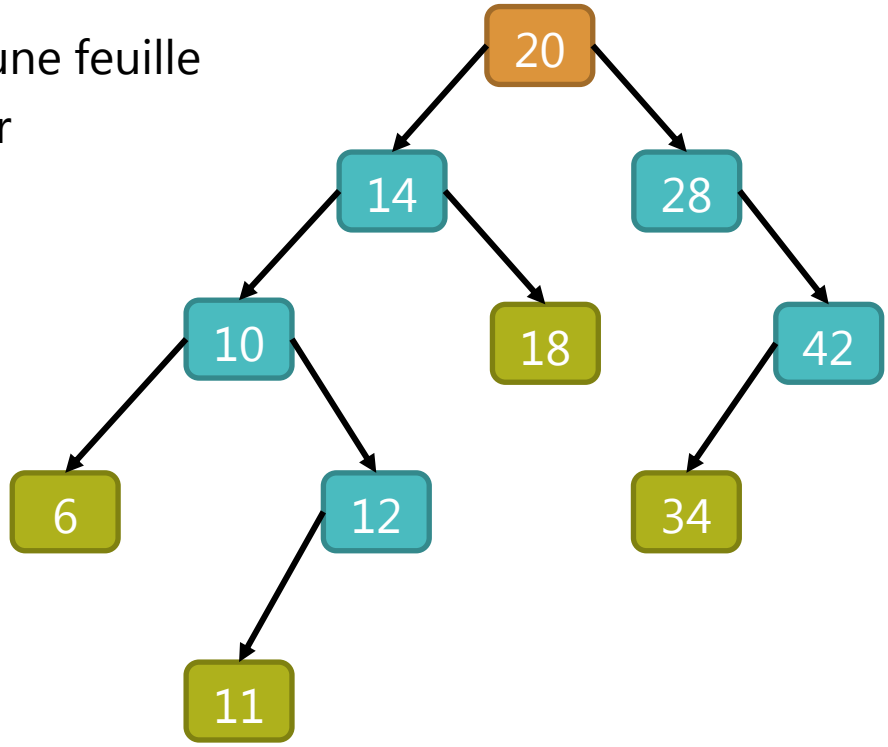
Suppression d'un noeud (1/8)

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



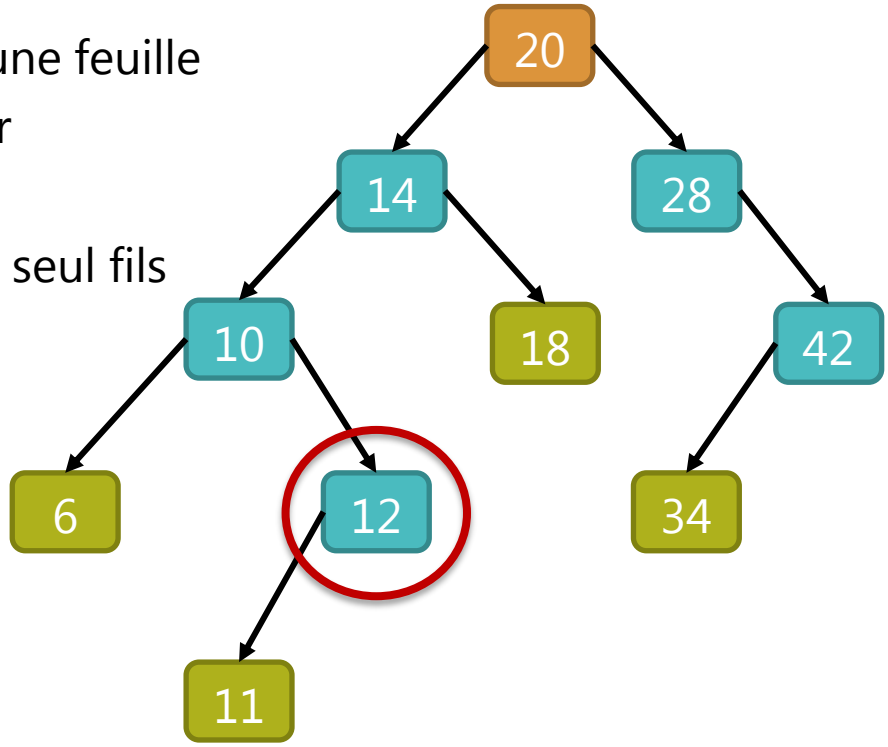
Suppression d'un noeud (2/8)

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



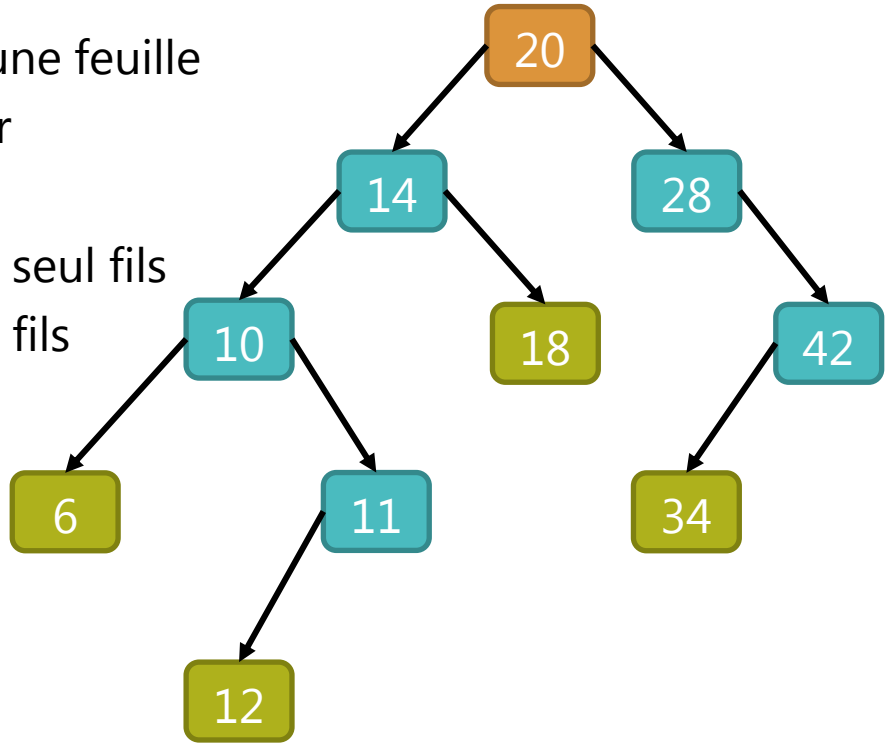
Suppression d'un noeud (3/8)

- 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



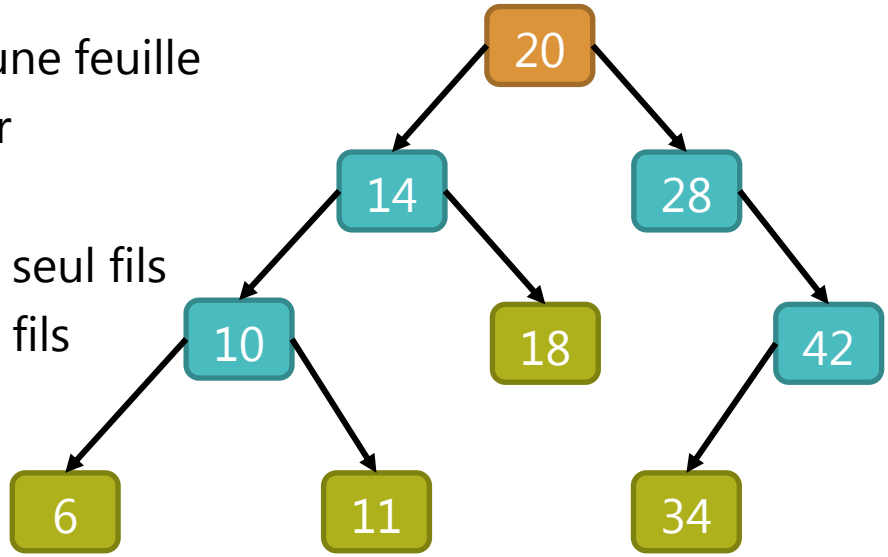
Suppression d'un noeud (4/8)

- 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



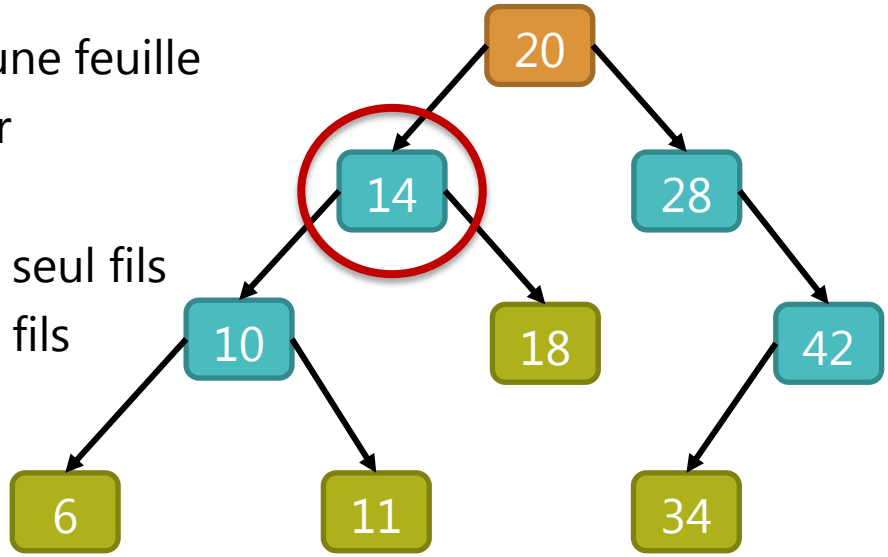
Suppression d'un noeud (5/8)

- 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



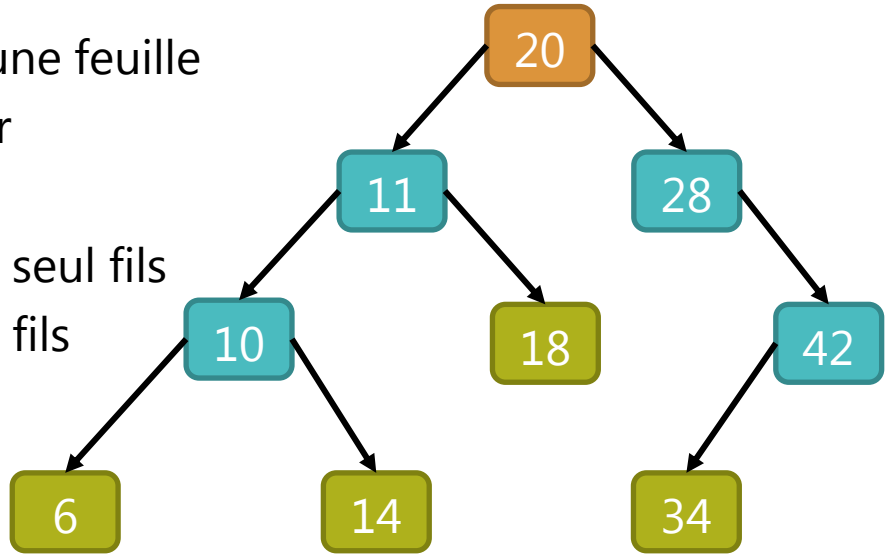
Suppression d'un noeud (6/8)

- 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



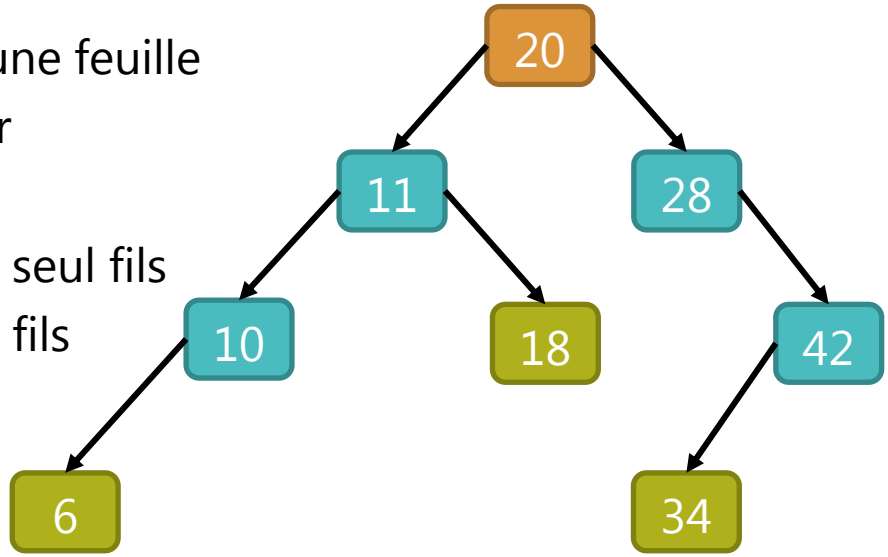
Suppression d'un noeud (7/8)

- 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



Suppression d'un noeud (8/8)

- 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
     - ➢ Puis on supprime le nœud 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 x TElement → TNode
SearchKey : TSearchTree x TElement → TNode
RemoveNode : TSearchTree x TElement → bool
Traversal : TSearchTree →
Clear : TSearchTree →
Delete : TSearchTree →
```

ARBRES BINAIRES

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

```c
/* 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;
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Ajout d'une clé dans l'ABR

```
PTNode TSearchTree_AddNode(const PTSearchTree 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;
}
```

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;
}
```

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.

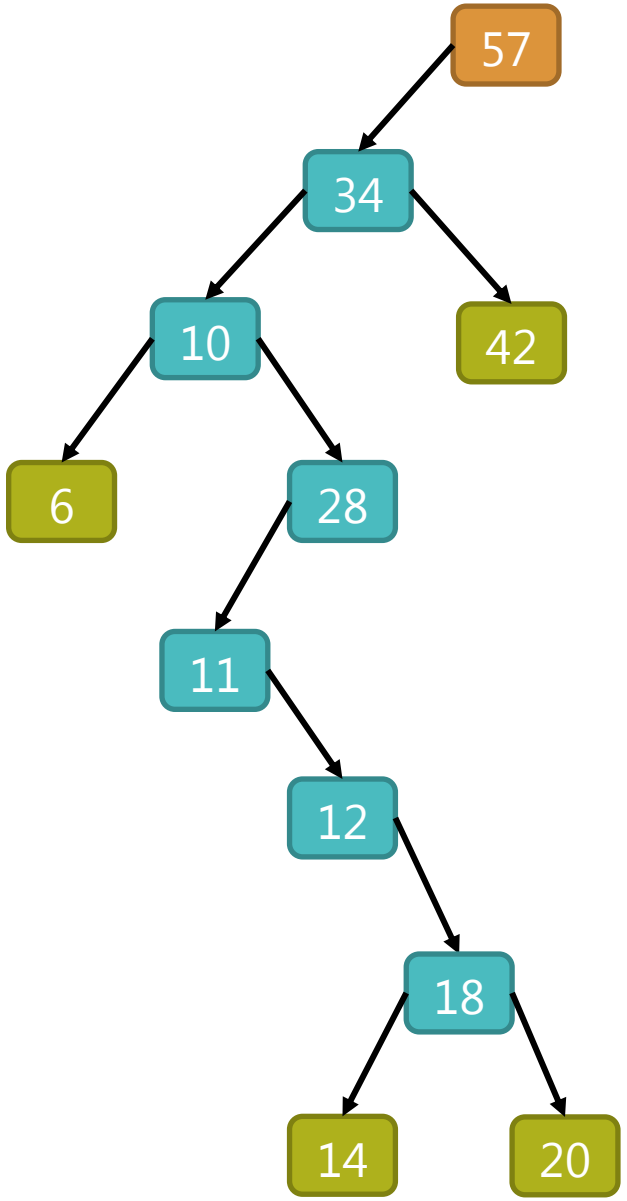
- L'action à réaliser est définie par la fonction de callback *action* passée en second paramètre.

```
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);
  }
}
```

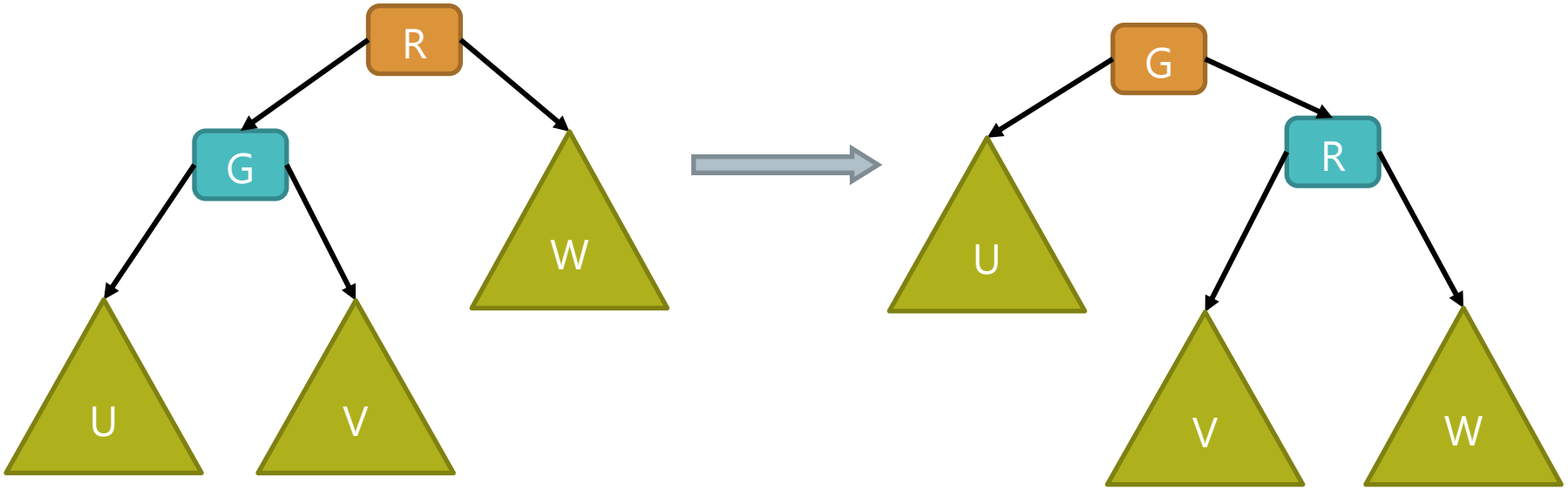
ARBRES BINAIRES

Complexité de la recherche dans un ABR

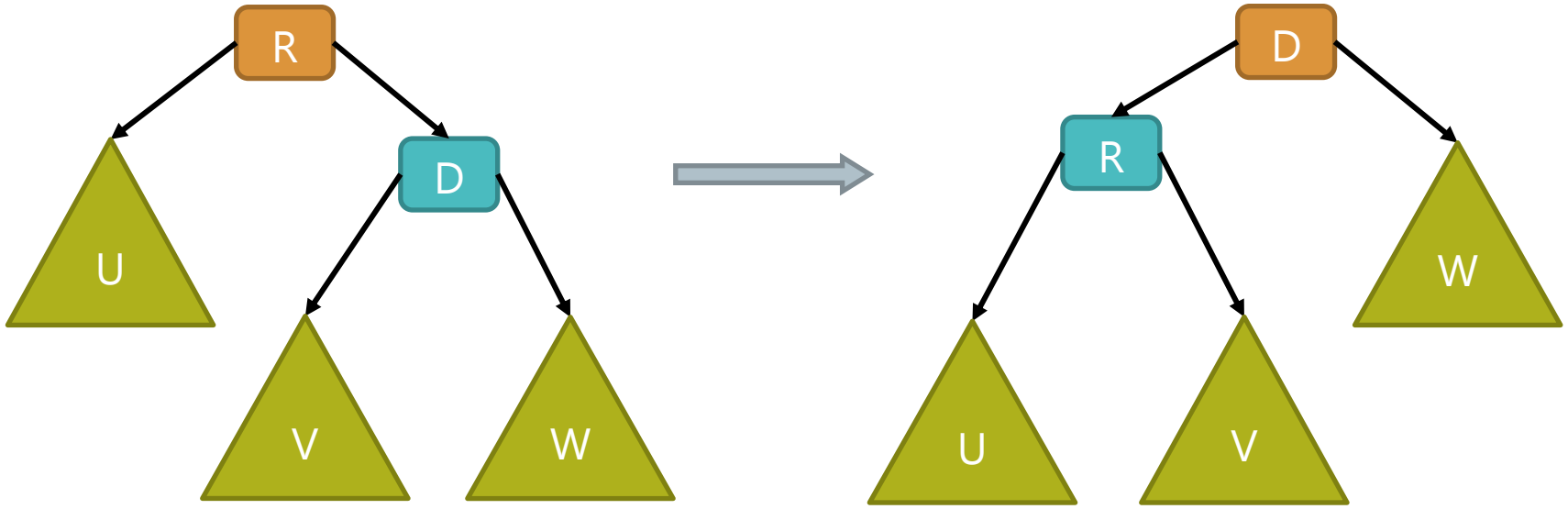
- La recherche d'un élément (une clé) dans un ABR est similaire à la recherche par dichotomie dans une collection triée, et la complexité est en $O(\log(n))$.
- Si l'arbre est mal équilibré, la complexité peut tendre vers $O(n)$, comme dans une liste chainée.
- Il peut alors être nécessaire de rééquilibrer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



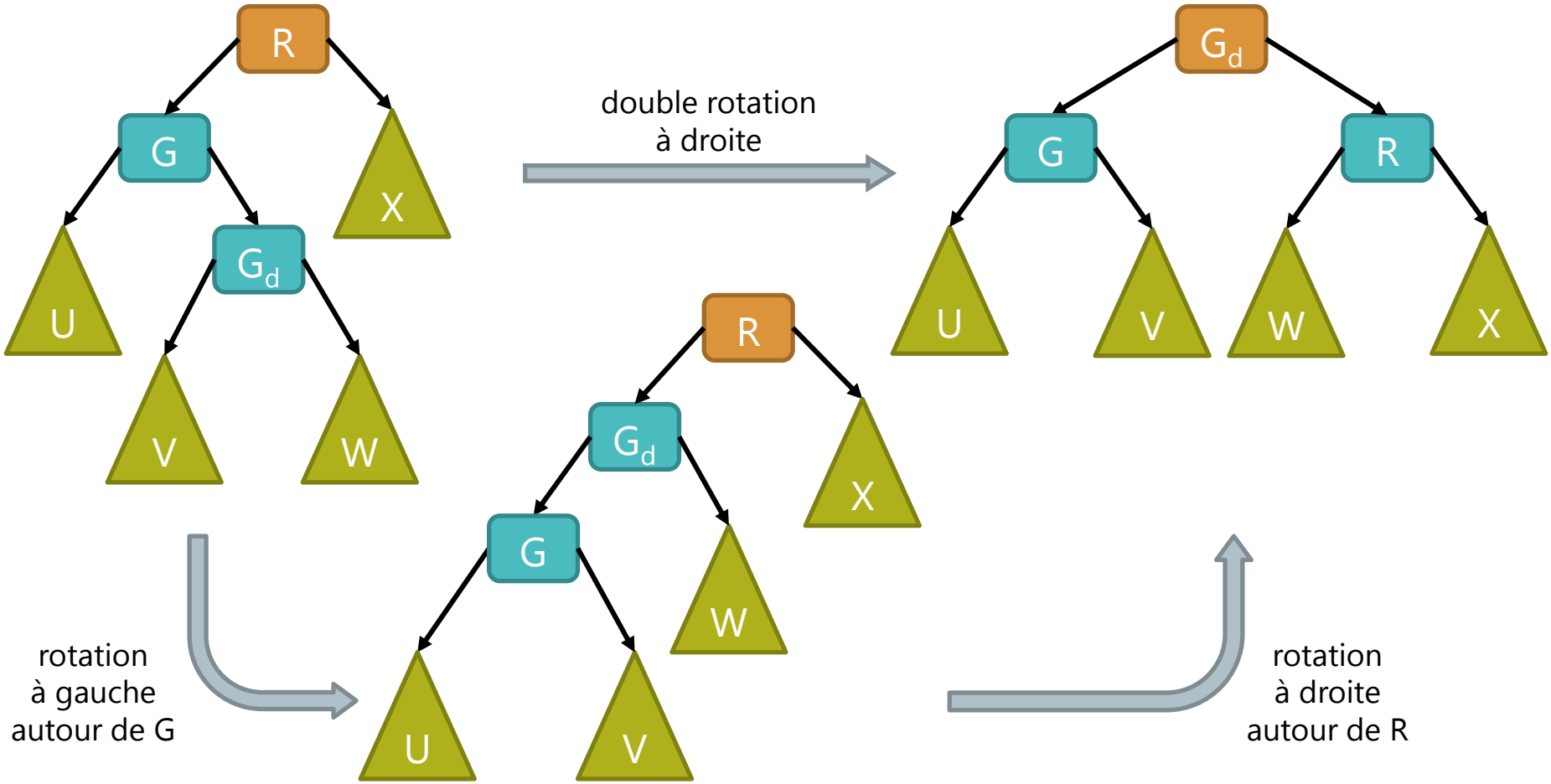
Simple rotation à droite



Simple rotation à gauche



Double rotation à droite



Arbres AVL

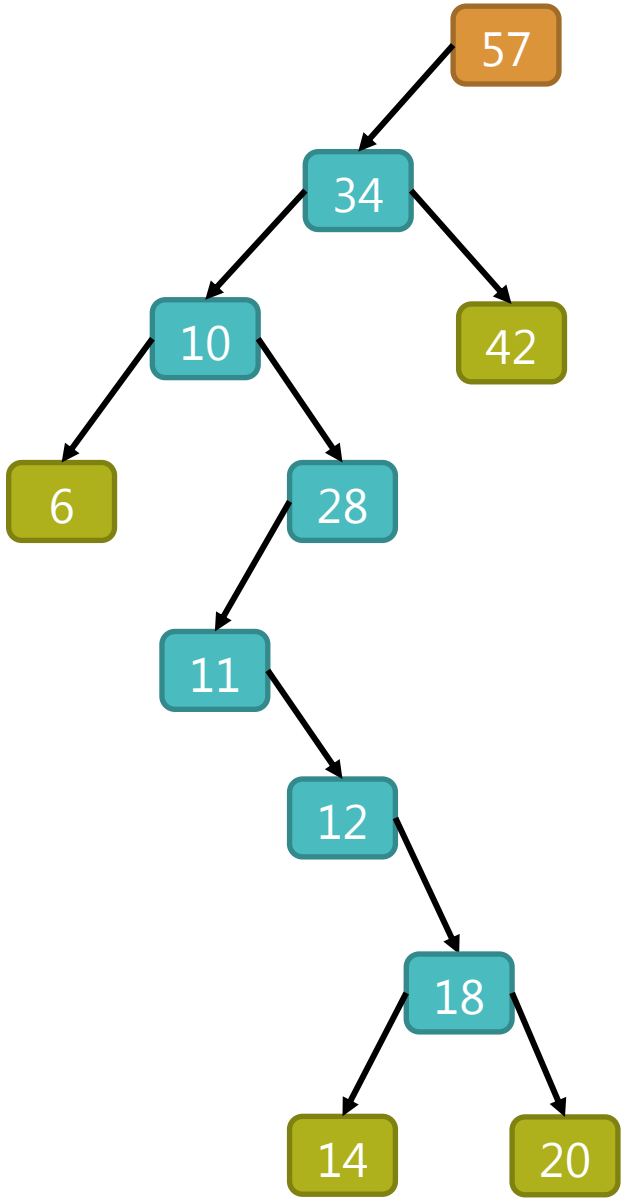
- Les arbres AVL sont des ABR automatiquement équilibrés.
- Inventés en 1962 par (*Georgii **A**delson-**V**elsky* et *Evguenii **L**andis*).
- Facteur d'équilibrage d'un nœud : différence entre la hauteur de son sous-arbre gauche et la hauteur de son sous-arbre droit.
- Un nœud équilibré a un facteur 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.



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

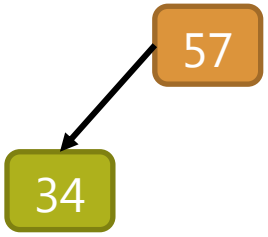
Insertion des éléments dans un AVL (2/20)

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

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

Insertion des éléments dans un AVL (3/20)

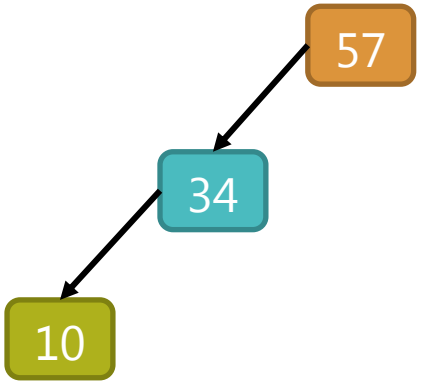
- Eléments restant à insérer : 10, 42, 28, 6, 11, 12, 18, 14, 20



- 34 < 57 → inséré à gauche

Insertion des éléments dans un AVL (4/20)

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

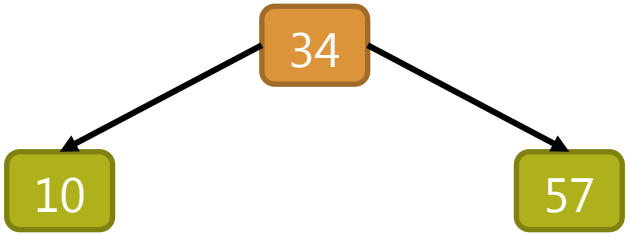


- 10 < 57 → à gauche
- 10 < 34 → inséré à gauche

→ déséquilibre au niveau racine

Insertion des éléments dans un AVL (5/20)

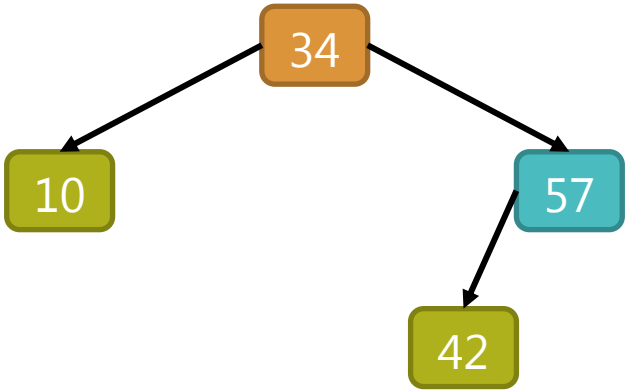
- Eléments restant à insérer : 42, 28, 6, 11, 12, 18, 14, 20



- arbre rééquilibré après rotation à droite

Insertion des éléments dans un AVL (6/20)

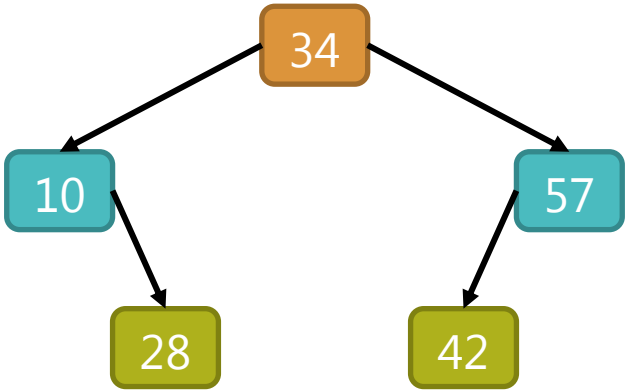
- Eléments restant à insérer : 28, 6, 11, 12, 18, 14, 20



- 42 > 34 → à droite
- 42 < 57 → inséré à gauche

Insertion des éléments dans un AVL (7/20)

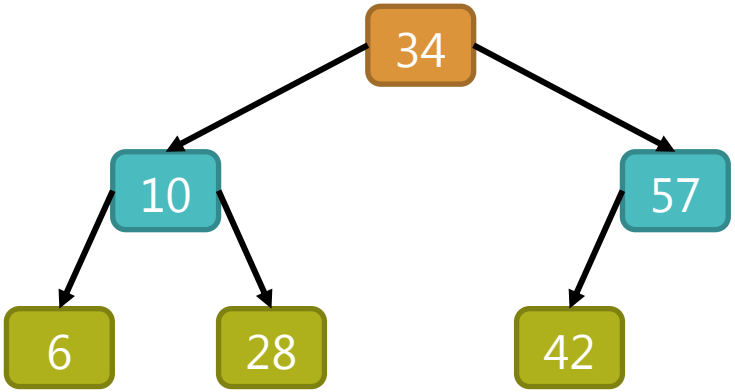
- Eléments restant à insérer : 6, 11, 12, 18, 14, 20



- 28 < 34 → à gauche
- 28 < 10 → inséré à droite

Insertion des éléments dans un AVL (8/20)

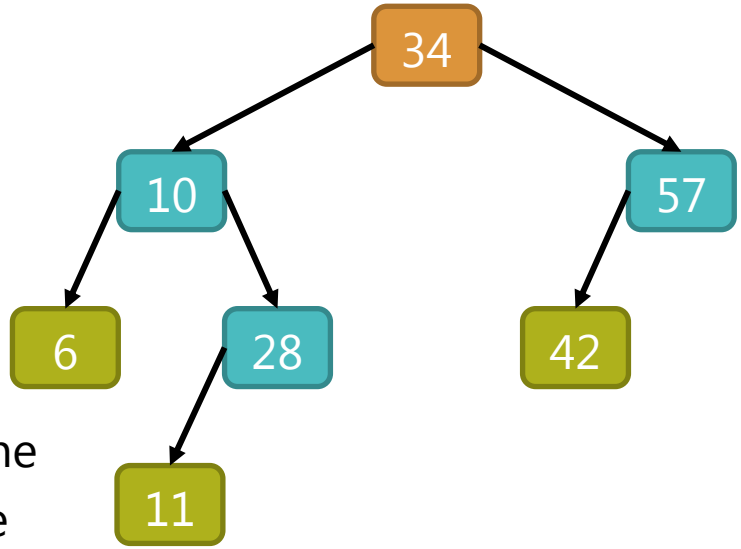
- Eléments restant à insérer : 11, 12, 18, 14, 20



- 6 < 34 → à gauche
- 6 < 10 → inséré à gauche

Insertion des éléments dans un AVL (9/20)

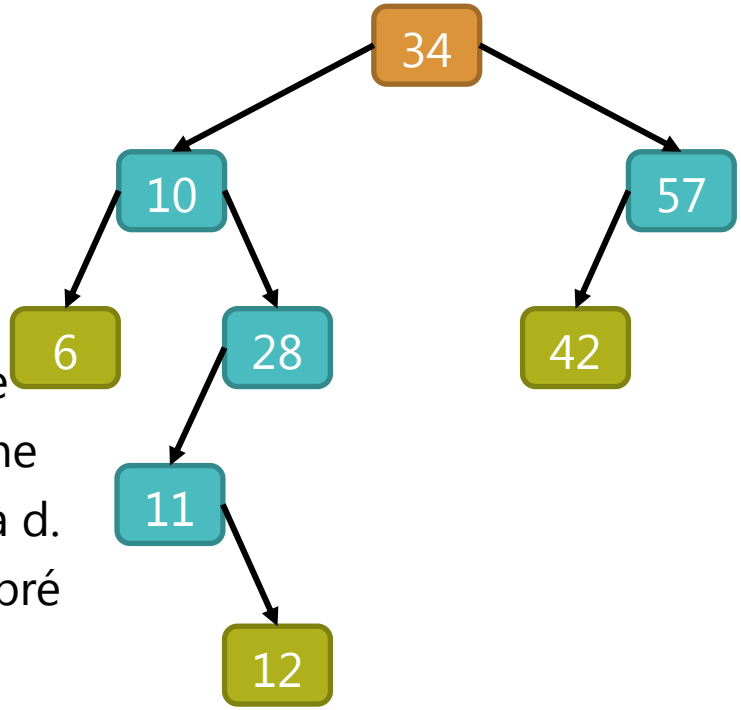
- Eléments restant à insérer : 12, 18, 14, 20



- 11 < 34 → à gauche
- 11 > 10 → à droite
- 11 < 28 → inséré à gauche

Insertion des éléments dans un AVL (10/20)

- Eléments restant à insérer : 18, 14, 20

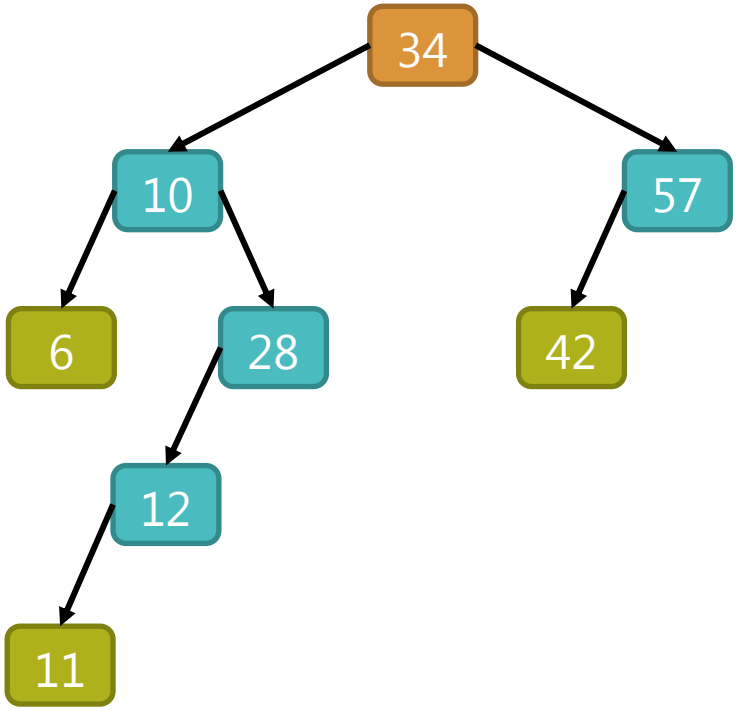


- 12 < 34 → à g.
- 12 > 10 → à droite
- 12 < 28 → à gauche
- 12 > 11 → inséré à d.

→ nœud 28 déséquilibré

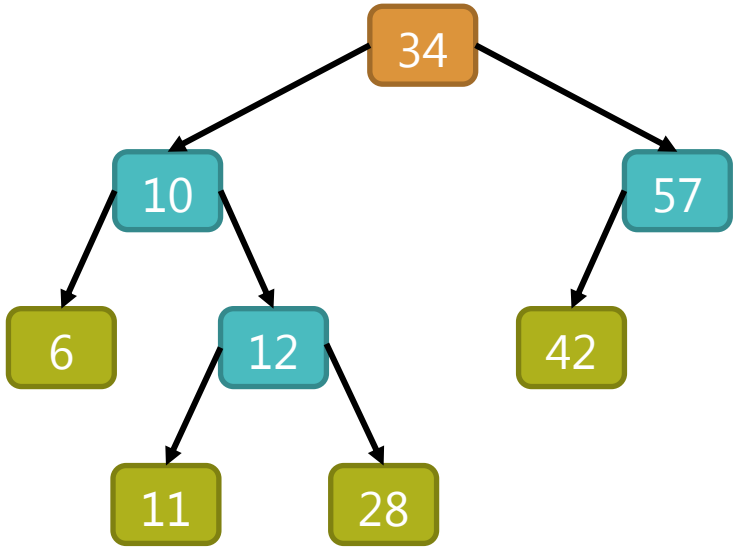
Insertion des éléments dans un AVL (11/20)

- Rotation à droite pour rééquilibrer le sous-arbre



Insertion des éléments dans un AVL (12/20)

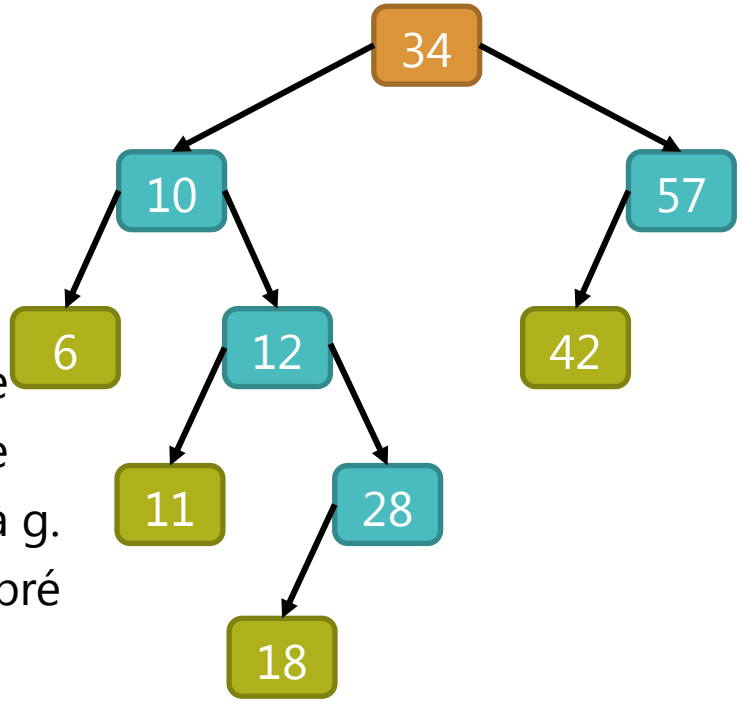
- Eléments restant à insérer : 18, 14, 20



- sous-arbre rééquilibré après rotation à droite

Insertion des éléments dans un AVL (13/20)

- Eléments restant à insérer : 14, 20

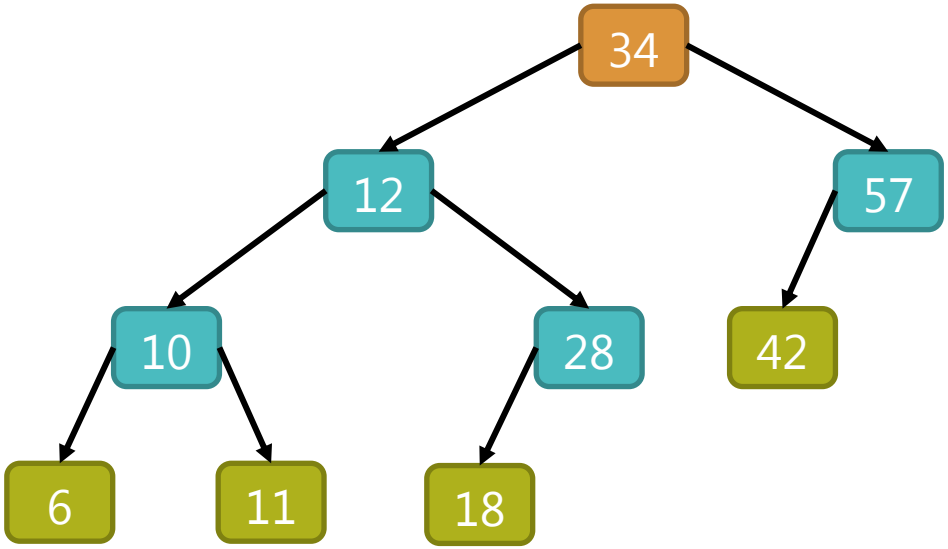


- 18 < 34 → à g.
- 18 > 10 → à droite
- 18 < 12 → à droite
- 18 < 28 → inséré à g.

→ nœud 10 déséquilibré

Insertion des éléments dans un AVL (14/20)

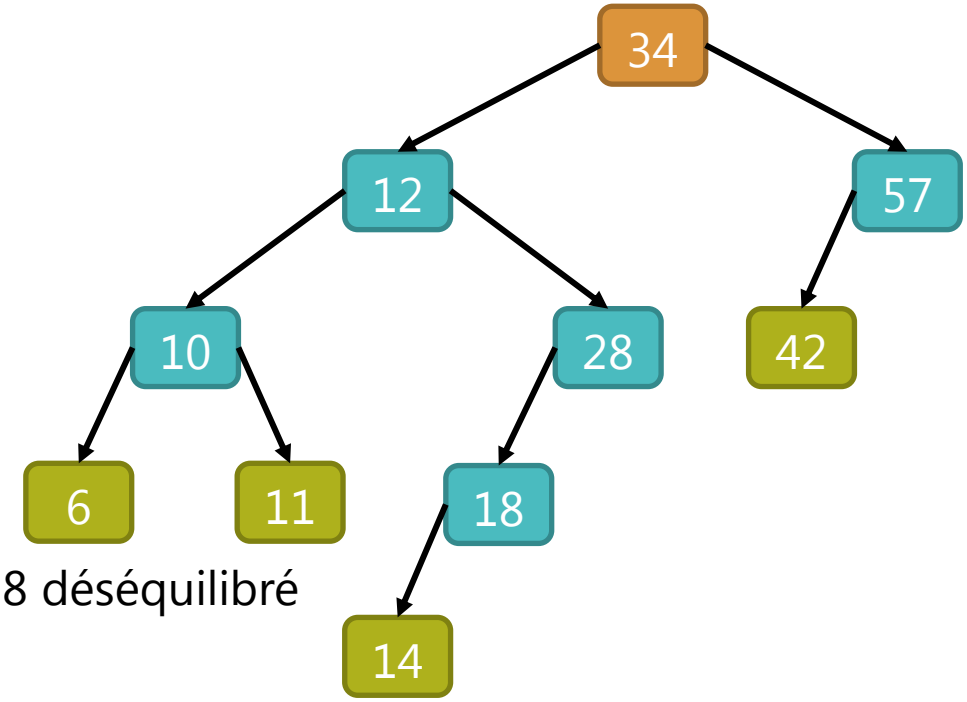
- Eléments restant à insérer : 14, 20



- sous-arbre rééquilibré après rotation à gauche

Insertion des éléments dans un AVL (15/20)

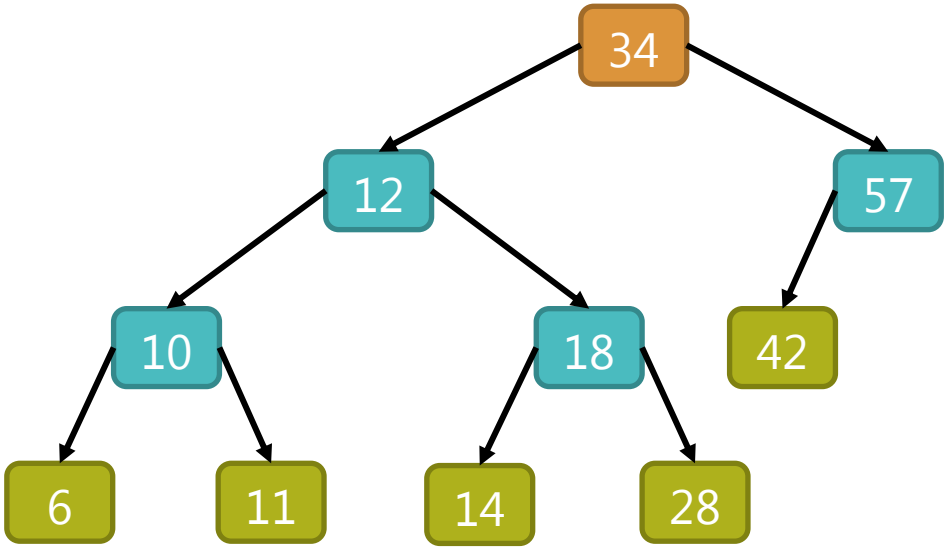
- Eléments restant à insérer : 20



- nœud 28 déséquilibré

Insertion des éléments dans un AVL (16/20)

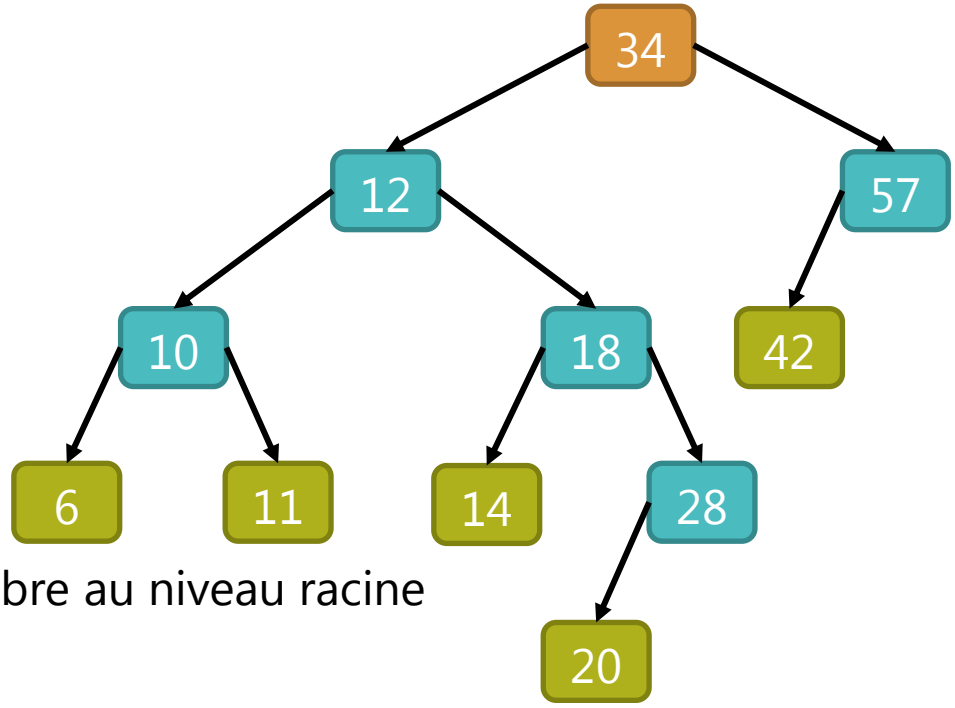
- Eléments restant à insérer : 20



- sous-arbre rééquilibré après rotation à gauche

Insertion des éléments dans un AVL (17/20)

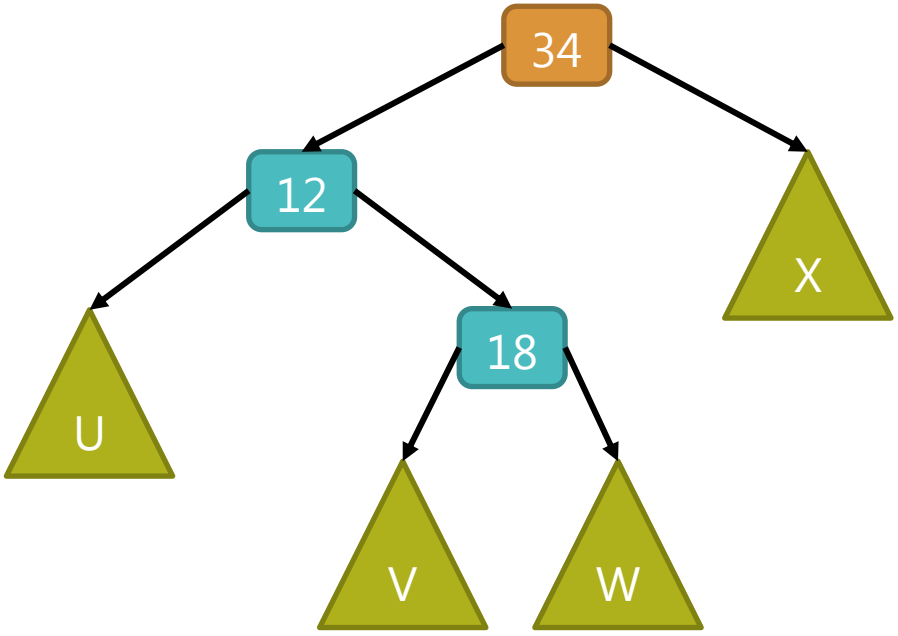
- après insertion du dernier élément (20)



→ déséquilibre au niveau racine

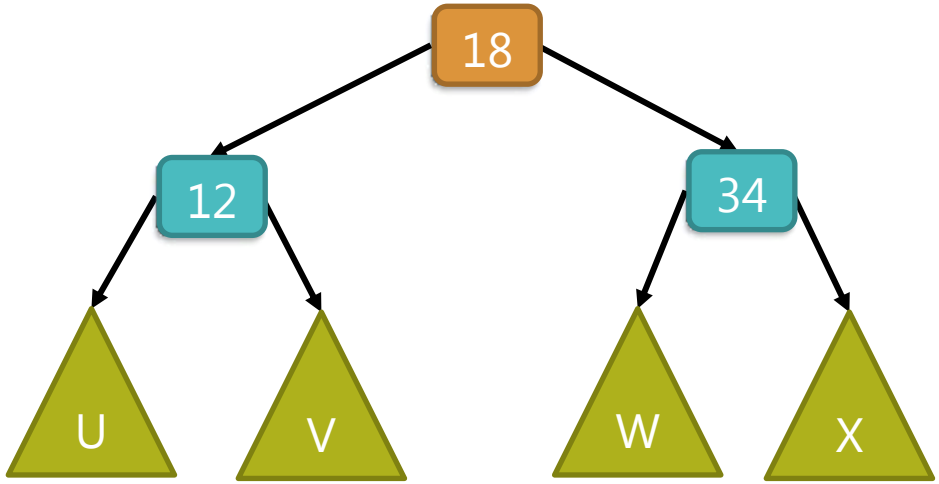
Insertion des éléments dans un AVL (18/20)

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



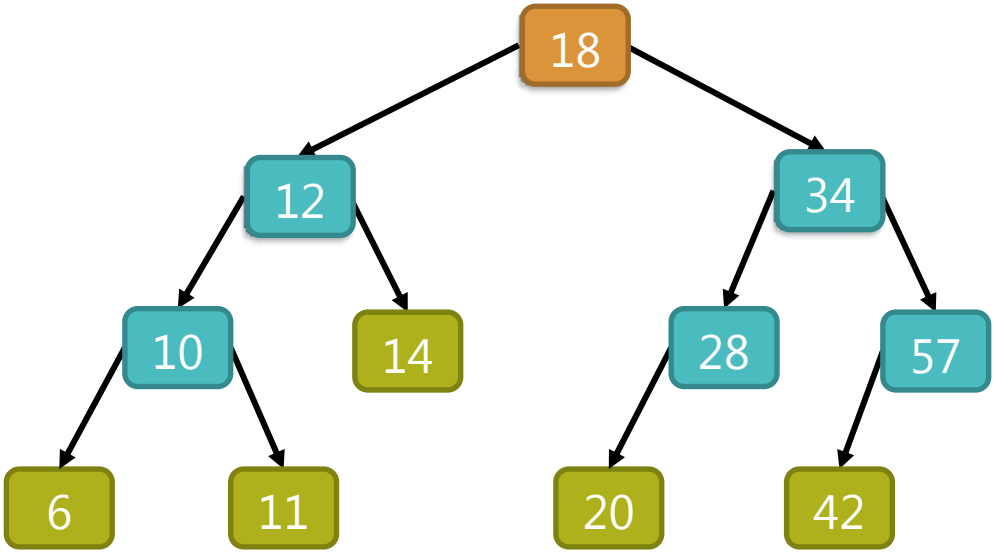
Insertion des éléments dans un AVL (19/20)

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



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.



Implémentation d'un arbre AVL

- Comment savoir s'il faut ré-équilibrer l'arbre quand on insère un nouveau nœud ?

- Il faut connaitre 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.

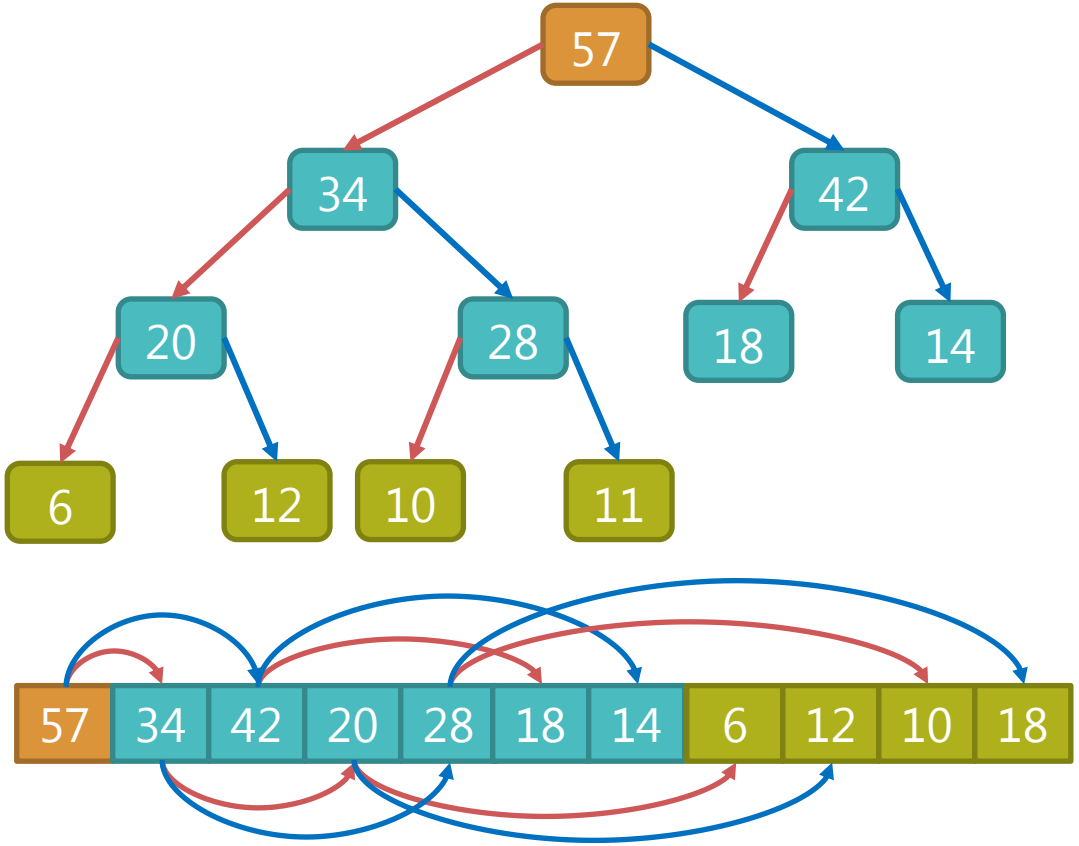
```
typedef struct Node {
  void *pKey;      /* Pointeur vers la clé */
  int balance;     /* Facteur d'équilibrage */
  struct Node *leftChild;
  struct Node *rightChild;
} TNode;

typedef TNode *PTNode;

typedef struct AVLTree {
  int Size;        /* Nombre de noeuds */
  int SizeOfKey;   /* Taille unitaire de clé */
  PTNode Root;     /* Noeud racine */
  PTNode Current;  /* Noeud courant */
} TAVLTree;

typedef TAVLTree *PTAVLTree;
```

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 >= (respectivement <=) aux clés de chacun de ses fils.
   - Si la relation d'ordre est ">=" on a un *tas-max* (*max-heap*)
   - Si la relation d'ordre est "<=" on a un *tas-min* (*min-heap*)

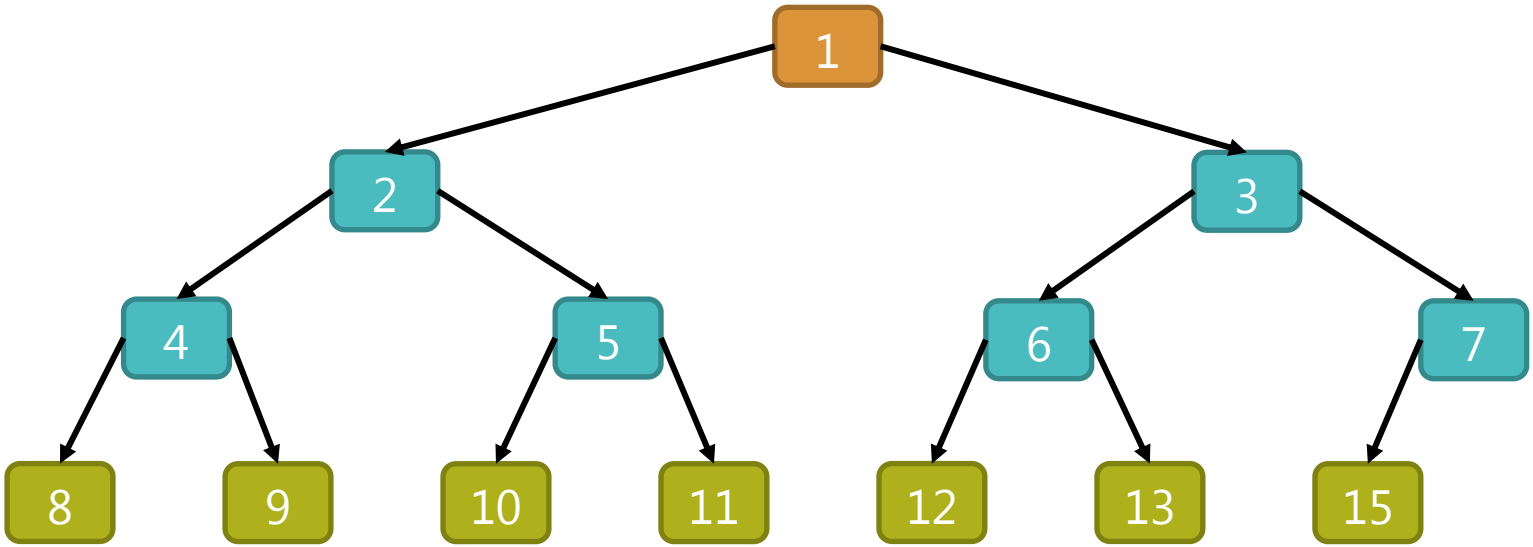
- Un tas étant un arbre binaire complet, il peut être représenté sous forme de tableau.

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



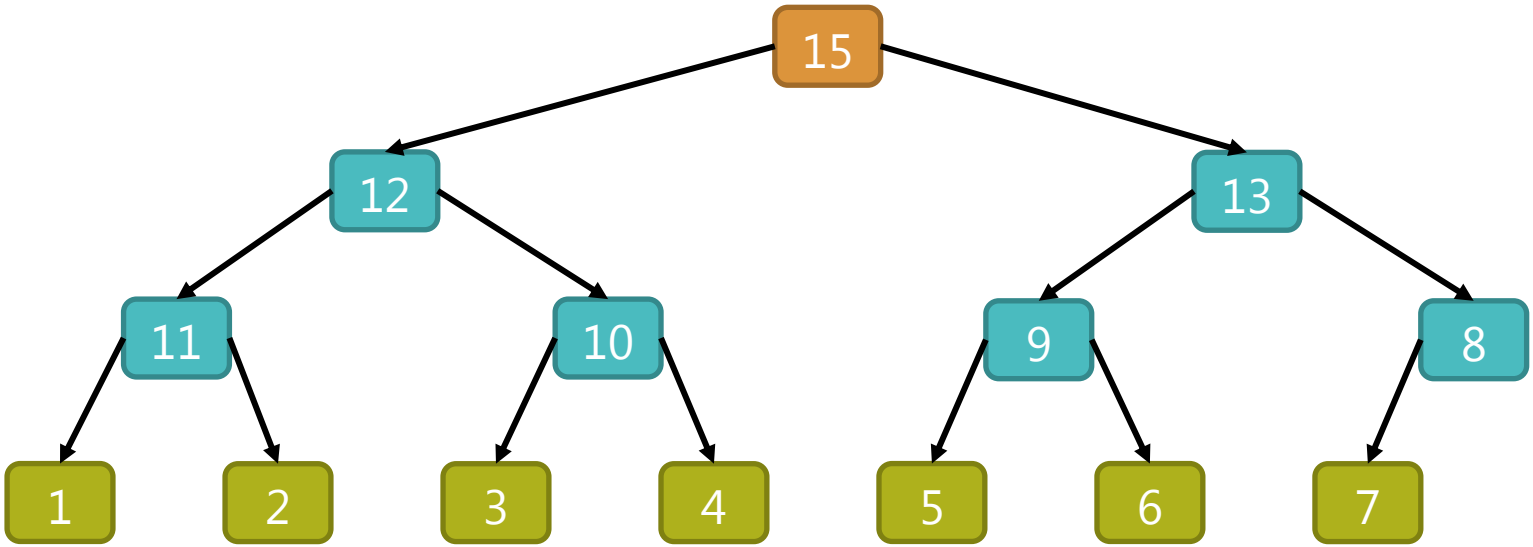
ARBRES BINAIRES

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



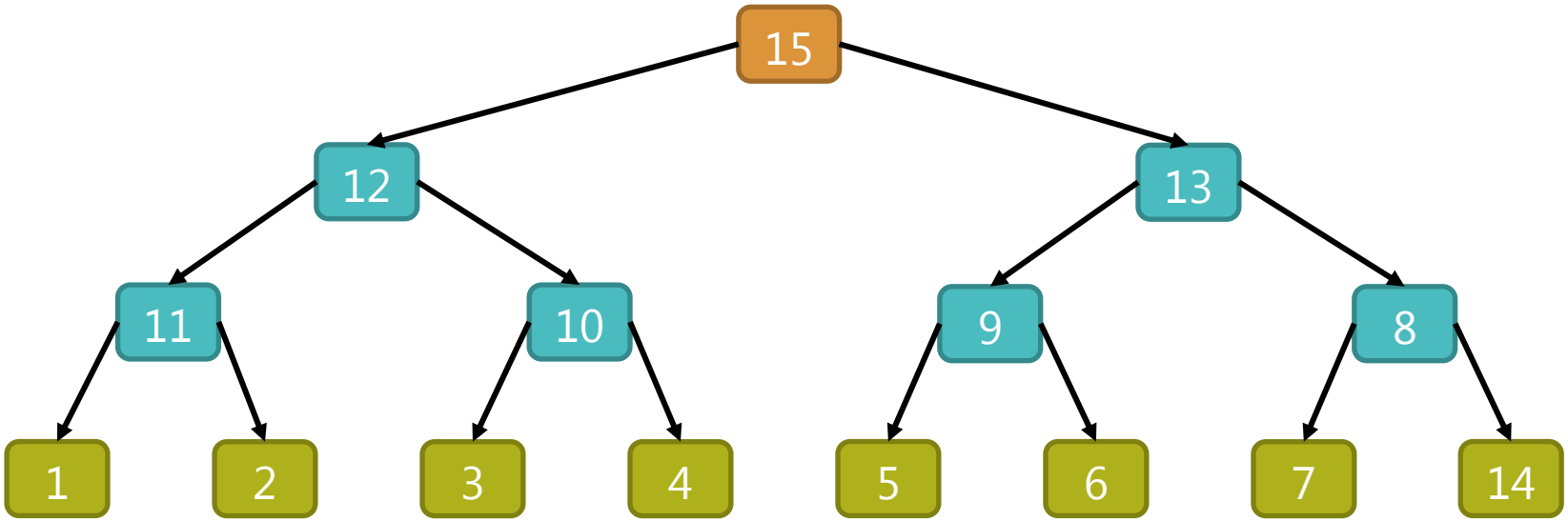
ARBRES BINAIRES

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



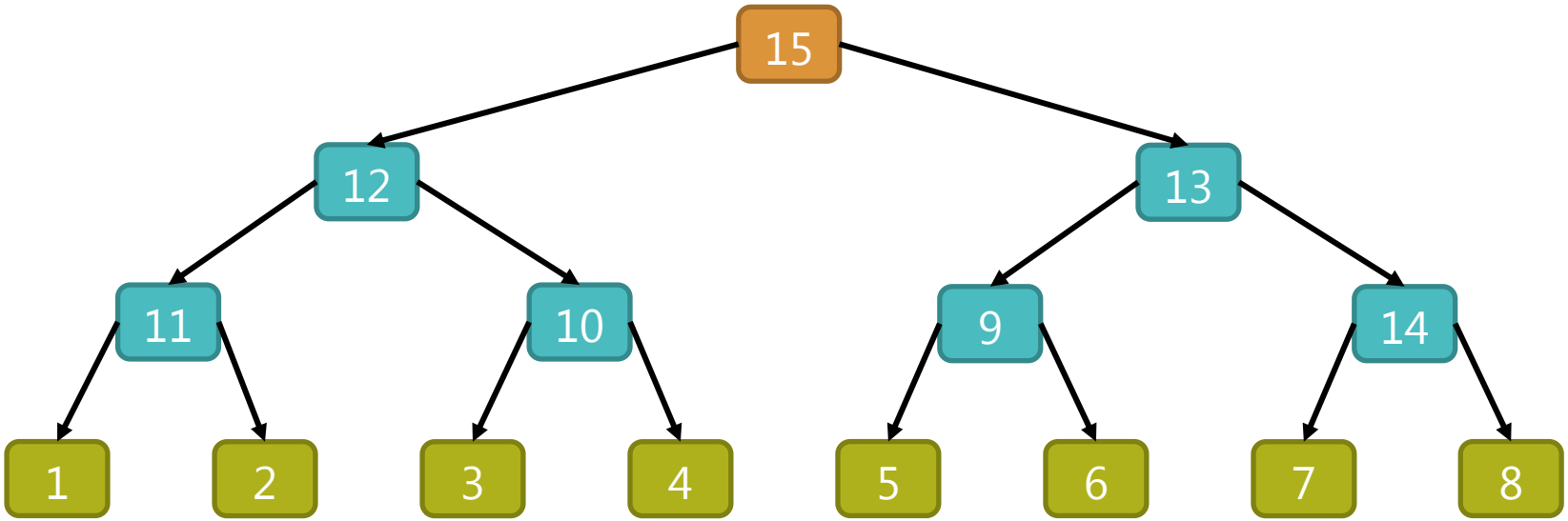
ARBRES BINAIRES

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



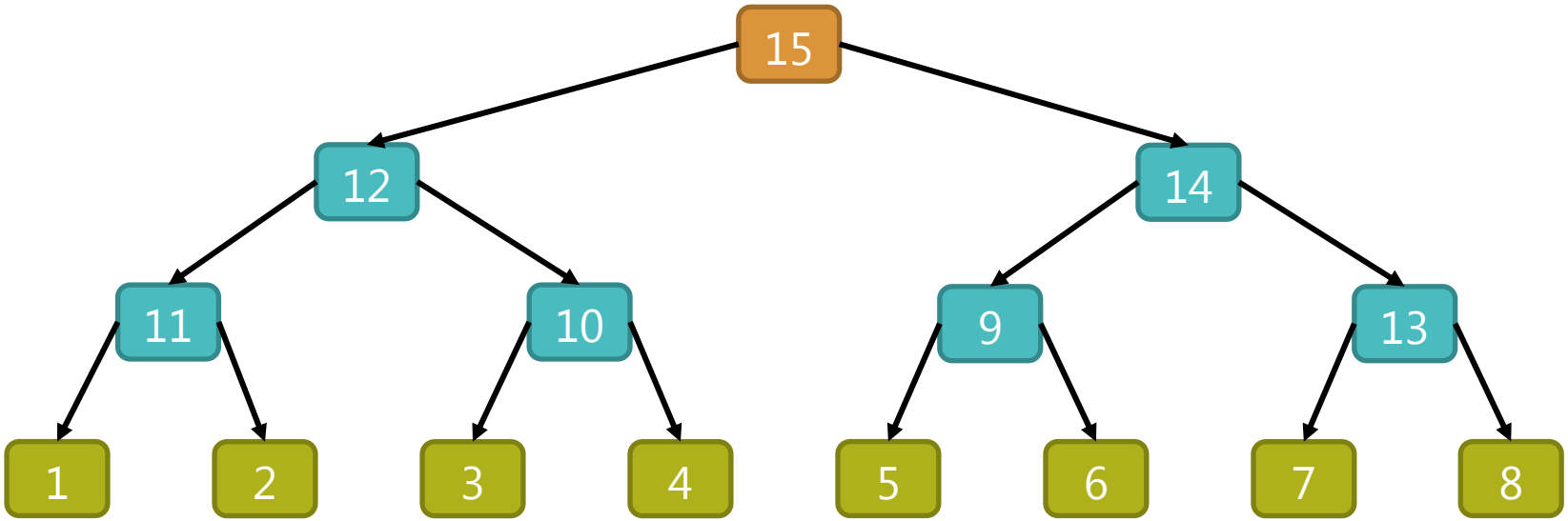
ARBRES BINAIRES

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



Tas et file de priorité

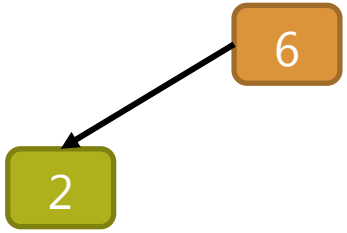
- Le 1er élément du tas (la racine) est toujours le plus grand.
- Cette propriété du tas le rend idéal pour implémenter une *file de priorité*.
- Lorsqu'on retire le 1er élément, il faut réorganiser le tas : si l'élément suivant est inférieur à au moins un de ses fils, on l'échange avec son fils le plus 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*).

Tri pas tas

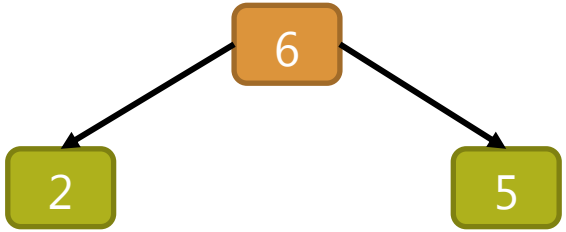
- On commence par insérer les éléments du tableau dans une structure de tas

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

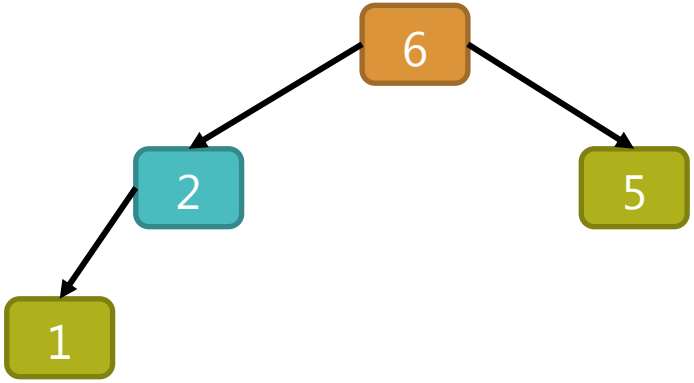
Tri pas tas : insertion des éléments dans le tas (2/12)



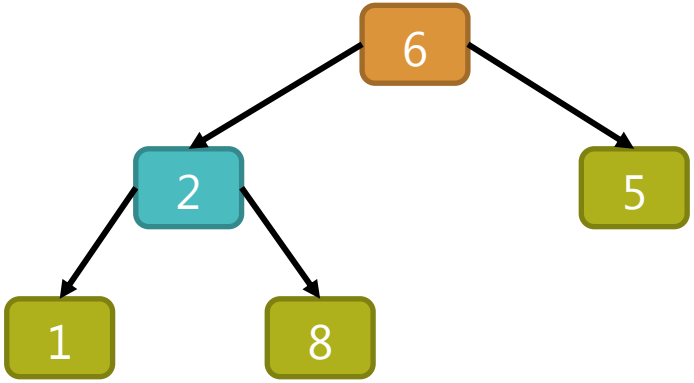
Tri pas tas : insertion des éléments dans le tas (3/12)



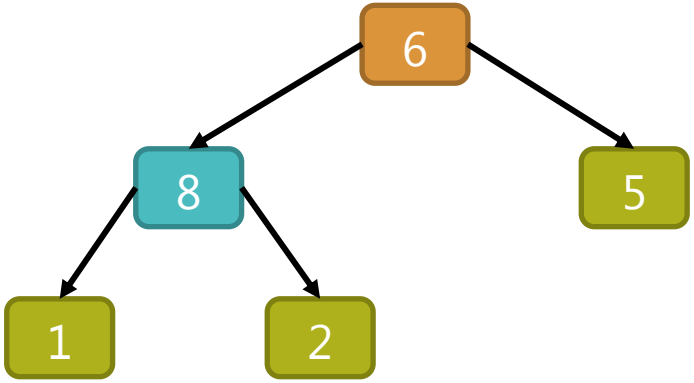
Tri pas tas : insertion des éléments dans le tas (4/12)



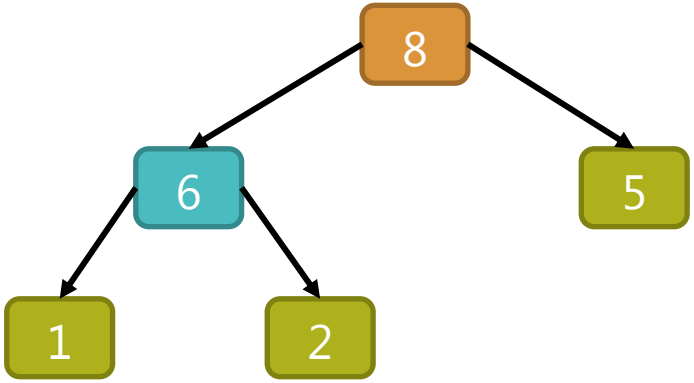
Tri pas tas : insertion des éléments dans le tas (5/12)



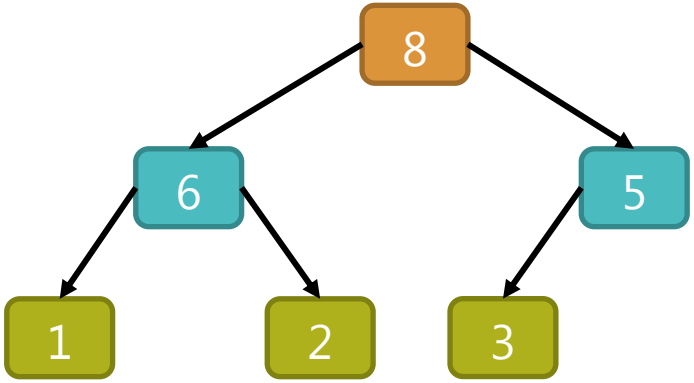
Tri pas tas : insertion des éléments dans le tas (6/12)



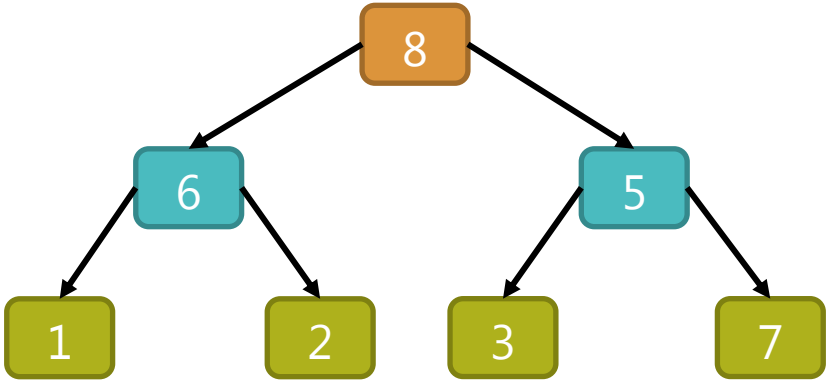
Tri pas tas : insertion des éléments dans le tas (7/12)



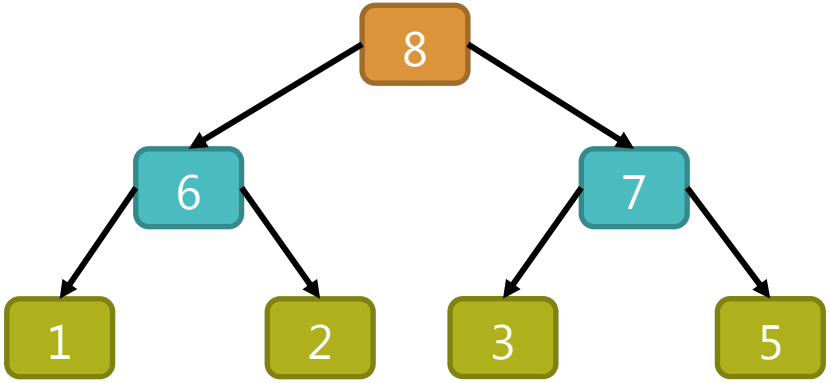
Tri pas tas : insertion des éléments dans le tas (8/12)



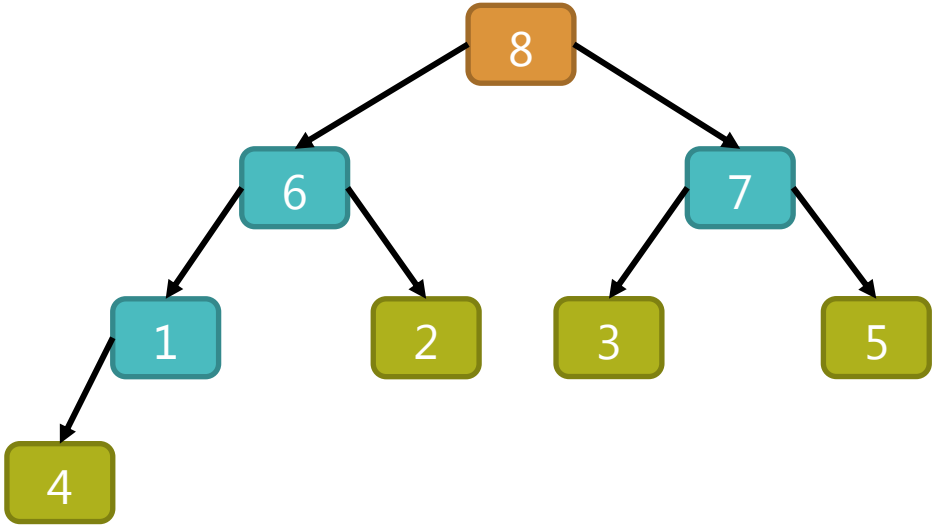
Tri pas tas : insertion des éléments dans le tas (9/12)



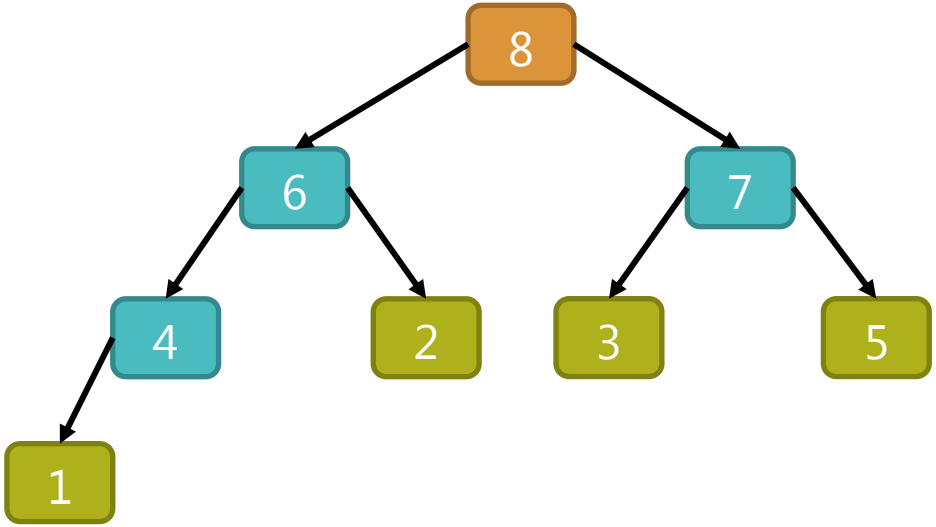
Tri pas tas : insertion des éléments dans le tas (10/12)



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



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



Tamisage

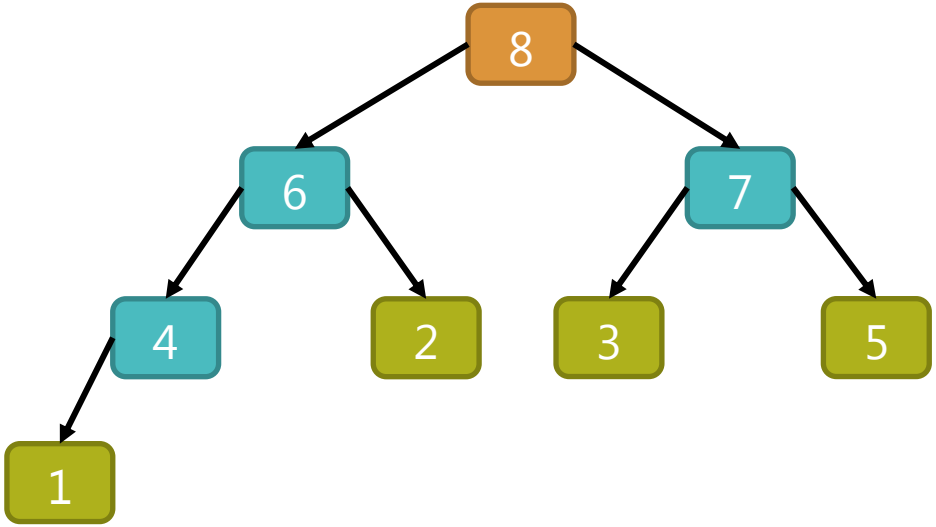
- A la fin de l'insertion dans le tas, les éléments ne sont pas pour autant triés : on sait seulement que la racine est l'élément de plus grand de la collection.

- On échange donc le dernier élément du tableau avec la racine (1er 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.

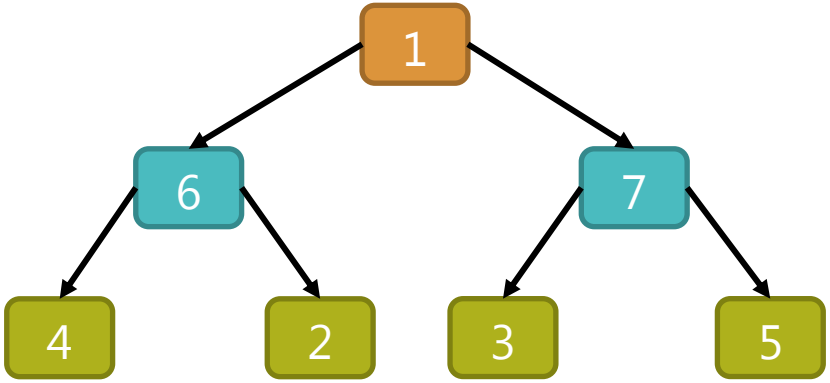
- On utilise l'opération de tamisage qui consiste à échanger la racine avec le plus grand de ses fils jusqu'à ce qu'elle soit à sa place.

- On recommence ensuite l'opération jusqu'à ce que toute la collection soit triée.

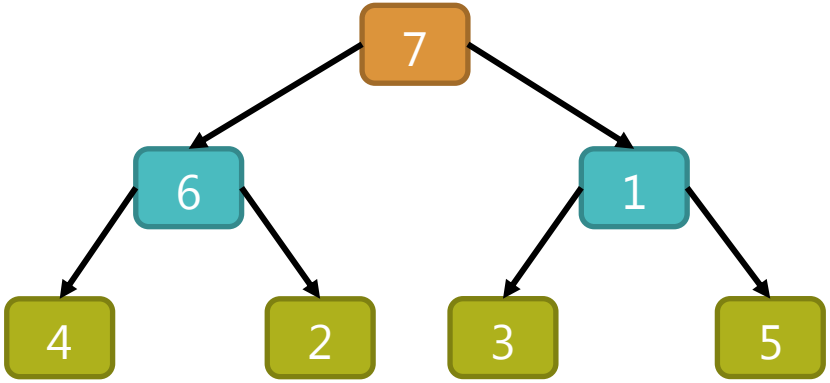
Tri pas tas : tamisages successifs (1/17)



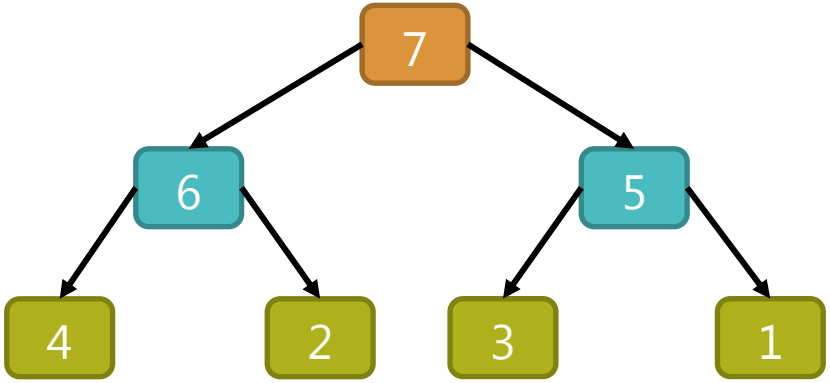
Tri pas tas : tamisages successifs (2/17)



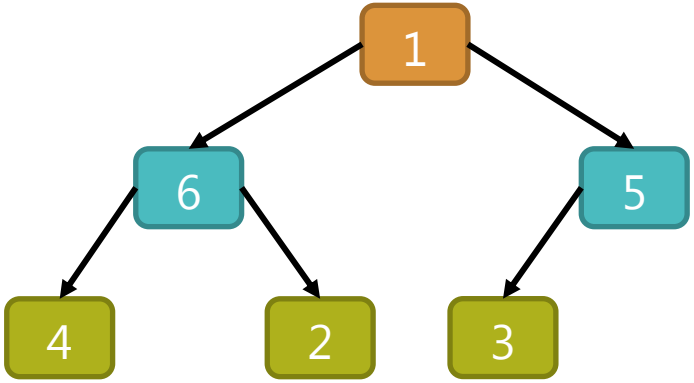
Tri pas tas : tamisages successifs (3/17)



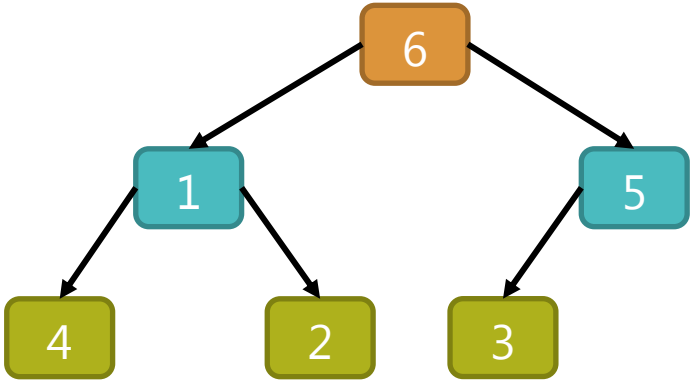
Tri pas tas : tamisages successifs (4/17)



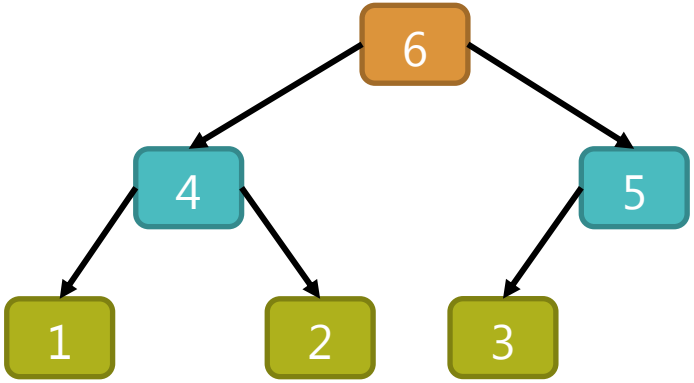
Tri pas tas : tamisages successifs (5/17)



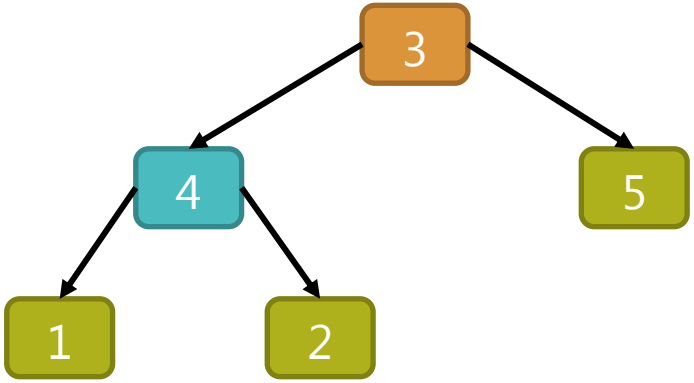
Tri pas tas : tamisages successifs (6/17)



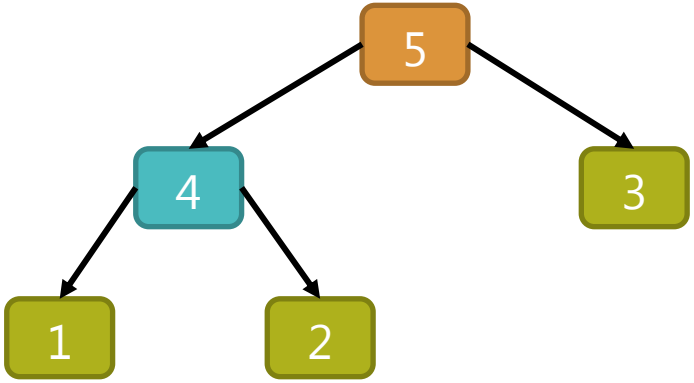
Tri pas tas : tamisages successifs (7/17)



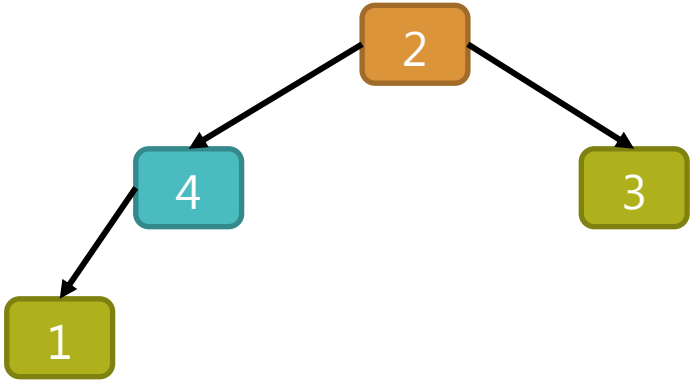
Tri pas tas : tamisages successifs (8/17)



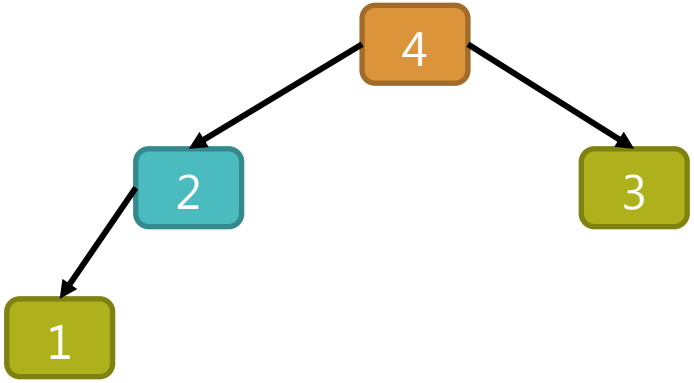
Tri pas tas : tamisages successifs (9/17)



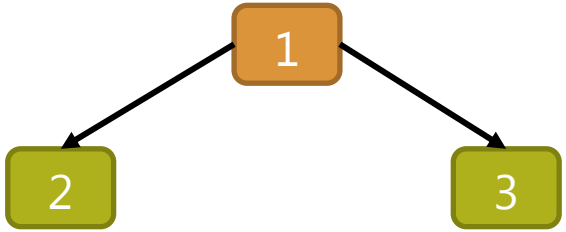
Tri pas tas : tamisages successifs (10/17)



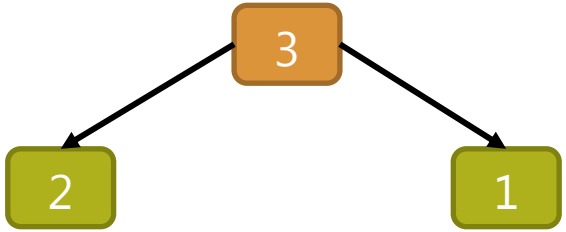
Tri pas tas : tamisages successifs (11/17)



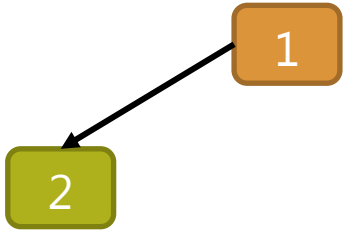
Tri pas tas : tamisages successifs (12/17)



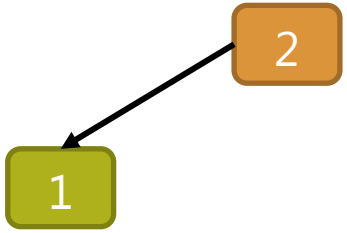
Tri pas tas : tamisages successifs (13/17)



Tri pas tas : tamisages successifs (14/17)



Tri pas tas : tamisages successifs (15/17)



Tri pas tas : tamisages successifs (16/17)

Tri pas tas : tamisages successifs (17/17)

Optimisations du tri pas 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 1ère 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

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

L'École des INGÉNIEURS Scientifiques