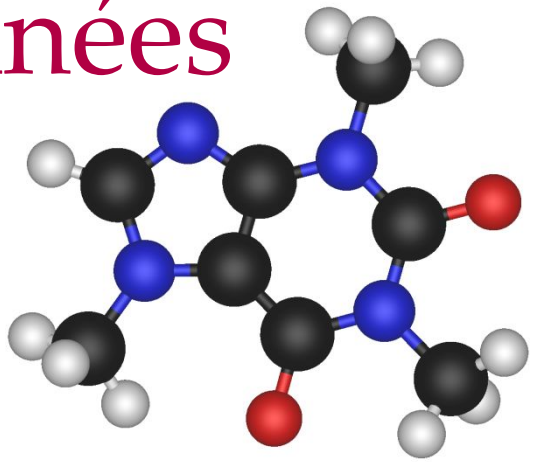
Structures de données



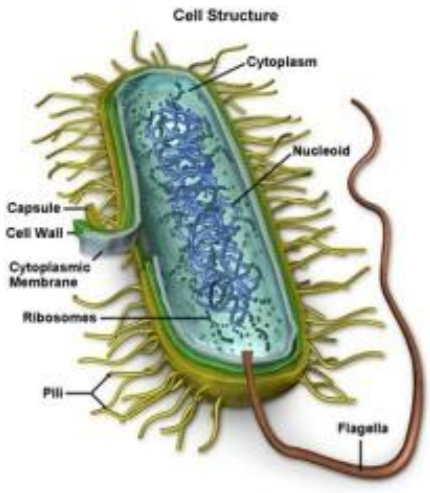
Thomas TANG

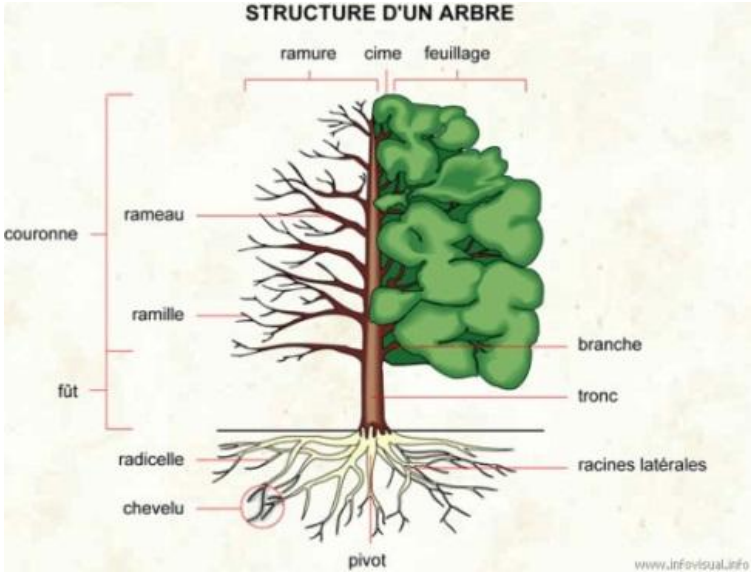
tang@ensea.fr

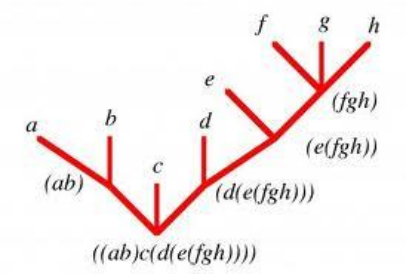
A partir du cours de N. SIMOND 2011-2012

- **dictionnaire** : manière dont les différentes parties d'un ensemble sont disposées entre elles et sont solidaires et ne prennent sens que par rapport à l'ensemble;
- **bâtiment** : ensemble des éléments nécessaires pour assurer la stabilité d'un ouvrage sous les actions qui lui sont appliquées;
- **sociologie** : regroupement des différents organes et relations (hiérarchiques et fonctionnelles) qui les lient;
- **géologie** : agencement des couches géologiques les unes par rapport aux autres;
- **chimie** : organisation agencée des différents éléments qui forment un ensemble;
- **math** : théorie fondée sur la théorie des ensembles complétée de tous les axiomes, signes et règles (str. algébrique, d'ordre, topologique, etc);

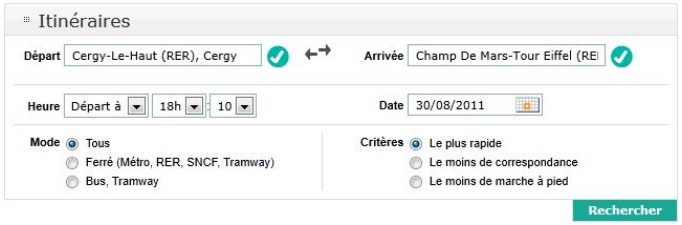
- C'est un regroupement sous la même entité du contenu de plusieurs variables :
  - où l'application exige que l'ordre d'apparition chronologique de nouvelles données justifie leur ordonnancement :
  - et où la notion de hiérarchie/dépendance entre les différents éléments doit clairement apparaître



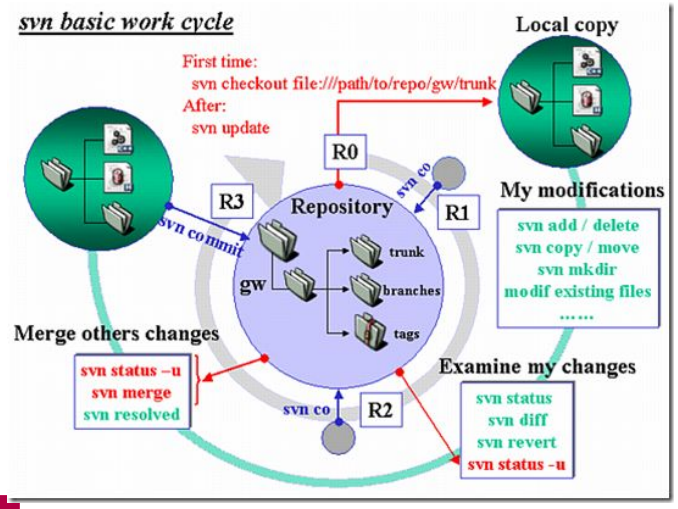


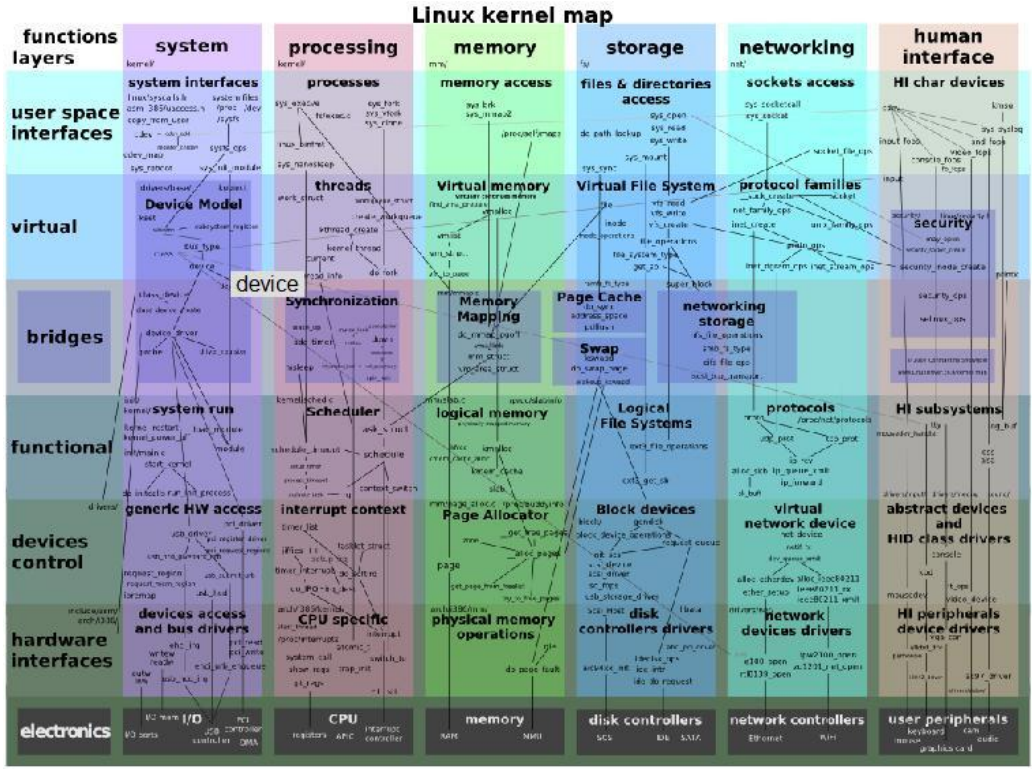

f
g
h
e
(fgh)
a
b
d
(e(fgh))
c
(ab)
(d(e(fgh)))
((ab)c(d(e(fgh))))


RUE DES CANETTES
RUE SAINT-SULPICE
NAVIGON


Itinéraires
Départ
Cergy-Le-Haut (RER), Cergy
Arrivée
Champ De Mars-Tour Eiffel (RE
Heure
Départ à
18h
10
Date
30/08/2011
Mode
Tous
Ferré (Métro, RER, SNCF, Tramway)
Bus, Tramway
Critères
Le plus rapide
Le moins de correspondance
Le moins de marche à pied
Rechercher


MySQL


svn basic work cycle
Local copy
First time:
svn checkout file:///path/to/repo/gw/trunk
After:
svn update
R0
R3
R1
svn co
svn commit
Repository
gw
trunk
branches
tags
My modifications
svn add / delete
svn copy / move
svn mkdir
modif existing files
.....
Merge others changes
svn status –u
svn merge
svn resolved
Examine my changes
svn status
svn diff
svn revert
svn status -u
svn co
R2


Linux kernel map
functions
layers
system
processing
memory
storage
networking
human interface
user space interfaces
system interfaces
processes
memory access
files & directories access
sockets access
HI char devices
virtual
Device Model
threads
Virtual memory
Virtual File System
protocol families
security
device
bridges
Synchronization
Memory Mapping
Page Cache
networking storage
Swap
functional
system run
Scheduler
logical memory
Logical File Systems
protocols
HI subsystems
devices control
generic HW access
interrupt context
Page Allocator
Block devices
virtual network device
abstract devices and HID class drivers
hardware interfaces
devices access and bus drivers
CPU specific
physical memory operations
disk controllers drivers
network devices drivers
HI peripherals device drivers
electronics
I/O
CPU
memory
disk controllers
network controllers
user peripherals

ENSEA

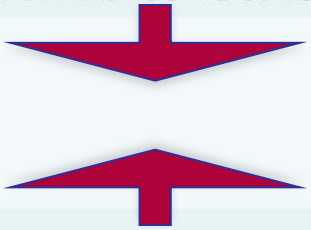
- Prérequis : cours langage C du premier semestre
- Requis pour comprendre
  - Microprocesseurs
  - Langages Orienté Objet

- Volume horaire
  - 2 cours magistraux
  - 3 séances de TP

- Notation (2 ECTS)
  - 1 note de TP
  - 1 note de CS

- Cours magistraux
  - Séance 1
    - Architecture matérielle & logicielle
    - Structures en C
    - Données abstraites : structures de données linéaires
  - Séance 2
    - Arbres
    - Graphes
- Séance de TP
  - 1 séance de formation
  - 2 séances d'application (fermeture de contours)

~ Variables globales et locales

Type mémoire	Zones	Données	STM32
ROM	Code	Programme	Code
RAM	Statique	Variables globales	Data
	Pile	Variables locales	Stack
			
	Tas	Allocation dynamique	Heap

- pile : variables locales
- tas : variables locales pouvant être allouées ou supprimées par l'utilisateur

```
char b;

int ordonnee(int a, int x){
    int y = a * x + b;
    return y;
}

int main(void){
    int p = 3, x = 4, res;
    b = 5;
    res = ordonnee(p,x);
    return 0;
}
```

0x28ABB0	00	00	00	00	
0x28ABB4	90	9B	0B	10	y
0x28ABB8	E8	AB	28	00	

0x28ABC0	00	00	00	03	a
0x28ABC4	00	00	00	04	x
0x28ABC8	00	00	00	00	

0x28ABDC	0C	32	0B	61	res
0x28ABE0	2B	00	00	04	x
0x28ABE4	00	00	00	03	p

0x4030DC	00	00	00	00	
0x4030E0	05	00	00	00	b
0x4030E4	00	00	00	00	

```
#include <stdio.h>

int main(void){
    int *pq;
    float *n = NULL;
    int q = 5;
    pq = &q;
    char s[10] = "Salut";
    long f[3] = {-2 147 483 648, 0, 1};
    return 0;
}
```

0x28ABB0	00	00	00	00	f
0x28ABB4	00	00	00	00	
0x28ABB8	01	00	00	00	
0x28ABC0	53	61	6C	75	s
0x28ABC4	74	00	00	00	
0x28ABC8	00	00	00	00	
0x28ABD4	B6	0E	DA	05	q
0x28ABD8	2F	00	00	00	n
0x28ABDC	60	38	A8	F1	pq

~ Intérêts des pointeurs

- Travailler au plus proche du matériel (adresses)
- Passage par adresse des arguments d'une fonction
  - Fonction échange
  - Fonction renvoyant plus d'une seule donnée
- Allocation dynamique de la mémoire
  - Problème : connaître à l'avance le nombre de cases mémoires
  - Inconvénient d'une allocation statique : pas assez ou trop de cases
  - Solution : Allocation du nombre exact de cases ET évolution possible.

```
#include <stdlib.h>

int main(void){
    int tab[10];
    int *pm, *pc;
    int i;
    pm = (int*)malloc(10*sizeof(int));
    for(i=0;i<10;i++) pm[i]=i;
    pc = (int*)calloc(10,sizeof(int));
    pm = (int*)realloc(pm,20*sizeof(int));
    for(i=10;i<20;i++) pm[i]=i;
    free(pm);
    free(pc);
    return 0;
}
```

0x28ABA4	00	00	00	00	i
0x28ABA8	20	03	9C	08	pc
0x28ABAC	20	03	9C	B8	pm
0x28ABB0	00	00	01	00	tab
0x28ABB4	00	00	00	00	

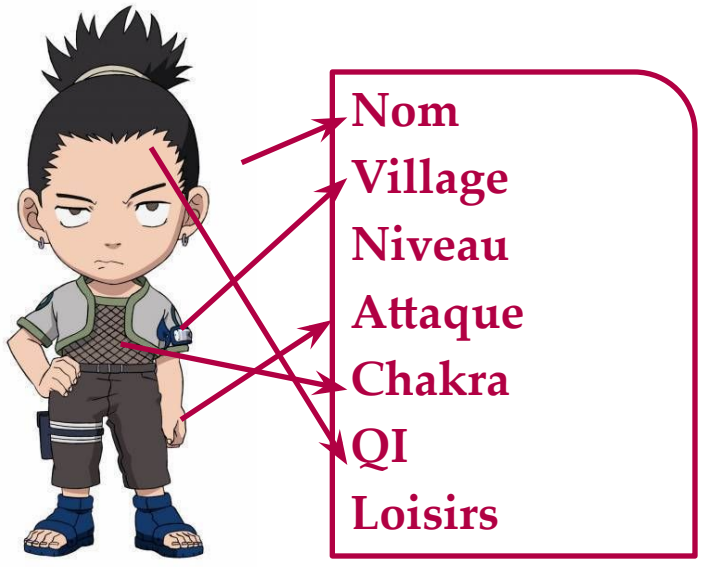
0x20039CB0	00000000	33000000	00000000	00000001
0x20039CC0	00000002	00000003	00000004	00000005
0x20039CD0	00000006	00000007	00000008	00000009
0x20039CE0	0000000A	0000000B	0000000C	0000000D
0x20039CF0	0000000E	0000000F	00000010	00000011

~ Récapitulatif

- INCLURE la librairie `stdlib`.
- INITIALISER un pointeur.
- ALLOUER (`malloc` ou `calloc`) la mémoire nécessaire en CONVERTISSANT EXPLICITEMENT au type du pointeur.
- VERIFIER si l'allocation a été réalisée.
- AFFECTER les valeurs au tableau alloué.
- REDIMENSIONNER (`realloc`) le tableau si besoin est.
- VERIFIER si le redimensionnement a fonctionné.
- AFFECTER les nouvelles valeurs.
- …
- DESALLOUER (`free`) la mémoire.

~ Besoin

- Le stockage de données sous forme de **tableaux** impose un type identique au sein d'un même ensemble. On veut pouvoir sauvegarder sous le même nom des données de types différents.




```
#define NBLETTRES 20

struct ninja{
    char name[NBLETTRES+1];
    char village[NBLETTRES+1];
    int level;
    char bestattack[NBLETTRES+1];
    float chakra;
    unsigned int qi;
    char leisure[NBLETTRES+1];
};
```

- Initialisation

```
#define NBLETTRES 20
struct ninja{
    char name[NBLETTRES+1];
    char village[NBLETTRES+1];
    int level;
    char bestattack[NBLETTRES+1];
    float chakra;
    unsigned int qi;
    char leisure[NBLETTRES+1];
};
```

```
struct ninja shika;
strcpy(«Shikamaru», shika.name);
strcpy(«Konoha», shika.village);
shika.level=3;
strcpy(«Kage Mane», shika.bestattack);
shika.chakra=1000.0;
shika.qi=200;
strcpy(«Shogi», shika.leisure);
```

- Définir un nouveau type en C

```
typedef struct ninja NINJA;
NINJA Shika;

#define NBLETTRES 20

typedef int LEVEL;
typedef char[NBLETTRES+1] STRING;

typedef struct ninja{
    STRING name;
    STRING village;
    LEVEL level;
    STRING bestattack;
    float chakra;
    double qi;
    STRING leisure;
}NINJA;
```

- Initialisation et accès aux champs

```
typedef struct ninja{
    STRING name;
    STRING village;
    LEVEL level;
    STRING bestattack;
    float chakra;
    double qi;
    STRING leisure;
}NINJA;
```

```
NINJA* shika;
shika=(NINJA*)malloc(sizeof(NINJA));
strcpy(«Shikamaru», shika->name);
strcpy(«Konoha», shika->village);
shika->level=3;
strcpy(«Kage Mane», shika->bestattack);
shika->chakra=1000.0;
shika->qi=200;
strcpy(«Shogi», shika->leisure);
```

~ Définition, objectif et moyen

- Un **ensemble** (math.) est une collection d'objets distinguables. Il est dit **dynamique** lorsqu'il peut subir des modifications (croissance, diminution) au cours du temps.

- Le but recherché dans ce cours est d'implémenter ces ensembles dynamiques.

- La meilleure façon d'implémenter un ensemble dynamique dépend des opérations qui doivent être supportées.

~ Elément d'un ensemble dynamique

- Représenté par un objet dont les champs peuvent être examinés et manipulés si on dispose d'un pointeur sur l'objet
- Présence possible d'une clé servant d'identifiant à l'objet.
- Données satellites pouvant:
  - ne pas intervenir dans l'implémentation de l'ensemble
  - au contraire, être manipulés par les opérations ensemblistes, comme des pointeurs sur d'autres objets de l'ensemble.

~ Opérations sur un ensemble dynamique

- Opérations de modification
  - Insérer
  - Supprimer
- Requêtes
  - Trouver_minimum
  - Trouver_maximum
  - Trouver_suivant
  - Trouver_précédent

~ Définition

- Une **structure de données** est une structure logique destinée à stocker des données, ayant une organisation prédéfinie afin d'en simplifier leur traitement. Elle est définie par le cahier des charges d'un type abstrait.

- Un **type abstrait** est une spécification mathématique
  - d'un ensemble de données et
  - de l'ensemble des opérations qu'elles peuvent effectuer.

- permet donc l'implémentation d'un ensemble dynamique.

- Opérateurs
  - constructeur/destructeur
    crée/supprime une nouvelle structure
  - modifieurs
    insère/retire un élément de structure
  - électeurs
    renseignent sur l'état de la structure (vide ou pas, profondeur de la structure, etc)
  - itérateurs
    permettent de parcourir la structure pour rechercher un élément.

- Files et piles
  - Structures de données pour lesquelles l'élément ôté par l'opération SUPPRIMER est spécifié à l'avance.

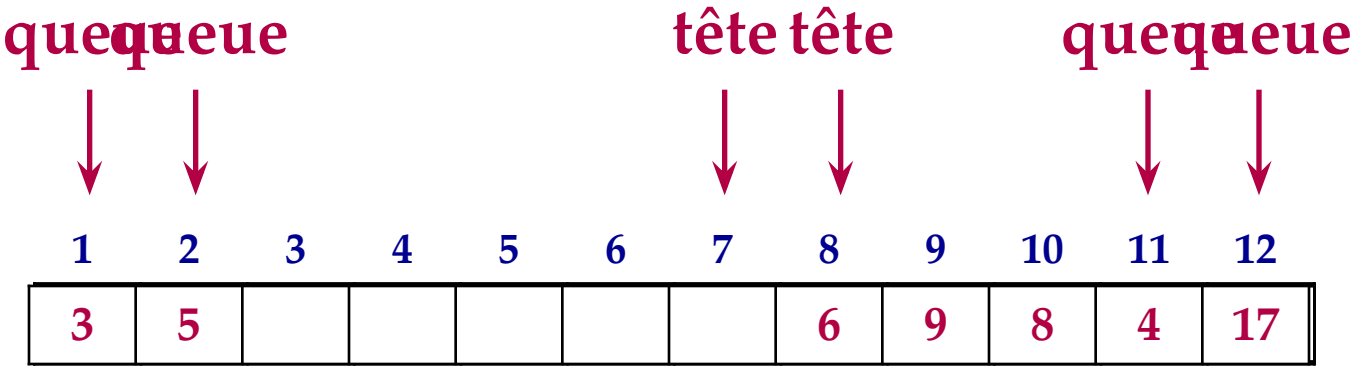
~ Définition

- Une **file** est une collection d'éléments équivalents souhaitant accéder à un service et classés uniquement selon leur ordre d'apparition.

~ Protocole

- La file est régie par le protocole FIRST IN FIRST OUT (FIFO).

~ Fonctionnement



ENFILER 17

ENFILER 3

ENFILER 5

DEFILER 15

~ Implémentation

ELEMENT	OPERATEURS

TAILLE = N ;
 	
structure File {
élément Tab[TAILLE] ;
entier tête ;
entier queue ;
}
	•constructeur/destructeur :
	[File F] = FileCrée(File F) ;
	[File F] = FileDétruit(File F) ;
	•modifieurs :
	[File F] = FileEnfile(File F, élément e) ;
	[File F, élément e] = FileDéfile(File F) ;
	•sélecteurs :
	[booléen file_vide] = FileVide(File F) ;
	[entier prof] = FileProfondeur(File F) ;
	•itérateur :
	[booléen trouvé, entier indice] =
	FileRecherche(File F, élément e) ;

~ Constructeur & Destructeur

[File F] = FileCrée(File F) ;

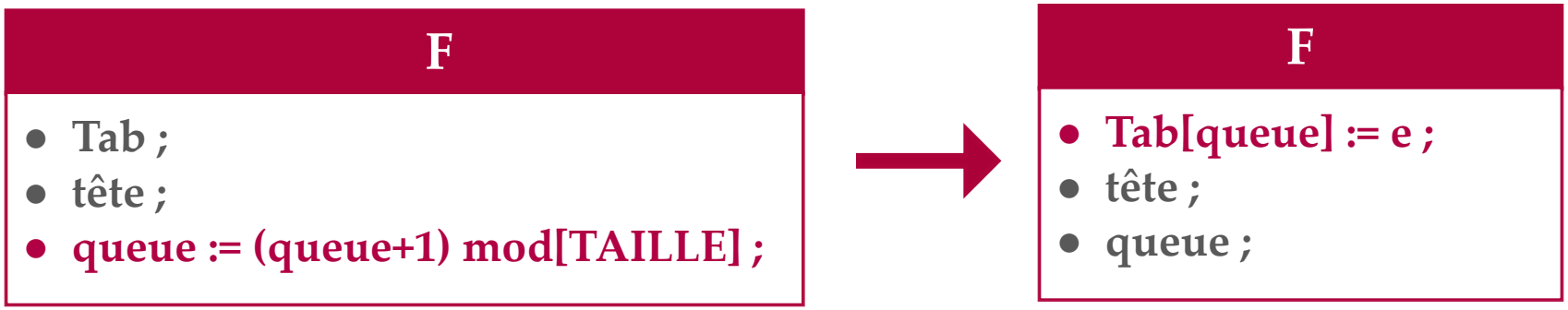
F
• allocation Tab[TAILLE]; • tête := 0 ; • queue := 0 ;

[File F] = FileDétruit(File F) ;

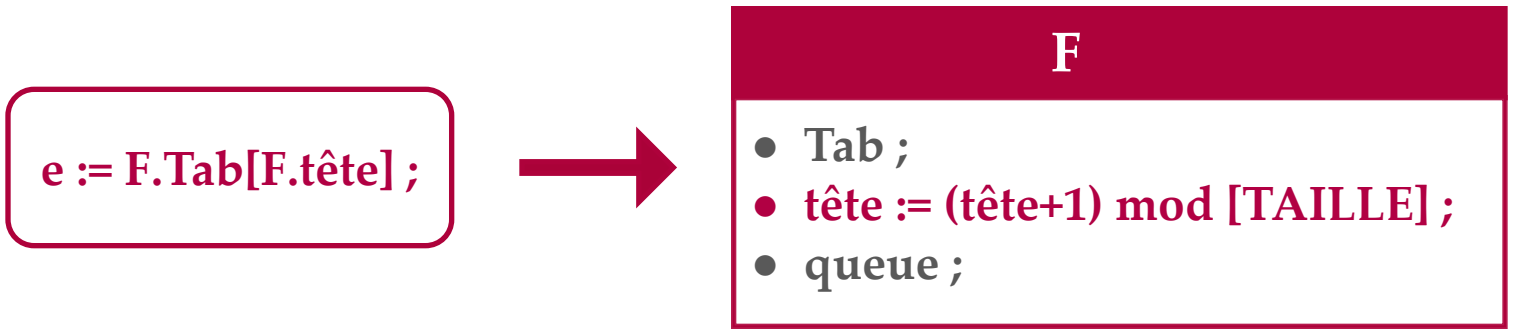
F
• désallocation Tab[]; • tête ; • queue ;

~ Modifieurs

[File F] = FileEnfile(File F, élément e) ;



[File F, élément e] = FileDéfile(File F) ;



~ Sélecteurs

[booléen file_vide] = FileVide(File F) ;

file_vide := (F.queue == F.tête) ;

[entier prof] = FileProfondeur(File F) ;

prof := (F.queue - F.tête) mod[TAILLE] ;

~ Itérateur

[booléen found, entier index] = FileRecherche(File F, élément e) ;

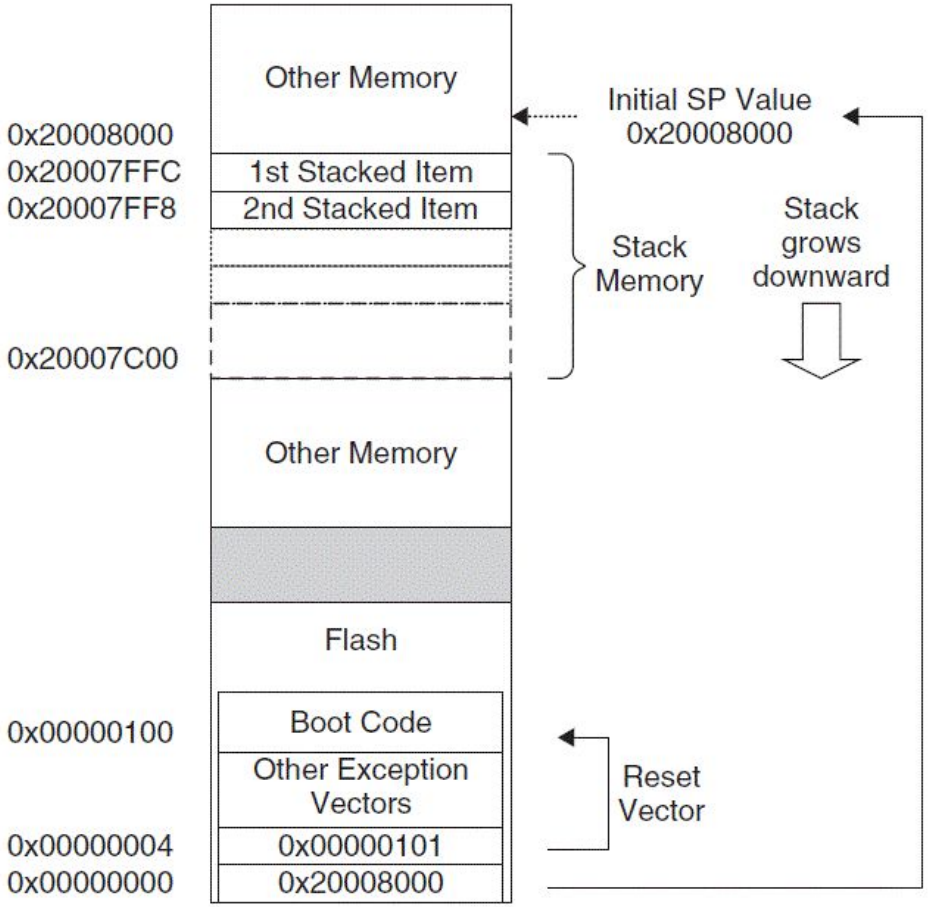
```
booléen trouvé := faux ;
entier indice := F.tête ;
Tant que ((trouvé==faux)  Et (indice ≤ F.queue) ) {
        Si F.Tab[indice] == e)
        Alors trouvé := vrai ;
        Sinon indice := indice+1 ;
        Fin Si
}
Renvoie trouvé, indice ;
```

~ Définition

- Une **pile** est une collection d'éléments à laquelle l'accès est limité au dernier élément ajouté, appelé sommet de la pile.

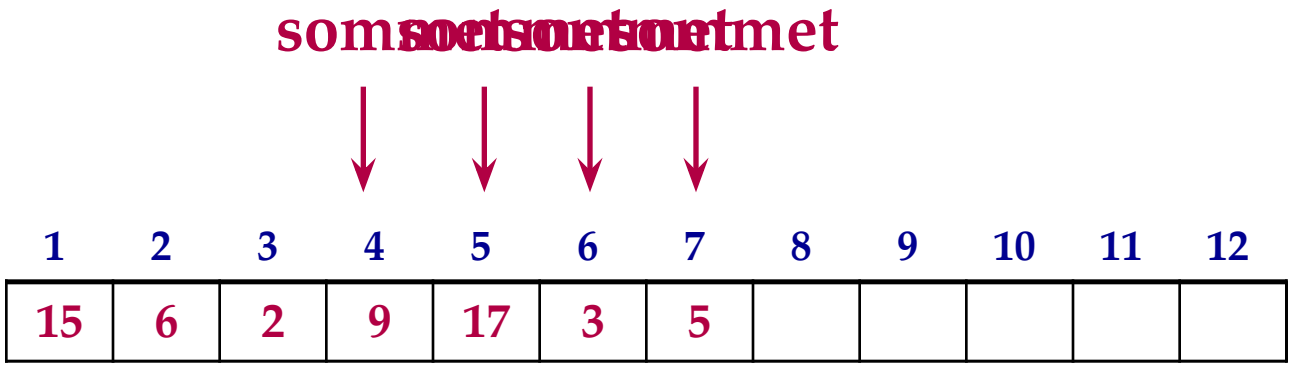
~ Protocole

- La pile est régie par le protocole LAST IN FIRST OUT (LIFO).



ENSEA

~ Fonctionnement



EMPILER 17

EMPILER 3

EMPILER 5

DEPILER

~ Implémentation

ELEMENT	OPERATEURS

TAILLE = N ;
 	
Structure Pile {
élément Tab[TAILLE] ;
entier sommet ;
}
	•constructeur/destructeur :
	[Pile P] = PileCrée(Pile P) ;
	[Pile P] = PileDétruit(Pile P) ;
	•modifieurs :
	[Pile P] = PileEmpile(Pile P, élément e) ;
	[Pile P, élément e] = PileDépile(Pile P) ;
	•sélecteurs :
	[booléen pile_vide] = PileVide(Pile P) ;
	[entier sommet] = PileProfondeur(Pile P) ;
	•itérateur :
	[booléen trouvé, entier indice] =
	PileRecherche(Pile P, élément e) ;

~ Modifieurs

[Pile P] = PileEmpile(Pile P, élément e) ;

P
• Tab ; • sommet := sommet+1;

→

P
• Tab[sommet] := e ; • sommet ;

[Pile P, élément e] = PileDépile(Pile P) ;

e := P.Tab[P.sommet] ;

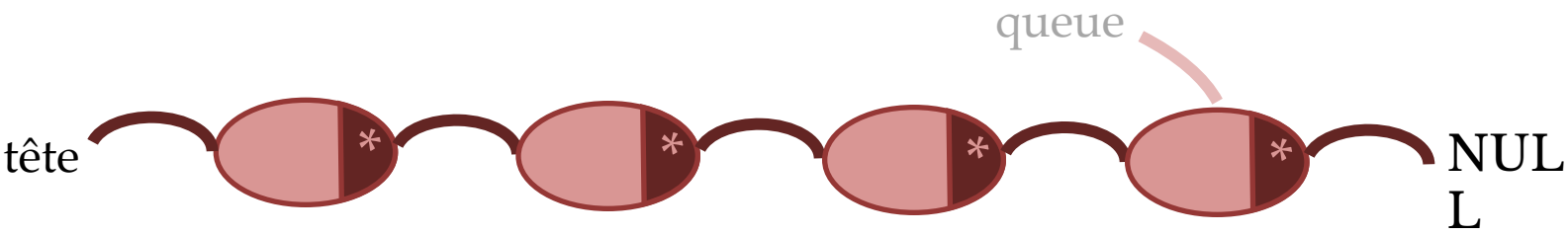
→

P
• Tab ; • sommet := (sommet+1);

~ Listes chaînées

Structures de données dans lesquelles les éléments sont arrangés linéairement.

- Les éléments de la liste sont reliés par un pointeur présent dans chaque objet (ordre virtuel en opposition à l'ordre linéaire des indices d'un tableau)
- Une liste chaînée est définie par sa tête et sa queue.
- Contrairement aux piles et aux files un élément peut être ajouté ou retiré où l'on veut.



- Avantages des tableaux
  - une unique réservation de mémoire (contigüe),
  - l'adresse du tableau est connue,
  - l'accès direct à l'élément d'index k.
- Avantages des listes chaînées
  - la souplesse de l'utilisation,
  - l'optimisation de la zone mémoire occupée.
- La liste chaînée est régie par :
  - l'allocation dynamique en mémoire,
  - la mise en œuvre moins évidente.

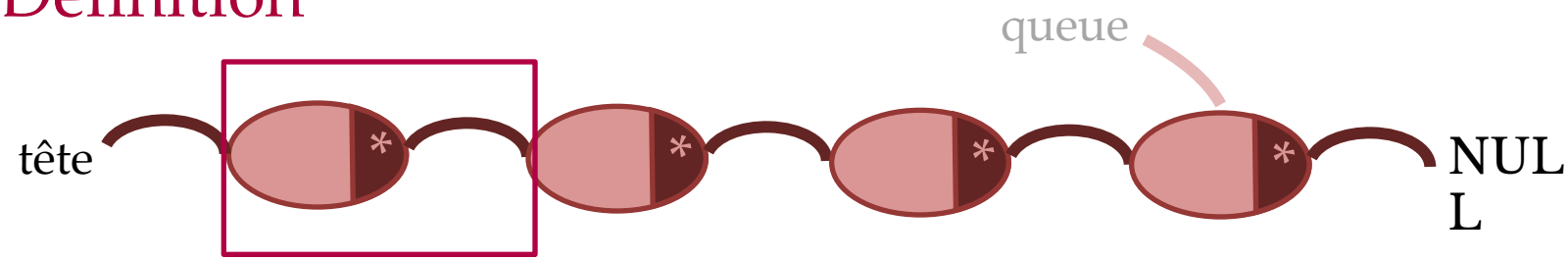
~ Définitions

- Une entité est définie récursivement si la définition comporte la mention de l'entité
- Un processus de traitement est récursif s'il se réfère à lui-même.

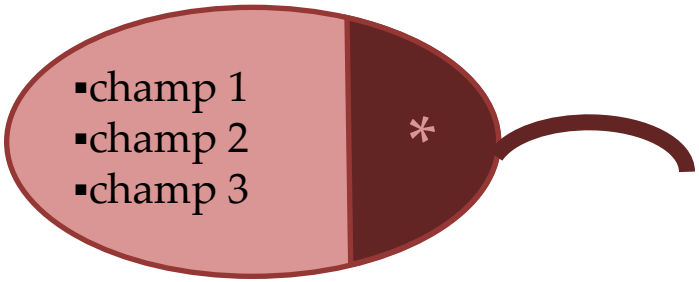
SI N vaut 1 ALORS **la valeur vaut 1** **SINON** **la valeur vaut N * valeur de Fact(N-1)** **FIN SI**	**valeur = N;** **Boucler** **SI i inférieur ou égal à 1** **sortir boucle** **FIN SI** **valeur = valeur * i** **décrémenter i** **FIN Boucler**
N-1 multiplications **N-1 appels récursifs de la fonction**	**N-1 multiplications**

	Récursivité	**Itérations**
Simplicité	+	-
Formulation	+	-
Temps d'éxécution	-	+
Mémoire	-	+

~ Définition



Une structure auto-référentielle (ou récursive) est une structure dont au moins un des champs est un **pointeur** vers une structure de même type. Les autres champs peuvent être de tous types :



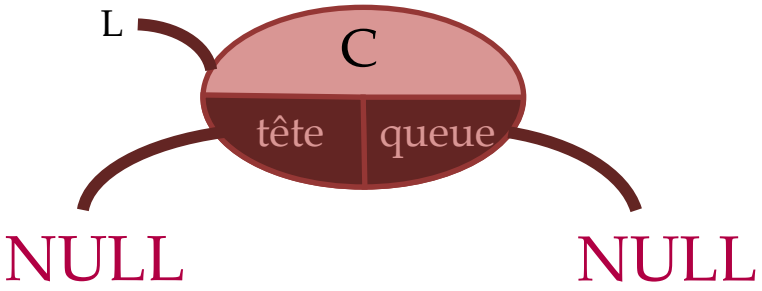
Structures et pointeurs

```
typedef enum {false,true} boolean;

typedef struct Node* Pointer;

typedef struct Node{
   elem e;
   Pointer link;
} NODE;

typedef struct Chain* LList;

typedef struct Chain {
   Pointer head;
   Pointer tail;
} CHAIN;
```

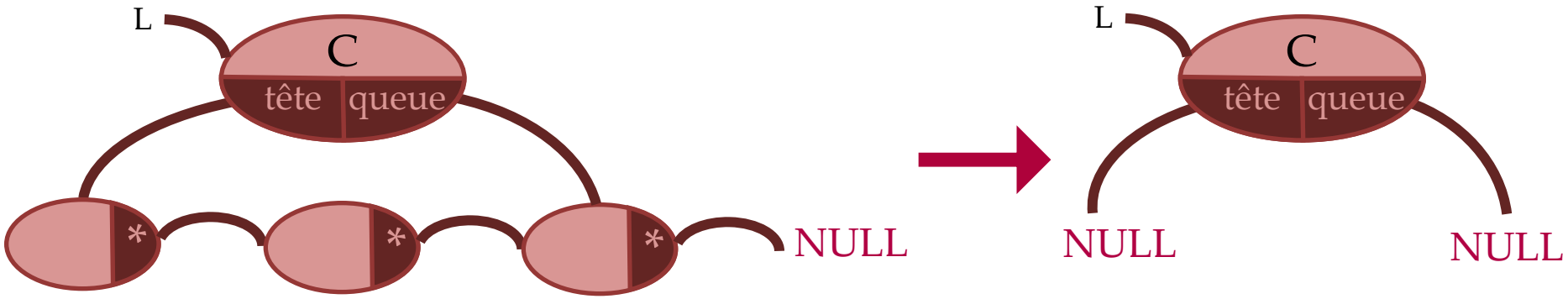
Opérateurs

- constructeur/destructeur :
  - void LLCreate(LList L);
  - void LLEmpty(LList L);
- modifieurs
  - void LLInsertHead(LList L, Pointer P);
  - void LLInsertAfter(LList L, Pointer P, elem cond);
  - void LLInsertBefore(LList L, Pointer P, elem cond);
  - void LLInsertTail(LList L, Pointer P);
  - Pointer LLDeleteHead(LList L);
  - Pointer LLDelete(LList L, elem cond);
  - Pointer LLDeleteTail(LList L);
- sélecteurs
  - boolean isLListEmpty(LList L);
  - int LListeSize (LList L);
- itérateurs
  - boolean LListSearch(LList L, Pointer P, elem cond);
  - Pointer LListeNodeK(LList L, int k);
  - int LListOccurence(LList L, elem cond);

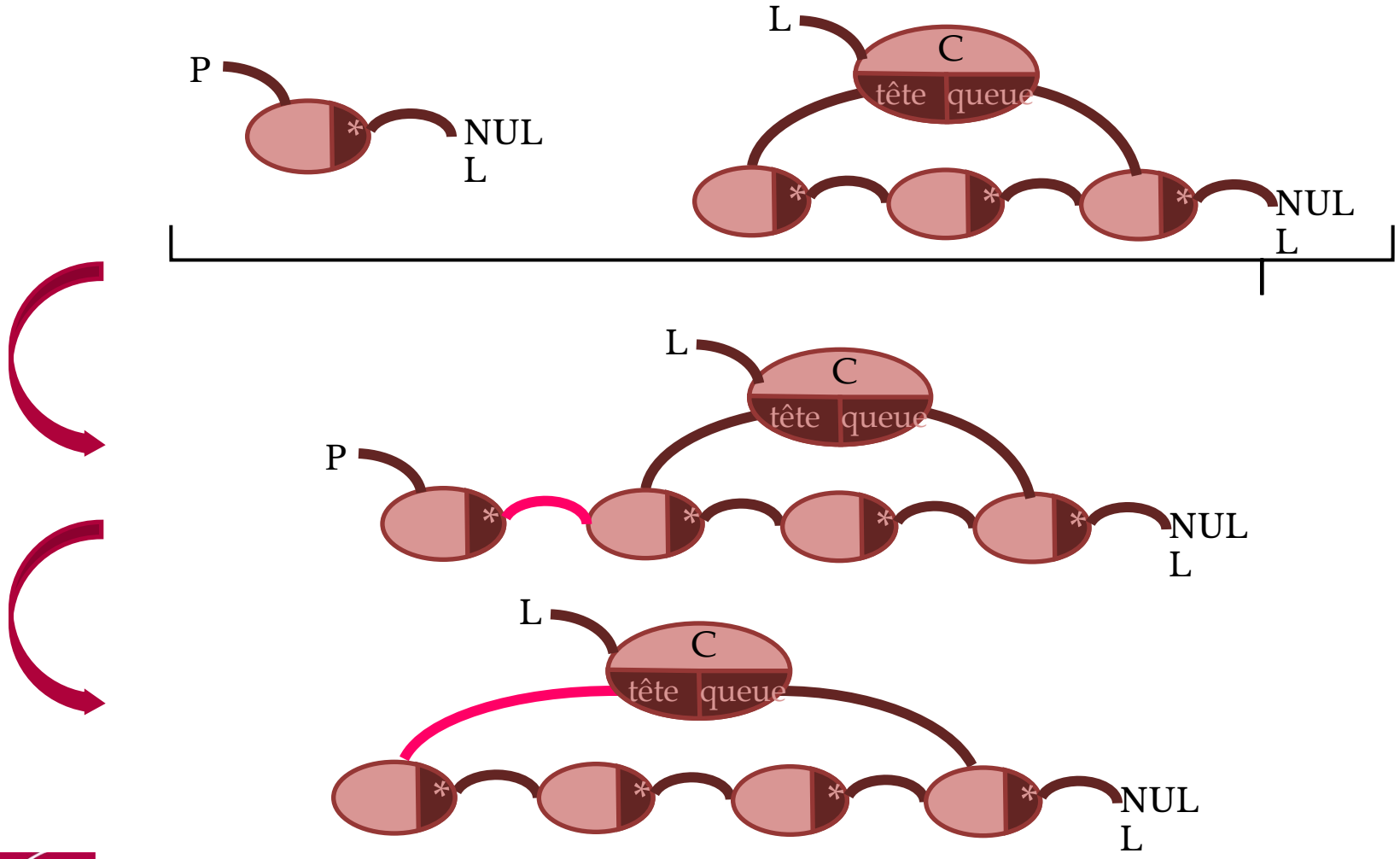
[LList L] = LLCreate(LList L) ;



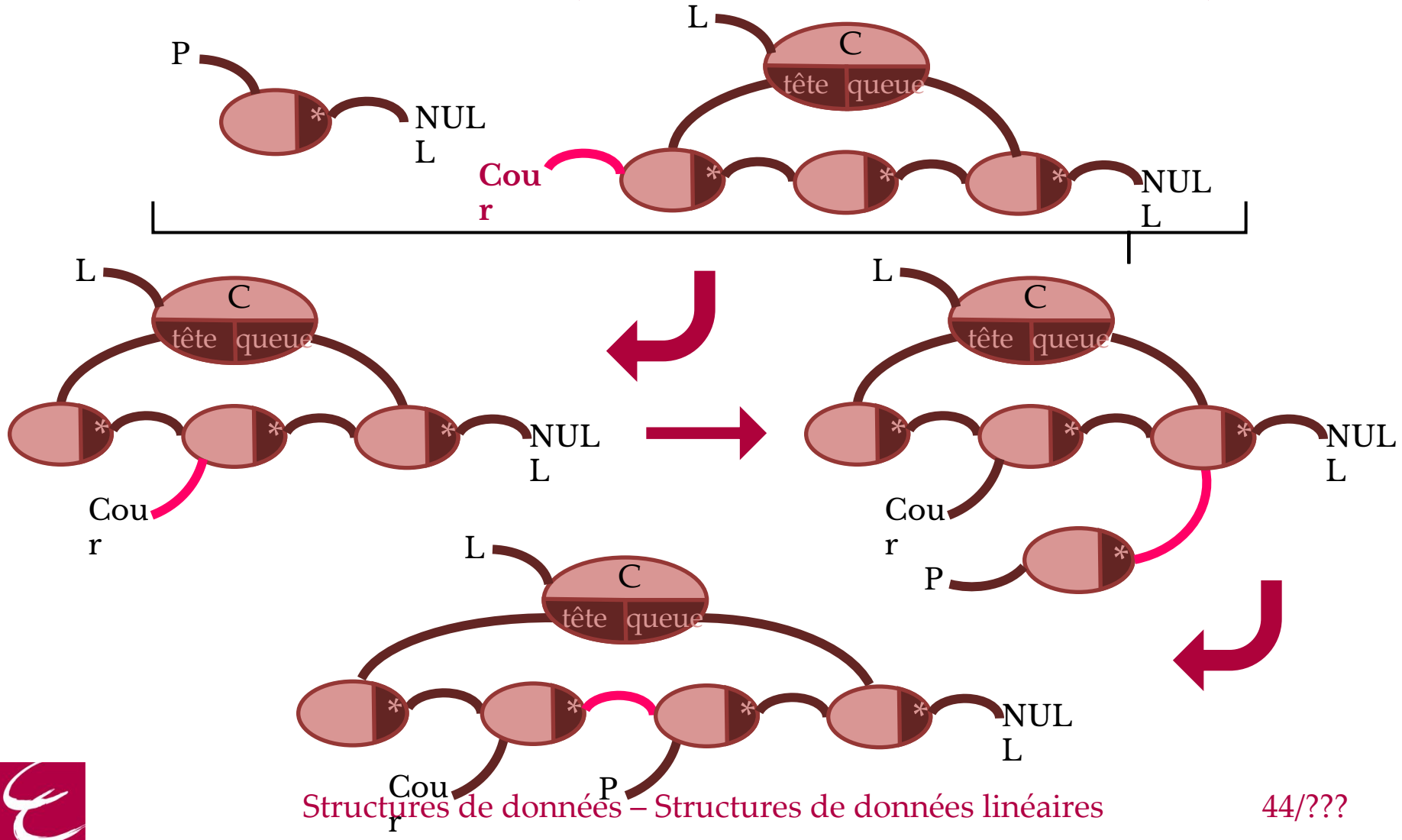
[LList L] = LLEmpty(LList L) ;



[LList L] = LLInsertHead(LList L, Pointer P) ;

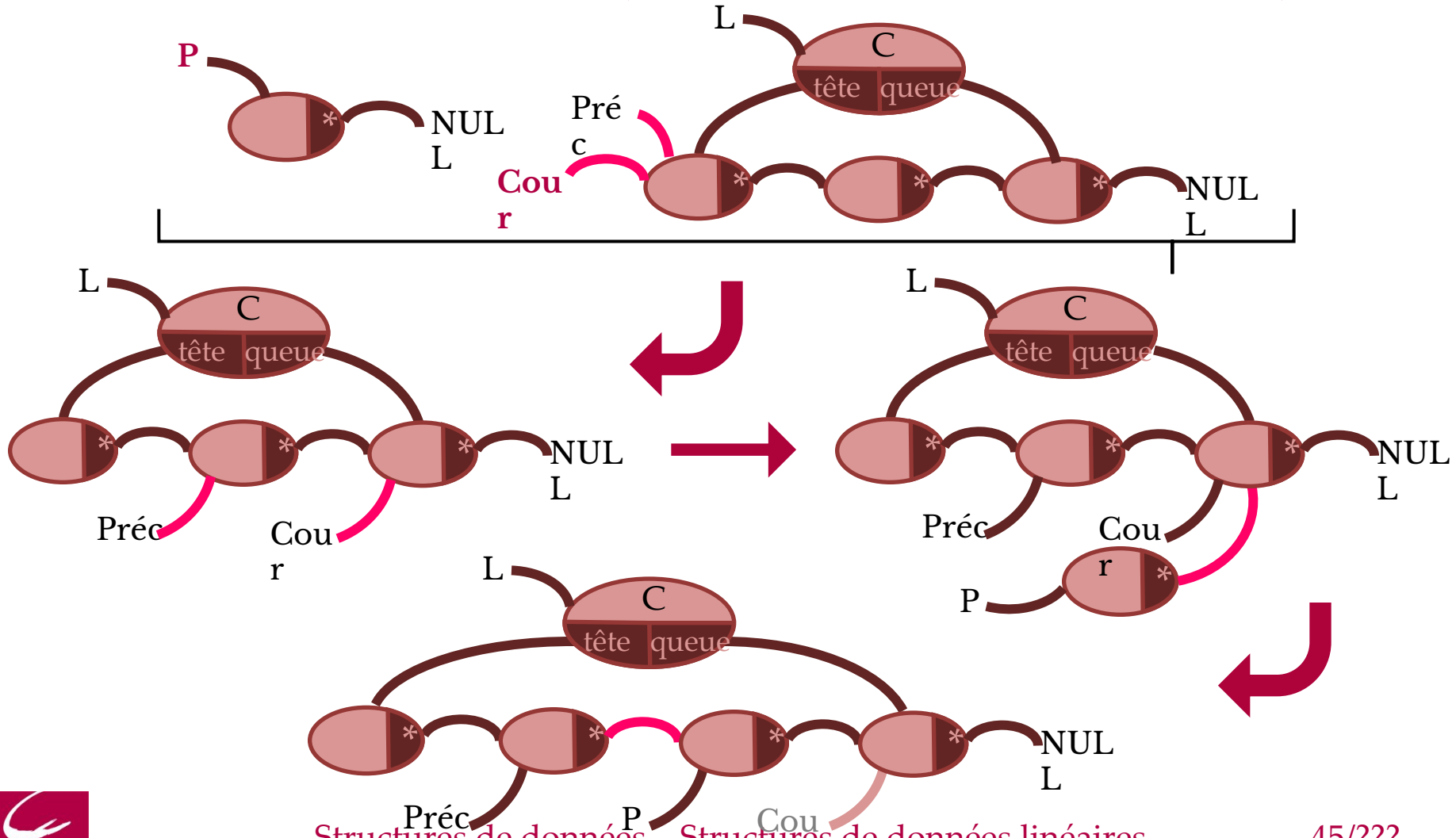


[LList L] = LLInsertAfter(LList L, Pointer P, elem cond) ;

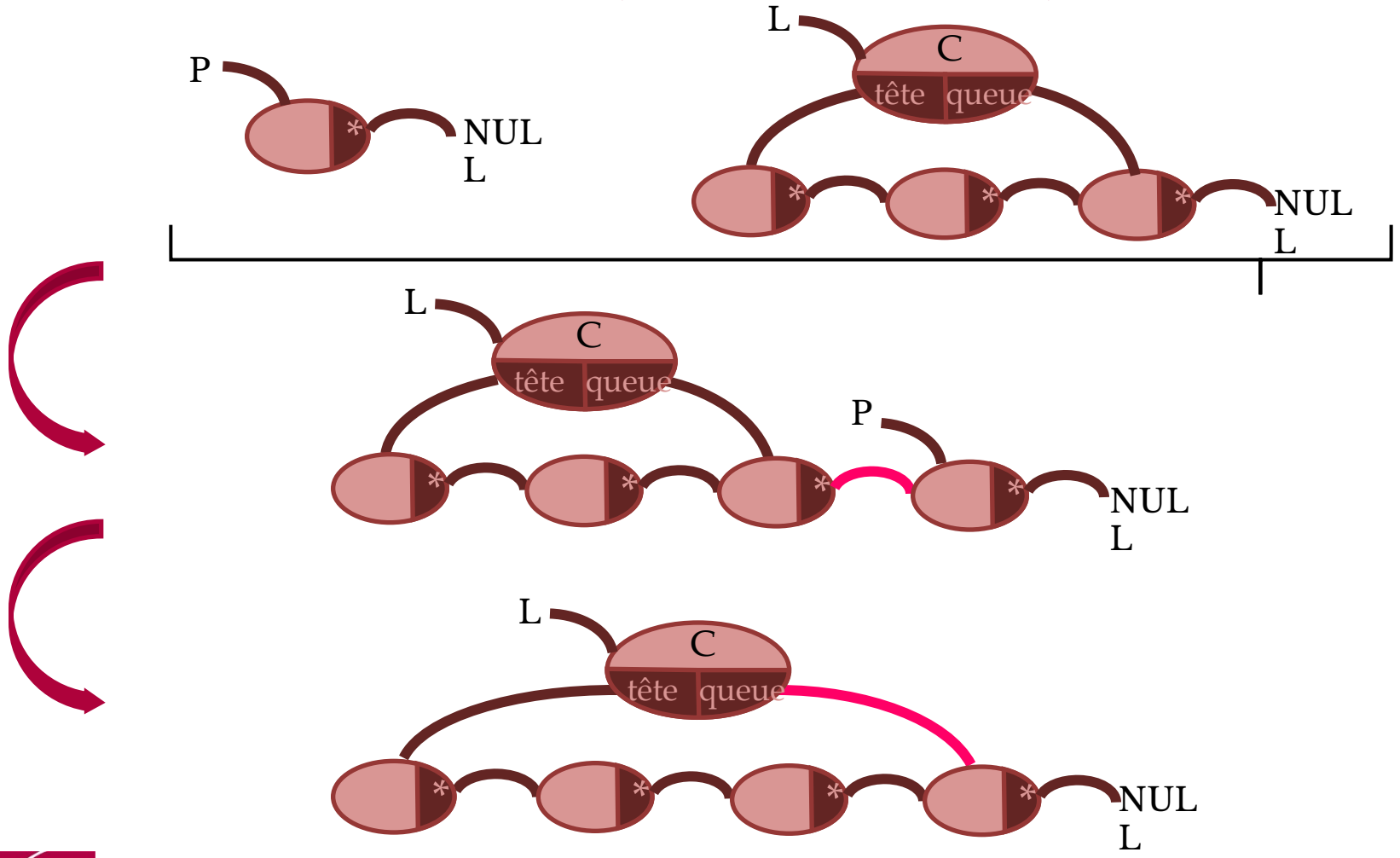


ENSEA

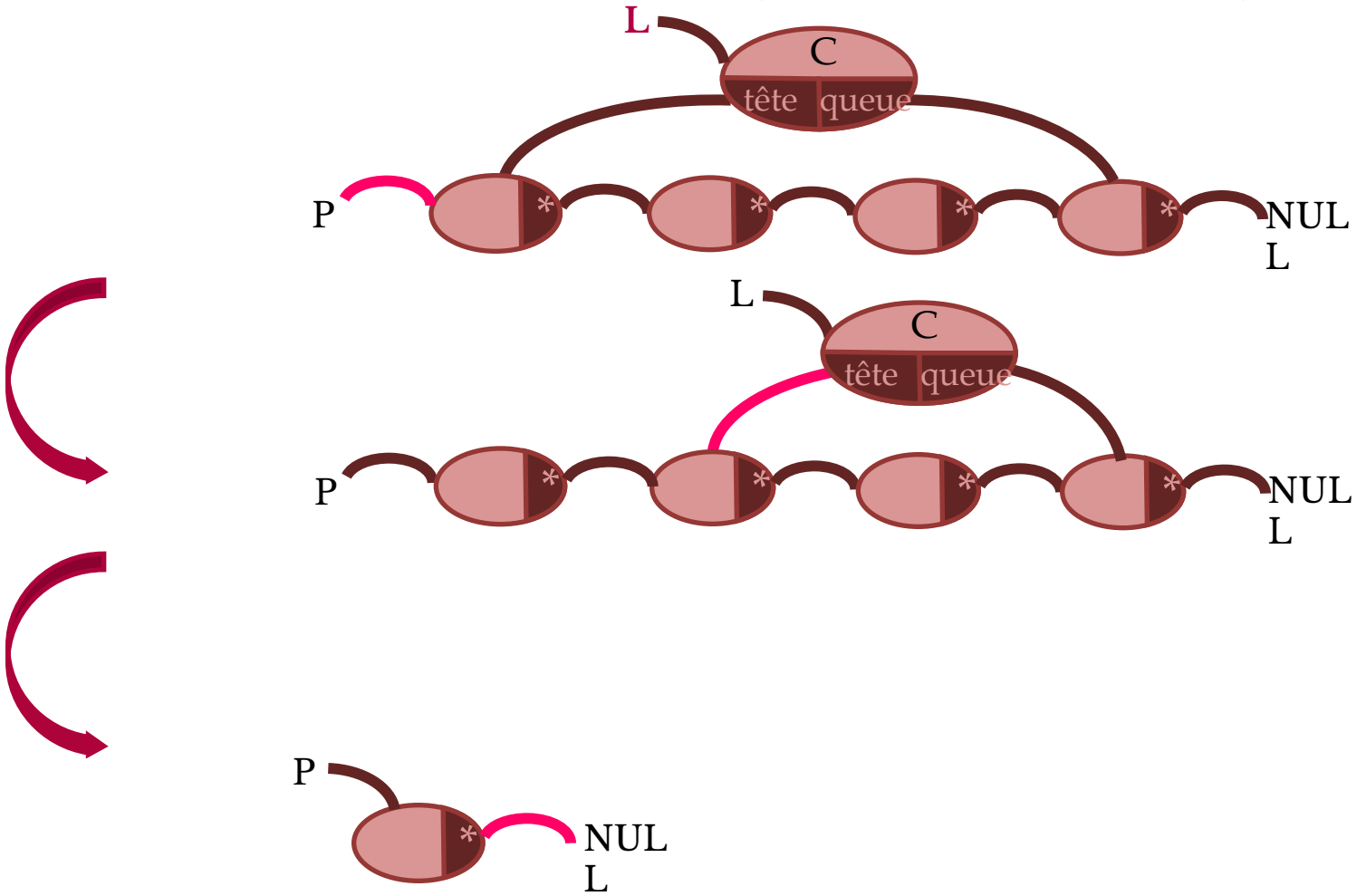
[LList L] = LLInsertBefore(LList L, Pointer P, elem cond) ;



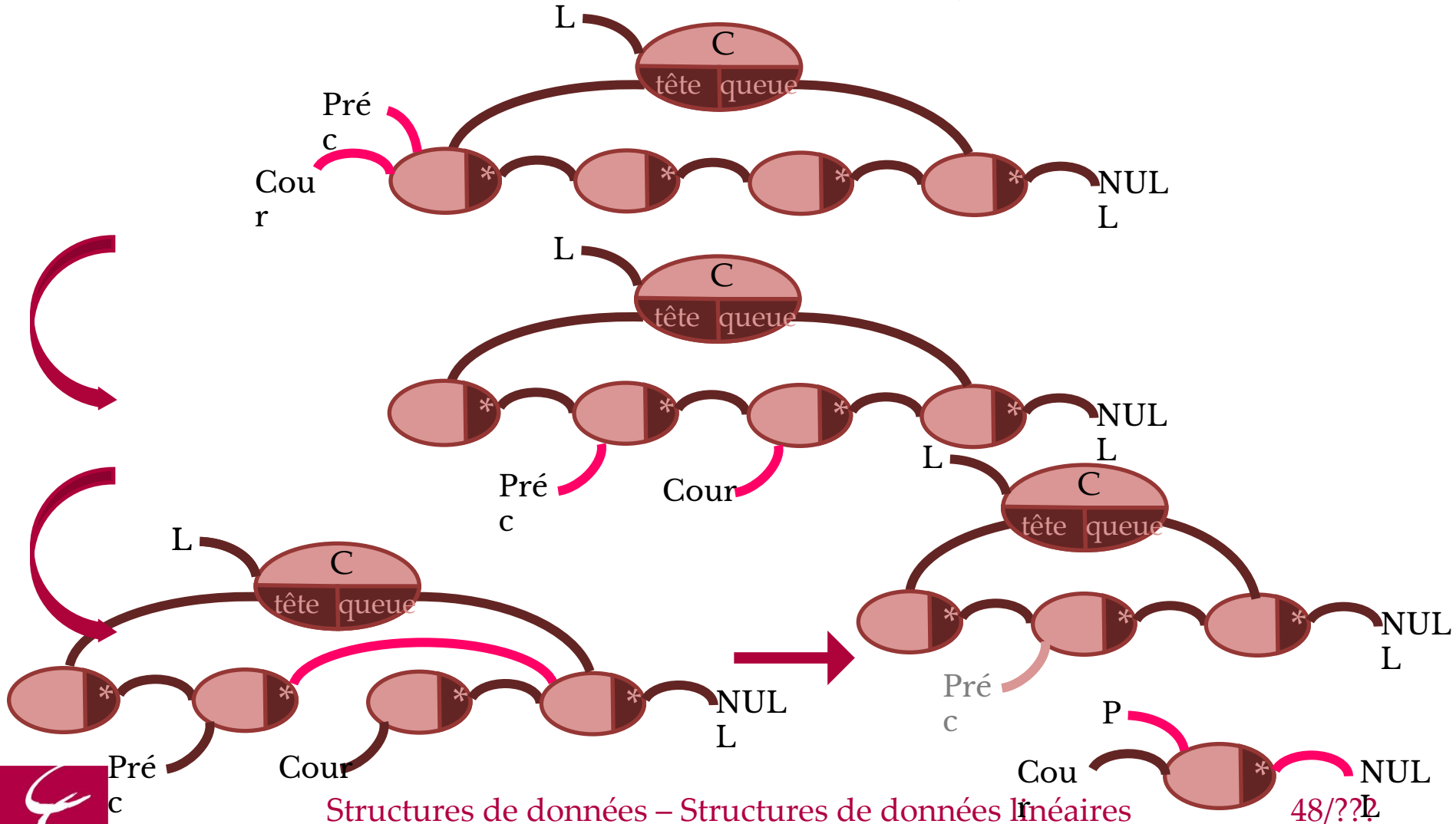
[LList L] = LLInsertTail (LList L, Pointer P) ;



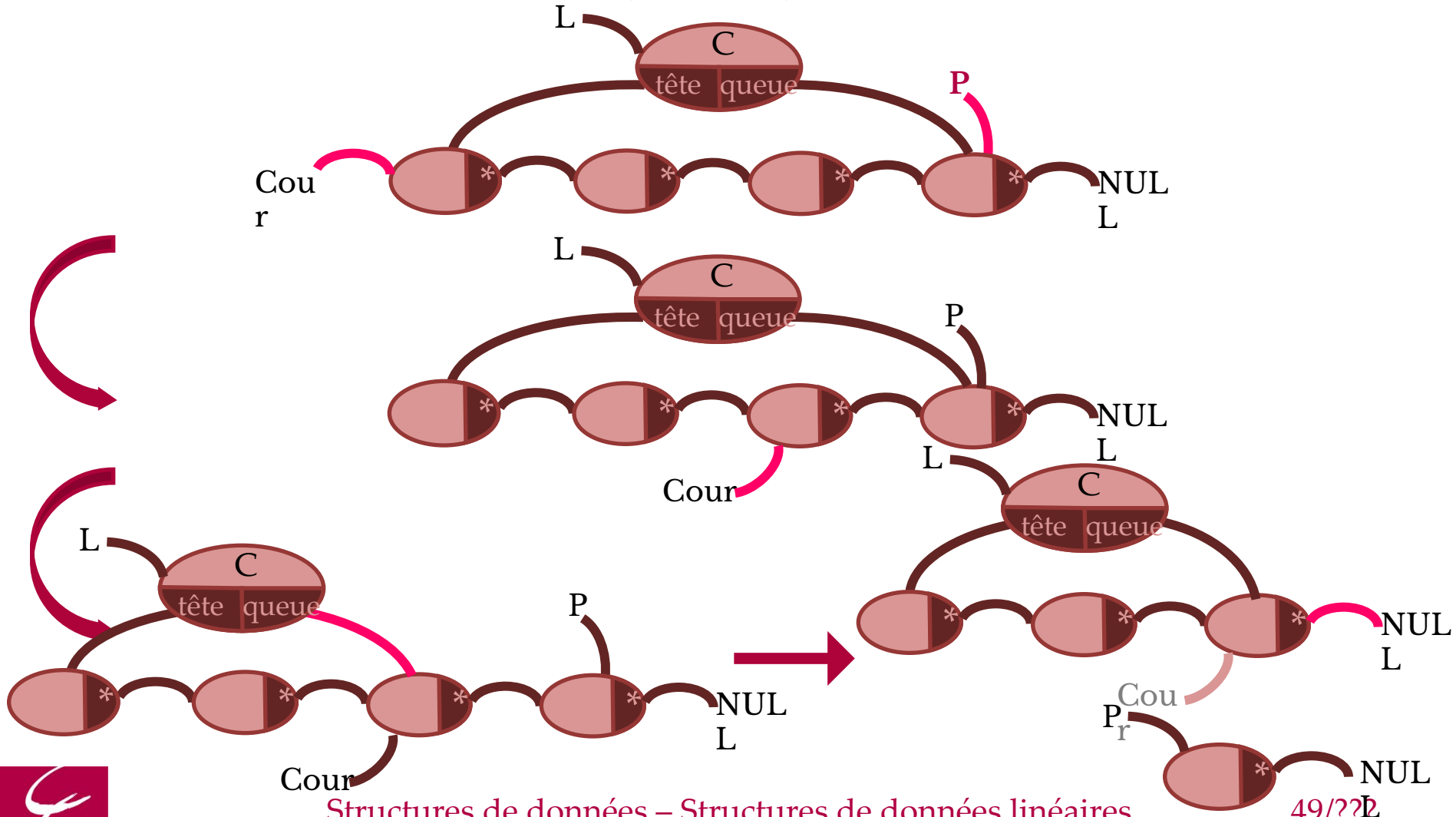
[Pointer P] = LLDeleteHead(LList L, elem cond) ;

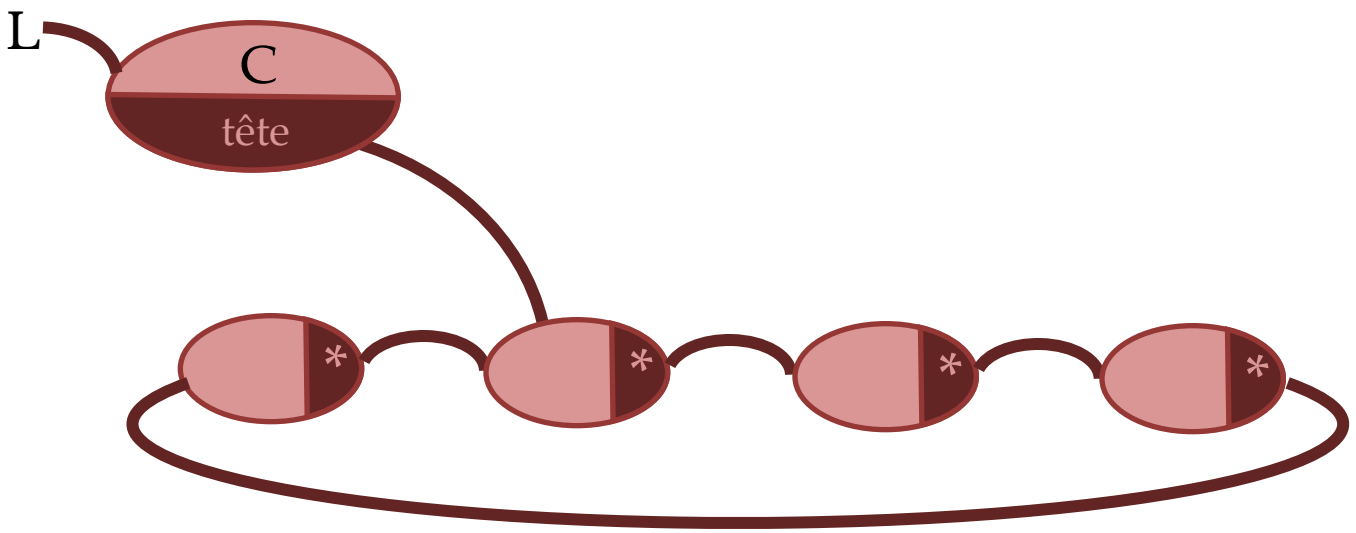


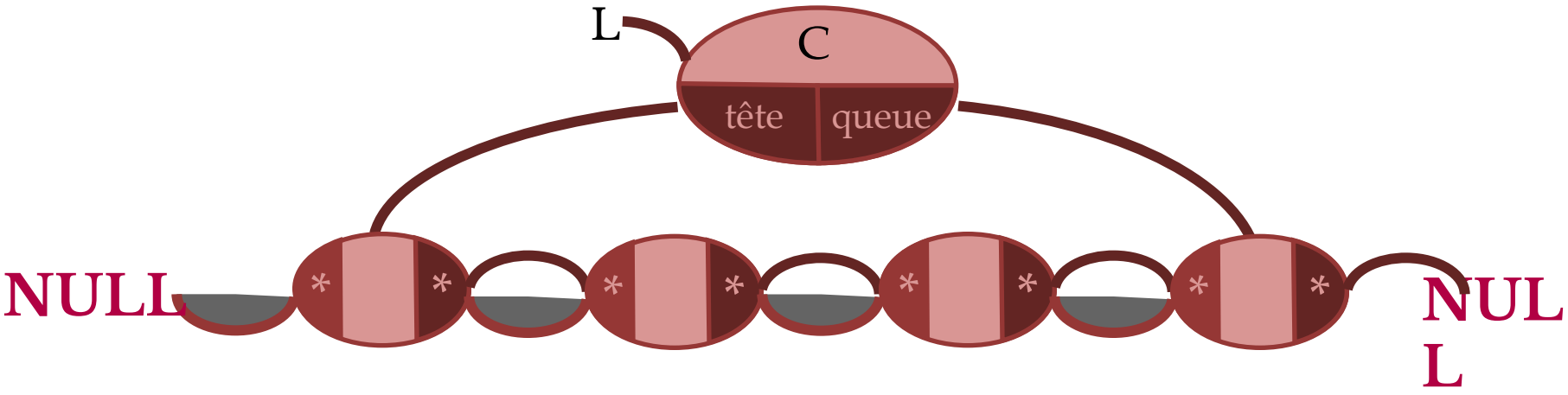
[Pointer P] = LLDelete(LList L, elem cond) ;



[Pointer P] = LLDeleteTail(LList L) ;




L
C
tête
*
*
*
*


L
C
tête
queue
NULL
NUL
L

Structures et pointeurs

```
typedef enum {false,true} boolean;

typedef struct Node* Pointer;

typedef struct Node{
   elem e;
   Pointer link;
} NODE;

typedef struct Chain* LList;

typedef struct Chain {
   Pointer head;
   Pointer tail;
} CHAIN;
```

Opérateurs

- constructeur/destructeur :
  - void LLCreate(LList L);
  - void LLEmpty(LList L);
- modifieurs
  - void LLInsertHead(LList L, Pointer P);
  - void LLInsertAfter(LList L, Pointer P, elem cond);
  - void LLInsertBefore(LList L, Pointer P, elem cond);
  - void LLInsertTail(LList L, Pointer P);
  - Pointer LLDeleteHead(LList L);
  - Pointer LLDelete(LList L, elem cond);
  - Pointer LLDeleteTail(LList L);
- sélecteurs
  - boolean isLListEmpty(LList L);
  - int LListeSize (LList L);
- itérateurs
  - boolean LListSearch(LList L, Pointer P, elem cond);
  - Pointer LListeNodeK(LList L, int k);
  - int LListOccurence(LList L, elem cond);

Structures et pointeurs

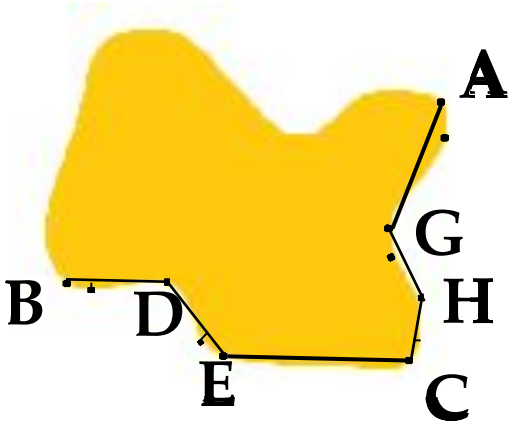
```
typedef enum {false,true} boolean;

typedef struct Node* Pointer;

typedef struct Node{
   elem e;
   Pointer link;
} NODE;

typedef struct Chain* LList;

typedef struct Chain {
   Pointer head;
   Pointer tail;
} CHAIN;
```

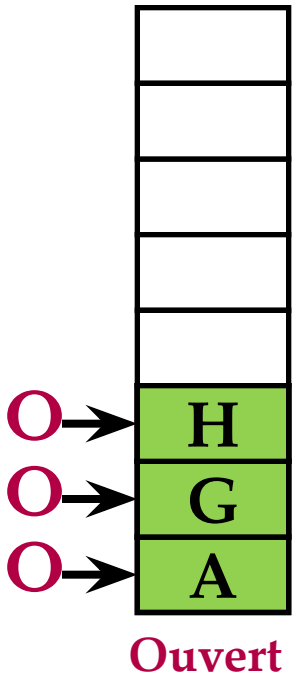
Opérateurs

- constructeur/destructeur :
  - void LLCreate(LList L);
  - void LLEmpty(LList L);
- modifieurs
  - void LLInsertHead(LList L, Pointer P);
  - void LLInsertAfter(LList L, Pointer P, elem cond);
  - void LLInsertBefore(LList L, Pointer P, elem cond);
  - void LLInsertTail(LList L, Pointer P);
  - Pointer LLDeleteHead(LList L);
  - Pointer LLDelete(LList L, elem cond);
  - Pointer LLDeleteTail(LList L);
- sélecteurs
  - boolean isLListEmpty(LList L);
  - int LListeSize (LList L);
- itérateurs
  - boolean LListSearch(LList L, Pointer P, elem cond);
  - Pointer LListeNodeK(LList L, int k);
  - int LListOccurence(LList L, elem cond);

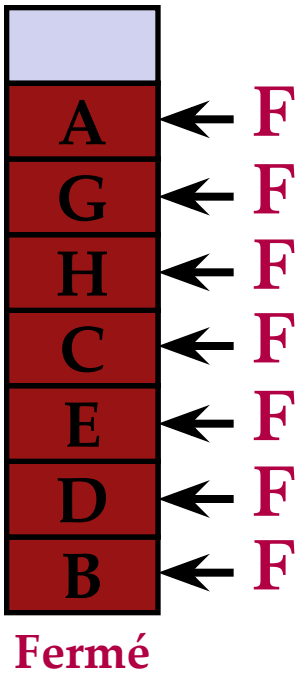
- Algorithme (best first)
  - initialisation des piles Ouvert et Fermé ;
  - **tant que** (Ouvert est non vide)
    - recherche du nouveau maillon M à partir des maillons en tête O(uvert) et F(ermé) ;
    - calcul de l'heuristique h(M) ;
    - **si** (h(M) > seuil) **alors** empiler M dans Ouvert ;
    - **sinon** dépiler O et l'empiler sur Fermé ;
    - **fin si**
  - **fin tant que**

2.seuil





Ouvert



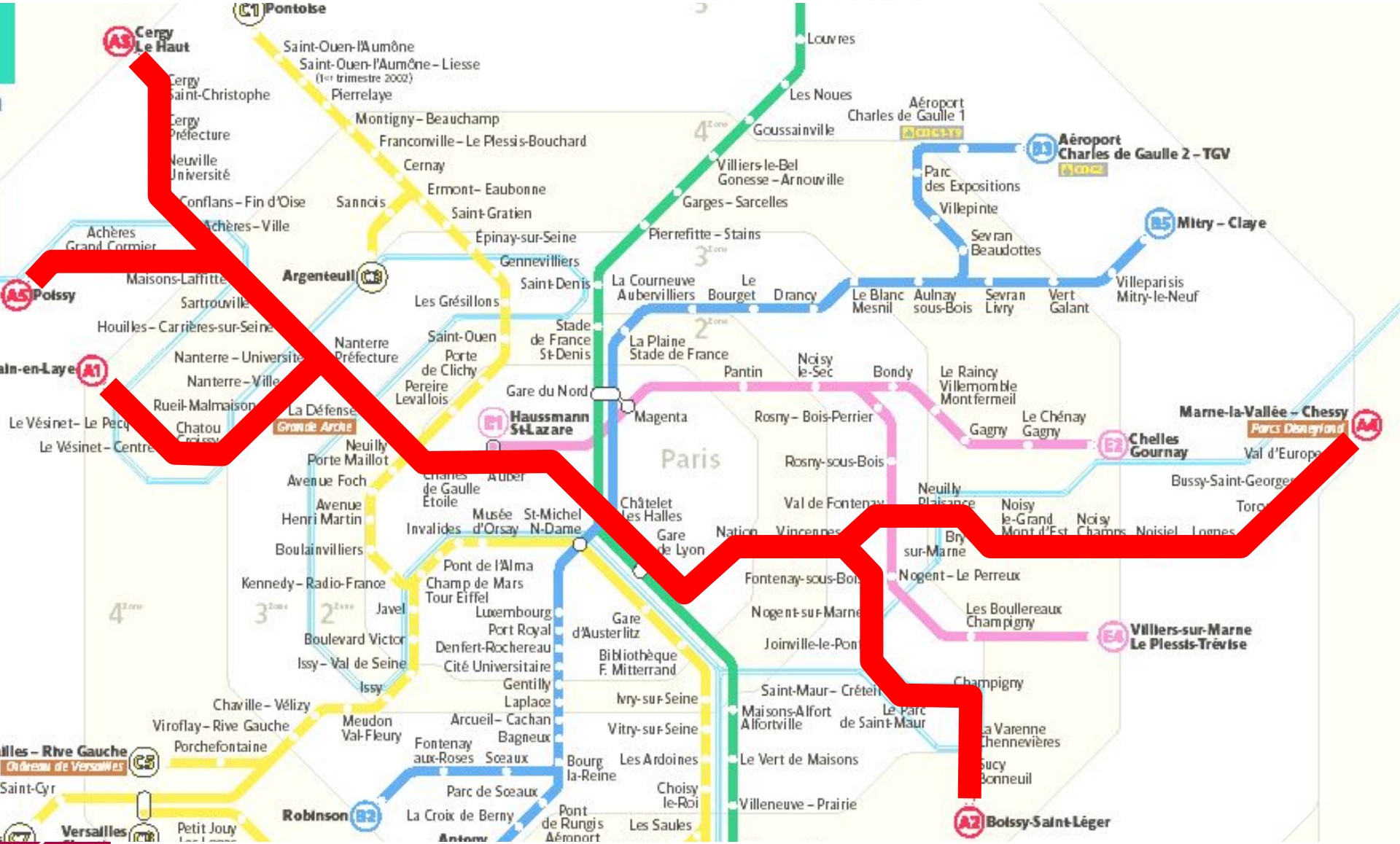
Fermé

ENSEA

~ Les files et piles permettent de modéliser nombre de systèmes séquentiels où seul l'ordre d'apparition des éléments importe.

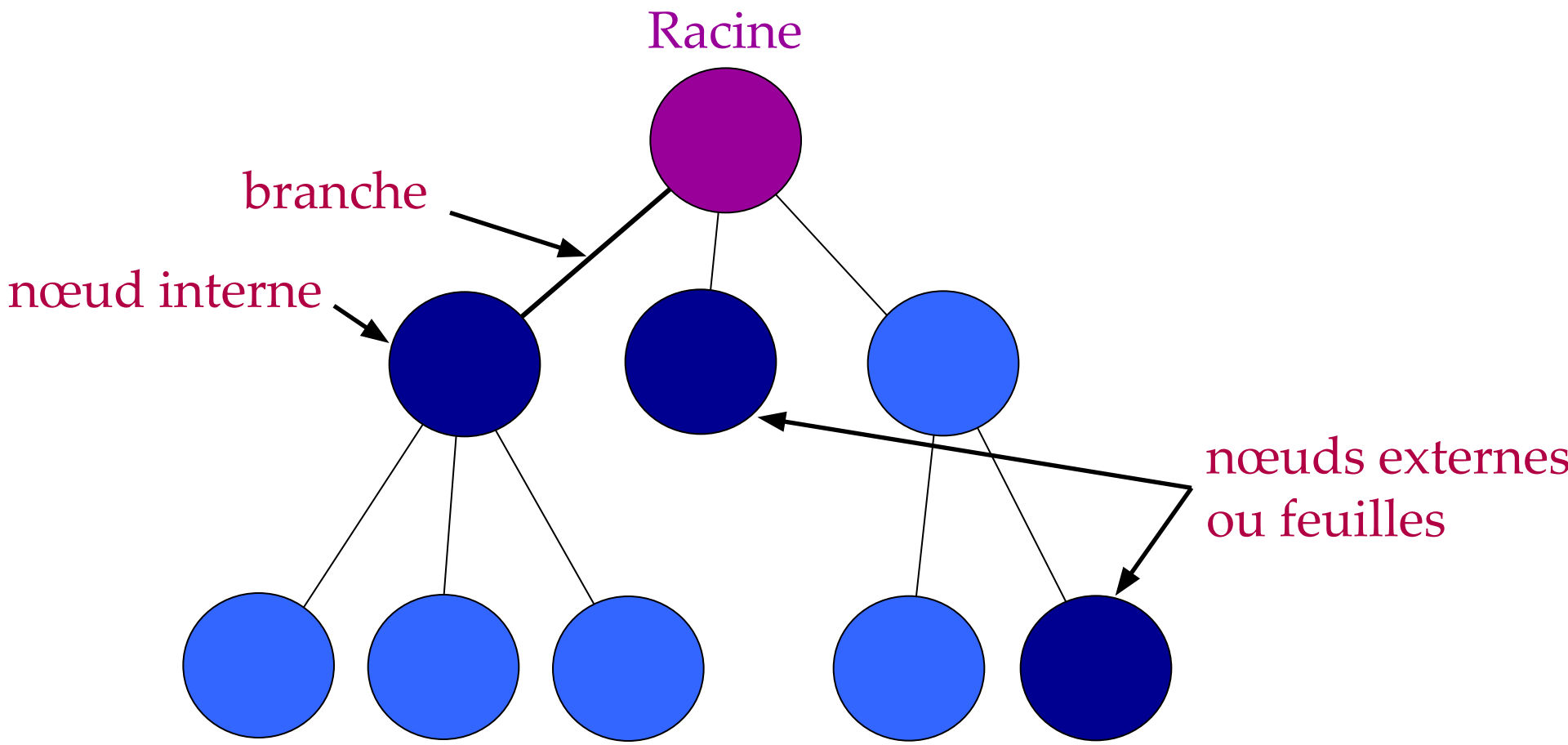
~ Lorsqu'une notion de priorité doit être prise en compte, les structures auto-référentielles s'imposent alors.

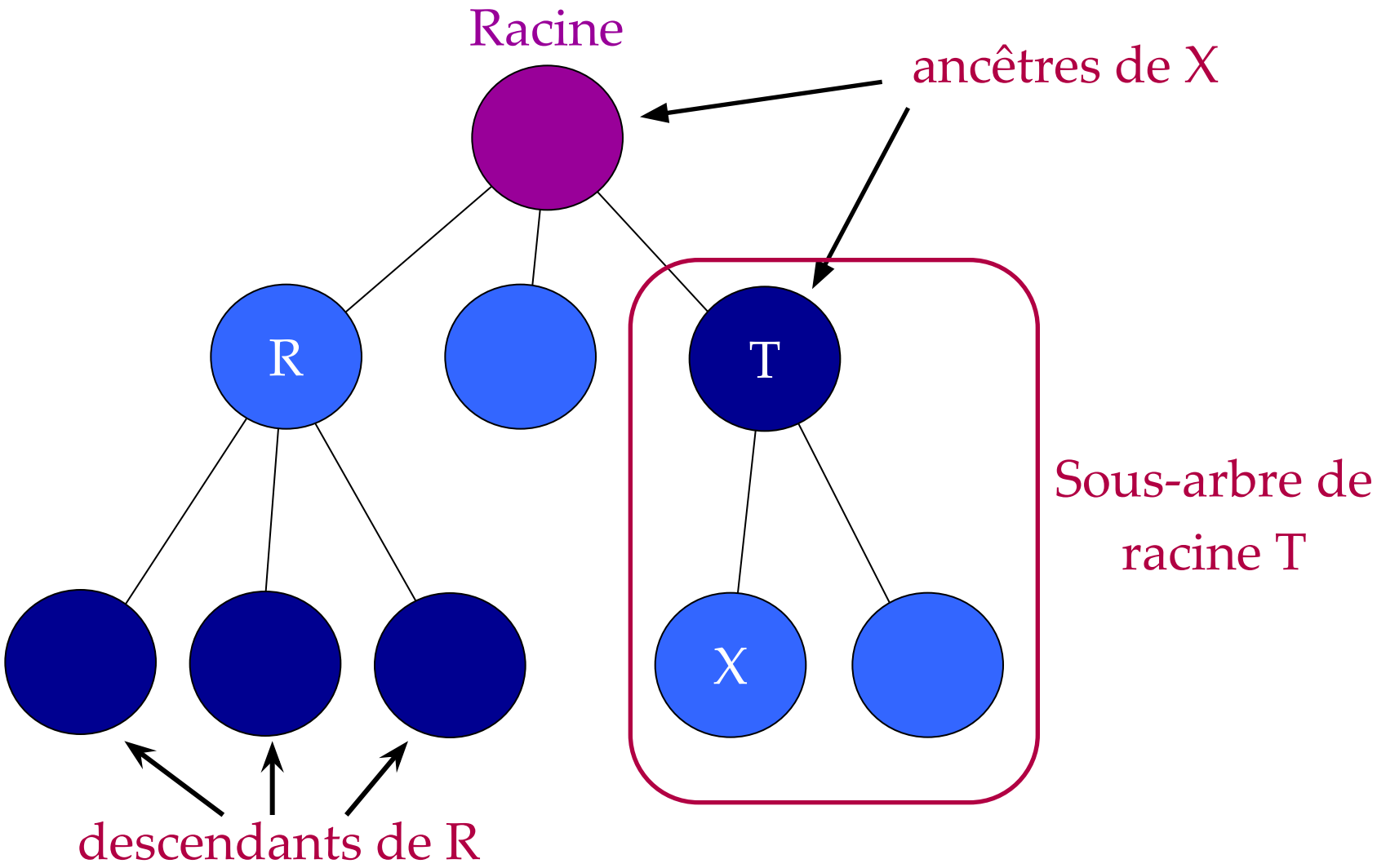
~ Les files et piles peuvent être implémentées à l'aide de tableaux, mais l'allocation dynamique, inhérente aux structures récursives, optimise les ressources mémoire au prix d'un coût en temps d'accès moins immédiat et d'un effort de conception pas si onéreux que cela.

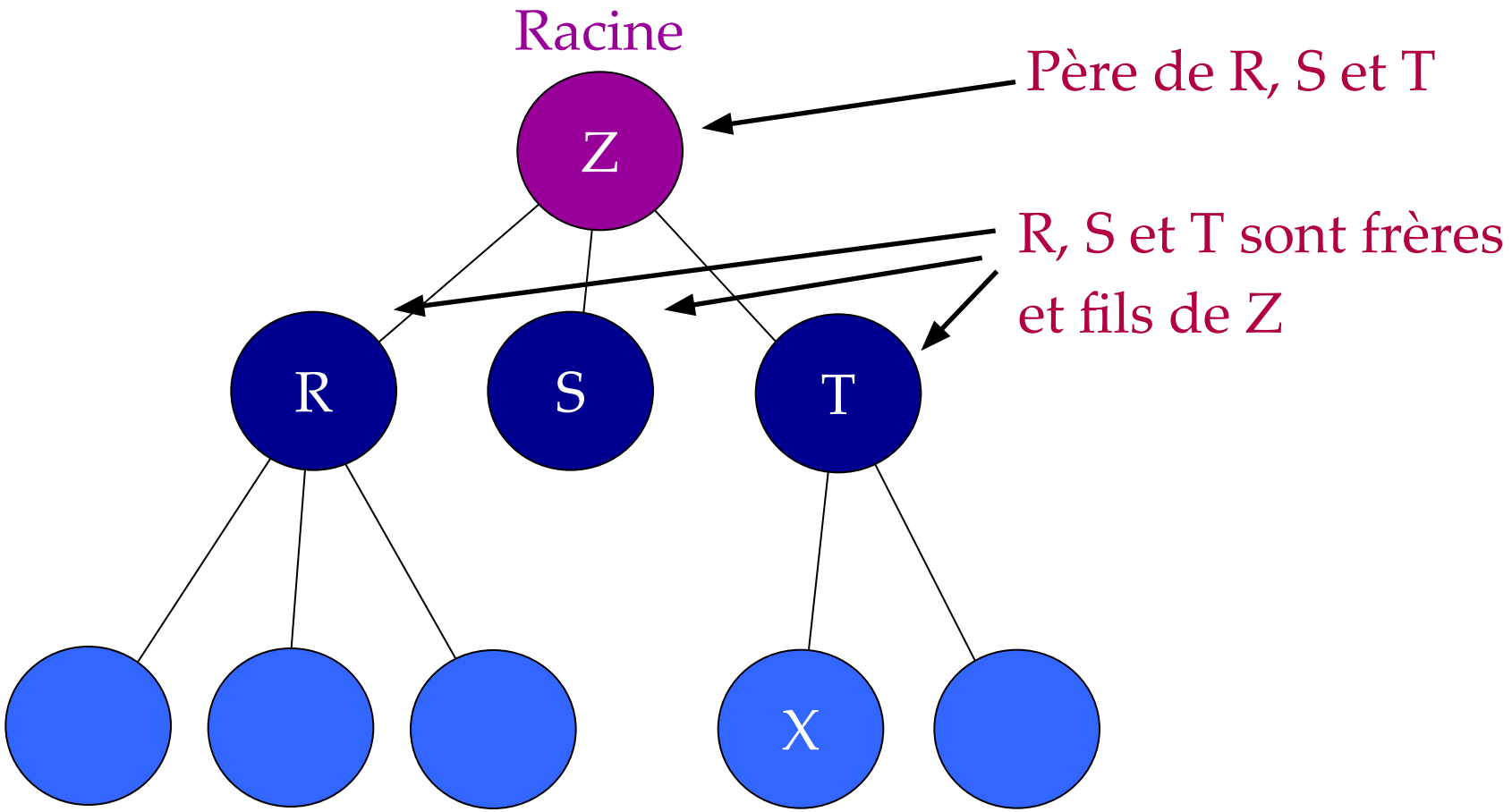

C1 Pontoise
A3 Cergy Le Haut
Saint-Ouen-l'Aumône
Saint-Ouen-l'Aumône – Liesse
(1er trimestre 2002)
Pierrelaye
Montigny – Beauchamp
Franconville – Le Plessis-Bouchard
Cernay
Ermont – Eaubonne
Saint-Gratien
Sannois
Conflans – Fin d'Oise
Achères – Ville
Achères Grand Cormier
Maisons-Laffitte
Sartrouville
Houilles – Carrières-sur-Seine
A5 Poissy
Argenteuil C3
Les Grésillons
Nanterre – Université
Nanterre Préfecture
Nanterre – Ville
Rueil-Malmaison
Chatou
Le Vésinet – Le Pecq
Le Vésinet – Centre
A1
Épinay-sur-Seine
Gennevilliers
Saint-Denis
Saint-Ouen
Porte de Clichy
Pereire Levallois
La Défense
Grande Arche
Neuilly Porte Maillot
Avenue Foch
Avenue Henri Martin
Boulainvilliers
Kennedy – Radio-France
Pont de l'Alma
Champ de Mars Tour Eiffel
Javel
Boulevard Victor
Issy – Val de Seine
Issy
Chaville – Vélizy
Viroflay – Rive Gauche
Porchefontaine
Meudon Val-Fleury
Arcueil – Cachan
Bagneux
Fontenay aux-Roses
Sceaux
Parc de Sceaux
Robinson B2
La Croix de Berny
Pont de Rungis Aéroport
Les Saules
Antony
Bourg la-Reine
Les Ardoines
Choisy le-Roi
Villeneuve – Prairie
Saint-Cyr
Versailles
Petit Jouy
Louvres
Les Noues
Goussainville
Aéroport Charles de Gaulle 1
Aéroport Charles de Gaulle 2 – TGV
Parc des Expositions
Villepinte
Mitry – Claye
Villeparisis Mitry-le-Neuf
Sevran Beaudottes
Vert Galant
Sevran Livry
Aulnay sous-Bois
Le Blanc Mesnil
Villiers-le-Bel Gonesse – Arnouville
Garges – Sarcelles
Pierrefitte – Stains
La Courneuve Aubervilliers
Le Bourget
Drancy
La Plaine Stade de France
Stade de France St-Denis
Gare du Nord
Haussmann St-Lazare
Magenta
Pantin
Noisy le-Sec
Bondy
Le Raincy Villemomble Montfermeil
Rosny – Bois-Perrier
Rosny-sous-Bois
Le Chénay Gagny
Gagny
Chelles Gournay
Marne-la-Vallée – Chessy
Parcs Disneyland
Val d'Europe
Bussy-Saint-Georges
Noisy le-Grand Mont d'Est
Noisy Champs
Noisiel
Lognes
Neuilly Plaisance
Bry sur-Marne
Nogent – Le Perreux
Les Boullereaux Champigny
Villiers-sur-Marne Le Plessis-Trévise
Champigny
La Varenne Chennevières
Bonneuil
Boissy-Saint-Léger
Paris
Châtelet Les Halles
Gare de Lyon
Nation
Vincennes
Val de Fontenay
Fontenay-sous-Bois
Nogent-sur-Marne
Joinville-le-Pont
Saint-Maur – Créteil
Maisons-Alfort Alfortville
Le Parc de Saint-Maur
Le Vert de Maisons
Invalides
Musée d'Orsay
St-Michel N-Dame
Luxembourg
Port Royal
Denfert-Rochereau
Cité Universitaire
Gentilly
Laplace
Gare d'Austerlitz
Bibliothèque F. Mitterrand
Ivry-sur-Seine
Vitry-sur-Seine
Charles de Gaulle Etoile
Auber

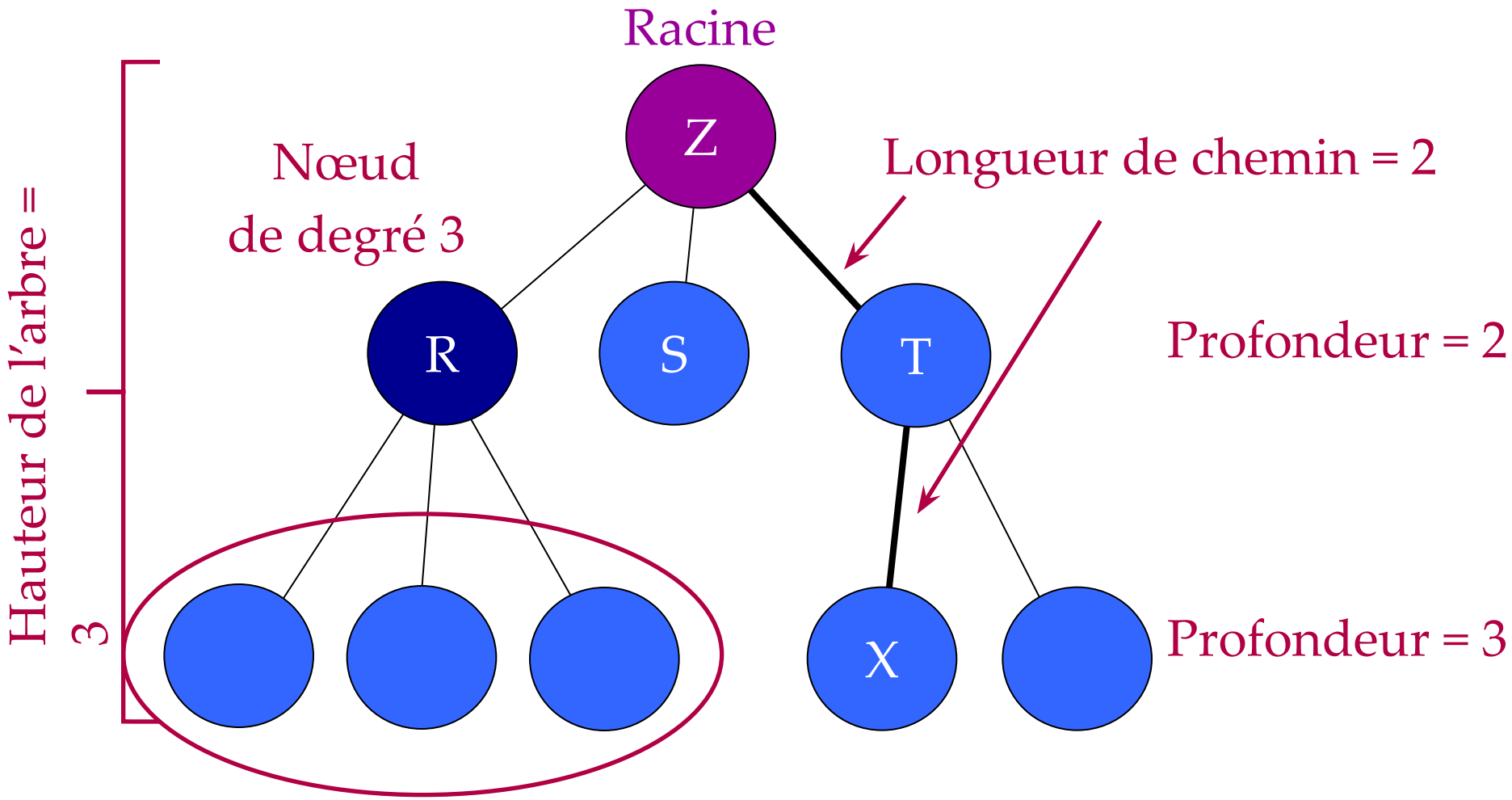
ENSEA

- Pourquoi faire?
  - Réalisation d'algorithmes de tri
  - Implémentation de files de priorités
  - Implémentation de la table des symboles d'un compilateur
  - Résolution de problèmes d'indexation


Racine
branche
nœud interne
nœuds externes
ou feuilles

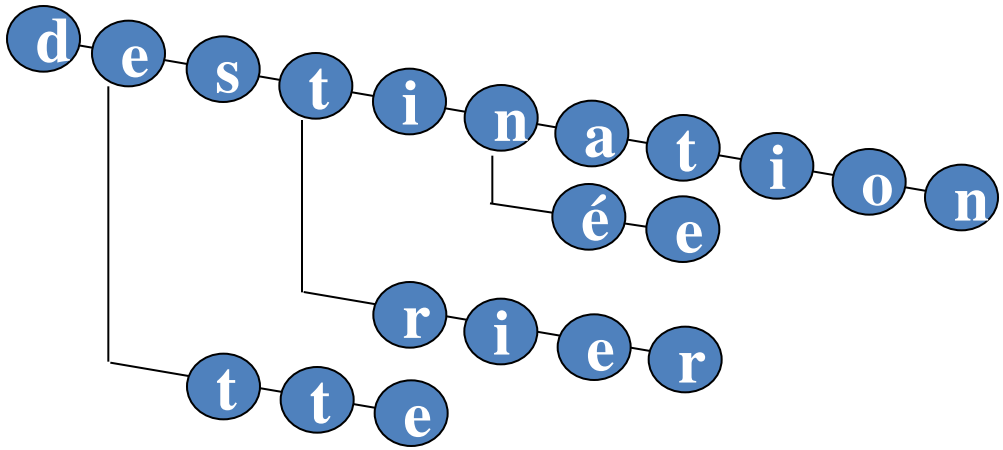

Racine
ancêtres de X
R
T
X
Sous-arbre de racine T
descendants de R

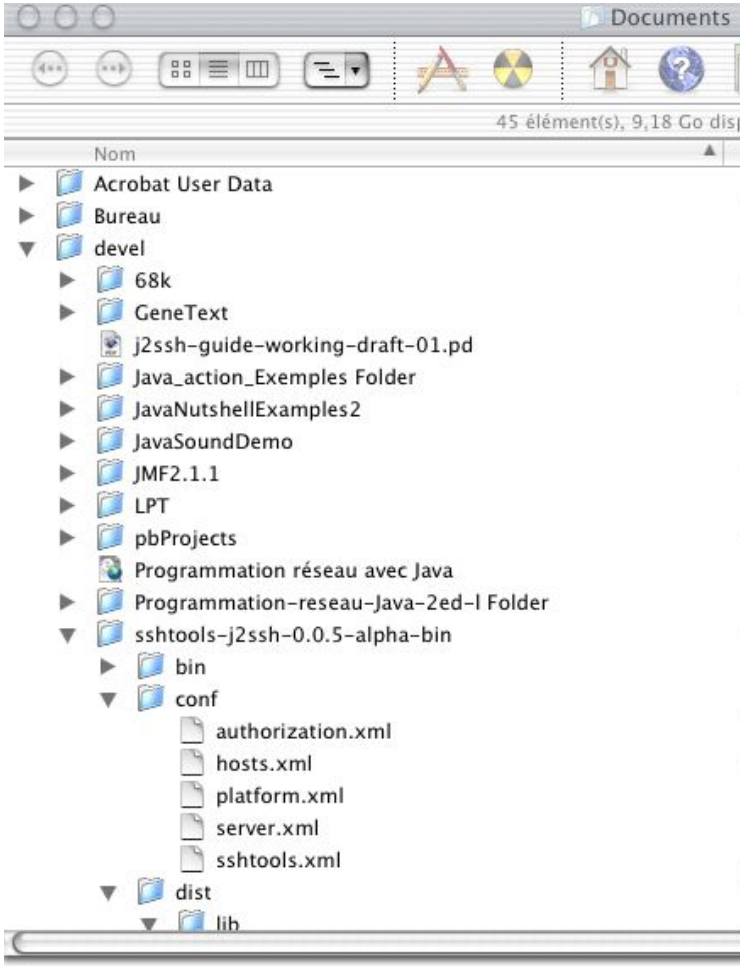

Racine
Père de R, S et T
Z
R, S et T sont frères
et fils de Z
R
S
T
X


Racine
Z
Nœud
de degré 3
Longueur de chemin = 2
R
S
T
Profondeur = 2
Hauteur de l'arbre =
3
X
Profondeur = 3

~ Objectifs

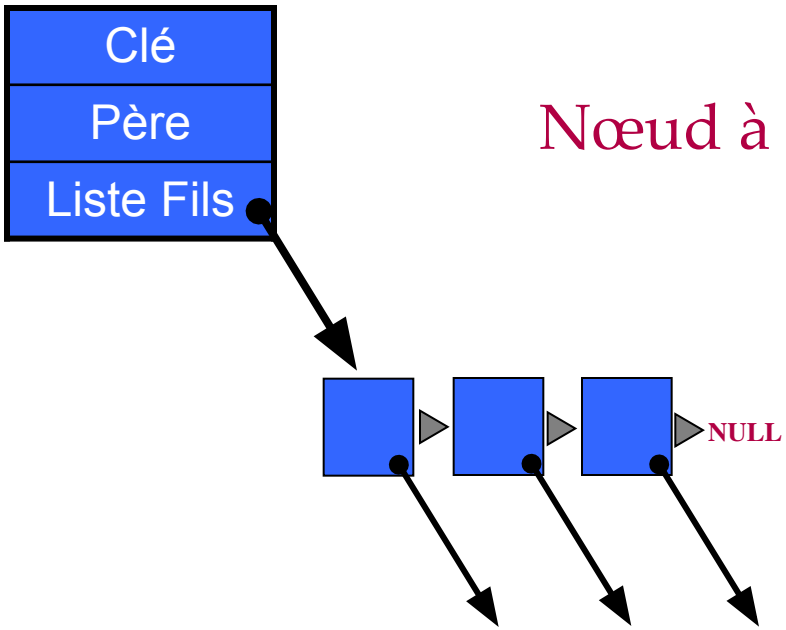
- économiser les ressources
- accélérer les recherches





- Tableaux de fils potentiels
  - Possibilité d'associer un tableau de fils potentiels à chacun des nœuds (exemple arbre 6-aires).

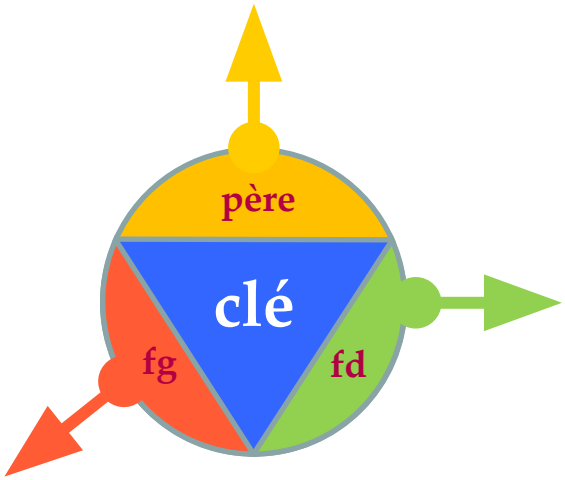
- Liste chaînée de fils potentiels
  - Possibilité d'associer une liste de fils

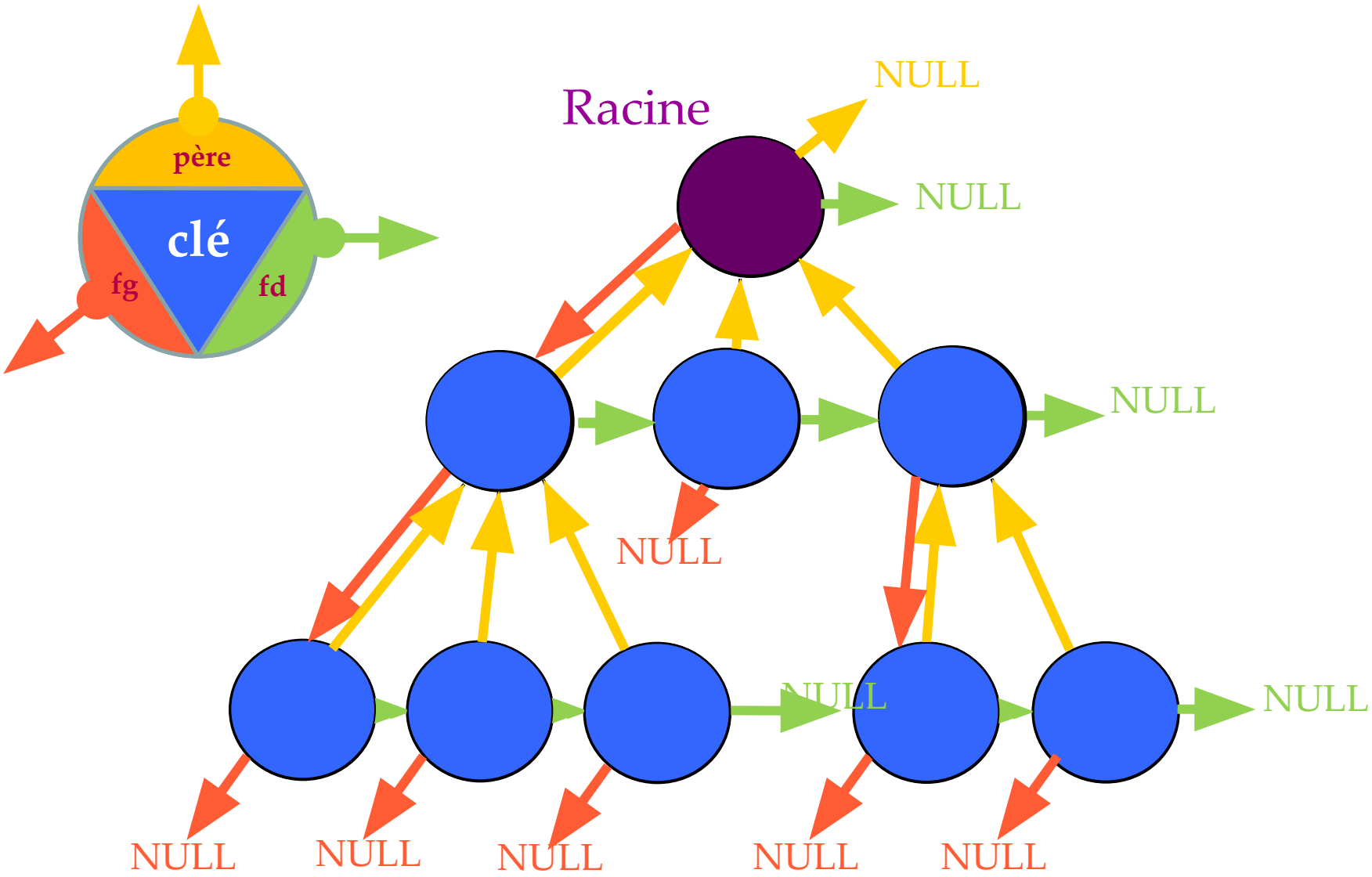


Nœud à 3 fils

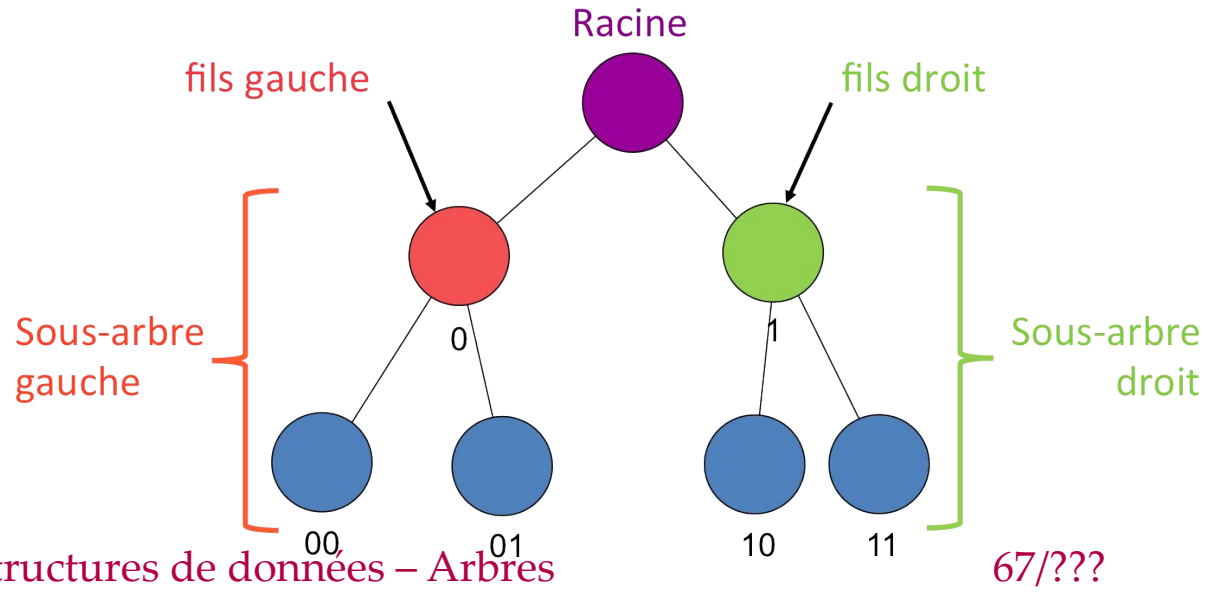
~ Réprésentation fils (aîné) gauche / frère droit

 Possibilité de représenter tout type d'arbre

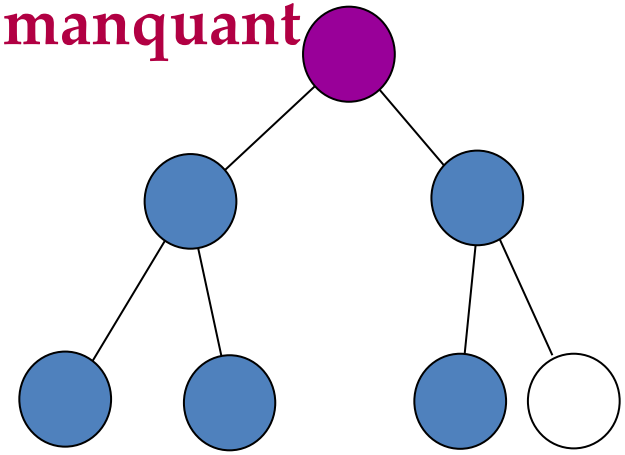



père
clé
fg
fd
Racine
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL

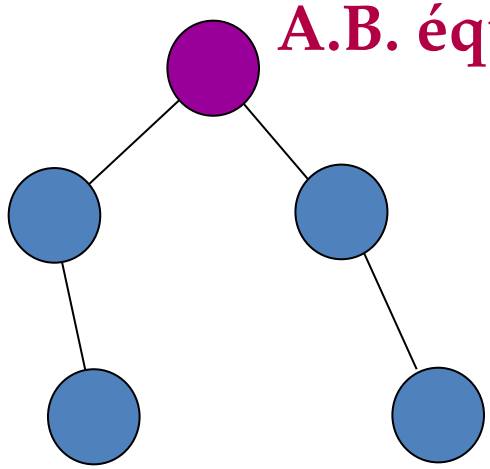
- Un arbre binaire est un arbre de degré 2. Il est formé :
  - d'aucun nœud => arbre vide
  - de 3 ensembles de nœuds :
    - un nœud racine,
    - un sous arbre gauche binaire aussi,
    - un sous arbre droit binaire aussi.



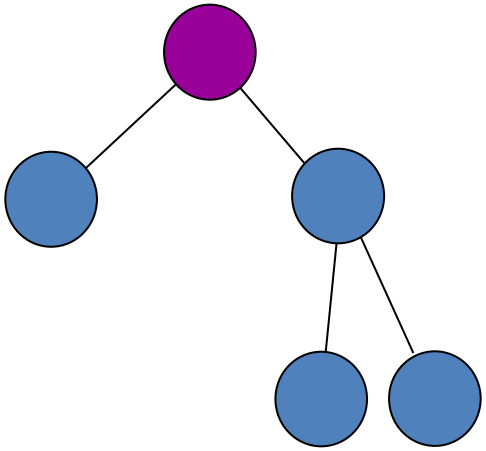
Fils manquant



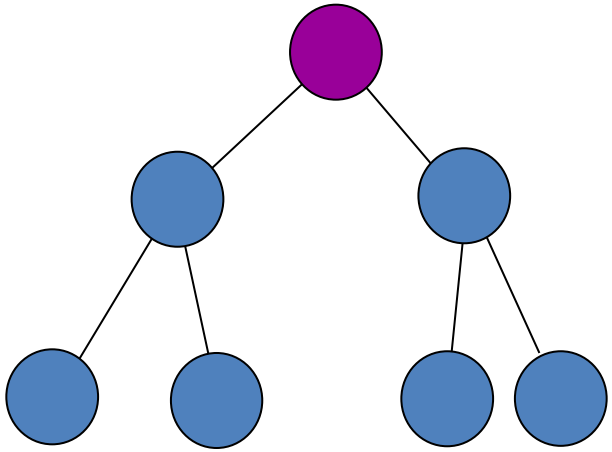
A.B. équilibré



A.B. entier



A.B. complet

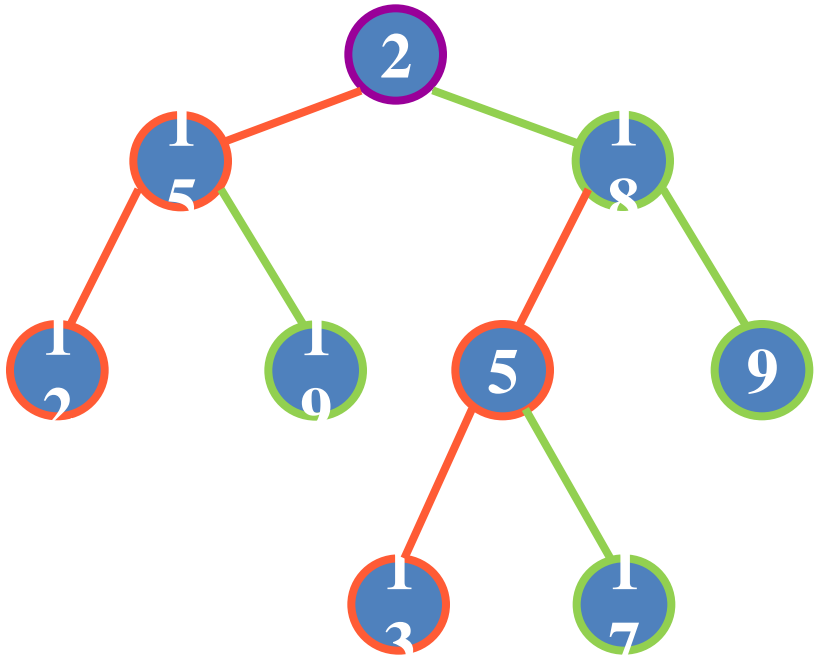


- Motivation
  - Retrouver ou examiner les nœuds
  - Définir un ordre sur les nœuds
  - Examiner les parcours possibles
    - en largeur d'abord
    - en profondeur d'abord
      - Préfixe
      - Infixe
      - Postfixe

~ Parcours en largeur d'abord

- niveau par niveau
  - de la racine vers les feuilles,
  - de la gauche vers la droite.

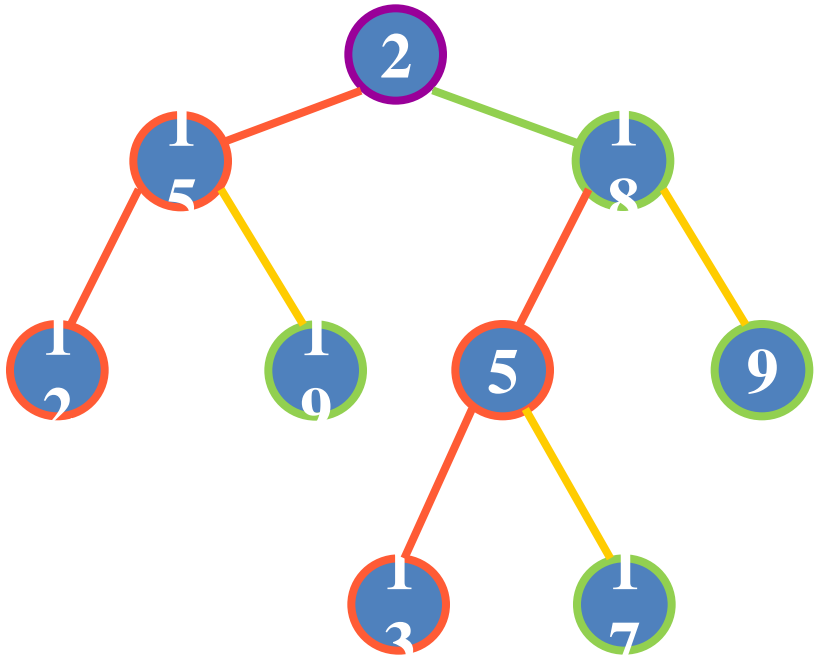
- séquence :



~ Parcours en profondeur (préfixe)

- Algorithme du parcours : Nœud, Gauche, Droite
  - ✷ Si l'arbre n'est pas vide,
    - ✶ Traiter la racine
    - ✶ Parcourir en PP le sous-arbre gauche
    - ✶ Parcourir en PP le sous-arbre droit

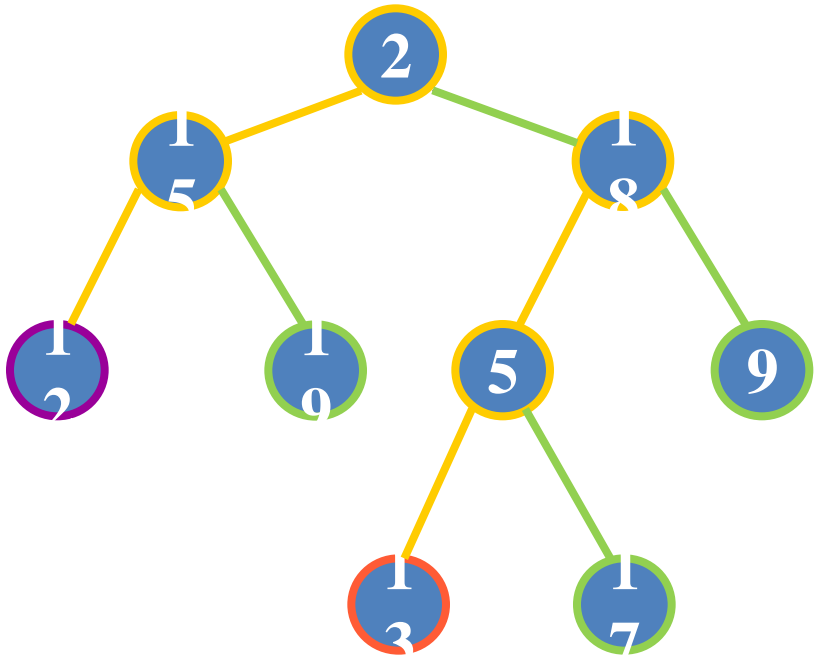
- Séquence



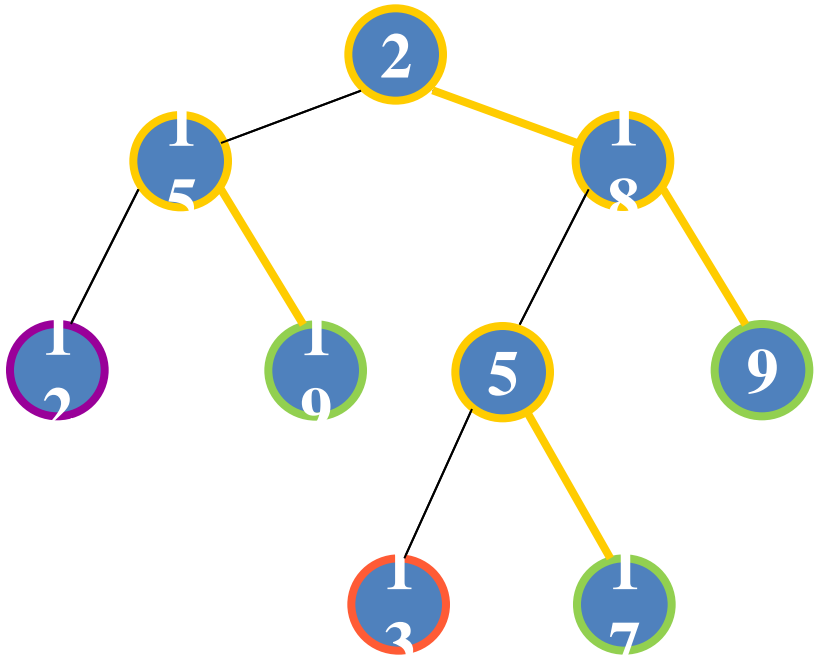
~ Parcours en profondeur (infixe)

- Algorithme du parcours : Gauche, Nœud, Droite
  - Si l'arbre n'est pas vide,
    - Parcourir en PP le sous-arbre gauche
    - Traiter la racine
    - Parcourir en PP le sous-arbre droit

- Séquence



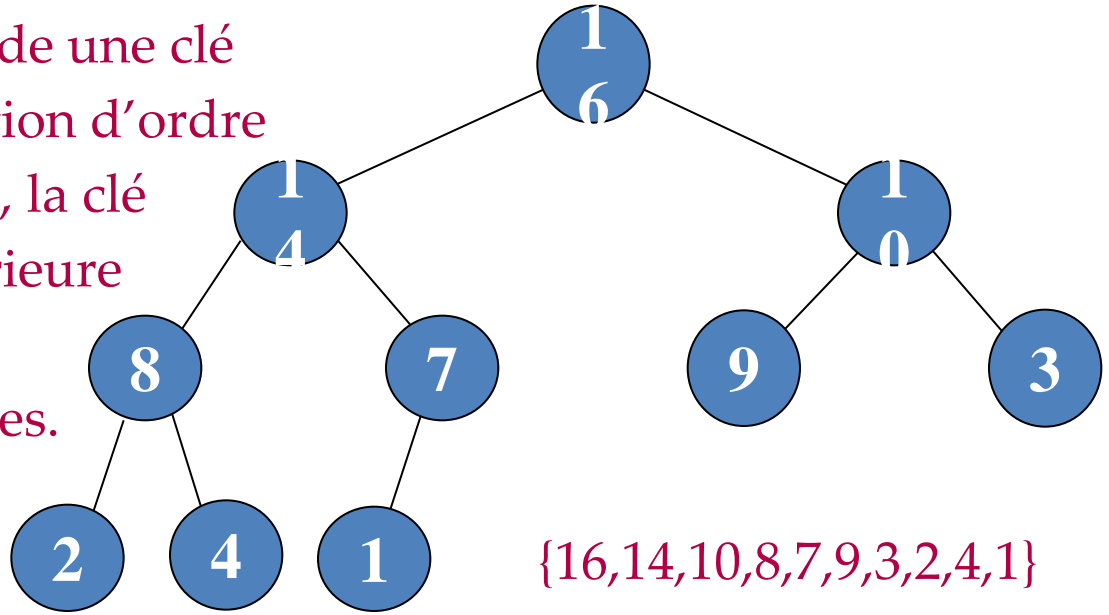
- Parcours en profondeur (postfixe)
  - Algorithme du parcours : Gauche, Droite, Noeud
    - Si l'arbre n'est pas vide,
      - Parcourir en PP le sous-arbre gauche
      - Parcourir en PP le sous-arbre droit
      - Traiter la racine
  - séquence :



~ Définition et propriétés d'un tas

- Un arbre binaire équilibré peut être compacté en un tableau, appelé tas. Le nombre de cases utilisées sera optimal si l'arbre binaire est complet. Le tas est partiellement ordonné.

- Propriétés:
  - Chaque nœud possède une clé permettant une relation d'ordre
  - Pour tout sous-arbre, la clé de la racine est supérieure ou égale à toutes les clés de ses sous-arbres.

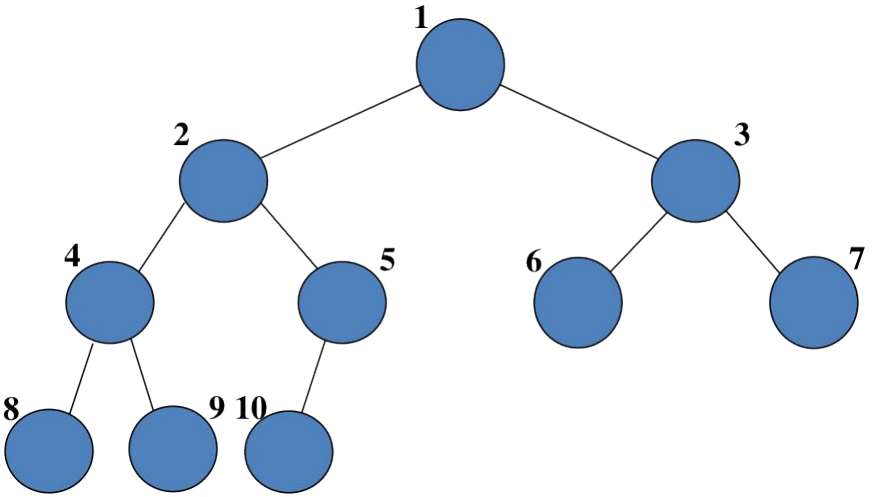


Règle de construction :
C(père) ≥ max{C(fg),C(fd)}

{16,14,10,8,7,9,3,2,4,1}

~ Construction d'un tas : entassement

- Insertion d'un nœud à droite de la dernière feuille.
- Organisation des indices
  - racine : k = 1
  - père(k) = k/2
  - fg(k) = 2.k
  - fd(k) = 2.k + 1



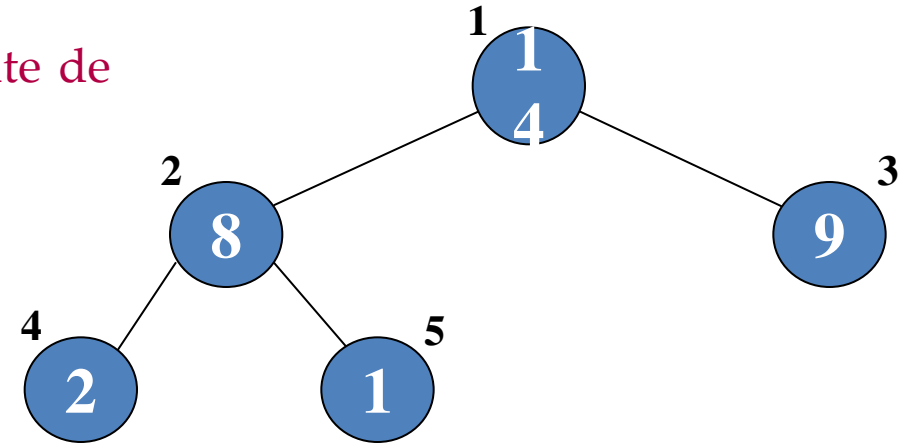
k	1	2	3	4	5	6	7	8	9	10
C[k]										

- Règle de tas : C[père(k)] ≥ C[k]

~ Construction d'un tas : exemple

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Insertion d'un nœud à droite de la dernière feuille.



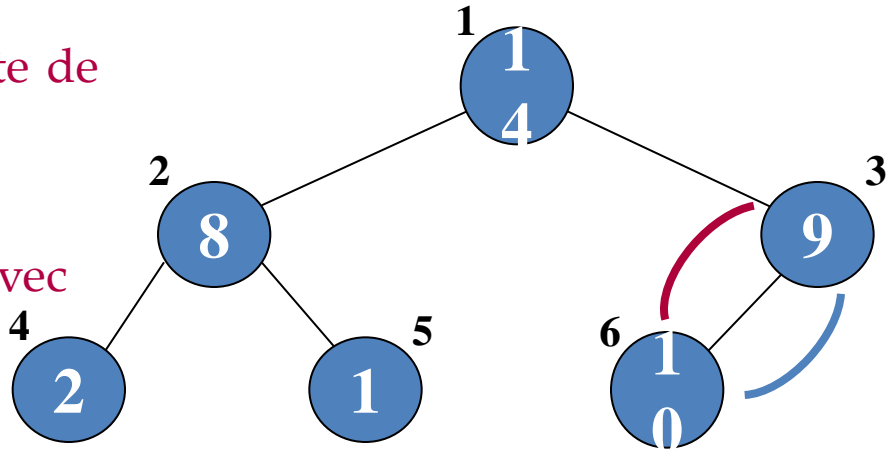
k	1	2	3	4	5	6	7	8	9	10
C[k]	14	8	9	2	1					

~ Construction d'un tas : exemple

séquence = {14, 8, 9, 2, 1, **10**, 3, 16, 4, 7}

Insertion d'un nœud à droite de la dernière feuille.

Un nœud qui viole la règle d'entassement est permuté avec son père.



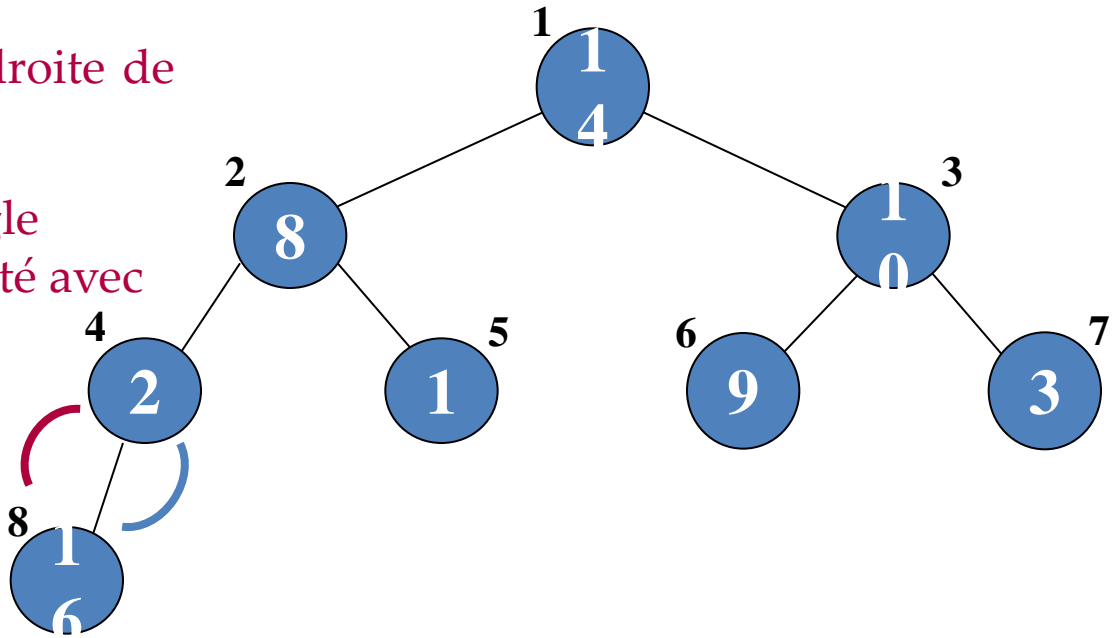
k	1	2	3	4	5	6	7	8	9	10
C[k]	**14**	**8**	**9**	**2**	**1**	**10**				

~ Construction d'un tas : exemple

séquence = {14, 8, 9, 2, 1, 10, 3, **16**, 4, 7}

Insertion d'un nœud à droite de la dernière feuille.

Un nœud qui viole la règle d'entassement est permuté avec son père.



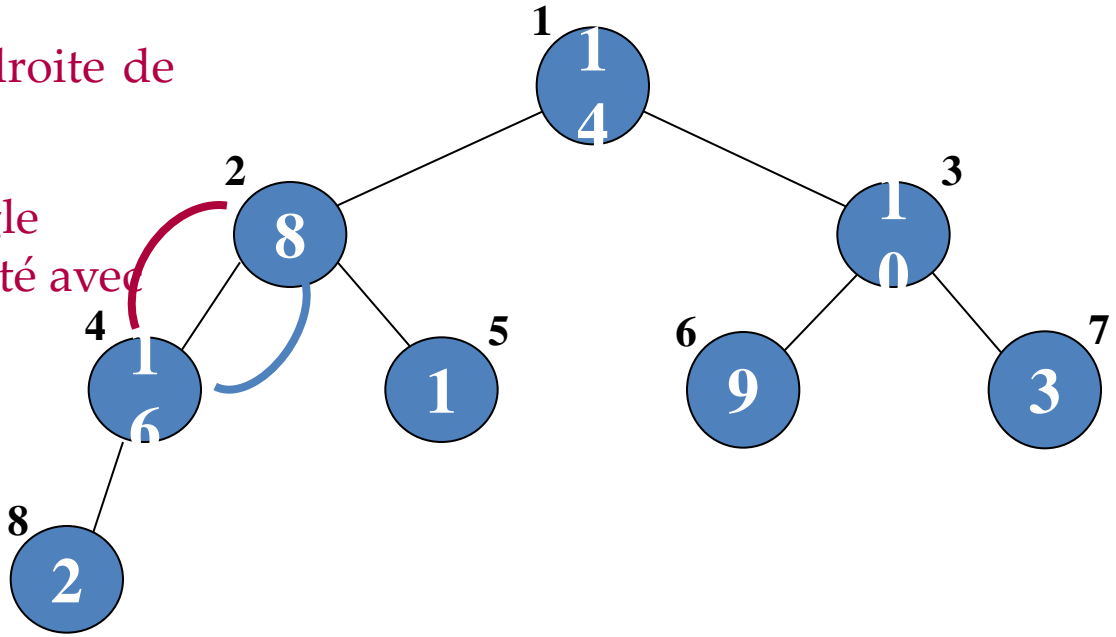
k	1	2	3	4	5	6	7	8	9	10
C[k]	**14**	**8**	**10**	**2**	**1**	**9**	**3**	**16**		

~ Construction d'un tas : exemple

séquence = {14, 8, 9, 2, 1, 10, 3, **16**, 4, 7}

Insertion d'un nœud à droite de la dernière feuille.

Un nœud qui viole la règle d'entassement est permuté avec son père.



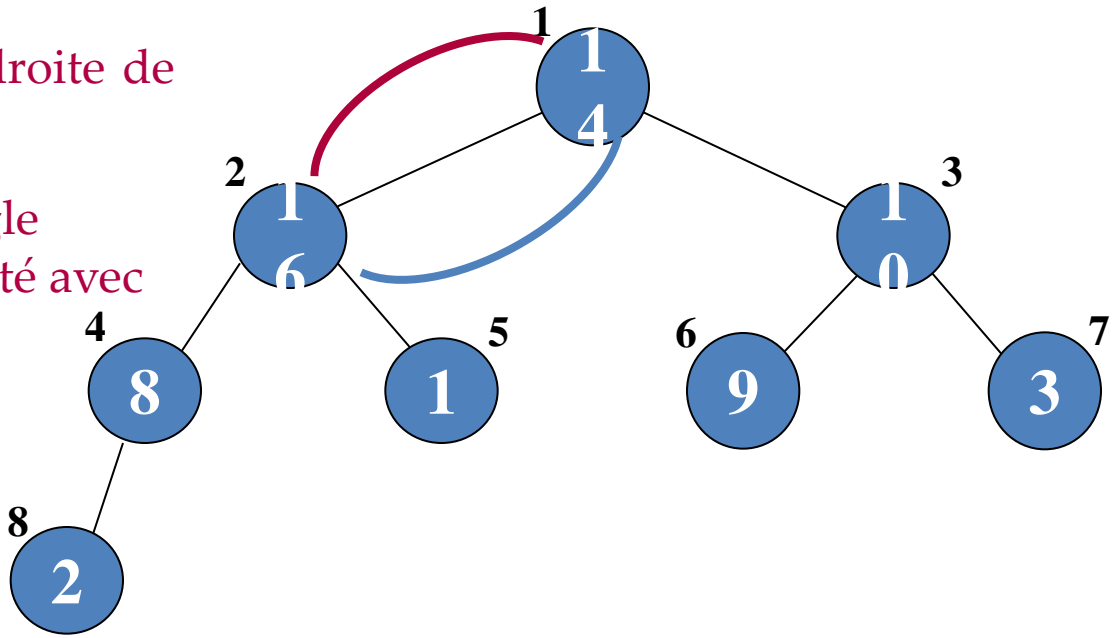
k	1	2	3	4	5	6	7	8	9	10
C[k]	14	8	10	16	1	9	3	2		

~ Construction d'un tas : exemple

séquence = {14, 8, 9, 2, 1, 10, 3, **16**, 4, 7}

Insertion d'un nœud à droite de la dernière feuille.

Un nœud qui viole la règle d'entassement est permuté avec son père.



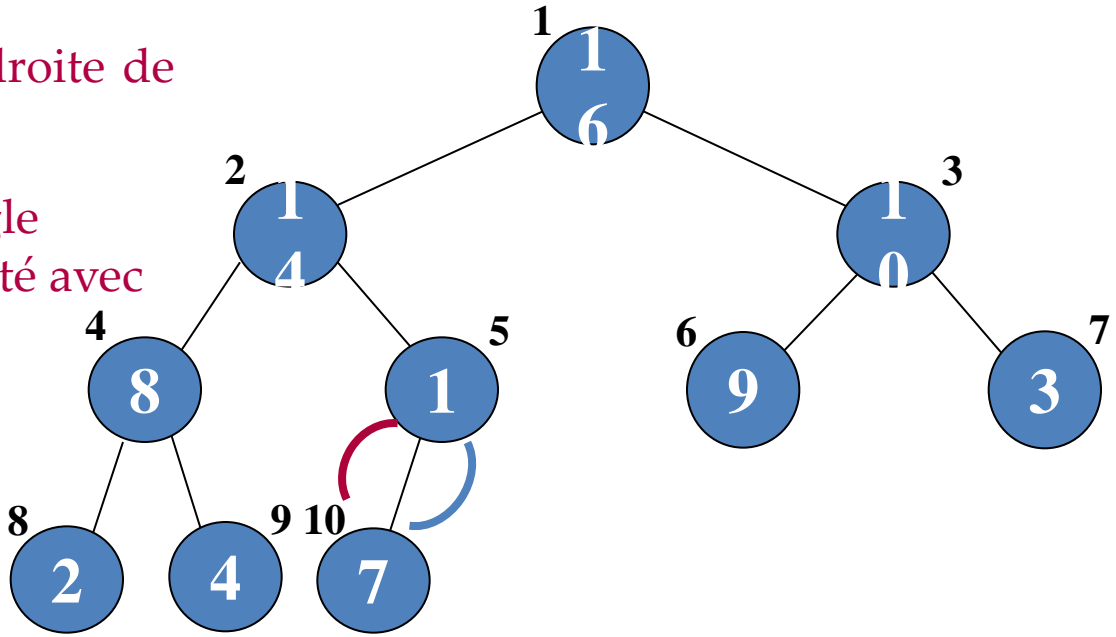
k	1	2	3	4	5	6	7	8	9	10
C[k]	14	16	10	8	1	9	3	2		

~ Construction d'un tas : exemple

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Insertion d'un nœud à droite de la dernière feuille.

Un nœud qui viole la règle d'entassement est permuté avec son père.

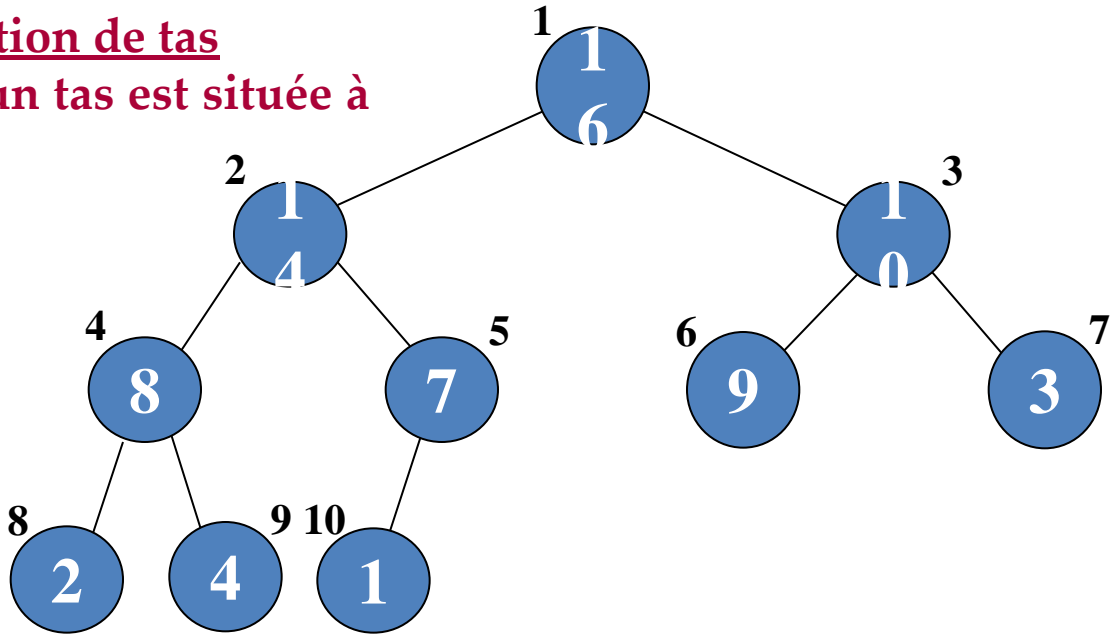


k	1	2	3	4	5	6	7	8	9	10
C[k]	16	14	10	8	1	9	3	2	4	7

~ Construction d'un tas : exemple

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Conséquence de la condition de tas
La plus grande valeur d'un tas est située à la racine



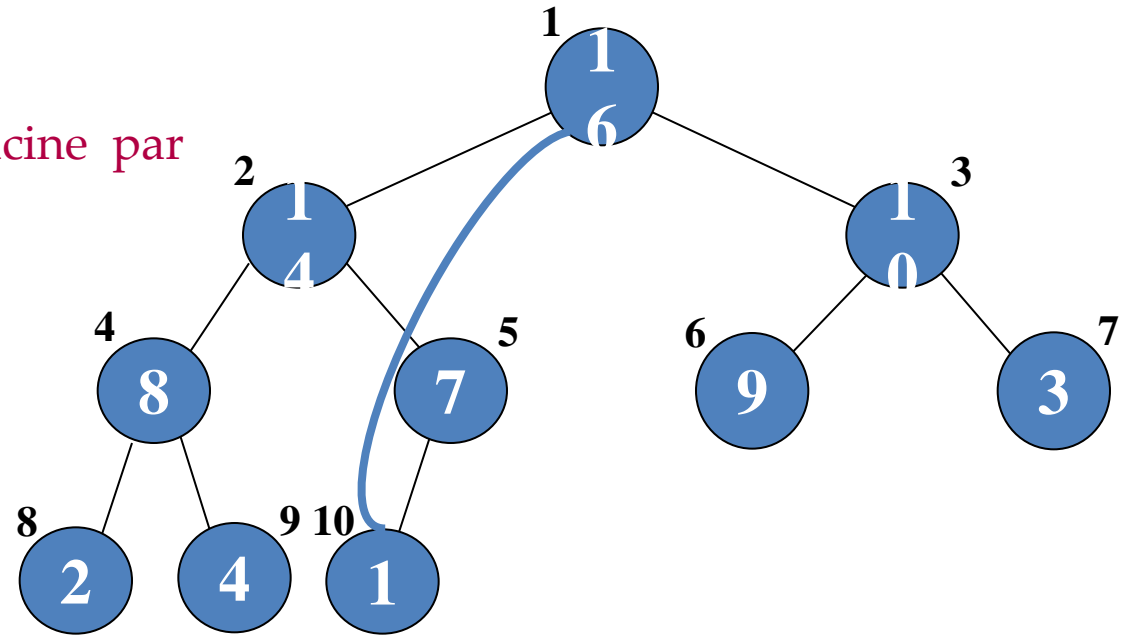
k	1	2	3	4	5	6	7	8	9	10
C[k]	**16**	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**

~ Tri d'un tas ordonné

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Sauvegarde de la racine.

Remplacement de la racine par la dernière feuille.

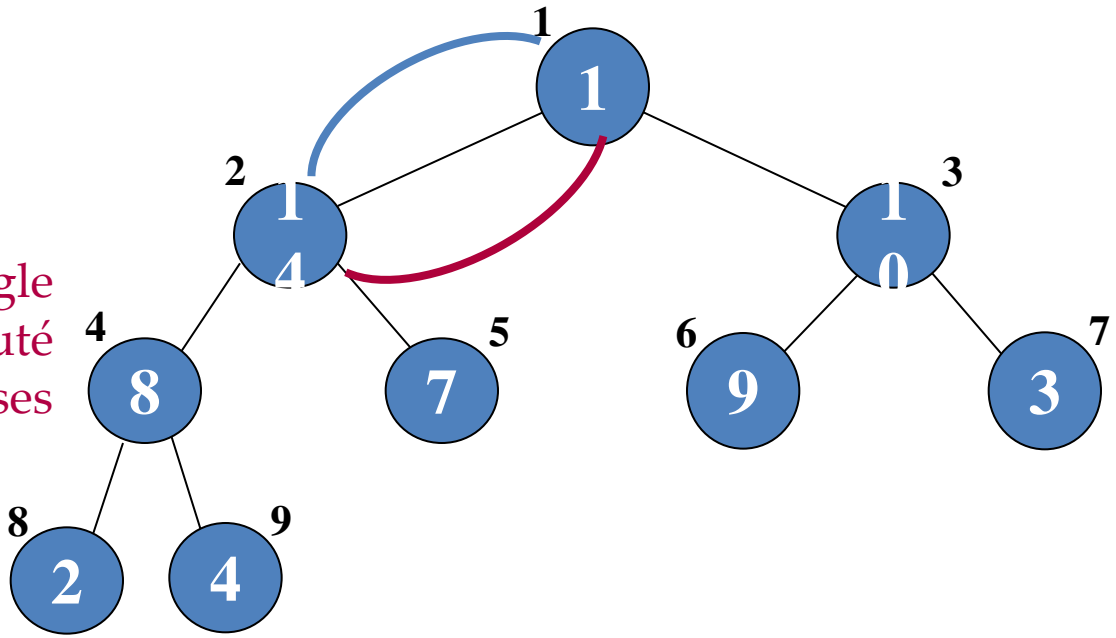


k	1	2	3	4	5	6	7	8	9	10
C[k]	1	14	10	8	7	9	3	2	4	16

~ Tri d'un tas ordonné

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Un nœud qui viole la règle d'entassement est permuté avec le plus grand de ses fils.

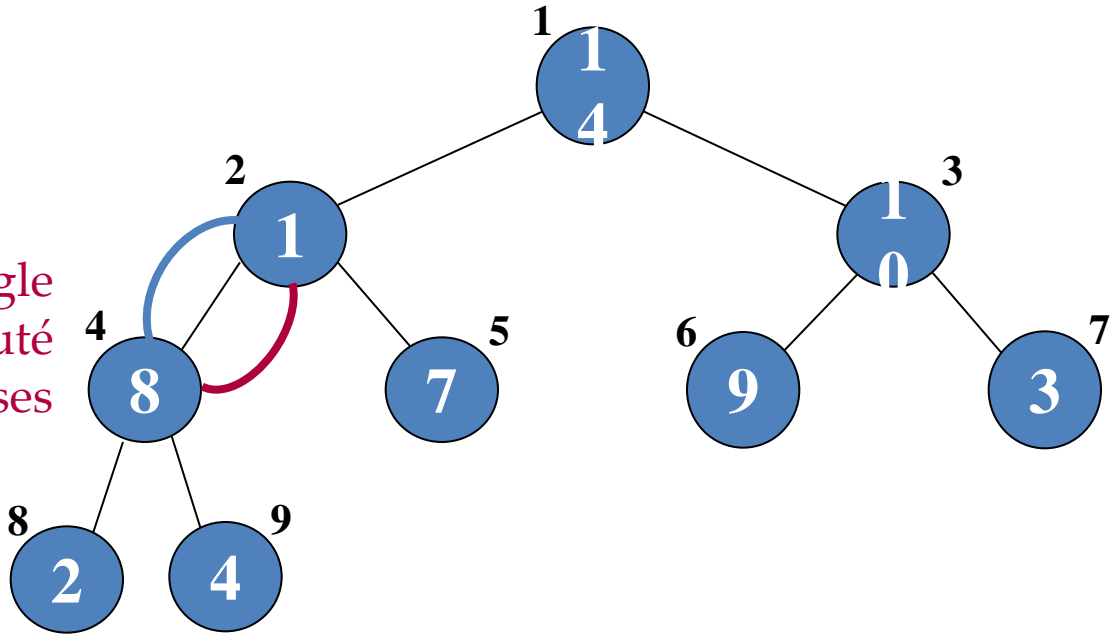


k	1	2	3	4	5	6	7	8	9	10
C[k]	1	14	10	8	7	9	3	2	4	16

~ Tri d'un tas ordonné

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Un nœud qui viole la règle d'entassement est permuté avec le plus grand de ses fils.

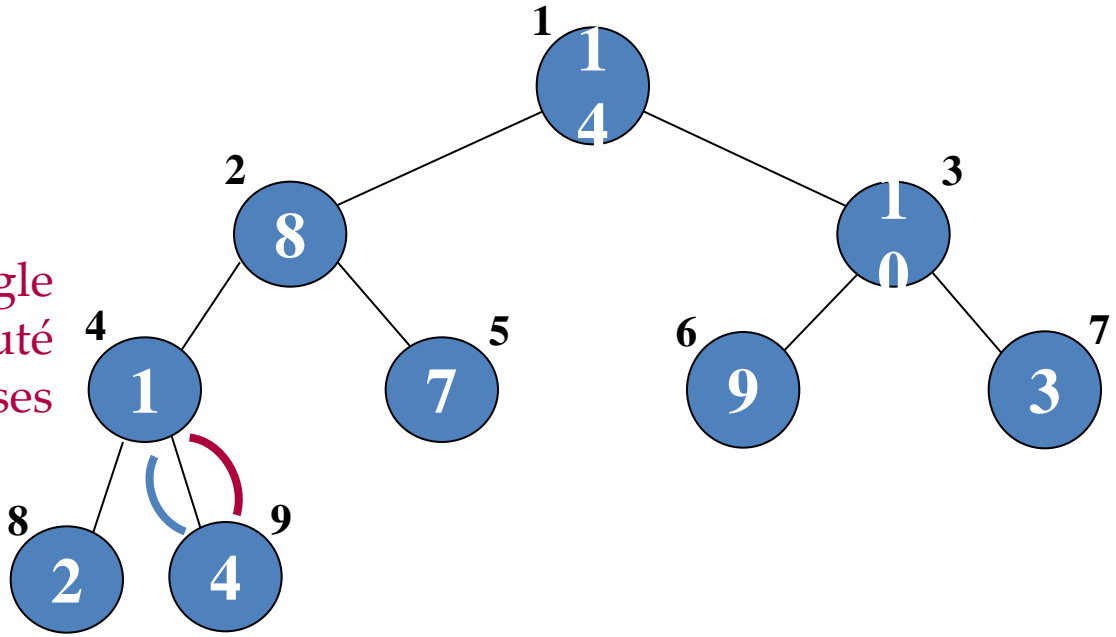


k	1	2	3	4	5	6	7	8	9	10
C[k]	14	1	10	8	7	9	3	2	4	16

~ Tri d'un tas ordonné

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Un nœud qui viole la règle d'entassement est permuté avec le plus grand de ses fils.

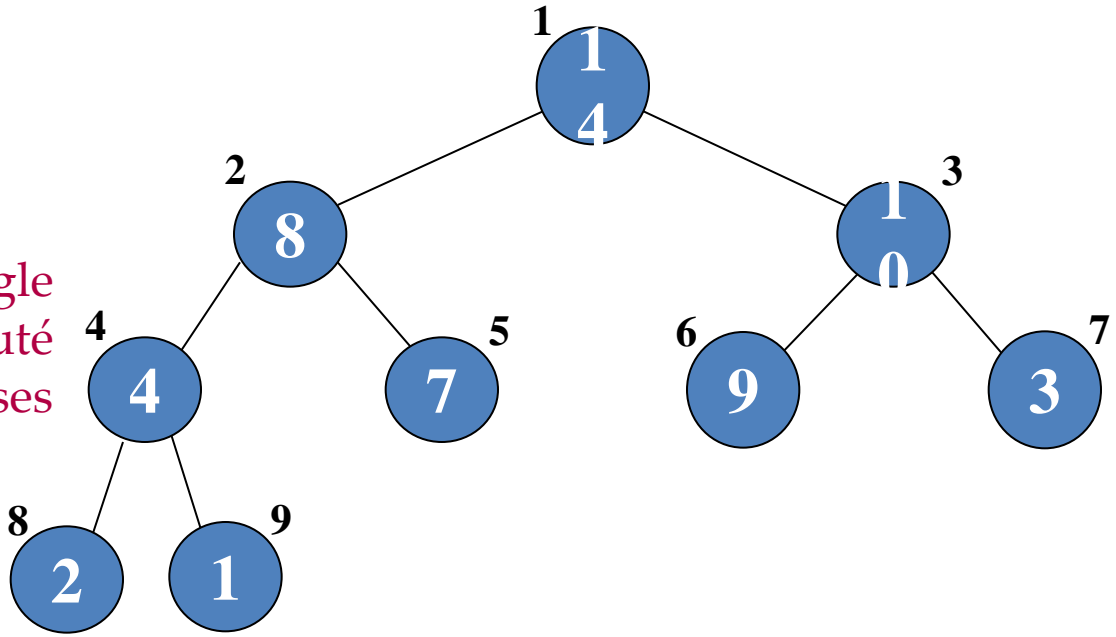


k	1	2	3	4	5	6	7	8	9	10
C[k]	14	8	10	1	7	9	3	2	4	16

~ Tri d'un tas ordonné

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Un nœud qui viole la règle d'entassement est permuté avec le plus grand de ses fils.



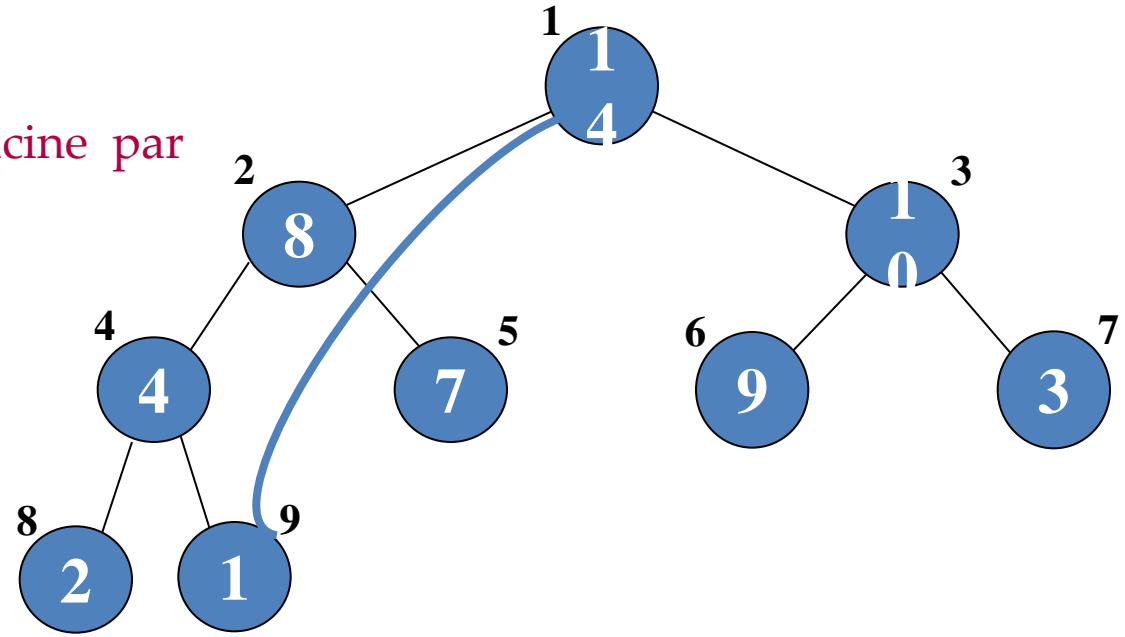
k	1	2	3	4	5	6	7	8	9	10
C[k]	**14**	**8**	**10**	**4**	**7**	**9**	**3**	**2**	**1**	**16**

~ Tri d'un tas ordonné

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Sauvegarde de la racine.

Remplacement de la racine par la dernière feuille.

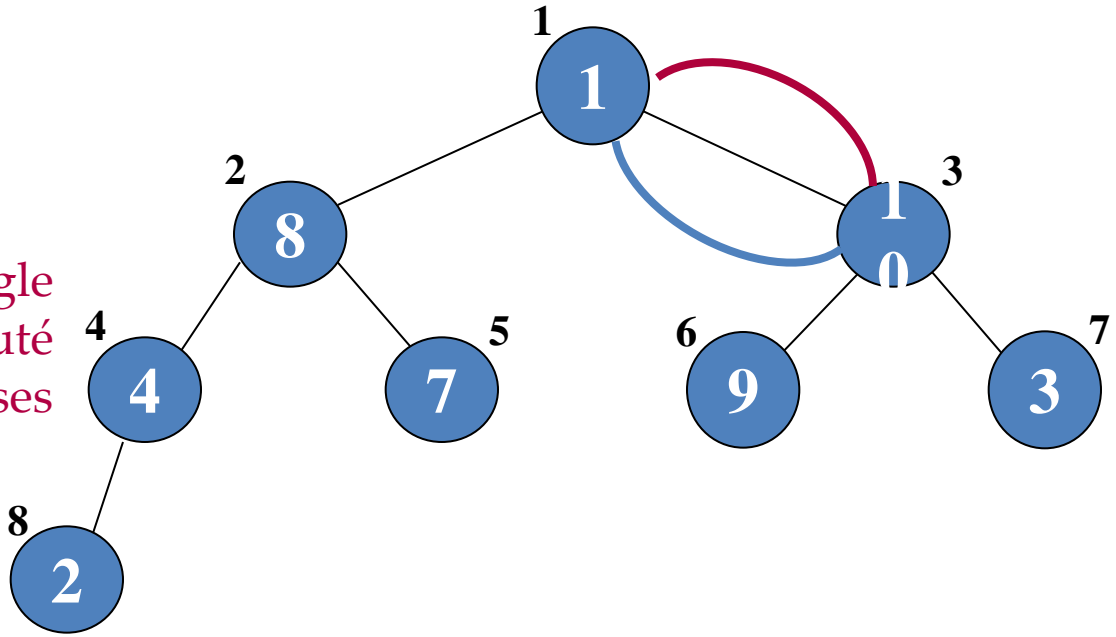


k	1	2	3	4	5	6	7	8	9	10
C[k]	**14**	**8**	**10**	**4**	**7**	**9**	**3**	**2**	**1**	**16**

~ Tri d'un tas ordonné

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

Un nœud qui viole la règle d'entassement est permuté avec le plus grand de ses fils.



k	1	2	3	4	5	6	7	8	9	10
C[k]	1	8	10	4	7	9	3	2	14	16

séquence = {14, 8, 9, 2, 1, 10, 3, 16, 4, 7}

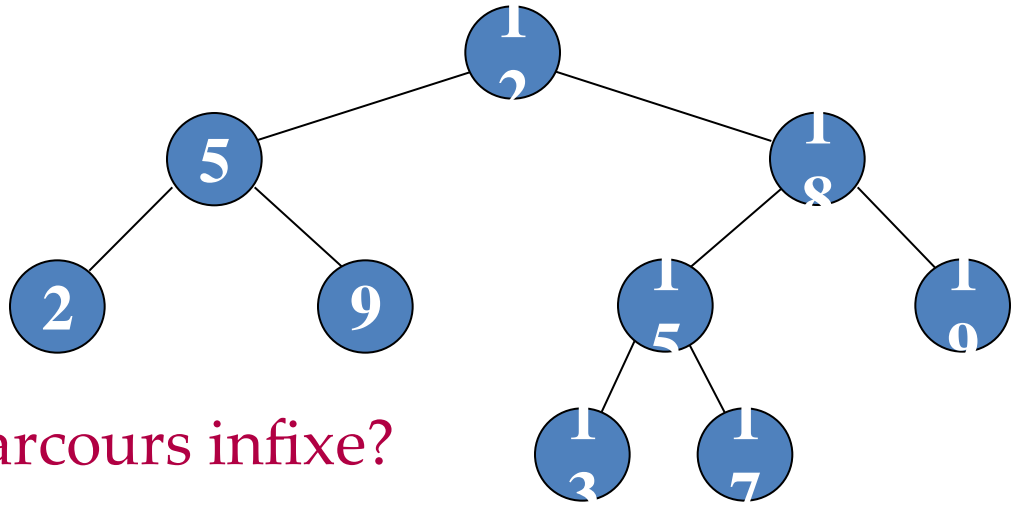
k	1	2	3	4	5	6	7	8	9	10
C[k]	10	8	9	4	7	1	3	2	14	16
C[k]	9	8	3	4	7	1	2	10	14	16
C[k]	8	7	3	4	2	1	9	10	14	16
C[k]	7	4	3	1	2	8	9	10	14	16
C[k]	4	2	3	1	7	8	9	10	14	16
C[k]	3	2	1	4	7	8	9	10	14	16
C[k]	2	1	3	4	7	8	9	10	14	16
C[k]	1	2	3	4	7	8	9	10	14	16

~ Arbre binaire de recherche

- Un arbre binaire de recherche est un arbre binaire satisfaisant aux propriétés suivantes:
  - Chaque nœud possède une clé (ici considérée comme unique) permettant une relation d'ordre
  - Pour tout sous-arbre, la clé de la racine est supérieure à toutes les clés de son sous-arbre gauche et inférieure à toutes celles de son sous-arbre droit.

Relation d'ordre:
C[SAG(k)] ≤ C[k]
C[SAD(k)] ≥ C[k]

{12,5,18,2,9,19,15,13,17}



Parcours infixe?

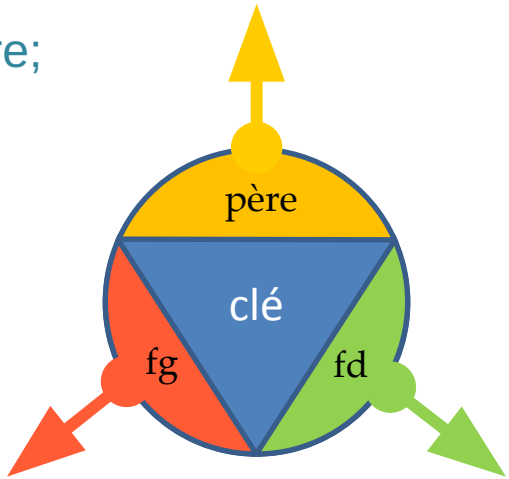
ENSEA

~ Arbre binaire de recherche

- On peut se rapprocher des implémentations mises en places sur les listes chaînées en utilisant des pointeurs. Chaque nœud peut alors être représenté par une SD auto-référentielle.

• structure et pointeur :

```
typedef short valeur;
typedef Nœud* pNoeud;
typedef struct Noeud {
    valeur cle;
    pNoeud pere;
    pNoeud fg;
    pNoeud fd;
} Noeud;
```



• constructeur/destructeur :

```
void ArbreCree(pNoeud A);
void ArbreDetruit(pNoeud A);
```

• modifieurs :

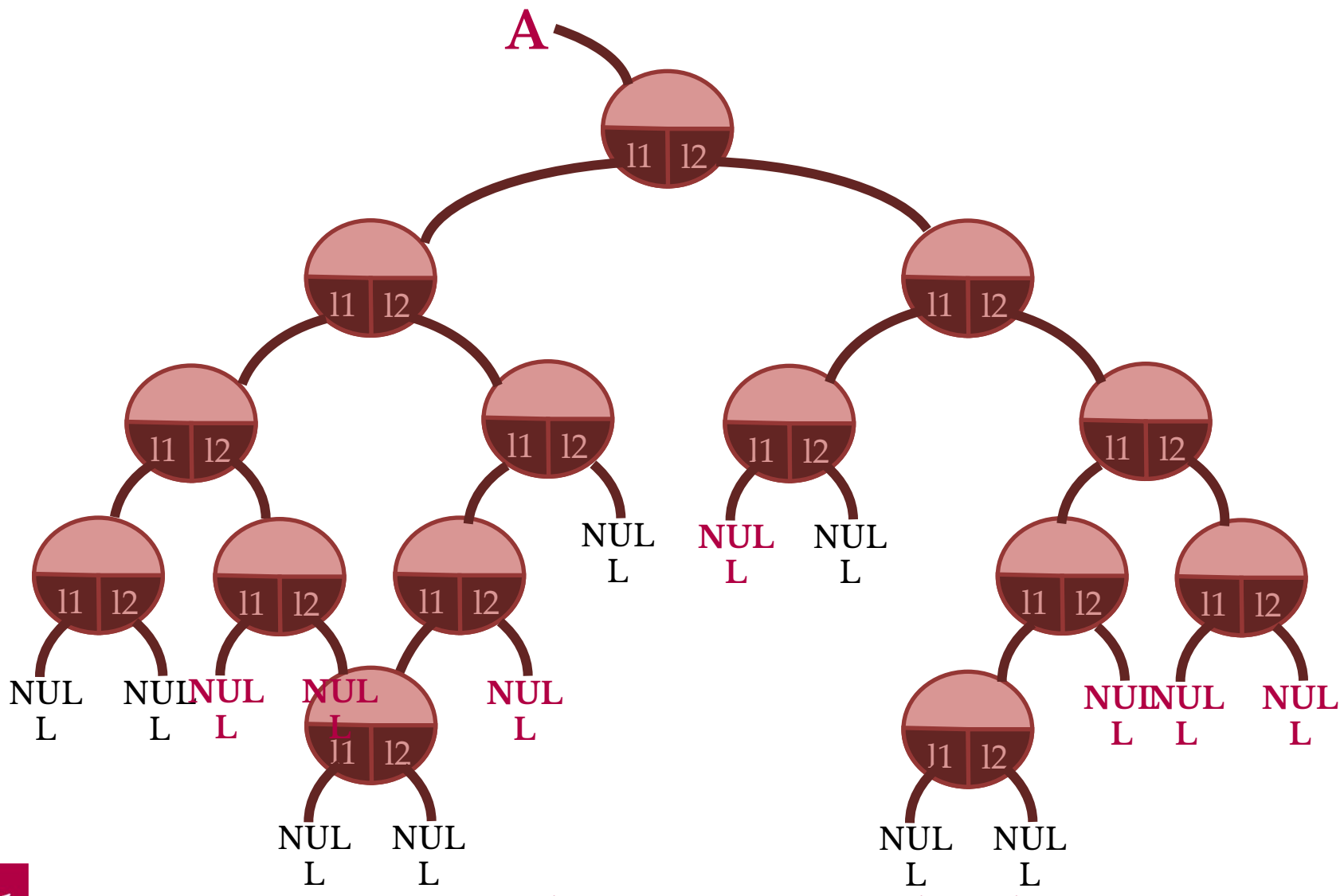
```
void ArbreInsere(pNoeud A, valeur cond);
void ArbreRetire(pNoeud A, valeur cond);
```

• sélecteurs :

```
booleen ArbreVide(pNoeud A) ;
short ArbreHauteur(pNoeud A);
```

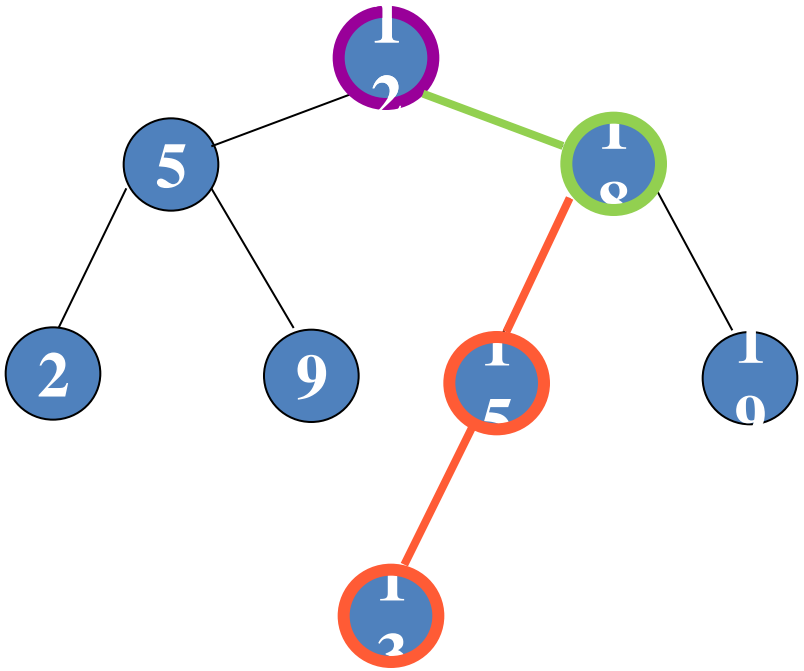
• itérateurs :

```
pNoeud ArbreRecherche(pNoeud A, valeur cond);
```


A
l1 l2
NULL

~ Arbre binaire de recherche : insertion

{12, 5, 18, 2, 9, 19, 15, **13**, 17}



~ Arbre binaire de recherche : suppression

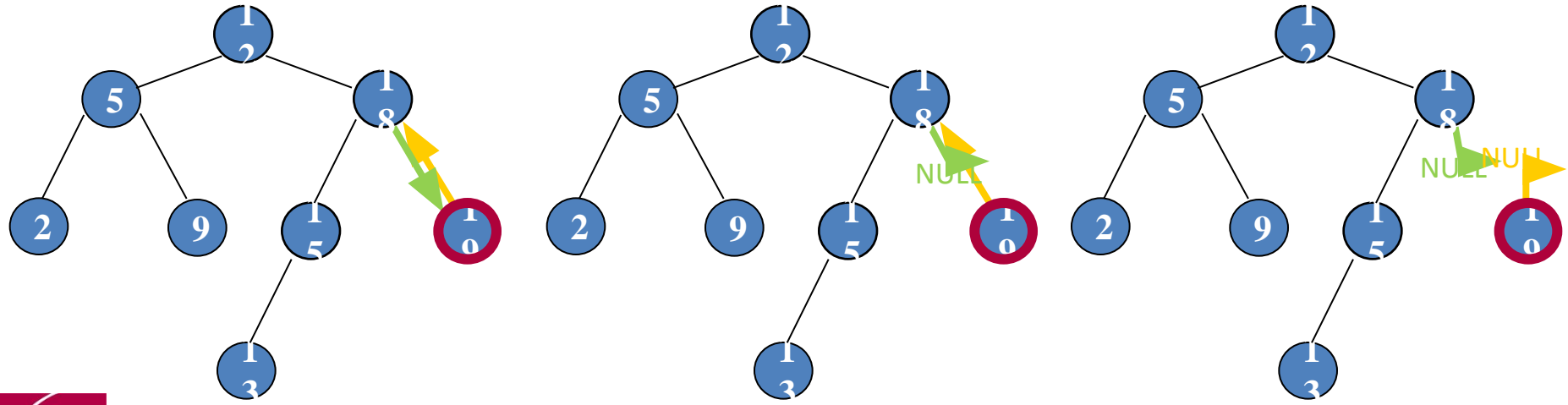
1er cas : nœud sans fils

{12,5,18,2,9,**19**,15,13}

1) localisation du nœud père

2) rupture de la paternité

3) rupture de la filiation



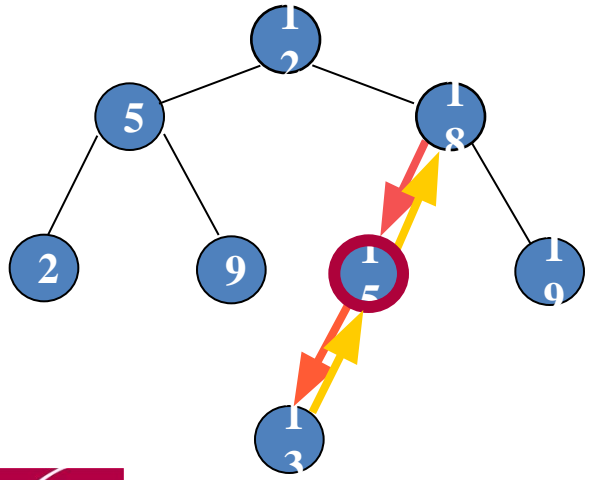
ENSEA

~ Arbre binaire de recherche : suppression

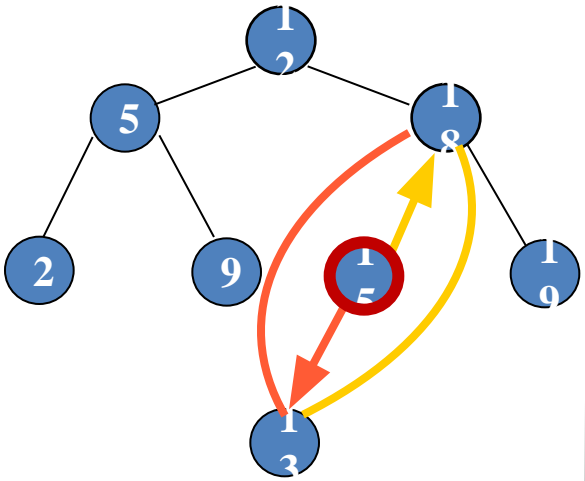
2ème cas : nœud à 1 fils

{12,5,18,2,9,19,**15**,13}

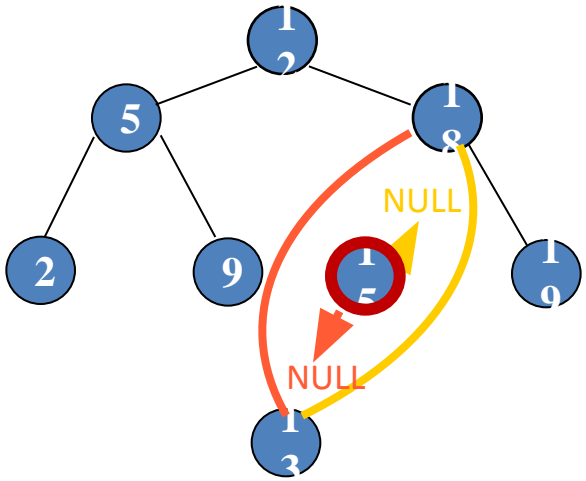
1) localisation des nœuds père et fils



2) adoption du petit-fils par le grand-père et réciproquement



3) père rompt tous les liens

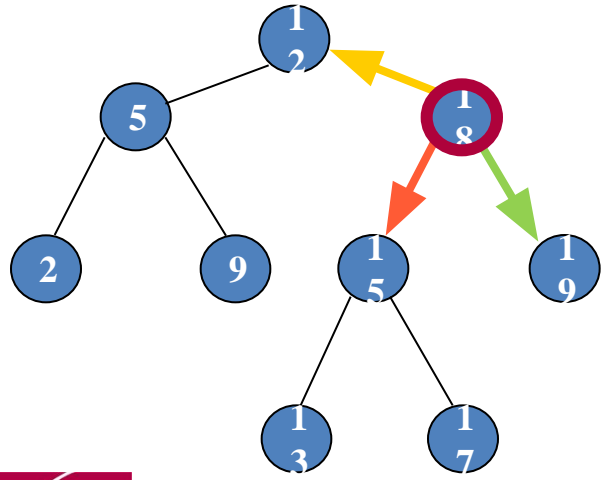


~ Arbre binaire de recherche : suppression

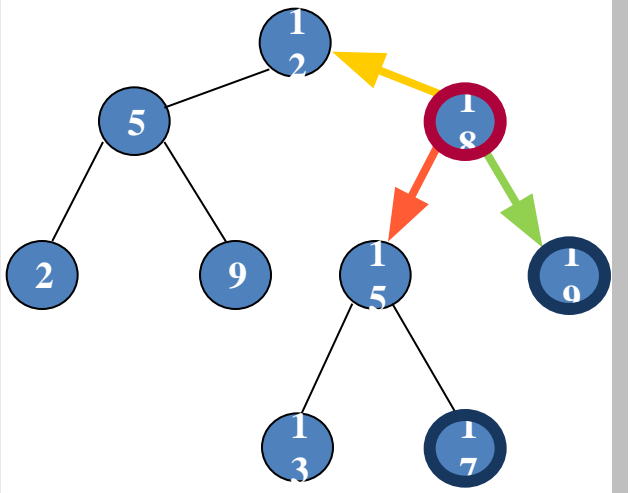
3ème cas : nœud à 2 fils

{12,5,**18**,2,9,19,15,13,17}

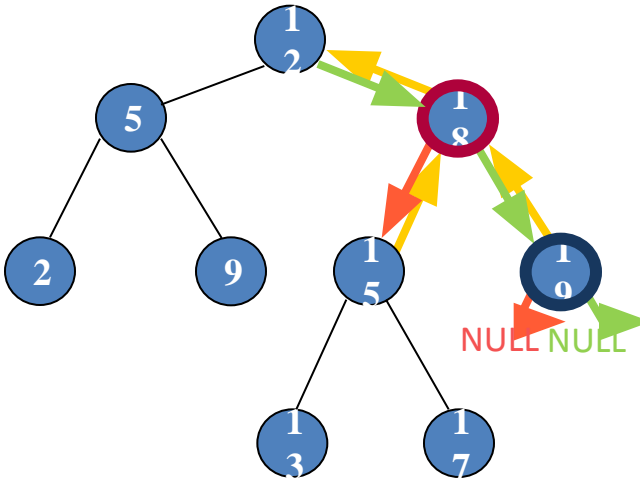
1) localisation du nœud père



2) localisation des remplaçants



3) sélection du remplaçant

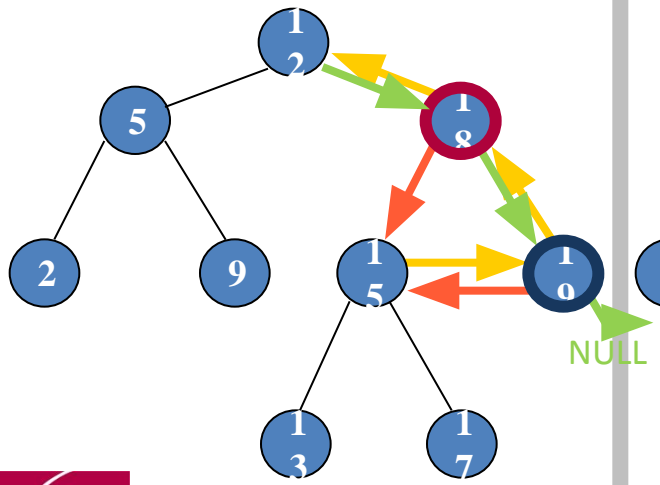


~ Arbre binaire de recherche : suppression

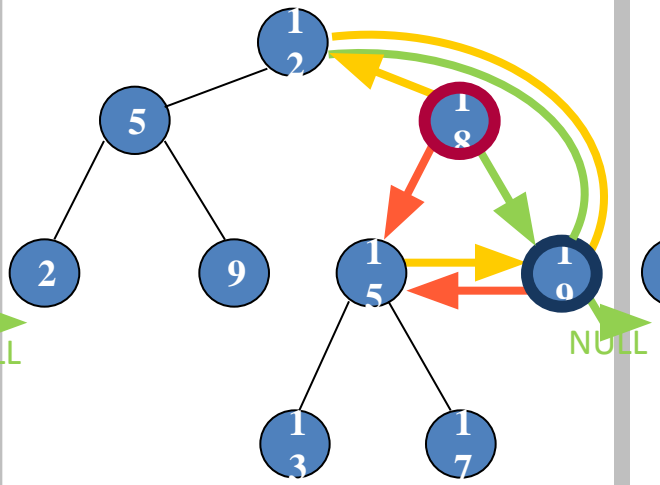
3ème cas : nœud à 2 fils

{12,5,**18**,2,9,19,15,13,17}

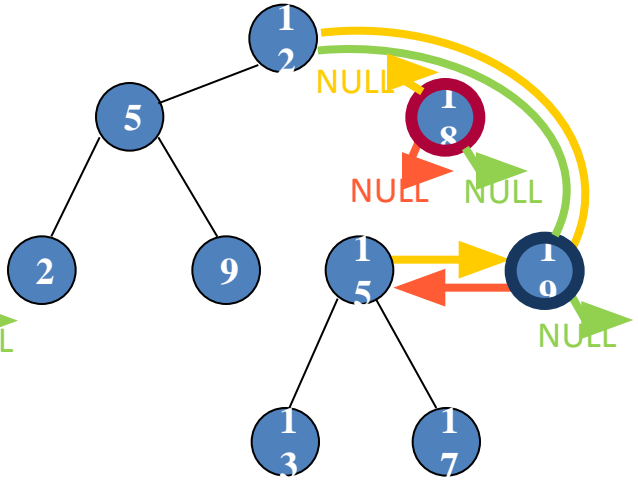
4) le remplaçant adopte les fils du père



5) le grand-père adopte le remplaçant, qui le reconnaît



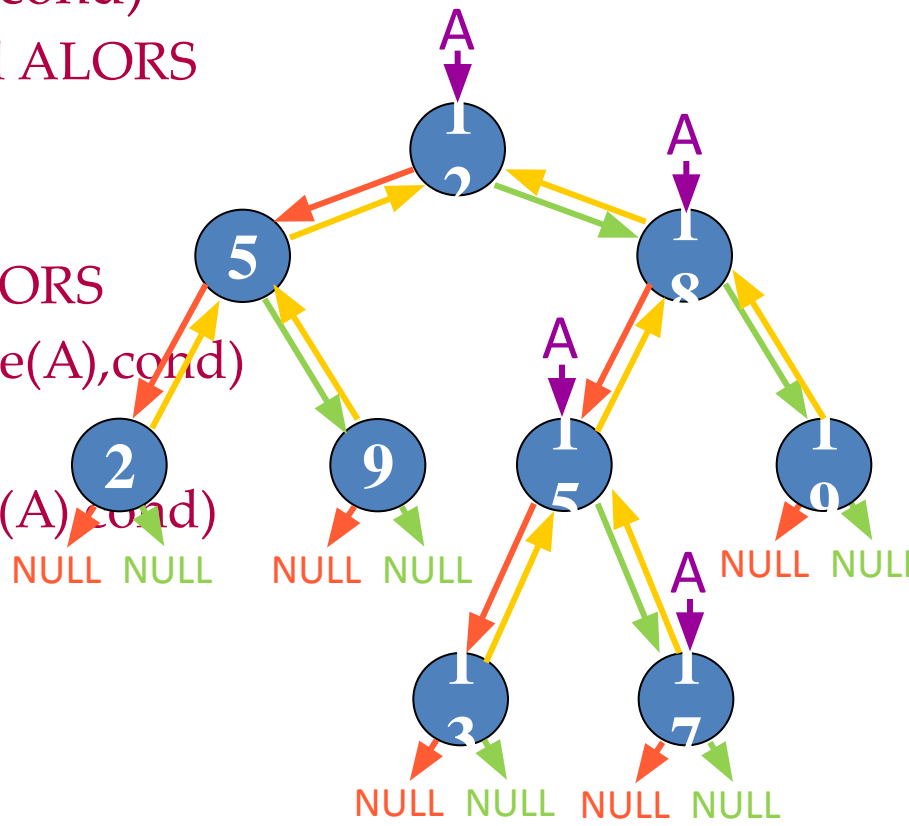
6) père rompt tous les liens



~ Arbre binaire de recherche : recherche

- Algorithme Recherche (arbre, cond)

```
SI A NULL ou clé(A) égale cond ALORS
   retourner A;
SINON
   SI clé(A) supérieur à cond ALORS
      retourner Recherche(gauche(A),cond)
   SINON
      retourner Recherche(droite(A),cond)
   FIN SI
FIN SI
```



~ Comment comparer deux algorithmes ?

- Temps d'exécution
  - ✷ Se mesure en fonction de la quantité d'éléments traités par l'algorithme
    - ✶ Instructions coûteuses ou au contraire opérations élémentaires
    - ✶ Nombre de valeurs dans un tableau, fichier
  - ✷ Complexité
    - ✶ Dans le pire cas (si pire cas > cas inadmissible)
    - ✶ En moyenne (généralement)
    - ✶ Dans le meilleur des cas
    - ✶ $f(n) = O(g(n))$: il existe une constante réelle positive c et un entier positif n_0 tels que $f(n) < c.g(n)$ pour tout $n > n_0$
- Utilisation de la mémoire

- Un tableau peut être trié en un temps quasi-linéaire
  - Cas d'un arbre équilibré
    - Relation entre hauteur h et nombre de nœuds n : $n \approx 2^h - 1$
    - Cas du tas
      - Entassement (insertion de n éléments sur une hauteur h): O(n log n)
      - Tri (permutation de n éléments sur une hauteur h) : O(n log n)
    - Cas de l'arbre de recherche
      - Insertion de n éléments sur une hauteur h : O(n log n)
      - Parcours des n éléments : O(n)
- A titre de comparaison :
  - tri à bulles, tri par insertion : $O(n^2)$
  - tri rapide : O(n log n)
  - tri par Arbre Binaire de Recherche (dégénéré) : $O(n^2)$

~ Pourquoi équilibrer un arbre?

- Complexité selon le nombre d'éléments

n	n log n	n^2
10	33,2	100
100	664	10.000
1.000	9.960	1.000.000
10.000	13.280	100.000.000

~ Comment?

- 2 approches possibles
  - Arbre construit non modifié : linéariser, puis arboriser.
  - Arbre souvent modifié : imposer une contrainte supplémentaire lors de la modification pour laisser l'arbre équilibré

~ Définition

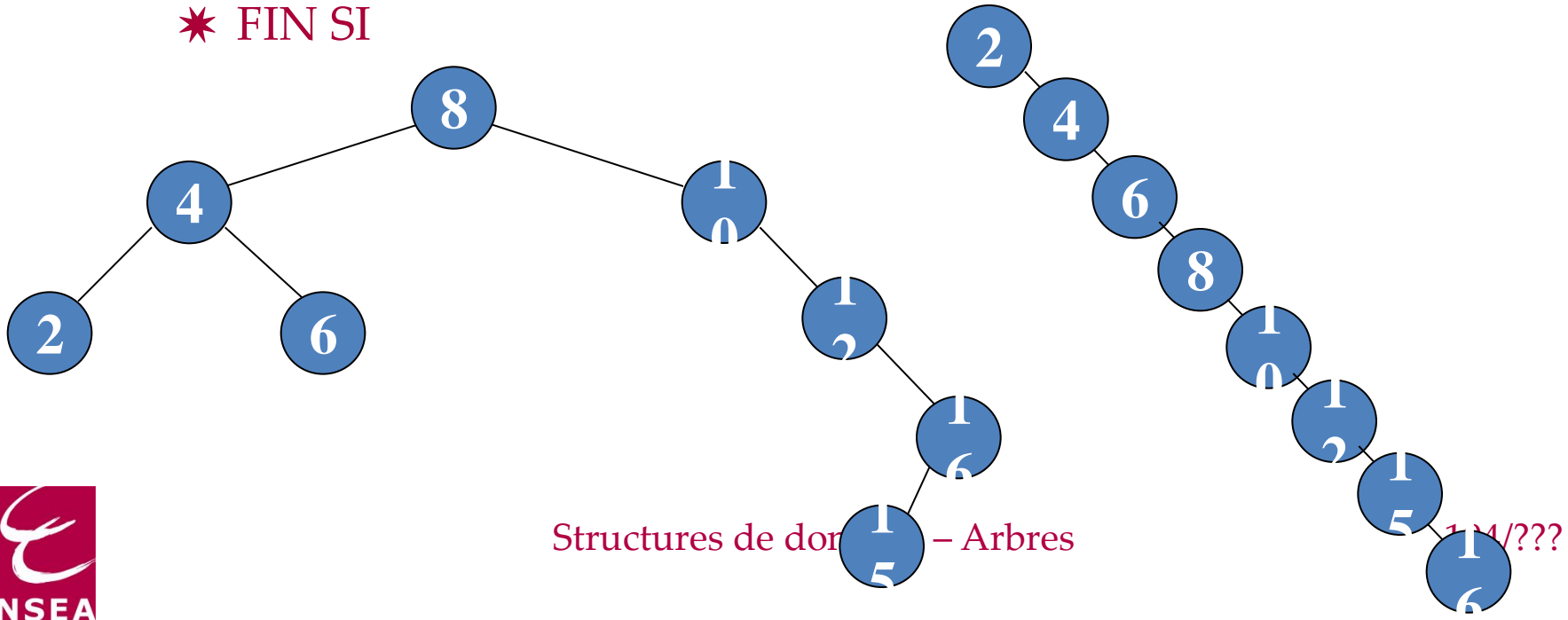
- Un arbre est équilibré si, pour chacun de ses nœuds, la différence entre la hauteur du sous-arbre gauche et celle du sous-arbre droit ne dépasse pas une unité.

$$-1 \leq \text{Hauteur(Droit)} - \text{Hauteur(Gauche)} \leq 1$$

~ Phase de linéarisation

Algorithme Linéariser

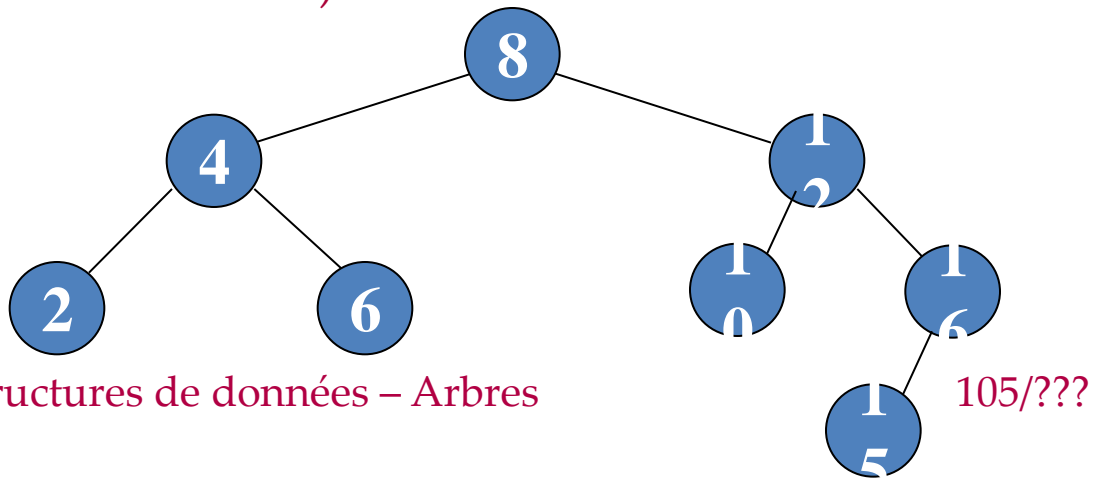
- SI l'arbre n'est pas vide ALORS
  - Linéariser le sous-arbre droit en utilisant la liste
  - Insérer la racine de l'arbre en tête de liste
  - Incrémenter de 1 le nombre d'éléments
  - Linéariser le sous-arbre gauche en utilisant la liste
- FIN SI



~ Phase d'arborisation

Algorithme Arboriser

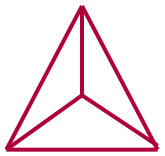
- SI le nombre de nœuds > 0 ALORS
  - Arboriser un sous-arbre gauche à partir des (n-1)/2 (premiers) nœuds de la liste
  - Extraire le premier nœud restant dans la liste et en faire la racine de l'arbre
  - Arboriser un sous-arbre droit à partir des n/2 nœuds restants de la liste
  - Accrocher le sous-arbre gauche à gauche de la racine
  - Accrocher le sous-arbre droit) droite de la racine
- FIN SI



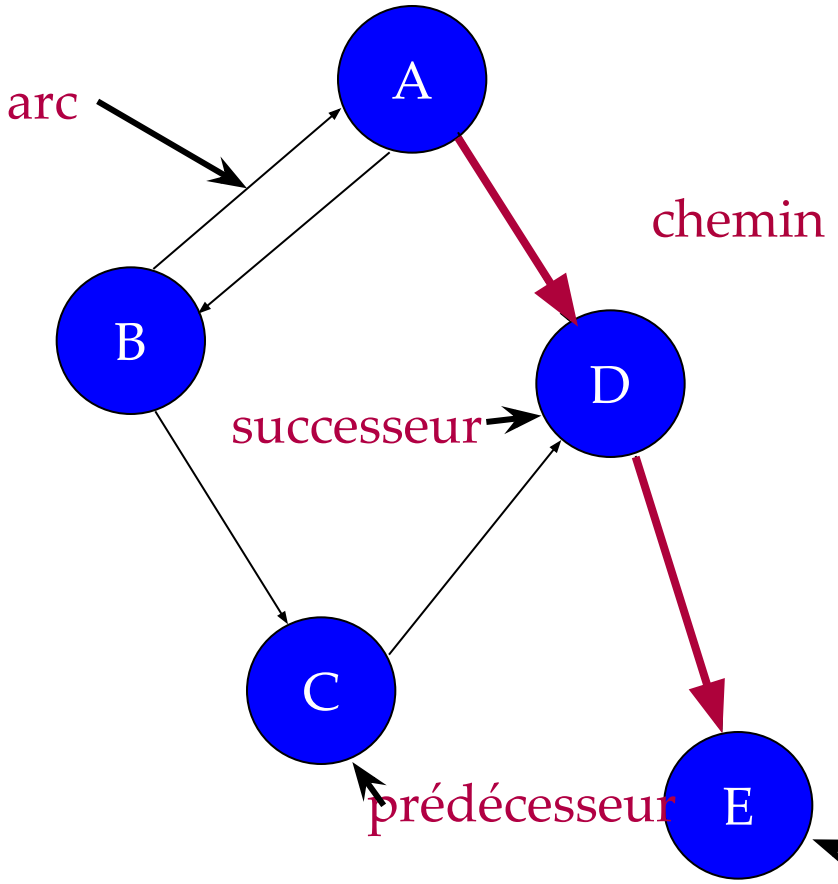
~ Notion de graphe

- Généralise la notion de relation sur un ensemble en formalisant l'existence des liaisons entre des objets.
- Constitue :
  - un modèle polyvalent d'organisation de données,
  - une méthode de description d'un large spectre de problèmes (calcul de distances par exemple),
  - un moyen de recherche de cycles dans les relations et de connections.

- Est représentable graphiquement s'il n'est pas trop complexe, MAIS il n'est pas toujours évident de reconnaître le même graphe derrière 2 représentations différentes comme c'est le cas ici

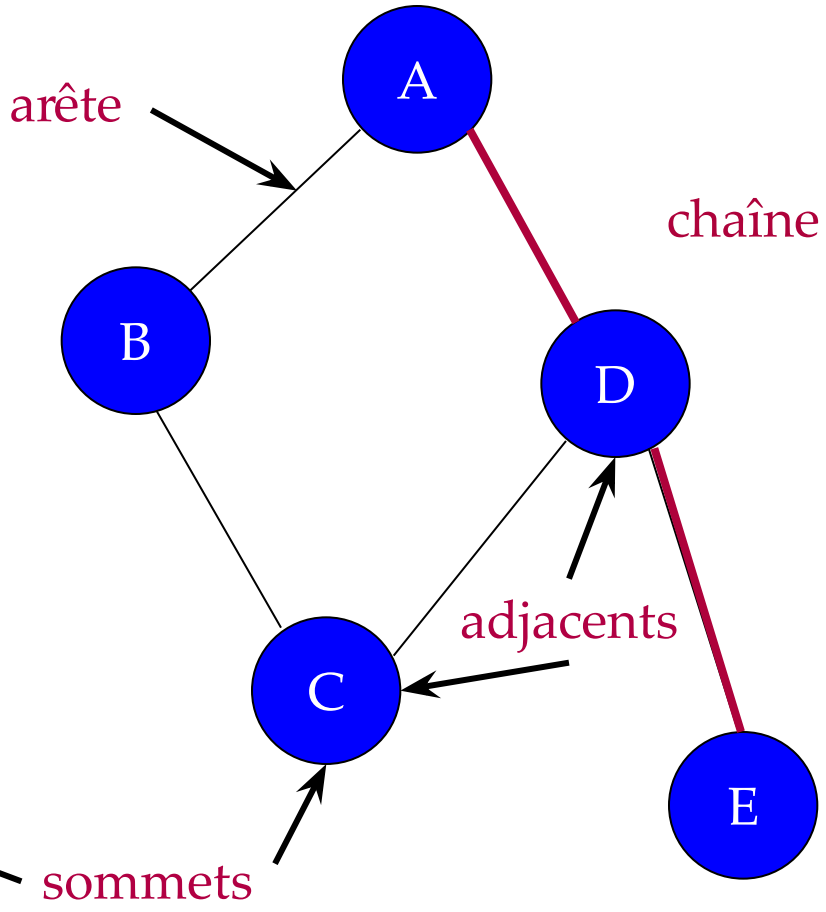


Graphe orienté



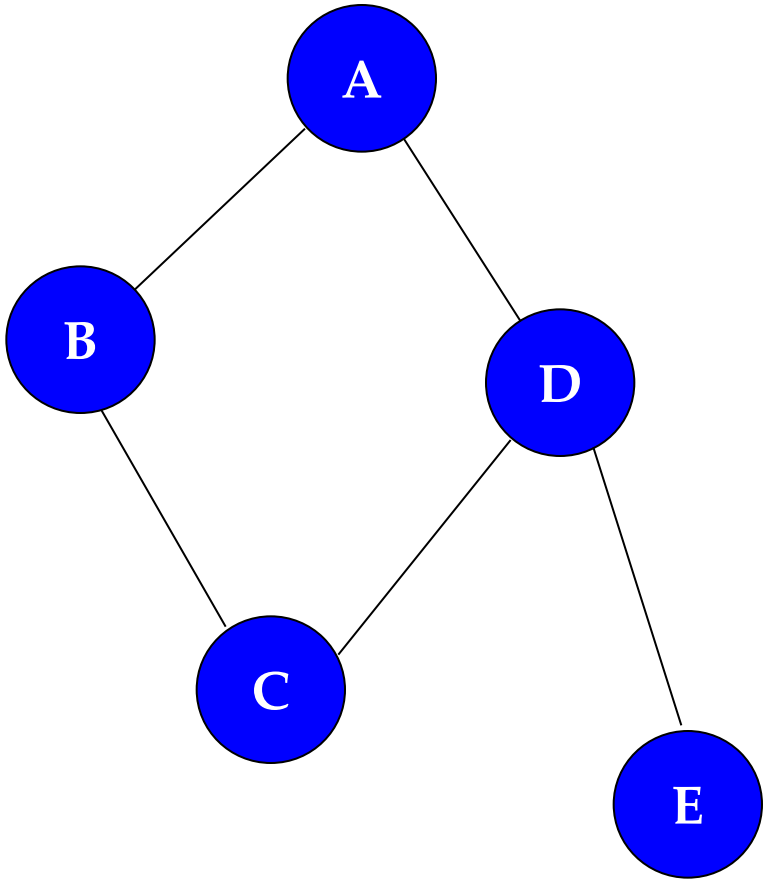
A = {(A,B),(A,D),(B,A),(B,C),…}

Graphe non-orienté

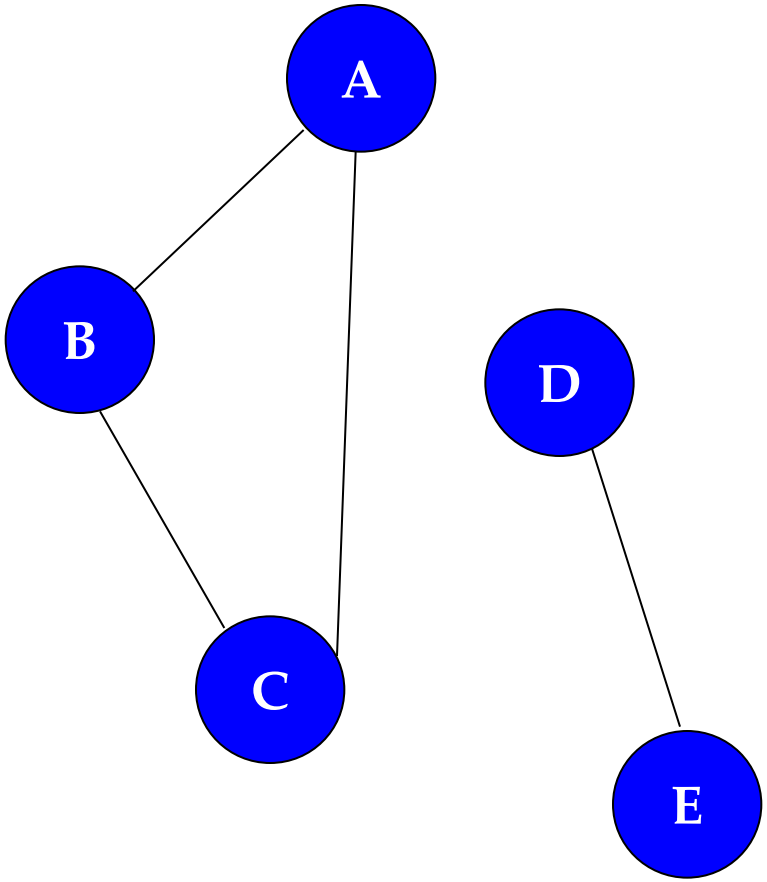


A = {{A,B},{A,D},{B,C},{C,D},{D,E}}

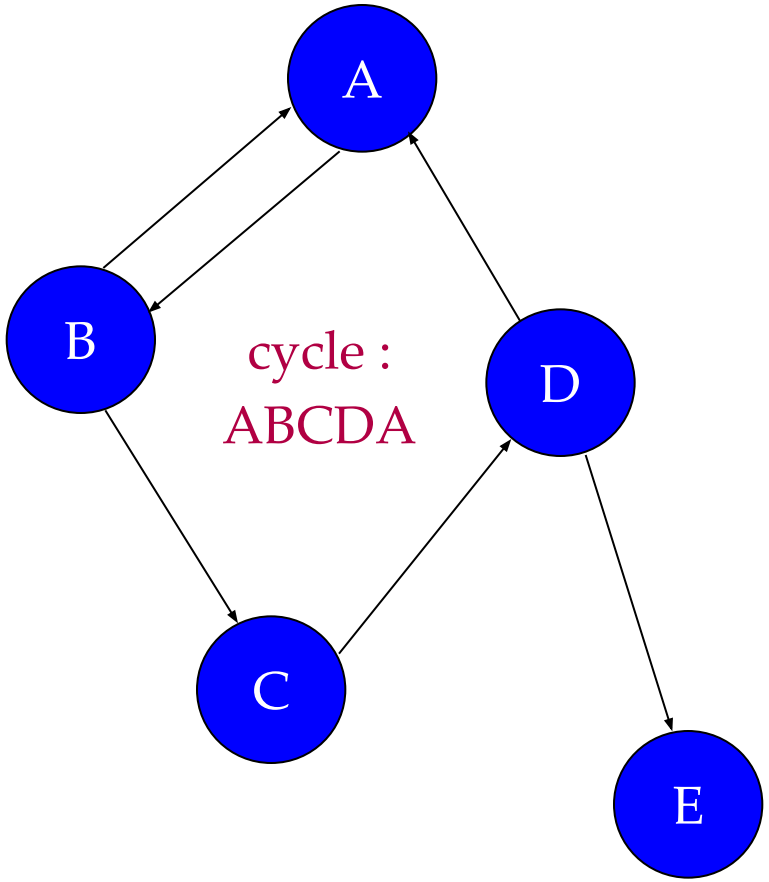
Graphe connexe



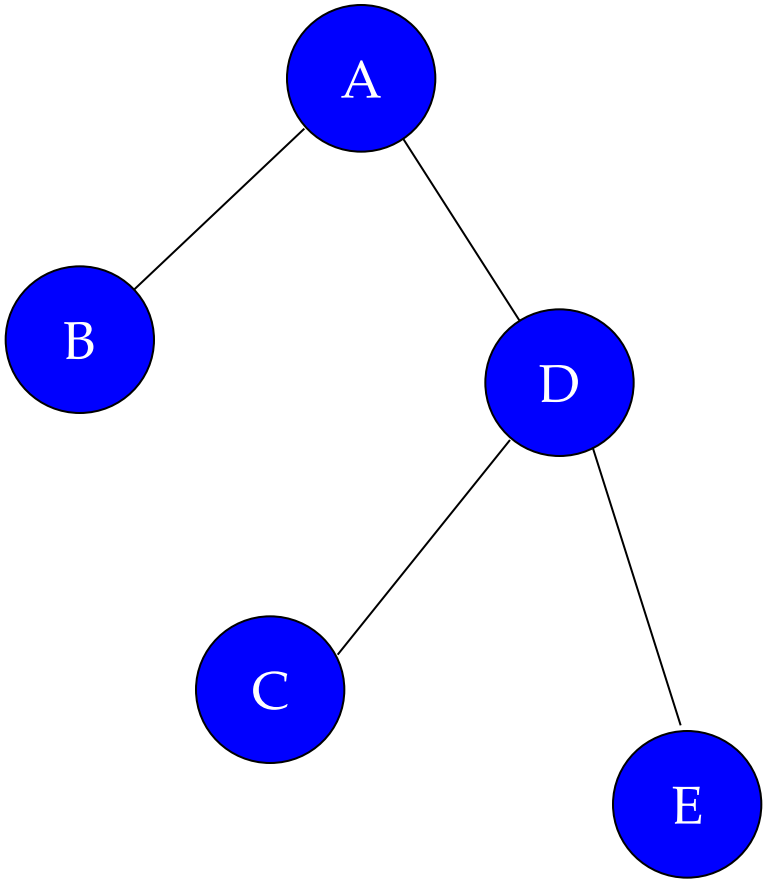
Graphe non-connexe



Graphe cyclique



Graphe acyclique

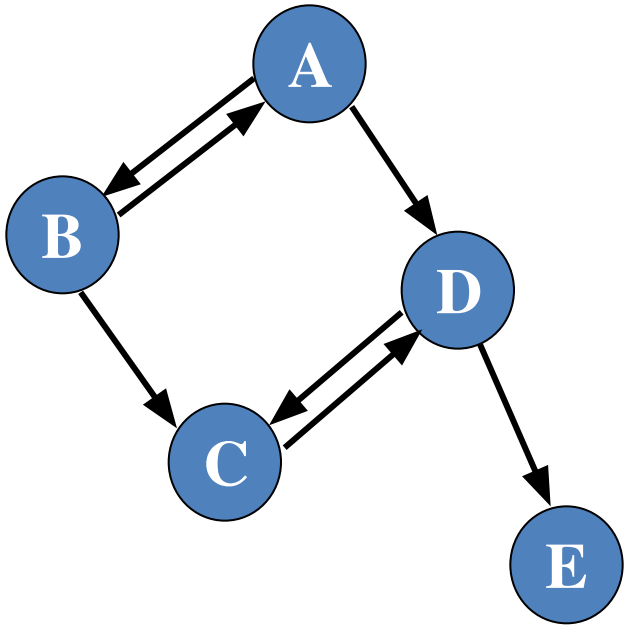


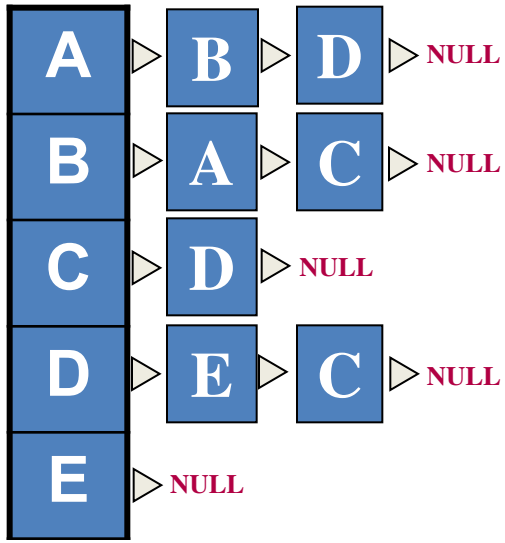
- L'information est portée par les **sommets** :
  - aucune lorsque seule la structure importe,
    ex : le problème des 7 ponts de Königsberg
  - une étiquette lorsque les sommets sont distincts
    ex : les atomes C, H, N d'une molécule
  - un ensemble de valeurs
    ex : intersection de réseaux ferrés, routiers, informatiques

- L'information est portée par les **liaisons** :
  - aucune lorsque seuls les sommets importent
    ex : molécules
  - une étiquette si l'ordre ou la numération importe
    ex :
  - un ensemble de valeurs
    ex : tronçons de réseaux ferrés, routiers, informatiques

~ Pour les graphes clairsemés $N_a << N_s^2$

- tableau indexé par les numéros des sommets,
- chaque $i^{ème}$ case est une liste chaînée des sommets adjacents au $i^{ème}$ sommet.





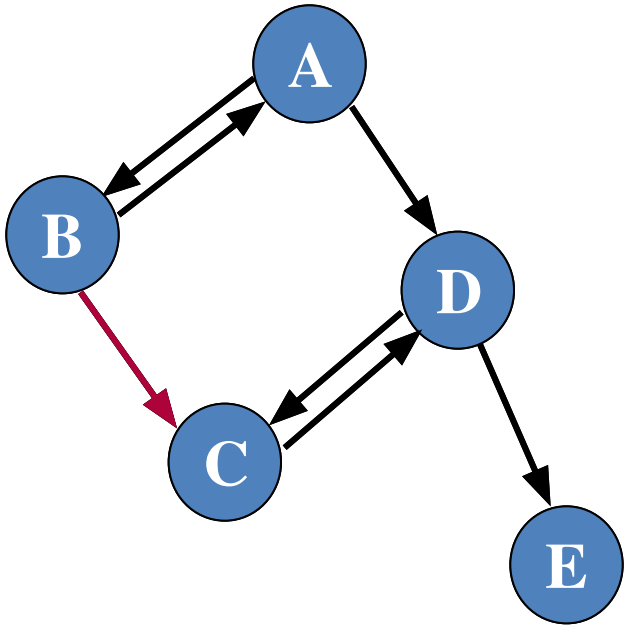
~ Pour les graphes clairsemés $N_a << N_s^2$

- tableau indexé par les numéros des sommets,
- chaque $i^{ème}$ case est une liste chaînée des sommets adjacents au $i^{ème}$ sommet.
- Limite les ressources mémoire nécessaires au stockage de l'information.
- Obligation de parcourir toute la liste d'adjacence d'un sommet pour vérifier l'existence d'une liaison.

L'espace mémoire est réduit au dépend de la rapidité d'exécution.

~ Pour les graphes denses $N_a \approx N_s^2$

- matrice dont les cordonnées sont les numéros des sommets,
- chaque case de coordonnées (i,j) indique la présence d'une liaison (1) ou pas (0) entre le sommet i et le sommet j.



i \ j	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	0	0
C	0	0	0	1	0
D	0	0	1	0	1
E	0	0	0	0	0

~ Pour les graphes denses $N_a \approx N_s^2$

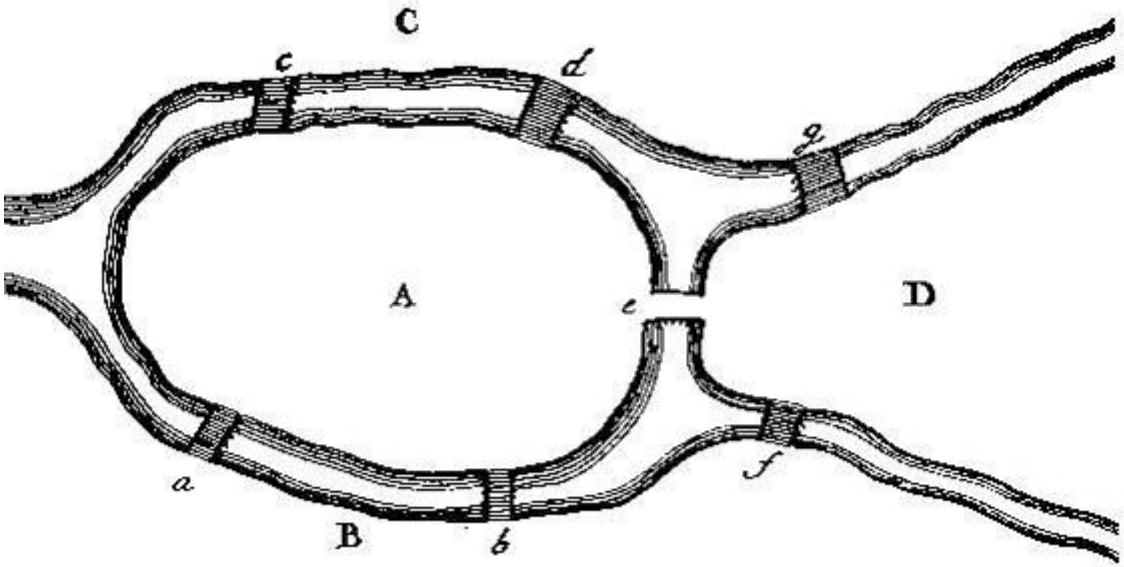
- matrice dont les cordonnées sont les numéros des sommets,
- chaque case de coordonnées (i,j) indique la présence d'une liaison (1) ou pas (0) entre le sommet i et le sommet j.`

- Les ressources mémoire nécessaires au stockage de l'information sont toujours de N_s^2 cases.

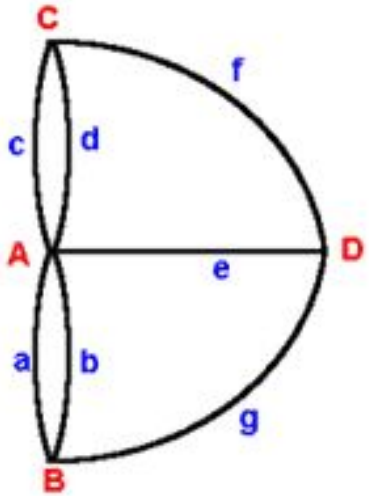
- L'accès à l'information de liaison est très rapide.

La rapidité d'exécution est privilégiée à l'espace mémoire.

~ Les 7 ponts de Königsberg

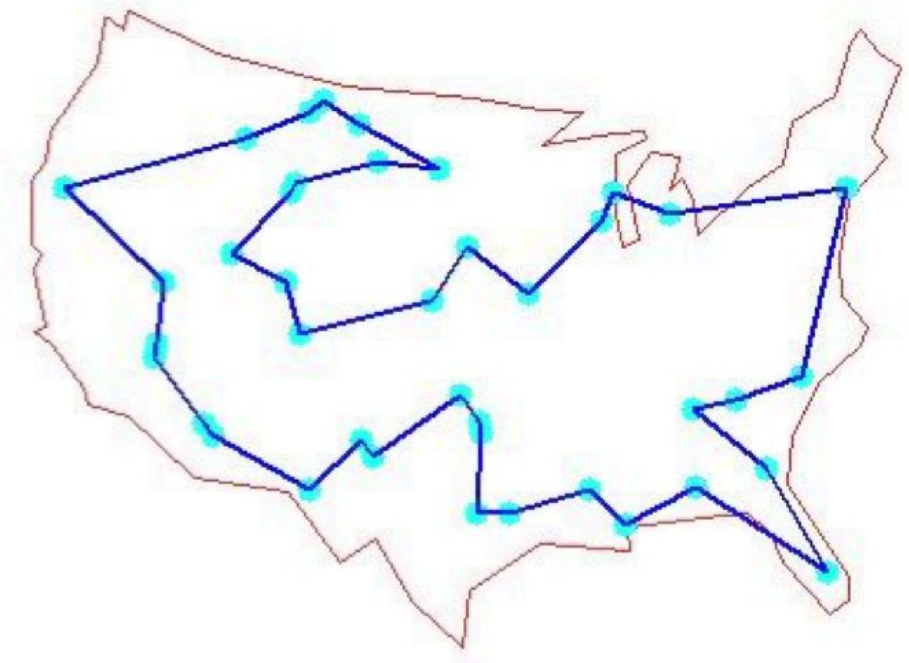


Leonhard EULER
(1707-1783)



les origines :
http://eurserveur.insa-lyon.fr/approphys/9Math&Phys/graphes/LA%20THEORIE%20DES%20GRAPHES/histoire.html

~ Le problème du VRP



~ Occurrence de mots

~ La théorie des graphes (1736) permet de représenter et modéliser de nombreux problèmes actuels.

~ Les arbres introduisent la notion de dépendance / héritage / chronologie entre les éléments.

~ Les arbres binaires et les tas sont des solutions à l'ordonnancement et au tri des éléments.

~ Les Bases de Données généralisent tous les concepts de SD pour gérer, organiser, modifier et accéder à un corpus de données conséquent.