Matière : Structures de données

Classe : 3A-IIIA

Professeur : Mr Y.OUASSIT

Année Universitaire : 2026/2027

Tour de Table



1. Nom & Prénom
2. Cursus Universitaire
3. Vos Attentes

Plan détaillé du cours

Liste des chapitres :

- **Complexité algorithmique**
- **Algorithmes de tri et recherche**
- **Rappel : Pointeurs, Tableaux et Enregistrements**
- **Listes chainées**
- **Piles et files**
- **Arbres : Arbres binaires et arbres binaires de recherche**
- **Graphes**
- **Tables de Hachage**

Objectifs du cours

Objectif : Permettre à l'étudiant de comprendre, concevoir et manipuler les principales structures de données utilisées en informatique pour résoudre efficacement des problèmes algorithmiques.

Répartition du volume horaire :

CM : *14 H*

TD : *10.5 H*

TP : *10.5 H*

Mode d'évaluation : Contrôles continues 20% + Travaux pratiques 20% + Examen écrit final 60%

Année Universitaire : 2026/2027

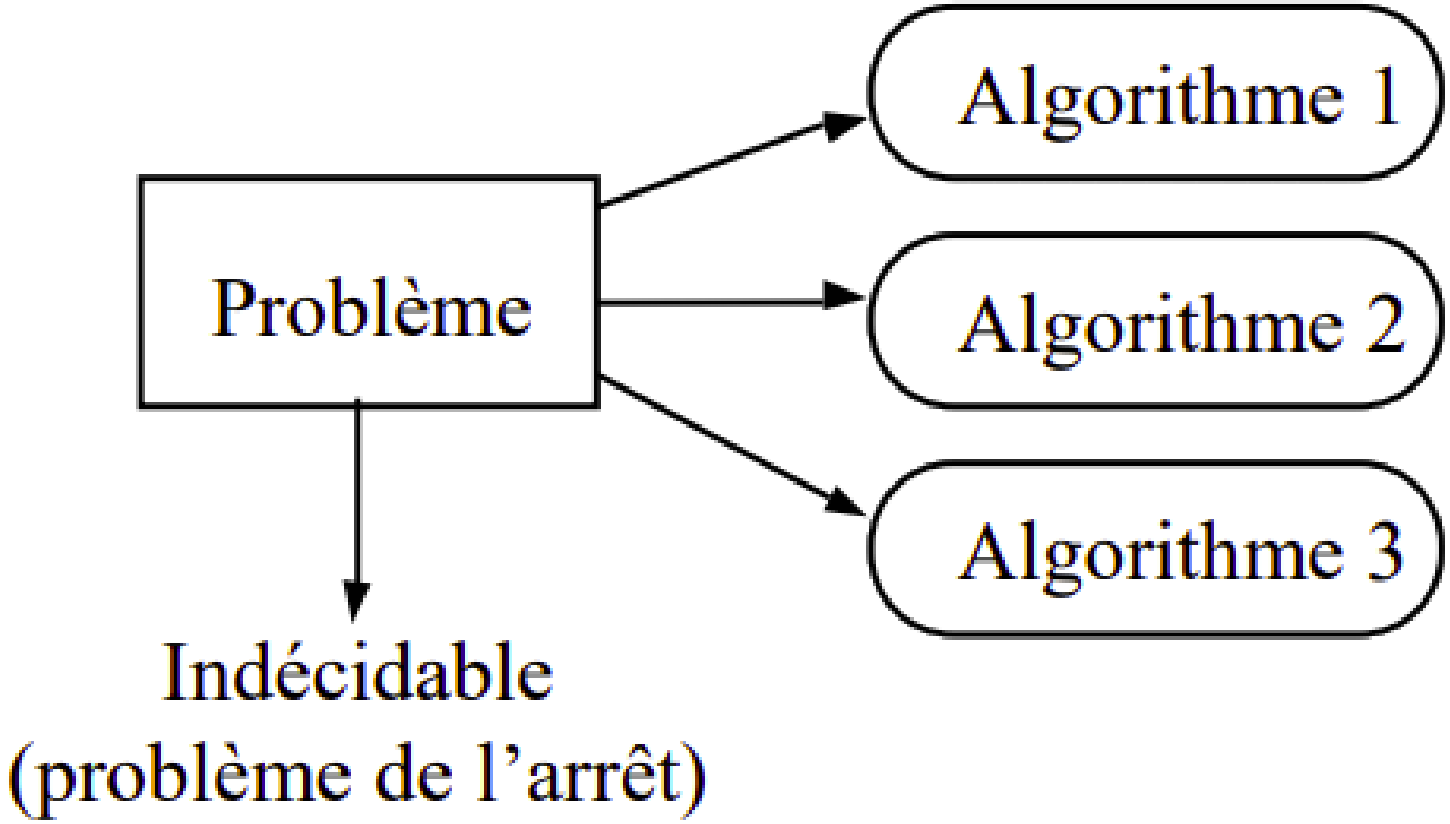
Complexité Algorithmique

HESTIM ENGINEERING & BUSINESS SCHOOL

Plan

- Introduction
- Calcul du coût d’un algorithme
- Types de complexité
- Complexité asymptotique, Notation O
- Complexité des fonctions récursives
- Exercices

Introduction



Introduction

La suite Fibonacci : Version récursive

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
long fibonacci(int n) {
    if (n <= 1)
        return n;

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Introduction

La suite Fibonacci : Version itérative

```
long fibonacci(int n) {
    if (n <= 1)  return n;
    long a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        long temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

Introduction

La suite Fibonacci : Version récursive avec mémoïsation

```
long fibonacciMemo(int n, long memo[]) {

    if (n <= 1)  return n;

    if (memo[n] != -1)  return memo[n];

    memo[n] = fibonacciMemo(n - 1, memo)
            + fibonacciMemo(n - 2, memo);

    return memo[n];
}
```

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

Introduction

La suite Fibonacci : Version programmation dynamique

```
long fibonacciDP(int n) {
    if (n <= 1) return n;

    long fib[n + 1];

    fib[0] = 0;
    fib[1] = 1;

    for (int i = 2; i <= n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];

    return fib[n];
}
```

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

Introduction

Quel est l'algorithme le plus efficace ?

Qu'est ce qui fait qu'un algorithme est plus efficace (critère de comparaison) ?

Comment comparer deux algorithmes ?

Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre!

Introduction

La complexité d'un algorithme : est une mesure de la quantité de ressources nécessaires à son exécution. On distingue deux types de complexité :

- **Complexité temporelle :** mesure le temps de calcul consommé lors de l'exécution d'un programme
- **Complexité spatiale :** mesure le nombre d'emplacements mémoire occupés lors de l'exécution d'un programme

On s'intéresse essentiellement à la complexité en temps (ce qui n'était pas forcément le cas quand les mémoires coutaient cher)

Complexité temporelle vs spatiale

- Règle (non officielle) de l'espace-temps informatique : pour gagner du temps de calcul, on doit utiliser davantage d'espace mémoire.

Exemple : échange de deux valeurs entières

```
// échange des valeurs de deux variables
entier x, y, z;
... // initialisation de x et y
z ← x;
x ← y;
y ← z;
```

```
// échange des valeurs de deux variables
entier x, y;
... // initialisation de x et y
x ← y-x;
y ← y-x;
x ← y+x;
```

Calcul du coût d'un algorithme

L'évaluation de la **complexité** peut se faire à plusieurs niveaux :

- au niveau de l'exécution du programme expérimentalement
- au niveau purement algorithmique, par l'analyse et le calcul

Calcul du coût d'un algorithme

Méthode expérimentale :

- La solution la plus intuitive pour choisir la meilleure solution consiste a mesuré le temps d'exécution exacte pour chaque algorithme.
- Pour ce faire, python offre le module time qui permet de calculer le temps d'exécution.

```
from time import time

t0 = time( )*1000
tri_fusion(L)
t1 = time( )*1000
print(t1-t0)
```

EN PYTHON

```
clock_gettime(CLOCK_MONOTONIC, &t0);
tri_fusion(L, n);
clock_gettime(CLOCK_MONOTONIC, &t1);

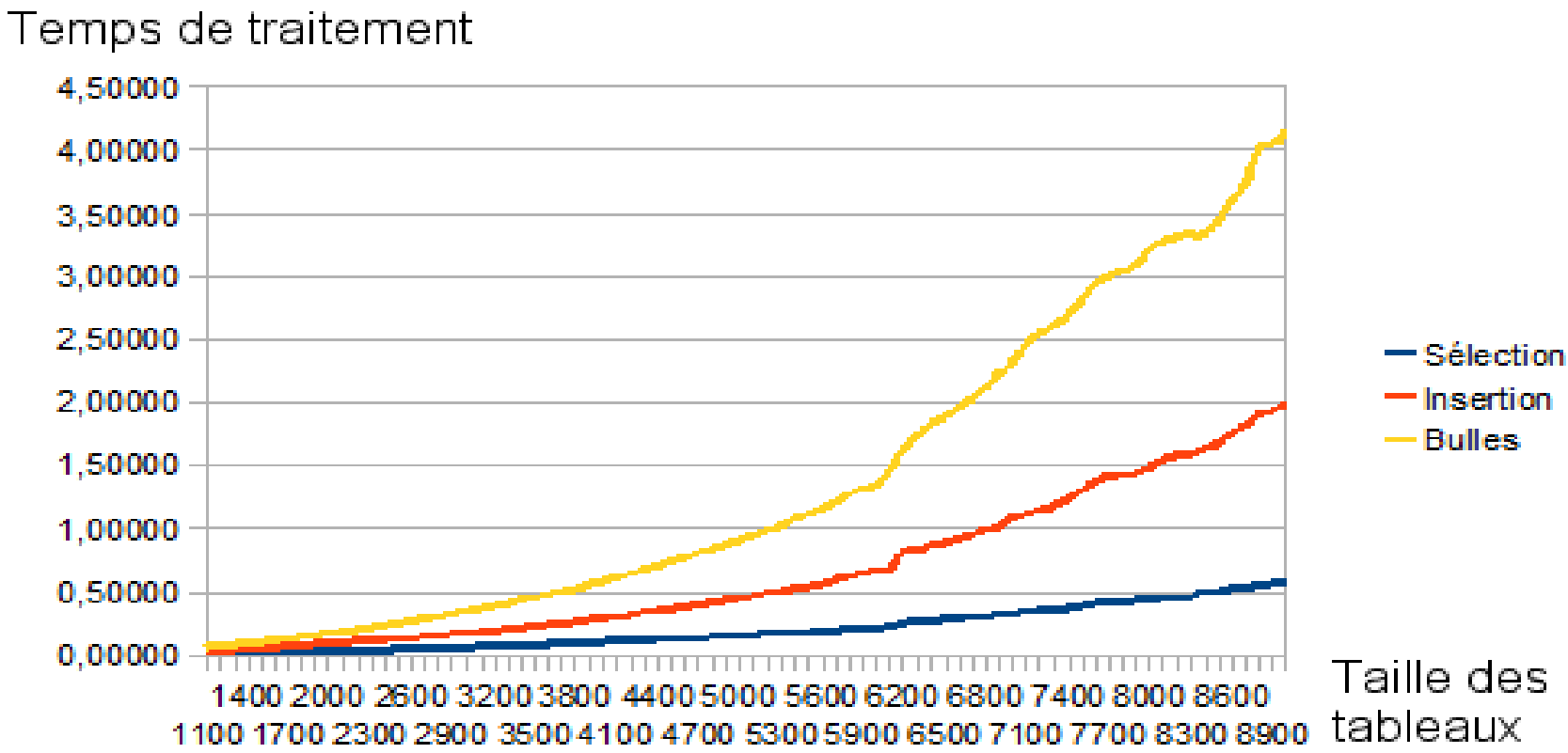
ms =  (t1.tv_sec - t0.tv_sec) * 1000LL +
      (t1.tv_nsec - t0.tv_nsec) / 1000000LL;
printf("%lld ms\n", ms);
```

EN Langage C

Calcul du coût d'un algorithme

Méthode expérimentale

Il est possible d'évaluer de façon expérimentale le temps d'exécution des programmes.



Calcul du coût d'un algorithme

Méthode expérimentale

Inconvénients :

- Implémentation de l'algorithme
- Résultats dépends de l'environnements.
- Lancer plusieurs exécutions avec des tailles de données différents: beaucoup de temps

Calcul du coût d'un algorithme

Méthode expérimentale

⇒**Environnement de simulation**

- Processeur Intel Core i5-4210U CPU 1,70GHz x 4
- Mémoire vive 8Go
- OS : Linux Ubuntu 14.04 LTS 64bits

⇒**Résultats de l'expérience**

N	10^2	10^3	10^4	10^5	10^6	10^7
Tri fusion	0,3ms	0,9ms	5,5ms	20ms	215ms	2,3s
Tri insertion	1,8ms	67ms	5,7ms	11mn	19h	66j



Méthode théorique

Avantages :

- Comparer l'efficacité des algorithmes théoriquement
- Pas besoin d'implémentation
- Taille des exemplaires n'est pas une contrainte

Calcul du coût d'un algorithme

Etapes de calcul de la complexité

1. Identifier le(s) paramètre(s) de complexité de l'algorithme

2. Calculer le coût en fonction du paramètre de la complexité

3. Calculer la complexité asymptotique

Calcul du coût d'un algorithme

Paramètre de complexité (1/5)

- Le paramètre de la complexité est ce qui va faire varier le temps d'exécution de l'algorithme
- Exemple : la calcul de la factorielle

```
long factoriel(int n) {
    long resultat = 1;
    for (int i = 2; i <= n; i++) {
        resultat *= i;
    }
    return resultat;
}
```

Le paramètre de complexité est la valeur de n

Paramètre de complexité (2/5)

Exemple : multiplication des éléments d'une liste par un nombre

```
void multiplier(int tab[], int n, int nombre) {
    for (int i = 0; i < n; i++) {
        tab[i] *= nombre;
    }
}
```

Le paramètre de complexité est n la taille du tableau

Paramètre de complexité (3/5)

Exemple : somme de la i ème ligne d'une matrice

```
int sommeLigne(int matrice[][], int i, int nbColonnes) {
    int somme = 0;

    for (int j = 0; j < nbColonnes; j++) {
        somme += matrice[i][j];
    }
    return somme;
}
```

Le paramètre de complexité est nbColonnes

Paramètre de complexité (4/5)

Exemple : table de multiplication d'un entier

```
void tableMultiplication(int n) {
    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", n, i, n * i);
    }
}
```

Pas de paramètre de complexité

Calcul du coût d'un algorithme

Paramètre de complexité (5/5)

Exemple : produit de deux matrices

Paramètres de complexité :
n = Nombre de lignes de A
m = Nombre de colonnes de A
p= Nombre de de colonnes de B

N = max(n, m, p)

```
void produitMatrices(int n, int m, int p,
                     int A[n][m],
                     int B[m][p],
                     int C[n][p]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            C[i][j] = 0;
            for (int k = 0; k < m; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 1 :

On appelle **opération de base**, ou **opération élémentaire**, toute :

- Affectation
- Test de comparaison : = =, <, ≤, ≥, !=
- Opération de lecture et écriture
- Opération arithmétique : +, -,*, /, %

Toute opération élémentaire a un coût qui égale à **1**, indépendamment

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 2 :

Le coût des instructions p et q en séquence est la somme des coûts de l'instruction p et de l'instruction q.

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 2 :

Exemple : Soit l'algorithme A

```
1  S=1        #Instruction 1
2  S=S*n      #Instruction 2
3  S=S-n      #Instruction 3
```

$T_A = T_{\text{instruction 1}} + T_{\text{instruction 2}} + T_{\text{instruction 3}} = 1 + 2 + 2 = 5$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 2 :

Exemple : Soit l'algorithme A

```
A=5;
B=A+5;
printf("S=%d", A+B);
```

$T_A = T_{\text{instruction 1}} + T_{\text{instruction 2}} + T_{\text{instruction 3}} = 1 + 2 + 2 = 5$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 3 :

Le coût d'un test **if (test) p else: q** est inférieur ou égal au maximum des coûts des instructions *p* et q, plus le temps d'évaluation de l'expression **test**.

$$T_A <= T_{test} + \max(T_q , T_{test})$$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 3 :

```
if (i % 2 == 0) {
    n = i / 2;
} else {
    i = i + 1;
    n = i / 2;
}
```

$T_A <= T_{test} + Max(T_{block\ 1} , T_{block\ 2}) = 2 + Max(2 , 4) = 6$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 4 :

Le coût d'une boucle **for** : le coût est égal au nombre d'éléments de l'itérable multiplié par le coût de l'instruction p si ce dernier ne dépend pas de la valeur de i. Quand le coût du corps de la boucle dépend de la valeur de i, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de i.

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 4 : Exemple

```
for(int i=a; i<b; i++){
    Q
}
```

$$T_A = \sum_{i=a}^{b-1} T_Q$$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 4 : Exemple

```
for (int i = 0; i < 10; i++) {
    scanf("%d", &a);
    printf("%d\n", a * 5);
}
```

$T_A = 10 \times 3 = 30$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 4 : Exemple

```
int S = 0;

for (int i = 0; i < n; i++) {
    S = S + 1;
}
```

$$\mathbf{T}(\boldsymbol{n}) = \boldsymbol{2n} + \mathbf{1}$$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 4 : Exemple

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        S = i + j;
    }
    printf("%d\n", S);
}
```

$$T(n) = \sum_{i=0}^{n-1} T_q = \sum_{i=0}^{n-1}(1 + \sum_{j=0}^{i-1} 3) = \sum_{i=0}^{n-1}(1 + 3i) = n + 3\sum_{i=0}^{n-1} i = 3\,\frac{n(n-1)}{2} + n$$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 5 :

Le cas des boucles **while** est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu a priori. On peut majorer le nombre de répétitions de la boucle de la même façon qu'on démontre sa terminaison et ainsi majorer le coût de l'exécution de la boucle.

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 5 :

```
while test :
    Q
```

$$T_A = \sum(T_{test} + T_Q)$$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 5 : Exemple

```
int i = 1;
int somme = 0;

while (i <= n) {
    somme = somme + i;
    i = i + 1;
}
```

$$T_A = 2 + (n+1) + \sum_{i=1}^{n} 4 = 2 + (n+1) + 4n = 5n + 3$$

Calcul du coût d'un algorithme

Un modèle pour calculer le coût d'un algorithme:

Règle 6 : Exemple

Lorsqu'une fonction ou une procédure est appelée, le cout cet appel est le nombre total d'opérations élémentaires et composées engendrées par l'appel de cette fonction

Calcul du coût d'un algorithme

Exercice : Calculer le coût des instructions suivantes

Programme	Coût
for (int i=0; i<10; i++) printf("%d", i * n)	
for (int i=0; i<n; i++) printf("%d", i * i)	

for (int i=0; i<n; i++) {
for (int j=0; j<n; j++)
printf("%d", i * j)
printf("\n")
}	

Programme	Coût

int i=1;
int s=0;
while (i <= n){
s=s+i;
i= i+1;
}
printf("%d", s)	

int i=1;
while (i <= n){
printf("%d", i);
i=i+2;
}	

Calcul du coût d'un algorithme

Nuances de la complexité temporelles

Lorsque, pour une valeur donnée du paramètre de complexité, le temps d'exécution varie selon les données d'entrée, on peut distinguer :

- La complexité au pire : temps d'exécution maximum, dans le cas le plus défavorable.
- La complexité au mieux : temps d'exécution minimum, dans le cas le plus favorable (en pratique, cette complexité n'est pas très utile).
- La complexité moyenne : temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution.

Calcul du coût d'un algorithme

Nuances de la complexité temporelles

Exemple :

```
bool rechercherElement(int L[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (L[i] == x) {
            return true;
        }
    }

    return false;
}
```

Le coût au mieux:
2

Le coût au pire :
2n+1

Nuances de la complexité temporelles

La complexité en moyenne :

La complexité en moyenne d'un algorithme est une mesure qui tient compte de la performance d'un algorithme sur un ensemble de cas d'entrée possibles, en prenant en considération les probabilités d'apparition de ces différents cas.

Complexité en moyenne = Σ (Complexité temporelle du cas i * Probabilité du cas i) pour tous les cas i

$$C_A^{moy}(n) = \sum_{d \in \mathcal{D}_{A,n}} p(d).C_A(d)$$

Nuances de la complexité temporelles

La complexité en moyenne : Exemple

- **Complexité moyenne :** Soit q la probabilité que X est dans le tableau T, alors :
  - La probabilité que X se trouve dans la i^{ème} case est $\frac{q}{n}$ puisqu'il peut se trouver dans n'importe quelle case avec la même probabilité
  - La probabilité que X ne soit pas dans le tableau T est (1 - q)
- Le cout dépend des nombre de comparaisons effectuées :
  - Si X est dans la i^{ème} case de T alors on effectuera « i » comparaisons
  - Si X n'est pas dans T alors on effectuera « n » comparaisons
- Donc : $T(\mathbf{n}) = \sum_{\mathbf{i=1}}^{\mathbf{n}} \frac{\mathbf{q}}{\mathbf{n}}\,\mathbf{i} + \mathbf{n}(\mathbf{1}-\mathbf{q}) = \frac{\mathbf{q}}{\mathbf{n}}\frac{\mathbf{n(n+1)}}{\mathbf{2}} + \mathbf{n}(\mathbf{1}-\mathbf{q}) = \mathbf{q}\frac{\mathbf{(n+1)}}{\mathbf{2}} + \mathbf{n}(\mathbf{1}-\mathbf{q})$
- Si on est sur que X appartient au tableau T (q=1) alors : $T(\mathbf{n}) = \frac{\mathbf{n+1}}{\mathbf{2}}$

Complexité asymptotique : La notation O

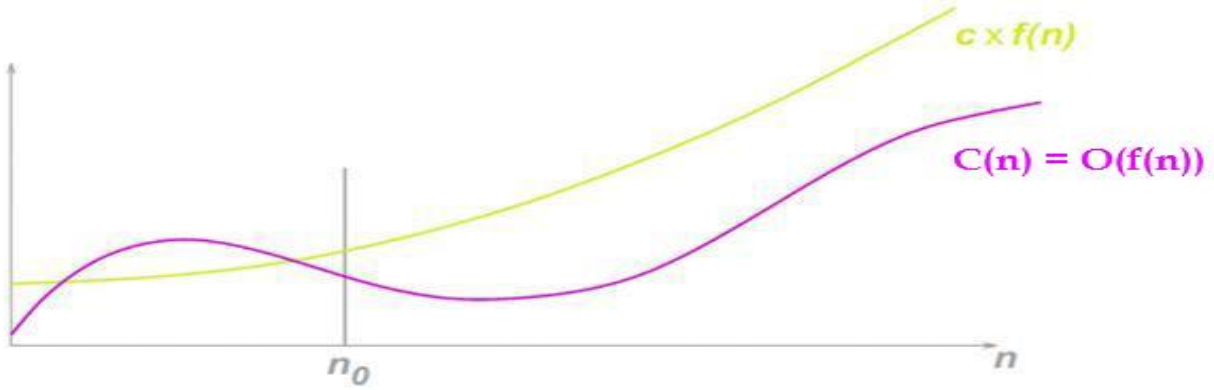
Ce qui nous intéresse n'est pas un temps précis d'exécution mais un ordre de grandeur de ce temps d'exécution en fonction de la taille des données.

- **Première approximation :** Considérer la complexité au pire des cas
- **Deuxième approximation :** Considérer le calcule asymptotique de la complexité, s'intéresser juste au grandes quantités des données.

Calcul du coût d'un algorithme

Complexité asymptotique : La notation O

- Soit T(n) une fonction qui désigne le temps de calcul d'un algorithme A.
- On dit que T(n) est en grand O de f(n) : T(n) = O(f (n)) si et seulement
- si : ∃(n0; c) telle que T(n) <= c*f (n) pour tous n >= n0



La notation O(f (n)) est aussi appelée Notation de Landau : (Complexité asymptotique) (i.e.quand n → ∞)

La complexité asymptotique

On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :

- On oublie les constantes multiplicatives (elles valent 1) ;
- On annule les constantes additives
- On ne retient que les termes dominants ;

Soit un algorithme effectuant $T(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

- => On remplace les constantes multiplicatives par 1 : $n^3 - n^2 + n + 3$
- => On annule les constantes additives : $n^3 - n^2 + n$
- => On garde le terme de plus haut degré : n^3
- Donc : $T(n) = O(n^3)$:

La complexité asymptotique

Notation O : Propriétés

- c *O(f) = O(f)
- O(f) + O(g) = O(f + g) = O(max(f, g))
- O(f) * O(g) = O(f * g)
- f = O(g) et g = O (h) alors f = O (h) Transivité

Les classes de complexité

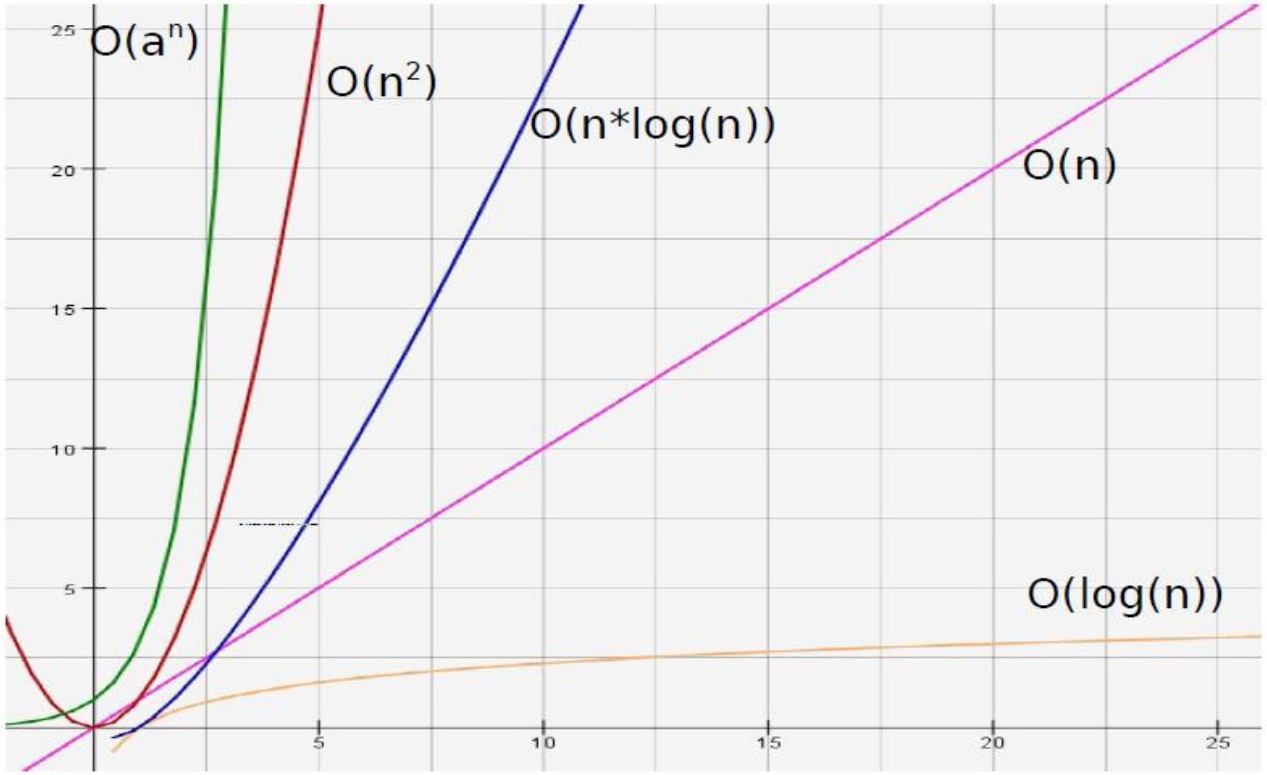
Classes de la complexité

On voit souvent apparaitre les complexités suivantes :

Complexité	Nom courant	Description
$O(1)$	constante	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare !
$O(log(n))$	logarithmique	augmentation très faible du temps d'exécution quand le paramètre croit.
$O(n)$	linéaire	augmentation linéraire du temps d'exécution quand le paramètre croit.
$O(nlog(n))$	quasi-linéaire	augmentation un peu supérieure à O(n)
$O(n^2)$	quadratique	quand le paramètre double, le temps d'exécution est multiplié par 4. Exemple : algorithmes avec deux boucles imbriquées.
$O(n^k)$	polynômiale	ici, n^k est le terme de plus haut degré d'un polynôme en n ; il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(k^n)$	exponentielle	quand le paramètre double, le temps d'exécution est élevé à la puissance k avec $k > 1$.
$O(n!)$	factorielle	asymptotiquement équivalente à n^n

Les classes de complexité

Classes de la complexité



Exercices

```
int factoriel(int n) {
    int f = 1;

    for (int i = 2; i <= n; i++) {
        f = f * i;
    }

    return f;
}
```

n est le paramètre de complexité

T(n) = 1 + (n-1) * 2 + 1 = 2n

Complexité linéaire : O(n)

Exercices

```
int factoriel(int n) {
    int f = 1;

    for (int i = 2; i <= n; i++) {
        f = f * i;
    }

    return f;
}
```

n est le paramètre de complexité

O(1)

n-1 boucle donc : O(n)

O(1)

O(1)

Complexité linéaire : O(1) + O(n)*O(1) + O(1) = O(n)

Exercices

```
int minimum(int L[], int n) {

    int min = L[0];

    for (int i = 0; i < n; i++) {
        if (L[i] < min) {
            min = L[i];
        }
    }

    return min;
}
```

n = len(L) est le paramètre de complexité

T(n) = 2 + n(2) + 1 = 2n + 3

Complexité linéaire : O(n)

Exercices

```
int minimum(int L[], int n) {

    int min = L[0];

    for (int i = 0; i < n; i++) {
        if (L[i] < min) {
            min = L[i];
        }
    }

    return min;
}
```

```
# n = len(L) est le paramètre de complexité

# O(1)

# n boucles donc : O(n)

# O(1)
# O(1)


# O(1)
```

Complexité linéaire : O(n)

Exercices

```
void tableMultiplication(int n) {
    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", n, i, n * i);
    }
}
```

pas de paramètre de complexité

T(n) = 10 x 4 = 40

Complexité constante : O(1)

Exercices

```c
void tableMultiplication(int n) {
    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", n, i, n * i);
    }
}
```

```
# pas de paramètre de complexité
# 10 boucle donc : O(1)
# O(1)
```

Complexité constante : O(1)

Exercices

Somme des éléments de la diagonale d'une matrice carrée :

```
int sommeMat1(int A[][100], int n) {
    int s = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                s = s + A[i][j];
            }
        }
    }

    return s;
}
```

Complexité quadratique : $O(n^2)$

```
int sommeMat2(int A[][100], int n) {
    int s = 0;

    for (int i = 0; i < n; i++) {
        s = s + A[i][i];
    }

    return s;
}
```

Complexité linéaire : O(n)

Exercices

```
int rechDichotomique(int T[], int n, int x) {
    int inf = 0;
    int sup = n - 1;

    while (inf <= sup) {
        int m = (inf + sup) / 2;

        if (T[m] == x) {
            return m;
        }

        if (T[m] < x) {
            inf = m + 1;
        } else {
            sup = m - 1;
        }
    }

    return -1;
}
```

Complexité logarithmique : O(log(n))

Exercices

```
bool rechercheDoublons(int T[], int n) {

    for (int i = 0; i < n - 1; i++) {

        for (int j = i + 1; j < n; j++) {

            if (T[i] == T[j]) {
                return true;
            }
        }
    }

    return false;
}
```

Complexité logarithmique : $O(n^2)$

Au pire des cas: pas de doublons ➔Comparer tous les éléments entre eux

$$\left(\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 2\right) + 1 = 1 + 2\sum_{i=0}^{n-2}(n-1-i) = \boldsymbol{O(n^2)}$$

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

La récursivité est le fait pour une méthode de s'appeler elle même. On parle alors de méthode récursive.

Exemple : Le calcul de la factorielle de N.

N != N*(N-1)*(N-2)*…*2*1 , on peut écrire ainsi N != N*(N-1)!

Pour calculer N! il faut savoir calculer (N-1) ! et pour calculer (N-1) ! Il faut savoir calculer (N-2) ! et ainsi jusqu'à 1! qui est égal à 1 et qui permet à la récursivité de s'arrêter après une série d'auto appels.

-> Il est impératif donc de prévoir une condition d'arrêt à la récursion sinon le programme ne s'arrête jamais!

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

Récursivité simple :

```
int factoriel(int n) {
    if (n == 0) {
        return 1;
    }

    return n * factoriel(n - 1);
}
```

Récursivité multiple :

```
int combinaison(int n, int p) {
    if (p == 0 || n == p) {
        return 1;
    }

    return combinaison(n - 1, p)
         + combinaison(n - 1, p - 1);
}
```

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

```
int factoriel(int n) {
    if (n == 0) {
        return 1;
    }

    return n * factoriel(n - 1);
}
```

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n-1) + 5, & sinon \end{cases}$$

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n-1) + 5, & sinon \end{cases}$$

La complexité d'un algorithme récursif se fait par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche.

T(n) = T(n-1) + b
= T(n-2) + 2*b
= T(n-3) + 3*b
= T(n-4) + 4*b
= ....
= T(n-i) + i*b
= T(n-i+1) + (i+1)*b
...
= T(n-n) + n*b = T(0) + n*b = 5n + 2 *=> donc complexité en O(n)*

Complexité des fonctions récursives

```
int somme(int n) {
    if (n == 0) {
        return 0;
    }

    return n + somme(n - 1);
}
```

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n-1) + 5, & sinon \end{cases}$$

T(n) = 5n + 2 donc complexité en O(n)

Complexité des fonctions récursives

```
int rechDichotomiqueRec(int T[], int x, int debut, int fin)
{

    if (debut > fin) {
        return -1;
    }

    int m = (debut + fin) / 2;

    if (T[m] == x) {
        return m;
    }

    if (T[m] < x) {
        return rechDichotomiqueRec(T, x, m + 1, fin);
    } else {
        return rechDichotomiqueRec(T, x, debut, m - 1);
    }
}
```

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n/2) + 9, & sinon \end{cases}$$

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

$$T(n) = \begin{cases} 2, & n = 0 \\ T(n/2) + 9, & sinon \end{cases}$$

La complexité d'un algorithme récursif se fait par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche.

T(n) = T(n/2) + b
= T(n/4) + 2*b
= T(n/8) + 3*b
= T(n/16) + 4*b
= ....
= $T(n/2^i) + i*b$
...
= T(1) + log(n)*b = 9 log(n) + 5 *=> donc complexité en O(log(n))*

Complexité des fonctions récursives

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

$$T(n) = \begin{cases} 2, & n = 0 \\ 2, & n = 1 \\ T(n-1) + T(n-2) + 5, & sinon \end{cases}$$

$O(2^n)$

Complexité des fonctions récursives

Equation : $c(n) = c(n-1) + b$

Solution : $c(n) = c(0) + b*n = O(n)$

Exemples : factorielle, recherche séquentielle récursive dans un tableau

Equation : $c(n) = a*c(n-1) + b$, $a \neq 1$

Solution : $c(n) = a^n*(c(0) - b/(1-a)) + b/(1-a) = O(a^n)$

Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif

Equation : $c(n) = c(n-1) + a*n + b$

Solution : $c(n) = c(0) + a*n*(n+1)/2 + n*b = O(n^2)$.

Exemples : traitement en coût linéaire avant l'appel récursif, tri à bulle

Equation : $c(n) = c(n/2) + b$

Solution : $c(n) = c(1) + b*\log_2(n) = O(\log(n))$

Exemples : élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive

Complexité des fonctions récursives

Equation : $c(n) = a*c(n/2) + b$, $a \neq 1$

Solution : $c(n) = n^{log2(a)} * (c(1) - b/(1-a)) + b/(1-a) = O(n^{log2(a)})$

Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique

Equation : $c(n) = c(n/2) + a*n + b$

Solution : $c(n) = O(n)$

Exemples : traitement linéaire avant l'appel récursif dichotomique

Equation : $c(n) = 2*c(n/2) + a*n + b$

Solution : $c(n) = O(n*log(n))$

Exemples : traitement linéaire avant double appel récursif dichotomique, tri fusion

Complexité des fonctions récursives

```
void triBulle(int L[], int n) {

    for (int i = n - 1; i > 0; i--) {

        for (int j = 0; j < i; j++) {

            if (L[j] > L[j + 1]) {
                int temp = L[j];
                L[j] = L[j + 1];
                L[j + 1] = temp;
            }
        }
    }
}
```

Paramètre de complexité : n=len(L)

C(n) $= \sum_{i=1}^{n-1} 6i + 1$

$= 6 \sum_{i=1}^{n-1} i + 1$

$= 6(n-1)(n)/2 + 1$

$= 3n^2 - 3n + 1$

Complexité quadratique: $O(n^2)$

Complexité des fonctions récursives

```
void triSelection(int T[], int n) {

    for (int i = 0; i < n - 1; i++) {

        int posmin = i;

        for (int j = i + 1; j < n; j++) {
            if (T[j] < T[posmin]) {
                posmin = j;
            }
        }

        int temp = T[i];
        T[i] = T[posmin];
        T[posmin] = temp;
    }
}
```

Paramètre de complexité : n=len(T)

T(n) $= \sum_{i=0}^{n-2}(3 + (\sum_{j=i+1}^{n-1} 2))$

$= \sum_{i=0}^{n-2} 3 + \sum_{i=0}^{n-2}(\sum_{j=i+1}^{n-1} 2)$

$= 3(n-1) + \sum_{i=0}^{n-2}(\sum_{j=1}^{n-i-1} 2)$

$= 3(n-1) + 2\sum_{i=0}^{n-2}(n-i-1)$

$= 3(n-1) + 2(\sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1)$

$= 3(n-1) + 2(n(n-1) - (n-1)(n)/2 - (n-1))$

$= n^2 - 5n - 2$

Complexité quadratique: $O(n^2)$

Complexité des fonctions récursives

```
void triInsertion(int T[], int n) {

    for (int i = 1; i < n; i++) {

        int k = i;

        while (k > 0 && T[k] < T[k - 1]) {

            int temp = T[k];
            T[k] = T[k - 1];
            T[k - 1] = temp;

            k = k - 1;
        }
    }
}
```

Paramètre de complexité : n=len(T)

$T(n) = \sum_{i=1}^{n-1}(1 + 10i)$

$T(n) = (n-1) + 10((n-1)(n)/2)$

$T(n) = 5n^2 - 4n - 6$

Complexité quadratique: $O(n^2)$

Algorithmes de Tri avancés

Plan

- Introduction
- Tri par fusion (Merge Sort)
  - Pseudo-code et illustration de l'algorithme
  - Avantages et inconvénients
- Tri rapide (Quick Sort)
  - Pseudo-code et illustration de l'algorithme
  - Avantages et inconvénients
- Autres algorithmes : Count Sort, Radix Sort, Tim, Shell...

Introduction

Définition :

Un algorithme de tri est en Informatique ou en mathématique, un algorithme qui permet de trier une collection d'objets selon un ordre bien déterminée. Les objets à trier font donc parti d'un ensemble muni d'une relation d'ordre.

Introduction

La vitesse de tri est un facteur critique dans de nombreuses applications :

- **Bases de Données:** Dans les systèmes de gestion de bases de données, le tri est essentiel pour exécuter des requêtes efficacement.
- **Traitement d'Images:** Les applications de traitement d'images, telles que la recherche d'images similaires ou la segmentation d'images, peuvent tirer parti de l'utilisation d'algorithmes de tri avancés
- **Algorithmes de Machine Learning:** Lors de la préparation de données pour des modèles d'apprentissage automatique, le tri peut être utilisé pour organiser les caractéristiques ou les échantillons.
- …

Introduction

Algorithmes de tri quadratiques:

Tri à bulles:

1. Parcourez la liste d'éléments à trier, en comparant chaque élément avec l'élément adjacent à sa droite.
2. Si l'élément actuel est plus grand (dans le cas d'un tri croissant) que l'élément adjacent, échangez-les.
3. Continuez à parcourir la liste en répétant les comparaisons et les échanges jusqu'à ce qu'il n'y ait plus d'échanges nécessaires lors d'un passage complet à travers la liste.
4. Répétez cette opération jusqu'à ce que la liste soit entièrement triée.

Introduction

Algorithmes de tri quadratiques :

Tri à bulles:

```
void triBulle(int tab[], int n) {
    int i, j, temp;

    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - 1 - i; j++) {

            if (tab[j] > tab[j + 1]) {
                temp = tab[j];
                tab[j] = tab[j + 1];
                tab[j + 1] = temp;
            }
        }
    }
}
```

Nombre de comparaisons :

Pour i=0 : n-1

Pour i=1 : n-2

….

Pour i=n-2 : 1

Donc on a :

(n-1) + (n-2) + … + 1 = n(n-1)/2

Complexité quadratique: $O(n^2)$

Introduction

Algorithmes de tri quadratiques :

Tri par sélection:

1. On parcourt le tableau pour trouver la plus petite valeur.
2. On échange cette valeur avec l'élément situé à la première position.
3. On recommence le même processus à partir de la deuxième position, puis de la troisième, jusqu'à ce que le tableau soit entièrement trié.

Introduction

Algorithmes de tri quadratiques :

Tri par sélection:

```
void triSelection(int tab[], int n) {
    int i, j, min, temp;
    for (i = 0; i < n - 1; i++) {
        min = i;
        for (j = i + 1; j < n; j++) {
            if (tab[j] < tab[min]) {
                min = j;
            }
        }
        // Échange
        temp = tab[i];
        tab[i] = tab[min];
        tab[min] = temp;
    }
}
```

Nombre de comparaisons :

Pour i=0 : n-1

Pour i=1 : n-2

….

Pour i=n-2 : 1

Donc on a :

(n-1) + (n-2) + … + 1 = n(n-1)/2

Complexité quadratique: $O(n^2)$

Introduction

Algorithmes de tri quadratiques :

Tri par insertion:

1. Initialement, la première élément de la liste est considéré comme la partie triée (car un élément unique est toujours trié).
2. Ensuite, l'algorithme passe en revue la partie non triée de la liste, un élément à la fois.
3. À chaque étape, l'algorithme prend l'élément actuel de la partie non triée et le compare à chaque élément de la partie triée, de droite à gauche, pour trouver la position appropriée où l'élément actuel devrait être inséré.
4. Lorsque la position appropriée est trouvée, l'élément actuel est inséré à cet endroit, et les éléments suivants dans la partie triée sont décalés vers la droite pour faire de la place pour l'élément inséré.
5. Répétez ces étapes jusqu'à ce que tous les éléments de la partie non triée aient été insérés dans la partie triée.

Introduction

Algorithmes de tri quadratiques :

Tri par insertion:

```
void triInsertion(int tab[], int n) {
    int i, j, valeur;

    for (i = 1; i < n; i++) {
        valeur = tab[i];
        j = i - 1;

        while (j >= 0 && tab[j] > valeur) {
            tab[j + 1] = tab[j];
            j--;
        }

        tab[j + 1] = valeur;
    }
}
```

Complexité quadratique: $O(n^2)$

Introduction

Paradigme diviser pour régner :

Diviser: Si la taille des données est trop grande pour les traiter d'une façon directe alors les diviser en deux ou plusieurs sous-ensembles disjoints.

Appliquer récursivement: Utiliser l'approche diviser pour régner afin de résoudre les sous-problèmes associés aux sous-ensembles de données.

Régner: Prendre les solutions aux sous-problèmes et "fusionner" ces solutions afin d'obtenir la solution au problème initial.

Le cas de base pour la récursivité est le sous-problème trivial, souvent de taille 0 ou 1.

Tri par fusion

Principe :

Le principe de cet algorithme tend à adopter une formulation récursive :

✓On découpe les données à trier en deux parties plus ou moins égales

✓On trie les 2 sous-parties ainsi déterminées

✓On fusionne les deux sous-parties pour retrouver les données de départ

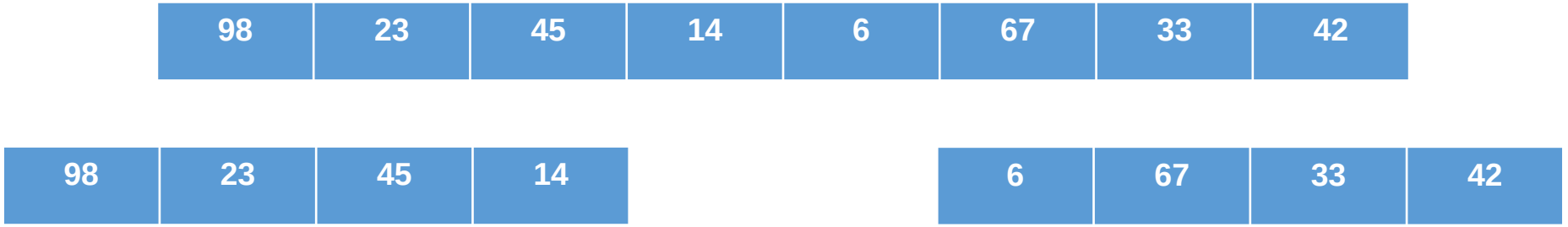
Tri par fusion

Exemple :

98	23	45	14	6	67	33	42

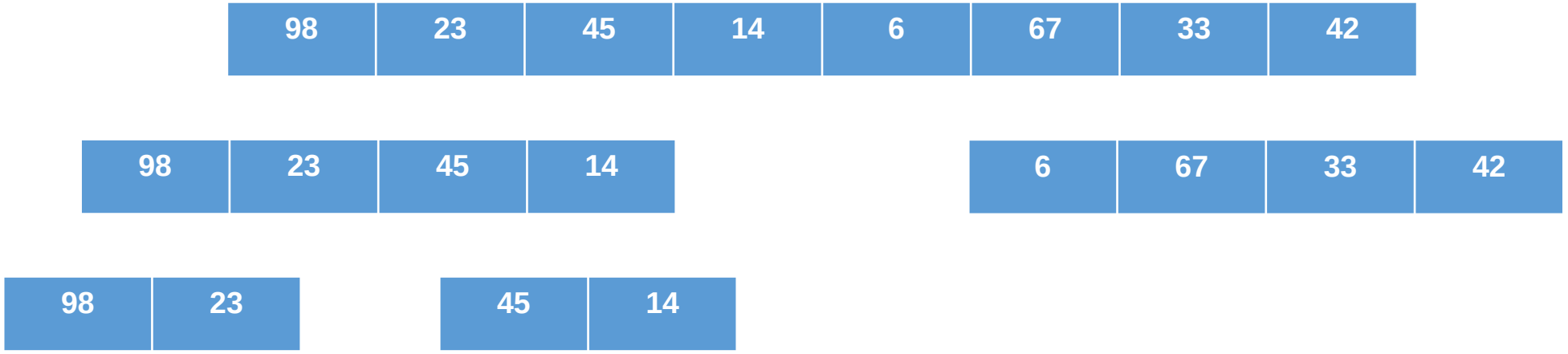
Tri par fusion

Exemple :



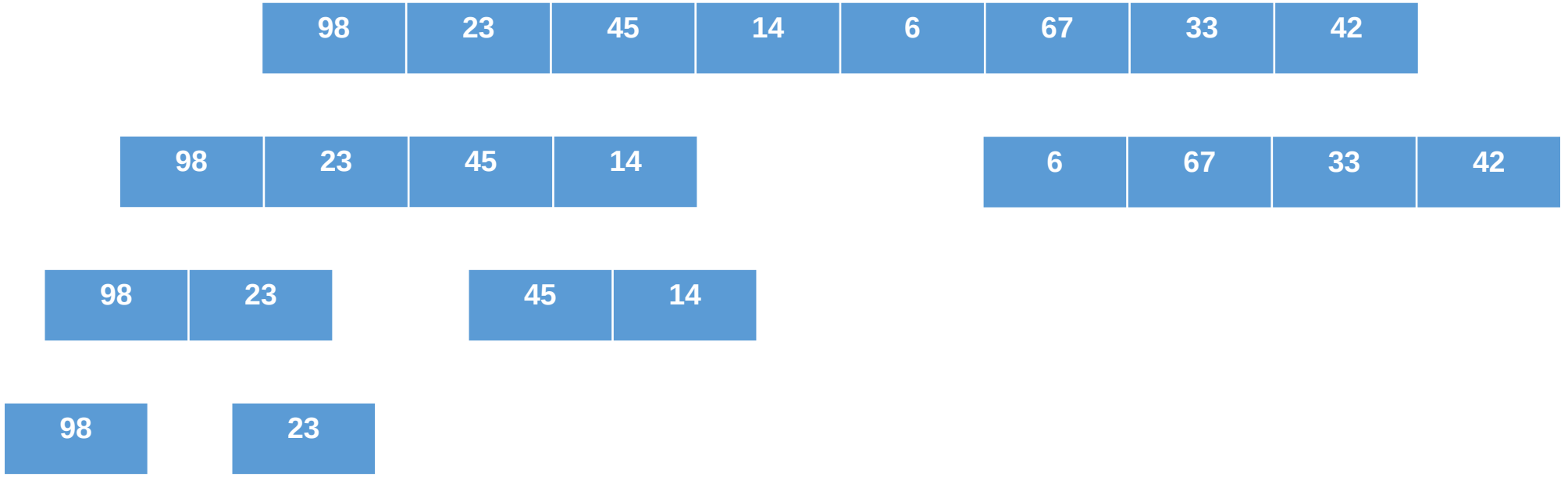
Tri par fusion

Exemple :



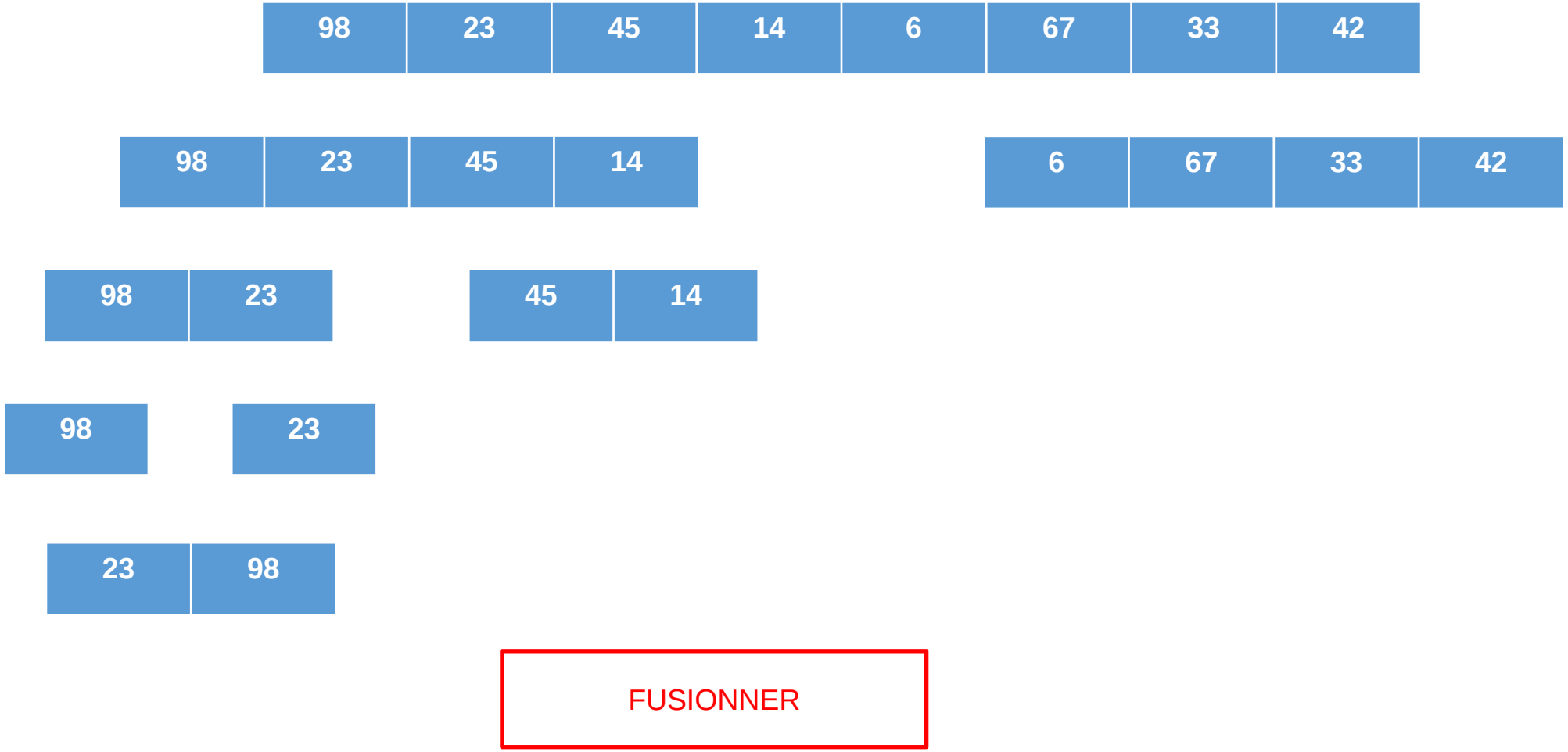
Tri par fusion

Exemple :



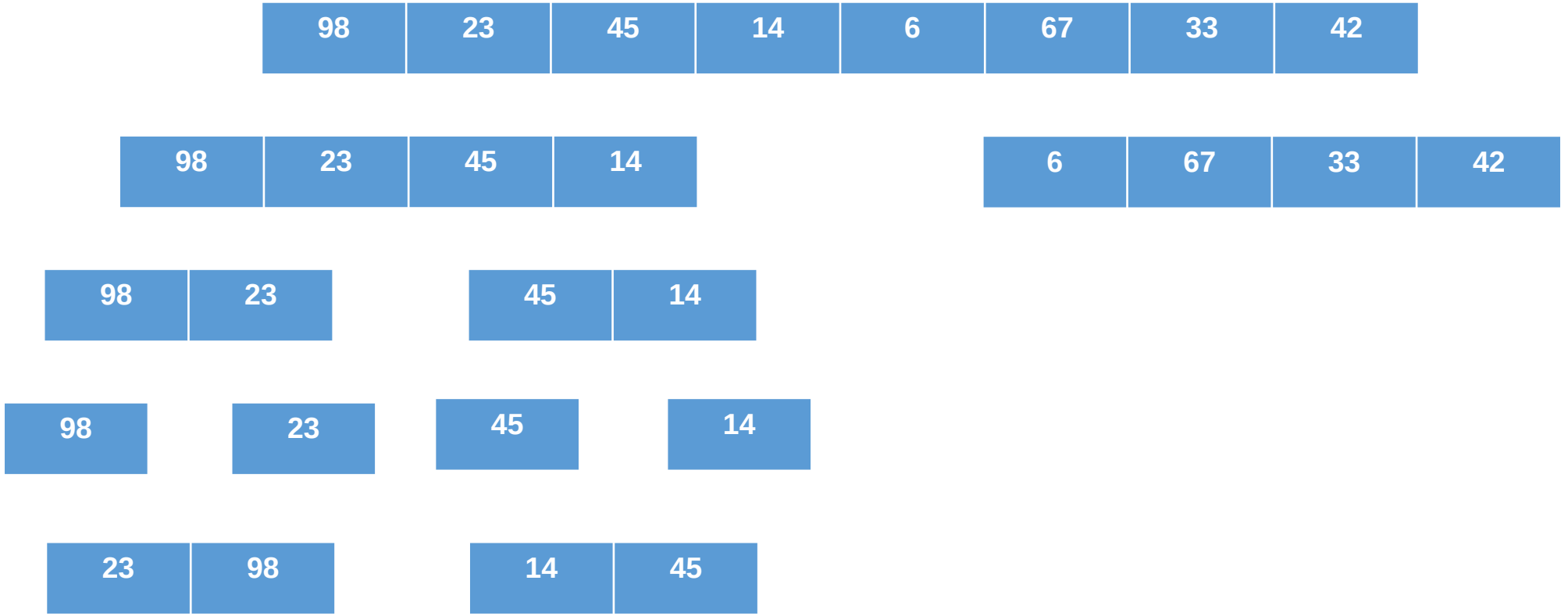
Tri par fusion

Exemple :



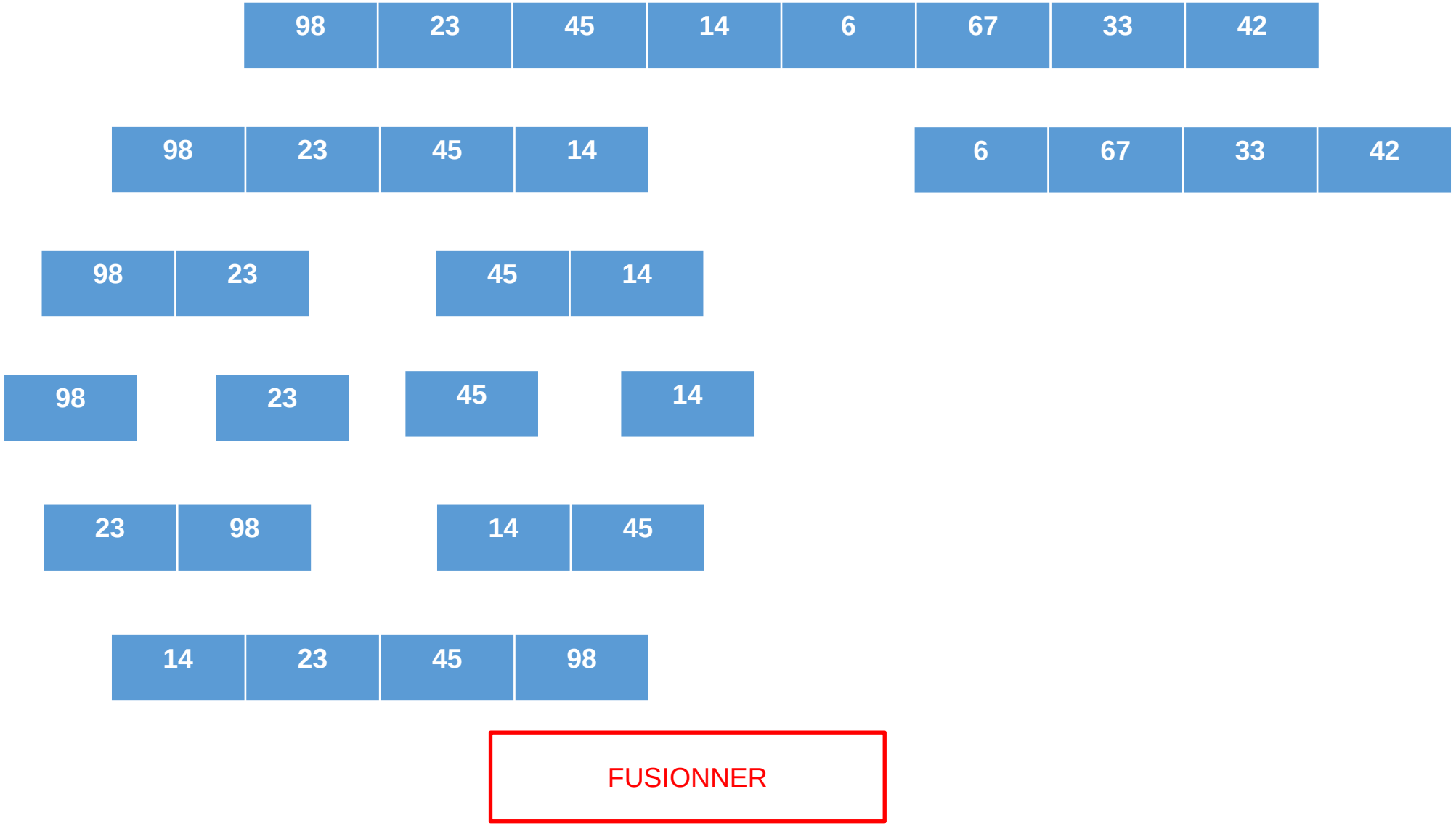
Tri par fusion

Exemple :



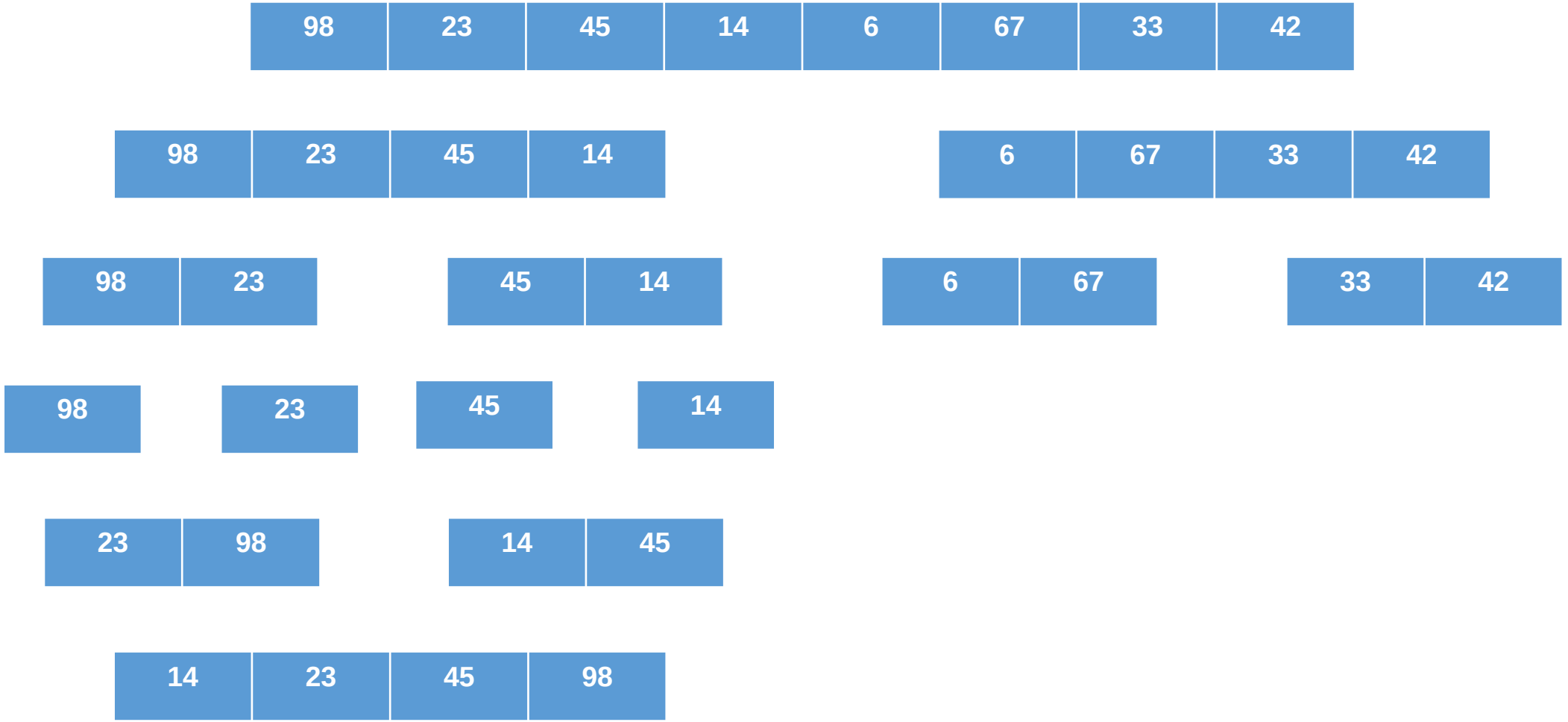
Tri par fusion

Exemple :



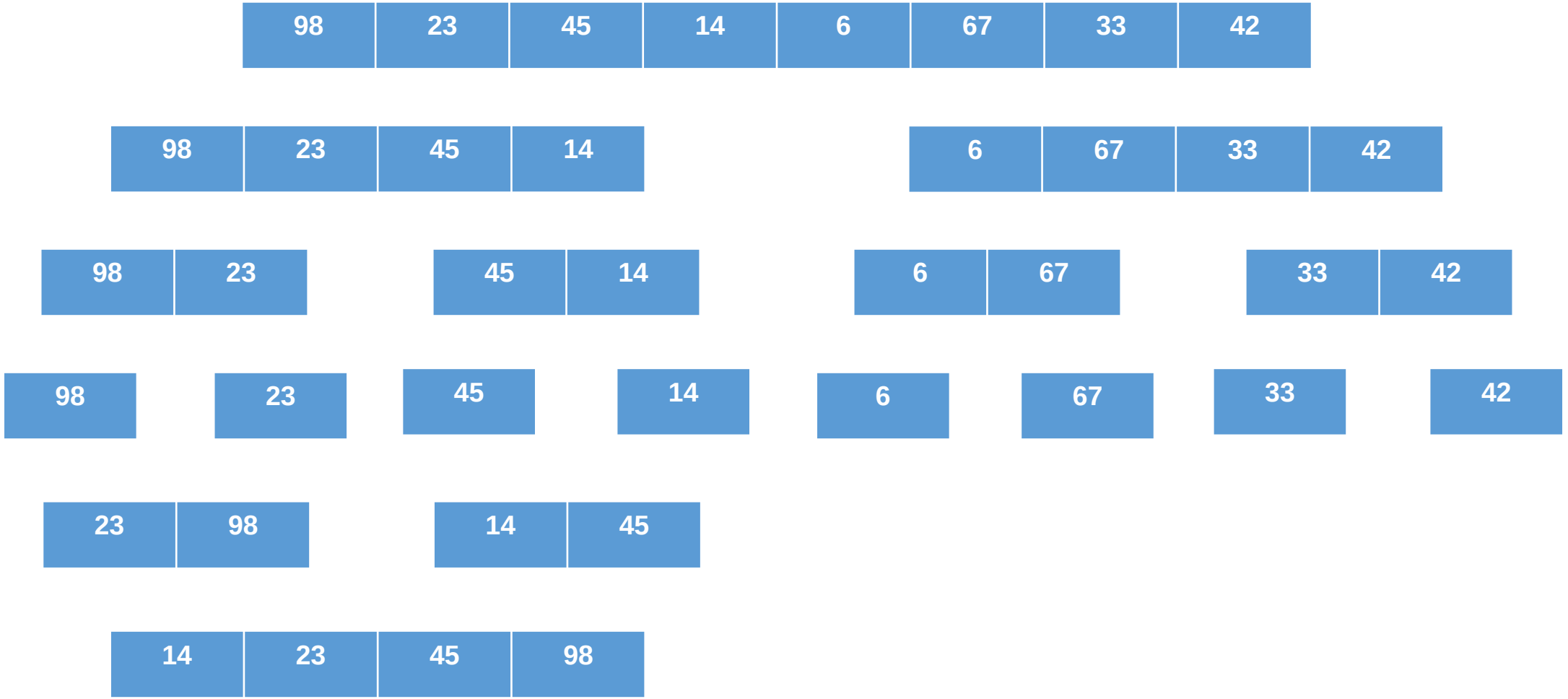
Tri par fusion

Exemple :



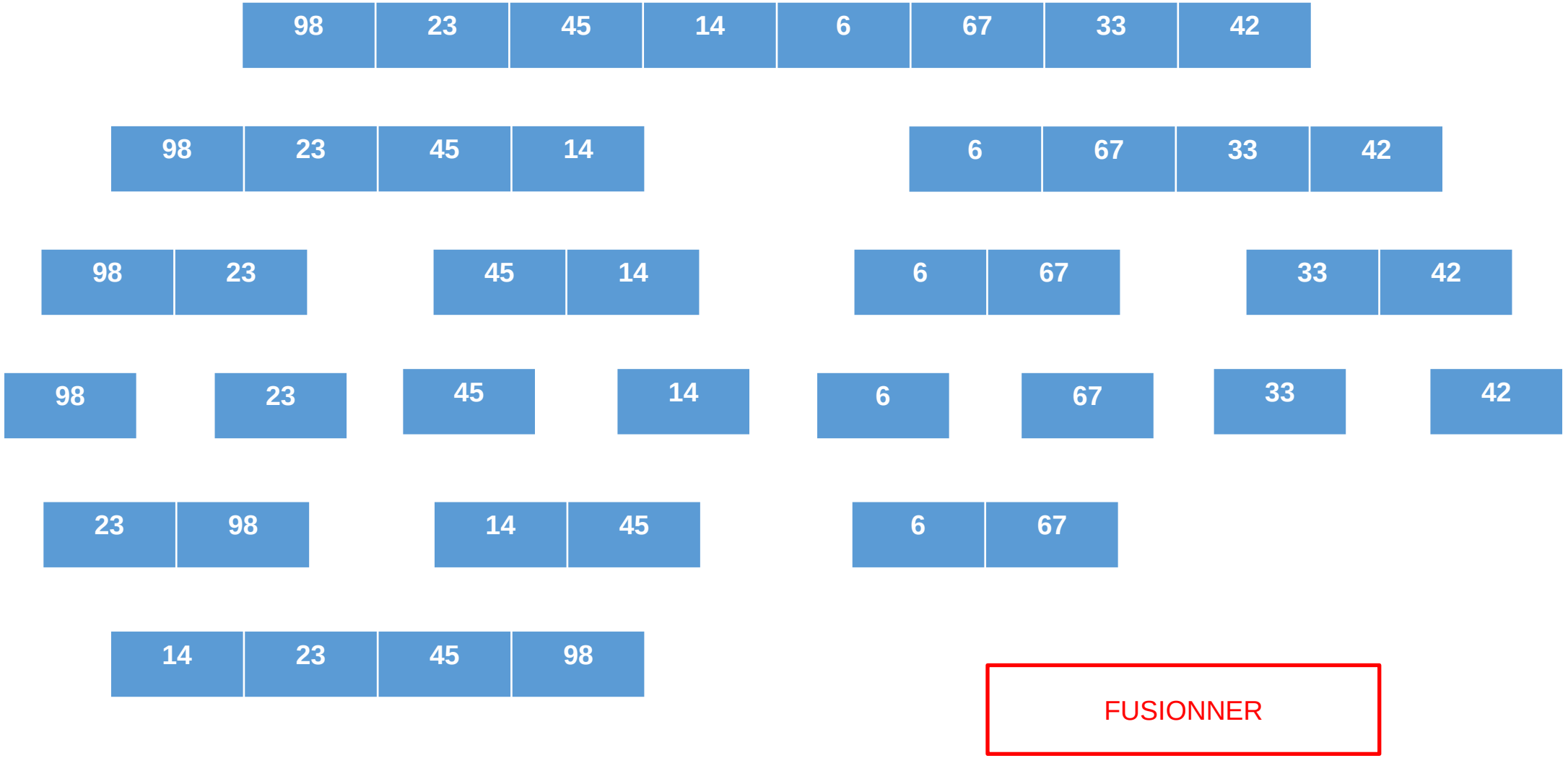
Tri par fusion

Exemple :



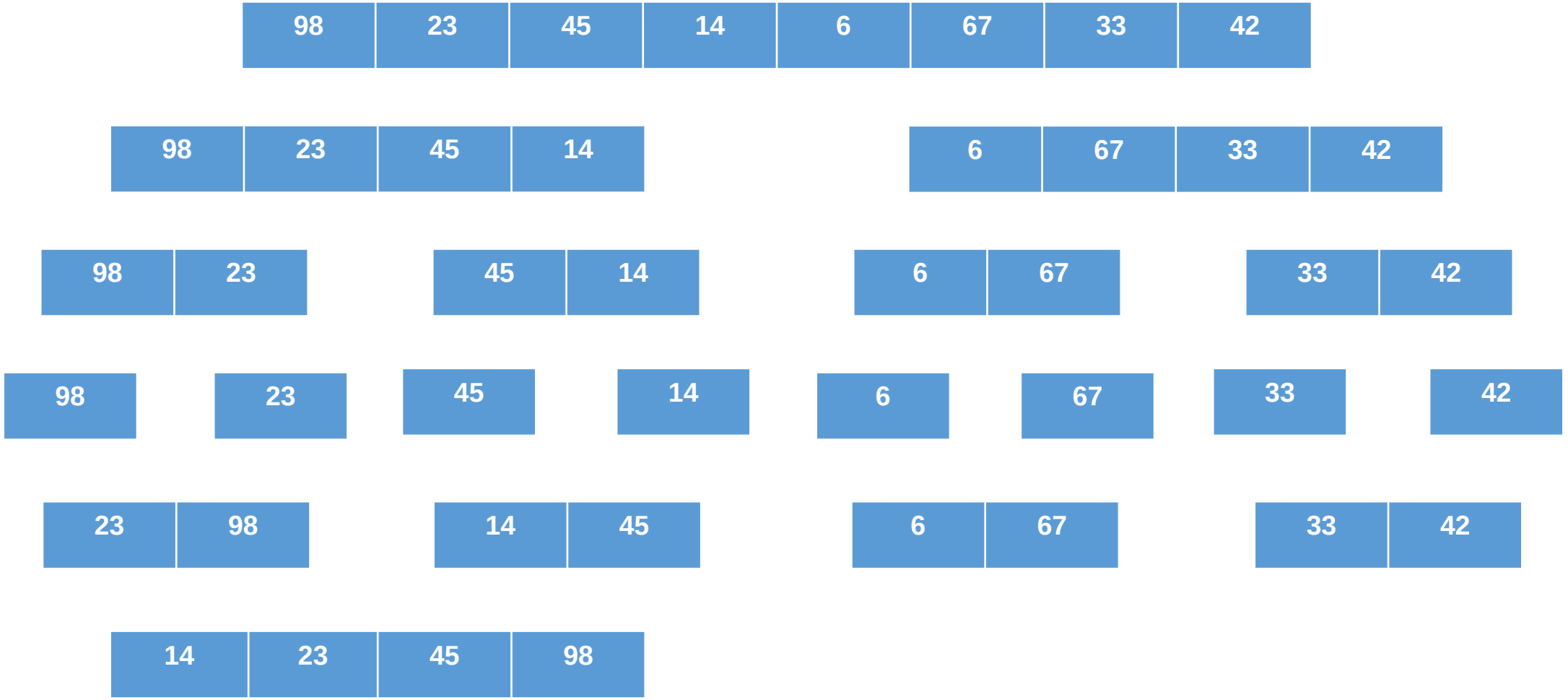
Tri par fusion

Exemple :



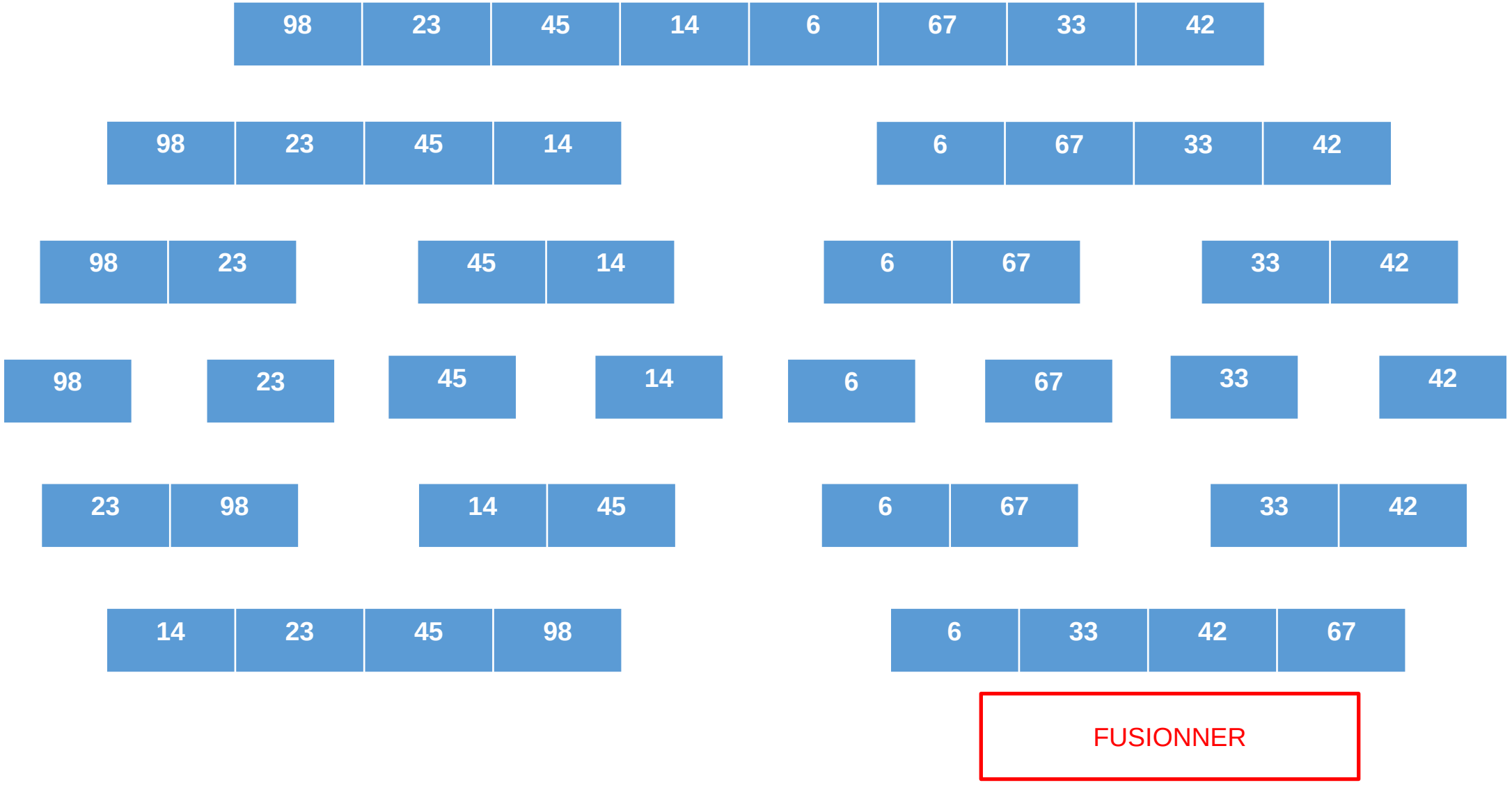
Tri par fusion

Exemple :



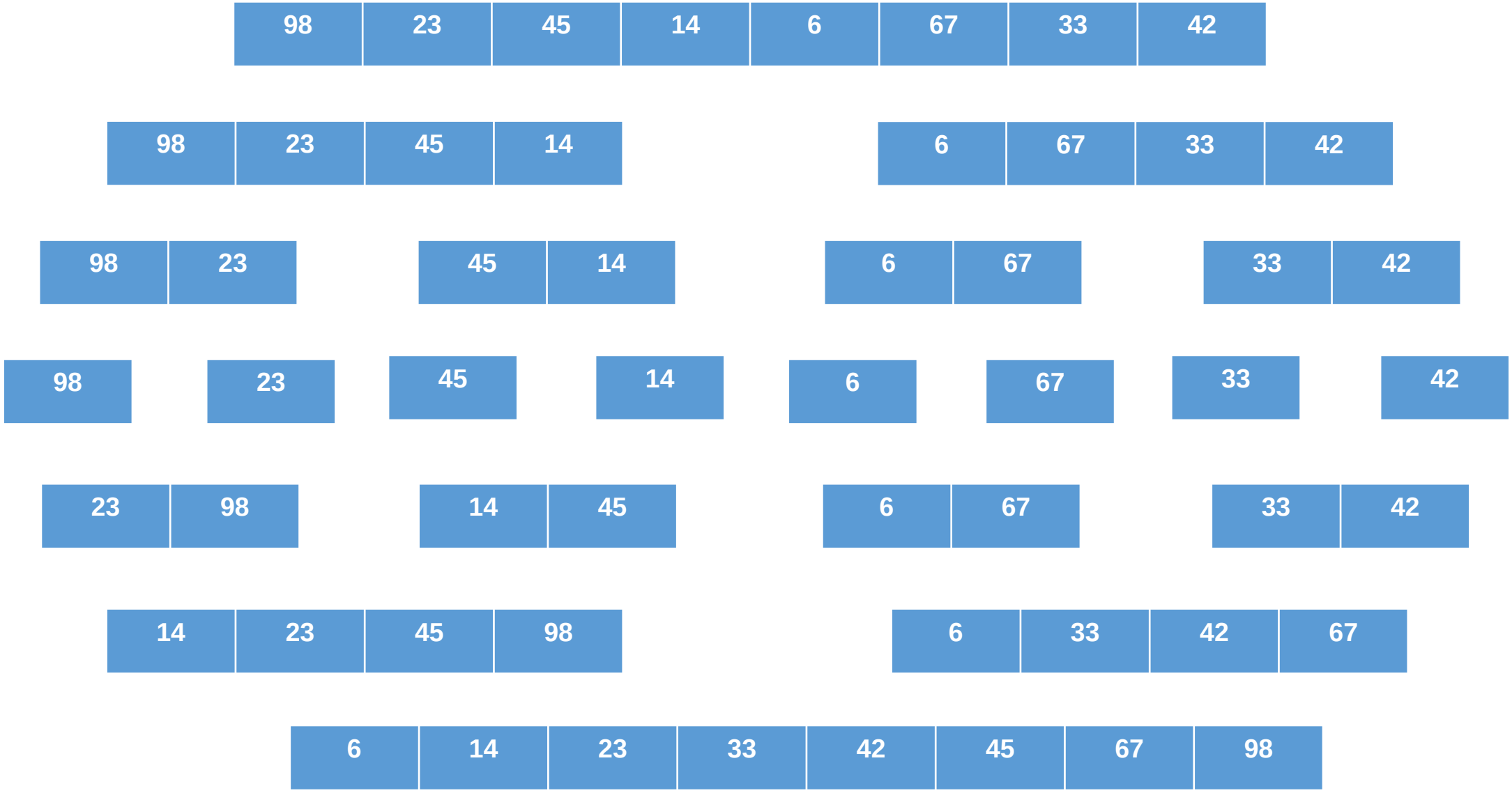
Tri par fusion

Exemple :



Tri par fusion

Exemple :



Tri par fusion

Algorithme :

```
TriFusion(liste)
    Si la taille de la liste est <= 1
        Retourner la liste non modifiée
    Diviser la liste en deux moitiés, gauche et droite
    gauche = TriFusion(gauche)     // Tri récursif de la moitié gauche
    droite = TriFusion(droite)     // Tri récursif de la moitié droite
    Fusionner les moitiés triées gauche et droite en une seule liste triée
    Retourner la liste fusionnée
```

Tri par fusion

Implémentation :

L'implémentation de cet algorithme repose essentiellement sur 2 fonctions :

fusionner : Cette fonction prend en paramètre deux listes triées dans un ordre croissant et retourne une nouvelle liste triée qui contient les éléments de A et B fusionnés.

tri_fusion : On vérifier si la liste L fournie en entrée n'est pas vide ou ne se résume pas à un seul élément, auquel cas, il n'y a rien à faire !
Si les tests sont concluants, on lancera la fonction de fusion sur 2 sous séquences récursives du tableau.

Tri par fusion

```
void fusionner(int tab[], int debut, int milieu, int fin) {
    int temp[100];
    int i = debut, j = milieu + 1, k = 0;

    while (i <= milieu && j <= fin) {
        if (tab[i] < tab[j]) {
            temp[k] = tab[i]; i++;
        } else {
            temp[k] = tab[j]; j++;
        }
        k++;
    }
    while (i <= milieu) {
        temp[k] = tab[i]; i++; k++;
    }
    while (j <= fin) {
        temp[k] = tab[j]; j++; k++;
    }
    // Recopier le tableau trier
    for (i = 0; i < k; i++) {
        tab[debut + i] = temp[i];
    }
}
```

Complexité : O(n)

Tri par fusion

```
void triFusion(int tab[], int debut, int fin) {

    if (debut < fin) {
        int milieu = (debut + fin) / 2;

        triFusion(tab, debut, milieu);
        triFusion(tab, milieu + 1, fin);

        fusionner(tab, debut, milieu, fin);
    }
}
```

Complexité quasi linéaire : O(n log(n))

Tri par fusion

Avantages :

1. **Stabilité** : Le tri fusion est stable, ce qui signifie que l'ordre relatif des éléments égaux est préservé. Cela peut être essentiel lorsque vous triez des données par plusieurs critères.
2. **Performance garantie** : Le tri fusion garantit un temps d'exécution de O(n log n) dans le pire des cas. Cela en fait l'un des algorithmes de tri les plus efficaces pour trier de grandes quantités de données, en particulier lorsque la taille des données est importante.
3. **Adaptabilité** : Le tri fusion est bien adapté aux listes chaînées, car il peut être mis en œuvre de manière récursive, ce qui facilite le tri de structures de données liées.

Tri rapide

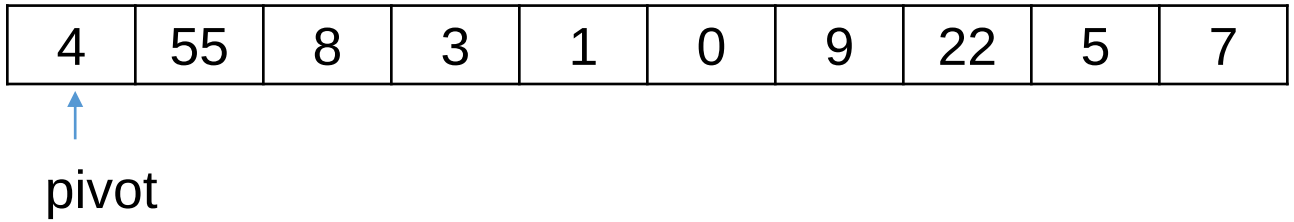
Principe :

L’algorithme de tri rapide, "quick sort", est un algorithme fondé sur la méthode de conception diviser pour régner; Son principe consiste à séparer l’ensemble des éléments en deux parties.

Pour effectuer la séparation, une valeur pivot est choisie au hasard. Les valeurs sont réparties en deux ensembles suivant qu’elles sont plus grandes ou plus petites que le pivot. Ensuite, les deux ensembles sont triés séparément, suivant la même méthode.

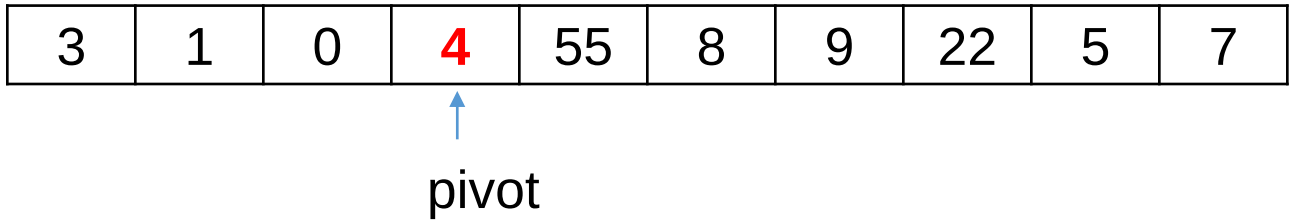
Tri Rapide – Quick Sort

1-Choisir un pivot



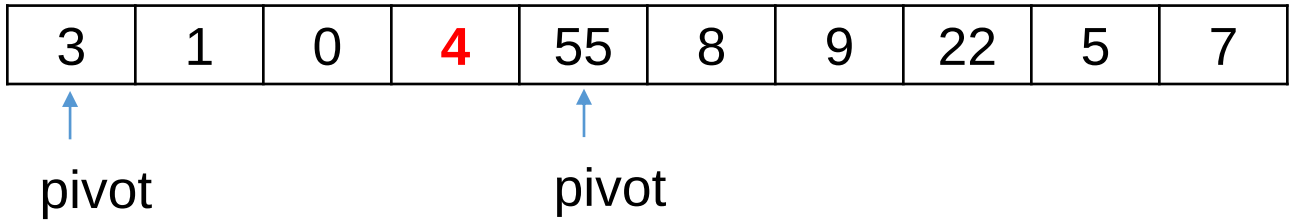
Tri Rapide – Quick Sort

2-Réorganiser les éléments autour du pivot



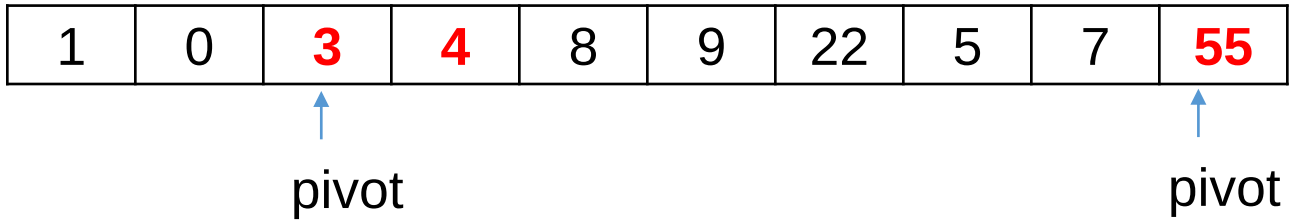
Tri Rapide – Quick Sort

3-Répétez 1 et 2



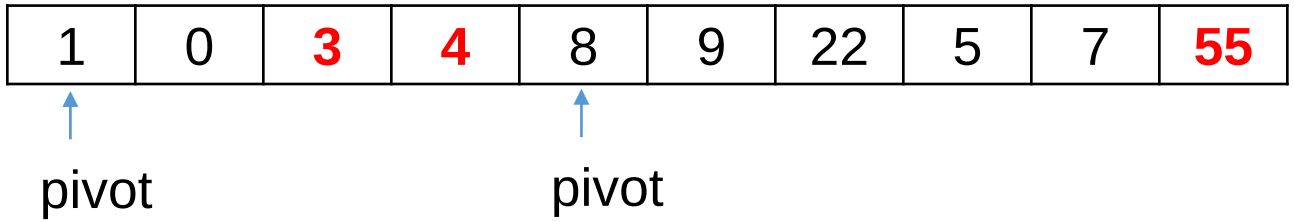
Tri Rapide – Quick Sort

3-Répétez 1 et 2



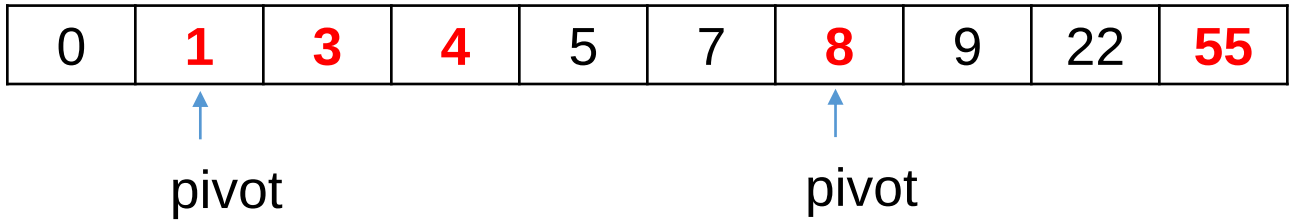
Tri Rapide – Quick Sort

3-Répétez 1 et 2



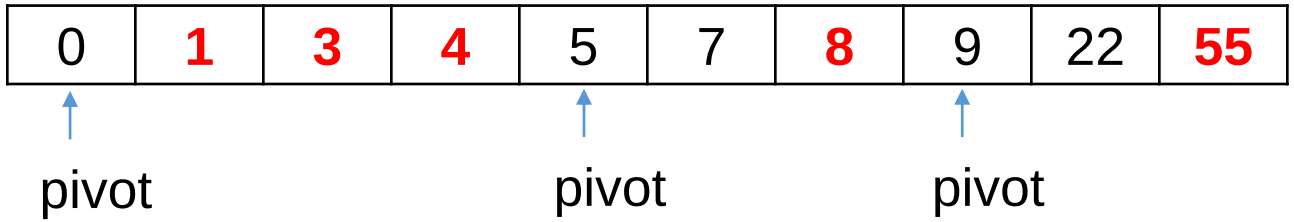
Tri Rapide – Quick Sort

3-Répétez 1 et 2



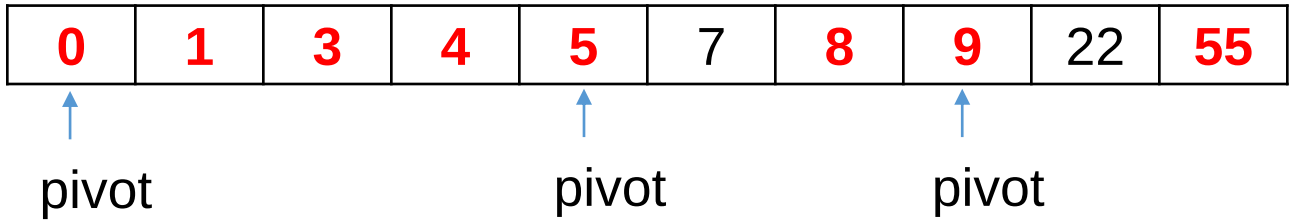
Tri Rapide – Quick Sort

3-Répétez 1 et 2



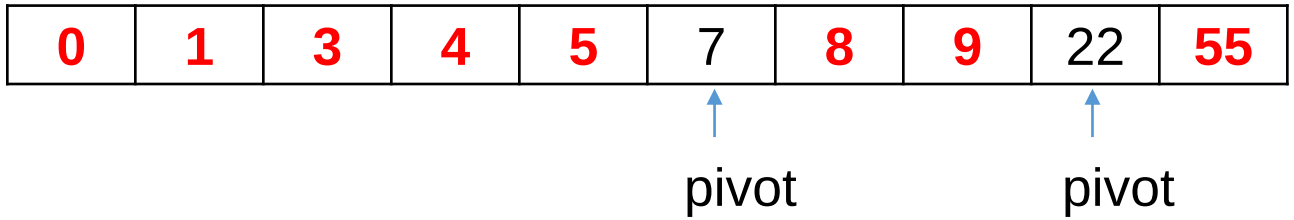
Tri Rapide – Quick Sort

3-Répétez 1 et 2



Tri Rapide – Quick Sort

3-Répétez 1 et 2



Tri Rapide – Quick Sort

3-Répétez 1 et 2

0	1	3	4	5	7	8	9	22	55

Tri rapide

Implémentation :

Ecrire la fonction « **partition** » : qui prend en paramètre une liste L un indice de début et un indice de fin.

Cette fonction choisit un pivot et réorganise les éléments de la liste autour du pivot, puis renvoie le nouvel indice du pivot.

Ecrire la fonction « **tri_rapide** » : : qui prend en paramètre une liste L un indice de début et un indice de fin. Cette fonction implémente l'algorithme du tri rapide, à la fin la liste L doit être triée.

Tri rapide

```c
void echanger(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int partition(int tab[], int debut, int fin) {
    int pivot = tab[debut];
    int i = debut;

    for (int j = debut+1; j <= fin; j++) {
        if (tab[j] < pivot) {
            i++;
            echanger(&tab[i], &tab[j]);
        }
    }
    echanger(&tab[debut], &tab[i]);

    return i + 1;
}
```

Complexité de la fonction partition : O(n)

Tri rapide

```
void triRapide(int tab[], int debut, int fin) {
    if (debut < fin) {

        int p = partition(tab, debut, fin);

        triRapide(tab, debut, p - 1);
        triRapide(tab, p + 1, fin);
    }
}
```

Complexité quadratique au pire : $O(n^2)$

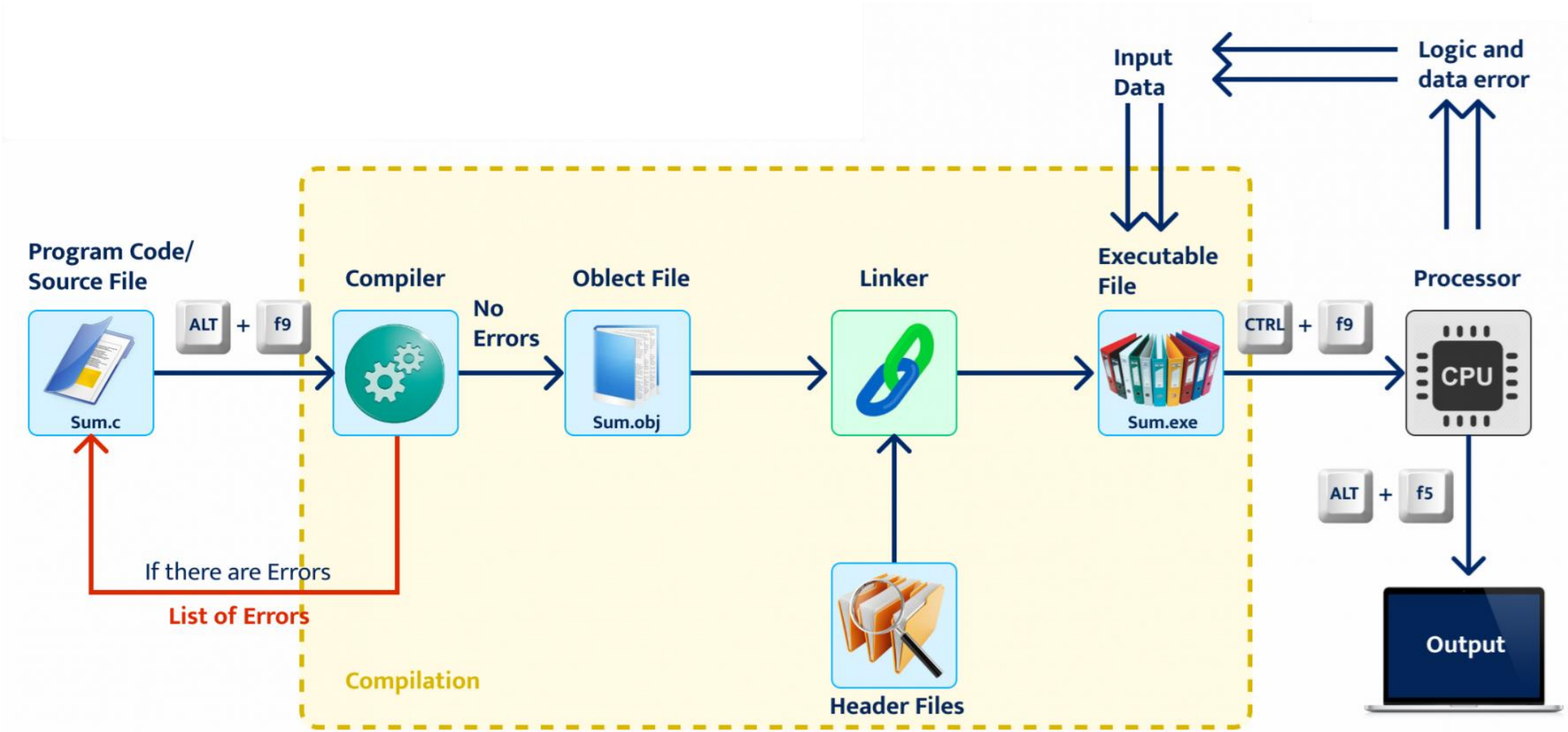
Rappel : Pointeurs, tableaux et enregistrements

HESTIM
ENGINEERING & BUSINESS SCHOOL

Les pointeurs

Notion d'adresse

Etapes d'exécution d'un programme C :



Gestion de la mémoire des variables en C :

1. Compilation

- Le compilateur analyse le **type**, la **taille** et la **portée** des variables.
- Il génère le code machine et les informations nécessaires à l'organisation du programme.
- À ce stade, le programme n’est pas encore chargé dans la mémoire vive (RAM) pour être exécuté.
- Pour les variables locales, le compilateur peut notamment déterminer leur organisation dans une frame de pile (ou les placer dans des registres selon les optimisations).

Gestion de la mémoire des variables en C :

2. Chargement

Lorsque le programme est exécuté, son espace mémoire est organisé conceptuellement en plusieurs zones :

- Zone du code : pour les instructions compilées.
- Zone des données statiques : pour les variables globales et statiques.
- Pile (stack) : pour les variables locales et les paramètres de fonctions.
- Tas (heap) : pour les allocations dynamiques (malloc, calloc, etc.).

0x000000	**Reserved**	
	.text .bss .data	Programme
	Allocated Data	Heap
sbrk(0)		Free memory
0x00FF23	Lib.so	links
	….	Stack
	Frame 2	
	Frame 1	
	Frame 0	

Gestion de la mémoire des variables en C :

3. Pendant l'exécution

- Variables globales / statiques → réservées une seule fois au démarrage.
- Variables locales → créées sur la pile quand une fonction est appelée et libérées à sa fin.
- Variables dynamiques → créées dans le tas avec malloc() et libérées par free().

Notion d'adresse

Mémoire

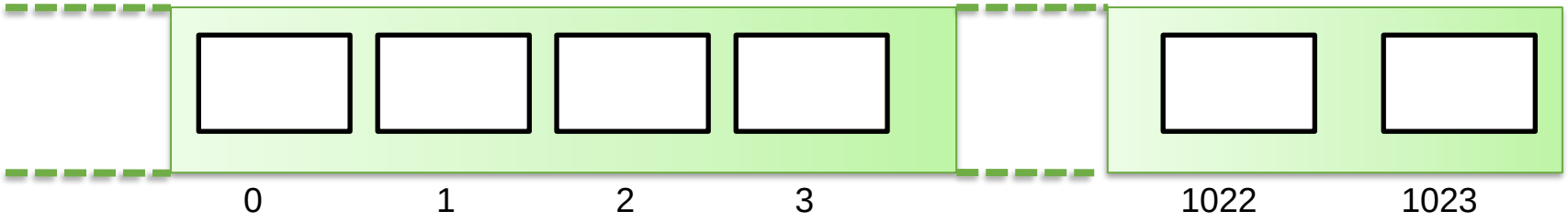
- La mémoire est constituée d’un certain nombre de cases appelées entités adressables.
- Dans la plupart des machines, le contenu d'une case est un octet (au max 2^8-1).
- Le nombre d'entités adressables(de cases) de la mémoire centrale est la capacité de la mémoire.



Adressage de la mémoire

- Chacune des cases de la mémoire centrale est numérotée, et **ce numéro est l'adresse de la case** ou de l'entité adressable(ensemble de cases).

- On numérote à partir de zéro, de sorte que s'il y a 2^{10} = 1024 entités adressables, elles sont numérotées de 0 à 2^{10} - 1 =1023.



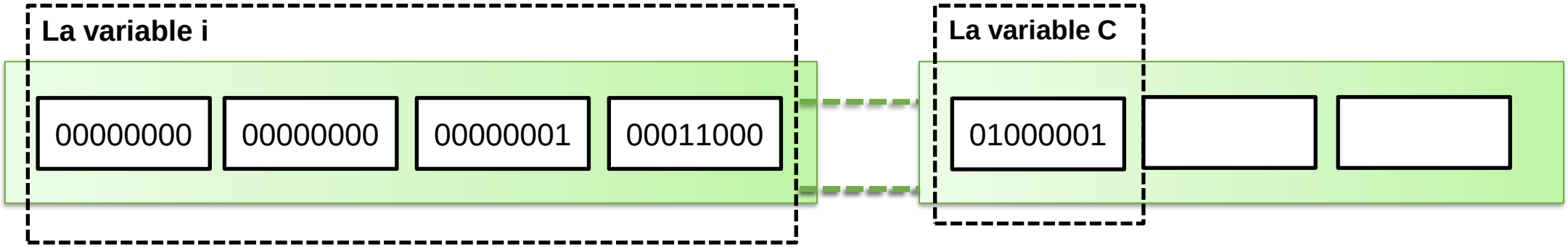
Stockage des données

En langage C les entiers (**int**) sont généralement codés sur 4 octets, donc pour stocker un entier on a besoin de 4 entités adressables. Les caractères (**char**) sont codés sur un octet, donc on a besoin d'une entité adressable. Les réels (**float**) sont codés sur 4 octets, donc on a besoin de 4 entités adressables.

Les valeurs des variables sont stockées dans la mémoire sous la forme binaire.

Exemples :

```
int i=280;        //280 en binaire est 100011000
char c='A';       //'A' en ASCII est 65 en binaire  1000001
```



Adresse d'une variable

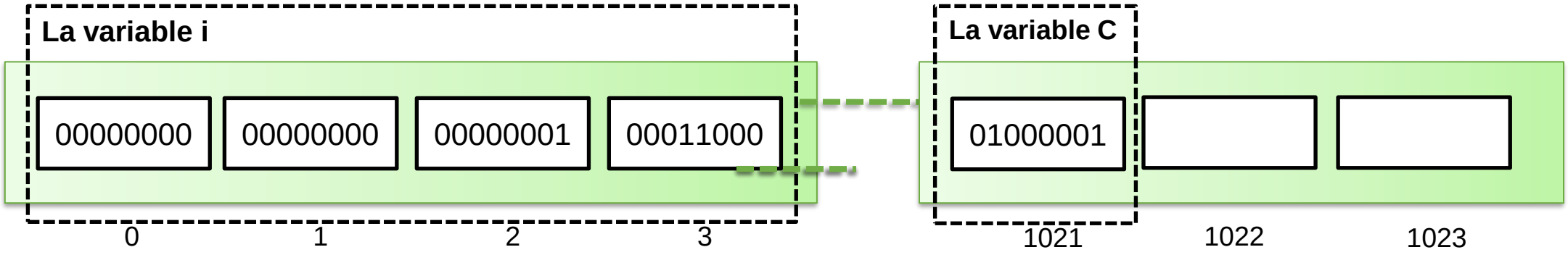
L'adresse d'une variable est l'adresse de sa première case mémoire.

Exemple :

```
int i=280;      //280 en binaire est 100011000
char c='A';     //'A' en ASCII est 65 en binaire  1000001
```

L'adresse de la variable i est 0

L'adresse de la variable c est 1021



L'operateur &

En langage C, Si X est une variable alors &X est l'adresse de la variable (début de l'emplacement mémoire réservé à X).

- & est appelé opérateur d'adresse.
- le format d'une adresse est **%p** (hexadécimal) ou **%d** (décimal) dans printf.

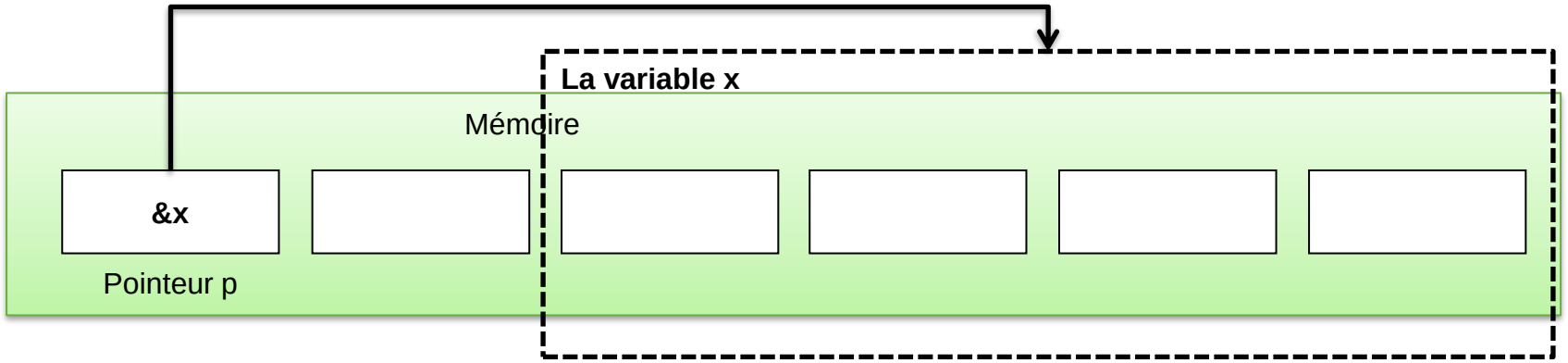
Exemple:

```
int x = 8;
printf("Voici x: %d",x);
printf("Voici son adresse en hexadecimal: %p",&x);
```

Notion de pointeur

- Un pointeur est une variable (d'un type spécial qui pourra contenir l'adresse d'une autre variable.

- On dit qu'il pointe vers cette variable.



Notion d'adresse

Déclaration

La déclaration d'un pointeur devra indiquer le type d'objet vers lequel il va pointer (on dit d'un pointeur qu'il est typé).

Syntaxe : **type *identificateur;**

Exemple: Pour déclarer un pointeur qui va pointer vers un entier on écrira:

```
int  *p;    // Recommandée
int*  p;
```

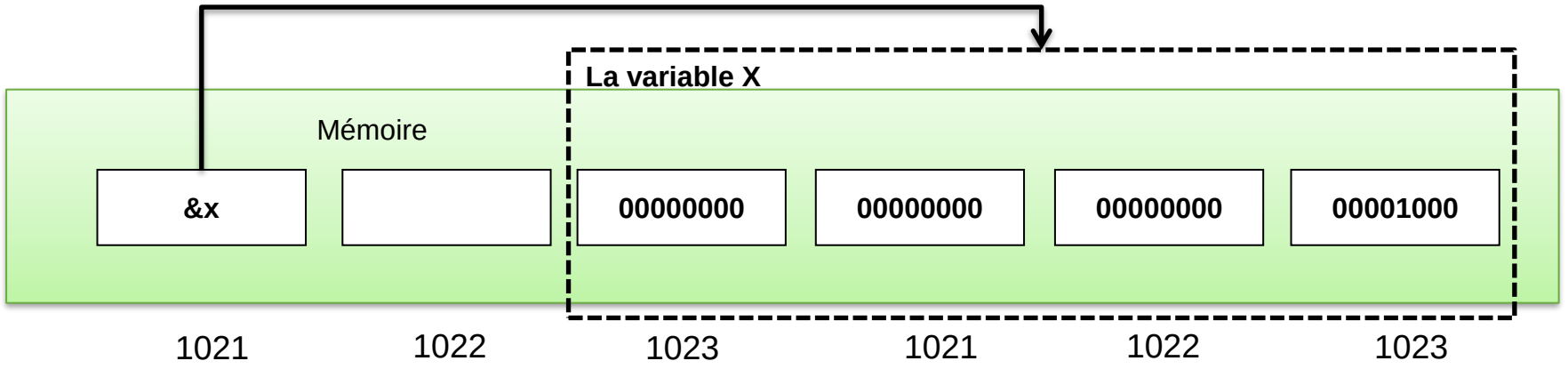
Notion pointeur

Initialisation

Pour initialiser un pointeur de type T on lui affecte l'adresse d'une variable de qui porte le même type T.

Exemple:

```
int x = 8;
int *p;
p=&x;   //On dit que le pointeur P pointe sur la variable X
```



Notion pointeur

L’opérateur *

- On accède à la valeur stockée dans l’emplacement qui porte l’adresse contenue dans un pointeur grâce à l’operateur unaire dit de référencement ou d’indirection * .
- **x** est équivalente à ***p**
- **p** est équivalente à **&x**

Notion pointeur

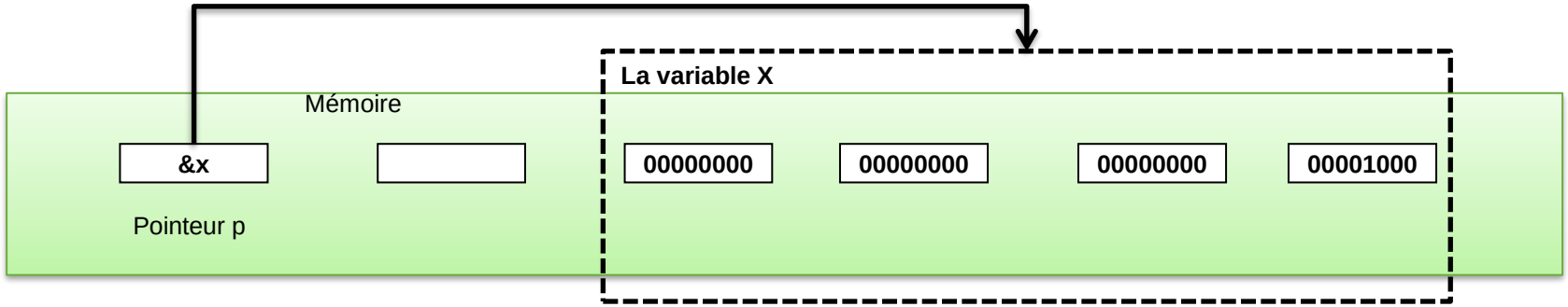
L'opérateur *

Exemple :

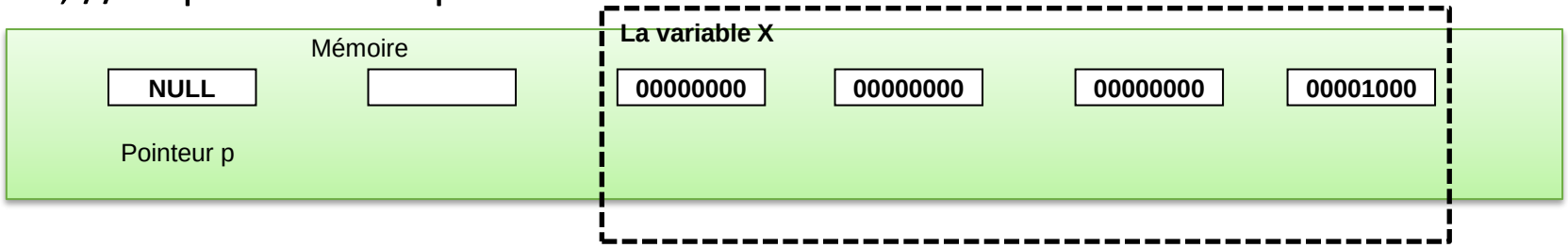
```
int x = 8;
Int *p;
p=&x;
printf(''l'adresse de la variable x est %d'',&x); //ou bien printf(''l'adresse de la variable x est %d'',p)
printf('la valeur de la variable x est %d'',x);  //ou bien printf(''la valeur de la variable x est %d'',*p)
```

Pointeur NULL

- Un pointeur qui ne pointe sur rien contient la valeur NULL
  - int x = 8;
  - int *p;
  - p=&x ; //On dit que le pointeur P pointe sur la variable X



  - p=NULL; // le pointeur ne pointe sur rien



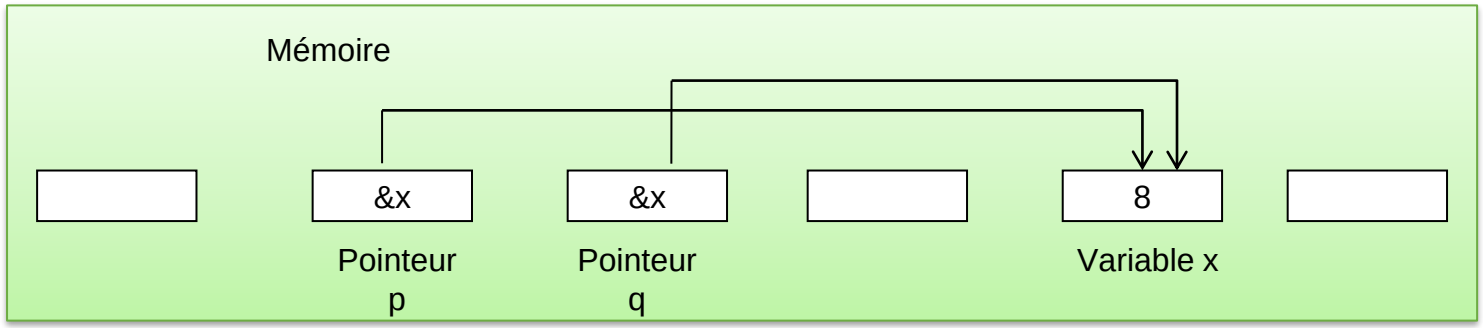
Copie d'un pointeur

Soit la déclaration suivante :

```
int x = 8;
Int *p;
Int *q;
p=&x;
q=p;
```

Les deux pointeurs p et q pointent sur le *même* emplacement mémoire, donc on a:

- p==q==&x
- et on a également *p==*q==x



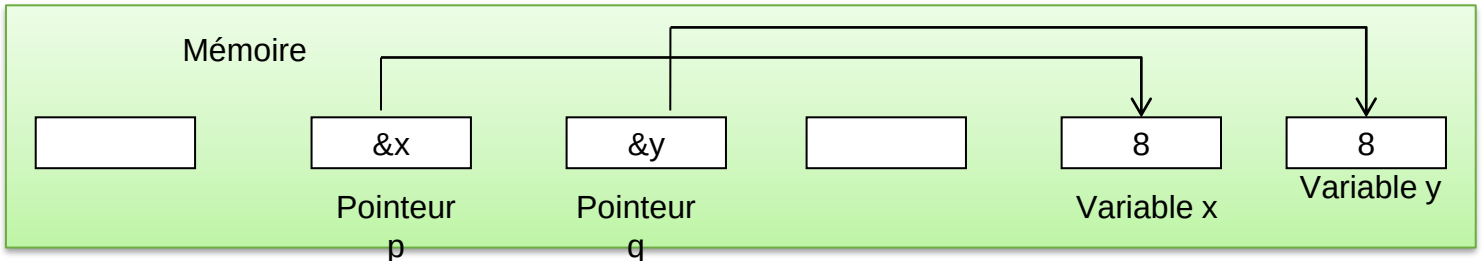
Copie d'une valeur pointée

Soit la déclaration suivante

```
int x = 8;
int y=10;
Int *p;
Int *q;
p=&x;
q=&y
*q=*p;
```

Les valeurs pointées sont identiques mais les pointeurs restent *différents*, donc on a:

- p==&x et q==&y
- Donc p!=q
- et on a également *p==*q==x==y



Notion pointeur

Exercice

Soit la déclaration suivante :

```
int a=8;
int b=10;
int *p;
int *q;
p=&a;
q=&b;
```

Donnez le contenu des variables a et b après l ’exécution de chaque instruction :

```
*p = *q + 2;
(*q)++;
*p = *p + 3;
b = *p + *q;
p = q;
*p = 20;
```

Solution:

a=15

b=20

Notion pointeur

Remarque

- Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur.

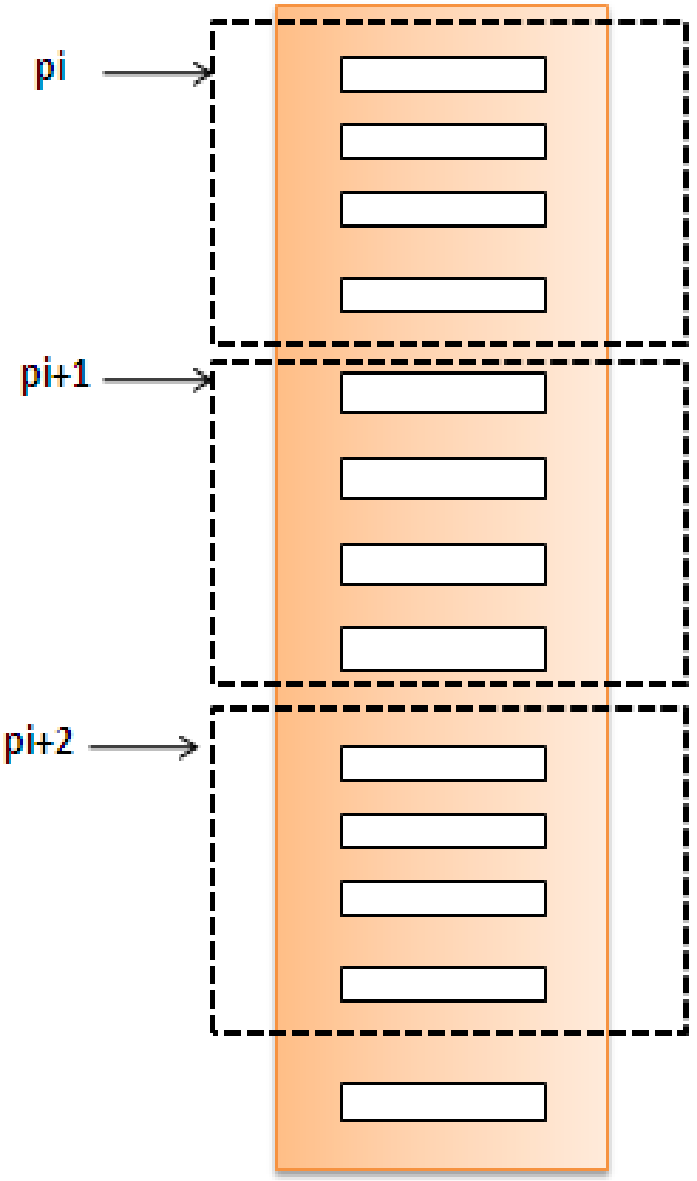
- Il faut quand même bien faire la différence:
  - Un ***pointeur*** est une variable qui peut 'pointer' sur différentes adresses.
  - Le ***nom d'une variable*** reste toujours lié à la même adresse d'un emplacement mémoire où on peut stocker plusieurs fois de nouvelles valeurs.

Arithmétique des pointeurs

- On peut **déplacer** un pointeur dans un plan mémoire à l'aide des opérateurs d'addition, de soustraction, d'incrémentation, et de décrémentation.

- Si **pi** est un pointeur de type int (int *pi) alors **pi+1** est **l'adresse de l'entier suivant**

- On ne peut déplacer **pi** que **d'un nombre de cases mémoire multiple du nombre de cases réservées en mémoire pour la variable sur laquelle il** pointe(pour le cas des entiers c'est 4)**.**



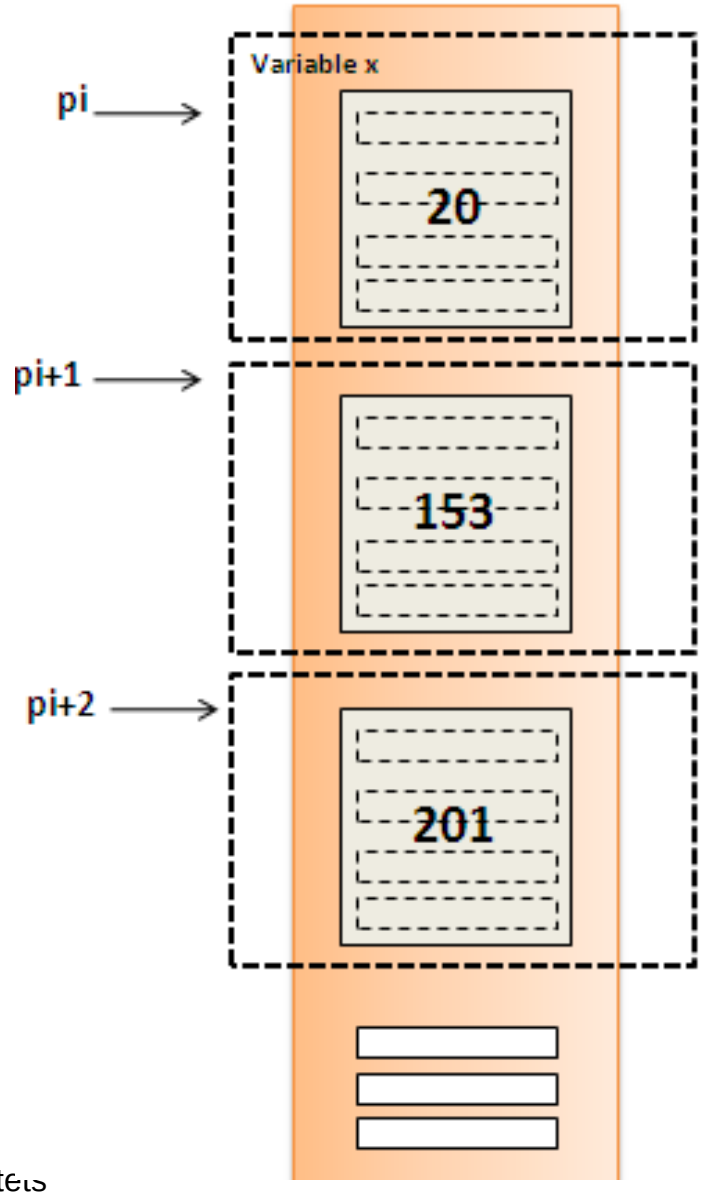
Une case mémoi

Notion pointeur

Arithmétique des pointeurs

Exemple:

```
int x;
int *pi;
pi=&x
*pi=20;
*(pi+1)=153;
*(pi+2)=201;
```



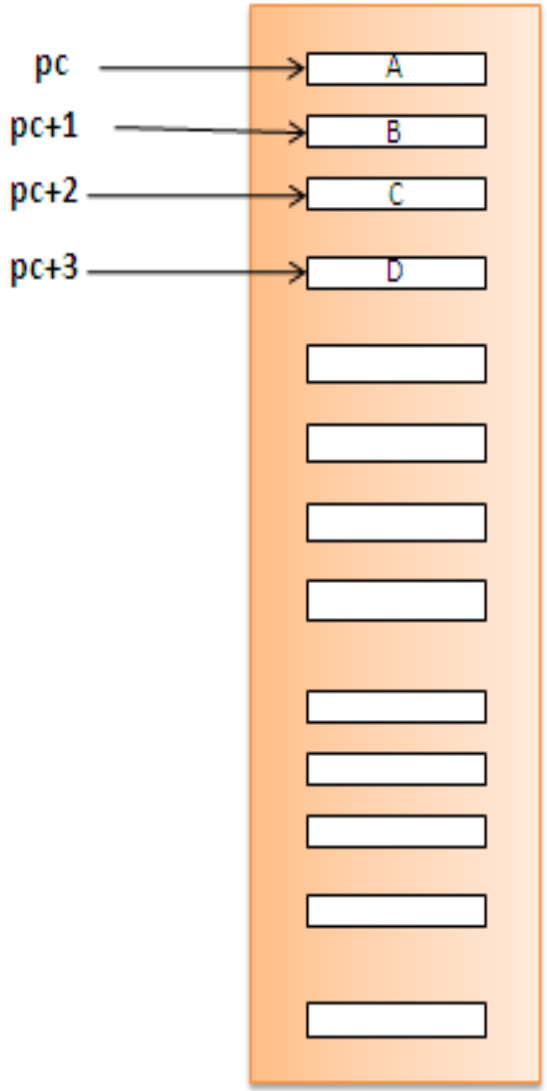
Une case mémoire = 1 octets

Notion pointeur

Arithmétique des pointeurs

Exemple:

```
 char c
 char *pc;
 pc=&c
*pc='A';
*(pc+1)='B';
*(pc+2)='C';
*(pc+3)='D';
```



Une case mémoire = 1 octets

A quoi sert les pointeurs ?

- **Passage par adresse des paramètres :**

Transmettre à une fonction l'adresse mémoire d'une variable, au lieu de sa valeur.

- **Allocation dynamique de la mémoire:**

Réserver de la mémoire pendant l'exécution du programme (et non au moment de la compilation).

Passage par adresse des paramètres

Passage par adresse des paramètres

En C nous pouvons passer les paramètres des fonctions par :

- Par valeur:
- Par adresse

Passage par adresse des paramètres

Passage par valeur :

- Les arguments sont copiés dans des objets temporaires qui sont crées lors du passage d'arguments et qui sont détruit à la fin de l'exécution de la fonction .
- Toutes les modifications effectuées sur ces objets temporaires seront donc perdues à la fin de la fonction et n'auront aucune incidence sur l'objet passé en argument lors de l'appel de la fonction.

Passage par adresse des paramètres

Passage par valeur :

```c
void incremente(int i) /* i est la copie de la valeur passée en paramètre */ {
    i++; /* Modifie i, mais pas la variable fournie par l'appelant. */
}

int main()
{
    int j = 12;
    incremente(j); /* Le contenu de j est copié dans i. j vaut toujours 12.*/
    printf("j=%d", j); // Affiche 12
}
```

Passage par adresse des paramètres

Passage par adresse :

- Les arguments sont des pointeurs qui contiennent l'adresse en mémoire des objets que l'on veut manipuler. Il n'y a donc plus de copie, plus de variable locale.
- Toute modification du paramètre dans la fonction appelée entraîne la modification de la variable passée en paramètre.

Passage par adresse des paramètres

Passage par adresse :

```
void incremente(int * i) { /*incremente attend l’adresse d’un entier*/
    (*i)++;
}

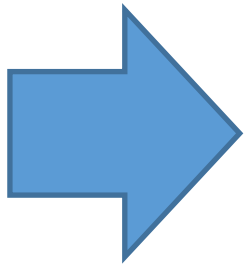
int main() {
    int j = 12; incremente(&j); /* On passe l'adresse de j en paramètre. */
    printf("j=%d", j); // Affiche 13
}
```

Passage par adresse des paramètres

Exercice : Corriger le code suivant

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

int main() {
    int x = 8, y = 10;
    echanger(x, y);
    printf("x=%d y=%d", x ,y);
}
```



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

int main() {
    int x = 8, y = 10;
    echanger(&x, &y);
    printf("x=%d y=%d", x ,y);
}
```

HESTIM
ENGINEERING & BUSINESS SCHOOL

Allocation dynamique de la mémoire

Allocation dynamique de la mémoire

Introduction :

Comment résoudre ce problème ?

Ecrire un programme qui calcule la moyenne de **n** étudiants entrés par l'utilisateur.

Solution 1: Déclarer un tableau statique avec un nombre de cases suffisant.

Solution 2: Ne réserver le tableau qu'après connaître la valeur exacte de n (Au moment de l'exécution).

Pointeurs et tableaux à une dimension

- L'utilisation des tableaux en langage C impose de réserver de l'espace mémoire avant de savoir le nombre exacte d'éléments dont on a besoin à traiter.

Exemple :

```
int T[100];
int n;
printf(''Donner le nombre des étudiants :'');
scanf(''%d'',&n);
```

Statique

La déclaration précédente implique que **100-N** cases mémoires seront inutiles.

- Ce qui montre une autre fois l'utilité des pointeurs et de l'allocation dynamique de la mémoire en particuliers.

Exemple:

```
int *T;
int n;
printf('Donner le nombre des étudiants : '');
scanf(''%d'',&n);
T=(int *)malloc(n*seizeof(int));
```

Dynamique

Allocation dynamique de la mémoire

Allocation statique vs Allocation dynamique

Allocation statique	Allocation dynamique
Taille connue à la compilation	Taille déterminée à l'exécution
Allouée sur la pile (stack) ou dans le segment global	Allouée sur le tas (heap)
Libération automatique (fin de portée)	Liberté de libérer manuellement (free())

Allocation dynamique de la mémoire

Les fonctions disponibles pour gérer la mémoire (bibliothèque **stdlib.h**) :

Fonction	Rôle	Exemple
malloc()	Alloue un bloc mémoire **non initialisé**	ptr = malloc(taille_en_octets);
calloc()	Alloue et **initialise à 0** un bloc mémoire	ptr = calloc(n, taille);
realloc()	Change la **taille** d'un bloc déjà alloué	ptr = realloc(ptr, nouvelle_taille);
free()	**Libère** la mémoire allouée	free(ptr);

La fonction malloc pour allouer la mémoire :

Pour allouer de l'espace mémoire on utilise la fonction **malloc** de la bibliothèque **stdlib.h**

Syntaxe:

pointeur = (type*) **malloc** (nombre de case mémoires à réserver)

Exemple :

```
char *pc;
int *pi,
float *pr;
pc =(char*) malloc(10)      /* on réserve 10 cases mémoire, la place pour 10 caractères */
pi =(int*) malloc(16);      /* on réserve 16 cases mémoire, soit la place pour 4 entiers  */
pr =(float*)malloc(24);     /* on réserve 24 places en mémoire, soit la place pour 6 réels */
```

La fonction calloc pour allouer la mémoire :

La fonction **calloc** permet d'allouer la mémoire comme la fonction malloc, mais elle initialise tous les octets à 0.

Syntaxe:

pointeur = (type*) **calloc** (size_t n, size_t size)

Exemple :

```
int *pi,
float *pr;
pi =(int*) calloc(4, sizeof(int));   /* soit la place pour 4 entiers  */
pr =(float*) calloc(6, sizeof(float));   la place pour 6 réels */
```

La fonction realloc pour allouer la mémoire :

La fonction **realloc** permet de modifier la taille d'un bloc mémoire déjà alloué dynamiquement.

Syntaxe:

ptr = **realloc(void *ptr, size_t new_size);**

Exemple :

```
int * pr,
ptr =(int*) malloc(4*sizeof(int));        /* on réserve 4 cases pour 4 entiers  */
ptr =(int*) realloc(ptr, 10*sizeof(int));    /* Ajouter 6 nouvelles cases*/
```

La fonction free pour libérer la mémoire:

La fonction **free**: pour libérer l'espace mémoire alloué:

```
int *pc = (int) malloc(5*sizeof(int));

free(pc)                    /* on libère la place précédemment réservée pour pc*/
```

Pointeurs et tableaux à une dimension

Un tableau est en fait un pointeur constant, donc il peut être manipuler comme un pointeur.

	Tableaux	Pointeur
Déclaration	**int T[n];**	**int *T; T = (int*)malloc(n*sizeof(int));**
Le 1er élément	**T[0]**	***T**
Un autre élément	**T[i]**	***(T+i)**
Adresse du 1er élément	**&T[0]**	**T**
Adresse d'un autre élément	**&(T[i])**	**(T + i)**

Allocation dynamique de la mémoire

Exemple 1

Les deux programmes équivalents suivants copient les éléments positifs d'un tableau T dans un deuxième tableau POS.

```
Void main()
{
   int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
   int POS[10];
   int I,J;    /* indices courants dans T et POS */
   for (J=0,I=0 ; I<10 ; I++)
      if (T[I]>0)
      {
         POS[J] = T[I];
         J++;
      }
}
```

```
Void main()
{
   int *T= {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
   int *POS;
   POS= (int*)malloc(10*sizeof(int));
   int I,J;    /* indices courants dans T et POS */
   for (J=0,I=0 ; I<10 ; I++)
        if (T[I]>0)
        {
            POS[J] = T[I];
            J++;
        }
}
```

Allocation dynamique de la mémoire

Remarques :

- Un pointeur est une variable qui peut pointer sur une variable ou un tableau qui portent le même type.
- Un tableau est un pointeur constant, alors il est impossible de changer son contenu par une affectation.
- Soit la déclaration suivante :
  - int T1[]={1,2,3,4,5,6};
  - int T2[]={10,11,12,13,14,15};
  - int *P1;
  - int *P2;
- Les instructions suivantes sont incorrecte(les tableaux sont des constantes)
  - T1=T2;
  - T2=P1;
- Les instructions suivantes sont correctes(les pointeurs sont des variables)
  - P1=T1
  - P2=T2

Tableaux statiques à 2 Dimensions

Utilisation du formalisme tableau uniquement .

- syntaxe : Type T[N][M];
- Exemple : int T[3][4];

	0	1	2	3
0	T[0][0]	T[0][1]	T[0][2]	T[0][3]
1	T[1][0]	T[1][1]	T[1][2]	T[1][3]
2	T[2][0]	T[2][1]	T[2][2]	T[2][3]

Tableaux dynamique à 2 Dimensions

Déclaration d'un pointeur de pointeur :

Exemple:

```
int **T;
T= (int **) malloc(3*sizeof(int*));
for(i=0;i<3;i++)
    T[i]=(int*)malloc(4*sizeof(int));
```

	0	1	2	3
0	T[0][0]	T[0][1]	T[0][2]	T[0][3]
1	T[1][0]	T[1][1]	T[1][2]	T[1][3]
2	T[2][0]	T[2][1]	T[2][2]	T[2][3]

Les enregistrements

Introduction

Les types utilisés, jusqu'à présent, sont :

- des types simples, tels que:
  - + réel,
  - + entier,
  - + caractère,
- et le type structuré tableau qui permet de stocker plusieurs données de même type.

Il est parfois nécessaire de définir un objet qui décrit plusieurs informations de types différents. Cet objet est alors définit à travers un type structuré appelé **enregistrement** ou **structure**.

Les enregistrements

Introduction

Comment modéliser un étudiants caractérisé par son nom, prénom, age et sa note ?

```
int main(){

    // les données d'un étudiant
    char nom[30];
    char prenom[30];
    int age;
    float note;

}
```

Problème :

Difficile à gérer plusieurs étudiants dans le programme.

Introduction

- Une structure est une collection de plusieurs variables (champs) groupées dans un ensemble pour un traitement commode,

- Les variables d'une structure sont appelées membres et peuvent être de n'importe quel type (tableaux, pointeurs, entiers ….)

Les enregistrements

Déclaration d'un type structure :

Pour déclarer un type structure, on utilise la syntaxe suivante :

```
typedef struct {
    char nom[30];
    char prenom[30];
    int age;
    float note;
} Etudiant;
```

Les membres de la structure

Le nom de la structure

Déclaration d'un type structure :

Une fois le type structure est déclaré, nous pouvons utiliser ce nouveau type pour déclarer des variables de ce type :

Syntaxe : type identificateur;

Exemple :

```
Etudiant et1 ,  et2; // Deux variables de type étudiants
Etudiant T[10];  // Un tableau de 10 étudiant;
```

Accéder aux membres d'une variable structure :

En C, pour accéder aux différents membres d'une variable de type structure, on utilise l'opérateur point . :

Syntaxe : identificateur.membre;

Exemple :

```
Etudiant et;
strcpy(et.nom , "Bahri");
strcpy(et.prenom , "Ahmed");
et.age = 16;
et.note = 14;
```

```
printf("Age : %d\n", et.age);
printf("Note : %.2f\n", et.note);
```

Activité 1:

Soit un produit est caractérisé par: Référence (entier), Désignation (chaine de caractère), Quantité (entier), Prix (réel)
Déclarer le type structure qui définie le produit en langage C.

Solution :

```
typedef struct {
    int reference;
    char designation[50];
    int quantite;
    float prix;
} Produit;
```

Les enregistrements

Activité 2:

Ecrire un programme qui déclare une variable de type Produit, puis lire les données du produit auprès de l'utilisateur. Puis afficher les valeur entrées.

```c
int main() {

    Produit p;

    printf("Reference : ");
    scanf("%d", &p.reference);
    printf("Designation : ");
    scanf("%s", p.designation);
    printf("Quantite : ");
    scanf("%d", &p.quantite);
    printf("Prix : ");
    scanf("%f", &p.prix);

    printf("\n--- Produit ---\n");
    printf("Reference   : %d\n", p.reference);
    printf("Designation : %s\n", p.designation);
    printf("Quantite    : %d\n", p.quantite);
    printf("Prix        : %.2f DH\n", p.prix);

    return 0;
}
```

Les enregistrements

Activité 2:

Ecrire un programme qui déclare et lit un tableau de 10 produits.

```c
int main() {

    Produit produits[10];
    int i;

    /* Lecture des 10 produits */
    for (i = 0; i < 10; i++) {

        printf("\n--- Produit %d ---\n", i + 1);
        printf("Reference : ");
        scanf("%d", &produits[i].reference);
        printf("Designation : ");
        scanf("%s", produits[i].designation);
        printf("Quantite : ");
        scanf("%d", &produits[i].quantite);
        printf("Prix : ");
        scanf("%f", &produits[i].prix);
    }

    return 0;
}
```

Les enregistrements

Activité 3:

Déclarer les deux types composés suivants :

- Adresse : numéro, rue et ville.
- Etudiant : numéro, nom et adresse.

```
typedef struct {
    int numero;
    char rue[50];
    char ville[30];
} Adresse;
```

```
typedef struct {
    int numero;
    char nom[30];
    Adresse adresse;
} Etudiant;
```

HESTIM
ENGINEERING & BUSINESS SCHOOL

Les pointeurs et enregistrements

Pointeurs sur enregistrements

Pointeur sur un enregistrement

De la même façon que les types simples, on peut déclarer des pointeurs sur des types complexes(enregistrement)

Exemple :

```
typedef  struct {
      char nom[20];
      char prenom[20];
      int   age;
} Personne;
```

```
int main()
{
    Personne  *p;
    p=(Personne*)malloc(sizeof(Personne));
    scanf("%s",(*p).nom);
    scanf("%s",(*p).prenom);
    scanf("%d",&(*p).age);
    printf("%s %s %d",(*p).nom, (*p).prenom, (*p).age);
    return 0
}
```

Pointeurs sur enregistrements

Pointeur sur un enregistrement

Langage c simplifie l'accès aux champs de l'enregistrement dans le cas de la déclaration sous forme de pointeur et cela à l'aide de l'opérateur →.

(*p).nom ⟶ p->nom

(*p).prenom ⟶ p->prenom

(*p).age ⟶ p->age

```
typedef  struct {
      char nom[20];
      char prenom[20];
      int   age;
} Personne;
```

```
int main()
{
     Personne  *p;
     p=(Personne*)malloc(sizeof(Personne));
     scanf("%s",p->nom);
     scanf("%s",p->prenom);
     scanf("%d",&p->age);
     printf("%s %s %d",p->nom, p->renom, p->age);
     return 0
}
```

Enregistrement récursif

- Supposons que la structure personne contient maintenant : un nom, un prénom, l'âge , une personne père et une personne mère.

- La structure sera récursive puisqu'on décrit une personne comme contenant deux variables de type Personne qui contiennent chacune deux variables de type Personne, etc.

- **Essayer de déclarer un type structuré qui répond à cette nouvelle extension?**

Pointeurs sur enregistrements

Enregistrement récursif

- Avec les compétence qu'on jusqu'à maintenant, on peut écrire:

```
typedef  struct {
    char nom[20];
    char prenom[20];
    int   age;
    personne pere;
    personne mere ;
} personne;
```

- Cette déclaration est incorrecte car le compilateur ne peut déterminer a priori la taille en octets de la structure personne.

Pointeurs sur enregistrements

Enregistrement récursif

- Pour dépasser ce problème , le langage C propose la solution suivante

```
typedef  struct pers {
        char nom[20];
        char prenom[20];
        int   age;
        struct pers *pere;
        struct pers *mere ;
 } Personne;
```

Les listes chaînées en C

Introduction

- Les tableaux:
  - ➢ Stocker un ensemble de données de même type d'une façon linéaire
  - ➢ Allocation d'un seul bloc en mémoire avec des cases contiguës
  - ➢ Taille connue et fixe des éléments

⇒ Accès au élément en **O(1)**

Rechercher un élément en **O(n)**

Ajouter un élément en **O(n)**

Supprimer un élément en **O(n)**

Introduction

- Les listes
  - ➢ Taille inconnue au départ, elle varie au cours du temps
  - ➢ Allocation indépendante de chaque élément, sous la forme d'un maillon (cellule).
  - ➢ Les éléments sont habituellement éparpillés en mémoire.
  - ➢ Peuvent contenir des types de données différents

⇨ Accès au élément en **O(n)**

Rechercher un élément en **O(n)**

Ajouter un élément en **O(1)**

Supprimer un élément en **O(1)**

Introduction

Plusieurs type de listes :

- **Liste chaînée simple** constituée d'éléments reliés entre eux par des pointeurs.
- **Liste doublement chaînée** où chaque élément dispose non plus d'un mais de deux pointeurs pointant respectivement sur l'élément précédent et l'élément suivant.
- **Liste circulaire** où le dernier élément pointe sur le premier élément de la liste. S'il s'agit d'une liste doublement chaînée alors de premier élément pointe également sur le dernier.

Introduction

- La façon dont on met en œuvre ces structures dépend des langages, dans notre cours nous allons adopter la représentation en langage C.

- Une méthode simple pour représenter une liste, consiste à dire qu'une liste L soit vide, soit elle est constituée:
  - ✓ d'une tête T (qui est donc la valeur du premier élément de la liste) et
  - ✓ d'une queue Q (qui est le reste de la liste).

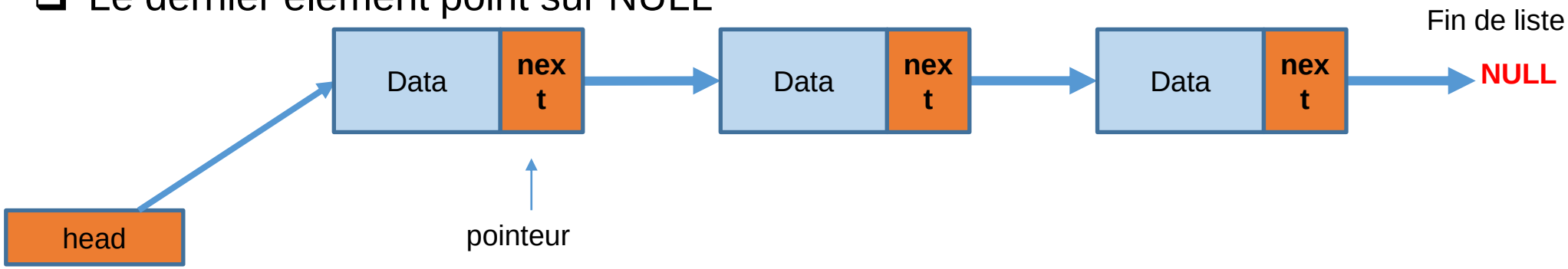
HESTIM
ENGINEERING & BUSINESS SCHOOL

LES LISTES SIMPLEMENT CHAÎNEES

Listes simplement chaînées

Principe :

- Contrairement aux tableaux, les éléments d'une liste chaînée ne sont pas placés côte à côte dans la mémoire.
- Chaque élément peut contenir ce que l'on veut: un ou plusieurs int, double, structures,...
- Chaque élément possède un pointeur vers l'élément suivant.
- Un pointeur "head" qui pointe sur le premier élément
- Le dernier élément point sur NULL



Listes simplement chaînées

Primitives :

- ❑ Créer une liste
- ❑ Ajouter un élément (Au début, au milieu , à la fin)
- ❑ Parcourir une liste
- ❑ Rechercher un élément
- ❑ Modifier un élément
- ❑ Supprimer un élément

Listes simplement chaînées

1 - Implémenter un Noeud :

L'implémentation d'une liste chaînées commence par implémenter une cellule (un nœud) :



Node

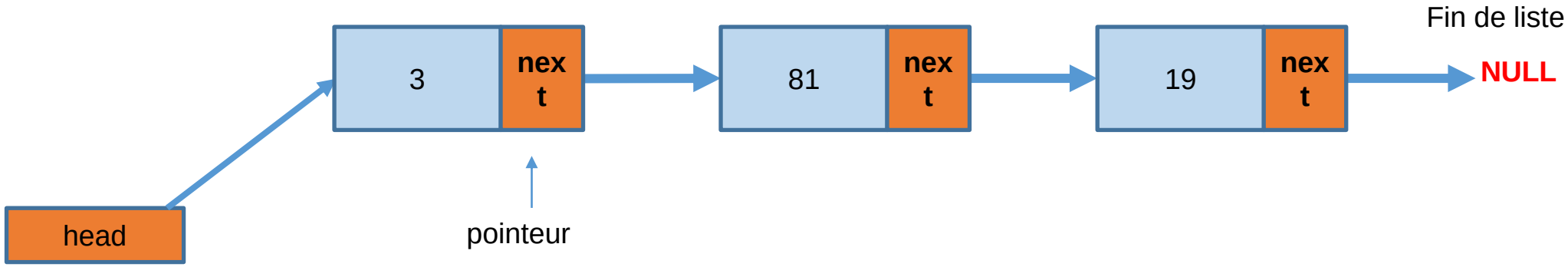


```
typedef struct node {
        // données
        struct node *next;
} Node;
```

Listes simplement chaînées

1 - Implémenter un Noeud :

L'implémentation d'une liste chaînées commence par implémenter une cellule (un nœud) :

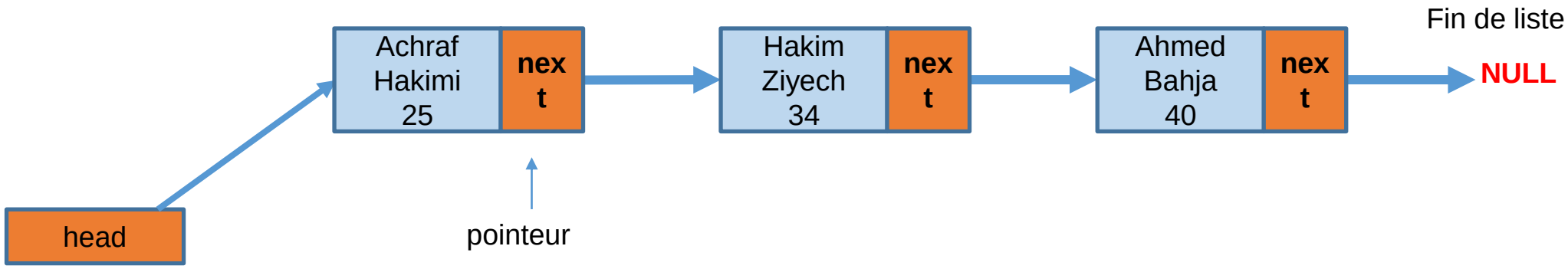



```c
typedef struct node {
    int val;
    struct node *next;
} Node;
```

Listes simplement chaînées

1 - Implémenter un Noeud :

L'implémentation d'une liste chaînées commence par implémenter une cellule (un nœud) :

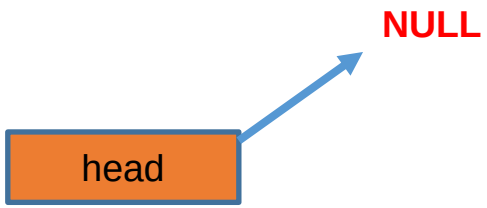



```
typedef struct node {
    char nom[20];
    char prenom[20];
    int age;
    struct node *next;
} Node;
```

Listes simplement chaînées

2 – Créer une liste vide :

Une liste vide est une liste qui ne contient aucun élément, donc le pointeur **tete** pointe sur NULL.



Pour déclarer maintenant une liste vide L :

```
Node *head = NULL;
```

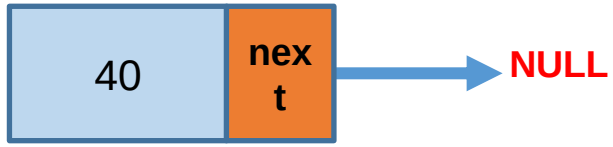
3 - Implémenter les primitives : Allouer un Node

Une fonction qui alloue un nouveau nœud :

```
Node* creerNoeud(int valeur) {
    Node* nouveau = (Node*) malloc(sizeof(Node));
    if (nouveau == NULL) {
        printf("Erreur d'allocation mémoire\n");
        exit(1);
    }
    nouveau->val = valeur;
    nouveau->next = NULL;
    return nouveau;
}
```

Exemple :

```
Node *n = creerNoeud(40)
```



Listes simplement chaînées

3 - Implémenter les primitives : Insérer au début

Insérer un nœud au début consiste à :

1. Allouer un nouveau nœud.
2. Le suivant de ce nouveau nœud est l'ancien nouveau nœud.
3. La tête de la liste est maintenant ce nouveau nœud.

```
Node* insererDebut(Node* head, int valeur) {
    Node* nouveau = creerNoeud(valeur);
    nouveau->next = head;
    return nouveau; // nouvelle tête
}
```

Exemple :

```
Node *head = NULL;
// autres instructions
head = insererDebut(head, 40);
head = insererDebut(head, 30);
head = insererDebut(head, 120);
```

Listes simplement chaînées

3 - Implémenter les primitives : Insérer à la fin

Insérer un nœud à la fin consiste à :

1. Allouer un nouveau nœud.
2. Si la liste est vide la nouvelle liste est notre nouveau nœud.
3. Si non, parcourir la liste jusqu'au dernier Node.
4. Le suivant du dernier Node est le nouveau élément.

```
Node* insererFin(Node* head, int valeur) {
    Node* nouveau = creerNoeud(valeur);
    if (head == NULL) return nouveau;

    Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = nouveau;
    return head;
}
```

Exemple :

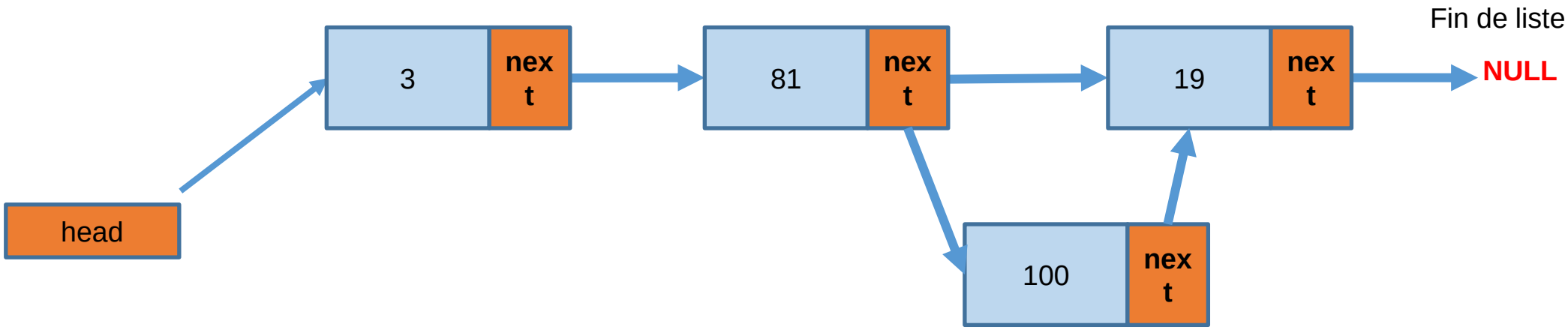
```
Node *head = NULL;
// autres instructions
head = insererFin(head, 40);
head = insererFin(head, 30);
head = insererFin(head, 120);
```

Listes simplement chaînées

3 - Implémenter les primitives : Insérer après une valeur

Insérer un nœud à la fin consiste à :

1. Allouer un nouveau nœud.
2. Si la liste est vide la nouvelle liste est notre nouveau nœud.
3. Si non, parcourir la liste jusqu'au dernier Node.
4. Le suivant du dernier Node est le nouveau élément.



3 - Implémenter les primitives : Insérer après une valeur

```
Node* insererApres(Node* head, int valeur, int cle) {
    Node* temp = head;
    while (temp != NULL && temp->val != cle)
        temp = temp->next;

    if (temp == NULL) {
        printf("Valeur %d non trouvée.\n", cle);
        return head;
    }

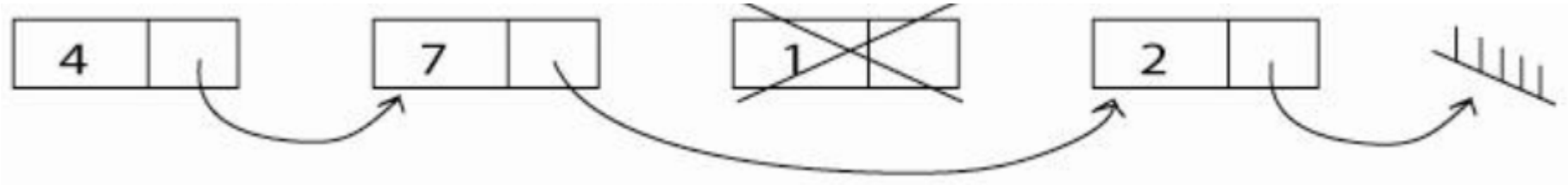
    Node* nouveau = creerNoeud(valeur);
    nouveau->next = temp->next;
    temp->next = nouveau;
    return head;
}
```

Listes simplement chaînées

3 - Implémenter les primitives : Supprimer un élément

Pour supprimer un élément :

1. Si la valeur à supprimer est la première valeur, faite pointer head sur l'élément suivant de l'élément à supprimer.
2. Si non : Chercher l'élément précèdent de l'élément à supprimer.
3. Faire pointer le suivant de cette élément au suivant de l'élément à supprimer.
4. Libérer la mémoire de l'élément supprimé.



Listes simplement chaînées

3 - Implémenter les primitives : Supprimer un élément

```
Node* supprimerValeur(Node* head, int cle) {
    if (head == NULL) return NULL;

    if (head->val == cle) {
        Node* temp = head;
        head = head->next;
        free(temp);
        return head;
    }

    Node* temp = head;
    while (temp->next != NULL && temp->next->val != cle)
        temp = temp->next;

    if (temp->next == NULL) {
        printf("Valeur %d non trouvée.\n", cle);
        return head;
    }

    Node* aSupprimer = temp->next;
    temp->next = aSupprimer->next;
    free(aSupprimer);
    return head;
}
```

Si l'élément à supprimer est le premier élément de la liste

Listes simplement chaînées

3 - Implémenter les primitives : Afficher une liste

Pour afficher une liste :

1. Déclarer un pointeur **tmp** qui pointe sur le premier élément.
2. Tant que **tmp** n'est pas NULL afficher sa valeur et passer à l'élément suivant.

```
void afficherListe(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->val);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Listes simplement chaînées

3 - Implémenter les primitives : Longueur d'une liste

Pour afficher une liste :

1. Déclarer un pointeur **tmp** qui pointe sur le premier élément.
2. Tant que **tmp** n'est pas NULL incrémenter le compteur et passer à l'élément suivant.

```
int longueurListe(Node* tete) {
    int count = 0;
    Node* temp = tete;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    return count;
}
```

Listes simplement chaînées

Un exemple complet :

```
int main() {
    Node* liste = NULL;

    liste = insererDebut(liste, 3);
    liste = insererDebut(liste, 2);
    liste = insererDebut(liste, 1);
    liste = insererFin(liste, 4);

    afficherListe(liste);

    liste = supprimerValeur(liste, 2);
    afficherListe(liste);

    printf("Longueur : %d\n", longueurListe(liste));

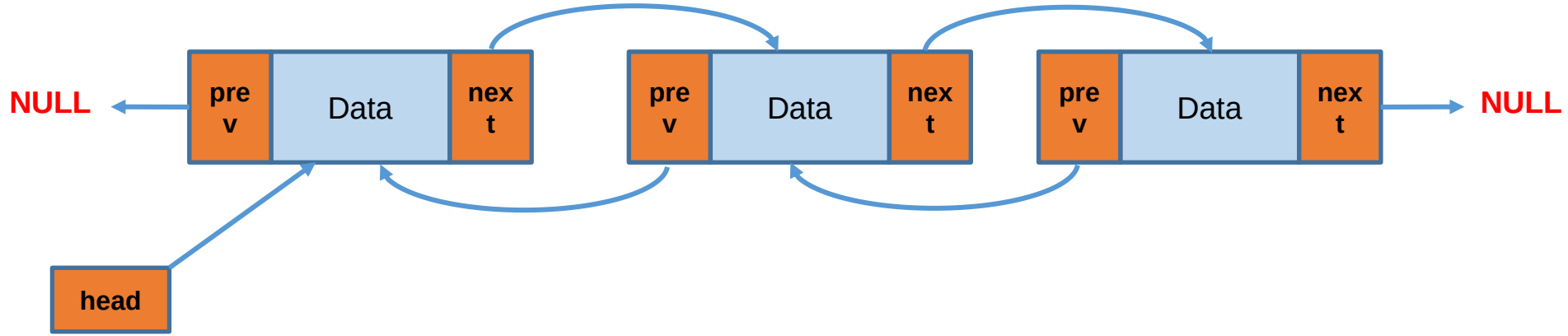
    return 0;
}
```

HESTIM
ENGINEERING & BUSINESS SCHOOL

LES LISTES DOUBLEMENT CHAÎNEES

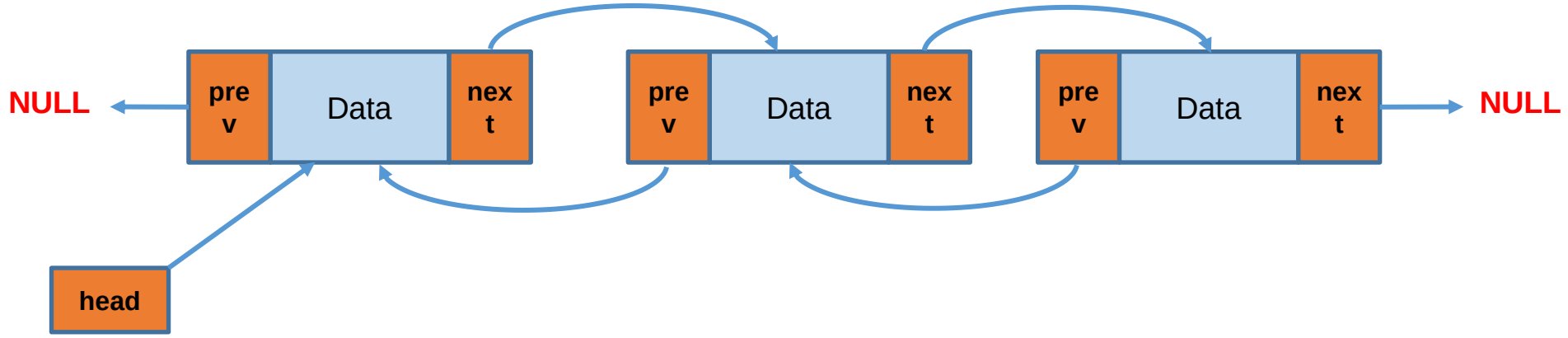
Listes doublement chaînées

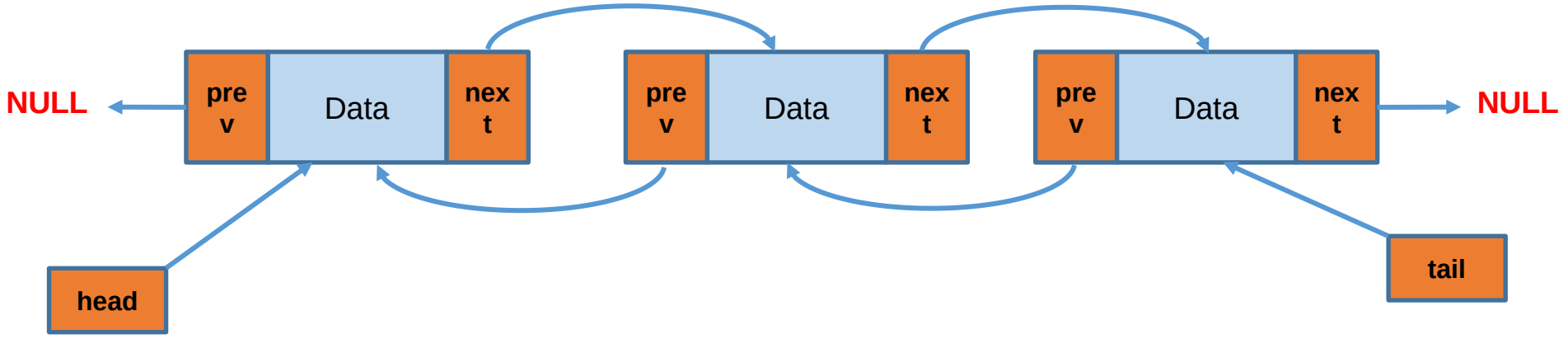
- Encore appelée liste symétrique, une liste doublement chaînée est une liste dans laquelle chaque élément de la liste a deux pointeurs, l'un pointant sur son suivant et l'autre sur son précédent. La tête d'une liste doublement chaînée à son précédent qui pointe à Null, tandis que la queue à son suivant qui pointe à Null.



Listes doublement chaînées

- Deux variantes :





Listes doublement chaînées

1 - Implémenter un Noeud :

L'implémentation d'une liste chaînées commence par implémenter une cellule (un nœud) :

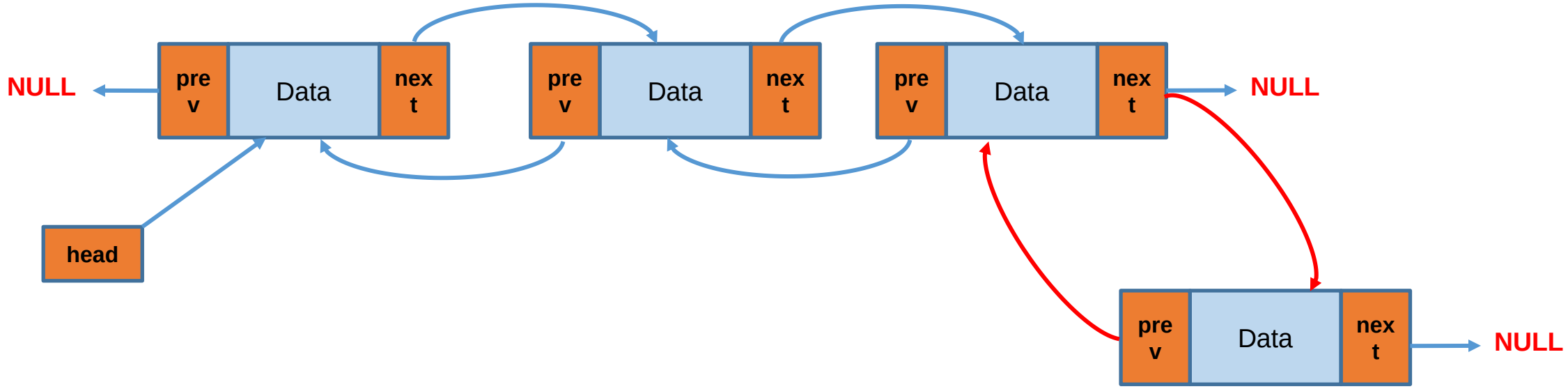


Node

```
typedef struct node {
        struct node *prev;
        // données
        struct node *next;
} Node;
```

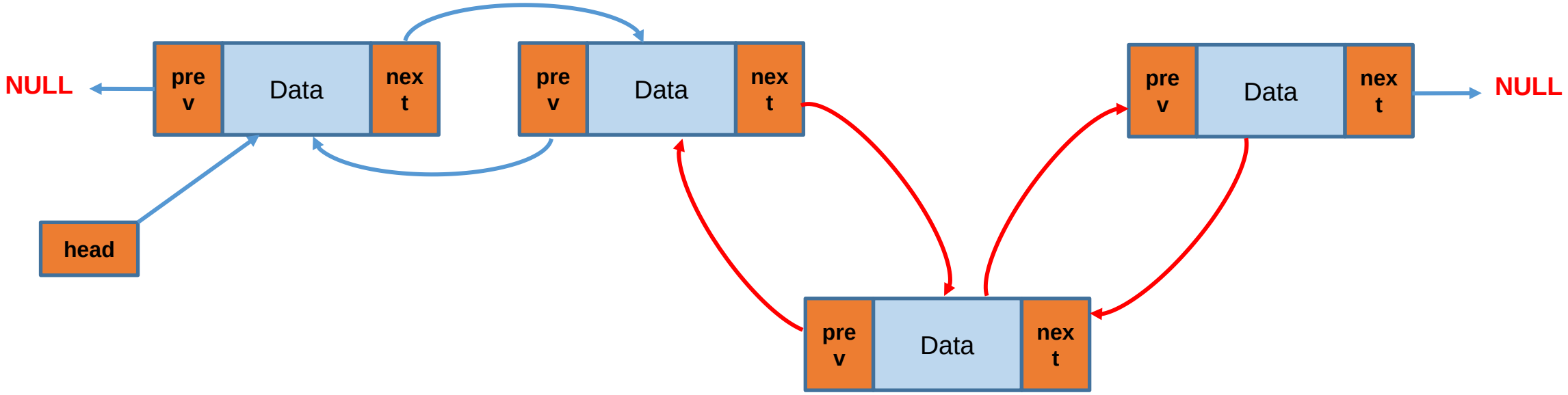
Listes doublement chaînées

2 – Primitives : Ajouter un élément à la fin



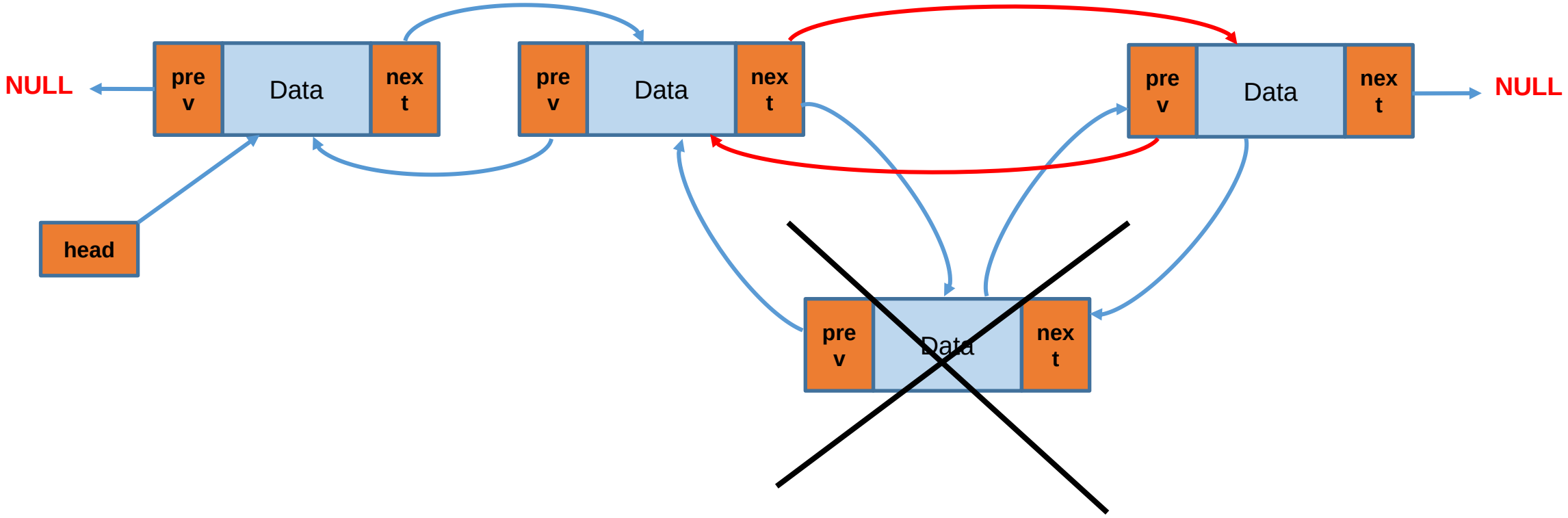
Listes doublement chaînées

2 – Primitives : Insérer un élément



Listes doublement chaînées

2 – Primitives : Supprimer un élément



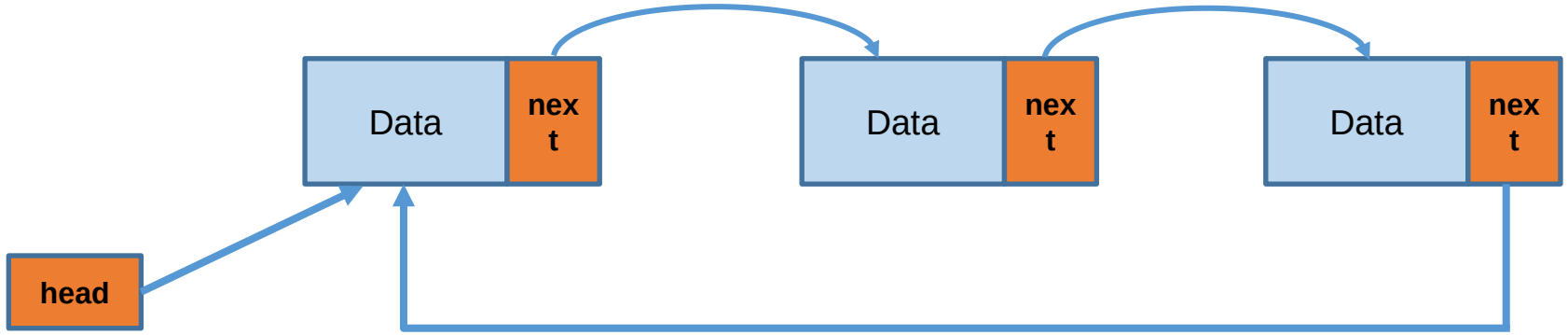
HESTIM
ENGINEERING & BUSINESS SCHOOL

LES LISTES CHAÎNEES CIRCULAIRES SIMPLES

Listes chaînées circulaire simple

C’est une liste simplement chaînée où le dernier nœud ne pointe pas sur NULL, mais sur le premier nœud, formant ainsi un cercle. Caractéristiques :

- Pas de début ni de fin “absolue”.
- Permet un parcours en boucle.
- Utilisée pour gérer des cycles (ex. tour de jeu, file circulaire...).



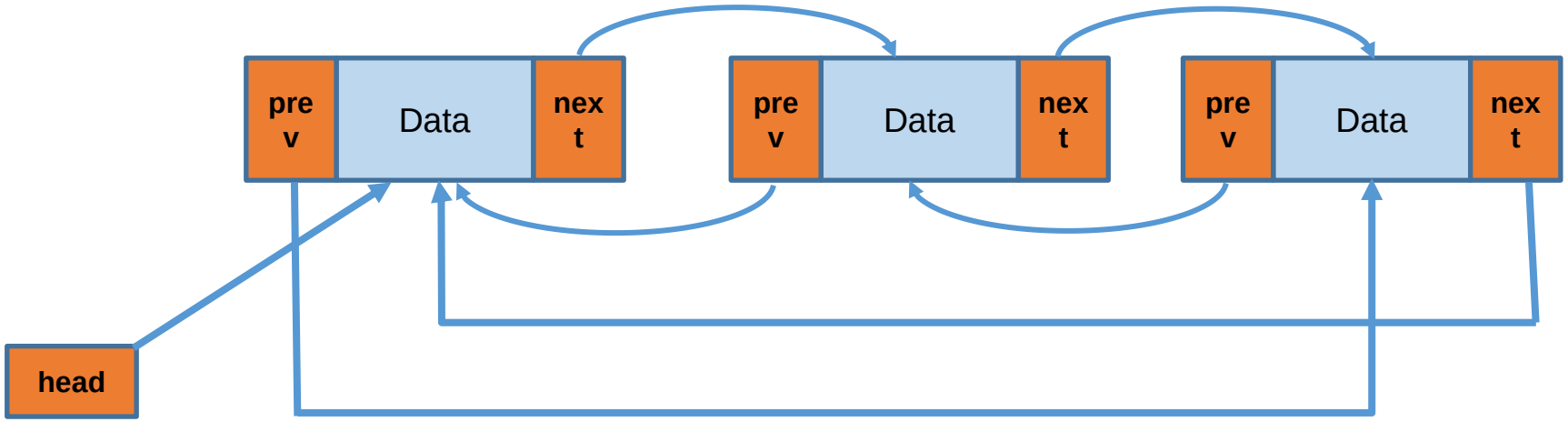
HESTIM
ENGINEERING & BUSINESS SCHOOL

LES LISTES DOUBLEMENT CHAÎNEES CIRCULAIRES

Listes doublement chaînées circulaire simple

C’est une liste doublement chaînée où :

- le pointeur next du dernier nœud pointe vers le premier,
- et le pointeur prev du premier nœud pointe vers le dernier.



Les Piles et Files

Introduction

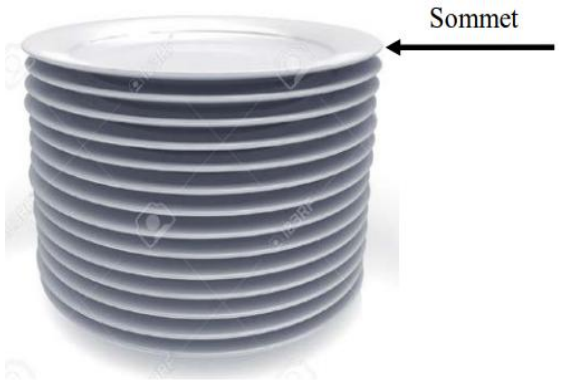
- Ce sont des structures de données linéaires
- Ce sont des structures de données ordonnées, mais qui ne permettent l'accès qu'à une seule donnée.
- Elles sont très souvent utiles, elles servent à mémoriser des évènements en attente de traitement.

La Pile

Les Piles

Définition d'une Pile (Stack)

- La Pile est une structure de données linéaire qui suit le principe LIFO (Last In First Out) → le dernier élément ajouté est le premier à sortir.





Les Piles

Exemple d'application d'une Pile (Stack):

- Gestion du retour arrière (Undo)
- Navigation dans un navigateur web
- Pile d'appel des fonctions (Call Stack)
- Système d'annulations dans les jeux
- ….

Les Piles

Les primitives d'une pile :

- **Empiler** : Ajouter un élément (push)
- **Dépiler** : Enlever un élément (pop)
- **Consulter** : Voir l'élément accessible au sommet (peek)
- **EstVide** : Tester si la pile est vide
- **EstPleine** : Tester si la pile est pleine (que pour les tableaux)

Les Piles

Implémentation d'un Pile :

- Il n'y a pas de structures spécifiques prévues dans tous langages de programmation, il faut donc les créer de toute pièce.

- Pour les piles, on utilisera :

  - Un tableau unidimensionnel en cas de piles de hauteur maximale prévisible

  - Une liste en cas de longueur très variable.

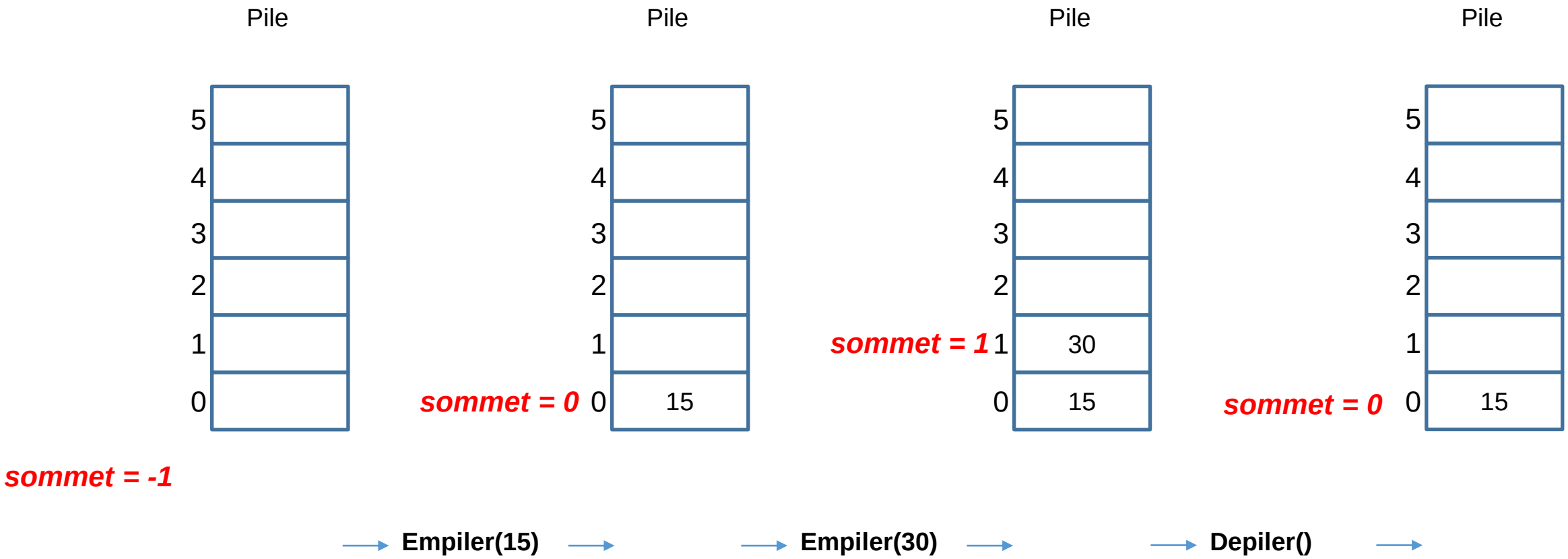
Les Piles

Implémentation d'un Pile : Par un tableau

- Une pile sera implémentée par un tableau avec une taille fixe, et une variable entière "sommet" qui contient l'indice du dernier élément de la pile.
- Pour une pile vide : **sommet = -1**
- Pour faciliter la gestion le tableau et l'entier seront mit dans un type enregistrement

Les Piles

Implémentation d'un Pile : Par un tableau



Les Piles

Implémentation d'un Pile : Par un tableau

- Modéliser une pile avec une structure :

```
#define MAX 500

typedef struct {

    int data[MAX];
    int sommet;

} Pile;
```

Les Piles

Implémentation d'un Pile : Par un tableau

- Les primitives :

```
int estVide(Pile* p) {
    return p->sommet == -1;
}


int estPleine(Pile* p) {
    return p->sommet == MAX - 1;
}
```

```
int consulter(Pile* p) {
    if (estVide(p)) {
        exit(1); // Échec de l'accès au sommet
    }
    return p->data[p->sommet];
}
```

Les Piles

Implémentation d'un Pile : Par un tableau

- Les primitives :

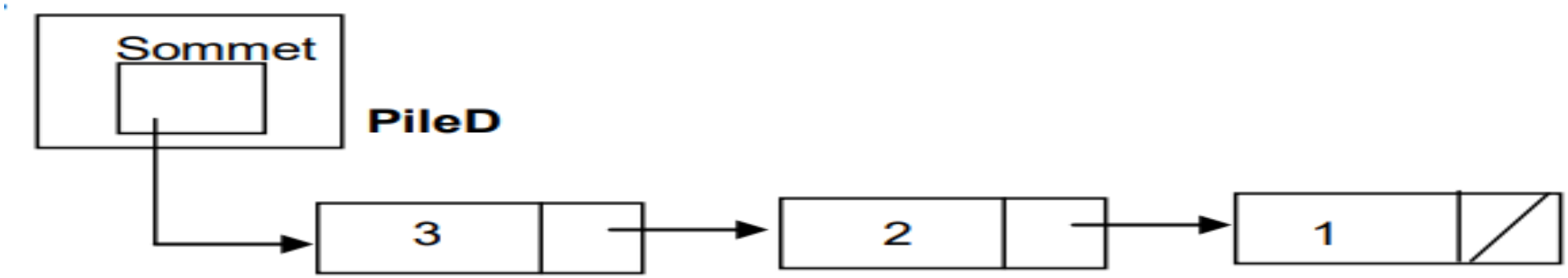
```
void empiler(Pile* p, int val) {
    if (estPleine(p)) {
        printf("Pile pleine");
        exit(1);
    }
    p->sommet++;
    p->data[p->sommet] = val;
}
```

```
int depiler(Pile* p) {
    if (estVide(p)) {
        printf("Pile vide");
        exit(1);
    }
    int val = p->data[p->sommet];
    p->sommet--;
    return val;
}
```

Les Piles

Implémentation d'un Pile : Avec une liste chaînée

- Une pile dynamique est une liste à laquelle on attache un pointeur sommet. C'est un enregistrement à une seule case : pointeur qui pointe la dernière valeur traitée dans la liste (sommet).



Les Piles

Implémentation d'un Pile : Avec une liste chaînée

Les primitives :

- Empiler : Ajouter un élément au début
- Dépiler : Supprimer le premier élément
- Consulter : Retourner la valeur du premier élément
- EstVide : Tester si la liste est vide

HESTIM
ENGINEERING & BUSINESS SCHOOL

La File

Les Files

Définition d'une File

- La File est une structure de données linéaire qui suit le principe FIFO (First In First Out) → le premier élément ajouté est le premier à sortir.



Les Files

Exemple d'application d'une File :

- File d'attente dans un service (banque, hôpital, guichet)
- Gestion des impressions (spooler d'impression)
- Gestion des requêtes dans un serveur web
- Simulation de circulation / gestion d'un pont, tunnel, péage
- ….

Les Files

Les primitives d'une file :

- **Enfiler** : Ajouter un élément
- **Défiler** : Enlever un élément
- **Consulter** : Voir l'élément prochain à sortir
- **EstVide** : Tester si la file est vide
- **EstPleine** : Tester si la file est pleine (que pour les tableaux)

Les Files

Implémentation d'une File :

- Il n'y a pas de structures spécifiques prévues dans tous langages de programmation, il faut donc les créer de toute pièce.

- Pour les files, on utilisera :

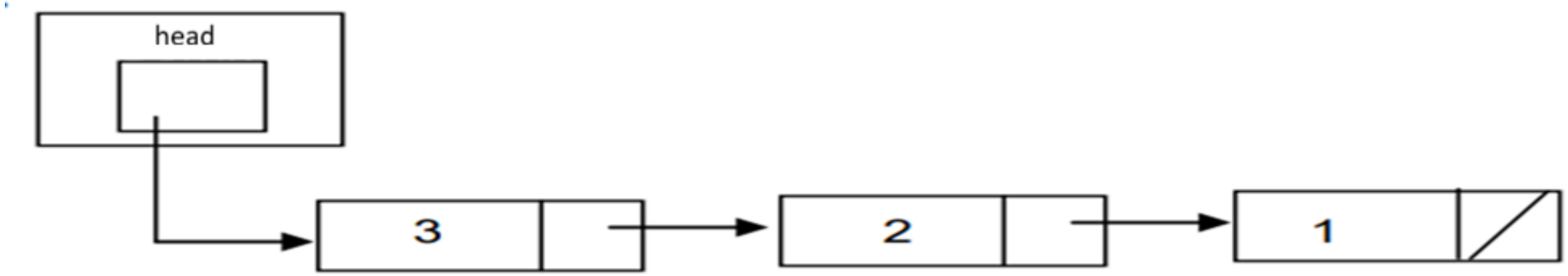
  - Une liste en cas de longueur très variable.

  - Un tableau unidimensionnel en cas de files de hauteur maximale prévisible

Les Files

Implémentation d'un File : Avec une liste chaînée

- Une file dynamique est une liste à laquelle on attache un pointeur sommet. C'est un enregistrement à une seule case : pointeur qui pointe la dernière valeur traitée dans la liste (tête).



Les Files

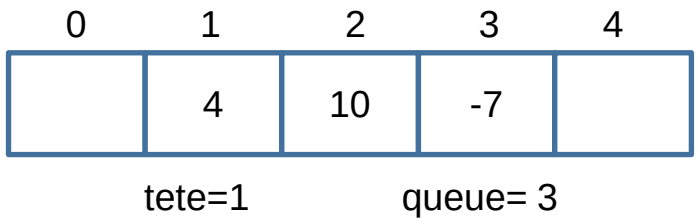
Implémentation d'un File : Avec une liste chaînée

Les primitives :

- Enfiler : Ajouter un élément à la fin
- Défiler : Supprimer le premier élément
- Consulter : Retourner la valeur du premier élément
- EstVide : Tester si la liste est vide

Les Files

Implémentation d'un File : Par un tableau

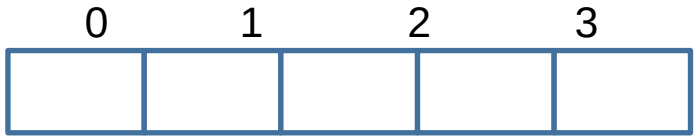


- Une file sera implémentée par un tableau avec une taille fixe, et deux variables entières "**tete**" qui contient l'indice de l'élément à la tête de la file et "**queue**" qui contient l'indice du dernier élément de la file.

Les Files

Implémentation d'un Pile : Par un tableau

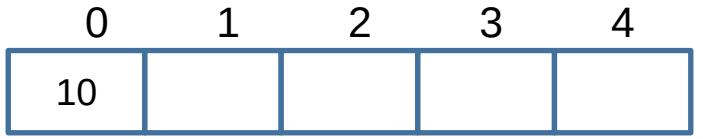
File



queue= -1 tete=0

Enfiler(10)

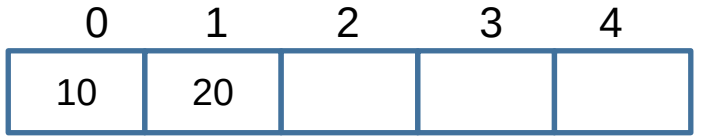
File



tete =0

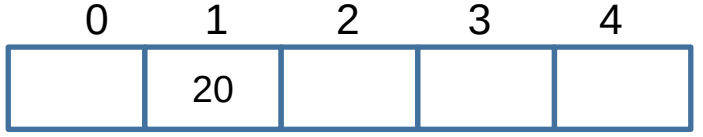
queue =0

Enfiler(20)



tete =0 queue =1

Defiler()



tete =1

queue =1

Defiler()



queue =1 tete =2

Les Files

Implémentation d'un File : Par un tableau

- Modéliser une file avec une structure :

```
#define MAX 500

typedef struct {

    int data[MAX];
    int tete;
    int queue;
    int taille;

} File;
```

Les Files

Implémentation d'un Pile : Par un tableau

- Les primitives :

```
void initialiser(File* f) {
    f->tete = 0;
    f->queue = -1;
    f->taille = 0;
}

int estVide(File* f) {
    return f->taille==0;
}

int estPleine(File* f) {
    return f->taille == MAX;
}
```

```
int consulter(File* f) {
    if (estVide(f)) {
        printf("Erreur: La file est vide.\n");
        exit(1); // File vide
    }
    int valeur = f->data[f->tete];
    return valeur;
}
```

Les Files

Implémentation d'un File : Par un tableau

- Les primitives :

```
void enfiler(File* f, int valeur) {
    if (estPleine(f)) {
        printf("La file est pleine.\n");
        exit(1); // File pleine
    }
    f->queue = ( f->queue + 1 ) % MAX;
    f->data[f->queue] = valeur;
    f->taille++;
}
```

```
int defiler(File* f) {
    if (estVide(f)) {
        printf("La file est vide.\n");
        exit(1); // File vide
    }
    int valeur = f->data[f->tete];
    f->tete = (f->tete + 1) % MAX;
    f->taille--;
    return valeur;
}
```

Les arbres binaires

Plan

- **Les arbres**
  - Définitions
  - Applications
  - Terminologies et propriétés
- **Les arbres binaires**
  - Définition
  - Implémentation
  - Parcours en Profondeur / Largeur
  - Les arbres binaires de recherche
  - Les arbres binaires parfaits
  - Le Tas

Introduction

Les structures de données linéaires :

- Les tableaux
- Les chaînes de caractères
- Les listes
- Les piles
- Les files
- …

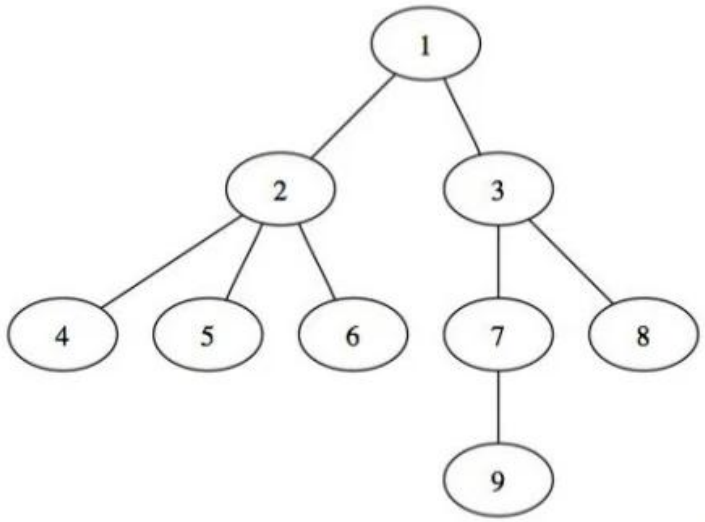
Les structures de données non linéaires :

- Les arbres
- Les graphes
- …

Introduction

Définition:

- Un arbre (tree en anglais) est une structure dynamique et non linéaire
- Un arbre est un ensemble de nœuds tel que :
  - ∃ un nœud unique appelé racine qui n'a pas de supérieur
  - Tous les autres nœuds sont atteints à partir de r par une branche unique
- Représentation graphique



Introduction

Applications:

Les arbres ont plusieurs applications fondamentales en Informatique, et ils sont bien adaptés à la représentation d'informations homogènes et organisées en hiérarchie.

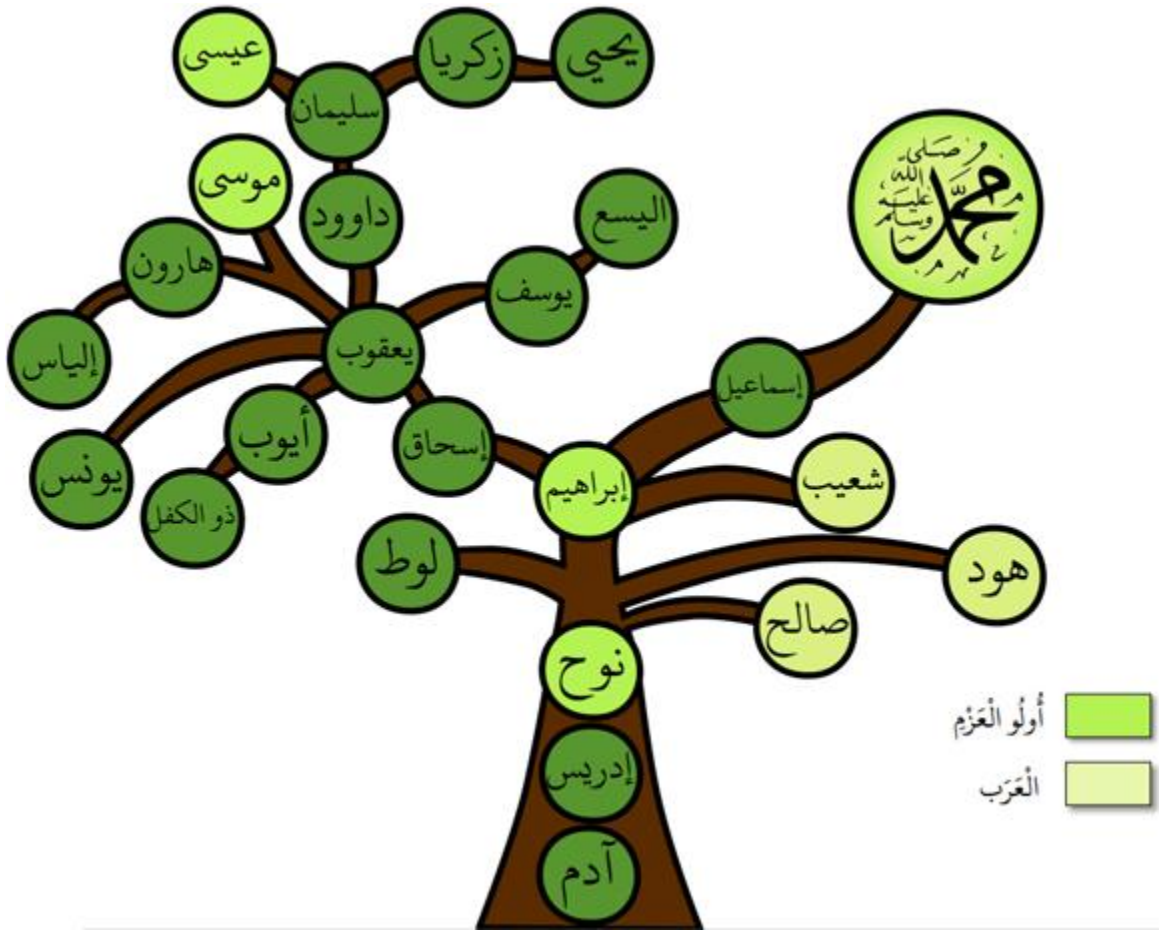
Exemples :

- Modèles pour les structures hiérarchisées
  - Arbre familiale
  - Arbre généalogique
  - Système de fichiers (Windows et Linux)
  - Organigramme d'une entreprise
- Les expressions arithmétiques
- Algorithme de recherche rapide et efficace
- Intelligence artificielle (Arbre de décision)
- …

Introduction

Applications:

Exemple 1 : Arbres généalogiques :



Introduction

Applications:

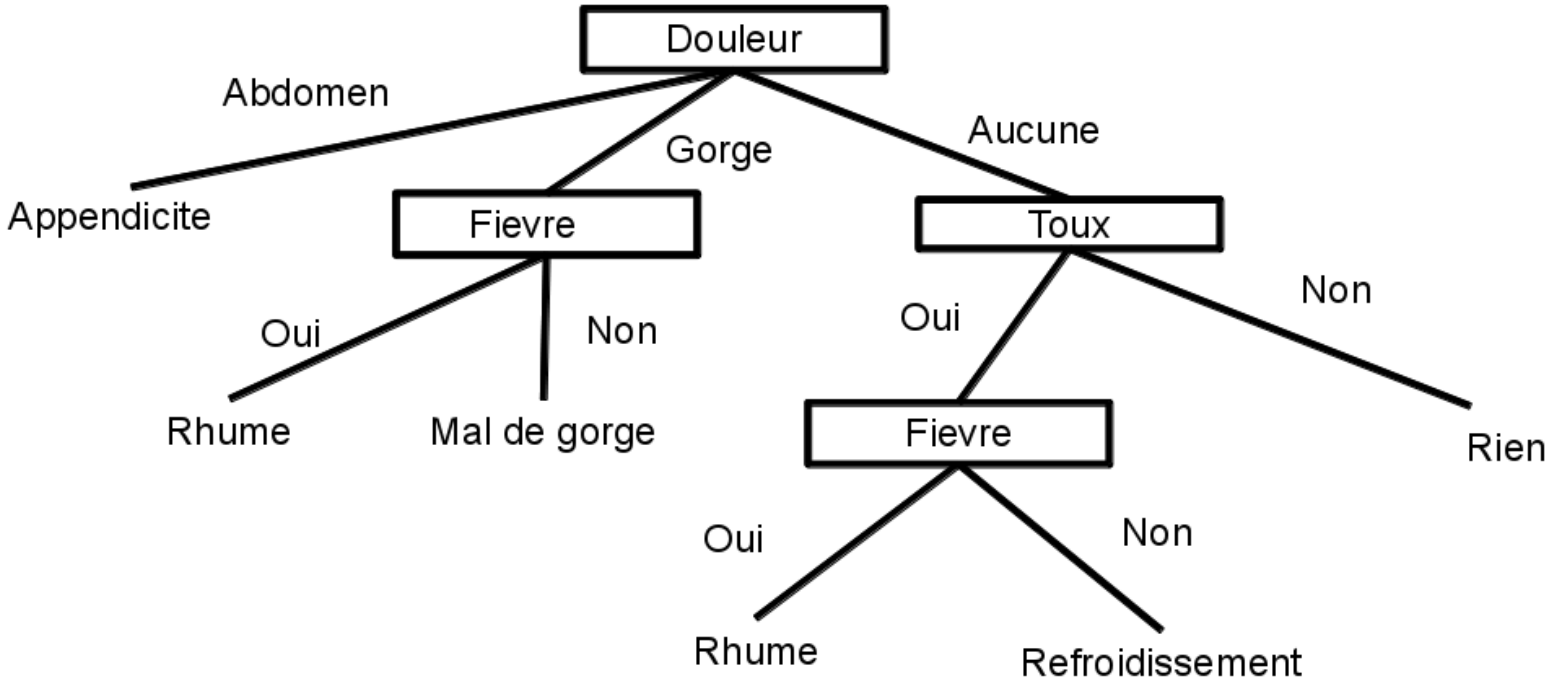
Exemple 2 : Expressions arithmétiques

4*3 + 2*7-5

Introduction

Applications:

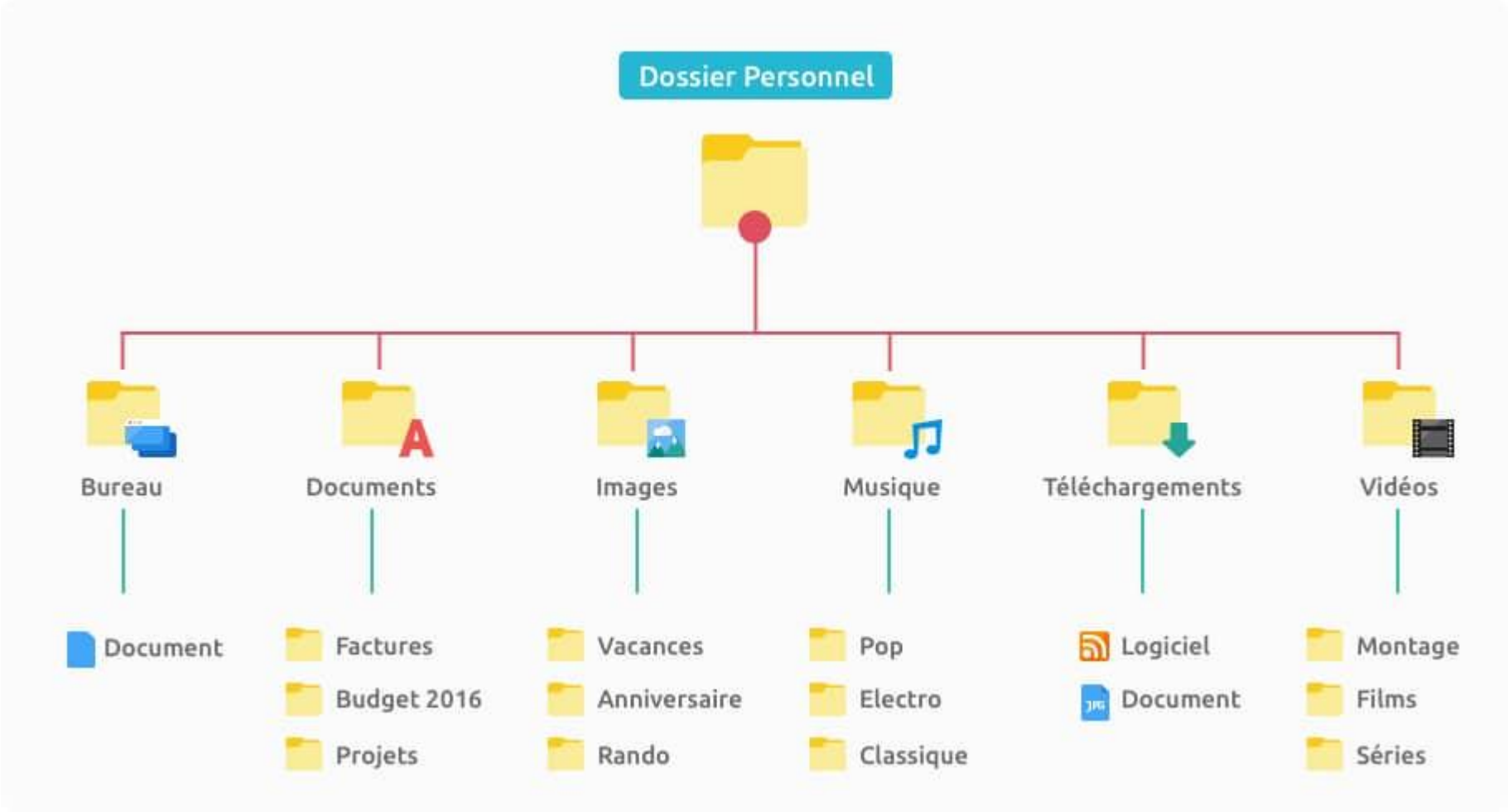
Exemple 3 : Système expert



Introduction

Applications:

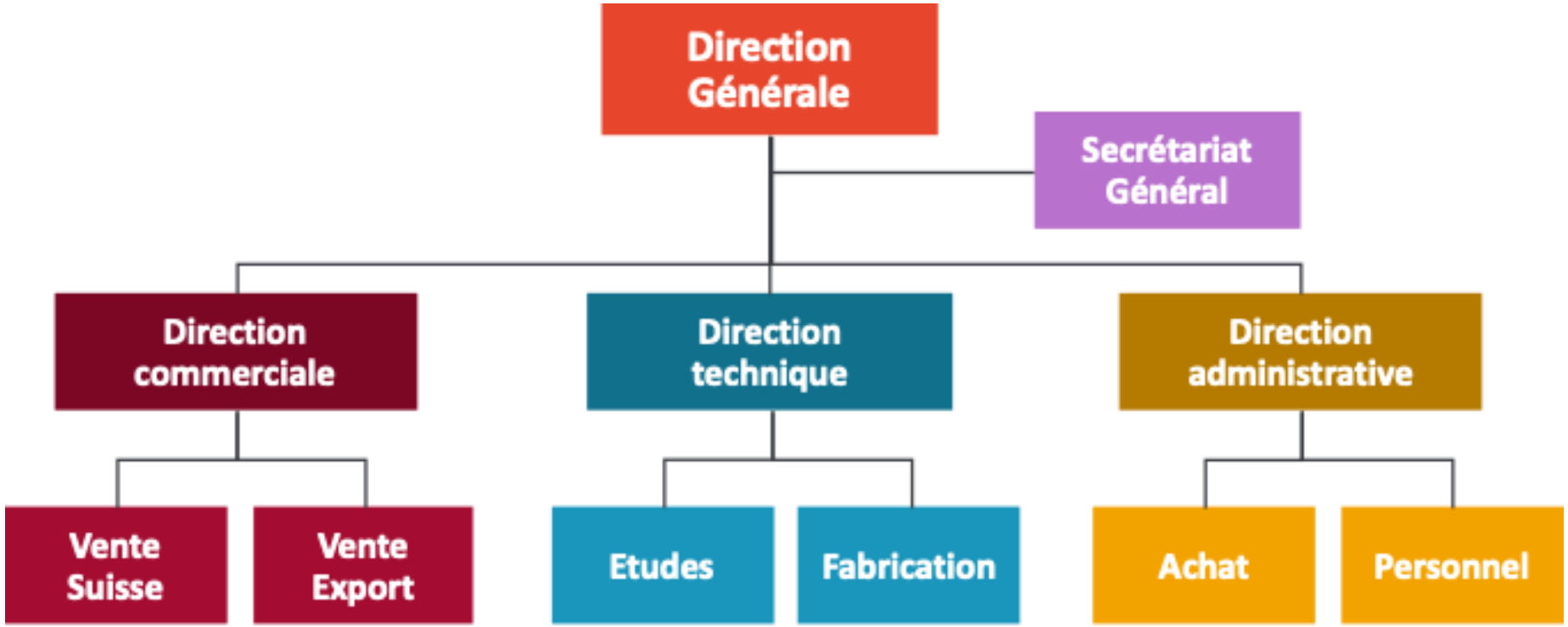
Exemple 4 : Système de fichiers



Introduction

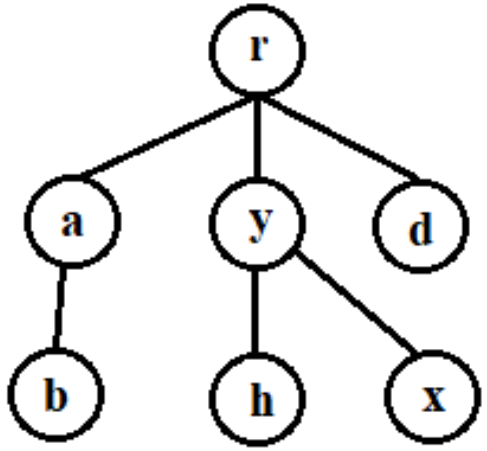
Applications:

Exemple 5 : Organisation d'une entreprise



Terminologie

- *Racine : Le seul nœud sans père (r est une racine)*
- *Feuille : Un nœud qui n'a pas de successeur (b, h, d et x sont des feuilles)*
- *Frère : Un nœud de même parent (a et y sont des frères)*
- *Sous-arbre : Une portion de l'arbre (Le nœud y avec ses fils)*
- *Branche : Une suite de nœuds connectés de la racine à une feuille (r-a-b; r-y-h; r-y-x ; r-d)*
- *Taille : Le nombre de nœuds qu'il possède (La taille est 7)*
- *Degré d'un nœud : Le nombre de ses fils (degre(r)=3 ; degre(a)=1; degre(h)=0)*
- *Degré d'un arbre : Le max des degré des nœuds (Le degré de l'arbre ci-dessous est 3)*

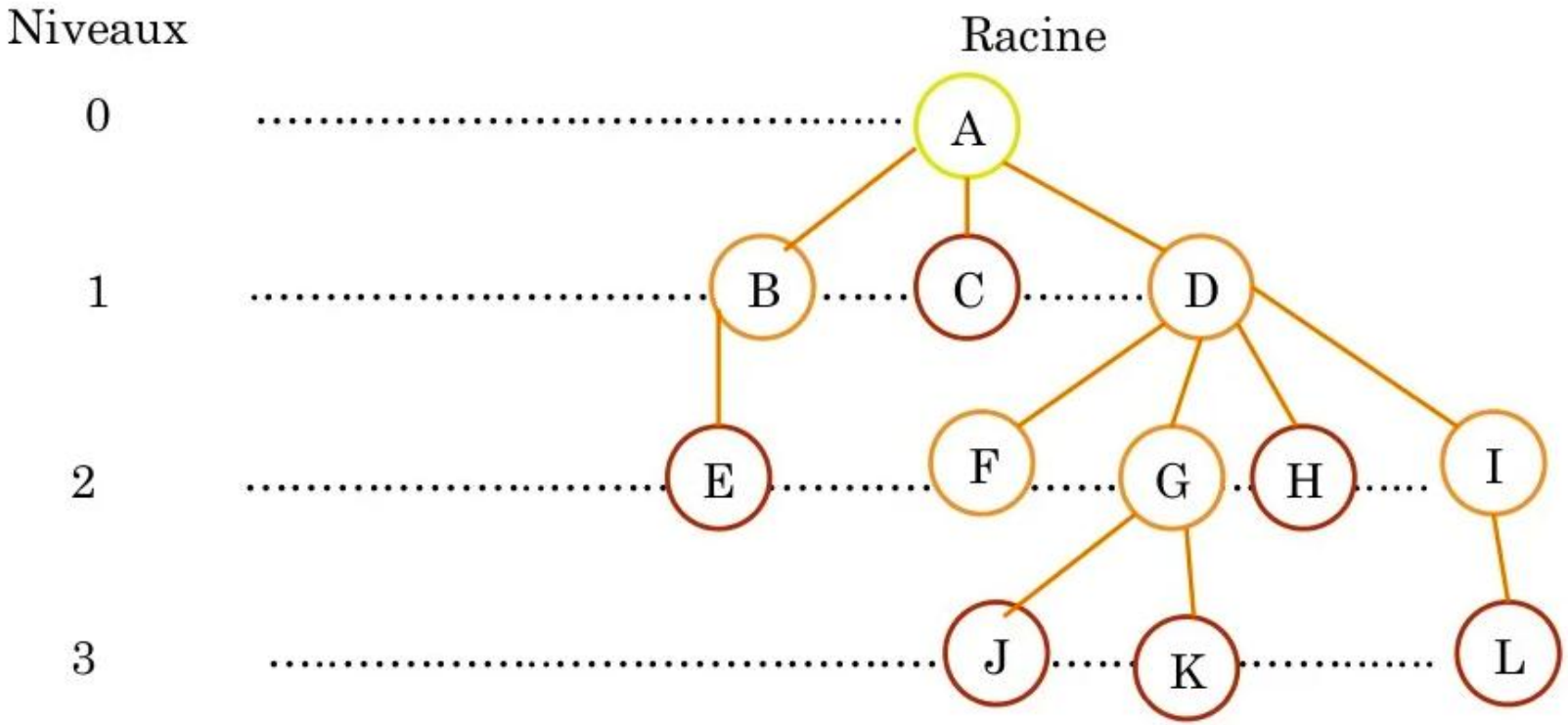


Terminologie

Le niveau d'un nœud (profondeur): *La distance qui le sépare de la racine*

Le niveau de la racine est : 0

Le niveau de chaque nœud est le niveau du père plus 1

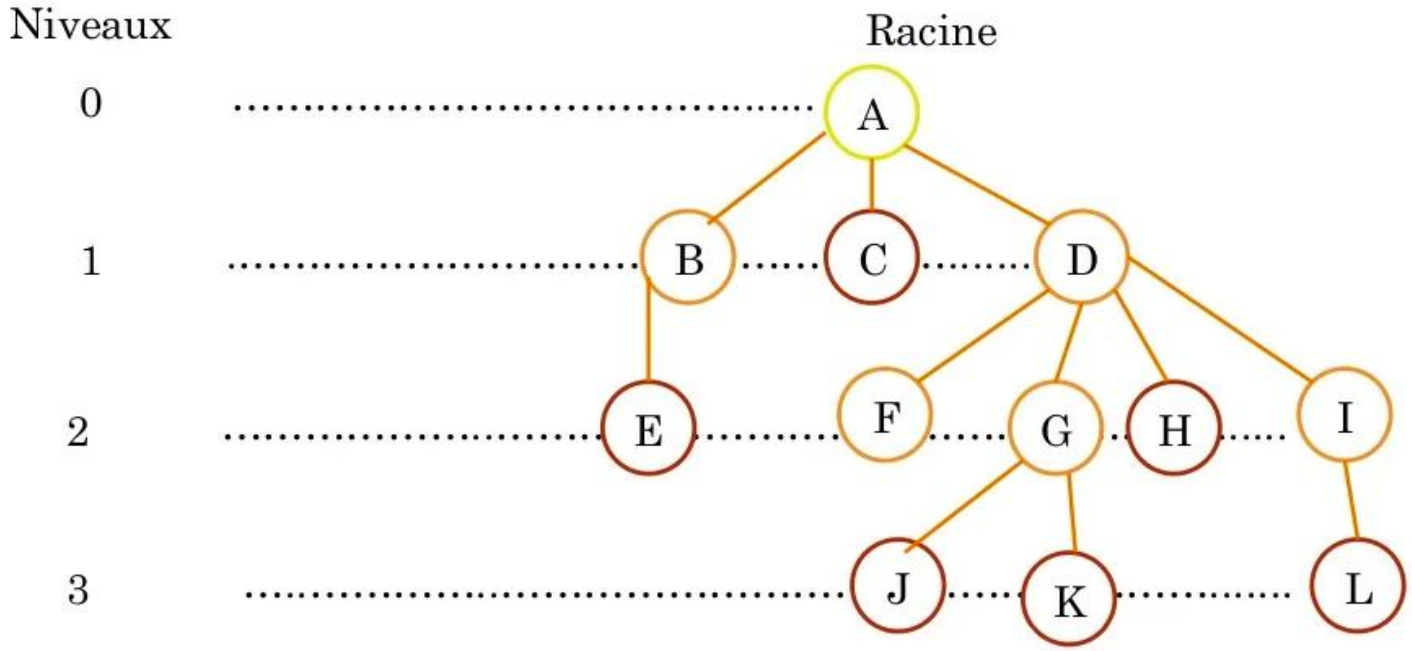


Terminologie

Hauteur d'un arbre (profondeur d'un arbre) : est le plus grand niveau des nœuds, c-à-d la distance entre la racine est la feuilles la plus lointaine

Par convention la hauteur d'un arbre vide est : -1

Hauteur = 3



Terminologie

Classification:

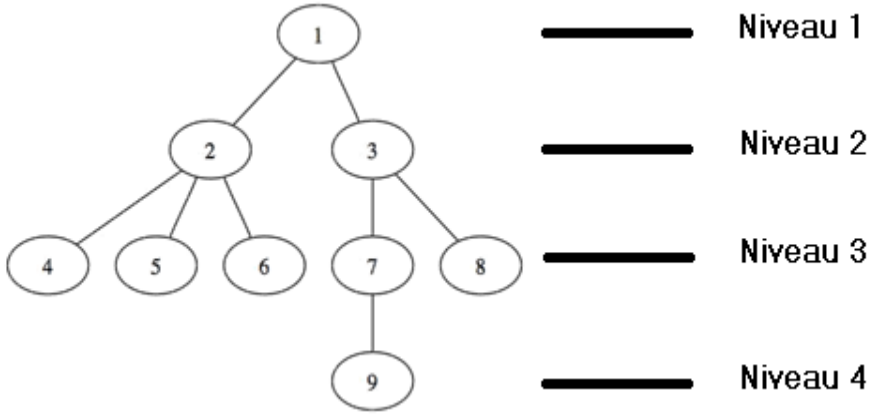
On peut classer les arbres selon deux critères de classification :

- Ordre de hiérarchie
- Arité

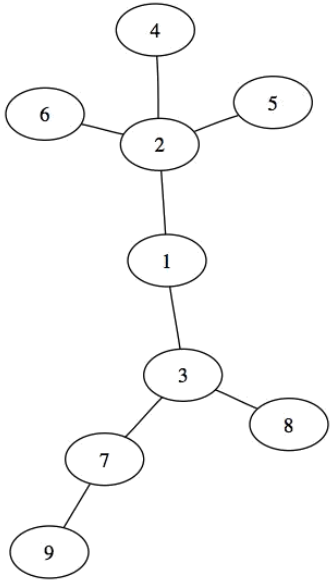
Terminologie

Classification: Classification selon l'ordre de hiérarchie:

- Enraciné :



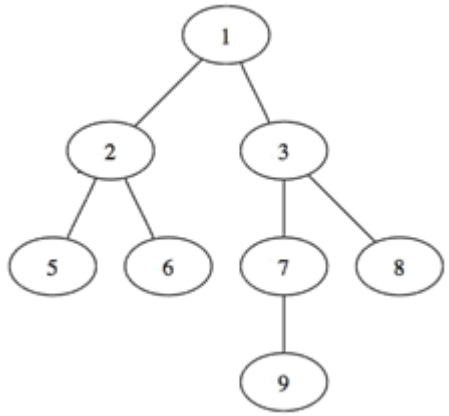
- Non enraciné :



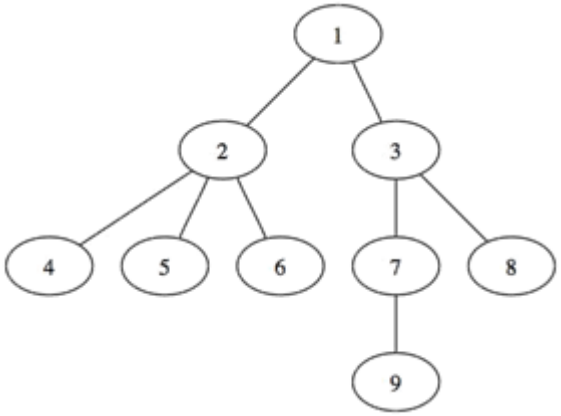
Terminologie

Classification: Classification selon l'arité des nœuds:

- Binaires :



- N-Aires:



Terminologie

Les arbres binaires :

- Les arbres binaires complets
- Les arbres binaires de recherche
- Les arbres binaires équivalents
- Les arbres binaires parfaits
- Les arbres AVL
- Les B-Arbres
- …

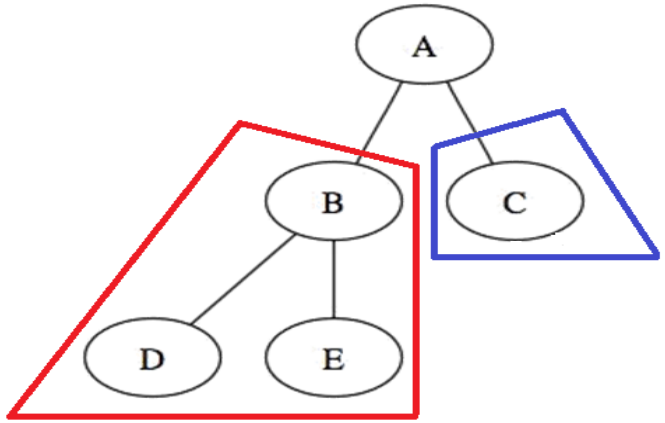
Implémentation

Les arbres binaires forment une structure de données qui peut être définie récursivement de la manière suivante :

Un arbre binaire est :

– soit vide,

– soit composé d'une racine portant une valeur et d'une paire d'arbres binaires, appelés fils gauche et droit.



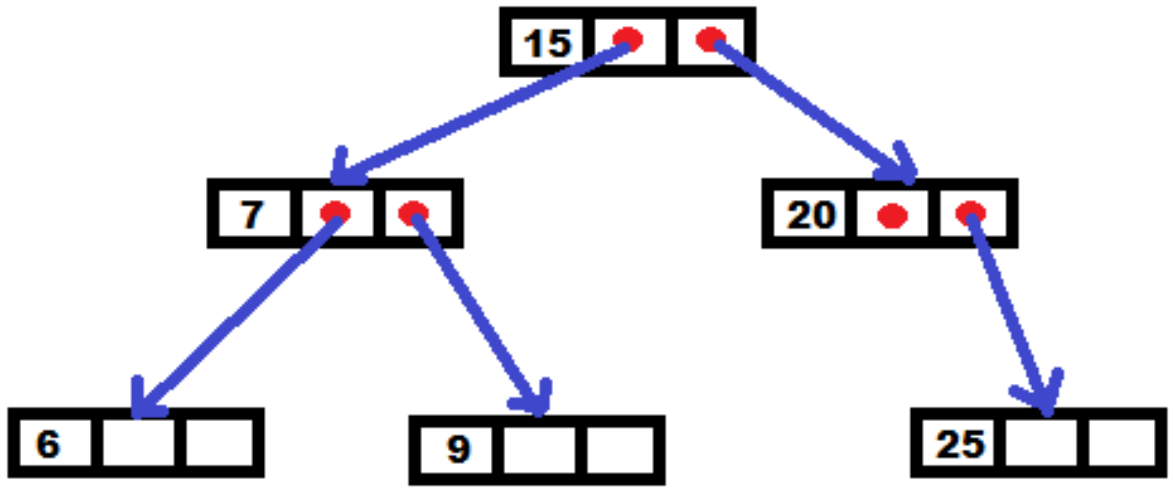
Implémentation

Un arbre est un ensemble de nœud relié entre eux, implémenter un arbre c'est tout d'abord modéliser un nœuds, en Langage C un nœud est une structure de trois éléments :

```
typedef struct Node {

    int valeur;
    struct Node *gauche;
    struct Node *droite;

} Node;
```



Pour déclarer un arbre vide :

```
Node *root = NULL;
```

Implémentation

Nous allons implémenter quelques primitives pour manipuler un arbre binaire :

1. Créer un nœud
2. Tester si l'arbre est vide
3. Obtenir l'arbre de gauche
4. Obtenir l'arbre de droite
5. Parcourir les éléments d'un arbre
6. La taille d'un arbre
7. La hauteur d'un arbre
8. Le nombre des feuilles
9. La valeur minimale
10. La valeur maximale
11. Tester si un arbre est une feuille
12. Insérer un nœud
13. Rechercher un nœud
14. Supprimer un nœud

Implémentation

1 - Créer un nœud

```
Node* creerNoeud(int val) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->valeur = val;
    newNode->gauche = NULL;
    newNode->droite = NULL;
    return newNode;
}
```

Implémentation

2 - Tester si l'arbre est vide

```
int estVide(Node* root) {
    return root==NULL;
}
```

Implémentation

3 – Obtenir l'arbre de gauche

```
Node* gauche(Node* root) {
    return root->gauche;
}
```

Implémentation

4 – Obtenir l'arbre de droite

```
Node* droite(Node* root) {
    return root->droite;
}
```

Implémentation

5 -Algorithmes de parcours :

Nous allons découvrir des algorithmes de parcours d'un arbre. Cela permet de visiter tous les nœuds de l'arbre et éventuellement appliquer une fonction sur ces nœuds.

Nous distinguerons deux types de parcours :

- Le parcours en profondeur permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante.

- Le parcours en largeur permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau un puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

5 -Algorithmes de parcours : Parcours en profondeur

Variantes du parcours en profondeur :

- Parcours préfixe
- Parcours infixe
- Parcours suffixe

Implémentation

5 -Algorithmes de parcours : Parcours en profondeur (Préfixe)

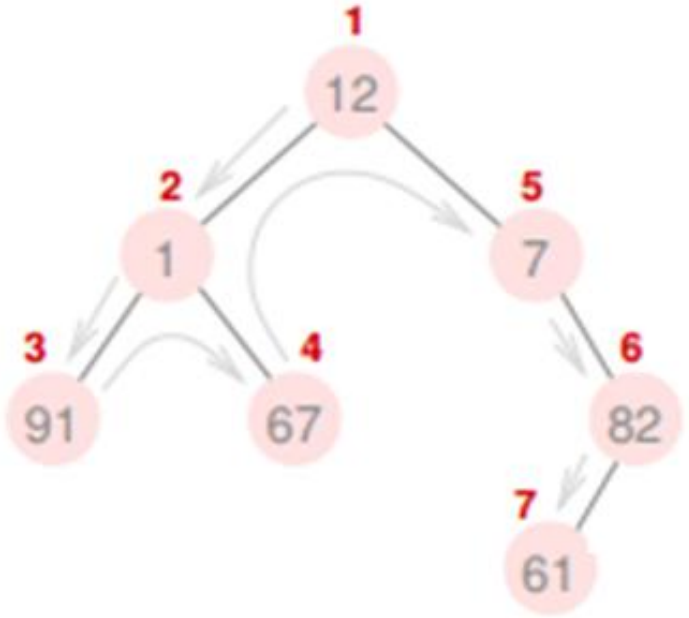
Parcours en profondeur préfixe :

Tout Nœud est suivi des nœuds de son

sous-arbre Gauche puis des nœuds

de son sous-arbre Droit

Le résultat d'affichage est :
12 ; 1 ; 91 ; 67 ; 7 ; 82 ; 61



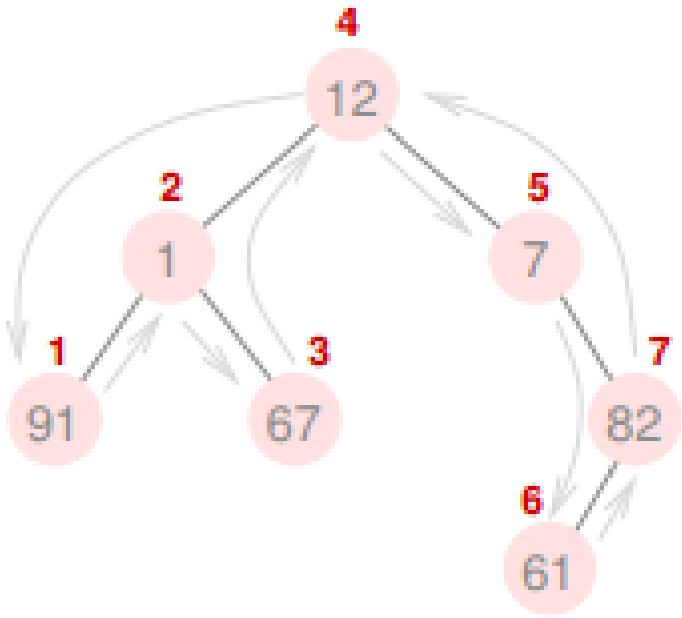
Implémentation

5 -Algorithmes de parcours : Parcours en profondeur (Infixe)

Parcours en profondeur infixe:

Tout Nœud est précédé des nœuds de son sous-arbre Gauche et suivi des noeuds de son sous-arbre Droit

Le résultat d'affichage est :
91 ; 1 ; 67 ; 12 ; 7 ; 61 ; 82



5 -Algorithmes de parcours : Parcours en profondeur (Postfixe)

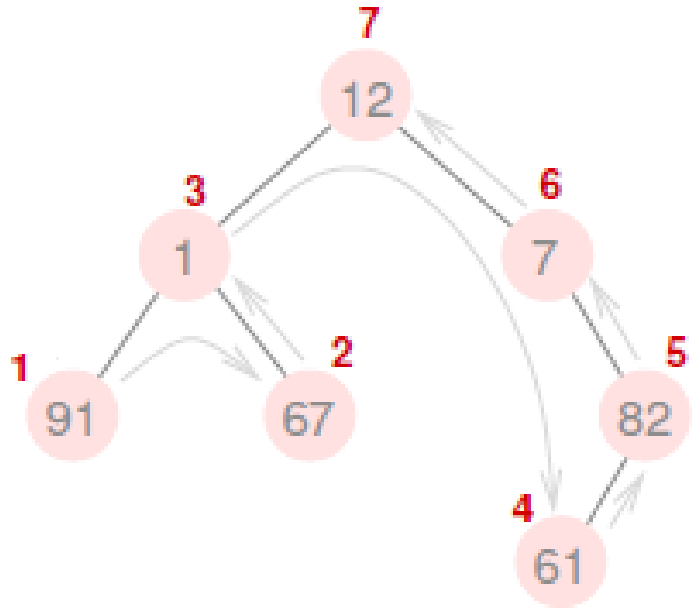
Parcours en profondeur suffixe (postfixe):

Tout Noeud est précédé des nœuds de

son sous-arbre Gauche et des nœuds

de son sous-arbre Droit

Le résultat d'affichage est :
91 ; 67 ; 1 ; 61 ; 82 ; 7 ; 12

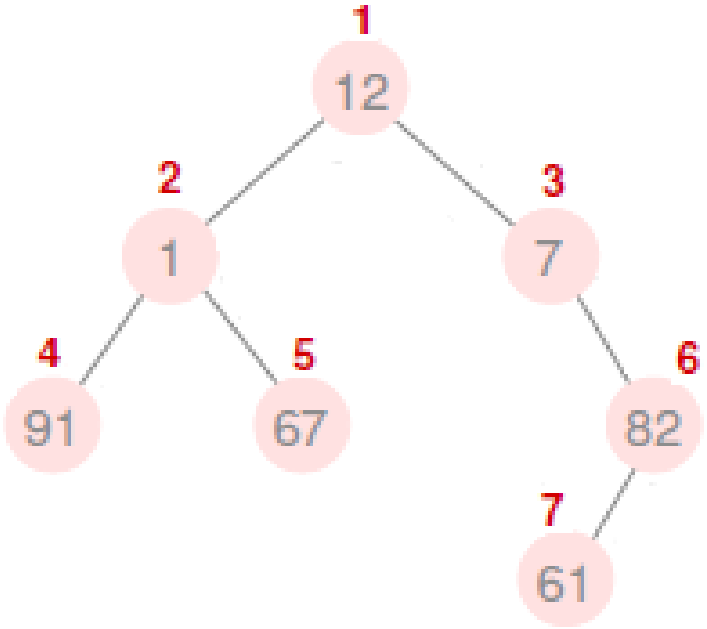


Implémentation

5 -Algorithmes de parcours : Parcours en largeur

Dans ce type de parcours on affiche les valeurs par niveau (les nœud de même profondeur).

Le résultat d'affichage est :
12 ; 1 ; 7 ; 91 ; 67 ; 82 ; 61



Implémentation

Exercices :

Ecrire la fonction **parcoursProfondeurPrefixe(Node* racine)** qui affiche les valeurs des nœuds par l'algorithme de parcours préfix.

```
void parcoursPrefixe(Node* racine) {
    if (racine != NULL) {
        printf("%d ", racine->valeur);
        parcoursPrefixe(racine->gauche);
        parcoursPrefixe(racine->droite);
    }
}
```

Implémentation

Exercices :

Ecrire la fonction **parcoursProfondeurInfixe(Node* racine)** qui affiche les valeurs des nœuds par l'algorithme de parcours infixe.

```
void parcoursInfixe(Node* racine) {
    if (racine != NULL) {
        parcoursInfixe(racine->gauche);
        printf("%d ", racine->valeur);
        parcoursInfixe(racine->droite);
    }
}
```

Implémentation

Exercices :

Ecrire la fonction **parcoursProfondeurPostfixe(Node* racine)** qui affiche les valeurs des nœuds par l'algorithme de parcours postfixe.

```
void parcoursPostfixe(Node* racine) {
    if (racine != NULL) {
        parcoursPostfixe(racine->gauche);
        parcoursPostfixe(racine->droite);
        printf("%d ", racine->valeur);
    }
}
```

Implémentation

Exercices :

Algorithme du parcours en largeur :

Lorsque nous sommes sur un nœud nous traitons ce nœud (par exemple nous l'affichons) puis nous mettons les fils gauche et droit non vides de ce nœud dans la file d'attente, puis nous traitons le prochain nœud de la file d'attente.

Au début, la file d'attente ne contient rien, nous y plaçons donc la racine de l'arbre que nous voulons traiter.

L'algorithme s'arrête lorsque la file d'attente est vide. En effet, lorsque la file d'attente est vide, cela veut dire qu'aucun des nœuds parcourus précédemment n'avait de sous arbre gauche ni de sous arbre droit.

Par conséquent, on a donc bien parcouru tous les nœuds de l'arbre.

Implémentation

```c
#define MAX 100
typedef struct {
    Node* elements[MAX];
    int tete;
    int queue;
    int taille;
} File;

File* creerFile();
estVide(File *f);
enfiler(File *f, Node* r);
defiler(File *, Node* r);
```

```c
void parcoursLargeur(Node* racine) {
    if (racine == NULL) {
        return;
    }

    File* file = creerFile();
    enfiler(file, racine);

    while (!estVide(file)) {
        Node* courant = defiler(file);
        printf("%d ", courant->valeur);

        if (courant->gauche != NULL) {
            enfiler(file, courant->gauche);
        }
        if (courant->droite != NULL) {
            enfiler(file, courant->droite);
        }
    }
}
```

Implémentation

Exercices :

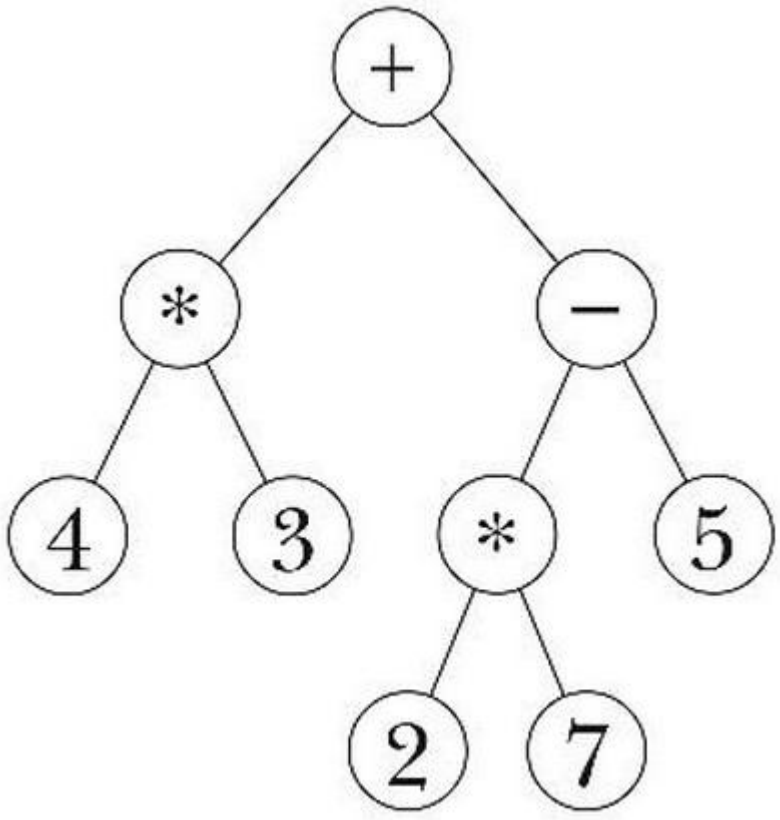
Quel type de parcours doit être utilisé pour reconstruire une expression arithmétique ?

Expression arithmétique:

Ordre préfixe : +x43-x275

Ordre infixe : 4x3+2x7-5

Ordre postfixe : 43x27x5-+



Implémentation

6 – Nombre de nœuds :

```
int tailleArbre(Node* racine) {
    if (racine == NULL) {
        return 0;
    }
    return 1 + tailleArbre(racine->gauche)
             + tailleArbre(racine->droite);
}
```

Implémentation

7 – Hauteur d'un arbre :

```
int hauteurArbre(Node* racine) {
    if (racine == NULL) {
        return -1;
    }
    int hauteurGauche = hauteurArbre(racine->gauche);
    int hauteurDroite = hauteurArbre(racine->droite);
    return 1 + max(hauteurGauche, hauteurDroit);
}
```

Implémentation

8 – Nombre de feuilles:

```
int nombreFeuilles(Node* racine) {
    if (racine == NULL) {
        return 0;
    }
    if (racine->gauche == NULL && racine->droite == NULL) {
        return 1;
    }
    return nombreFeuilles(racine->gauche)
           + nombreFeuilles(racine->droite);
}
```

Implémentation

9 – Max d'un arbre:

```
int maxArbre(Node* racine) {
    if (racine->gauche == NULL && racine->droite == NULL) {
        return racine->valeur;
    }

    if(racine->gauche == NULL) {
        return max(racine->valeur, maxArbre(racine->droite));
    }

    if(racine->droite == NULL) {
        return max(racine->valeur, maxArbre(racine->gauche));
    }

    return max(racine->valeur, maxArbre(racine->gauche), maxArbre(racine->droite));
}
```

HESTIM
ENGINEERING & BUSINESS SCHOOL

Arbre Binaire de Recherche
(ABR)

Arbre binaire de recherche

Les arbres binaires :

- Les arbres binaires complets
- Les arbres binaires de recherche
- Les arbres binaires équivalents
- Les arbres binaires parfaits
- Les arbres AVL
- Les B-Arbres
- …

Arbre binaire de recherche

Définitions :

Un arbre binaire de recherche (ABR) est un arbre binaire qui possède la propriété fondamentale suivante :

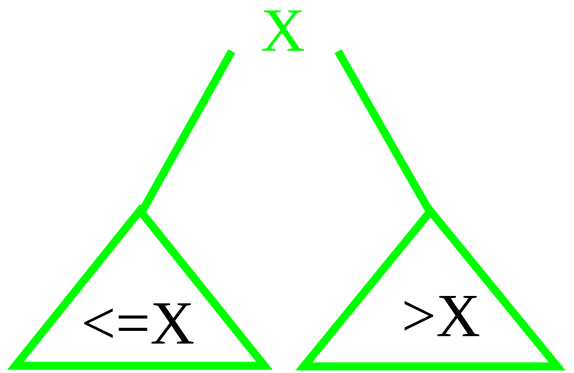
- Tous les nœuds du sous-arbre gauche d'un nœud de l'arbre ont une valeur inférieure ou égale à ce nœud.
- Tous les nœuds du sous-arbre droit d'un nœud de l'arbre ont une valeur supérieure à ce nœud.

Arbre binaire de recherche

Récursivement un *arbre binaire de recherche* est un arbre binaire tel que :

1 - Pour tout nœud x , les nœuds de son sous arbre-gauche s'ils en existent ont des valeurs inférieures ou égales à celle de x, et les nœuds de son sous arbre-droit des valeurs strictement supérieures.

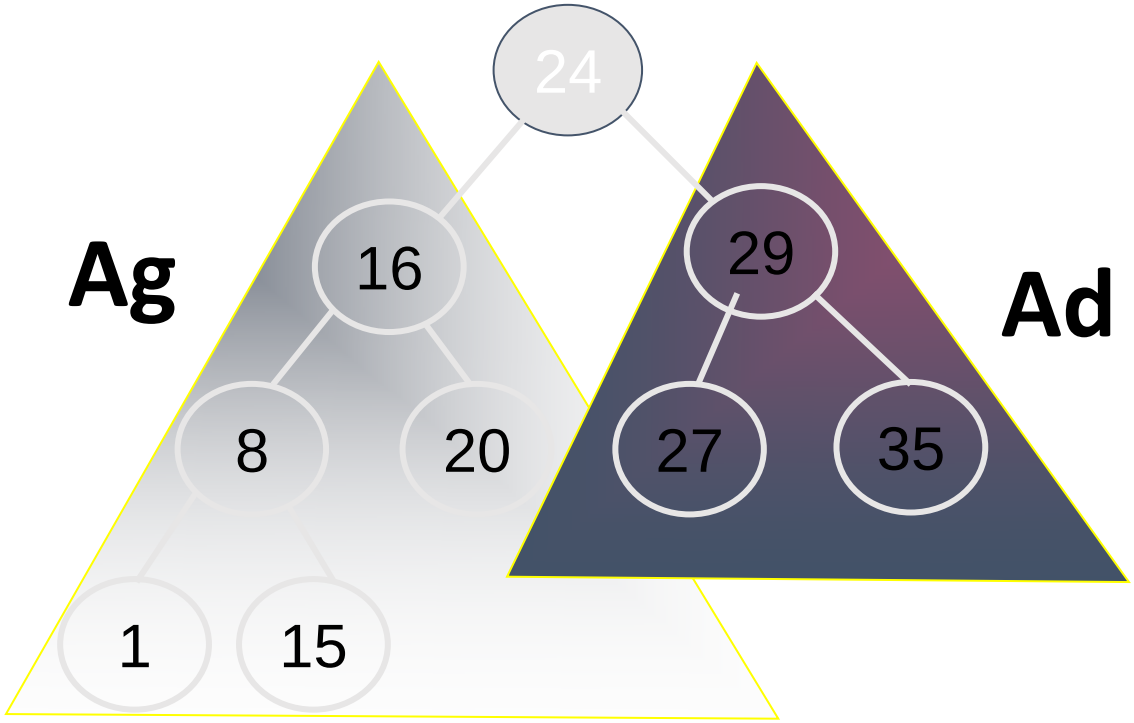
Ce que l'on traduit par $g(A) \leq racine(A) < d(A)$.



Arbre binaire de recherche

Récursivement un *arbre binaire de recherche* est un arbre binaire tel que :

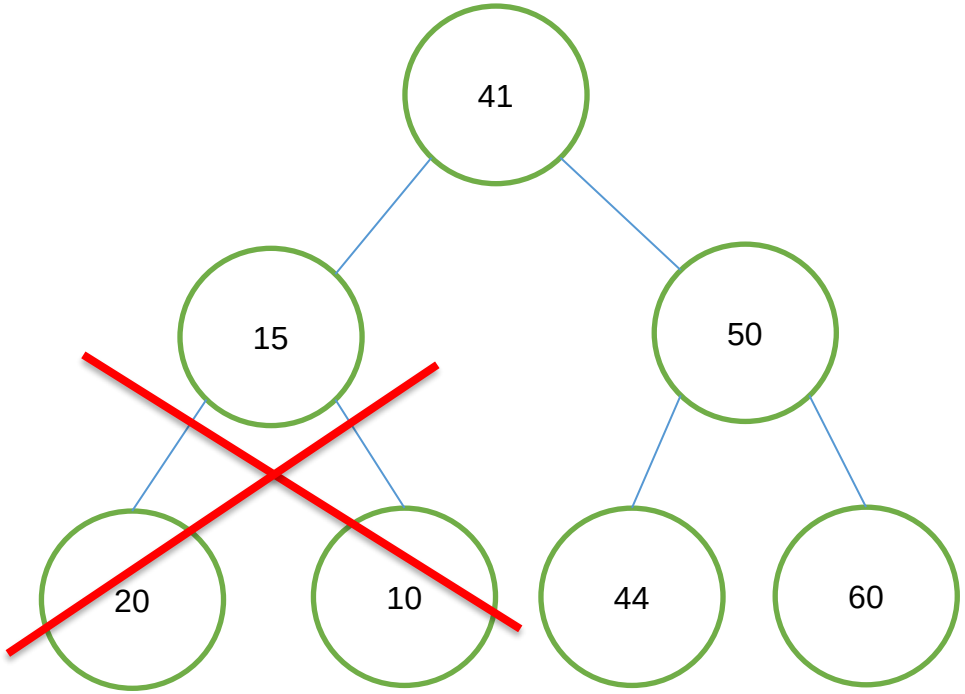
2 - Tout sous-arbre d'un ABR est un ABR : .



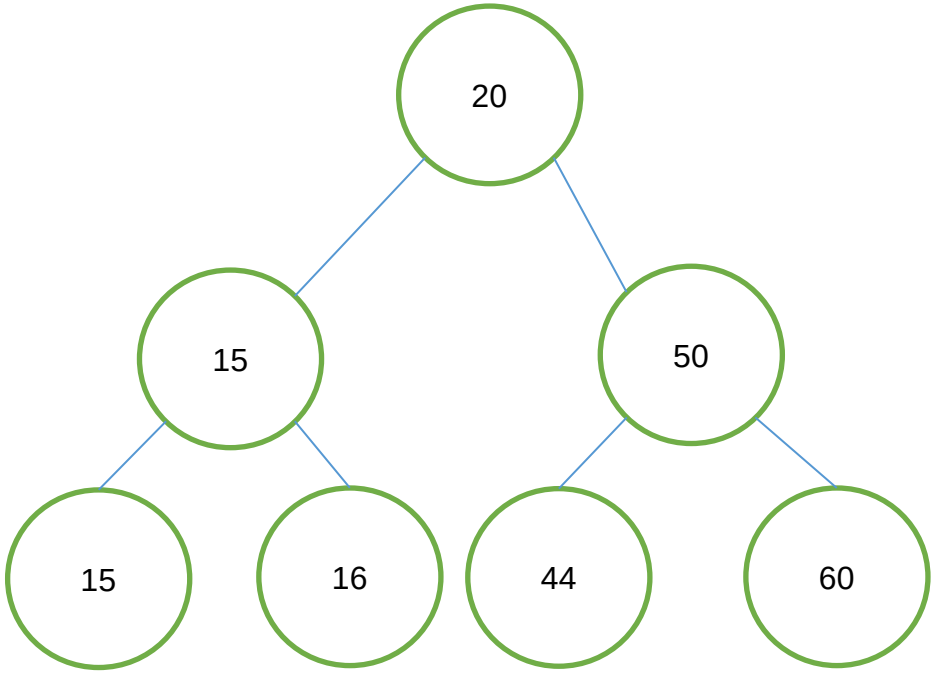
Exemple d'arbre binaire de recherche

Arbre binaire de recherche

Exemples :



Ce n’est pas un arbre binaire de recherche



C’est un arbre binaire de recherche

Arbre binaire de recherche

Applications :

Un arbre binaire de recherche ou ABR est une structure de données utilisée pour représenter un ensemble de valeurs ou objets qui subit à une relation d'ordre.

Exemples :

- Un ensemble d'employé trié par leur hiérarchie dans l'organisation
- Un ensemble de personnes trié par leurs âges

Arbre binaire de recherche

Question : Pourquoi ne pas utiliser un tableau ou liste triée au lieu d'un ABR ?

Comparaison :

Opération	Liste	ABR
Recherche	O(log(n))	O(log(h))
Insertion	O(n)	O(h)
Suppression	O(n)	O(h)
Min	O(n)	O(h)
Max	O(n)	O(h)

NB : Ces complexités suppose que les arbres sont équilibrés

Arbre binaire de recherche

Applications :

Ecrire une fonction qui vérifie si un arbre binaire est un arbre de binaire de recherche:

Vous pouvez utiliser les fonctions maxArbre(A) et minArbre(A) qui retourne respectivement le max et le min d'un arbre binaire.

```
int estAbr(Node* racine) {
    if (racine == NULL) {
        return 1;
    }

    if (racine->gauche != NULL && racine->gauche->valeur > racine->valeur) {
        return 0;
    }

    if (racine->droite != NULL && racine->droite->valeur < racine->valeur) {
        return 0;
    }

    return estAbr(racine->gauche) && estAbr(racine->droite);
}
```

Arbre binaire de recherche

Ecrire une fonction qui retourne la valeur maximale d'un arbre binaire de recherche:

```
int maxAbrR(Node* racine) {
    if(racine->droite == NULL) {
        return racine->valeur;
    }
    return maxAbrR(racine->droite);
}
```

Version récursive

```
int maxAbr(Node* racine) {
    Node* courant = racine;
    while (courant->droite != NULL) {
        courant = courant->droite;
    }
    return courant->valeur;
}
```

Version itérative

Arbre binaire de recherche

Ecrire une fonction qui retourne la valeur minimale d'un arbre binaire de recherche:

```
int minAbrR(Node* racine) {
    if(racine->gauche == NULL) {
        return racine->valeur;
    }
    return minAbrR(racine-> gauche);
}
```

Version récursive

```
int minAbr(Node* racine) {
    Node* courant = racine;
    while (courant->gauche != NULL) {
        courant = courant->gauche;
    }
    return courant->valeur;
}
```

Version itérative

Arbre binaire de recherche

Recherche par dichotomie:

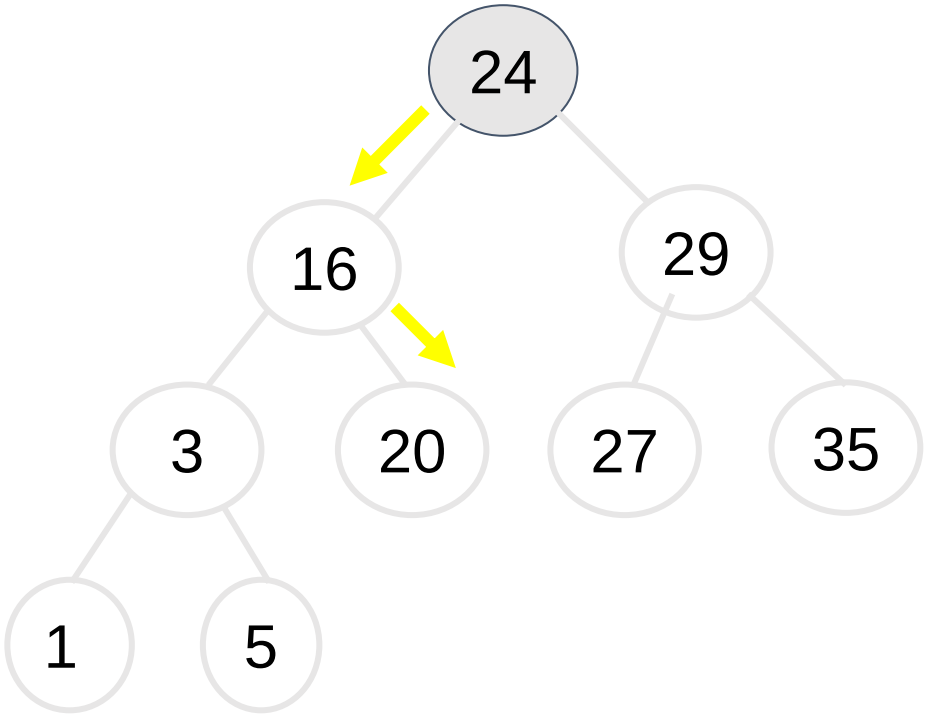
L'algorithme :

- si le sous-arbre sélectionné est vide, l'élément est absent la fonction retourne False
- On compare l'élément à la valeur de la racine , si égalité la fonction retourne True
- si la valeur est plus petite, on recommence **récursivement** dans le sous-arbre gauche ; et réciproquement si la valeur est plus grande dans le sous-arbre droit.

Arbre binaire de recherche

Recherche par dichotomie:

Exemple : Soit à rechercher 20 dans l'arbre suivant



20 est plus petit que 24

20 est plus grand que 16

20 est trouvé

Arbre binaire de recherche

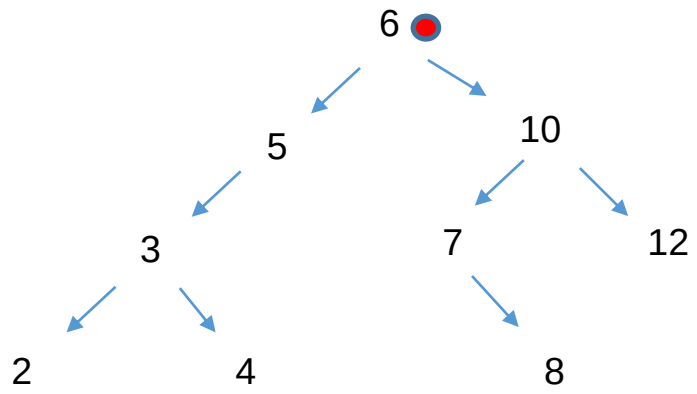
Recherche par dichotomie:

```
int rechercheDichotomique(Node* racine, int valeur) {
    if (racine == NULL) {
        return 0;
    }
    if (valeur == racine->valeur) {
        return 1;
    }
    if (valeur < racine->valeur) {
        return rechercheDichotomique(racine->gauche, valeur);
    }
    return rechercheDichotomique(racine->droite, valeur);
}
```

Arbre binaire de recherche

Insertion dans un ABR :

On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
si elle est inférieure, le nouveau nœud sera à gauche ;
sinon il sera à droite.

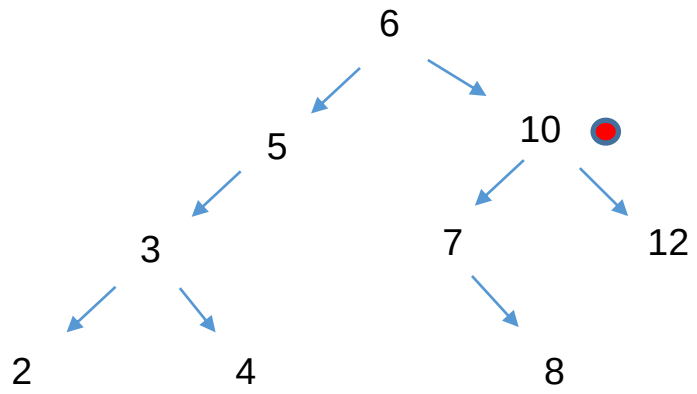


Insérer 9

Arbre binaire de recherche

Insertion dans un ABR :

On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
si elle est inférieure, le nouveau nœud sera à gauche ;
sinon il sera à droite.

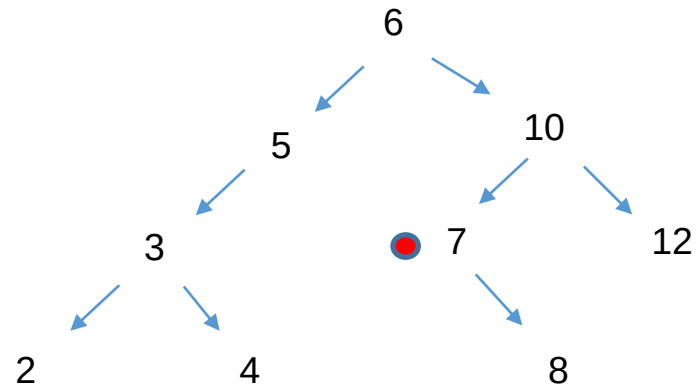


Insérer 9

Arbre binaire de recherche

Insertion dans un ABR :

On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
si elle est inférieure, le nouveau nœud sera à gauche ;
sinon il sera à droite.

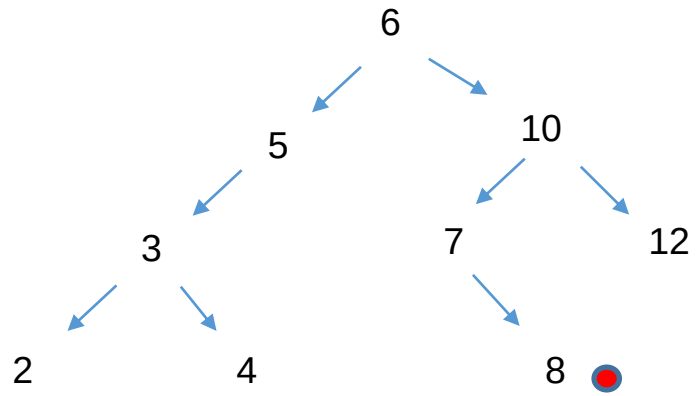


Insérer 9

Arbre binaire de recherche

Insertion dans un ABR :

On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
si elle est inférieure, le nouveau nœud sera à gauche ;
sinon il sera à droite.

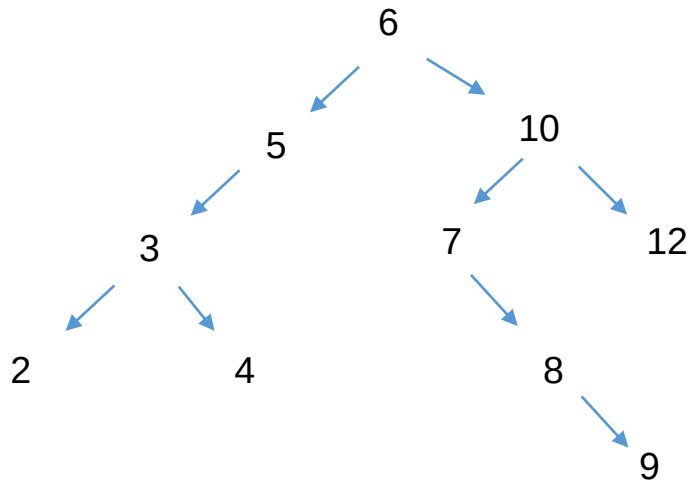


Insérer 9

Arbre binaire de recherche

Insertion dans un ABR :

On cherche la clé du nœud à insérer ;
Lorsqu'on arrive à une feuille, on ajoute le nœud comme fils de la feuille en comparant sa clé à celle de la feuille :
si elle est inférieure, le nouveau nœud sera à gauche ;
sinon il sera à droite.



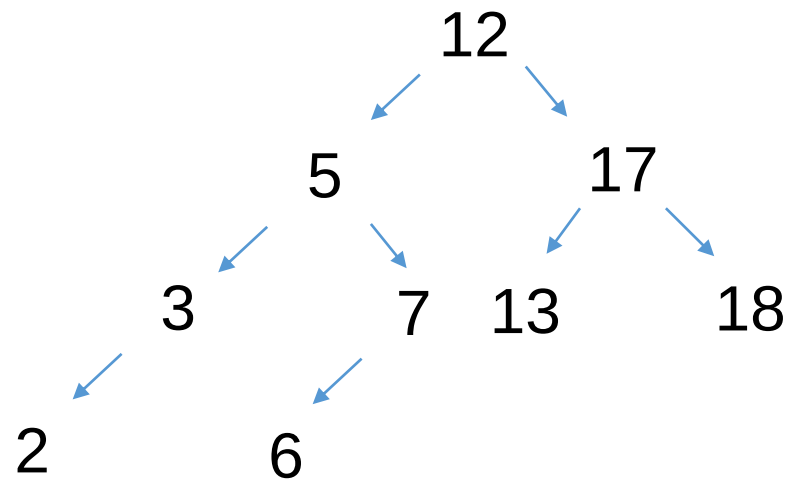
Insérer 9

Arbre binaire de recherche

Insertion dans un ABR :

Exercice :

Quel arbre obtient-on en insérant dans l'ordre , 12, 5, 3, 17, 7, 13, 18, 2 et 6 dans un arbre vide ?



Arbre binaire de recherche

Insertion dans un ABR :

Exercice :
Ecrire une fonction récursive **inserer(abr , v)** qui permet d'insérer correctement la valeur v dans l'arbre binaire de recherche abr. La fonction retourne un arbre avec la nouvelle valeur à insérer.

```
Node* insererAbr(Node* racine, int valeur) {
    if (racine == NULL) {
        return creerNoeud(valeur);
    }

    if (valeur < racine->valeur) {
        racine->gauche = insererAbr(racine->gauche, valeur);
    } else if (valeur > racine->valeur) {
        racine->droite = insererAbr(racine->droite, valeur);
    }

    return racine;
}
```

Arbre binaire de recherche

Supprimer un élément d'un ABR :

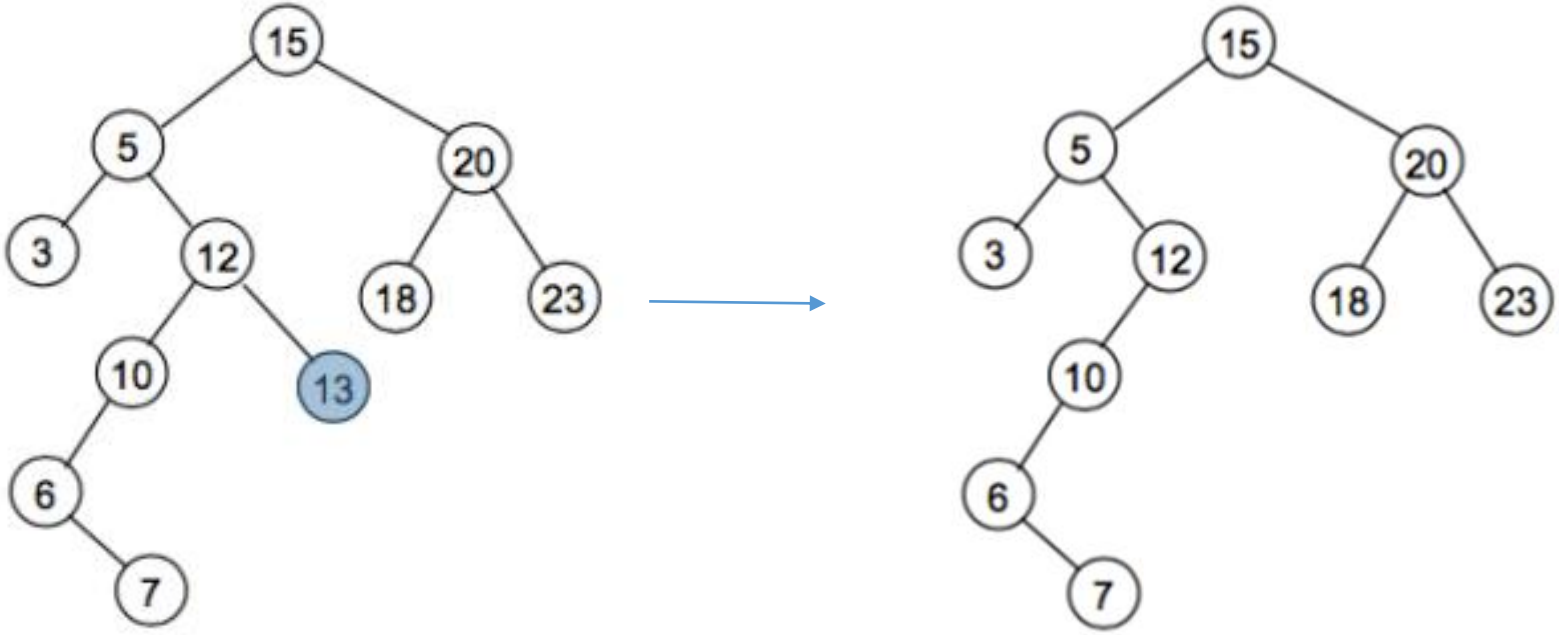
Pour supprimer un nœud d'un arbre binaire de recherche, on distingue trois cas à traiter :

- L'élément à supprimer n'a pas de fils : il est terminal et il suffit de le supprimer
- L'élément a un fils unique : on supprime le nœud et on relie son fils à son père
- L'élément à supprimer a deux fils : on le remplace par son successeur qui est toujours le minimum de ses descendants droits OU pas son prédécesseur qui est toujours le maximum de ses descendant gauche.

Arbre binaire de recherche

Supprimer un élément d'un ABR :

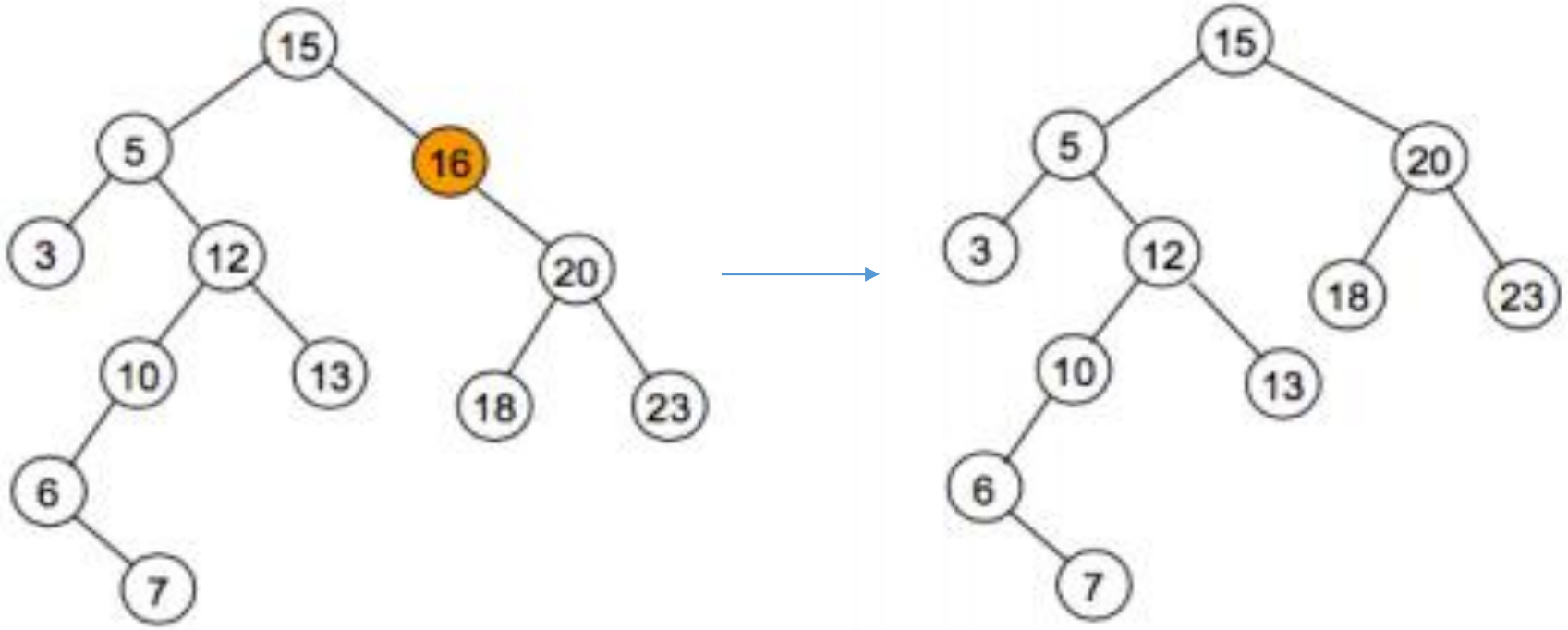
- Cas N° 1 : Supprimer une feuille



Arbre binaire de recherche

Supprimer un élément d'un ABR :

- Cas N° 2 : Supprimer un nœud qui a un seul fils

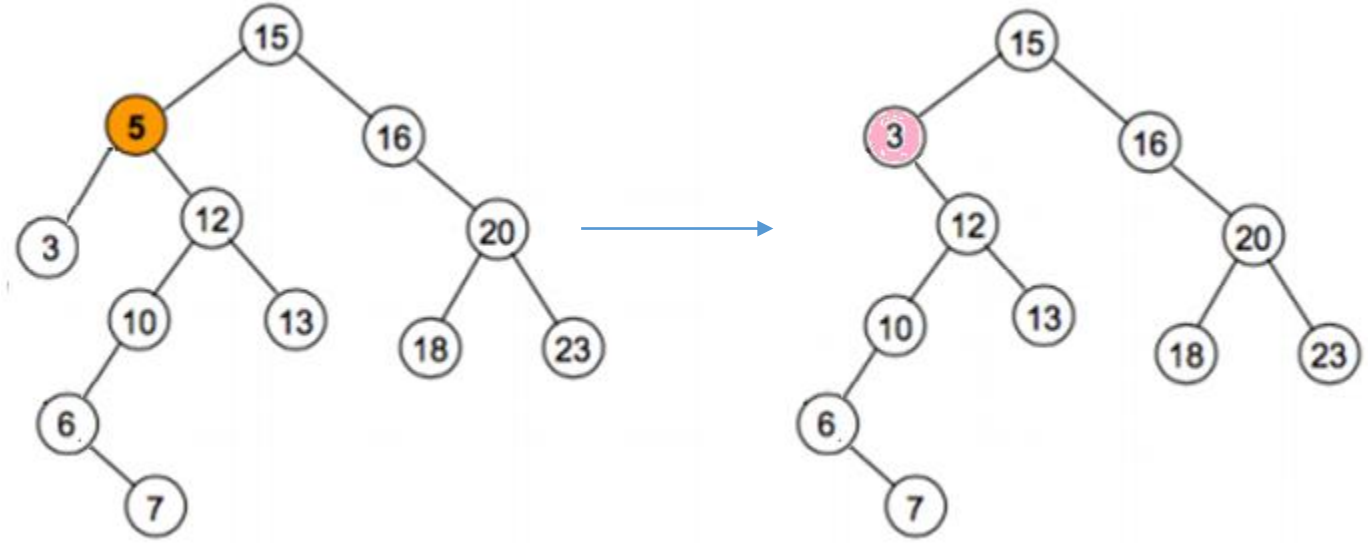


Arbre binaire de recherche

Supprimer un élément d'un ABR :

- Cas N° 3 : Supprimer un nœud qui a un deux fils

Soit : remplacer le nœud à supprimer par la feuille du sous-arbre gauche contenant la valeur maximal

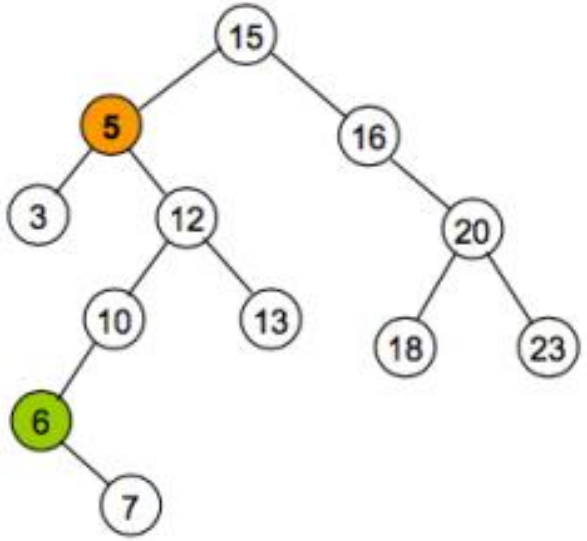


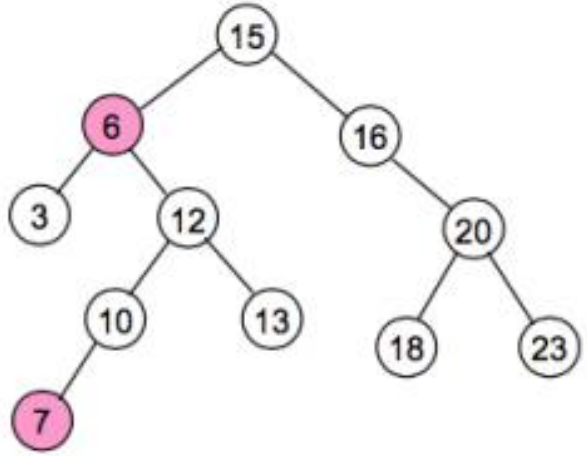
Arbre binaire de recherche

Supprimer un élément d'un ABR :

- Cas N° 3 : Supprimer un nœud qui a un deux fils

Ou : ou par la feuille du sous-arbre droit contenant la valeur minimale





Arbre binaire de recherche

Equilibre :

Un arbre binaire de hauteur h peut contenir jusqu'à $2^{h+1} - 1$ nœuds :

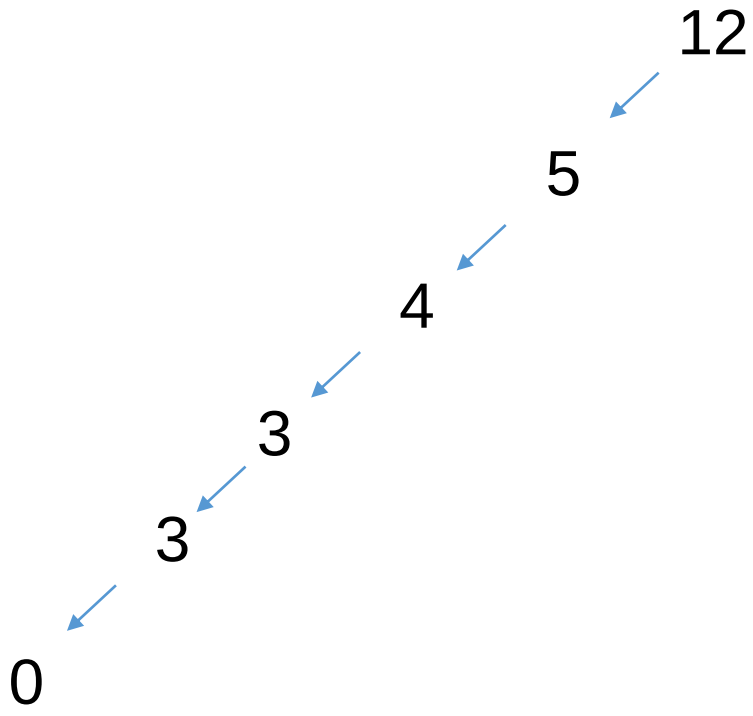
$$1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$$

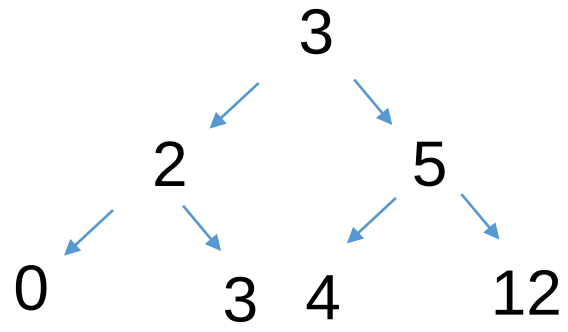
Définition :
Un arbre binaire est équilibré si sa hauteur h est d'un ordre de grandeur logarithmique par rapport au nombre de sommet.

Arbre binaire de recherche

Equilibre :

Exemples : pour la liste de valeurs suivante : 12, 5, 4 , 3, 3 , 0





HESTIM
ENGINEERING & BUSINESS SCHOOL

Arbre Binaire Parfait

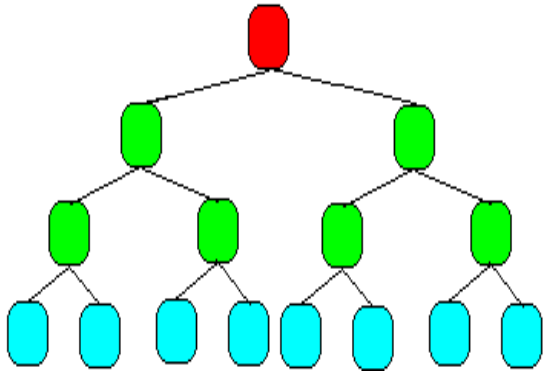
Arbre binaire parfait

Définition :

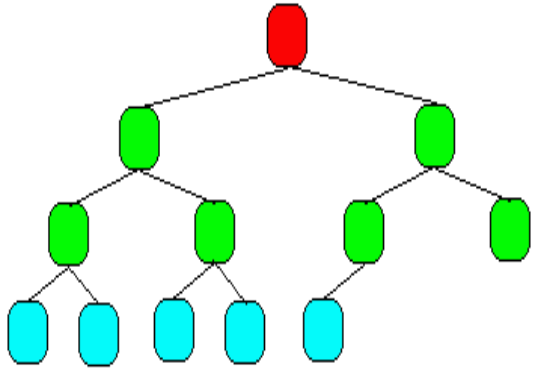
Un arbre binaire est parfait si :

1. Tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux = feuilles)

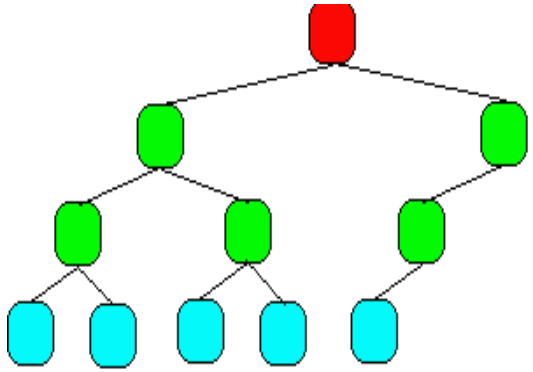
2. Les feuilles du dernier niveau doivent être regroupées à partir de la gauche de l'arbre.



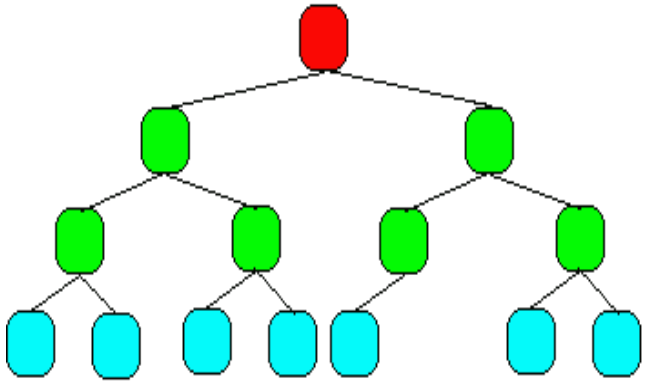
ARBRE BINAIRE PARFAIT



ARBRE BINAIRE PARFAIT



ARBRE BINAIRE NON PARFAIT



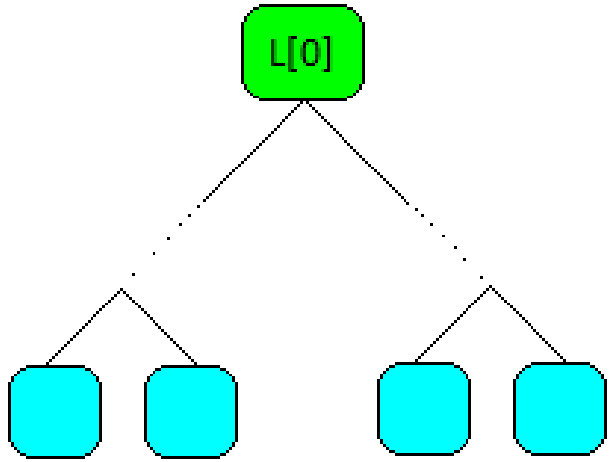
ARBRE BINAIRE NON PARFAIT

Arbre binaire parfait

Implémentation :

Un arbre binaire parfait peut être implémenter avec un simple tableau T , avec les règles suivantes :

Règle 1 : T[0] est la racine de l'arbre



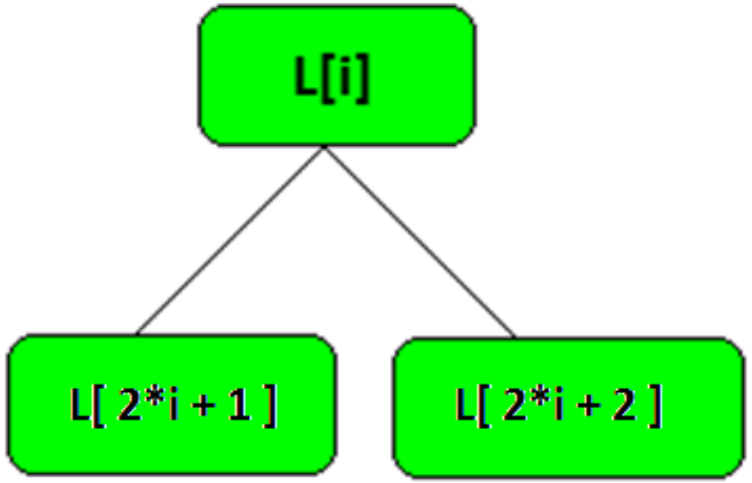
Arbre binaire parfait

Implémentation :

Un arbre binaire parfait peut être implémenter avec une simple liste L , avec les règles suivantes :

Règle 1 : L[0] est la racine de l'arbre

Règle 2: T[2*i+1] et T[2*i+2] sont les feuilles de T[i] :



Arbre binaire parfait

Implémentation :

Un arbre binaire parfait peut être implémenter avec une simple liste L , avec les règles suivantes :

Règle 1 : T[0] est la racine de l'arbre

Règle 2: T[2*i+1] et T[2*i+2] sont les feuilles de T[i] :

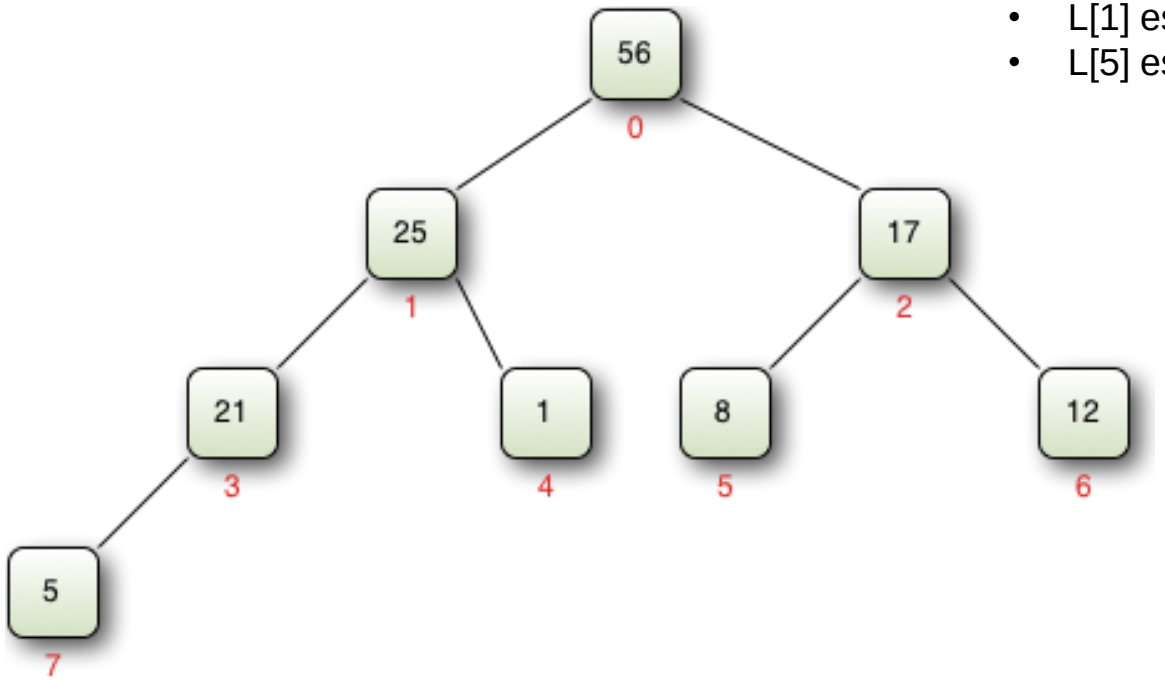
Règle 3 : Si i est supérieur ou égale à len(T)//2, T[i] est une feuille.

Règle 4: Le père de T[i] est T[(i-1)//2]

Arbre binaire parfait

Exemple :

56	25	17	21	1	8	12	5
0	1	2	3	4	5	6	7



- L[1] est le père de L[3] et L[4]
- L[5] est une feuille

Arbre binaire parfait

Applications:

L'idée qui sous-tend d'un arbre parfait consiste à voir le tableau comme un arbre binaire parfait. Il existe plusieurs applications des arbres binaires parfaits :

1. Tas binaire : File de priorité (Tas Min, Tas Max)
2. Tri Maximier

HESTIM
ENGINEERING & BUSINESS SCHOOL

Tas binaire (Max / Min)

Le tas binaire

Problème:

On veut une structure de données qui permet de stocker un ensemble d'éléments qui se distingue par une propriété de comparaisons entre eux, et de pouvoir trouver efficacement à chaque fois l'élément le plus prioritaire.

On veut optimiser :

- Extraire l'élément le plus prioritaire
- Tester si il existe des éléments dans la file de priorité
- Insérer un élément dans la file de priorité
- Modifier la priorité d'un élément

Le tas binaire

Exemple :

Gérer un ensemble d'objet représentant des événements caractérisés par des dates {e1, e2, e3, e4, …}.

e1: conférence le 30/12/2022

e2: anniversaire le 30/09/2022

e3: mariage le 30/11/2022

e4: soutenance le 30/10/2022

Une solution possible : une liste triée par ordre décroissant des dates

e2	**e4**	**e3**	**e1**

Le tas binaire

Complexité des primitives d'une file de priorité implémenté avec une liste :

— supprimer l'élément de priorité maximale en O(n);
— accéder à l'élément de priorité maximale en O(1);
— insérer un nouvel élément associé à une priorité en O(n);
— modifier la priorité d'un élément (si vous pouvez le retrouver dans le tableau) en O(n).

Le tas binaire

Un tas binaire est une structure de données permettant de modéliser une file à priorités. Elle permet de réaliser les opérations suivantes :

— supprimer l'élément de priorité maximale en O(h) (ou O(log(n));

— accéder à l'élément de priorité maximale en O(1);

— insérer un nouvel élément associé à une priorité en O(h) (ou O(log(n));

— modifier la priorité d'un élément (si vous pouvez le retrouver dans le tableau) en O(h) (ou O(log(n)).

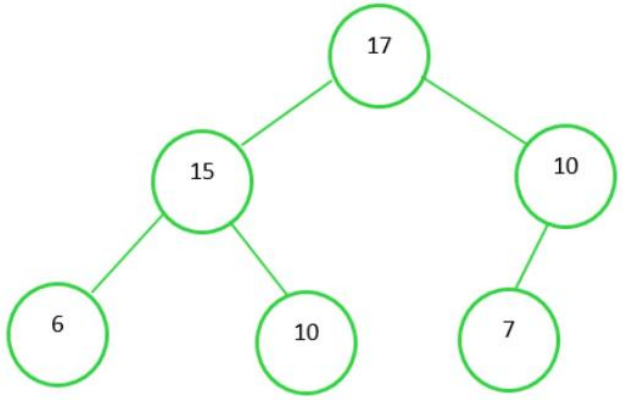
Le tas binaire

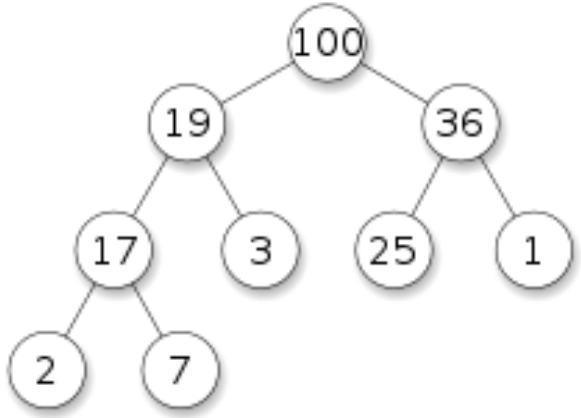
Un arbre binaire est appelé Tas binaire Max (Tas binaire Min) , si :

1. C’est un arbre binaire Parfait
2. La clé présente au nœud racine doit être supérieure ou égale (Inférieur ou égale) aux clés présentes à tous ses enfants. La même propriété doit être récursivement vraie pour tous les sous-arbres de cet arbre binaire

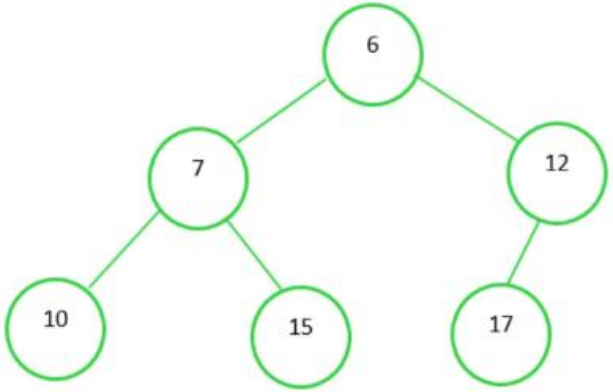
Le tas binaire

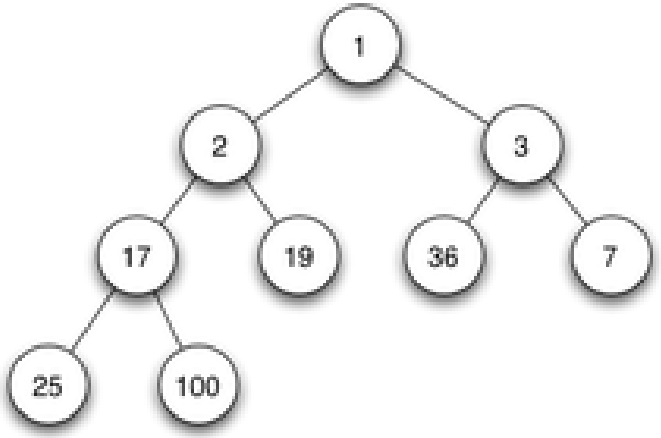
Exemples Tas Max :





Exemples Tas Min :

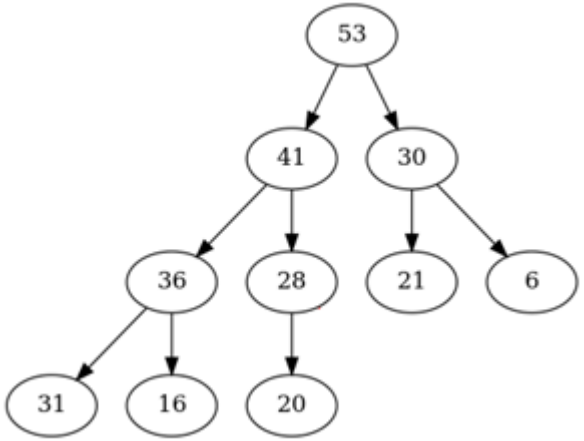


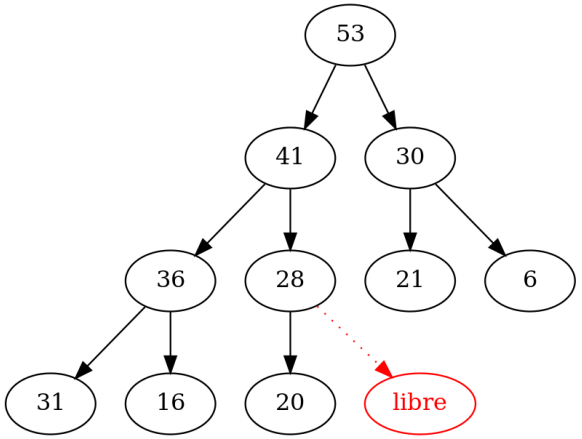


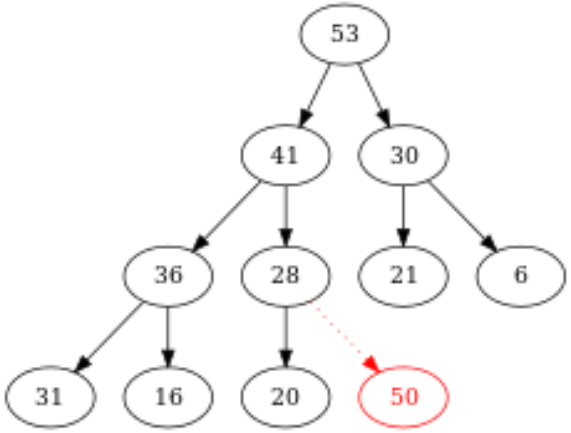
Le tas binaire

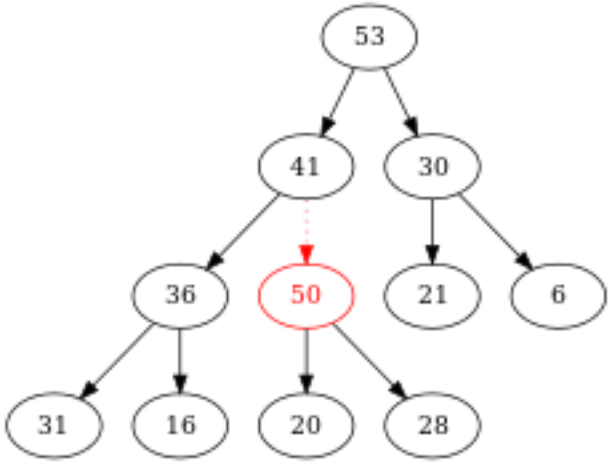
Insérer un élément :

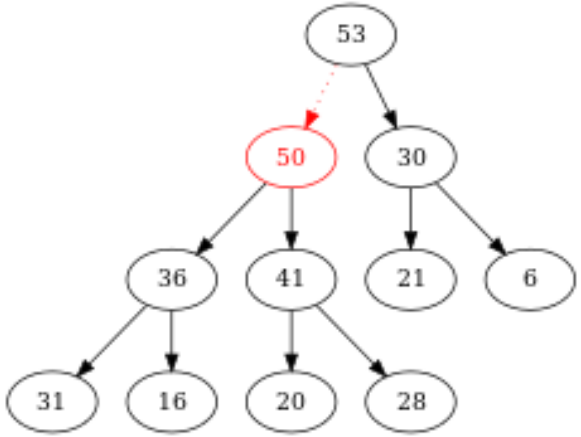
Exemple : Insérer la valeur 50 (Percolation vers le haut)

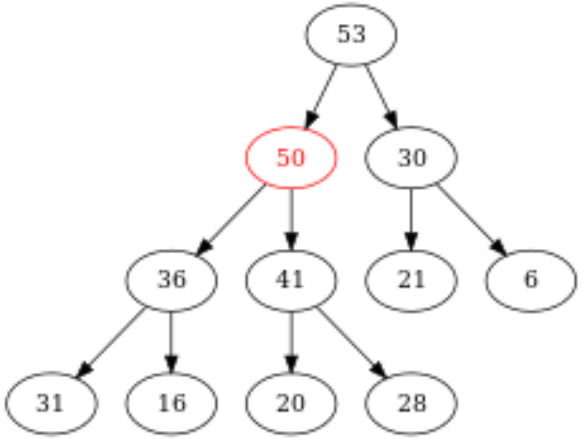












Le tas binaire

Insérer un élément :

```c
#define MAX 100

typedef struct {
    int taille;
    int T[MAX];
} Tas;
```

```c
void inserer(Tas *H, int valeur) {
    if (H->taille == MAX) {
        printf("Erreur : tas plein !\n");
        return;
    }
    // 1. insérer la valeur à la fin
    int i = H->taille;
    H->T[i] = valeur;
    H->taille++;

    // 2. faire remonter l'élément (up-heap bubbling)
    while (i > 0) {
        int parent = (i - 1) / 2;

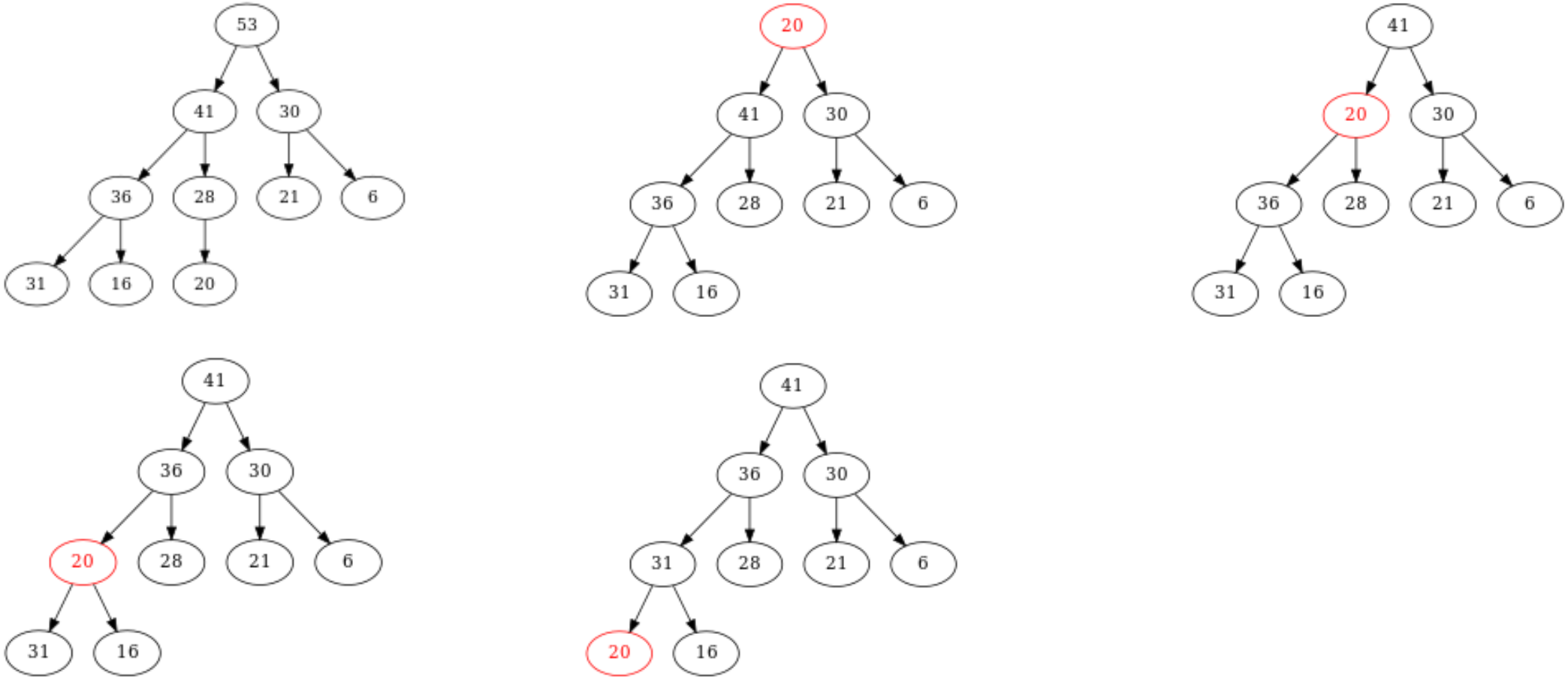
        if (H->T[parent] >= H->T[i])
            break;  // tas valide

        // échanger avec le parent
        int tmp = H->T[parent];
        H->T[parent] = H->T[i];
        H->T[i] = tmp;
        i = parent;  // continuer la remontée
    }
}
```

Le tas binaire

Supprimer l'élément prioritaire :

Exemple : Retirer l'élément prioritaire (la racine 53) (Percolation vers le bas) :



Le tas binaire

Supprimer un élément :

```
void entasser(Tas *T, int i) {
    // Indices des enfants gauche et droite
    int gauche = 2 * i + 1;
    int droite = 2 * i + 2;
    // On suppose au début que le plus grand élément est à la position i
    int plusGrand = i;
    // Vérifier si l'enfant gauche existe et est plus grand que l'élément actuel
    if (gauche < T->taille && T->T[gauche] > T->T[plusGrand]) {
        plusGrand = gauche;
    }
    // Vérifier si l'enfant droit existe et est plus grand que le plus grand
    if (droite < T->taille && T->T[droite] > T->T[plusGrand]) {
        plusGrand = droite;
    }
    // Si un enfant est plus grand que le parent, on échange et on continue récursivement
    if (plusGrand != i) {
        // échanger les valeurs
        echanger(&T->T[i], &T->T[plusGrand]);
        entasser(T, plusGrand);
    }
}
```

Le tas binaire

Supprimer un élément :

```c
int supprimer(Tas *H) {
    if (H->taille == 0) {
        printf("Erreur : tas vide !\n");
        return -1; // Indiquer une erreur
    }

    int valeurSupprimee = H->T[0];

    // Remplacer la racine par le dernier élément
    H->T[0] = H->T[H->taille - 1];
    H->taille--;

    // Faire descendre l'élément à la racine (down-heap bubbling)
    entasser(H, 0);

    return valeurSupprimee;
}
```

Le tas binaire

Construire un Tas :

On souhaite construire un tas binaire à partir d'un ensemble d'éléments. De manière naïve on part du tas vide et on ajoute les éléments un par un avec l'algorithme ajouter, ce qui donne une complexité en O(n log n). Ce n'est pas optimal, il existe une manière de construire notre tas binaire en O(n):

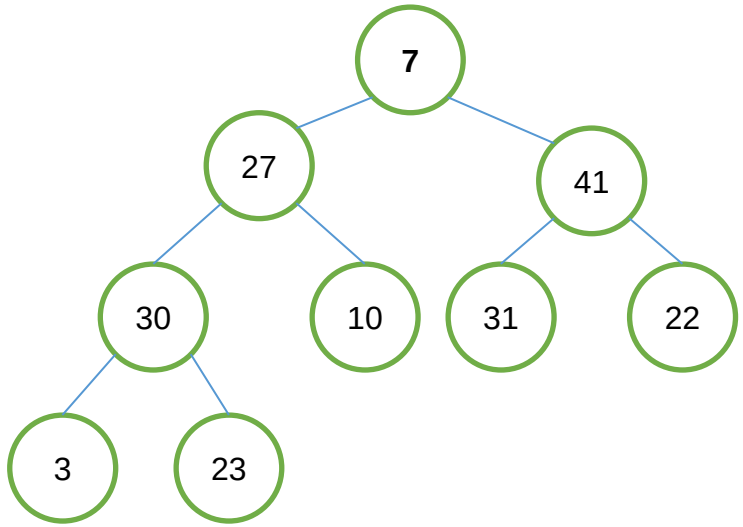
1. On construit un arbre binaire parfait avec tous les éléments sans se soucier de la contrainte d'ordre. Cela prend donc un temps linéaire en O(n).

2. On parcourt les sommets niveau par niveau en partant de l'avant dernier niveau (profondeur h-1) et dans chaque niveau on parcourt les sommets de la droite vers la gauche. Lors de notre parcours on effectue un percolate-down à partir de chaque sommet.

Le tas binaire

Construire un Tas Max

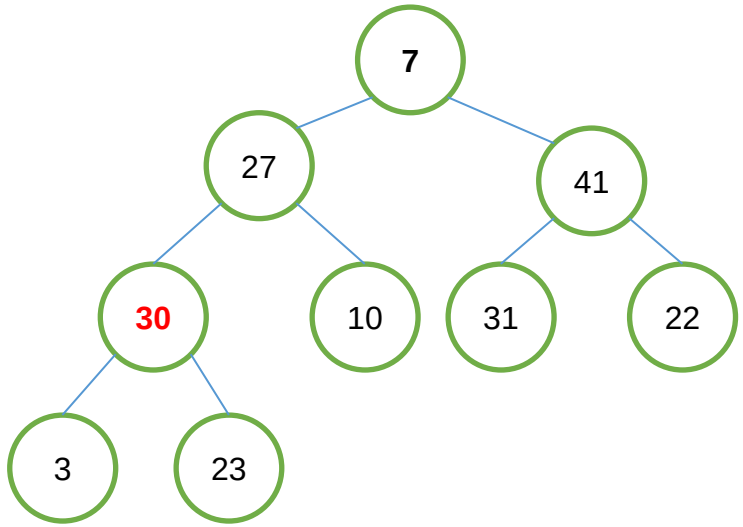
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

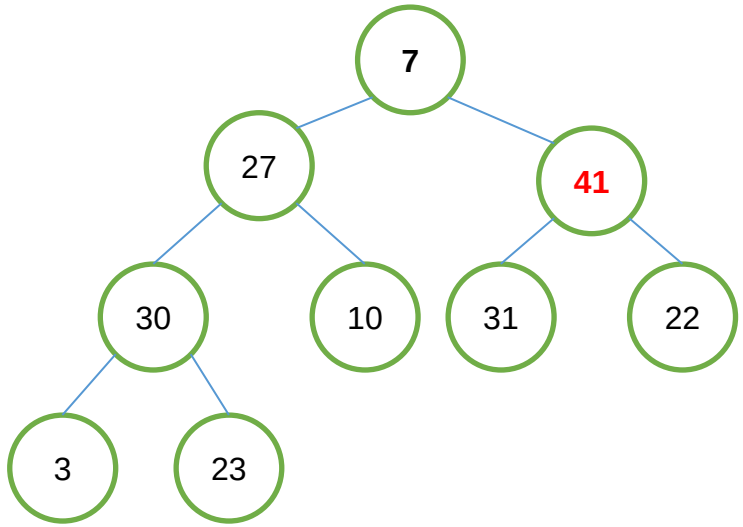
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

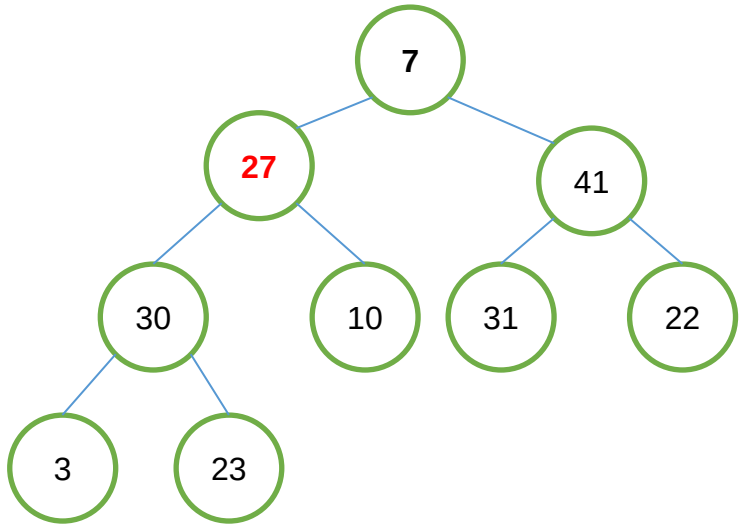
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

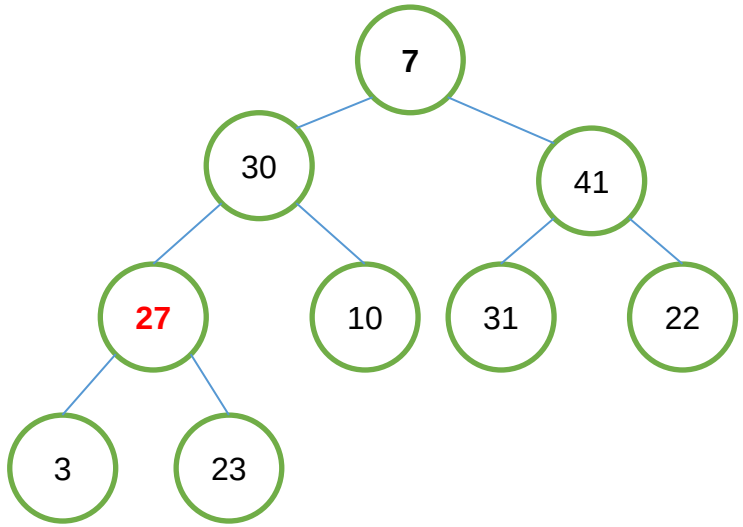
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

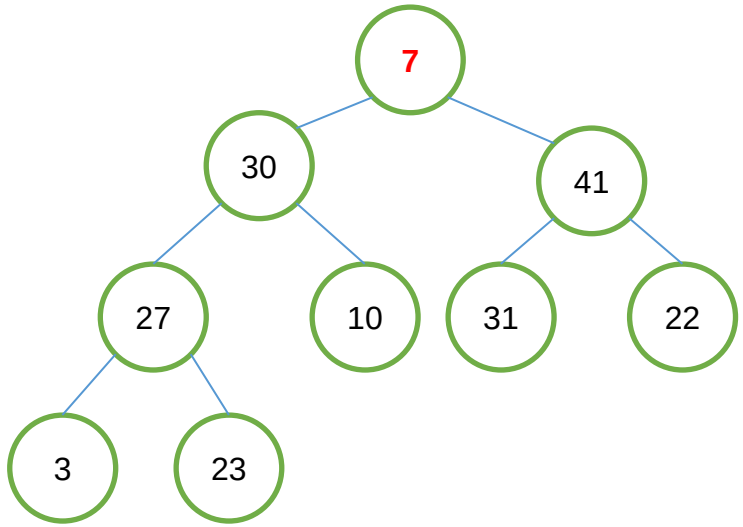
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

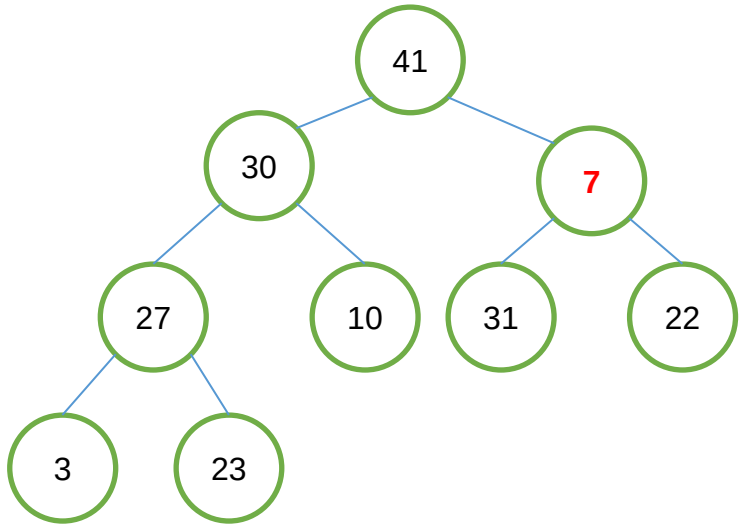
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

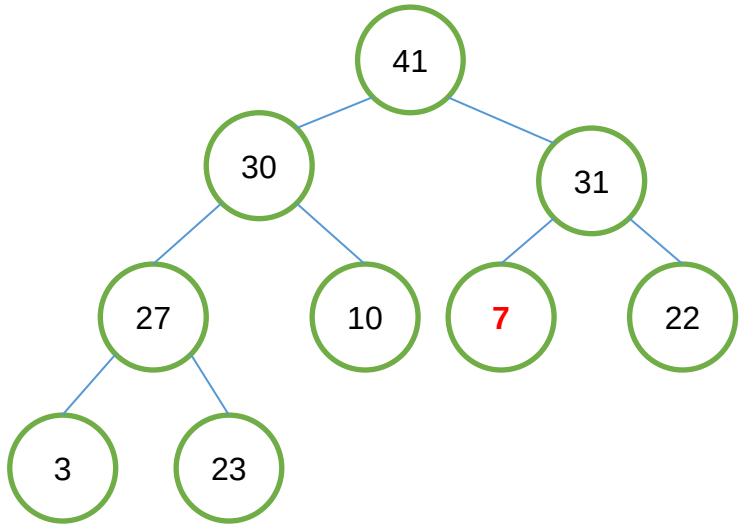
7	27	41	30	10	31	22	3	23



Le tas binaire

Construire un Tas Max

7	27	41	30	10	31	22	3	23



Le tas binaire

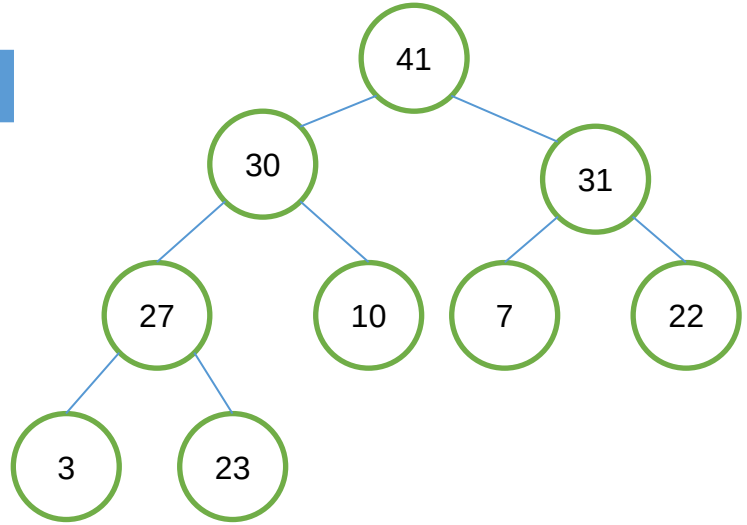
Construire un Tas Max

Liste initiale :

7	27	41	30	10	31	22	3	23

Nouvelle liste (Le tas) :

41	30	31	27	10	7	22	3	23



Le tas binaire

Construire un Tas Max

Exercice :

Donner manuellement le tas du tableau suivant L = { 3, 0, 12, 4, 3, 1, 9}

Le tas est : L = { 12 , 4, 9 , 0 , 3 , 1 , 3 }

Les graphes

HESTIM ENGINEERING & BUSINESS SCHOOL

Plan

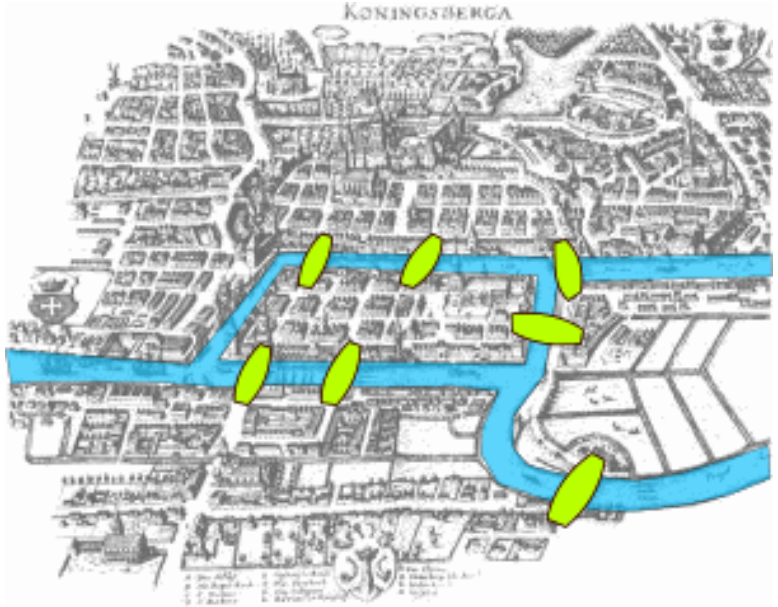
- Définitions et terminologies
- Implémentation des graphes
- Algorithmes de parcours d'un graphe
- Applications
  - Plus courts chemin(Algorithme Dijkstra, Bellman-Ford, Floyd-Warsha
  - Arbre couvrant (Kruskal, Prim)
  - Détection de cycles et connexité
  - Coloration de graphes
  - Algorithme A*

Objectifs

- Comprendre les notions de base des graphes
- Savoir modéliser certains problèmes afin de les résoudre à l'aide des techniques et des algorithmes simples.
- Maîtriser les algorithmes de recherche des plus courts chemins (Dijkstra et A*), des composantes connexes (Tarjan), des arbres couvrants minimaux (Kruskal et Prim), de coloration des sommets (Welch-Powell et Dsatur), etc.

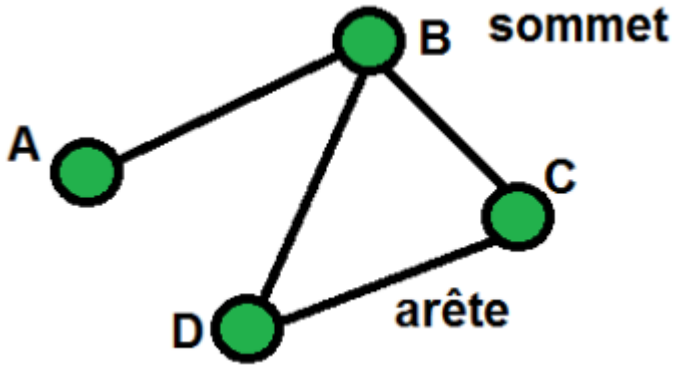
Origine

La théorie des graphes est née en 1736 quand Euler a posé et résolu le problème des sept ponts de la ville russe de Konigsberg (aujourd'hui Kaliningrad) :
Peut-on, partant d'un point quelconque (rive ou ile) parcourir les 7 ponts une fois et une seule et revenir à son point de départ ?



Définitions et terminologies

Un graphe est la donnée d'un certain nombre de points du plan, appelés ***sommets***, certains étant reliés par des segments de droites ou de courbes appelés ***arêtes***, la disposition des sommets et la forme choisie pour les arêtes n'intervenant pas.



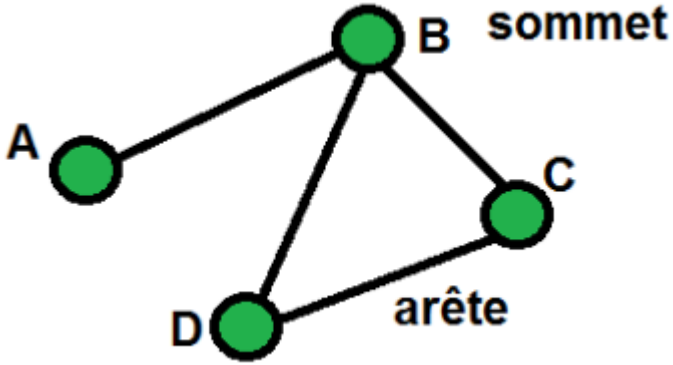
Définitions et terminologies

Un graphe non orienté G = (S, A) est déterminé par deux ensembles :

- S un ensemble fini des sommets,
- A un ensemble de couples non ordonnés de sommets $(a_i, a_j) \in S^2$.

Exemple :

G = ({A,B,C,D} , { (A,B),(B,C),(C,D),(B,D) })



Définitions et terminologies

Exemples des situations modélisées par un graphe :

- Carte géographique : Recherche du chemin le plus court entre deux villes.
- Les lignes aériennes
- Le web : chaque page est un sommet du graphe, chaque lien hypertexte est une arête entre deux sommets.
- Le routage des réseaux informatiques
- Un réseau social : les sommets sont les personnes, deux personnes sont adjacentes dans ce graphe lorsqu'elles sont "amies". Si la notion d'amitié n'est pas réciproque, le graphe est orienté.

Exemple d'application 1

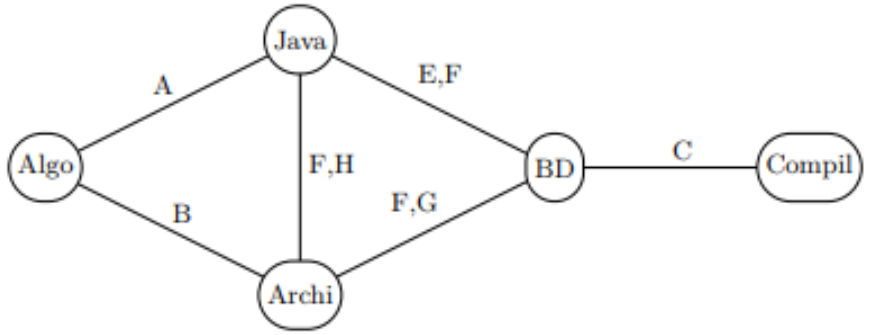
Des étudiants A, B, C, D, E et F doivent passer des examens dans différentes disciplines, chaque examen occupant une demi-journée :

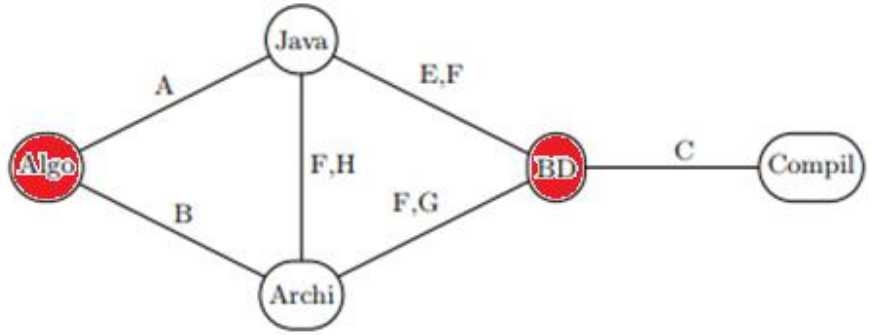
– Algorithmique : étudiants A et B.
– Compilation : étudiants C et D.
– Bases de données : étudiants C, E, F et G.
– Java : étudiants A, E, F et H.
– Architecture : étudiants B, F, G et H.

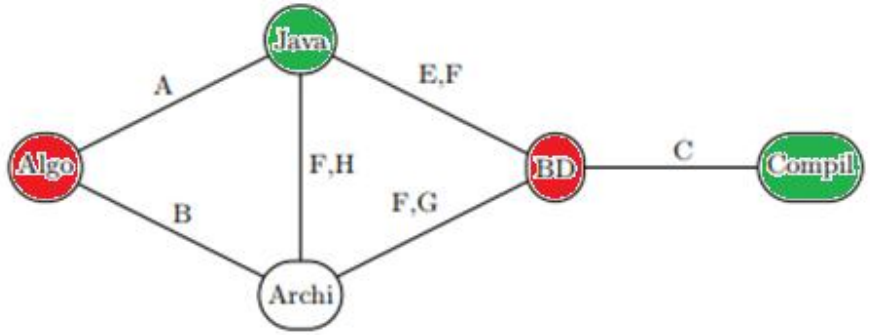
On cherche à organiser la session d'examen la plus courte possible.

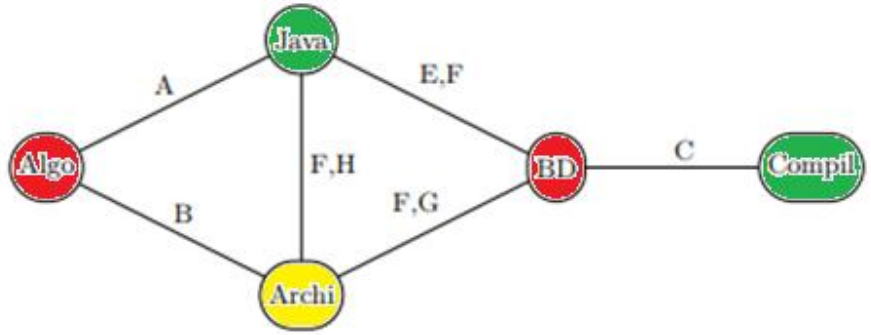
Définitions et terminologies

Exemple d'application 1









Exemple d'application 1

Session 1:
Algo : A,B
BD : C, E, F, G

Session 2:
Java : A, E, F, H
Comp : C

Session 3:
Arch : B, F, G, H

Exemple d'application 2

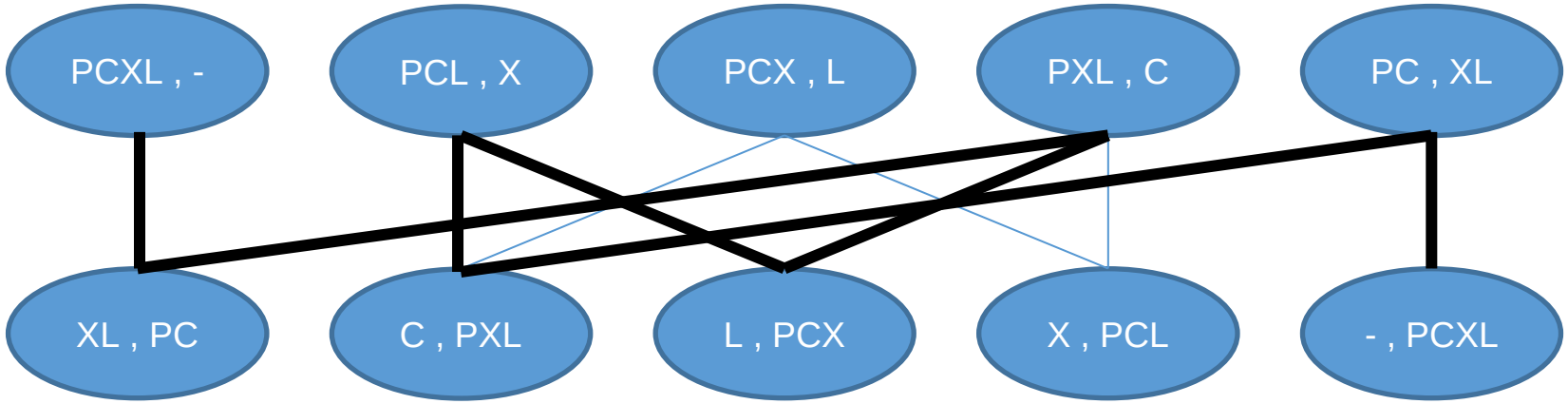
Une chèvre, un chou et un loup se trouvent sur la rive d'un fleuve ; un passeur souhaite les transporter sur l'autre rive mais, sa barque étant trop petite, il ne peut transporter qu'un seul d'entre eux à la fois.

Comment doit-il procéder afin de ne jamais laisser ensemble et sans surveillance le loup et la chèvre, ainsi que la chèvre et le chou ?

Exemple d'application 2

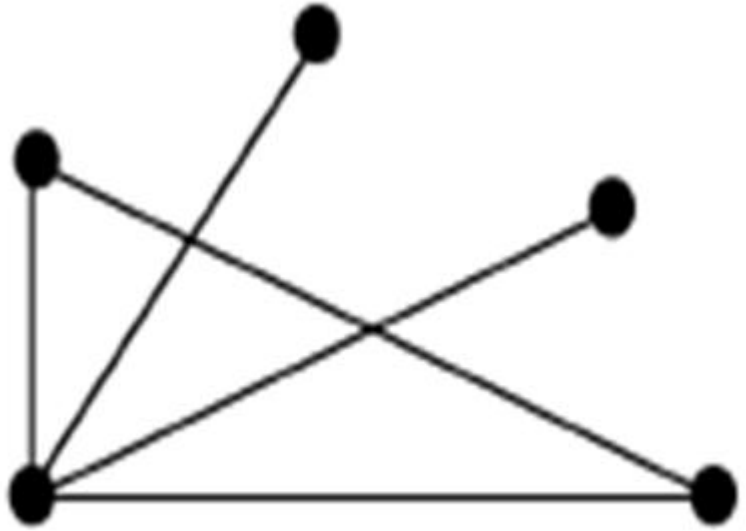
Cette situation peut être modélisée à l'aide d'un graphe.

Désignons par **P** le passeur, par **C** la chèvre, par **X** le chou et par **L** le loup. Les sommets du graphe sont des couples précisant qui est sur la rive initiale, qui est sur l'autre rive. Ainsi, le couple **(PCX,L)** signifie que le passeur est sur la rive initiale avec la chèvre et le chou (qui sont donc sous surveillance), alors que le loup est sur l'autre rive.

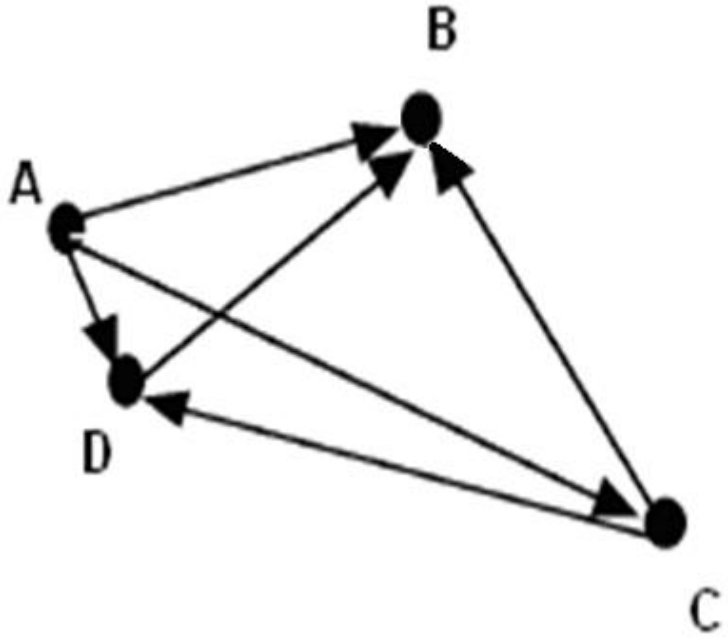


Définitions et terminologies

- **Graphe non orienté :**

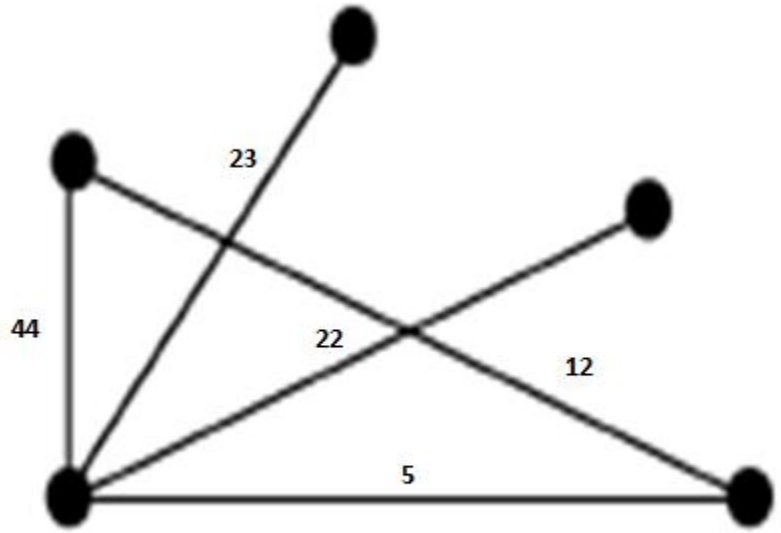


- **Graphe orienté :**

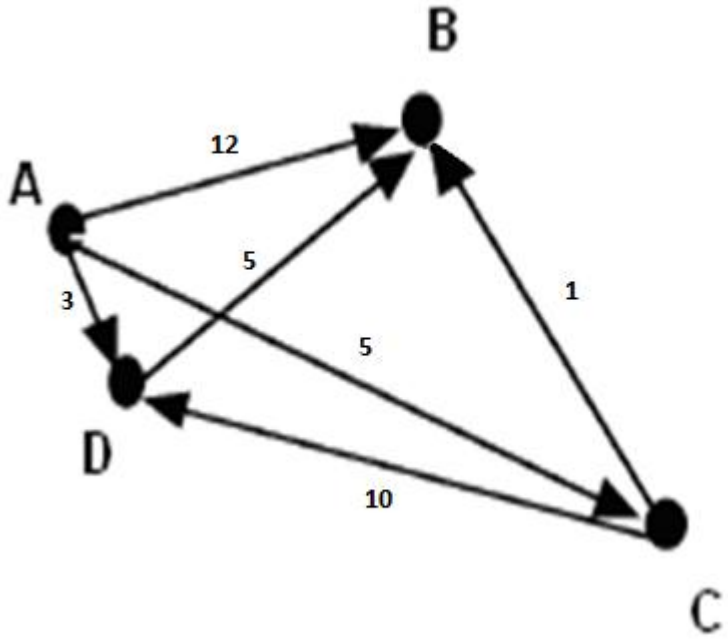


Définitions et terminologies

- **Graphe non orienté pondéré :**

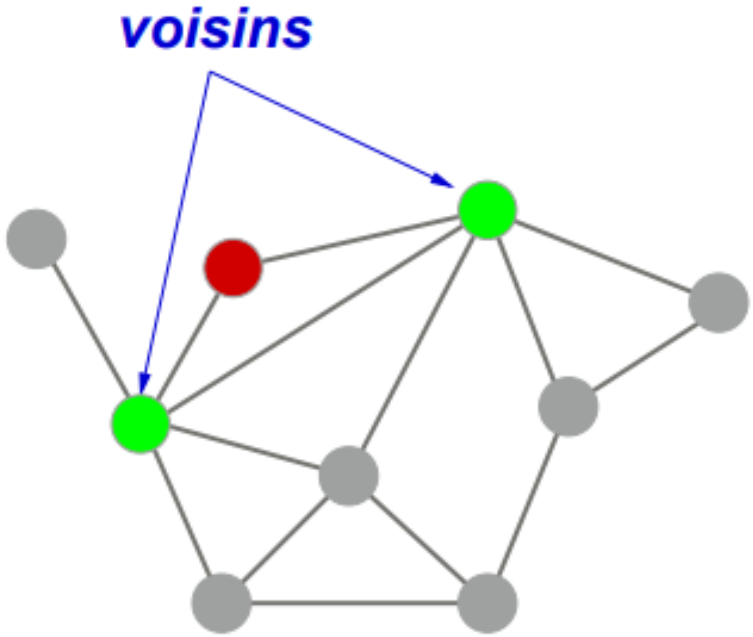


- **Graphe orienté pondéré:**



Définitions et terminologies

Deux sommets d'un graphe sont dits *adjacents* s'il existe une arête (ou un arc) qui les relie.



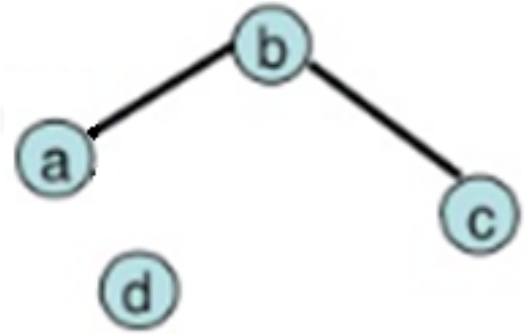
Définitions et terminologies

L'ordre d'un graphe G(S, E) est le nombre de ses sommets. n =|S|

La **taille** d'un graphe G(S, E) est le nombre de ses arêtes. m =|E|

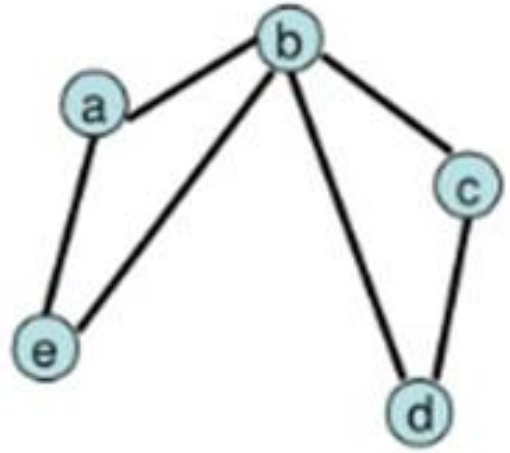
Graphe d'ordre 4

Graphe de taille 2



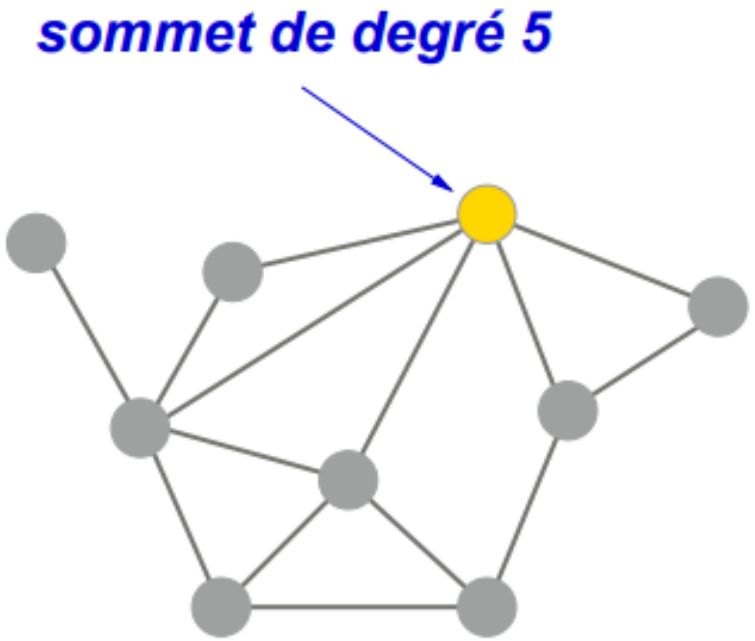
Graphe d'ordre 5

Graphe de taille 6



Définitions et terminologies

Degré d'un sommet : nombre d'arêtes reliées à ce sommet.



Définitions et terminologies

Propriété 1.

Dans un graphe non orienté G = (S, E), la somme des degrés des sommets d'un graphe est égale à deux fois le nombre d'arêtes :

$$\sum_{x \in S} d(x) = 2 \mid E \mid$$

Définitions et terminologies

Exercice 1:

Dans une réunion de 5 personnes, combien de poignées de main sont échangées ? On suppose que tout le monde salue tout le monde

Solution :

On peut modéliser la situation par un graphe non orienté G = (S, E) où les sommets représentent les personnes et les arêtes représentent les poignées de mains échangées.

Le problème revient à trouver le nombre des arêtes possibles (la taille du graphe|E|) sachant que :

- l'ordre du graphe égale à 5 (n = |S| = 5) et
- le degré de chaque sommet égale à 4

$$\sum_{x \in S} d(x) = 2 \mid E \mid \Leftrightarrow 5 * 4 = 2|E \mid \Leftrightarrow \quad E = 10$$

Définitions et terminologies

Exercice 2:

Est-il possible de relier 15 ordinateurs de sorte que chaque ordinateur soit relié exactement avec 3 autres?

Solution :

On peut modéliser la situation par un graphe non orienté G = (S, E) où les sommets

représentent les ordinateurs et les arêtes représentent les liens entre eux.

Le problème revient à trouver le nombre des arêtes possibles (la taille du graphe|E|) sachant que :

- l'ordre du graphe égale à 15 (n = |X| = 15) et
- le degré de chaque sommet égale à 3

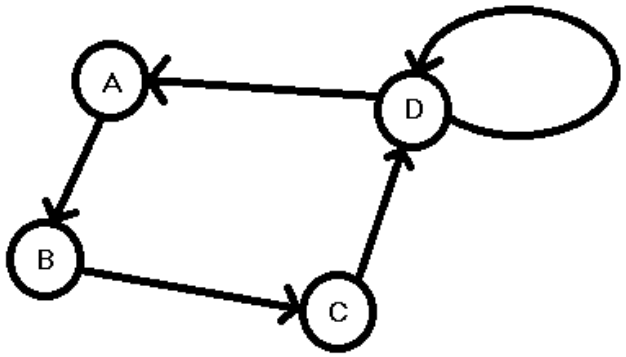
$$\sum_{x \in S} d(x) = 2 \mid E \mid \Leftrightarrow 15 * 3 = 2|E \mid \Leftrightarrow |E| = 22.5$$

Impossible

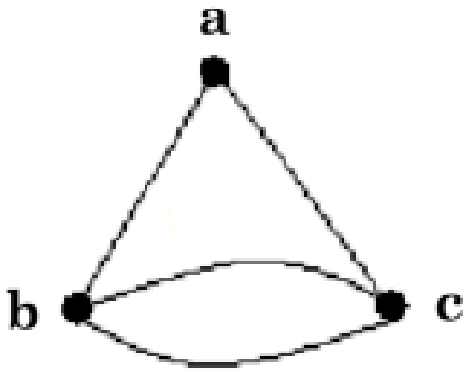
Définitions et terminologies

- *Une boucle $\{x_i, x_i\}$ est une arête reliant un sommet x_i à lui-même.*
- *Un graphe non-orienté est dit simple s'il ne comporte pas de boucles, et s'il ne comporte jamais plus d'une arête entre deux sommets distincts.*
- *Un graphe non orienté qui n'est pas simple est un multi-graphe.*

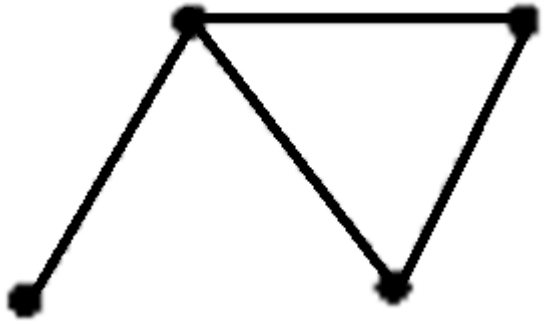
Exemples:



Multi-graphe



Multi-graphe



Graphe simple

Définitions et terminologies

Chaîne : Une chaine de longueur n est une suite de n arêtes qui relient un sommet i à un autre j ou à lui-même.

Exemples :

Des chaînes de longueur 2 :

A D C

A B C

Des chaînes de longueur 3 :

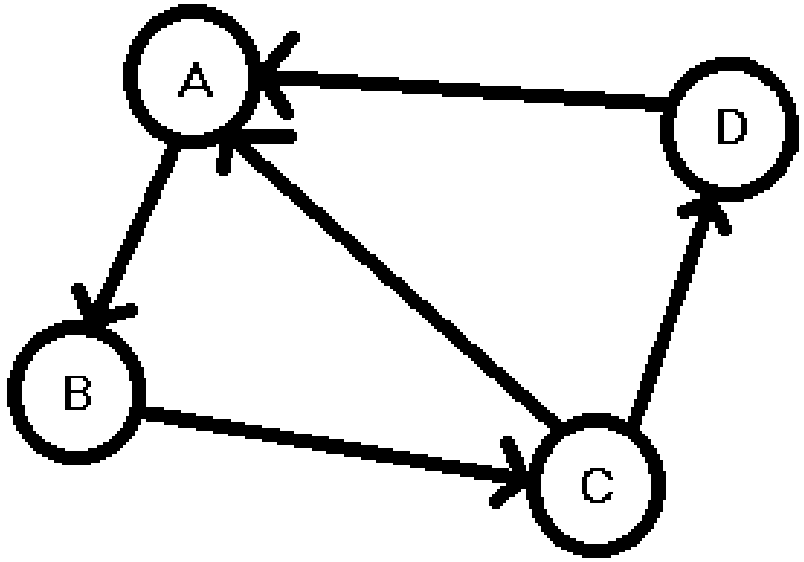
A B C D

B A D C

Des chaînes de longueur 4 :

A B C D A

A B C A D



Définitions et terminologies

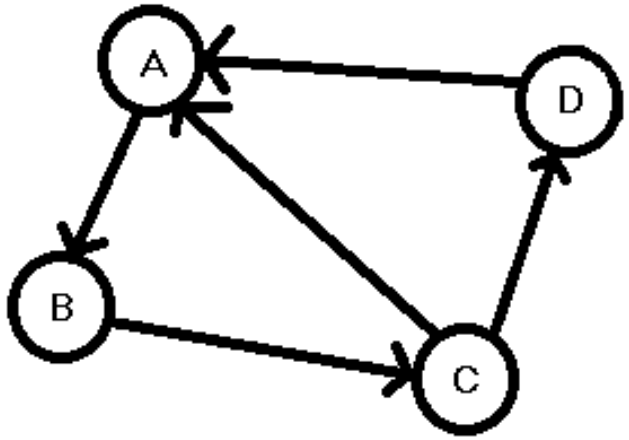
Cycle : Un cycle est défini comme une chaine fermée, tel que l'extrémité initiale est à la fois extrémité finale.

Exemples :

A B C D A est un cycle

A B C A est un cycle

D C A D est un cycle



Définitions et terminologies

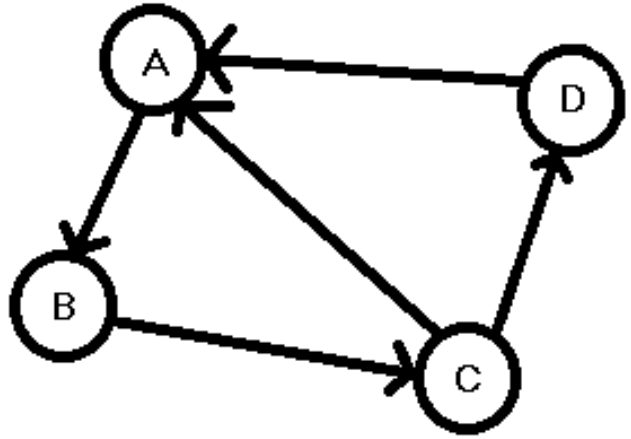
Chemin : c’est une chaine bien orientée

Exemples :

A B C D est un chemin

A D C A B n'est pas un chemin

A B C A D n'est pas un chemin



Définitions et terminologies

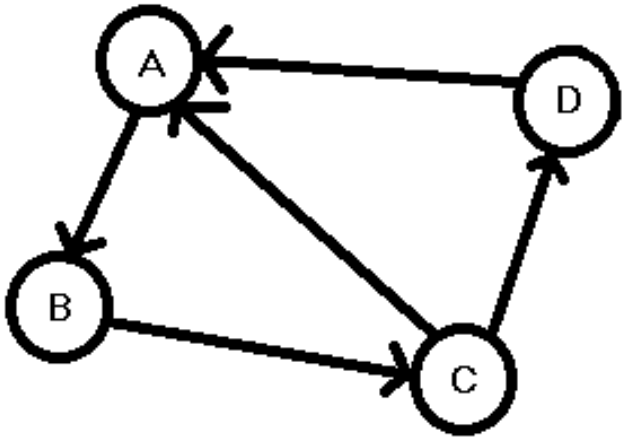
Circuit : est un cycle "bien orienté", à la fois cycle et chemin.

Exemples :

A B C A est un circuit

A B C D A es un circuit

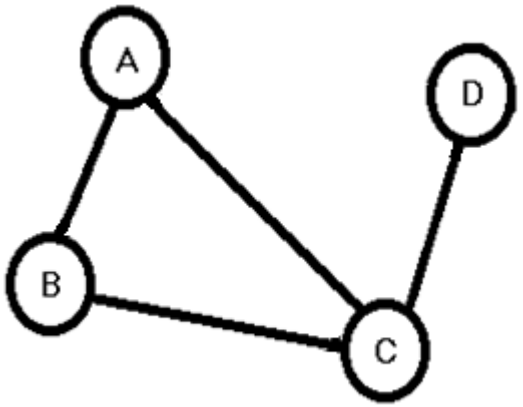
A D C A est un cycle mais pas un circuit



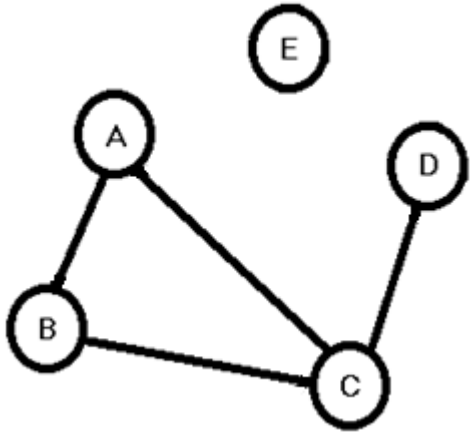
Définitions et terminologies

Graphe Connexe : Un graphe connexe est un graphe dont tout couple de sommets peut être relie par une chaine de longueur n>=1

- **Graphe connexe :**

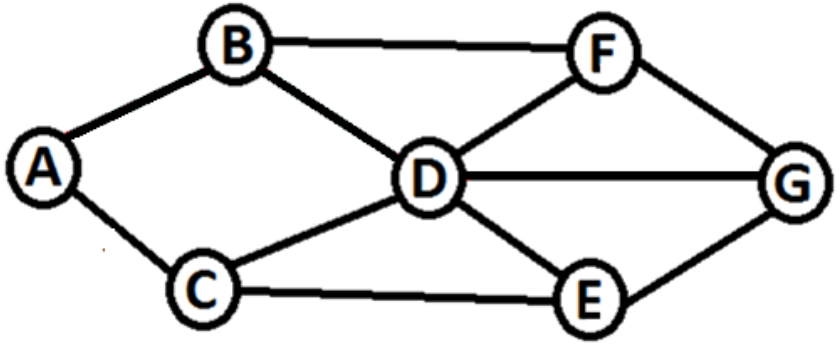


- **Graphe non connexe:**

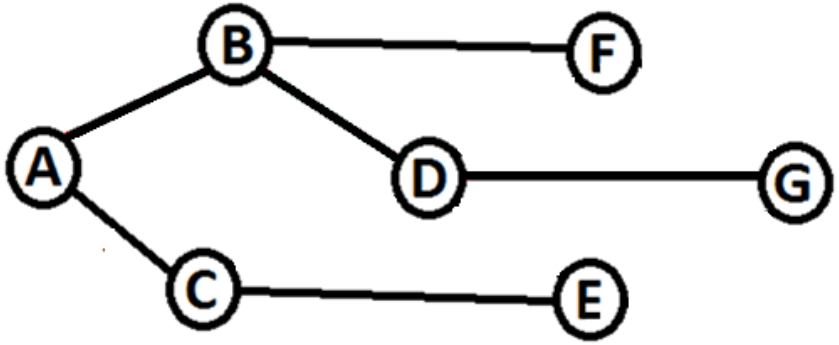


Définitions et terminologies

Propriété 2 : Un graphe est connexe s'il possède un arbre couvrant.



Graphe G



Un Arbre couvrant de G

Algorithme de recherche d'un arbre minimal d'un graphe :

Algorithme de Kuskal

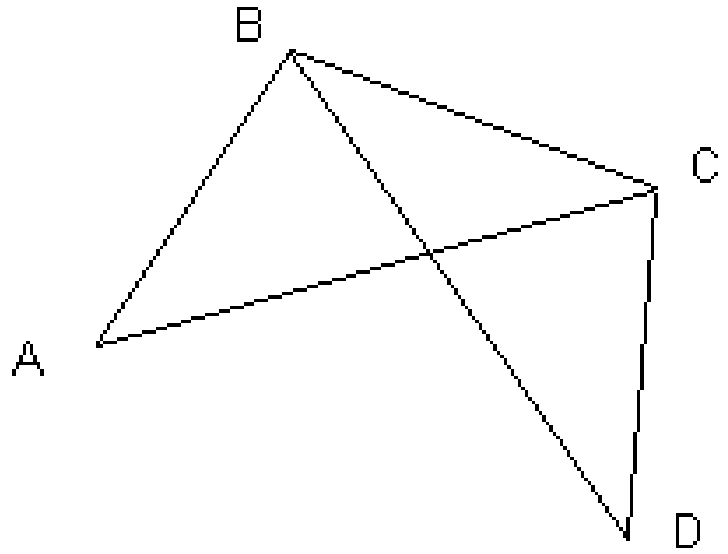
Algorithme de Prime

Définitions et terminologies

Chaîne hamiltonienne : Chaîne passant une seule fois par tous les sommets d'un graphe.

Exemples :

ABCD, ABDC, ACBD,ACBD, ACDB



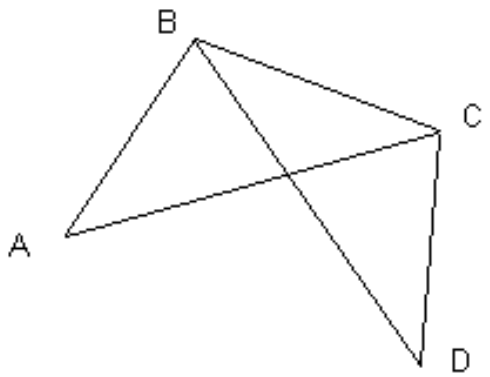
Définitions et terminologies

Chaîne eulérienne : Chaîne passant une seule fois par toutes les arêtes d'un graphe.

Exemples :

BACBDC est une chaîne eulérienne

ACDBACB n'est pas une chaîne eulérienne



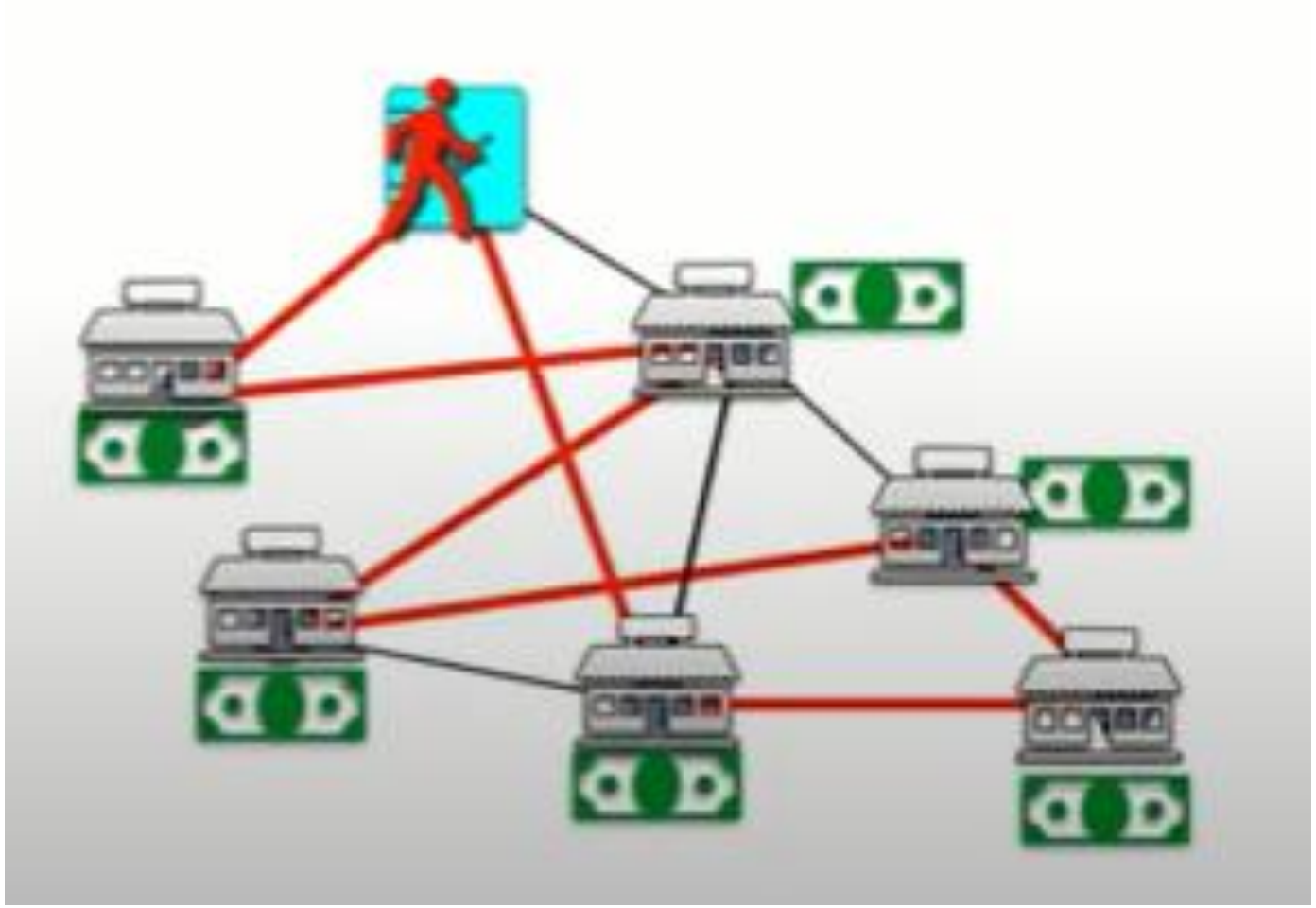
Définitions et terminologies

Cycle hamiltonien : passant une seule fois par tous les sommets d'un graphe et revenant au sommet de départ

Cycle eulérien : passant une seule fois par toutes les arêtes d'un graphe et revenant au sommet de départ.

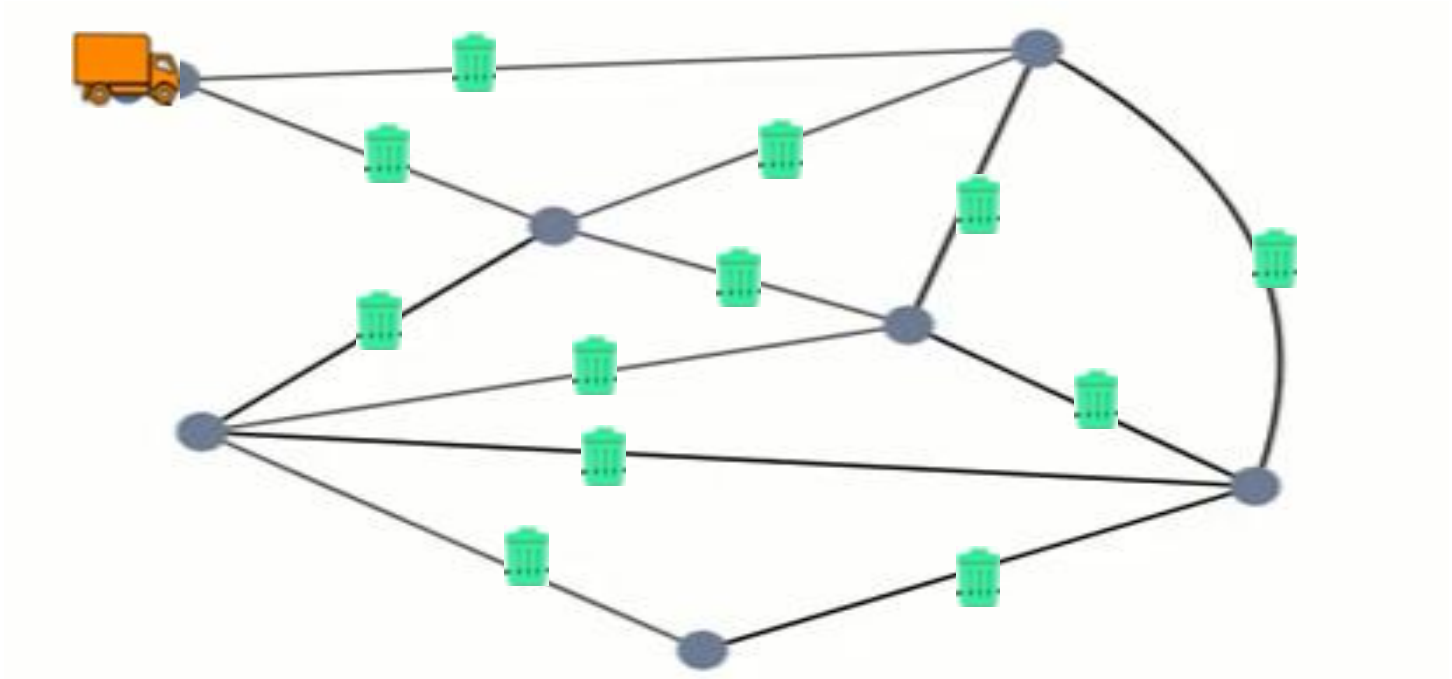
Définitions et terminologies

Exemple cycle Hamiltonien :



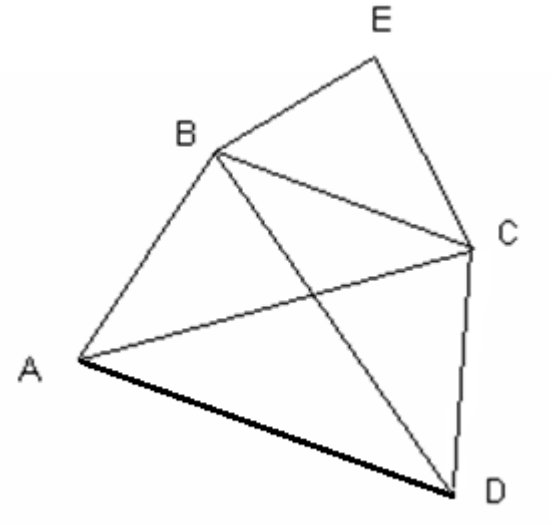
Définitions et terminologies

Exemple cycle Eulérien :

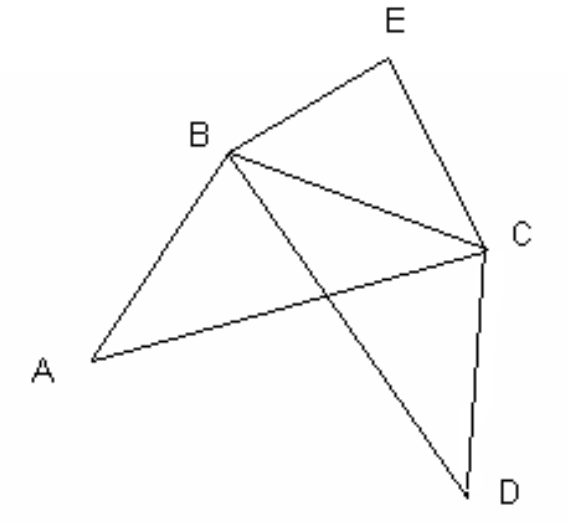


Définitions et terminologies

Existe-t-il un cycle eulérien ?



Réponse :
NON



Réponse : ABECDBCA

Définitions et terminologies

- **Graphe hamiltonien :** Graphe qui possède au moins un cycle Hamiltonien

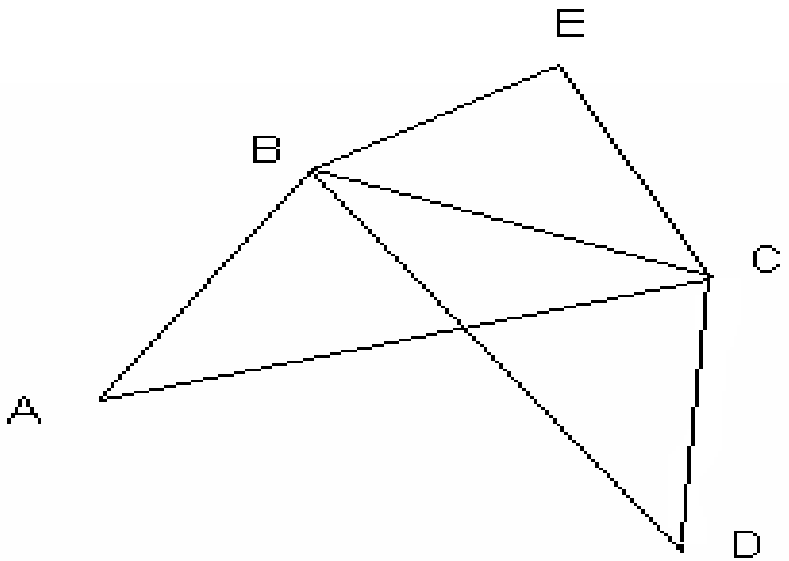
- **Graphe eulérien :** Graphe qui possède au moins un cycle Eulérien

Définitions et terminologies

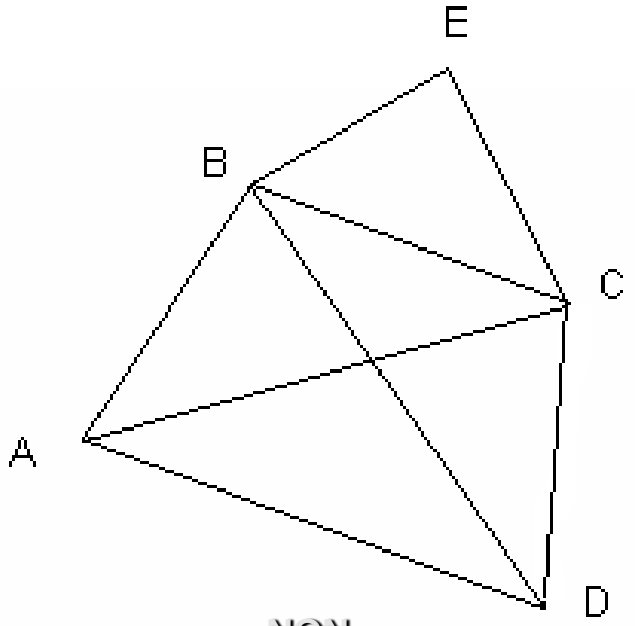


Théorème d'Euler (1766)

Un graphe est eulérien si tous les sommets du graphe ont un degré pair



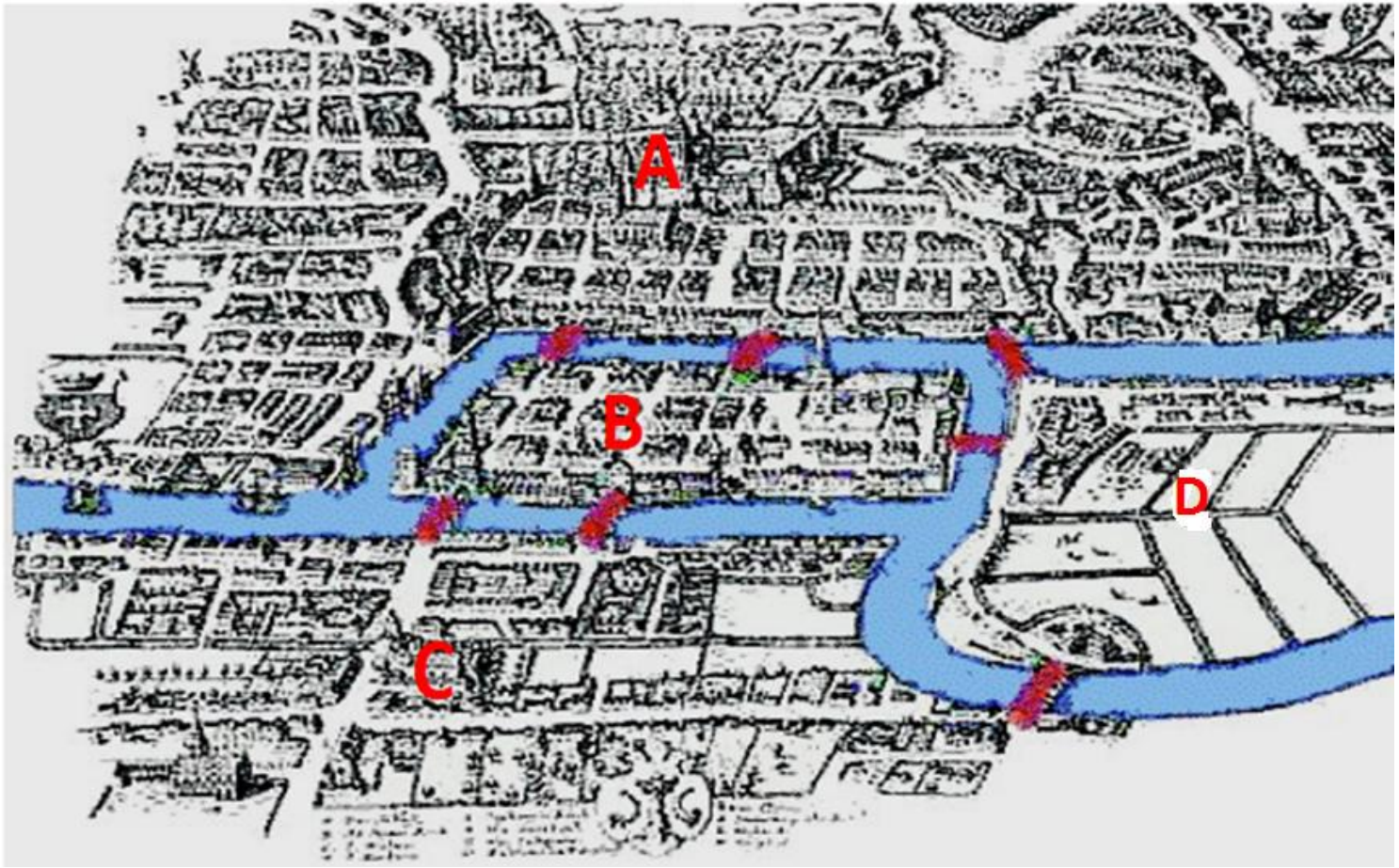
OUI



NON

Définitions et terminologies

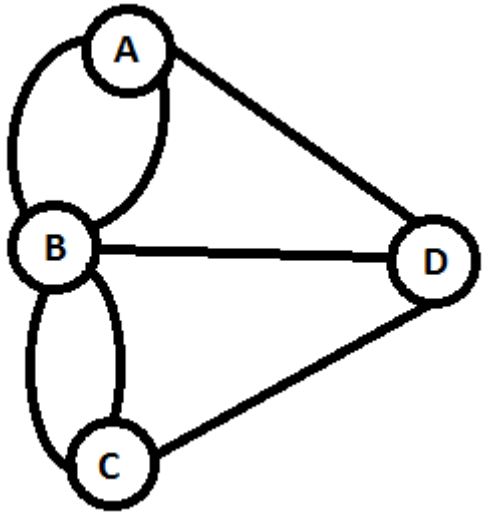
Retour à Konigsberg



Définitions et terminologies

Sous forme de graphe

- **Les sommets = quartiers**
- **Les arcs = Les ponts**
- **Le problème ⇔ le graphe est il eulérien ?**
- **Théorème ⇒ NON**



Implémentation des graphes

Deux implémentations possibles :

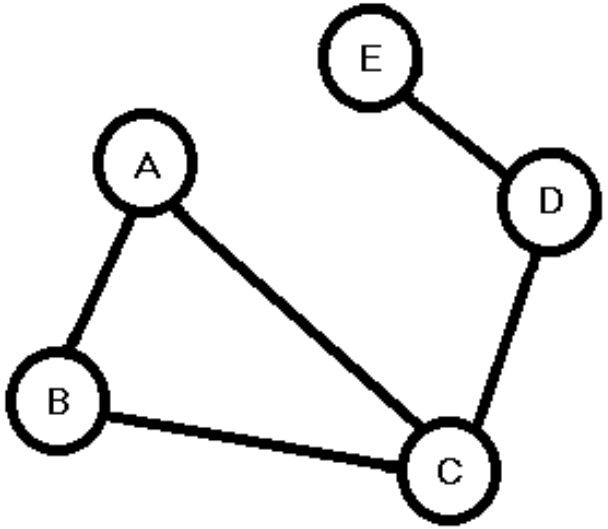
1. Par les listes d’adjacences

2. Par la matrice d’adjacences

Implémentation des graphes

Implémentation par : Matrice d'adjacence

Graphe non orienté :

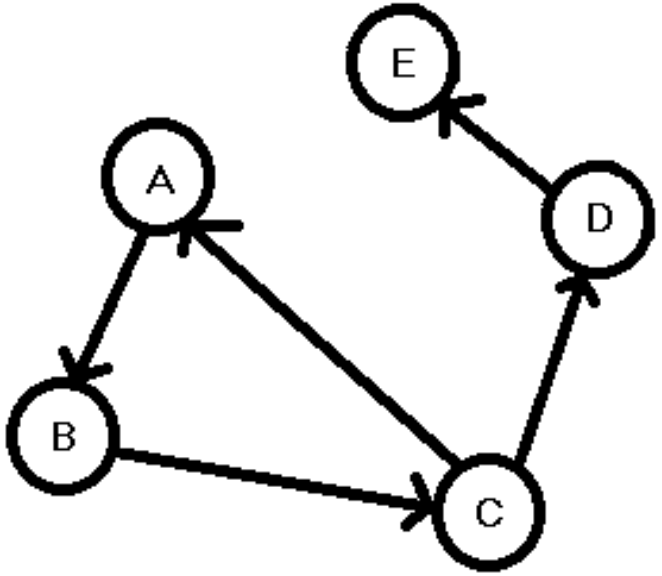


	A	B	C	D	E
A	0	**1**	**1**	0	0
B	**1**	0	**1**	0	0
C	**1**	**1**	0	**1**	0
D	0	0	**1**	0	**1**
E	0	0	0	**1**	0

Implémentation des graphes

Implémentation par : Matrice d'adjacence

Graphe orienté :

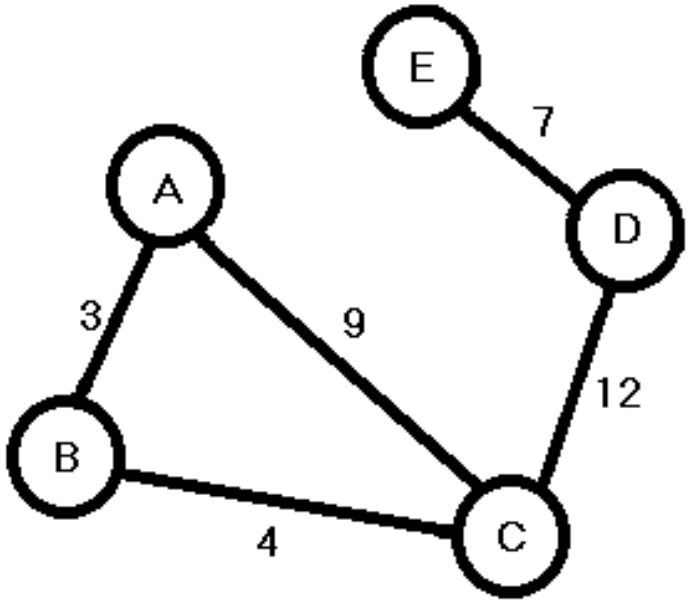


	A	B	C	D	E
A	0	**1**	0	0	0
B	0	0	**1**	0	0
C	**1**	0	0	**1**	0
D	0	0	0	0	**1**
E	0	0	0	0	0

Implémentation des graphes

Implémentation par : Matrice d'adjacence

Graphe non orienté et valué :

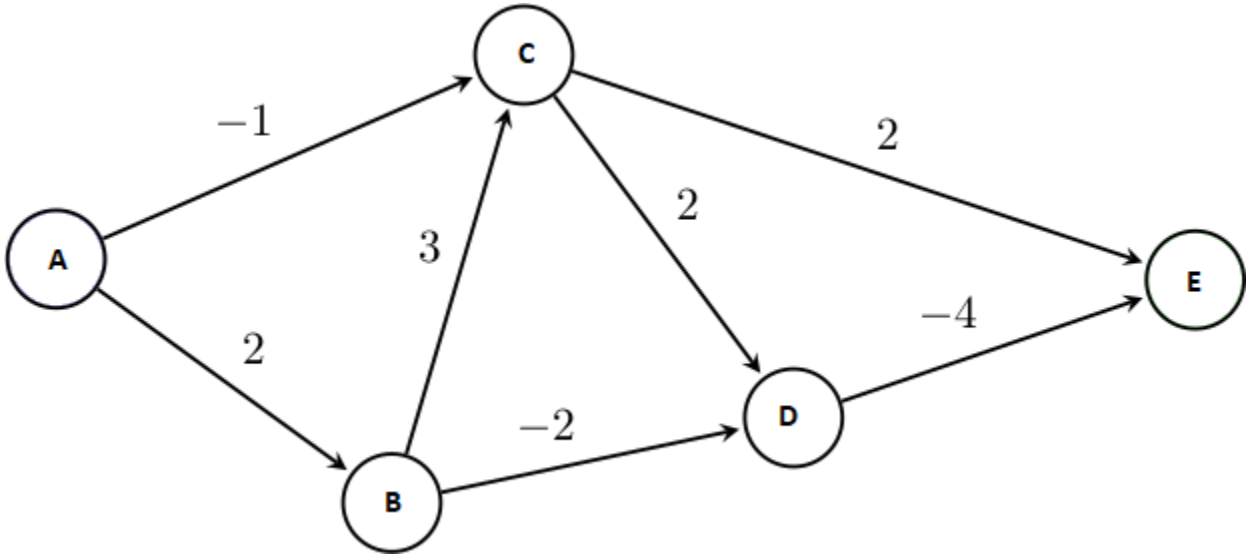


	A	B	C	D	E
A	0	**3**	**9**	0	0
B	**3**	0	**4**	0	0
C	**9**	**4**	0	**12**	0
D	0	0	**12**	0	**7**
E	0	0	0	**7**	0

Implémentation des graphes

Implémentation par : Matrice d'adjacence

Exercice : Donner l'implémentation avec la matrice d'adjacence du graphe suivant.



0	2	-1	0	0
0	0	3	-2	0
0	0	0	2	2
0	0	0	0	-4
0	0	0	0	0

Implémentation des graphes

Implémentation par : Matrice d'adjacence

```
#define MAX 100
typedef struct {
    int matrice[MAX][MAX];
    int ordre;
} Graphe;
```

Algorithmes

- **Parcours d'un graphe**
- **Recherche chemin plus court : Dijkstra, L'algorithme A*,…**
- Coloration de Graphe
- Arbre couvrant minimal
- Détection de la présence de cycles
- Détection de la connexité d'un graphe non orienté
- Distances dans un graphe (Floyd-Warshall).
- ….

Algorithmes de parcours

Le parcours d'un graphe sert à :

- explorer tous les sommets et arêtes,
- vérifier la connectivité,
- trouver un chemin,
- détecter des cycles,
- générer des arbres couvrants,
- résoudre des problèmes (labyrinthes, réseaux, web crawling…).

Le parcours d'un graphe sert à :

- Parcours en largeur
- Parcours en profondeur

Algorithmes de parcours

Parcours en largeur :

Ce parcours peut être utilisé pour :

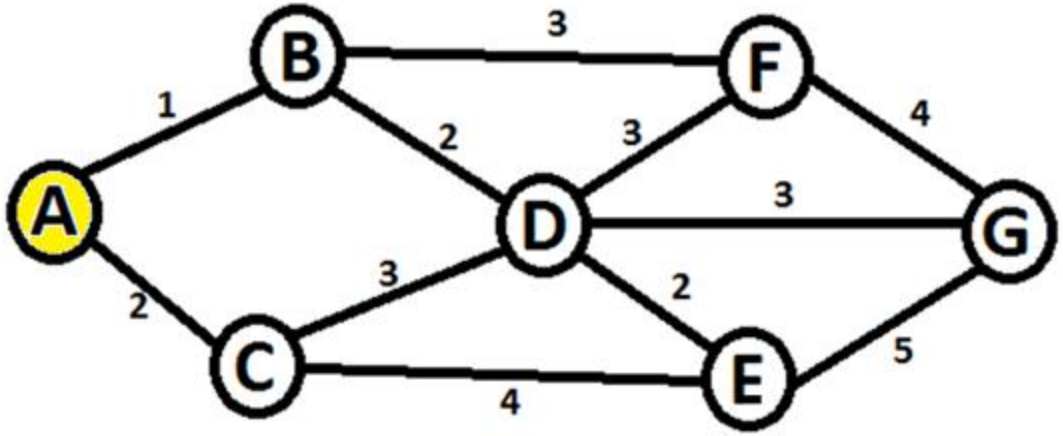
- Chemin plus court entre deux sommets
- Vérifier si un graphe est connexe

Algorithme :

1- **d** est le sommet de départ
2- Cet algorithme liste d'abord les voisins de d pour ensuite les explorer un par un.
3- Répétez (1) pour chaque voisin.

Algorithmes de parcours

Parcours en largeur :

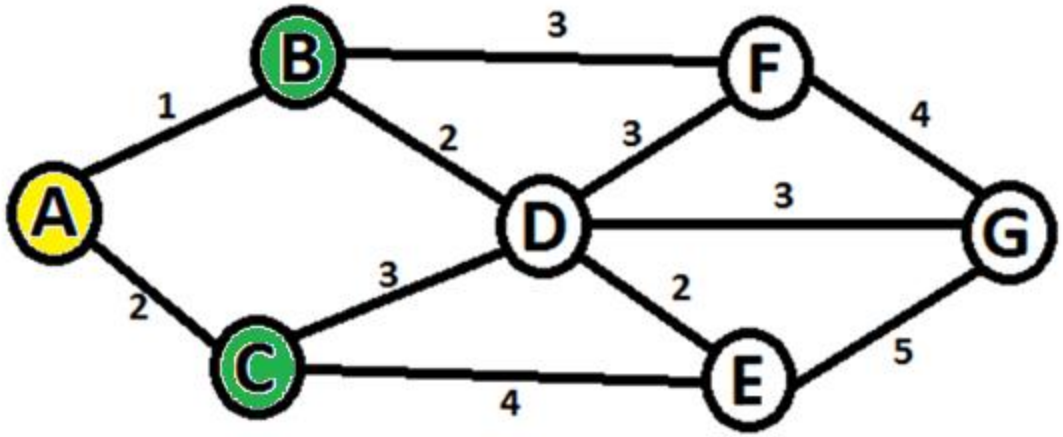


Liste des sommets visités : [ ]

Liste des sommets à visités : [A]

Algorithmes de parcours

Parcours en largeur :

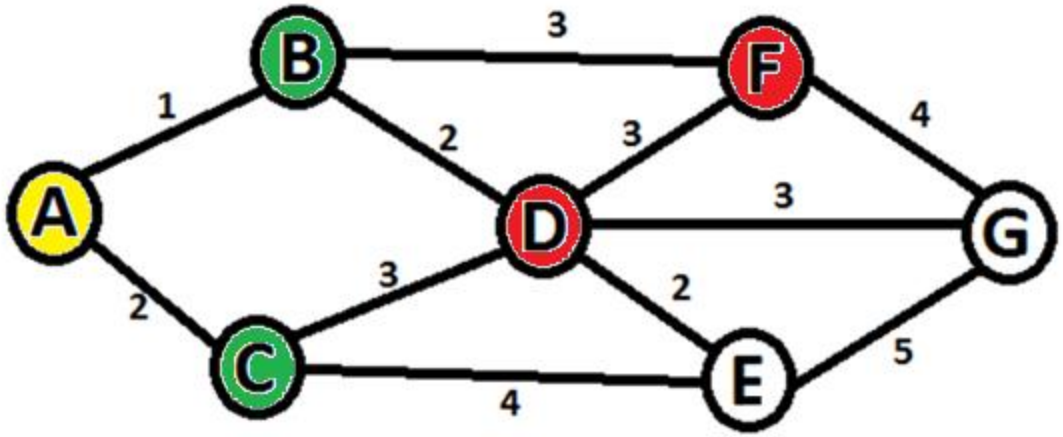


Liste des sommets visités : [A]

Liste des sommets à visités : [B, C]

Algorithmes de parcours

Parcours en largeur :

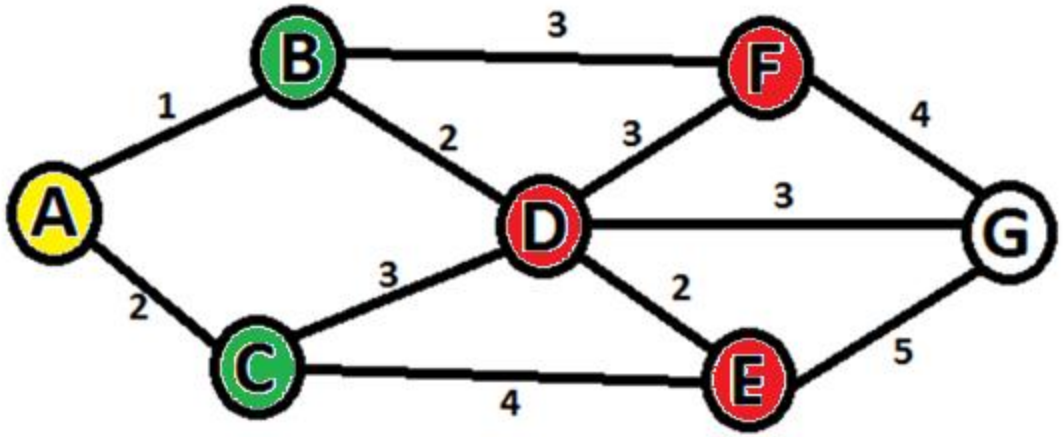


Liste des sommets visités : [A, B]

Liste des sommets à visités : [C, F, D]

Algorithmes de parcours

Parcours en largeur :

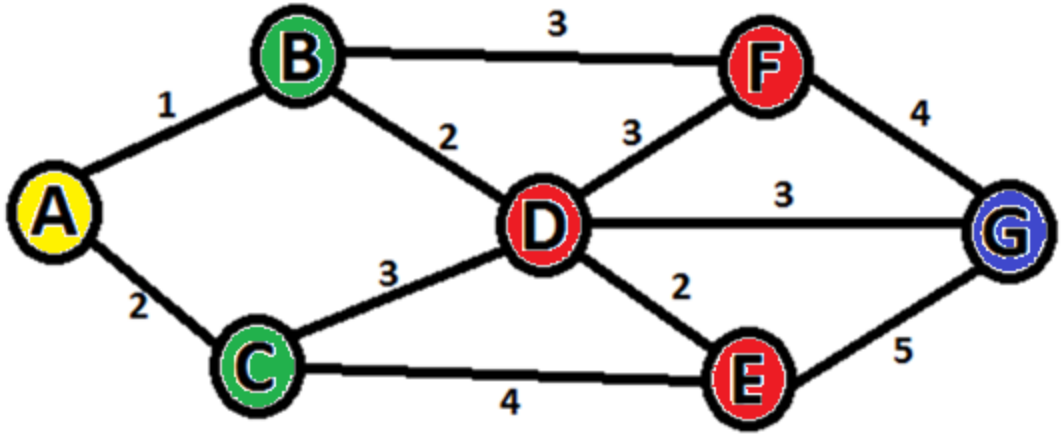


Liste des sommets visités : [A, B, C]

Liste des sommets à visités : [F, D, E]

Algorithmes de parcours

Parcours en largeur :

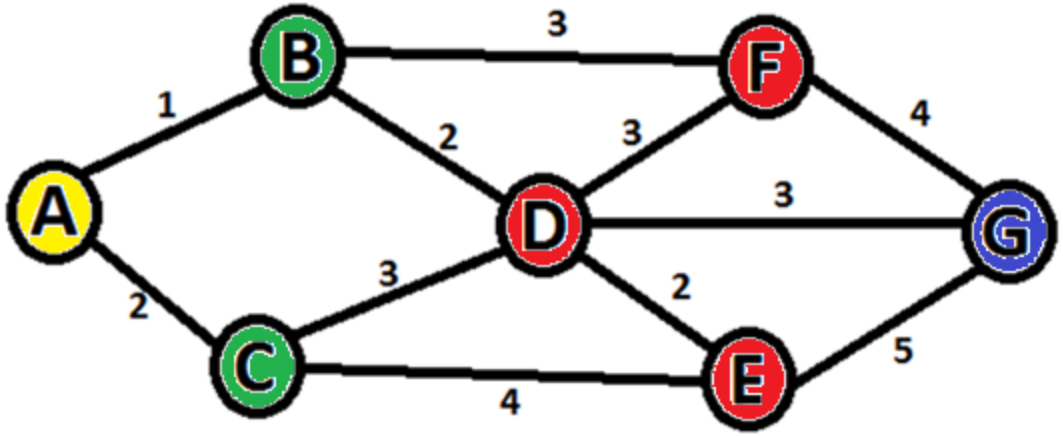


Liste des sommets visités : [A, B, C, F]

Liste des sommets à visités : [D, E, G]

Algorithmes de parcours

Parcours en largeur :

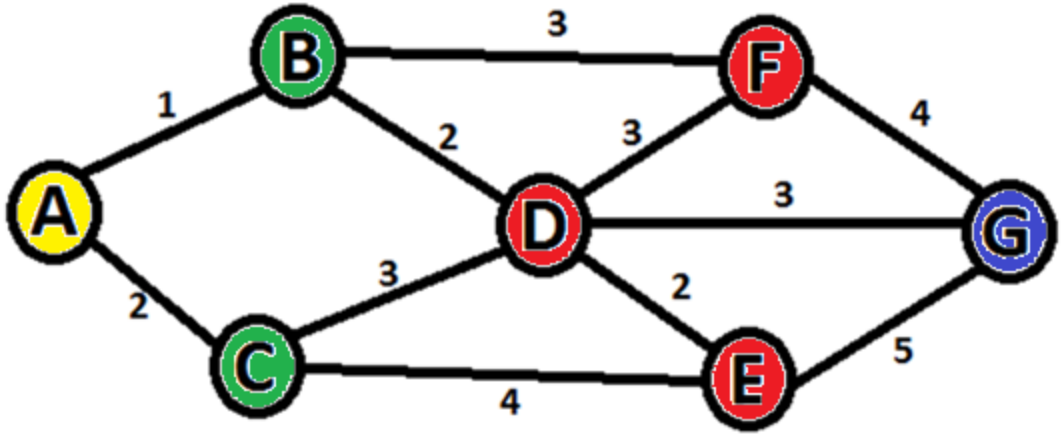


Liste des sommets visités : [A, B, C, F, D]

Liste des sommets à visités : [E, G]

Algorithmes de parcours

Parcours en largeur :

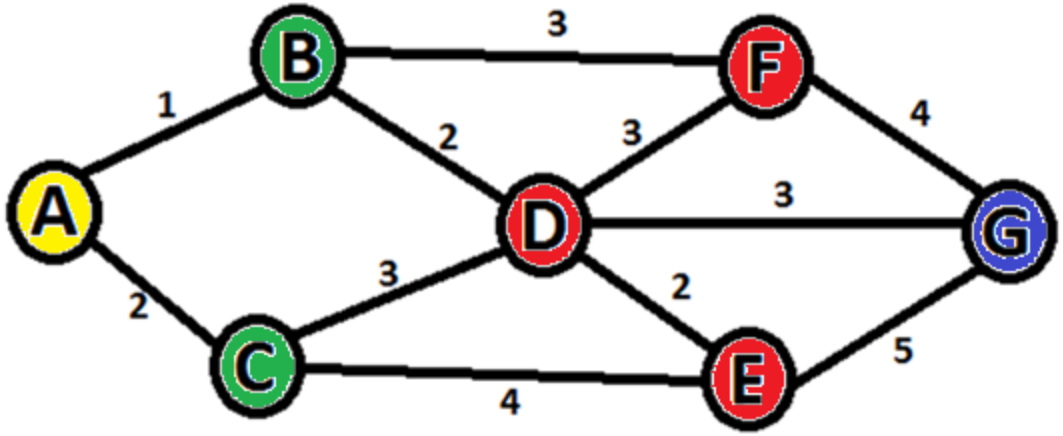


Liste des sommets visités : [A, B, C, F, D, E]

Liste des sommets à visités : [G]

Algorithmes de parcours

Parcours en largeur :



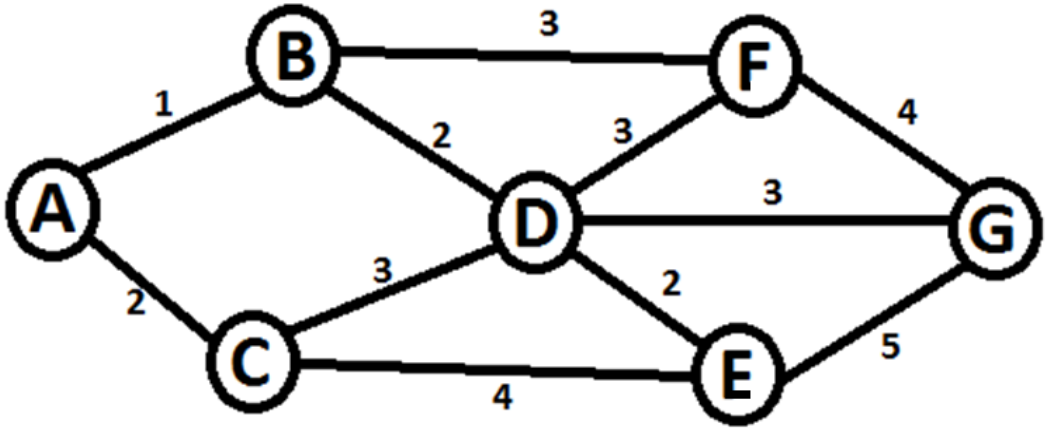
Liste des sommets visités : [A, B, C, F, D, E, G]

Liste des sommets à visités : [ ]

Algorithmes de parcours

Parcours en largeur :

Ecrire la fonction « parcours_largeur(Graphe* g, int debut) » qui fait le parcours en largeur du graphe **G** passé en paramètre à partir du sommet de départ **d**.



Algorithmes de parcours

Parcours en largeur :

```
void parcoursLargeur(Graphe* g, int debut) {
    int visite[MAX] = {0};
    int file[MAX];
    int debutFile = 0, finFile = 0;

    visite[debut] = 1;
    file[finFile++] = debut;

    while (debutFile < finFile) {
        int sommet = file[debutFile++];
        // Traiter le sommet (par exemple, l'afficher)
        for (int i = 0; i < g->ordre; i++) {
            if (g->matrice[sommet][i] == 1 && !visite[i]) {
                visite[i] = 1;
                file[finFile++] = i;
            }
        }
    }
}
```

Algorithmes de parcours

Parcours en profondeur :

Ce parcours peut être utilisé pour :

- Chemin plus court entre deux sommets
- Vérifier si un graphe est connexe

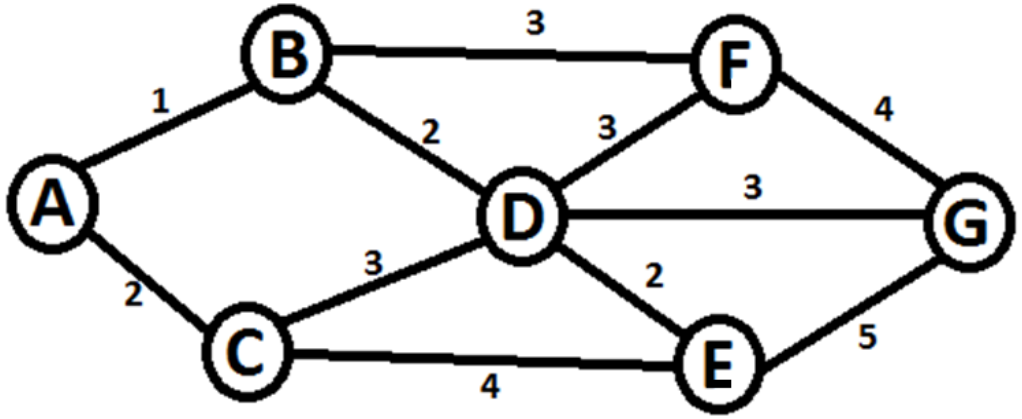
Algorithme :

Recherche qui progresse à partir d'un sommet d en s'appelant récursivement pour chaque sommet voisin de d :
Pour chaque sommet, il prend le premier sommet voisin jusqu'à ce qu'un sommet n'ait plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père.

Algorithmes de parcours

Parcours en profondeur :

Ecrire la fonction « parcours_profondeur» qui fait le parcours en profondeur du graphe **G** passé en paramètre à partir du sommet de départ **d**.



Le résultat du parcours en profondeur du graphe en dessus à partir de A sera : A, B, F ,G ,D, E, C

Algorithmes de parcours

Parcours en profondeur :

```
void parcoursProfondeur(Graphe* g, int sommet, int *visite) {
    visite[sommet] = 1;
    // Traiter le sommet (par exemple, l'afficher)
    for (int i = 0; i < g->ordre; i++) {
        if (g->matrice[sommet][i] == 1 && !visite[i]) {
            parcoursProfondeur(g, i, visite);
        }
    }
}
```

Algorithmes de Dijkstra

En théorie des graphes, l'**algorithme de Dijkstra** sert à résoudre le problème du plus court chemin.

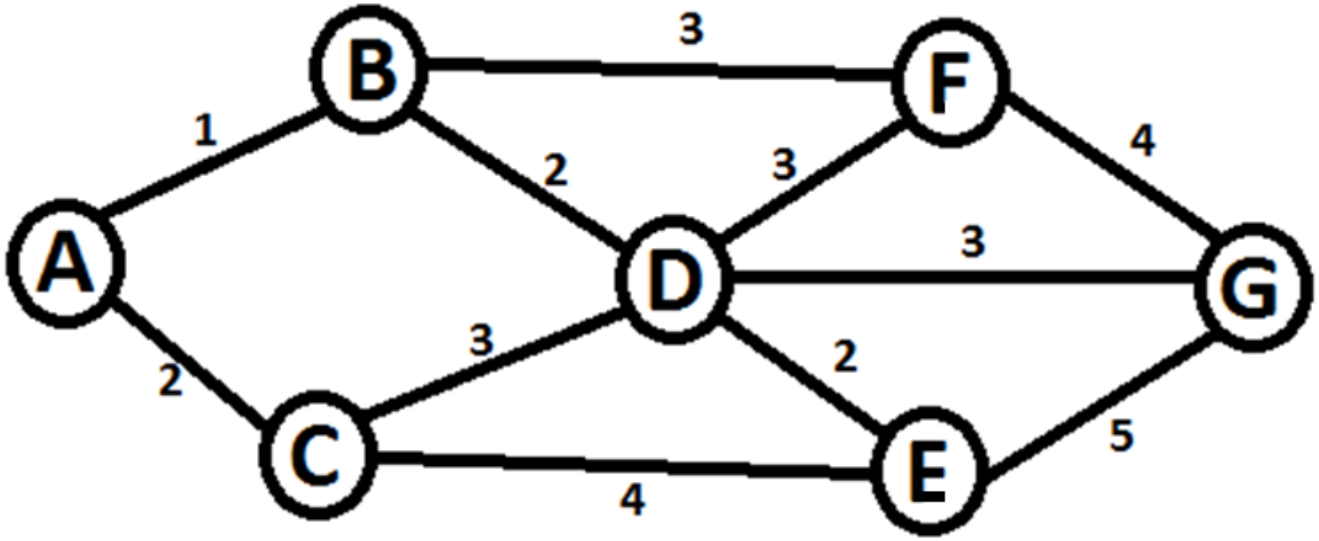
Il s'applique à un graphe connexe dont le poids lié aux arêtes est un réel positif.

Applications :

1. Chemin le plus court entre deux villes
2. Chemin le plus rapide entre deux poins en ville.
3. Routage d’information optimal
4. Intéligence artificielle (A*)
5. ….

Algorithmes de Dijkstra

Exemple : Trouver le chemin optimal entre A et G



Algorithmes de Dijkstra

Initialisation de l'algorithme :

Étape 1 : On affecte le poids 0 au sommet origine (E) et on attribue provisoirement un poids ∞ aux autres sommets.

Répéter les opérations suivantes de l'étape 2 et 3 tant que le sommet de sortie (s) n'est pas affecté d'un poids définitif :

Étape 2 : Parmi les sommets dont le poids n'est pas définitivement fixé choisir le sommet X de poids *p* minimal. Marquer définitivement ce sommet **X** affecté du poids **p(X)**.

Étape 3 : Pour tous les sommets Y qui ne sont pas définitivement marqués, adjacents au dernier sommet fixé **X** :

- Calculer la somme *s* du poids de X et du poids de l'arête reliant X à Y.
- Si la somme *s* est inférieure au poids provisoirement affecté au sommet Y, affecter provisoirement à Y le nouveau poids *s* et indiquer entre parenthèses le sommet X pour se souvenir de sa provenance.

Quand le sommet s est définitivement marqué

Le plus court chemin de E à S s'obtient en écrivant de gauche à droite le parcours en partant de la fin S.

Algorithmes de Dijkstra

Etapes de l’algorithme :

	A	B	C	D	E	F	G
Etape 1							
Etape 2							
Etape 3							
Etape 4							
Etape 5							
Etape 6							
Etape 7							

Algorithmes de Dijkstra

Règles pour remplir les cases de chaque cellule:

Soit **e** le sommet de départ.

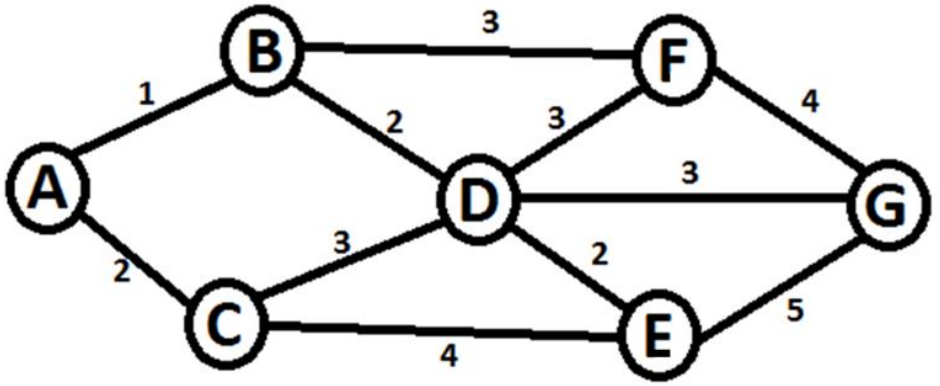
1. La valeur de chaque cellule est un couple : (d(v), p(v))

Où : d(v) est la distance minimale du sommet de départ **e** au somment **v** et p(v) représente le sommet précédent du chemin optimal reliant e et v.

2. d(e)=0 (e est le sommet de départ)
3. d(v)=∞ si la distance est non encore calculé

Algorithmes de Dijkstra

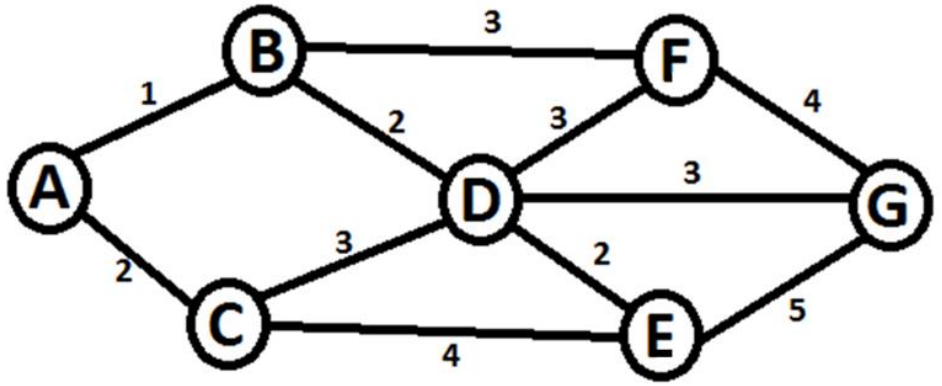
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X						
Etape 3	X						
Etape 4	X						
Etape 5	X						
Etape 6	X						
Etape 7	X						

Algorithmes de Dijkstra

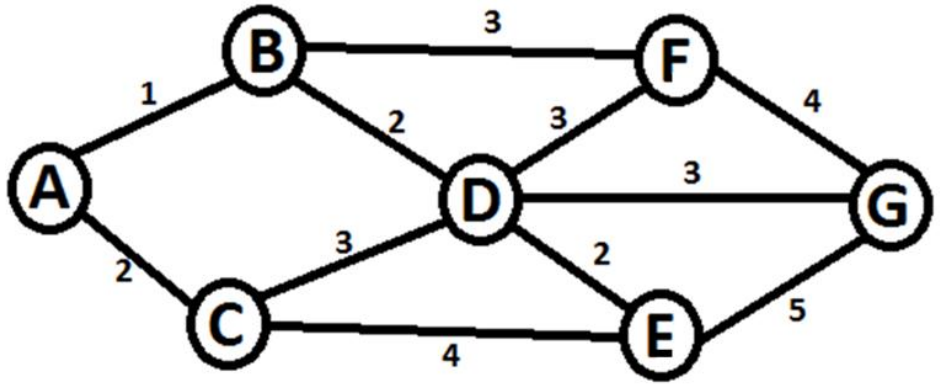
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)				
Etape 3	X						
Etape 4	X						
Etape 5	X						
Etape 6	X						
Etape 7	X						

Algorithmes de Dijkstra

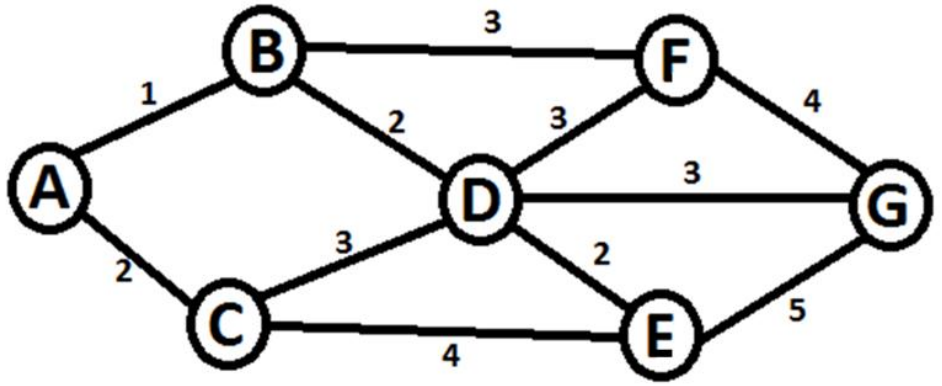
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X						
Etape 4	X						
Etape 5	X						
Etape 6	X						
Etape 7	X						

Algorithmes de Dijkstra

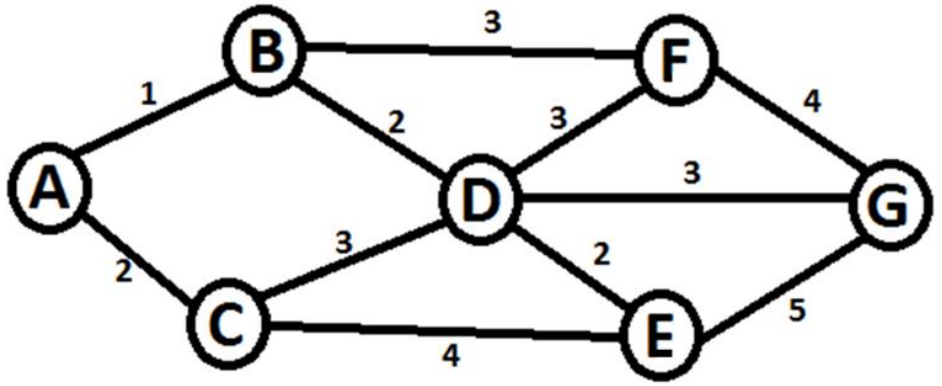
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X					
Etape 4	X	X					
Etape 5	X	X					
Etape 6	X	X					
Etape 7	X	X					

Algorithmes de Dijkstra

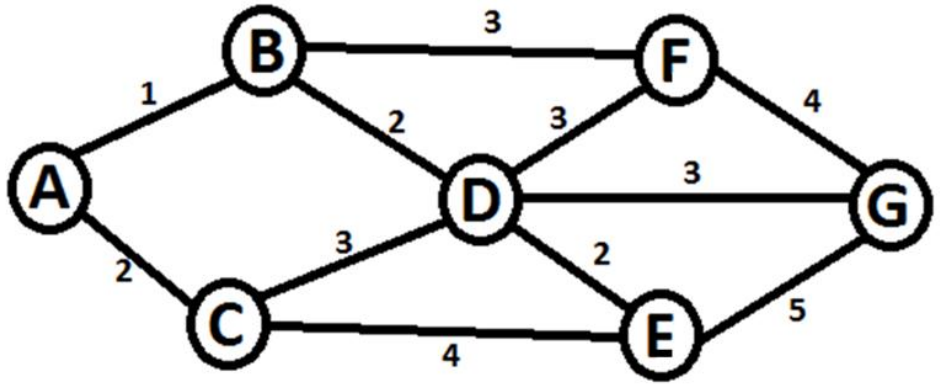
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X		(3,B)		(4,B)	
Etape 4	X	X					
Etape 5	X	X					
Etape 6	X	X					
Etape 7	X	X					

Algorithmes de Dijkstra

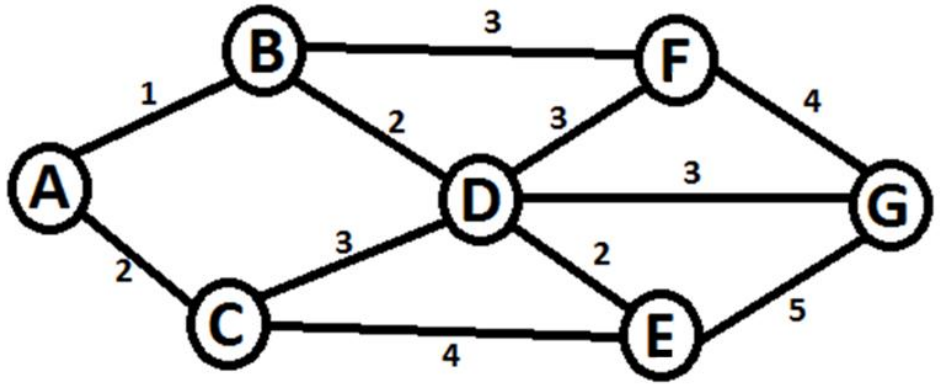
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X	(2,A)	(3,B)	∞	(4,B)	∞
Etape 4	X	X					
Etape 5	X	X					
Etape 6	X	X					
Etape 7	X	X					

Algorithmes de Dijkstra

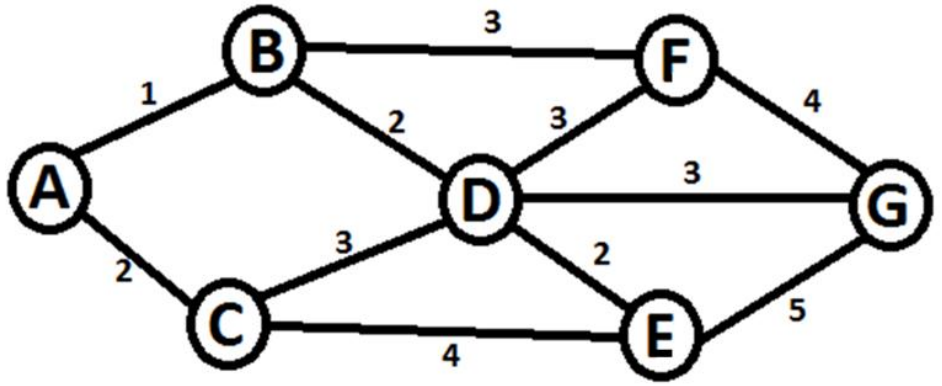
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X	(2,A)	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X		(6, C)		
Etape 5	X	X	X				
Etape 6	X	X	X				
Etape 7	X	X	X				

Algorithmes de Dijkstra

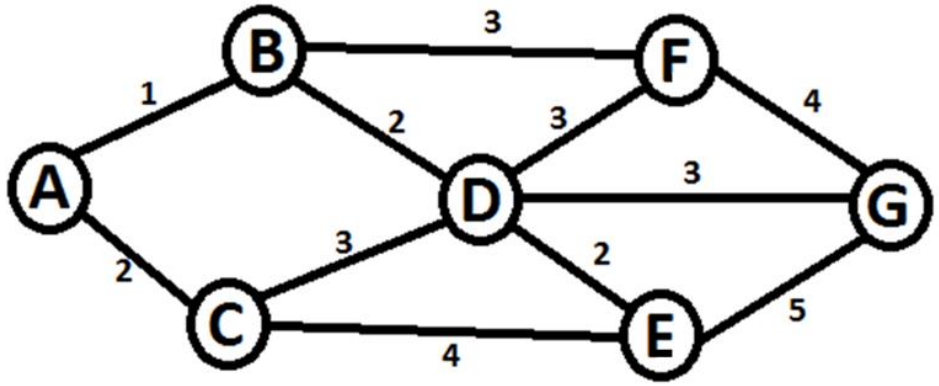
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X	(2,A)	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	(3,B)	(6, C)	(4,B)	∞
Etape 5	X	X	X				
Etape 6	X	X	X				
Etape 7	X	X	X				

Algorithmes de Dijkstra

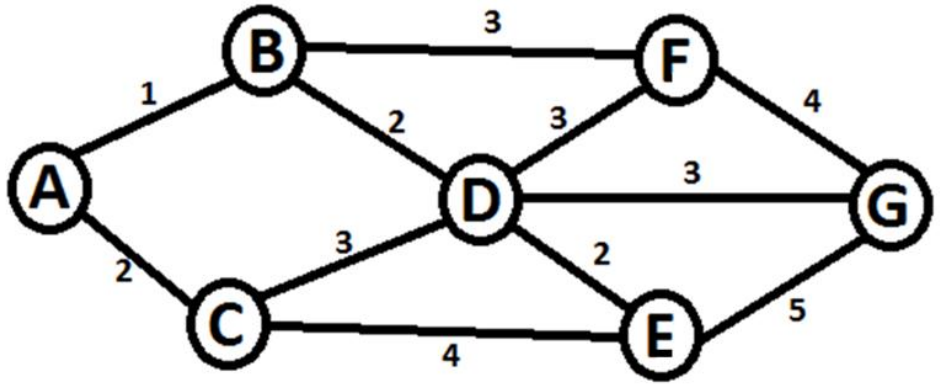
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X	(2,A)	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	(3,B)	(6,C)	(4,B)	∞
Etape 5	X	X	X	X	(5,D)	(4,B)	(6,D)
Etape 6	X	X	X	X			
Etape 7	X	X	X	X			

Algorithmes de Dijkstra

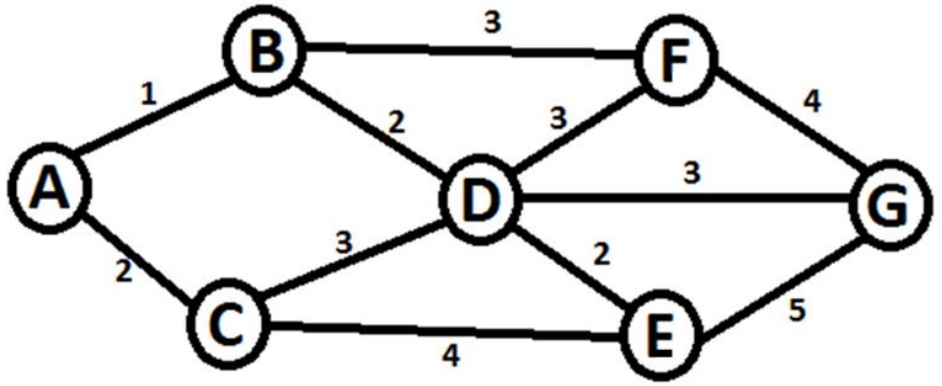
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X	(2,A)	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	(3,B)	(6,C)	(4,B)	∞
Etape 5	X	X	X	X	(5,D)	(4,B)	(6,D)
Etape 6	X	X	X	X		X	
Etape 7	X	X	X	X		X	

Algorithmes de Dijkstra

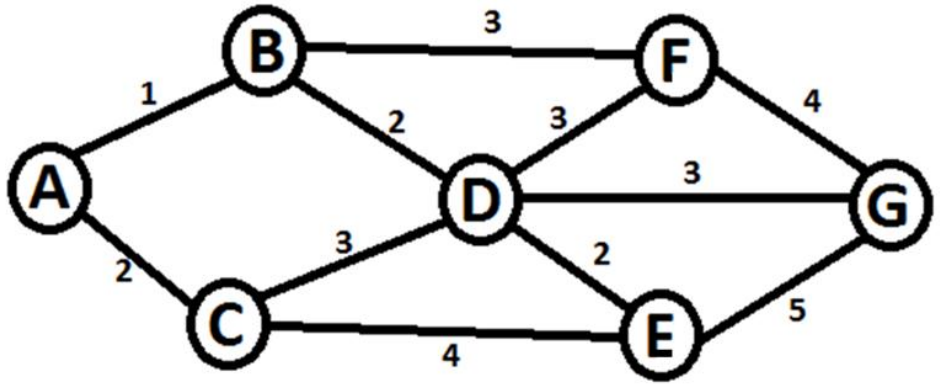
Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X	(2,A)	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	(3,B)	(6,C)	(4,B)	∞
Etape 5	X	X	X	X	(5,D)	(4,B)	(6,D)
Etape 6	X	X	X	X	(5,D)	X	(6,D)
Etape 7	X	X	X	X		X	

Algorithmes de Dijkstra

Etapes de l'algorithme :



	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X	X	(2,A)	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	(3,B)	(6,C)	(4,B)	∞
Etape 5	X	X	X	X	(5,D)	(4,B)	(6,D)
Etape 6	X	X	X	X	(5,D)	X	(6,D)
Etape 7	X	X	X	X	X	X	**(6,D)**

Algorithmes de Dijkstra

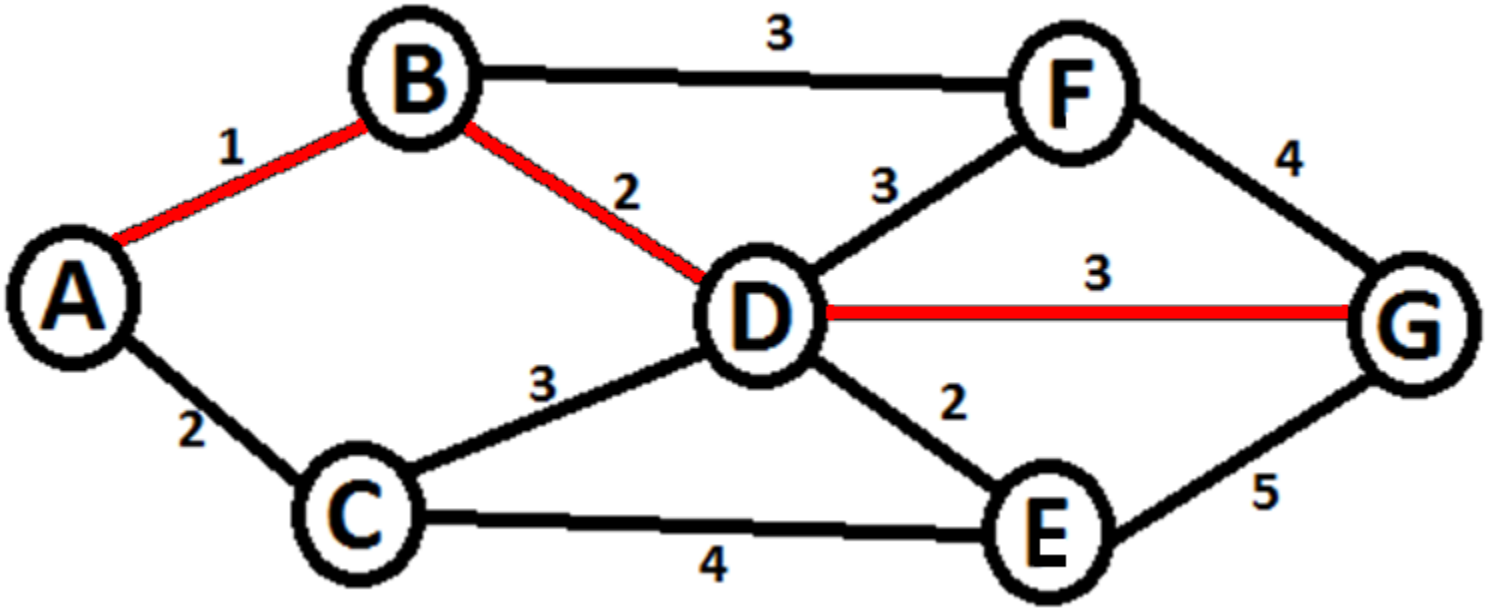
Etapes de l'algorithme :

Etapes de l'algorithme :

	A	B	C	D	E	F	G
Etape 1	(0,)	∞	∞	∞	∞	∞	∞
...							
Etape 7	(0,)	(1,A)	(2,A)	(3,B)	(5,D)	(4,B)	(6,D)

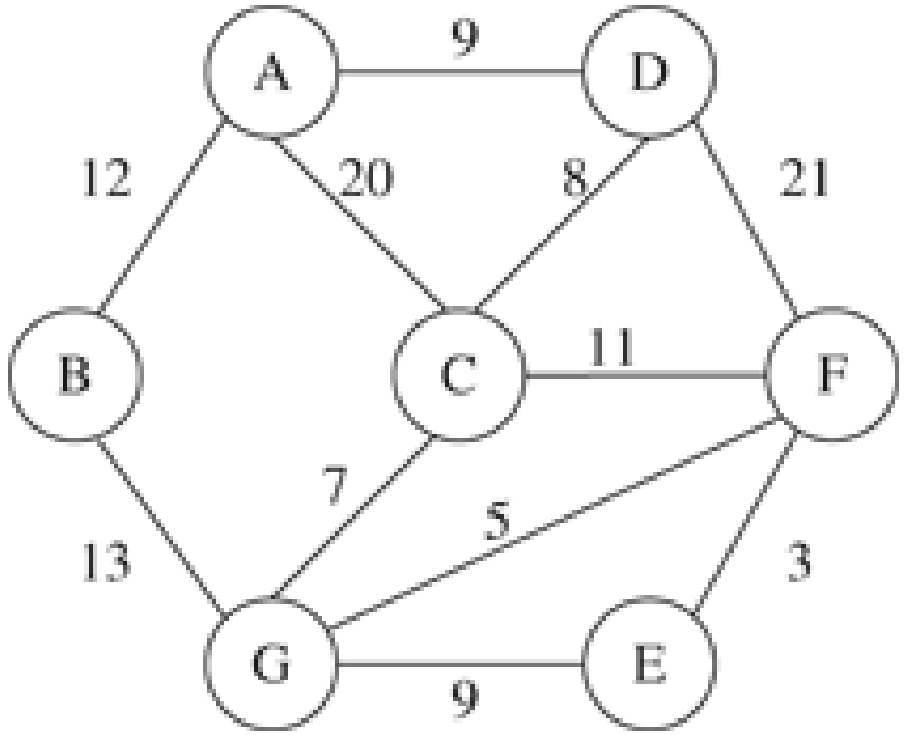
Algorithmes de Dijkstra

Le chemin optimal est : ABDG de coût égal à 6



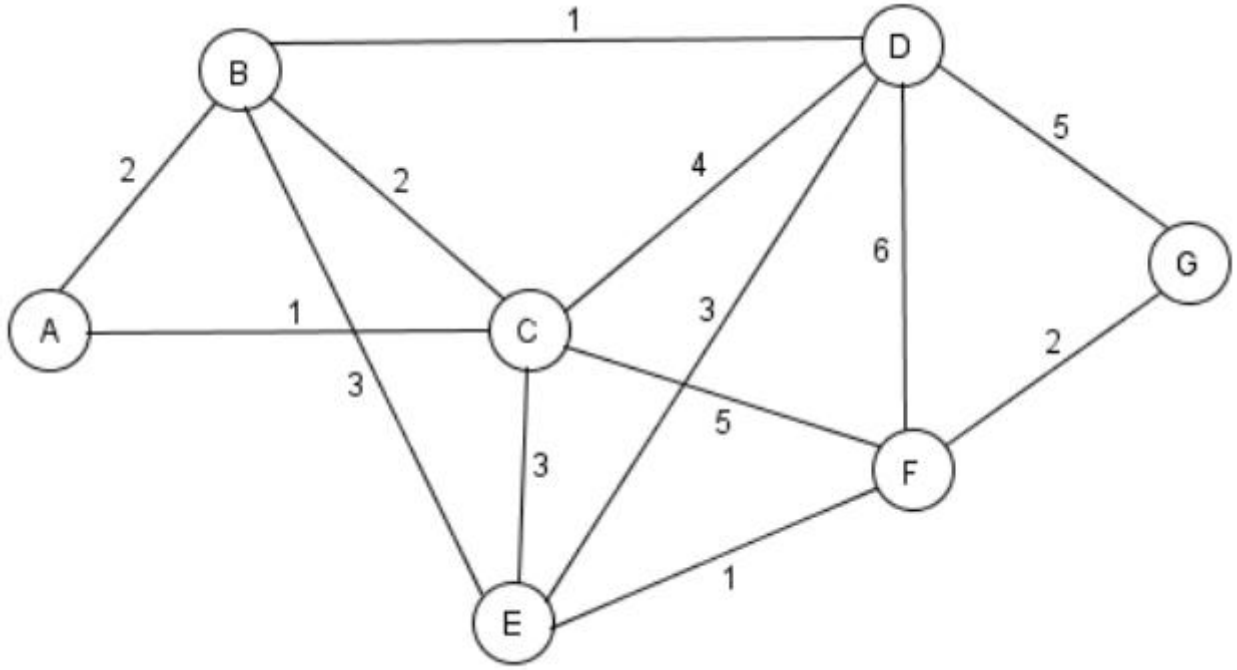
Algorithmes de Dijkstra

Exercice : Trouver le chemin optimal entre A et E



Algorithmes de Dijkstra

Exercice : Trouver le chemin optimal entre A et G



Algorithmes de Dijkstra

Programme de Dijkstra

```c
int minDistance(int dist[], int visited[], int n) {
    int min = INF, index = -1;
    for (int i = 0; i < n; i++) {
        if (!visited[i] && dist[i] < min) {
            min = dist[i];
            index = i;
        }
    }
    return index;
}
```

```c
int* dijkstra_path(Graphe* g, int source, int destination, int* path_length) {

    int n = g->ordre;
    int dist[MAX], visited[MAX], parent[MAX];

    // Initialisation
    for (int i = 0; i < n; i++) {
        dist[i] = INF;
        visited[i] = 0;
        parent[i] = -1;
    }

    dist[source] = 0;

    // Dijkstra
    for (int count = 0; count < n - 1; count++) {

        int u = minDistance(dist, visited, n);
        if (u == -1) break;

        visited[u] = 1;

        for (int v = 0; v < n; v++) {
            int poids = g->matrice[u][v];

            if (!visited[v] &&
                poids > 0 &&
                dist[u] != INF &&
                dist[u] + poids < dist[v]) {

                dist[v] = dist[u] + poids;
                parent[v] = u;   // mémoriser le prédécesseur
            }
        }
    }

    // Vérifier si destination est atteignable
    if (dist[destination] == INF) {
        *path_length = 0;
        return NULL;
    }

    // Reconstruction du chemin (à l'envers)
    int temp_path[MAX];
    int idx = 0;
    int current = destination;

    while (current != -1) {
        temp_path[idx++] = current;
        current = parent[current];
    }

    // Le chemin est inversé → on le remet dans l'ordre
    *path_length = idx;
    int* path = malloc(idx * sizeof(int));

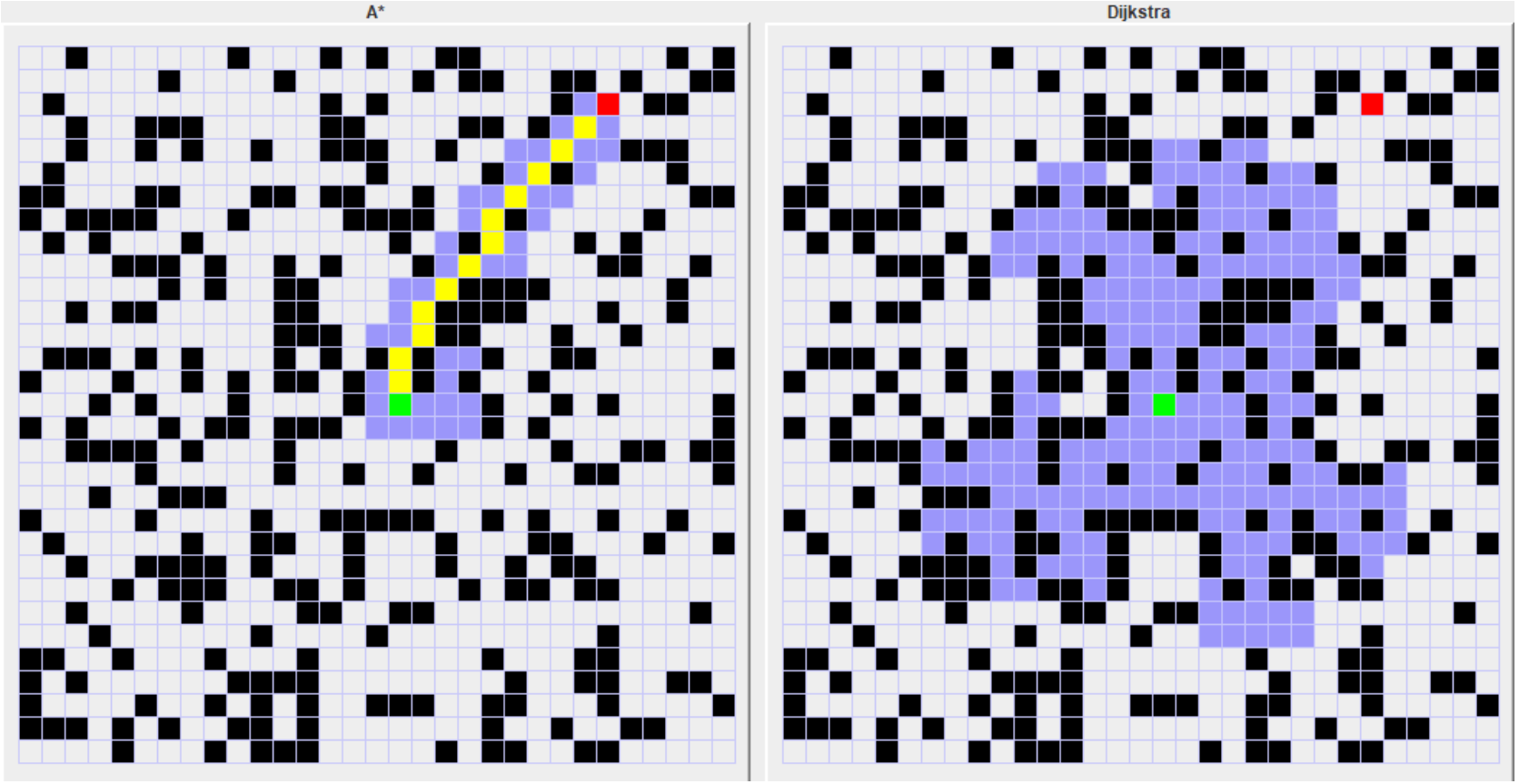
    for (int i = 0; i < idx; i++) {
        path[i] = temp_path[idx - 1 - i];
    }

    return path;
}
```

Algorithme A*

Rechercher le chemin le court :

Dijkstra : Donne toujours une solution optimale mais trop lent

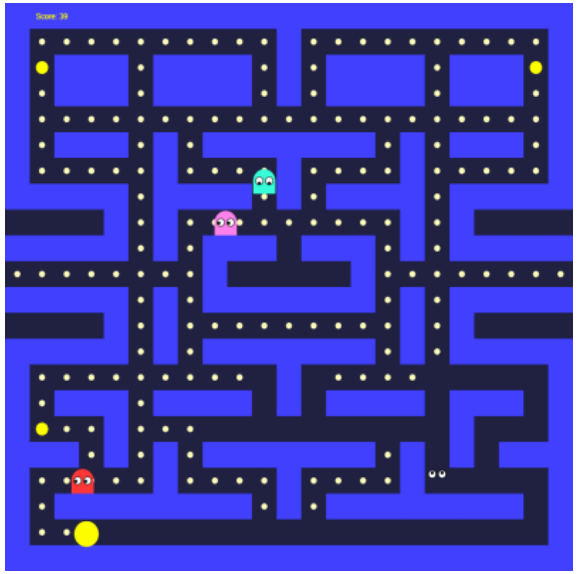


Algorithme A*

L’algorithme A*

- L’algorithme A* est un algorithme heuristique
- Objectif : Trouver très rapidement un court chemin entre deux points avec d’éventuels obstacles.

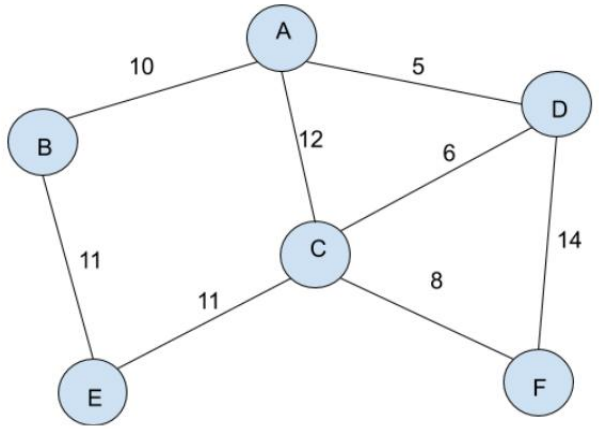
Outre sa vitesse, cet algorithme est réputé pour garantir une solution en sortie.



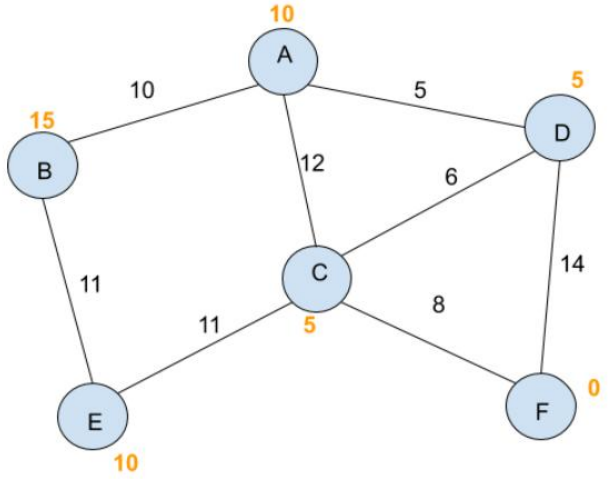
Algorithme A*

A* est une extension de l'algorithme de Dijkstra.

→ Ajout d'une heuristique



Dijkstra se base sur les distances réelles

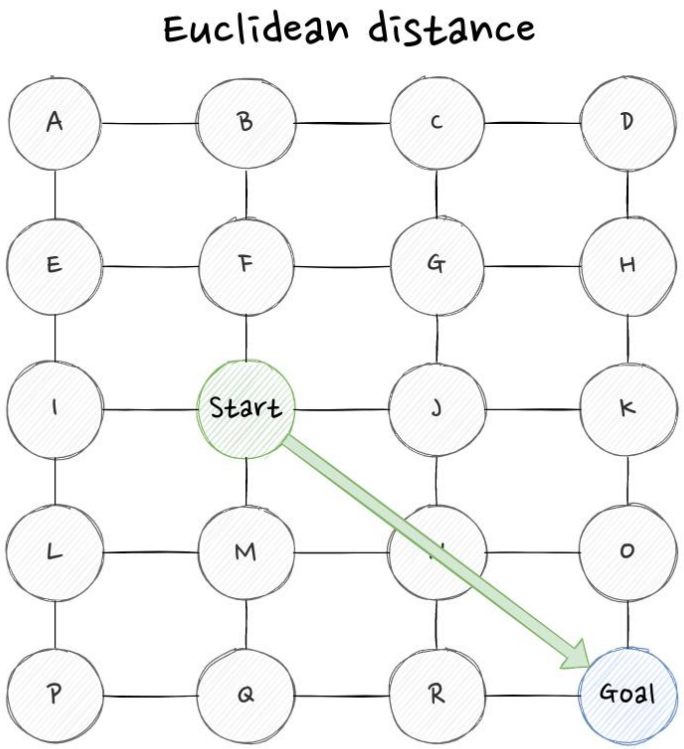


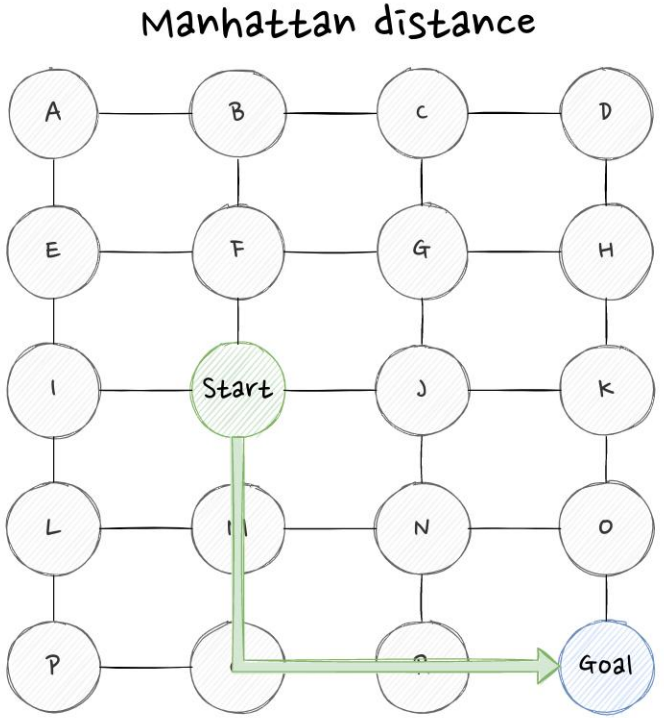
A* utilise distances réelles + une heuristique

Algorithme A*

Exemple heuristique :

Par exemple, L'heuristique peut être obtenu par un calcul de la distance euclidienne (à vol d'oiseau) entre les différents sommets et le sommet but.





Algorithme A*

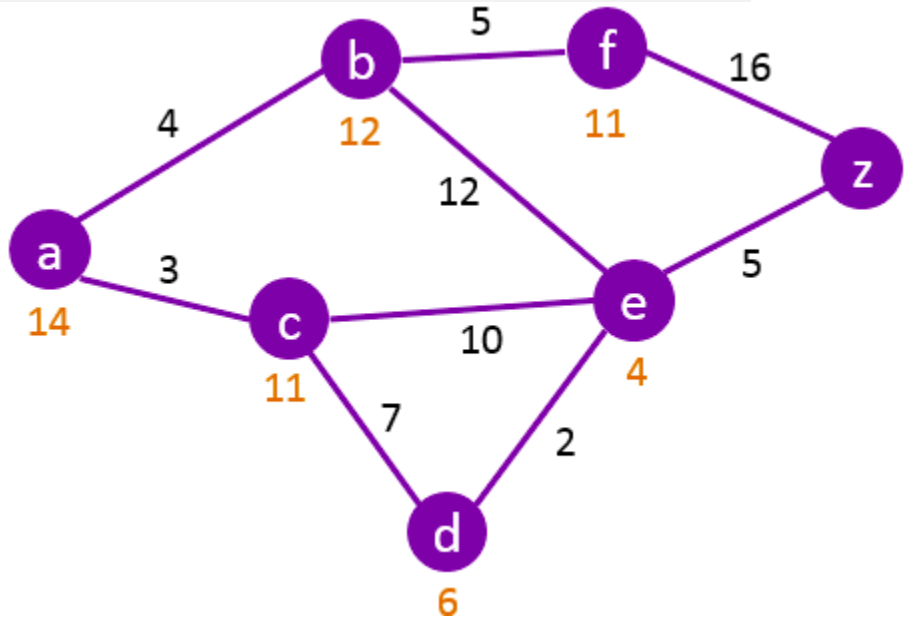
- On évite d'explorer les chemins qui sont déjà chers.
- La mesure d'utilité est donnée par une fonction d'évaluation f.

Pour chaque sommet n, $f(n) = g(n) + h(n)$

- g(n) : est le coût jusqu'à présent pour atteindre n
- h(n) : est le coût estimé pour aller de n vers un état final.
- f(n) : est le coût total estimé pour aller d'un état initial vers un état final en passant par n.

Algorithme A*

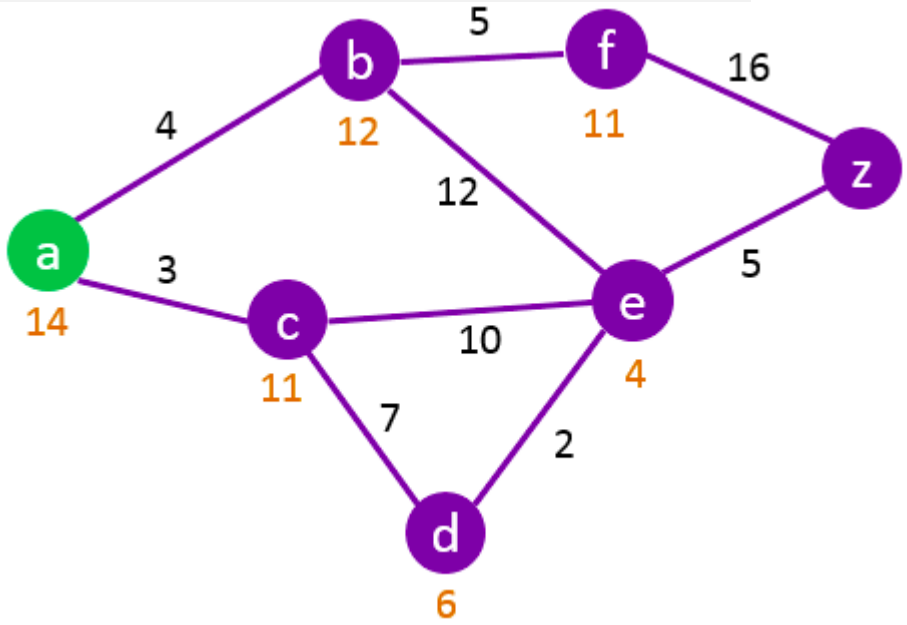
Exemple : Chercher le minimal chemin de **a** à **z**



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A					
B					
C					
D					
E					
F					
Z					

Algorithme A*

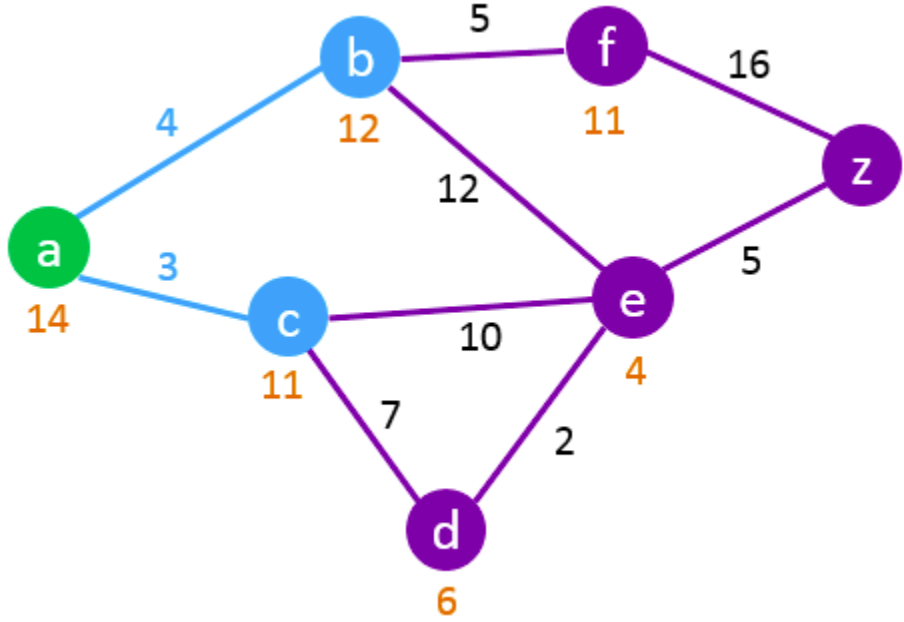
Etape 1



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	Open	0	14	14	
B		∞	12		
C		∞	11		
D		∞	6		
E		∞	4		
F		∞	11		
Z		∞	0		

Algorithme A*

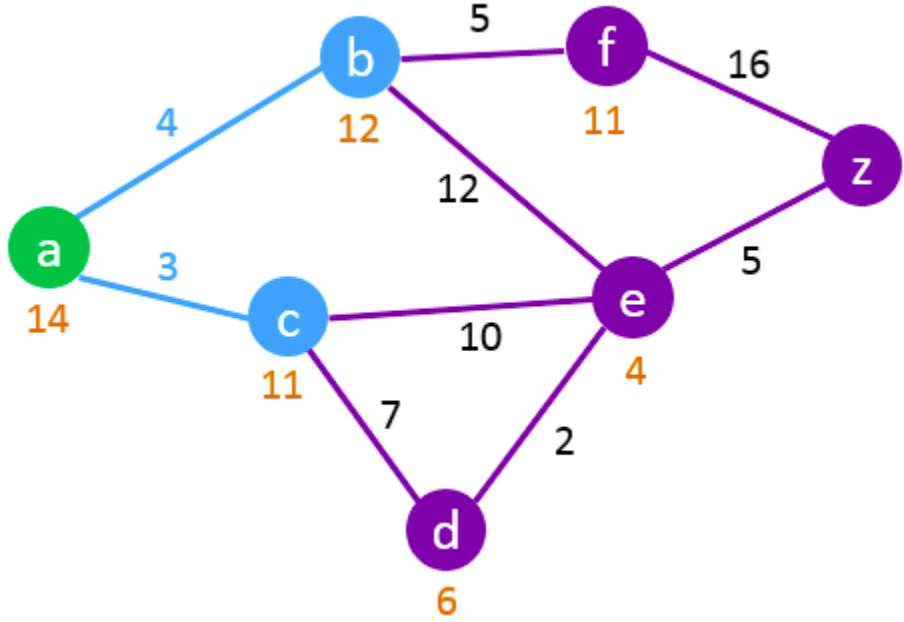
Etape 3



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	Open	0	14	14	
B		∞ → 4	12		
C		∞ → 3	11		
D		∞	6		
E		∞	4		
F		∞	11		
Z		∞	0		

Algorithme A*

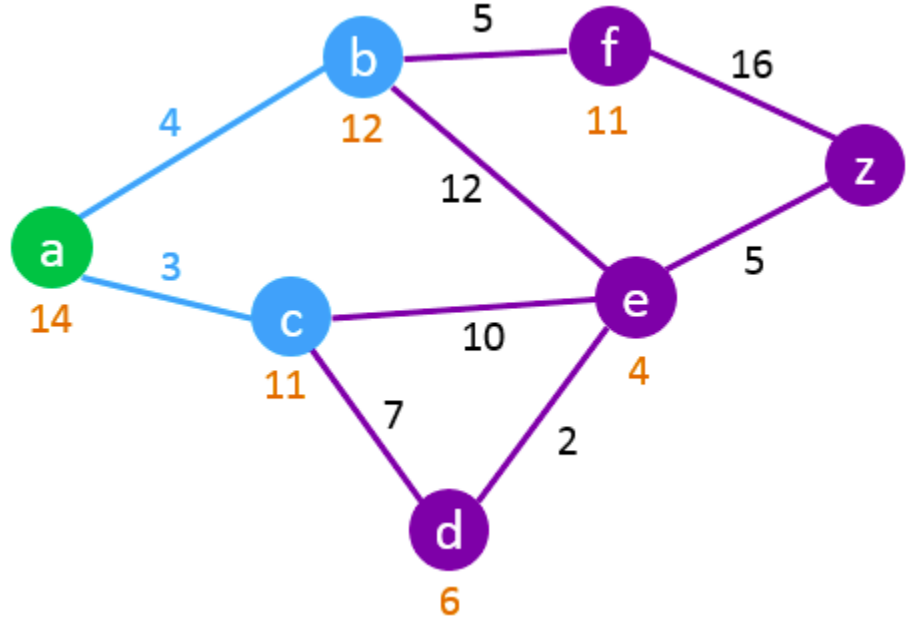
Etape 3



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	Open	0	14	14	
B		∞ → 4	12	16	A
C		∞ → 3	11	14	A
D		∞	6		
E		∞	4		
F		∞	11		
Z		∞	0		

Algorithme A*

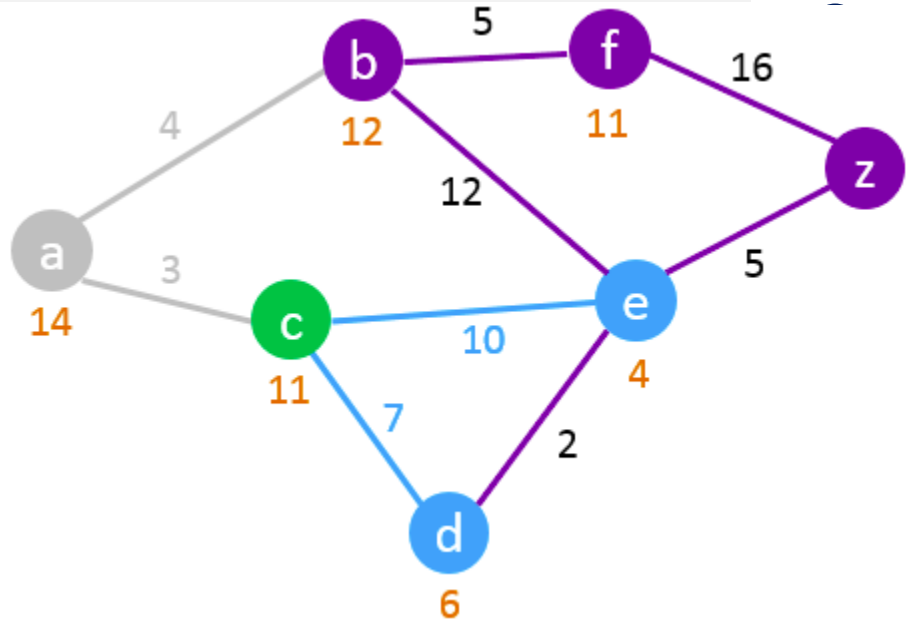
Etape 4



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	Closed	0	14	14	
B	Open	∞ → 4	12	16	A
C	Open	∞ → 3	11	14	A
D		∞	6		
E		∞	4		
F		∞	11		
Z		∞	0		

Algorithme A*

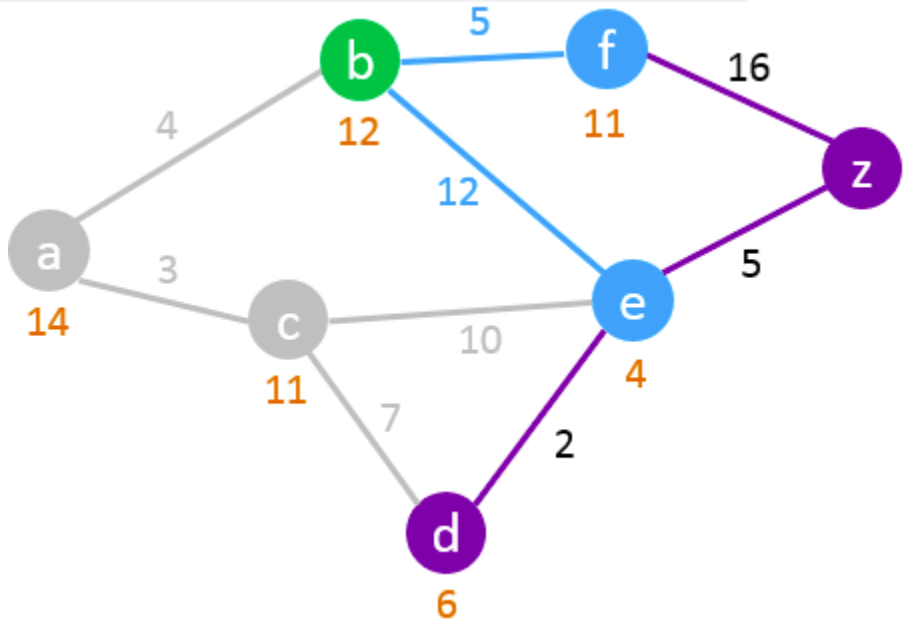
Etape 5



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	**Closed**	0	14	14	
B	Open	4	12	16	A
C	**Closed**	3	11	14	A
D	Open	∞ → 10	6	16	C
E	Open	∞ → 13	4	17	C
F		∞	11		
Z		∞	0		

Algorithme A*

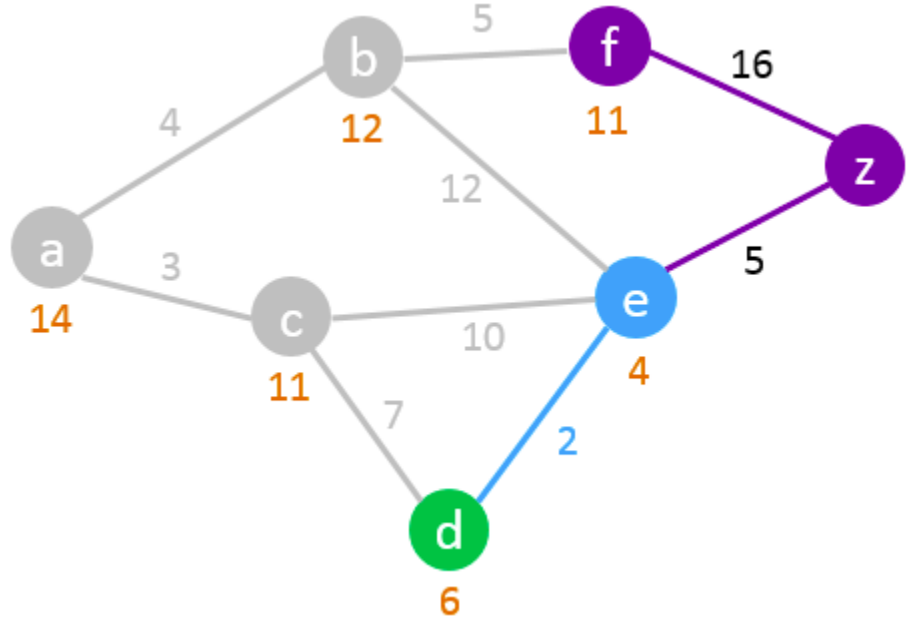
Etape 6



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	Closed	0	14	14	
B	Closed	4	12	16	A
C	Closed	3	11	14	A
D	Open	10	6	16	C
E	Open	13	4	17	C
F	Open	9	11	20	B
Z		∞	0		

Algorithme A*

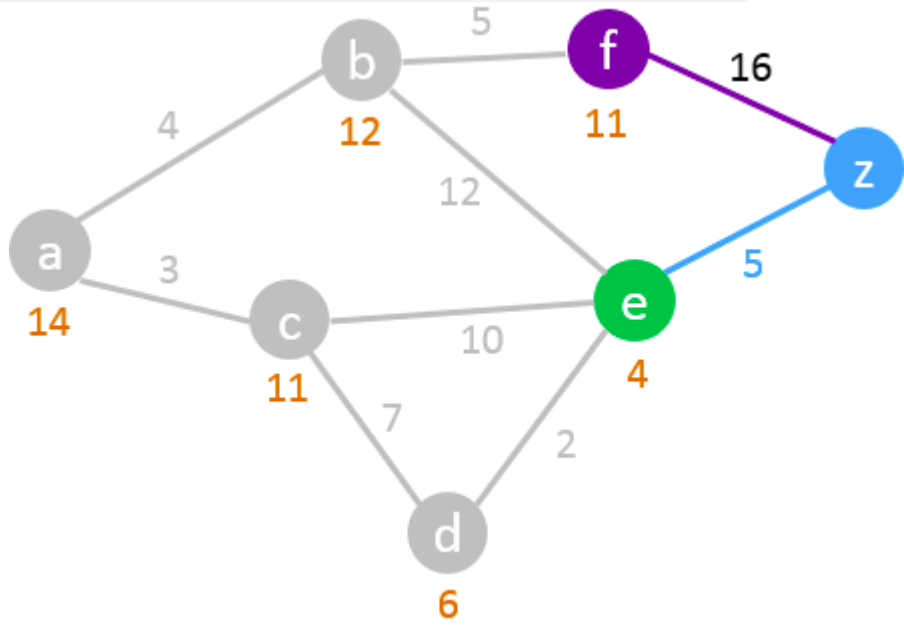
Etape 7



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	**Closed**	0	14	14	
B	**Closed**	4	12	16	A
C	**Closed**	3	11	14	A
D	**Closed**	10	6	16	C
E	Open	13 → 12	4	17	D
F	Open	9	11	20	B
Z		∞	0		

Algorithme A*

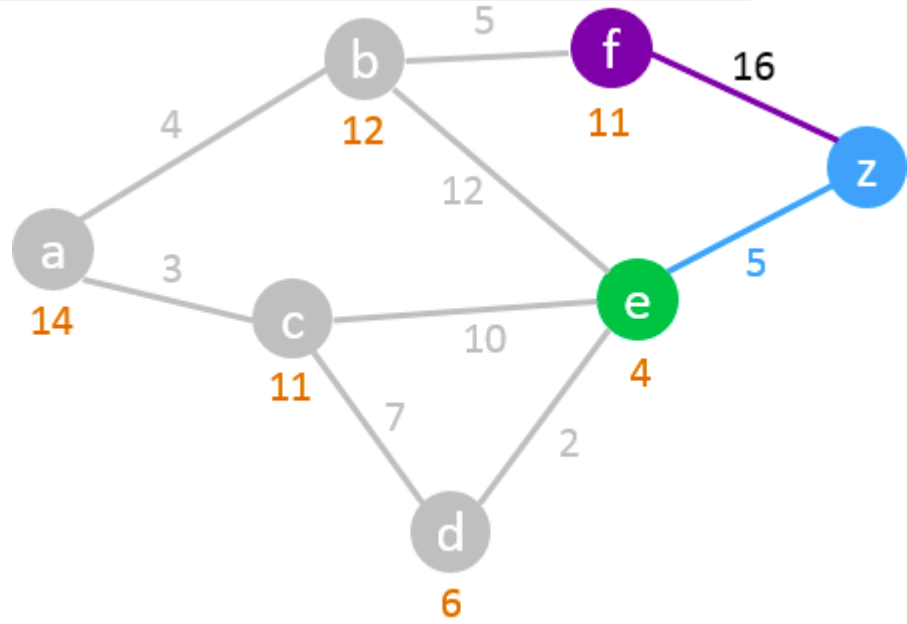
Etape 8



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	Closed	0	14	14	
B	Closed	4	12	16	A
C	Closed	3	11	14	A
D	Closed	10	6	16	C
E	Closed	12	4	17	D
F	Open	9	11	20	B
Z	Open	∞ → 17	0	17	E

Algorithme A*

Etape 9



Nœud n	Statut	g(n)	h(n)	f(n)	Précédent
A	Closed	0	14	14	
B	Closed	4	12	16	A
C	Closed	3	11	14	A
D	Closed	10	6	16	C
E	Closed	12	4	17	D
F	Open	9	11	20	B
Z	Open	17	0	17	E

Coût : 17
Chemin : A-C-D-E-Z

Algorithme A*

Pseudo code:

Entrées:

Sommet source (S)

Sommet destination (D)

Initialisation

Liste des sommets à explorer (**open**) : sommet source S

Liste des sommets visités (**closed**) : vide

Tant que (la liste open est non vide) et (D n'est pas dans open) Faire

+ Récupérer le sommet X de coût total F minimum.

+ Ajouter X à la liste **closed**

+ Ajouter les successeurs de X (non déjà visités) à la liste **open** en évaluant leur coût total f et en identifiant leur prédécesseur.

+ Si (un successeur est déjà présent dans **open**) et (nouveau coût est inférieur à l'ancien) Alors

Changer son coût total

Changer son prédécesseur

FinSi

FinFaire

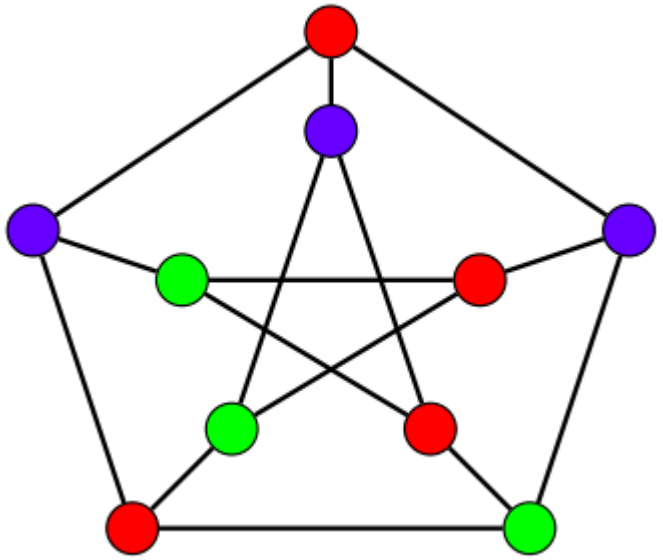
Algorithme A* : Propriétés

Le choix de l'heuristique d'estimation est crucial :

- Sur-estimation → Risque de ne pas envoyer le chemin optimal
- Sous-estimation → la complexité de l'algorithme augmente
- Heuristique nulle → algorithme de Dijkstra

Algorithme Coloration de graphe

Colorer un réseau signifie attribuer une couleur à chacun de ses nœuds de manière que deux nœuds voisins soient de couleur différente.



Une coloration du graphe de Petersen avec 3 couleurs

Algorithme Coloration de graphe

Applications :

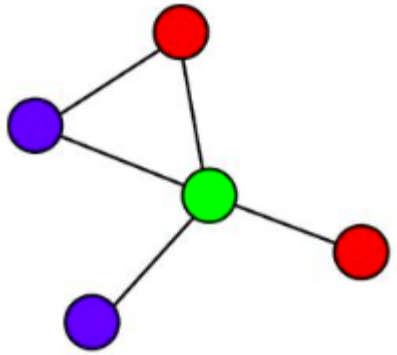
La coloration de sommet a de nombreuse applications pratiques. Par exemple :

- Résoudre des problèmes d'organisation (Session minimale à programmer).
- Allocation de fréquences dans les réseaux GSM
- Coloration de carte géographique
- ….

Algorithme Coloration de graphe

Algorithmes :

Quel algorithme permet de colorier un graphe ?



Deux algorithmes principaux :

1. Algorithme naïf (glouton), non optimal mais rapide
2. Welsh et Powell

Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

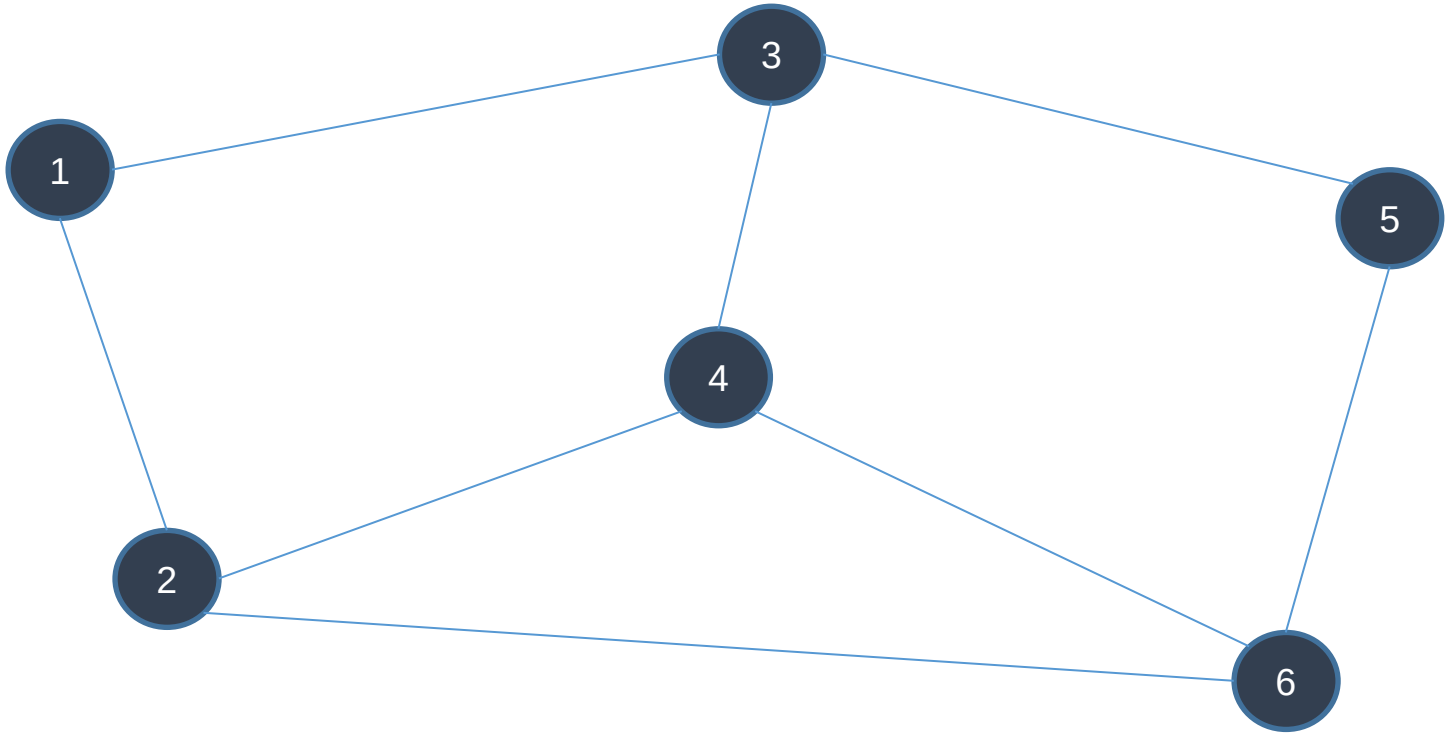
Algorithm 1: Algorithme glouton

tant que *il existe un noeud non-coloré v* **faire**

    Colorié *v* avec la couleur minimale qui n'entre pas en conflit avec ces voisins

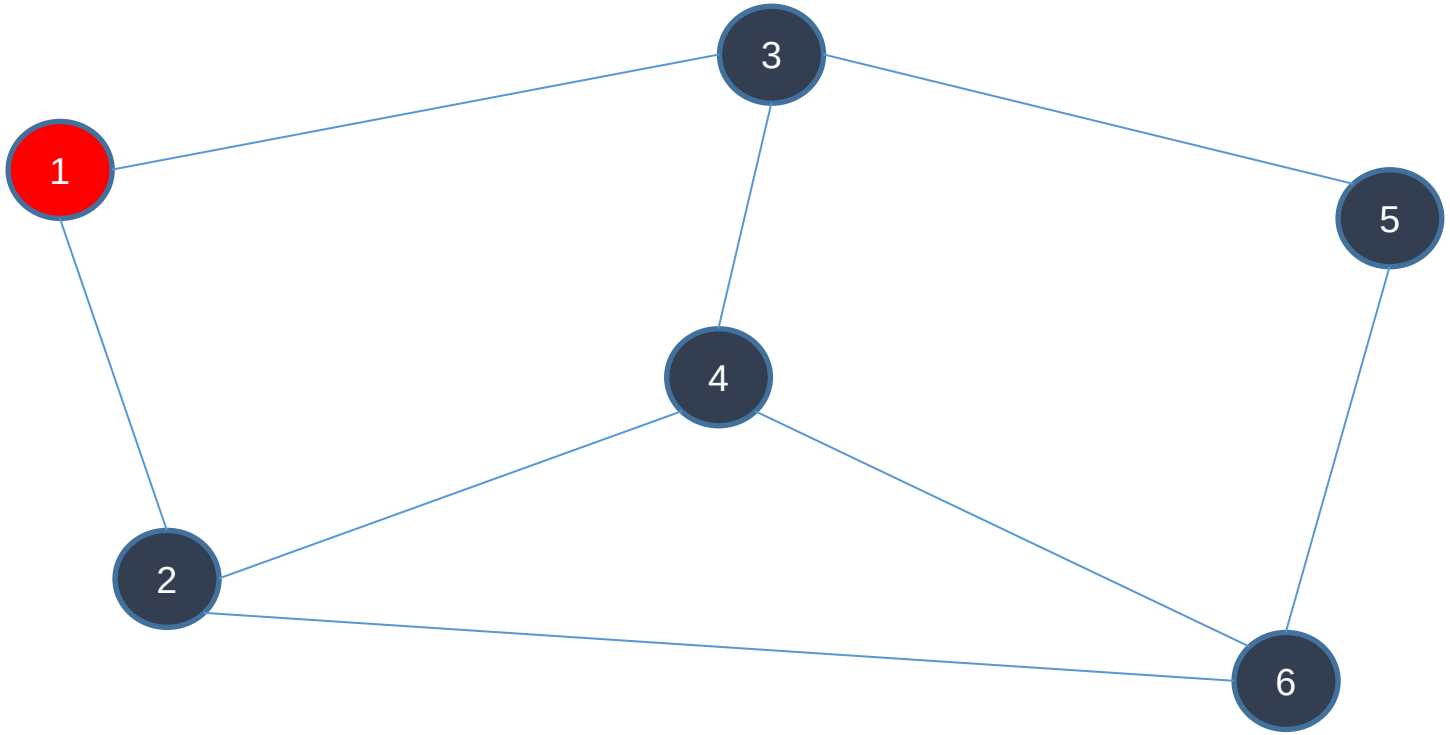
Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle



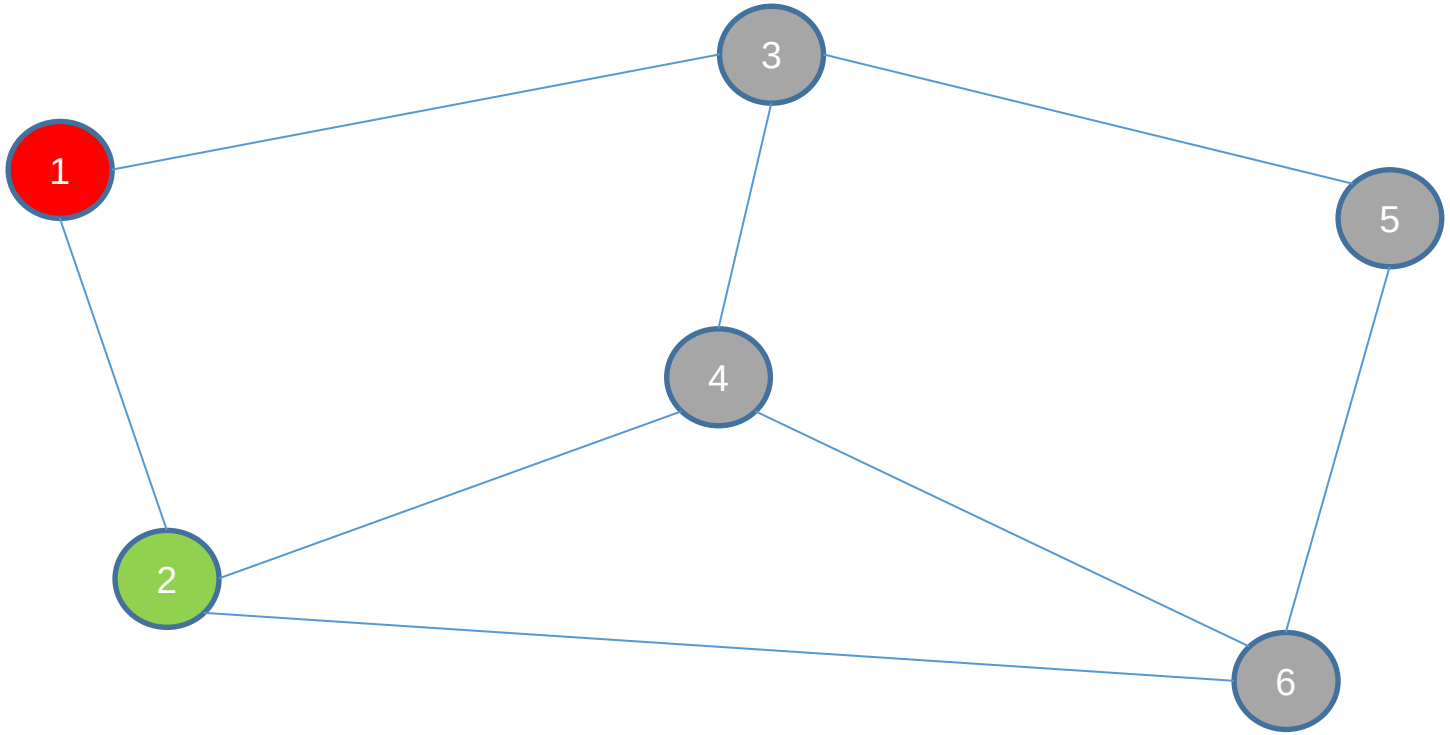
Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle



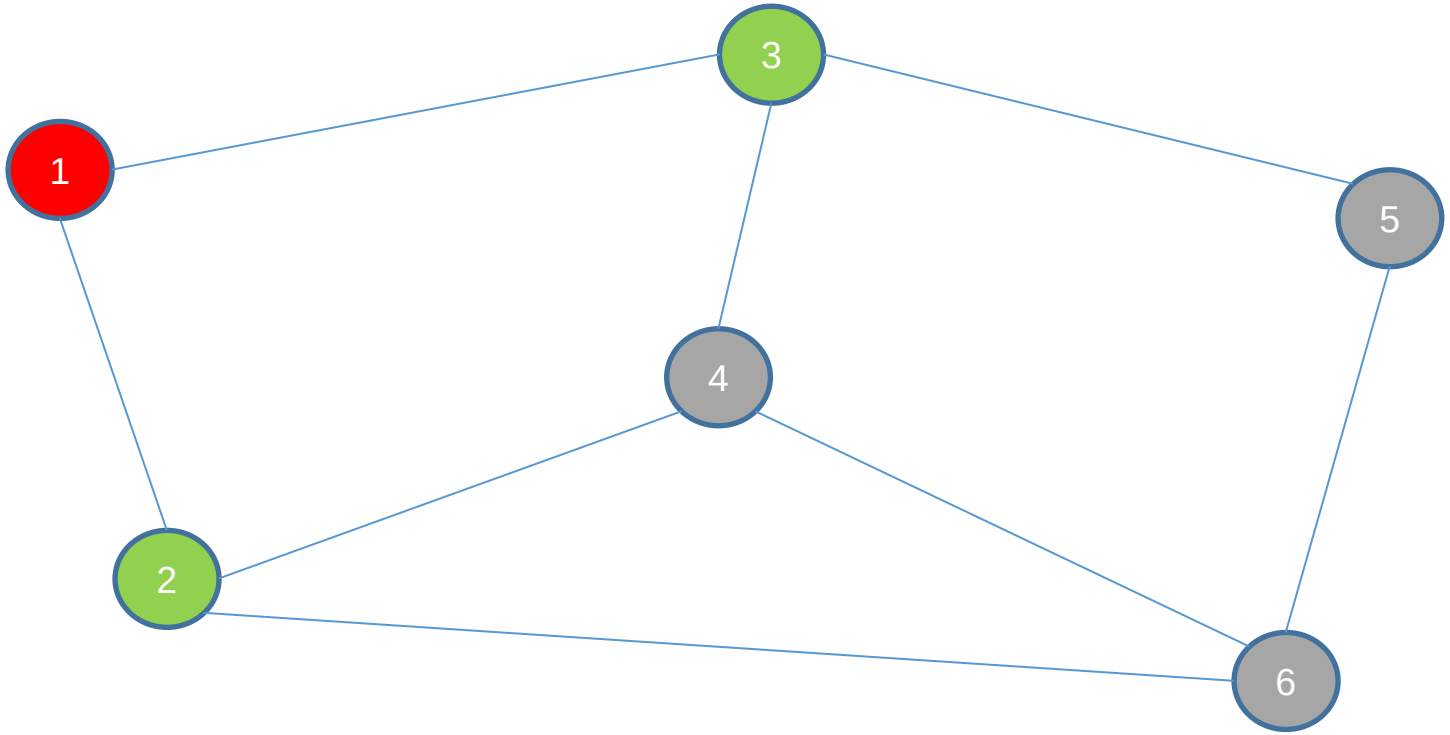
Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle



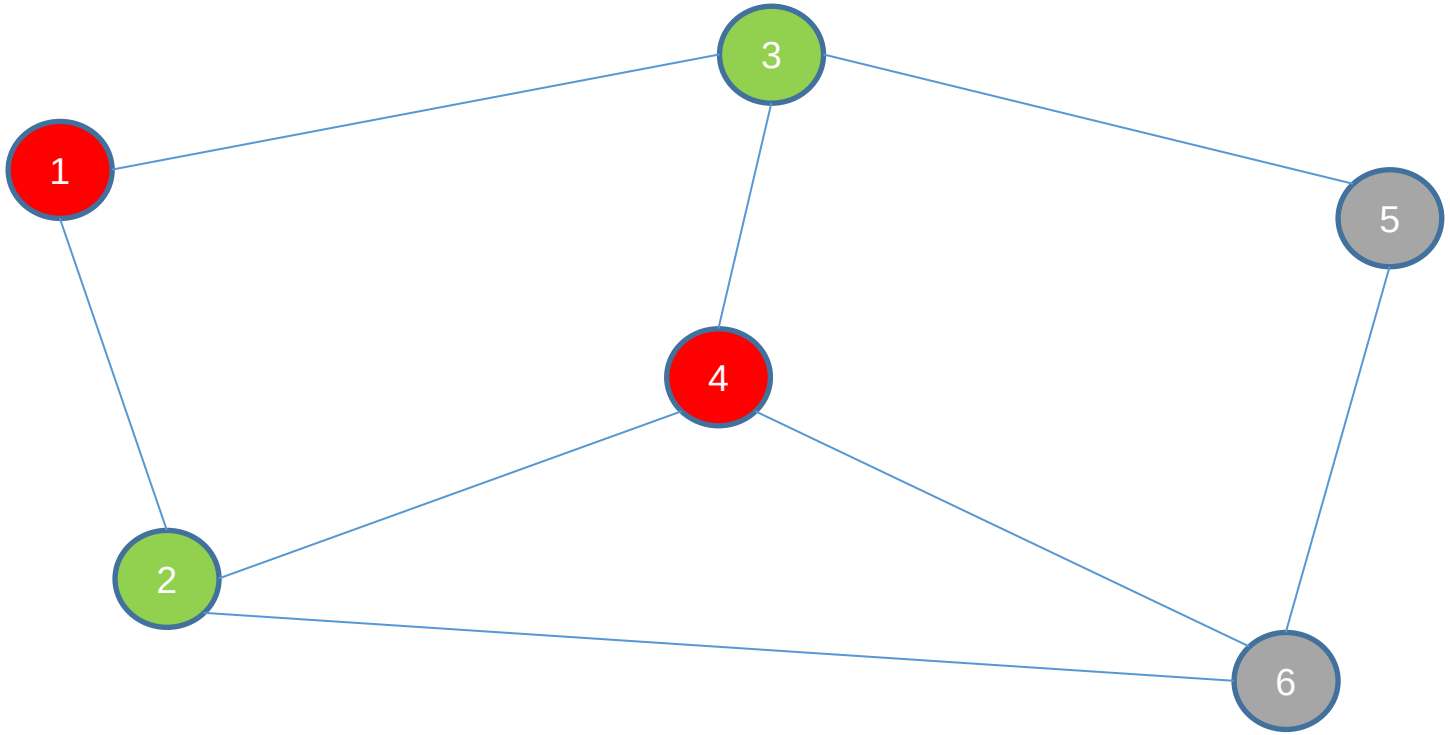
Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle



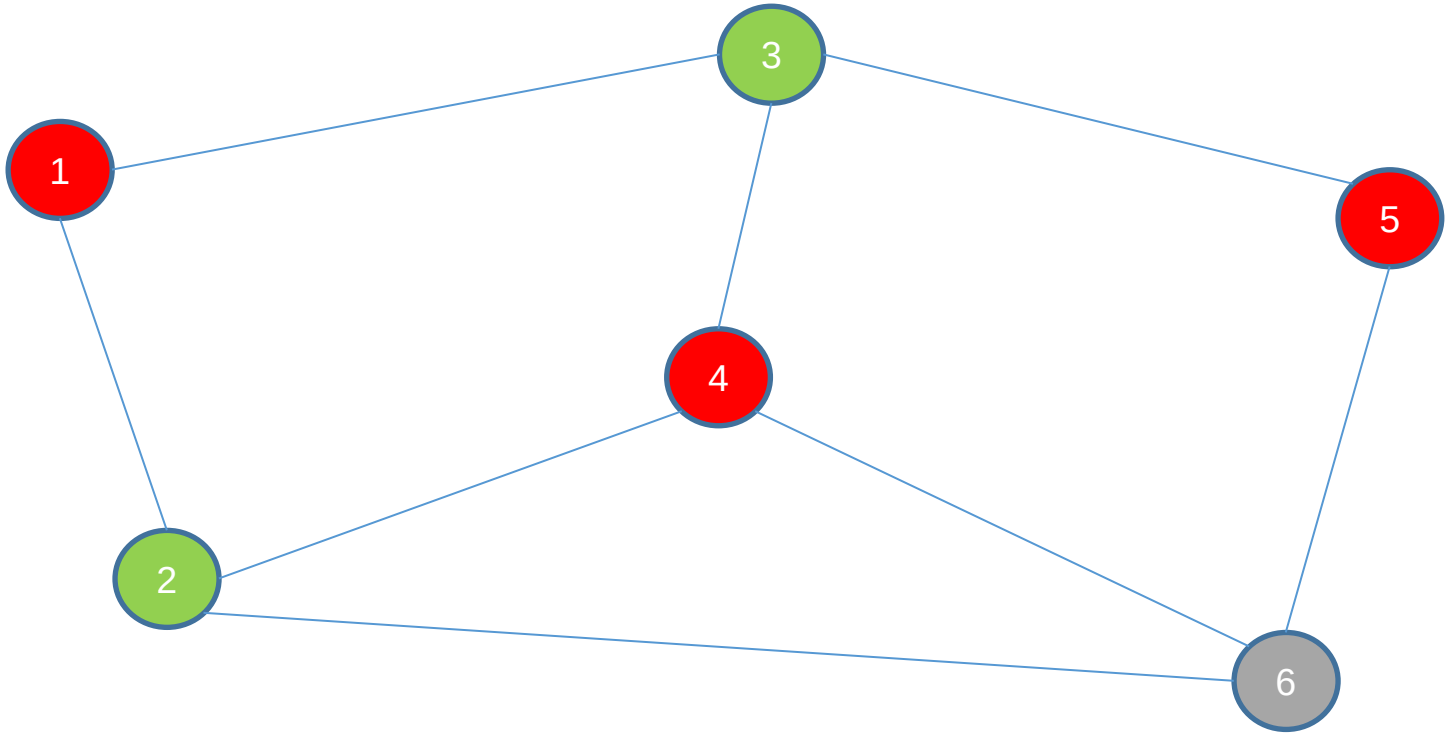
Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle



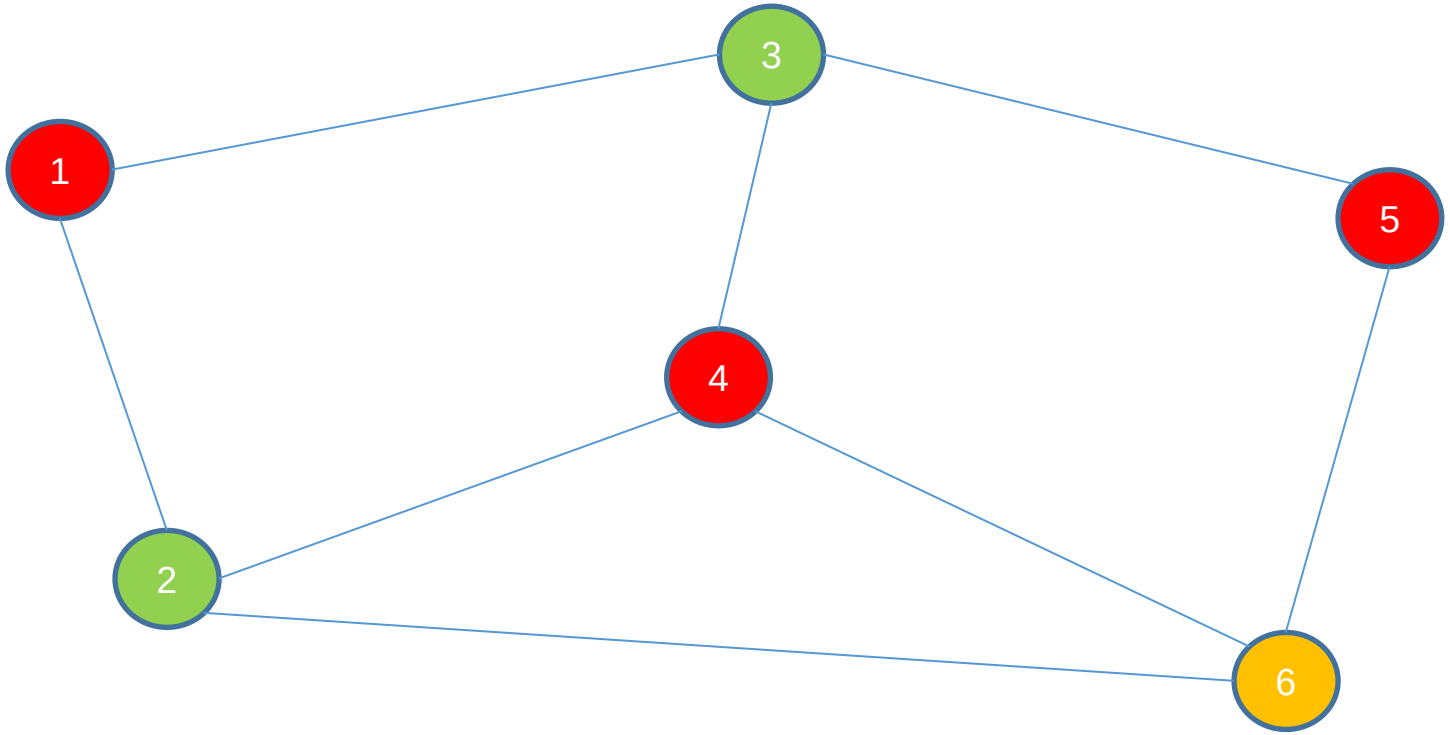
Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle



Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle



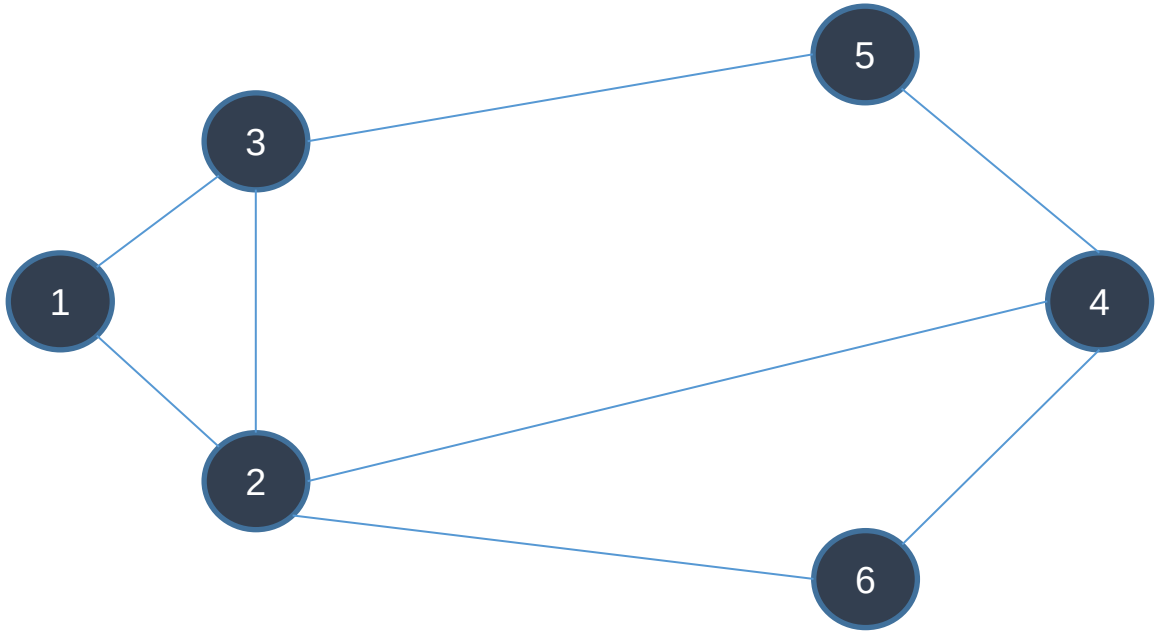
Nombre chromatique : 3

Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 1:

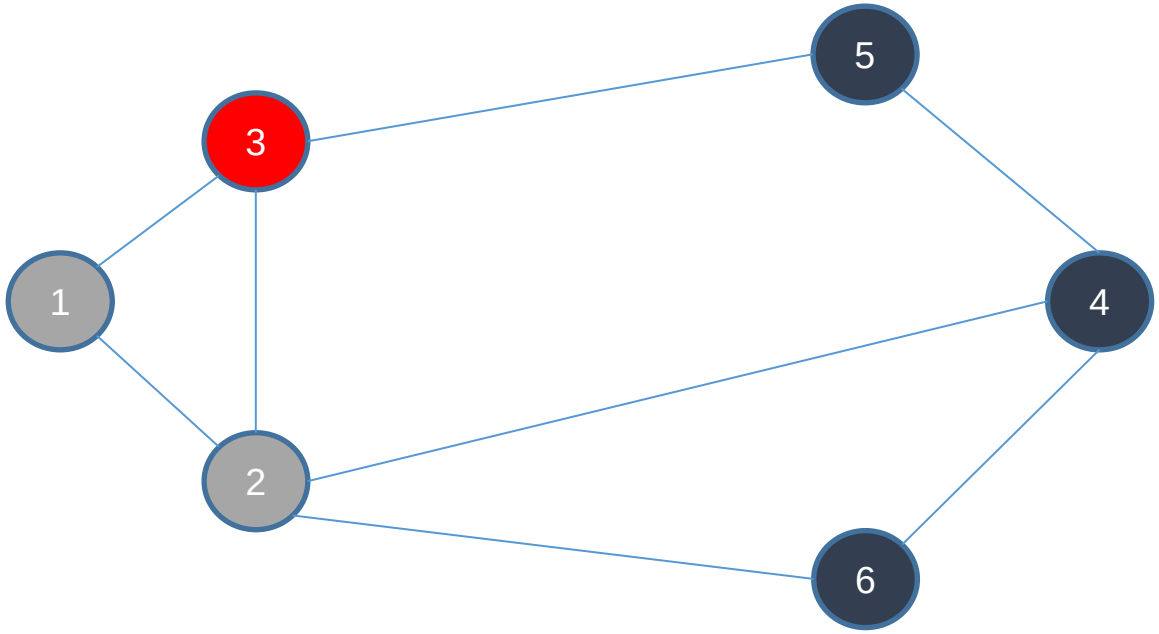


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 1:

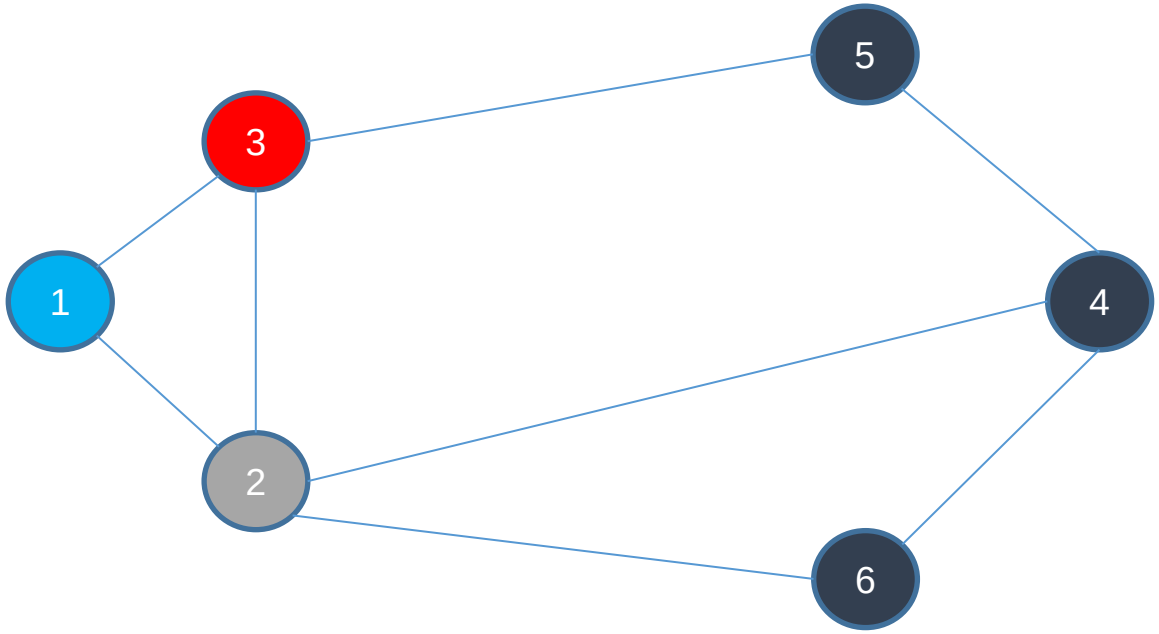


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 1:

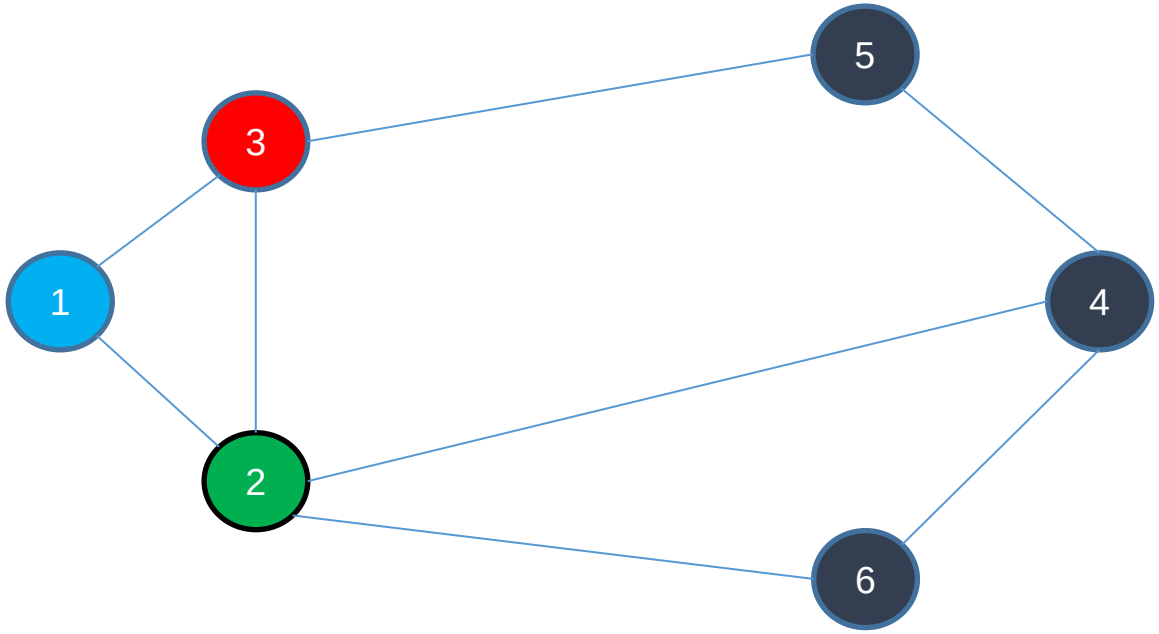


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 1:

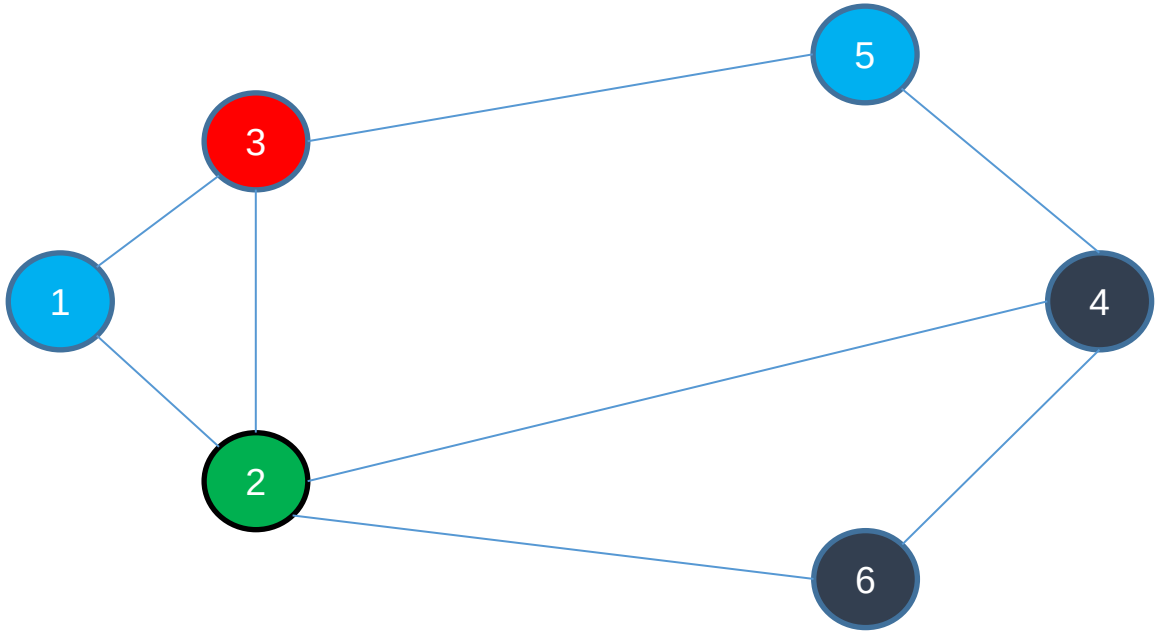


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 1:

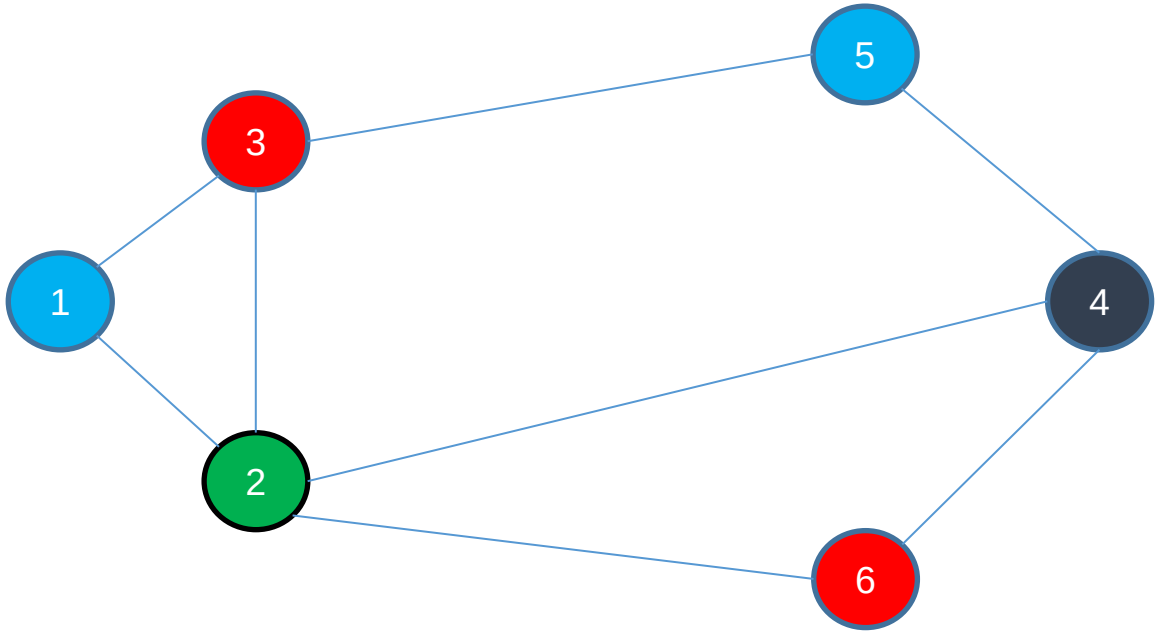


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 1:

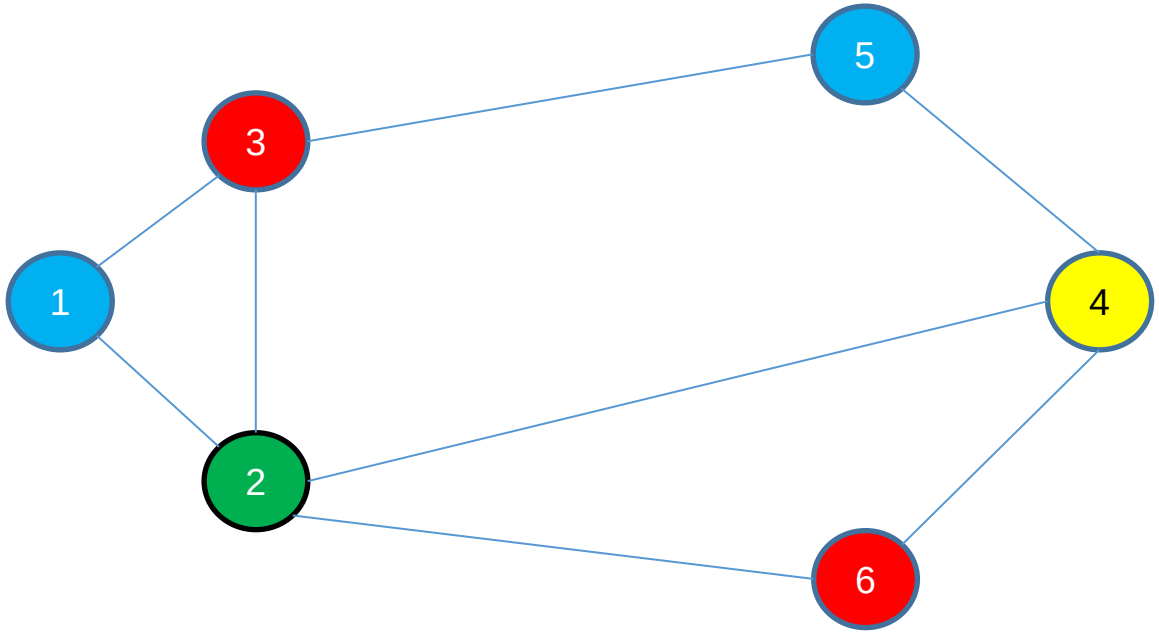


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 1:



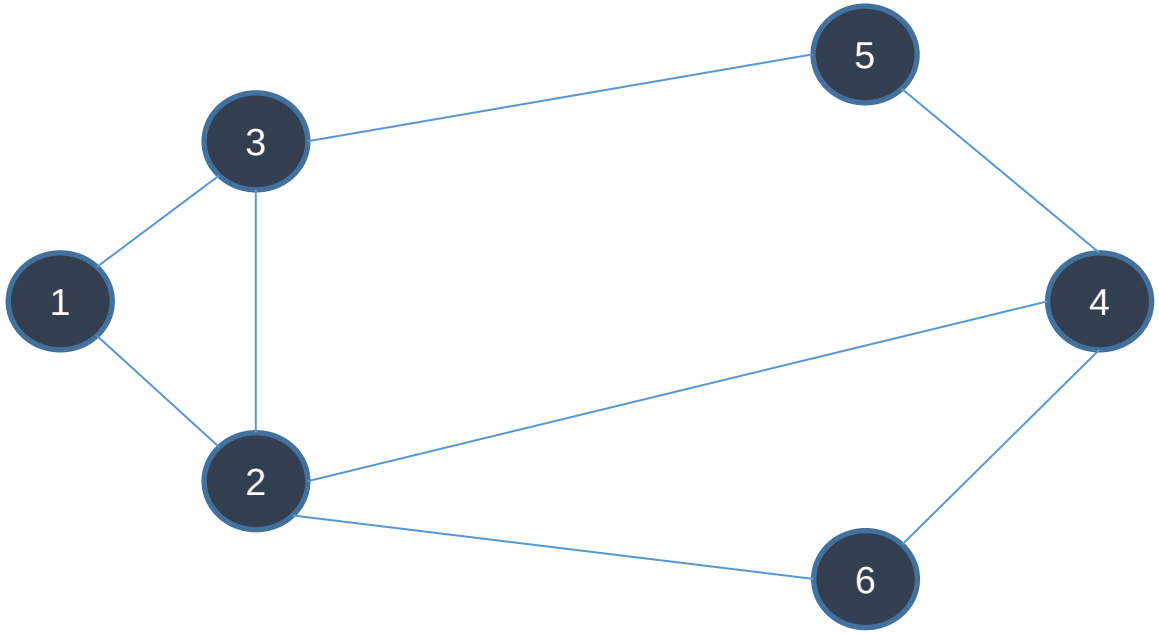
4 couleurs

Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 2:

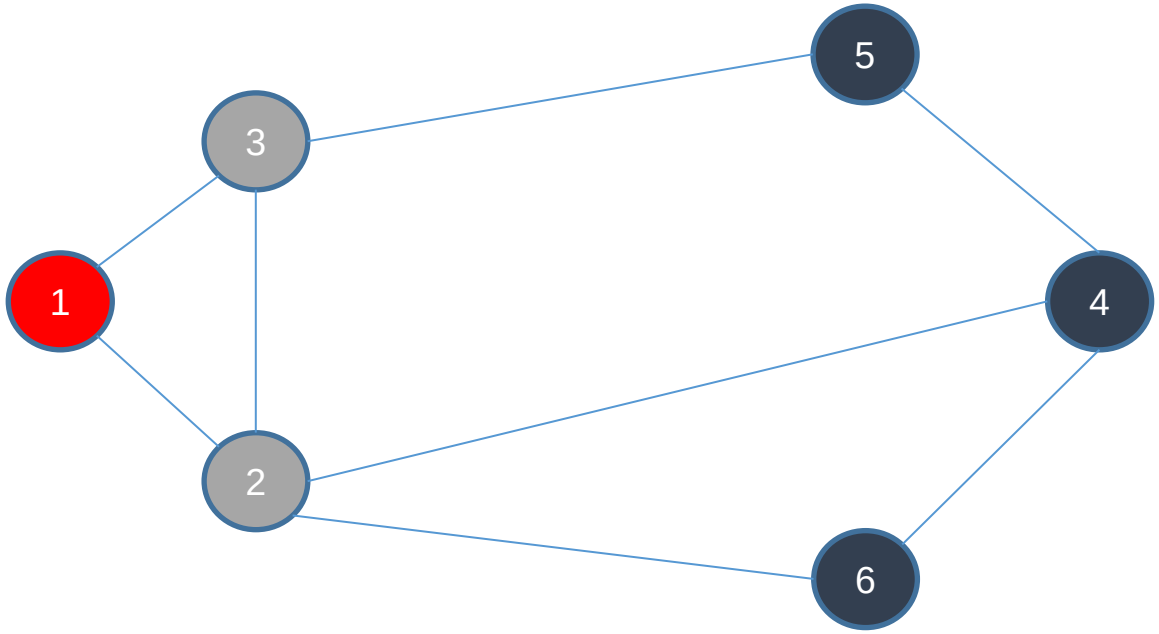


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 2:

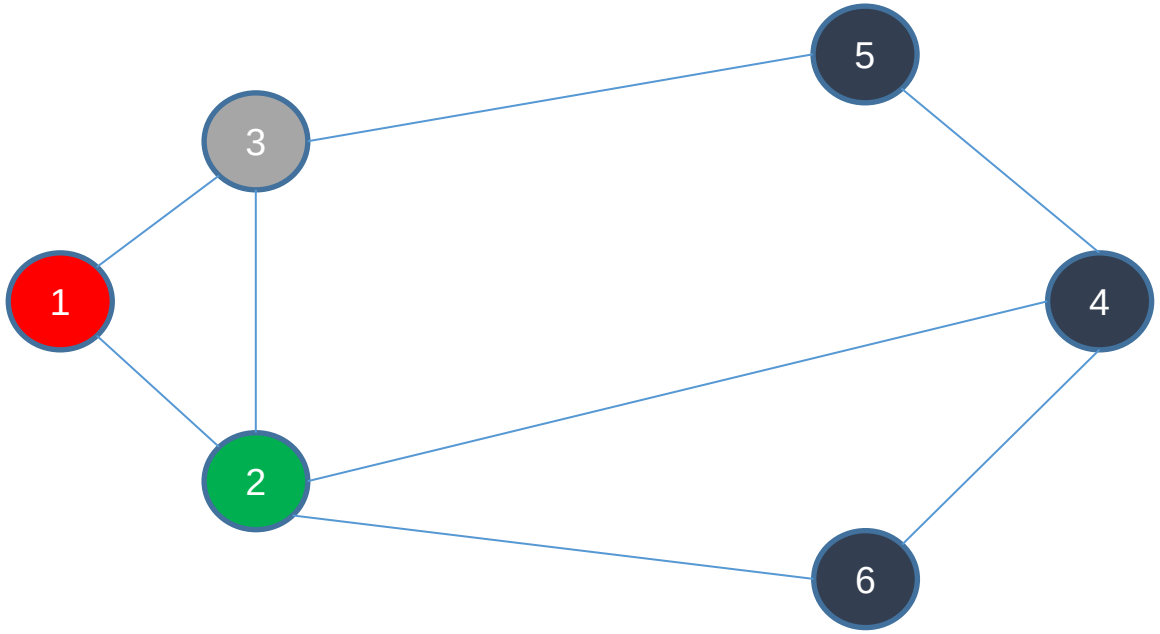


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 2:

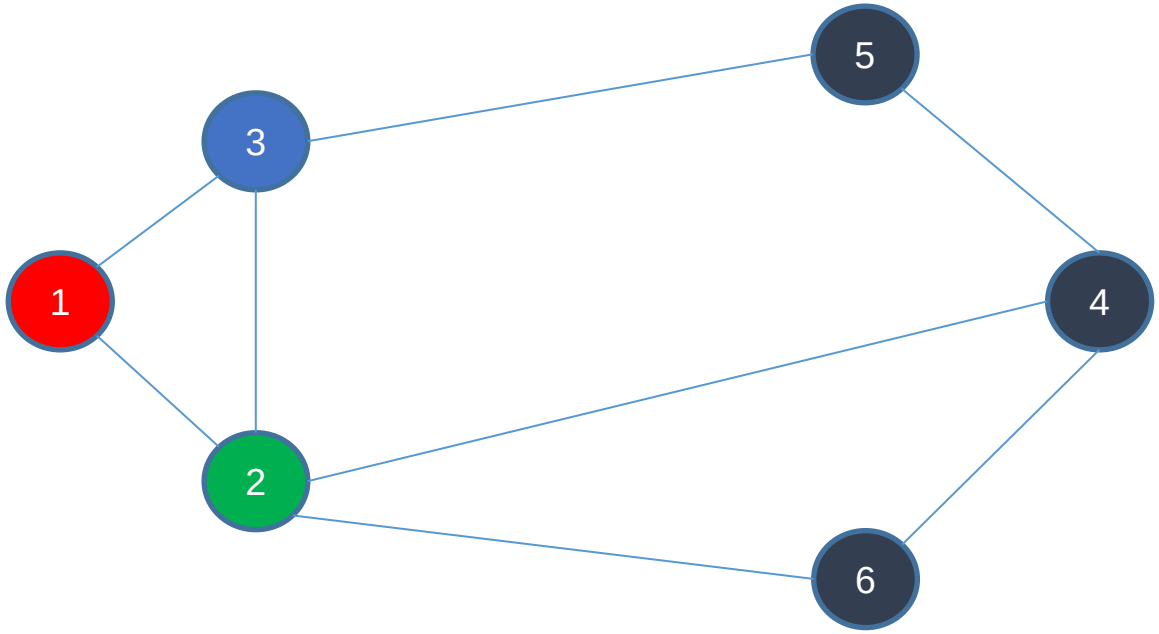


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 2:

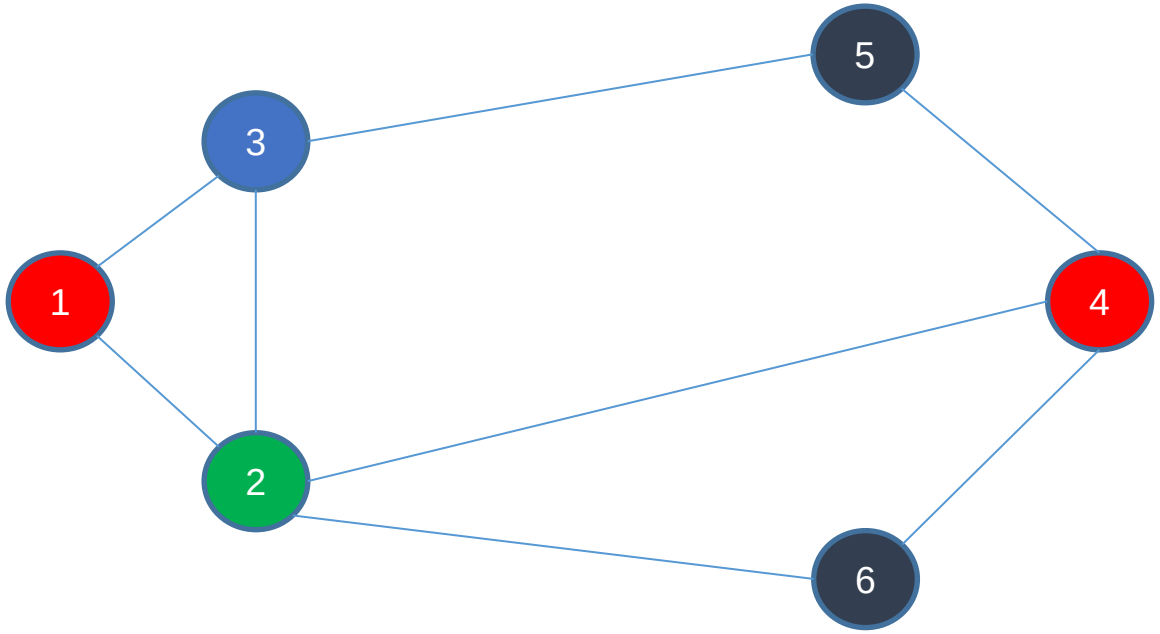


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 2:

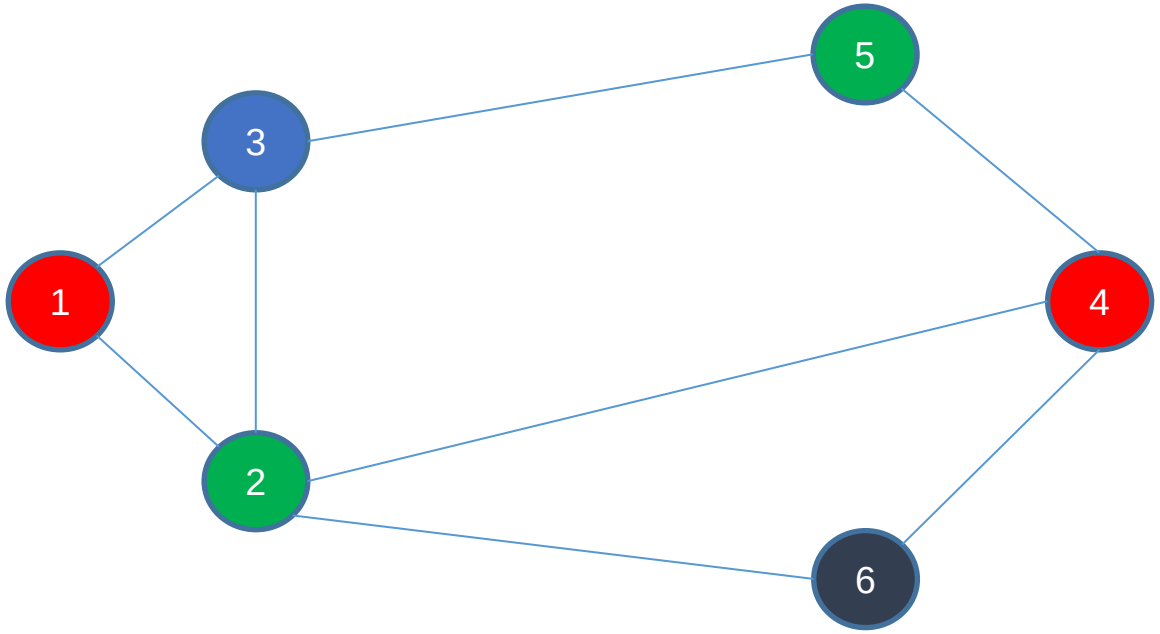


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 2:

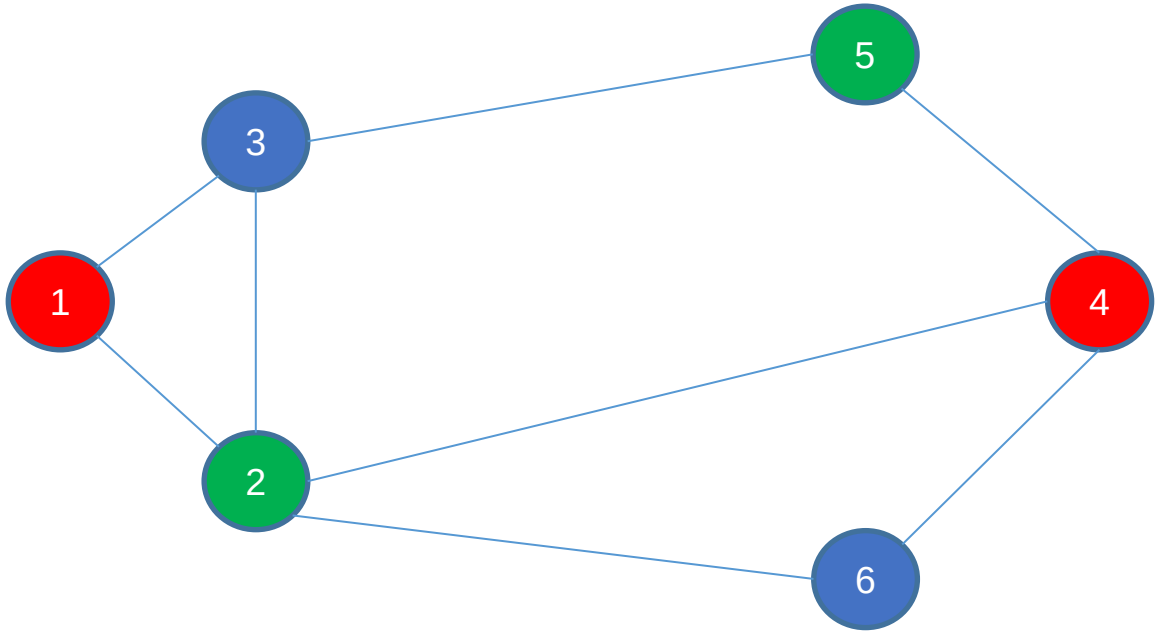


Algorithme Coloration de graphe

Algorithme 1 : Coloration séquentielle

Problème : Le nombre de couleur dépend du sommet de départ, de la manière de parcourt, …

Exemple 2:



3 couleurs

Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell :

L’algorithme de Welsh & Powell consiste à colorer séquentiellement le graphe en visitant les sommets par ordre de degré décroissant. L’idée est que les sommets ayant beaucoup de voisins seront plus difficiles à colorer, et donc il faut les colorer en premier.

Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell :

1.Repérer le degré de chaque sommet.

2.Ranger les sommets par ordre de degrés décroissants. (dans certains cas plusieurs possibilités)

3.Attribuer au premier sommet (A) de la liste une couleur.

4.Suivre la liste en attribuant la même couleur au premier sommet (B) qui ne soit pas adjacent à (A).

5.Suivre (si possible) la liste jusqu'au prochain sommet (C) qui ne soit adjacent ni à A ni à B.

6.Continuer jusqu'à ce que la liste soit finie.

7.Prendre une deuxième couleur pour le premier sommet (D) non encore colorié de la liste.

8.Répéter les opérations 4 à 6.

9.Continuer jusqu'à avoir coloré tous les sommets.

Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell :

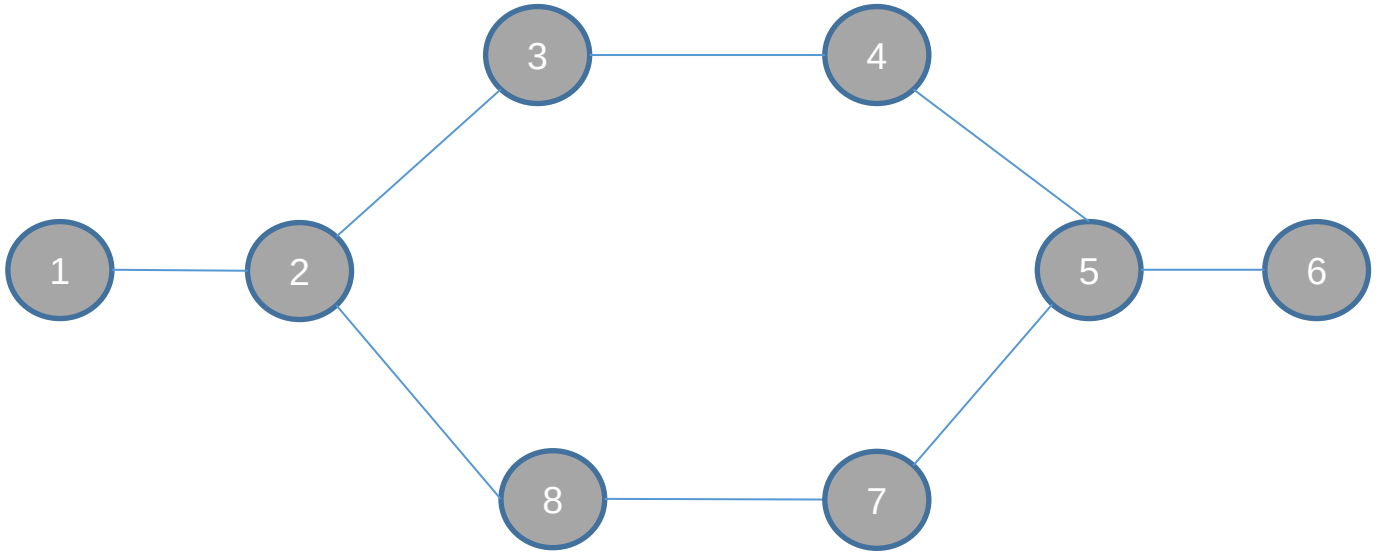
On note L la liste des sommets si classes suivant l'ordre décroissant de leur degré :
(s1) d(s2) d(s3) ::: d(sn).
Initialisation :
L : liste des sommets dans l'ordre décroissant du degré
couleur = 0
Tant que L ≠ ∅ ; faire
    couleur=couleur+1
    couleur(s) = couleur
    V = voisins(s)
    Pour tout t dans L faire
        Si t ∉ V
            couleur(t)=couleur
            V = V ∪ t
        Fin si
    Fin faire
    Retirer les sommets colores de L
Fin faire

Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell : Contre exemple

Liste sommets triés par degrés décroissants :

2
5
4
3
7
8
6
1

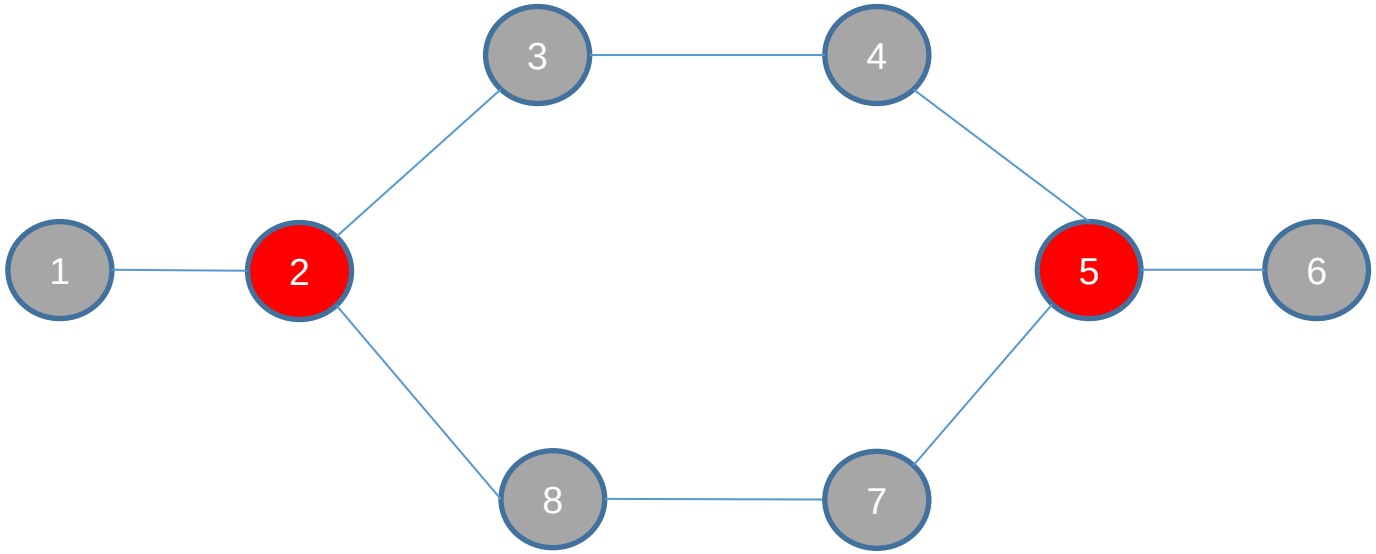


Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell : Contre exemple

Liste sommets triés par degrés décroissants :

4
3
7
8
6
1



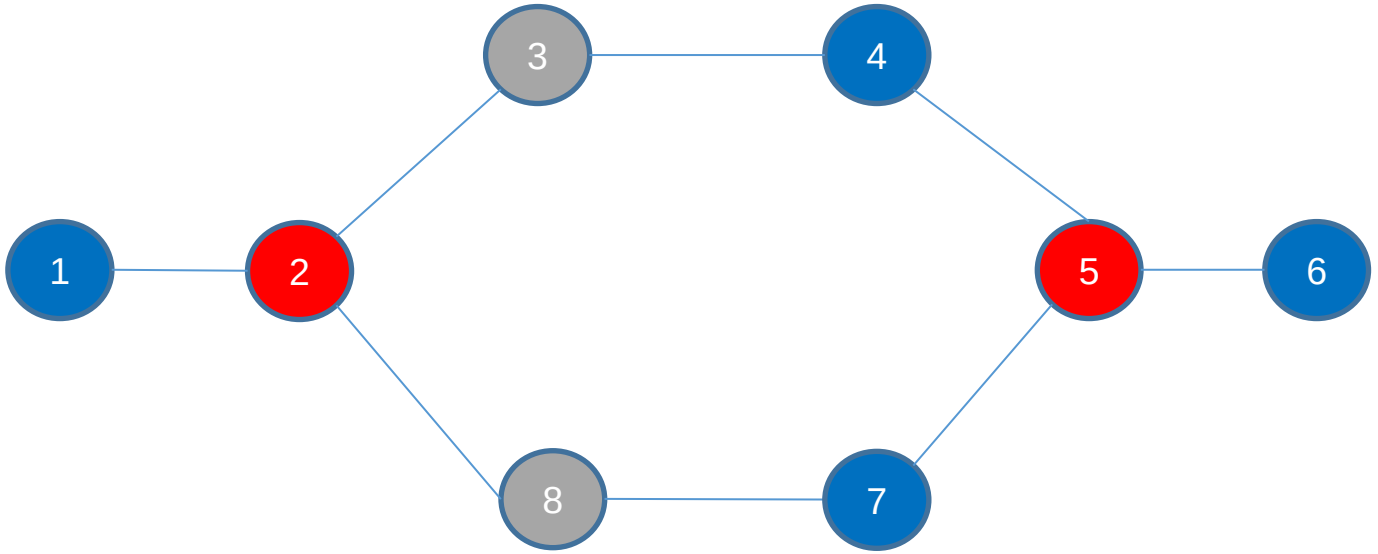
Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell : Contre exemple

Liste sommets triés par degrés décroissants :

3

8



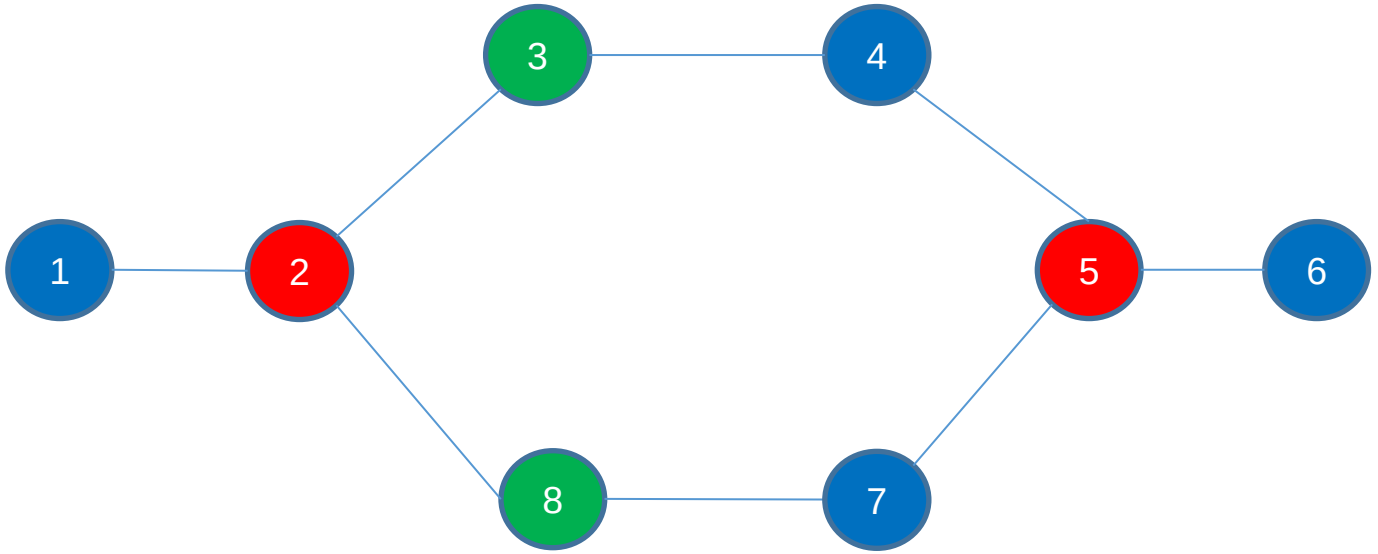
Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell : Contre exemple

Liste sommets triés par degrés décroissants :

3

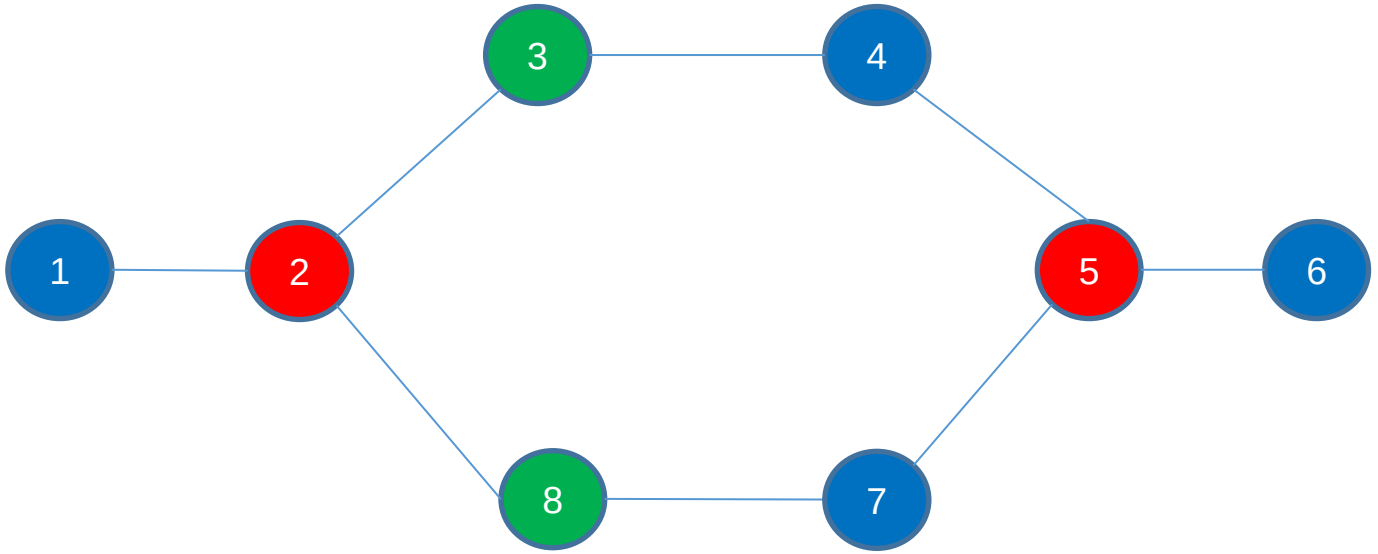
8



Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell: Contre exemple

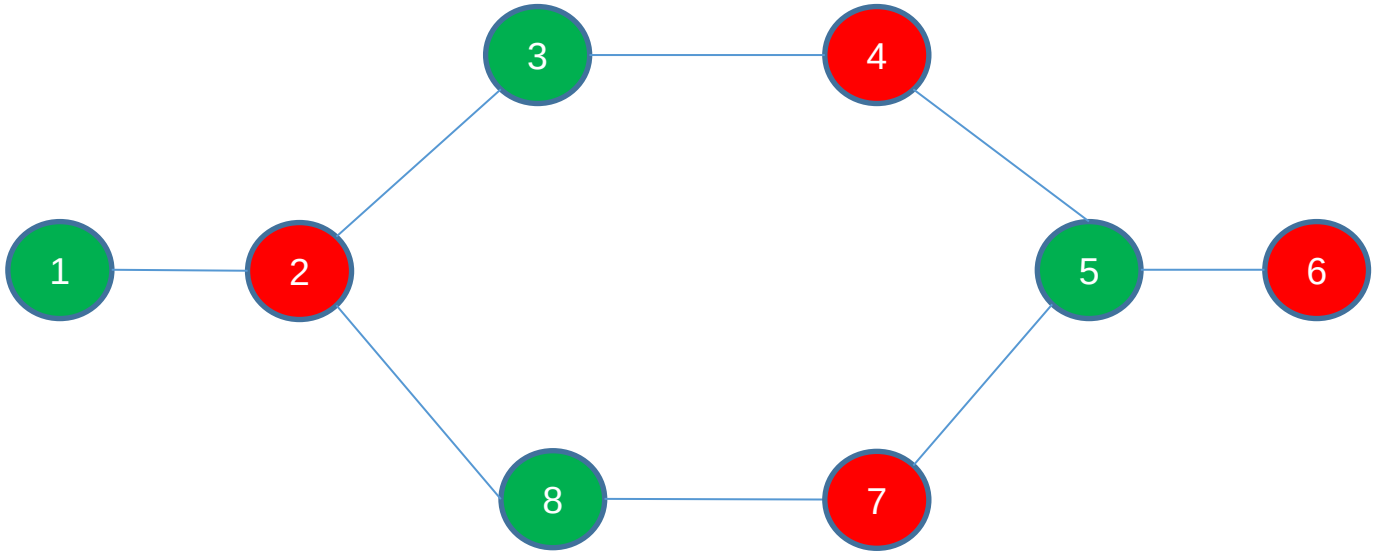
Nombre chromatique : 3



Algorithme Coloration de graphe

Algorithme 2: de Welsh et Powell: Contre exemple

Alors que il existe une autre solution :
Nombre chromatique : 2



Les tables da hachage

HESTIM ENGINEERING & BUSINESS SCHOOL

Introduction aux Tables de Hachage

Définition

Une table de hachage est une structure de données fondamentale qui permet de stocker des paires clé-valeur et d'accéder rapidement aux valeurs associées à une clé donnée.

Concept

Le fonctionnement repose sur le "hachage", qui transforme une clé en un indice pour un tableau, permettant ainsi des opérations d'insertion, de suppression et de recherche très efficaces.

Caractéristiques principales

- Stockage efficace de paires clé-valeur
- Accès rapide aux données via la clé
- Complexité temporelle moyenne O(1) pour les opérations
- Implémentation possible en C avec des structures de données personnalisées
- Applications variées dans la science informatique

Principes de Base

Les tables de hachage permettent d'effectuer les trois opérations fondamentales avec une complexité temporelle moyenne de O(1) :



Insertion

`O(1) moyenne`

Ajoute une nouvelle paire clé-valeur à la table. Le hachage transforme la clé en un indice, où la paire est stockée directement.

En cas de collision, l'opération peut dans le pire des cas.



Suppression

`O(1) moyenne`

Retire une paire clé-valeur existante de la table en utilisant sa clé. Le hachage permet de localiser rapidement l'élément à supprimer.

La suppression efficace nécessite une appropriée des collisions.



Recherche

`O(1) moyenne`

Récupère la valeur associée à une clé donnée. Le hachage transforme la clé en un indice, permettant un accès direct à la valeur.

La recherche est très rapide, mais dépend qualité de la fonction de hachage.

HESTIM ENGINEERING & BUSINESS SCHOOL

Structure de Données en C

KeyValuePair

Structure pour une paire clé-valeur

```
struct KeyValuePair {
  char key[50]; // Clé de type chaîne
  int value; // Valeur de type entier
  struct KeyValuePair* next; // Pour le chaînage
};
```

Node

Structure pour un nœud de liste chaînée

```
struct Node {
  KeyValuePair* pair; // Pointeur vers paire clé-valeur
  struct Node* next; // Pointeur vers le nœud suivant
};
```

HashTable

Structure pour la table de hachage

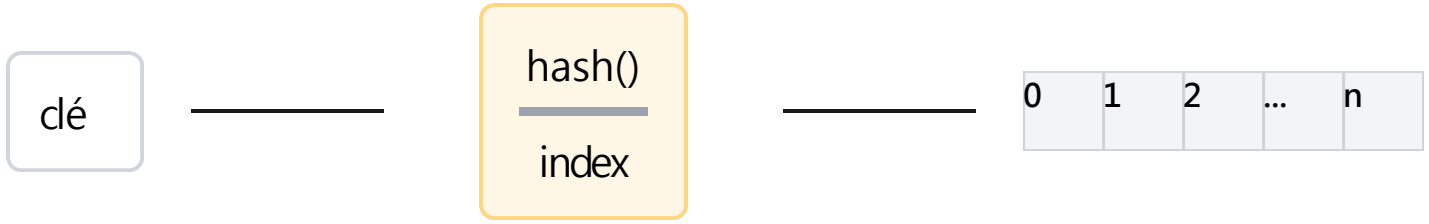
```
struct HashTable {
  int size; // Taille du tableau
  KeyValuePair** table; // Tableau de pointeurs
};
```

- Chaque "seau" (bucket) de la table de hachage pointe vers une liste chaînée de nœuds
- Les nœuds contiennent des paires clé-valeur, permettant de gérer les collisions

La Fonction de Hachage - Rôle

Rôle principal

La fonction de hachage transforme une clé d'entrée en un index numérique de taille fixe (code de hachage) qui correspond à une position dans le tableau sous-jacent de la table de hachage.



Importance

- Permet un accès rapide aux données stockées
- Transforme des clés de taille variable en indices fixes

Résultat

- Position dans le tableau pour stocker ou récupérer la valeur
- Fondement de la performance des tables de hachage

HESTIM ENGINEERING & BUSINESS SCHOOL

Qualités d'une Bonne Fonction de Hachage

Déterminisme

Pour une même entrée, la fonction doit toujours produire le même code de hachage. Cela garantit la cohérence et la reproductibilité des recherches.

Efficacité

Le calcul du code de hachage doit être rapide. Une fonction de hachage lente annulerait l'avantage de rapidité des tables de hachage.

Uniformité

La fonction doit distribuer les clés de manière aussi uniforme que possible sur toute la plage des indices de la table. Cela minimise la probabilité de collisions.

Sensibilité aux variations

Même une petite modification de la clé d'entrée devrait entraîner un code de hachage significativement différent.

Hachage par Division (Modulo)

Méthode

C'est l'une des méthodes les plus simples et courantes. Elle calcule l'index en prenant le reste de la division de la clé par la taille de la table.

```
index = clé % taille_table
```

Pour de meilleurs résultats, la taille de la table est souvent choisie comme un nombre premier.

Avantages

- Implémentation simple et efficace
- Opération modulo rapide à calculer
- Distribution uniforme des clés

Visualisation

$$\boxed{23} \div \boxed{7} = \boxed{3}$$

$$23 = 7 \times 3 + 2$$

$$\text{hash}(23) = 23 \% 7 = 2$$

Représentation de la table de hachage:

0	1	2	3	4	5	6	7	8	9

Utiliser un nombre premier comme taille de table réduit les collisions et améliore la distribution des clés.

Hachage pour Chaînes de Caractères

Spécialités

Les chaînes de caractères nécessitent des fonctions de hachage plus complexes car elles sont de longueur variable. Les algorithmes doivent traiter chaque caractère comme un coefficient dans un polynôme ou une combinaison additive/multiplicative.

Algorithmes

- **djb2** : Multiplication par 33 et addition du caractère courant
- **sdbm** : Similar à djb2 mais avec un décalage de bits
- **Méthodes polynomiales** : Chaque caractère est traité comme un coefficient dans un polynôme

Visualisation djb2

```
unsigned int djb2(const char* key) {
  unsigned int hash = 5381;
  while (char c = *key++) {
    hash = (hash << 5) + hash + c;
  }
  return hash;
}
```

d	j	b	2

hash = 5381 → hash = (5381 << 5) + + 'd' → hash = (hash << 5) + hash hash + 'j'

HESTIM ENGINEERING & BUSINESS SCHOOL

Implémentation de Fonction de Hachage

Exemple en C

```c
unsigned int hashFunction(const char* key, int tableSize)
{
    unsigned int hash = 0;
    while (*key) {
hash = (hash << 5) + *key++; // Décalage de 5 bits et
addition
}
return hash % tableSize; // Application du modulo
}
```

- **Décalage de bits (« << 5 »)** : Multiplie la valeur par 32 et distribue uniformément les bits.
- **Modulo (« % tableSize »)** : Assure que l'index est dans les limites de la table.

Clé d'entrée	Valeur
key	"exemple"

↓ ↓

Processus	Résultat
hash = (hash << 5) + *key++	valeur_intermédiaire

↓ ↓

Index final	Dans la table
return hash % tableSize	index (0 à tableSize-1)

Gestion des Collisions - Concept

Qu'est-ce qu'une collision?

Une collision se produit lorsque deux clés différentes génèrent le même indice dans la table de hachage.

Impact sur les performances

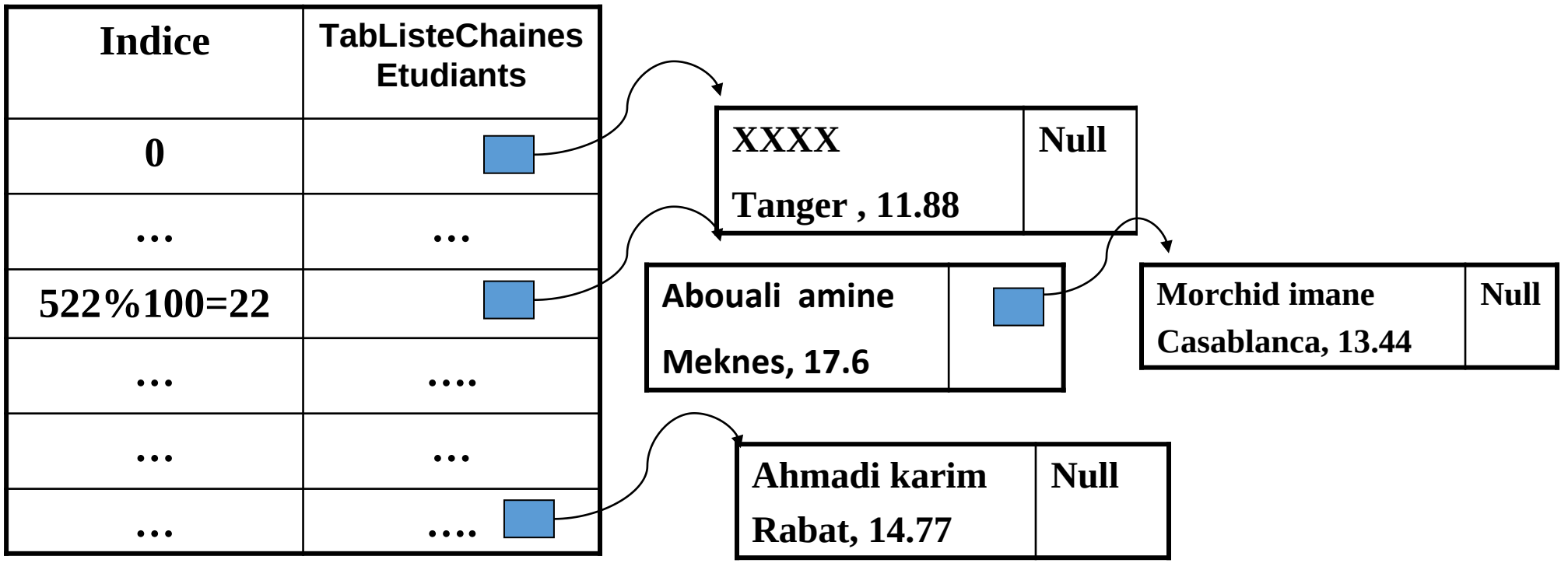
Les collisions mal gérées peuvent dégrader le temps d'accès aux données, réduisant l'efficacité des tables de hachage.

Index	Clé	Valeur
2	"Endymion"	35
5	"Melcha"	41
7	"Ufuoma"	55
8	"Clover"	38
9	"Taqqiq"	42

Exemple

Les clés "Taqqiq" et "Khazhak" ont haché au même index (9), créant une collision à gérer.

Gestion des Collisions - Concept



HESTIM ENGINEERING & BUSINESS SCHOOL

Comparaison avec Autres Structures

Les tables de hachage offrent des performances exceptionnelles pour les opérations d'insertion, de recherche et de suppression en moyenne, avec une complexité temporelle de O(1).

Structure de Données	Insertion	Recherche	Suppression	Ordre des Éléments
Arbres binaires de recherche équilibrés	O(log n)	O(log n)	O(log n)	Oui
Tableaux	O(n)	O(n)	O(n)	Oui (triés)
Listes chaînées	O(1)	O(n)	O(n)	Non
Tables de hachage	O(1) moyen	O(1) moyen	O(1) moyen	Non

Avantages des Tables de Hachage

- Performances optimales en moyenne (O(1)) pour les opérations fondamentales
- Indépendance à l'ordre des éléments

Cas d'Utilisation Optimaux

- Recherche fréquente de données (bases de données, compilateurs)
- Implémentation de dictionnaires et de tableaux associatifs

HESTIM ENGINEERING & BUSINESS SCHOOL

Conclusion

Les tables de hachage sont des structures de données fondamentales et puissantes, offrant une efficacité remarquable pour le stockage et la récupération de paires clé-valeur.

Efficacité

Capacité à fournir des opérations d'insertion, de recherche et de suppression en temps quasi-constant (O(1) en moyenne) grâce à une fonction de hachage efficace.

Polyvalence

Applications variées dans la science informatique, des compilateurs aux bases de données, en passant par les systèmes de cache et les moteurs de recherche.

Implémentation

Base de nombreuses structures de données dans les langages de programmation modernes, offrant une solution robuste pour la gestion efficace des données.

La clé de la performance des tables de hachage réside dans le choix d'une fonction de hachage efficace et d'une stratégie de résolution des collisions adaptée, garantissant une distribution uniforme des données et minimisant les regroupements.